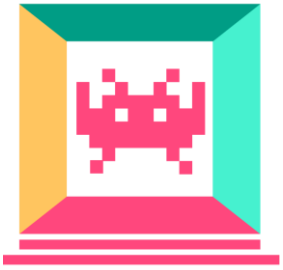


Godot Game Development for Beginners

By Daniel Buckley

Godot Game Developer

Before diving into this eBook, why not check out some resources that will supercharge your coding skills:

Table of Contents

How to Make Your First Game with Godot

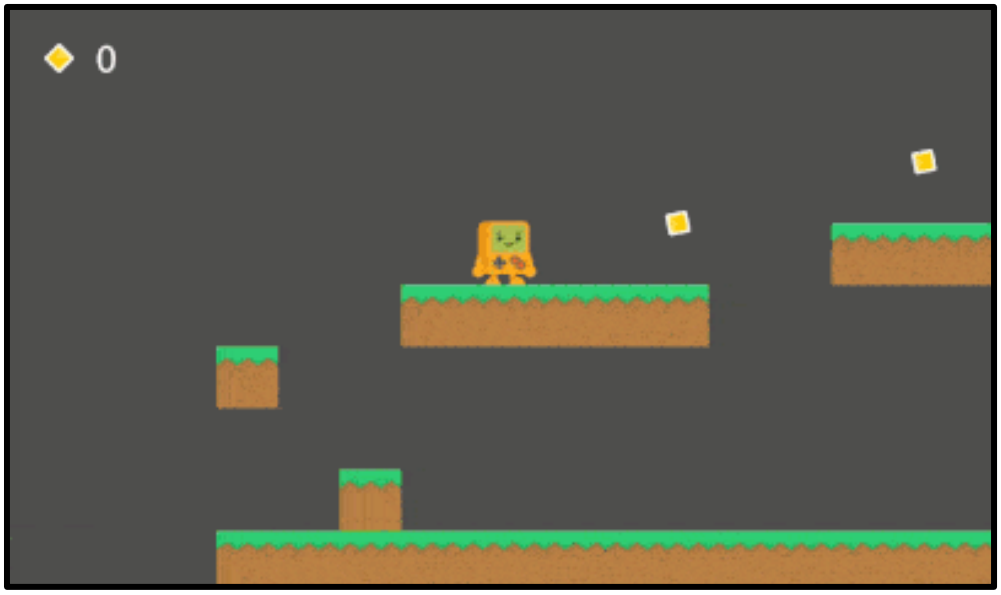
Introduction

Are you eager to get started making your own games?

Game development has never been more popular before - with sales stretching into the billions, and thousands of indie developers sharpening their skills and bringing their creations to life. Even if you're a beginner who has never coded before, with numerous engines available, just about anybody can create and program their dream game project.

In this comprehensive tutorial, we're going to be learning how to create your first game using Godot, a free, open-source game engine that allows you to create 2D and 3D games. Due to its open-source nature, which means users can add and remove things from the engine at their leisure, it has quickly been gaining vast popularity. With a vibrant community ready to assist, it is a perfect choice for creating your first game.

After installing Godot and learning the basics of the editor, we're going to be creating a 2D platformer game, so strap yourself in and get ready to start developing games!



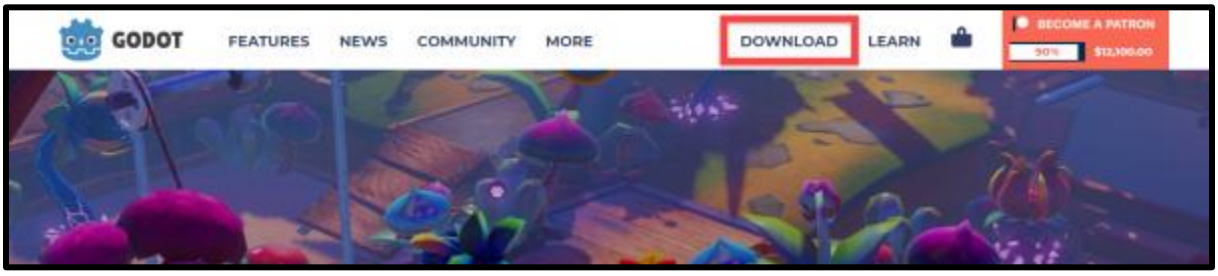
Project Files

For this project, we're going to be needing a few assets for the player, tiles, coins, enemy, etc. You can create your own or use the ones featured in this tutorial. The assets can be downloaded here.

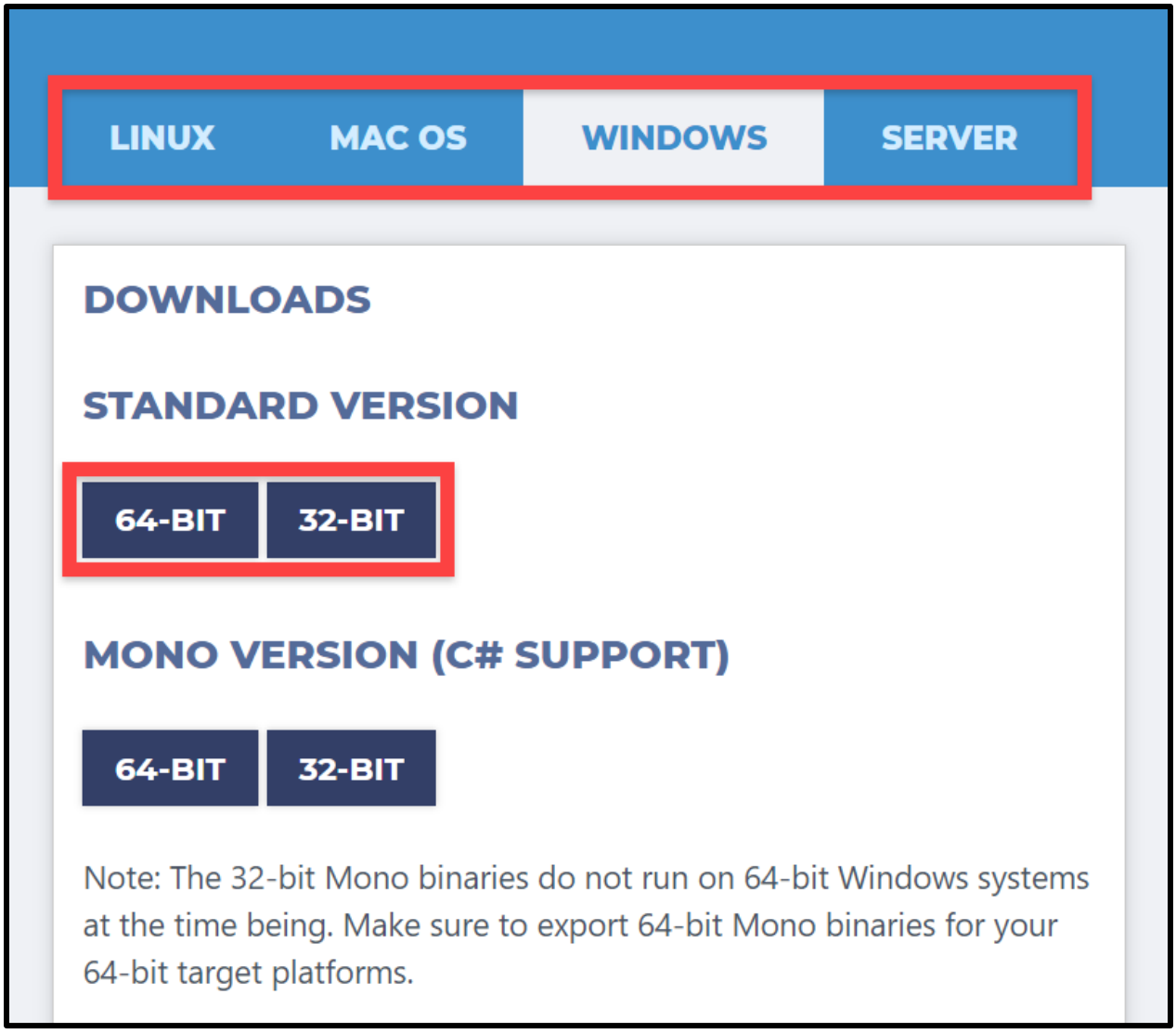
You can also download the completed Godot project via the same link!

Installing Godot

To download and install Godot, let's go over to https://godotengine.org/. Here, you can view Godot's features, community pages and more. We want to click on the **Download** button.



Select your platform, then download the 64 or 32 bit standard version (depending on your operating system).

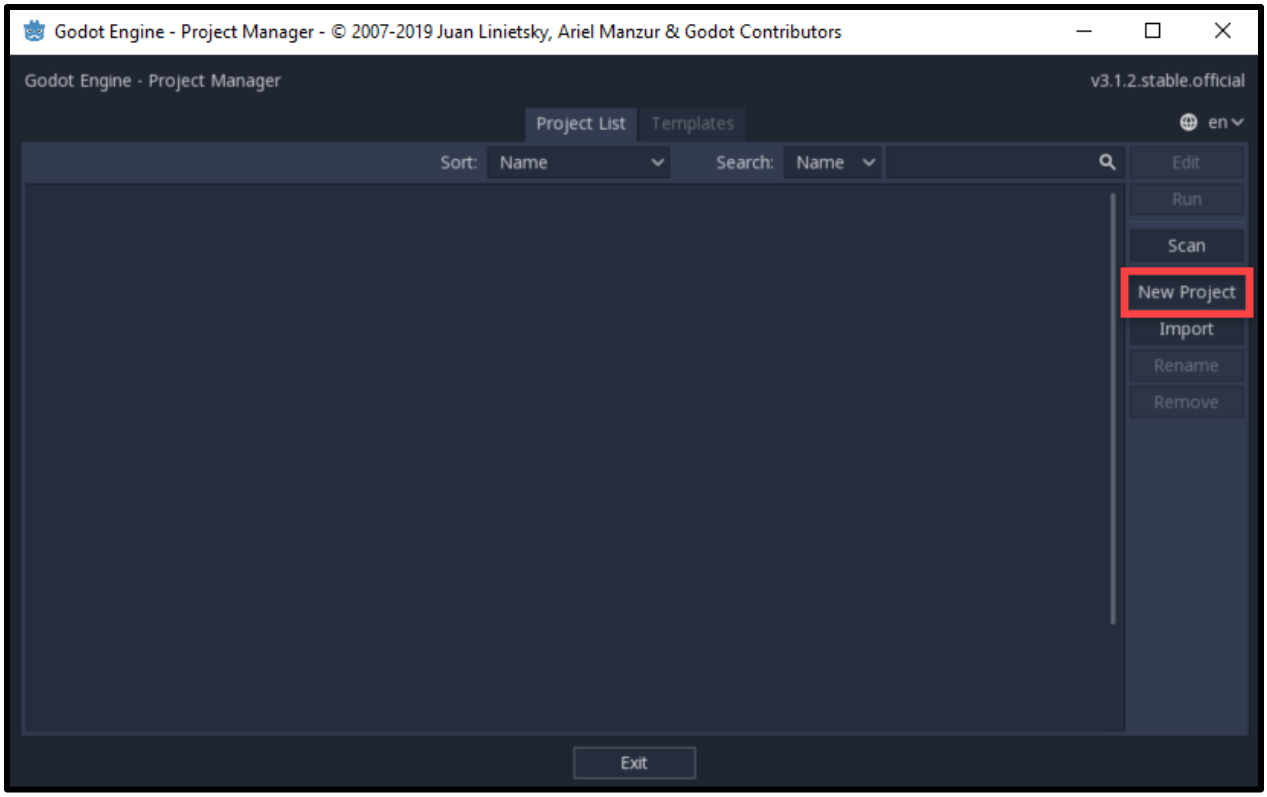


This will download a .ZIP file. Inside of that is an application which you can extract to anywhere on your computer, and that's it - Godot is installed.

Creating a New Project

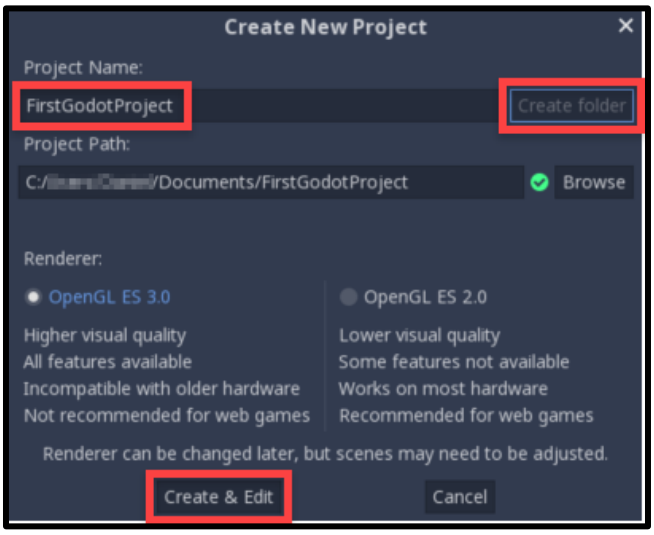
Open up the Godot application we download to see the **Project Manager**. Here, we can create projects, view others and download templates.

Click on the **New Project** button to create a new project.



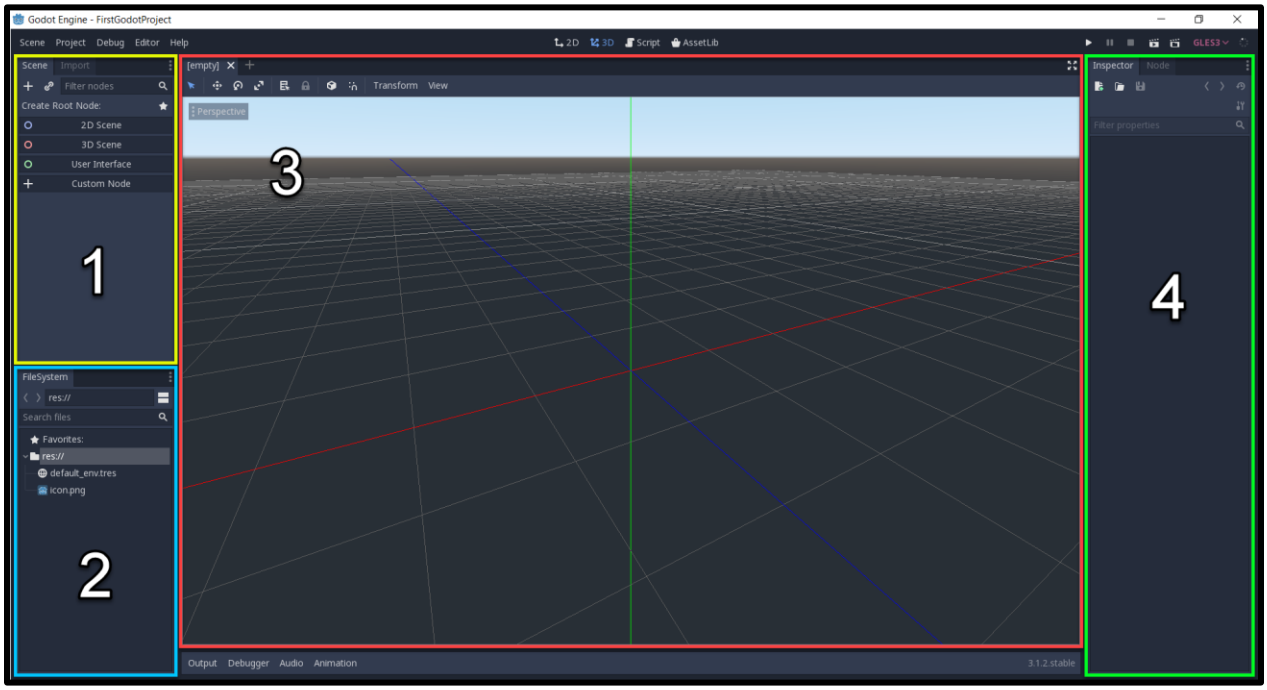
This will open up a new window.

1. Enter in a name for your project
2. Click the **Create Folder** button to create a new folder for the project in the Documents folder
3. Click the **Create & Edit** button to launch the engine and begin creating the game



Exploring the Editor

When the editor pops up it may look pretty daunting with all the buttons and options, but let's break it down.



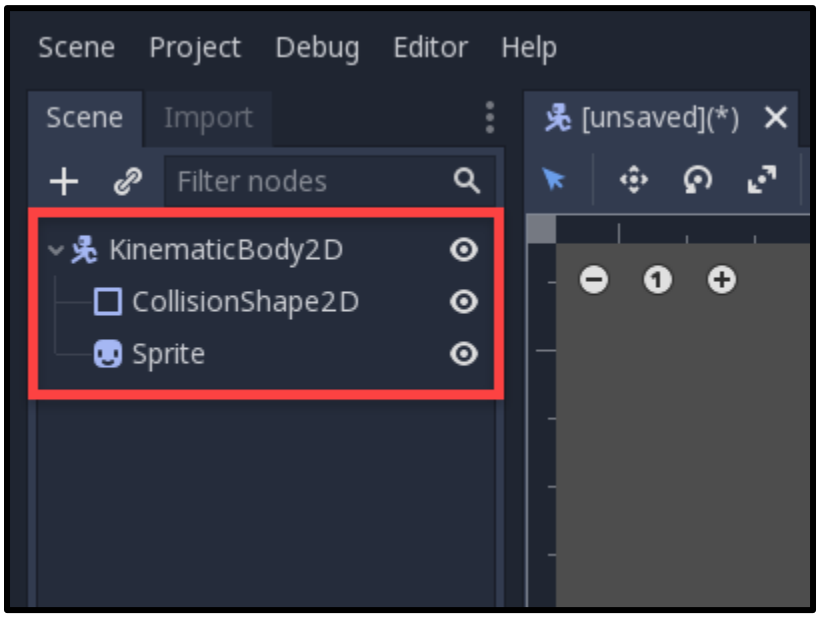
Godot has 4 main panels which we'll be using to create our game and each serves a specific purpose.

1. This here is the **Scene** panel. It will display the node layout and hierarchy of the scene we're currently in. We'll go over what a scene and a node are very shortly.
2. The **FileSystem** panel shows us all of the assets and files we have. Sprites, models, scripts, scenes, folders, audio, etc.
3. This is where we can see and create our game. Moving things around, selecting, scripting, etc. Above this panel are four buttons and they toggle what the panel becomes. We can switch between 2D and 3D modes, the script editor and external asset library.
4. This is the **Inspector** and this shows us the details of a node when we select one. The position, rotation and any other attributes which we can modify.

How Godot Works

Above we were talking about scenes and nodes. What are they? Well, a game in Godot is made up of a hierarchy of nodes. A node can be anything: a player, camera, 3D model, light, UI, etc. Nodes make up all of the entities in your game and also have the ability to be a child of another node.

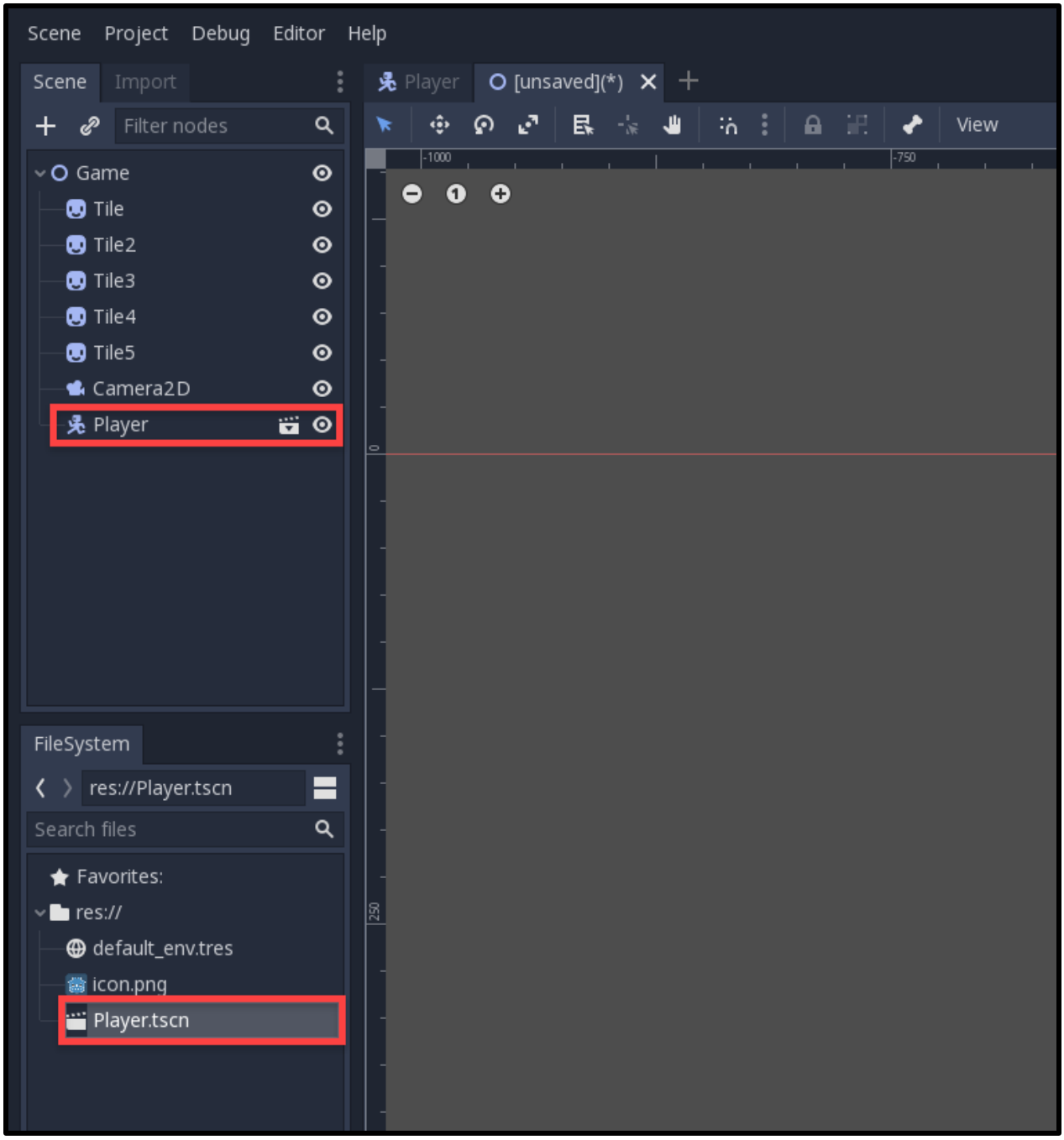
Here's an example of a player in a game. First, we have the KinematicBody node which can control movement and some physics interactions. Then, as a child of this, we have a collider and sprite node. When the kinematic body node moves, rotates, etc, the others will follow. Even if we delete the parent node, the children will follow.



Games in Godot are made up of many parent-children nodes to create the various different elements and systems of a game.

Since a game is made up of nodes, it will eventually get to a point where there are hundreds or even thousands of them in the scene tree panel. This will make it hard to find certain ones and overall make working on the game confusing. To solve this issue, we can divide up our nodes into **scenes**.

Scenes are self-contained node packages which we can then drop into other scenes as nodes. Let's take our player example and turn that node hierarchy into a scene. It is a saved file in the FileSystem which we can then drag into another scene.



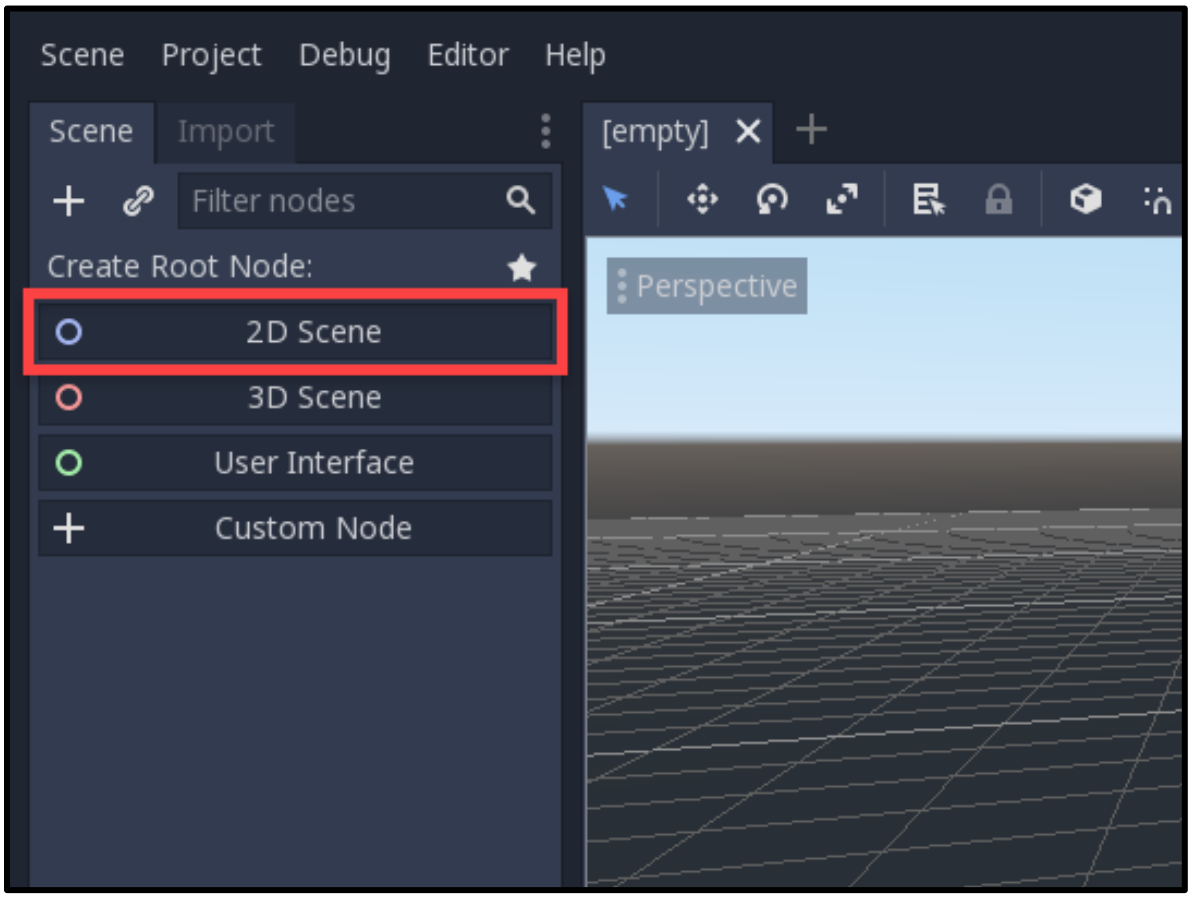
A benefit of dividing your game up into nodes is also the fact that we can remove the need to repeat common node structures. Instead of having 100 tile nodes which all have a sprite, collider, etc all in the same scene, we can just create one of those as a scene and drag in multiple instances.

We'll be exploring nodes and scenes throughout this project.

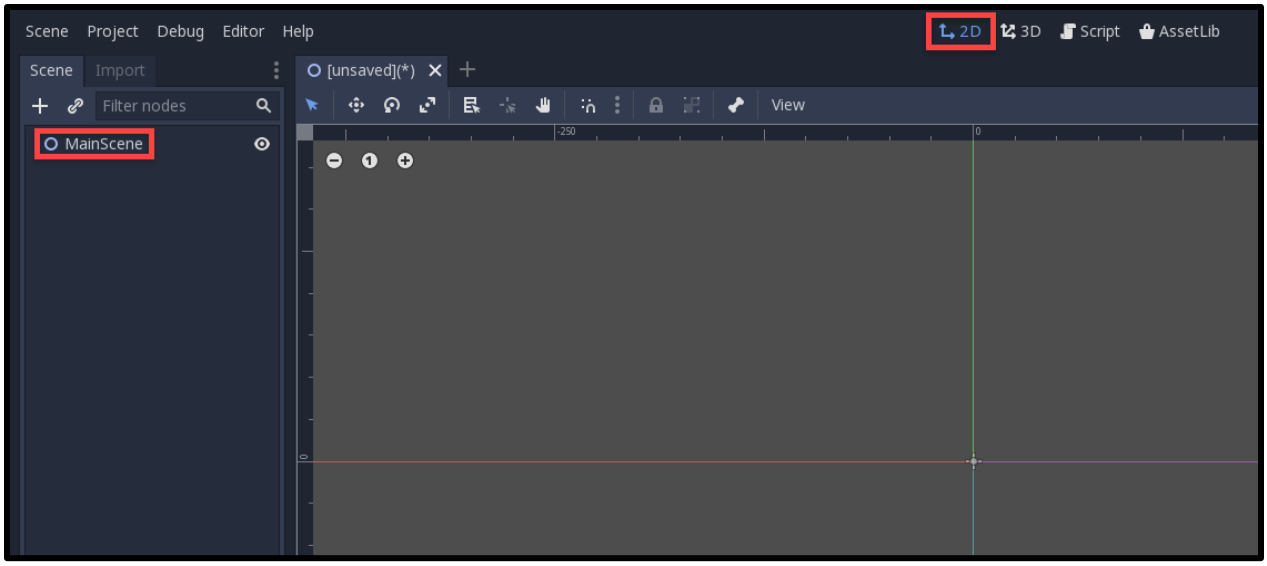
Creating our First Scene

Alright, let's get started on our 2D platformer project. First, we'll need to create a main scene which will be the basis for our game, containing other scenes such as the player, tiles, coins, and enemies.

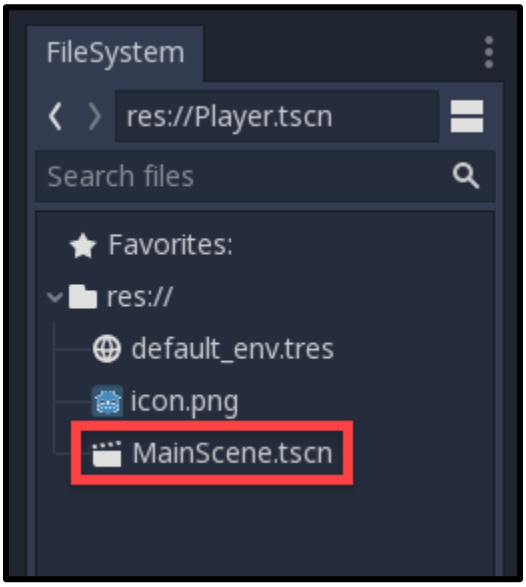
In the scene panel, click on the **2D Scene** button to create a new 2D scene.



With our new node, let's double click it and rename it to **MainScene**.



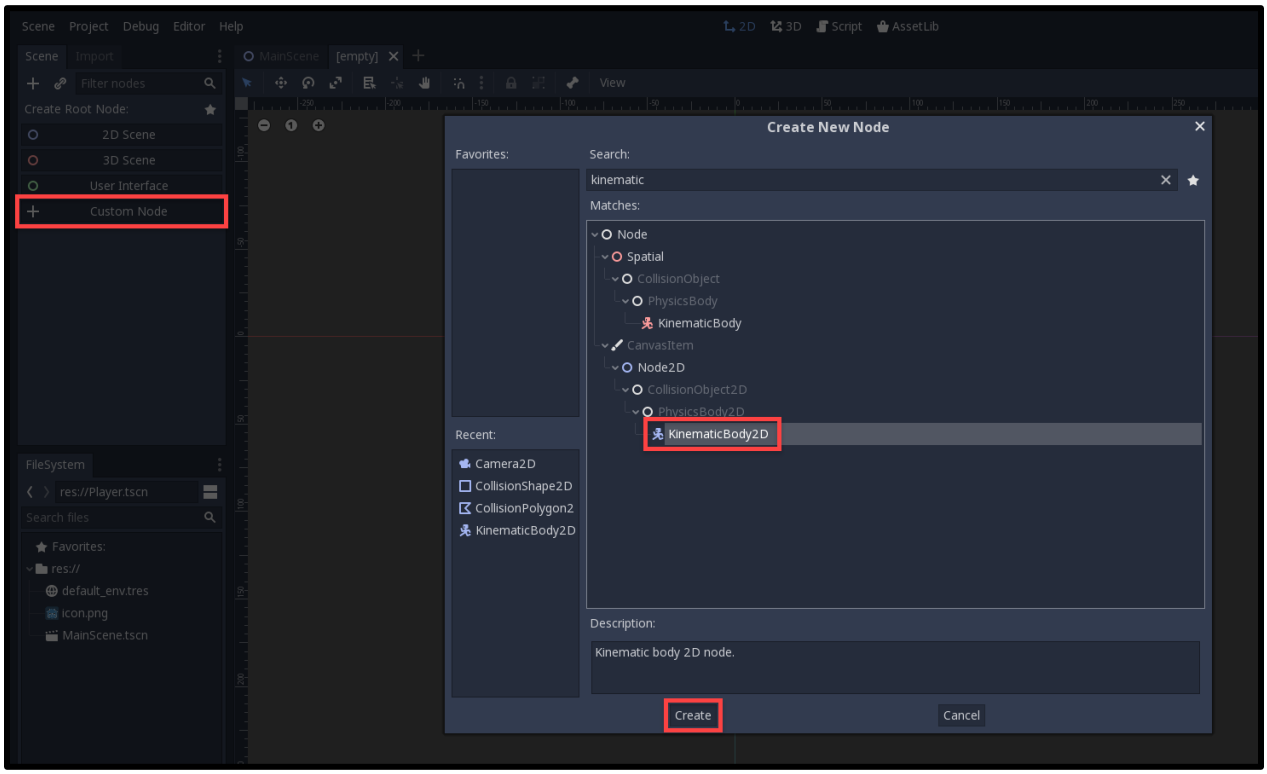
It's also good to save your scenes when you create them. Save with either **Ctrl + S** or Scene > Save Scene. Hit enter and you should see it down in the file system.



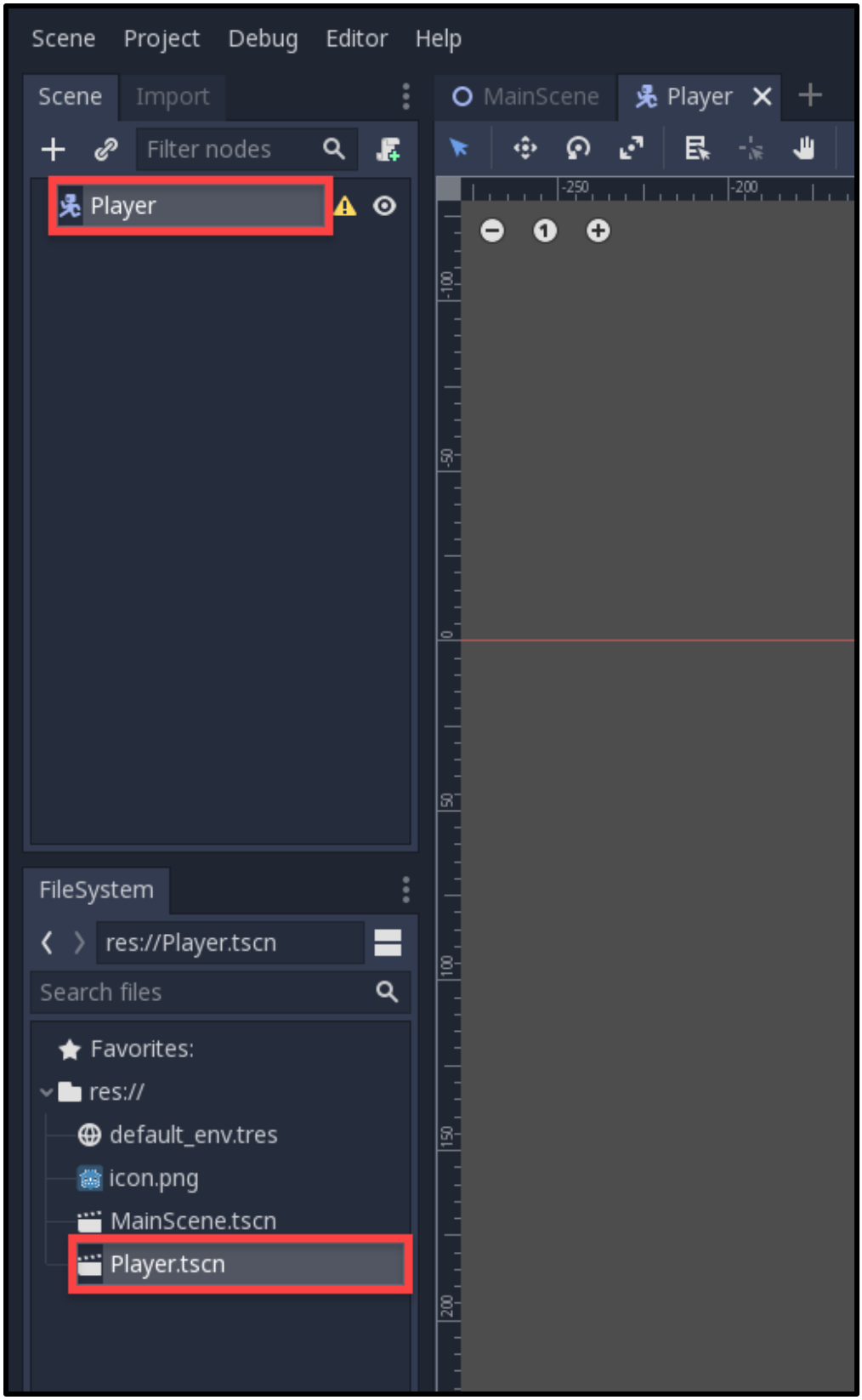
Creating the Player

Alright, we got our main scene. Next, let's go and create our player scene which will hold all the nodes we need and the corresponding script.

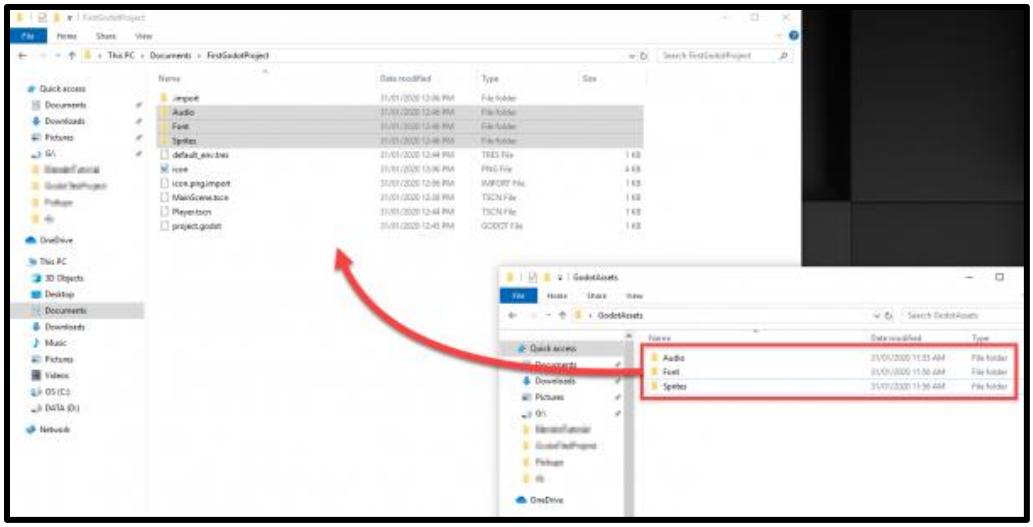
To create a new scene, we can go Scene > New Scene. In the scene panel, select **Custom Node**. This will open a window and we want to create the **KinematicBody2D** node.



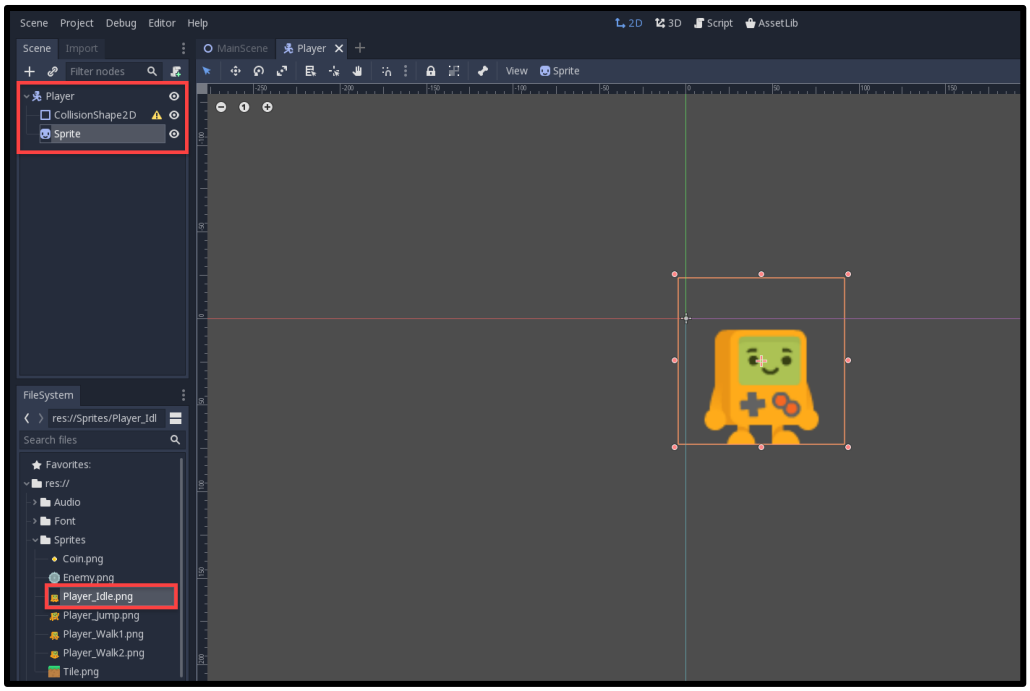
Rename the node to **Player**, then save it.



With our platformer game, we'll be needing a few sprites and other assets. Included with this tutorial, is a .ZIP file containing all the assets we'll need. Download that and drag the three folders into our Godot project folder.

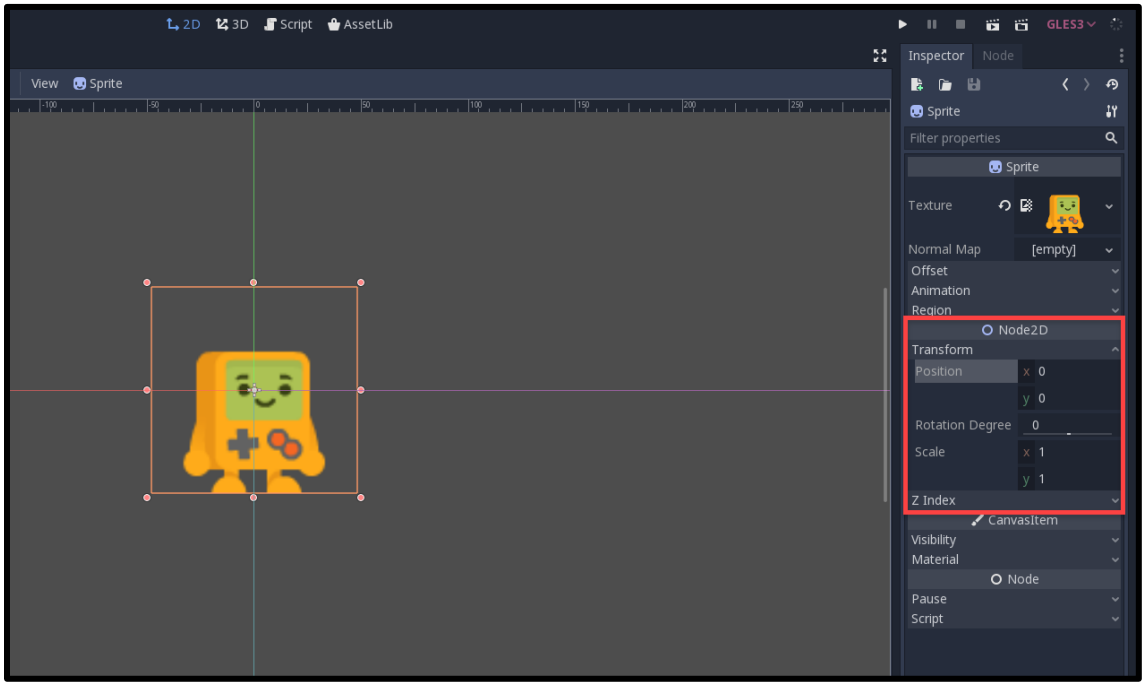


Back in Godot, let's right-click on the Player node and select **Add Child Node**. We want to add in a **CollisionShape2D** node as a child. Then in the file system, find the **Player_Idle** image and drag that into the scene window to add that as a child node. Rename that node to **Sprite**.

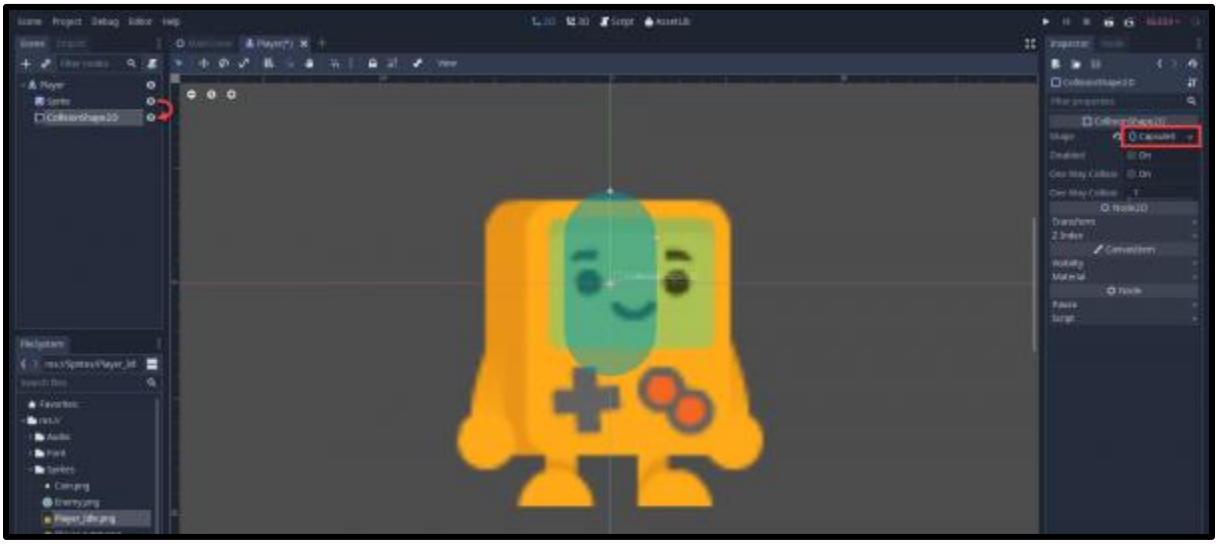


We want our sprite to be centered, so select it and over in the Inspector:

- Open the **Transform** drop-down
- Set the **Position** to 0, 0

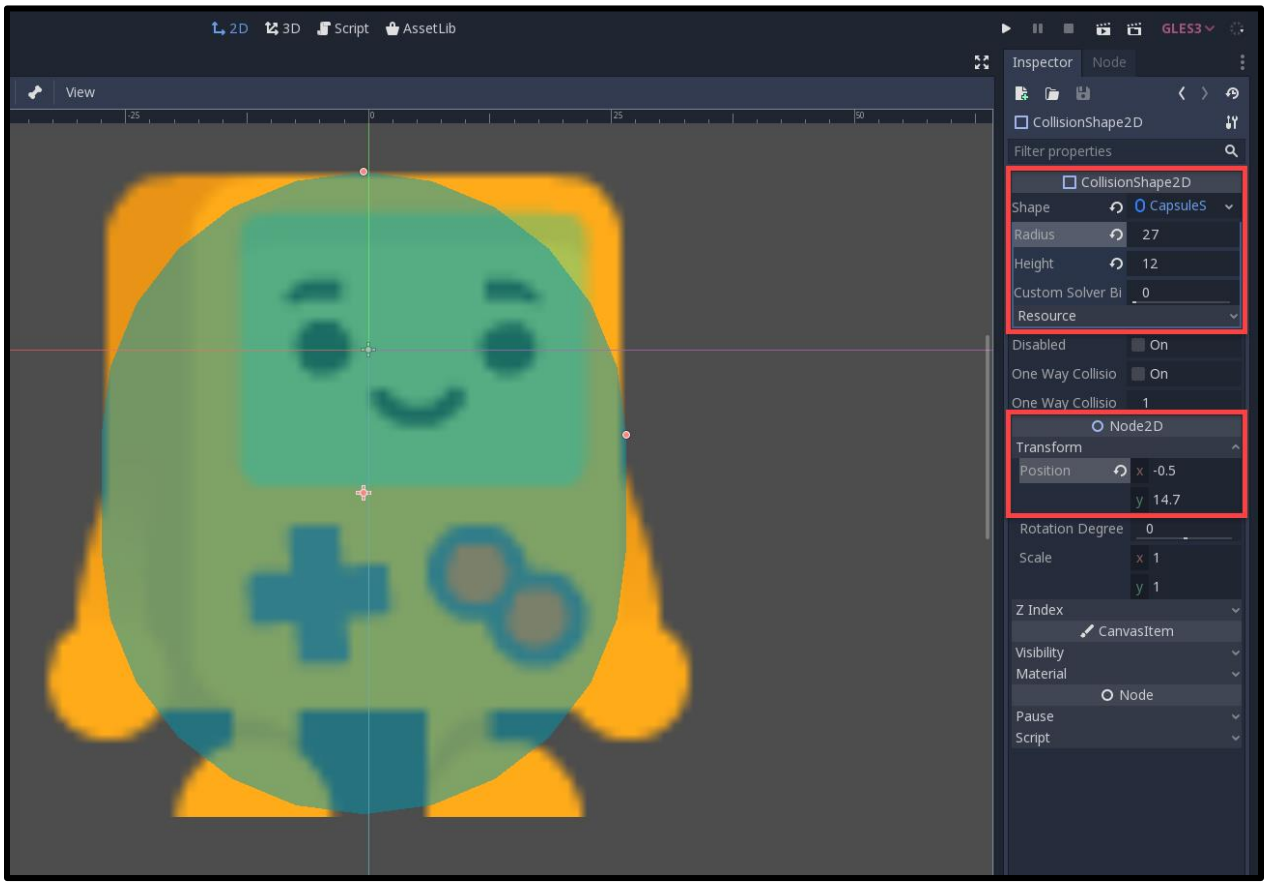


You might see that the collision node has an error symbol next to it. This means we need to give it a shape. Select the collision node and in the inspector set the **Shape** to Capsule. To make it visible, re-order the node hierarchy by dragging the collider node underneath the sprite node.



To edit the collider properties, select the shape in the inspector and it will open up more options.

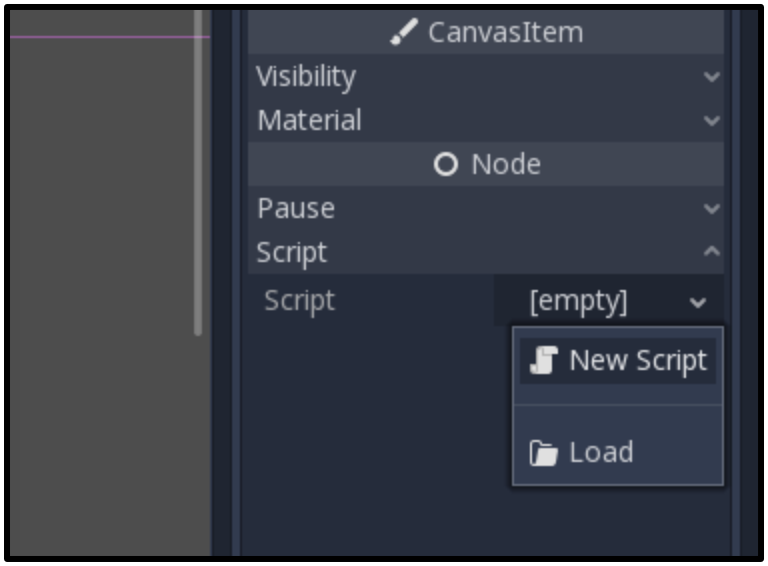
- Set the **Radius** to 27
- Set the **Height** to 12
- Set the **Position** to -0.5, 14.7



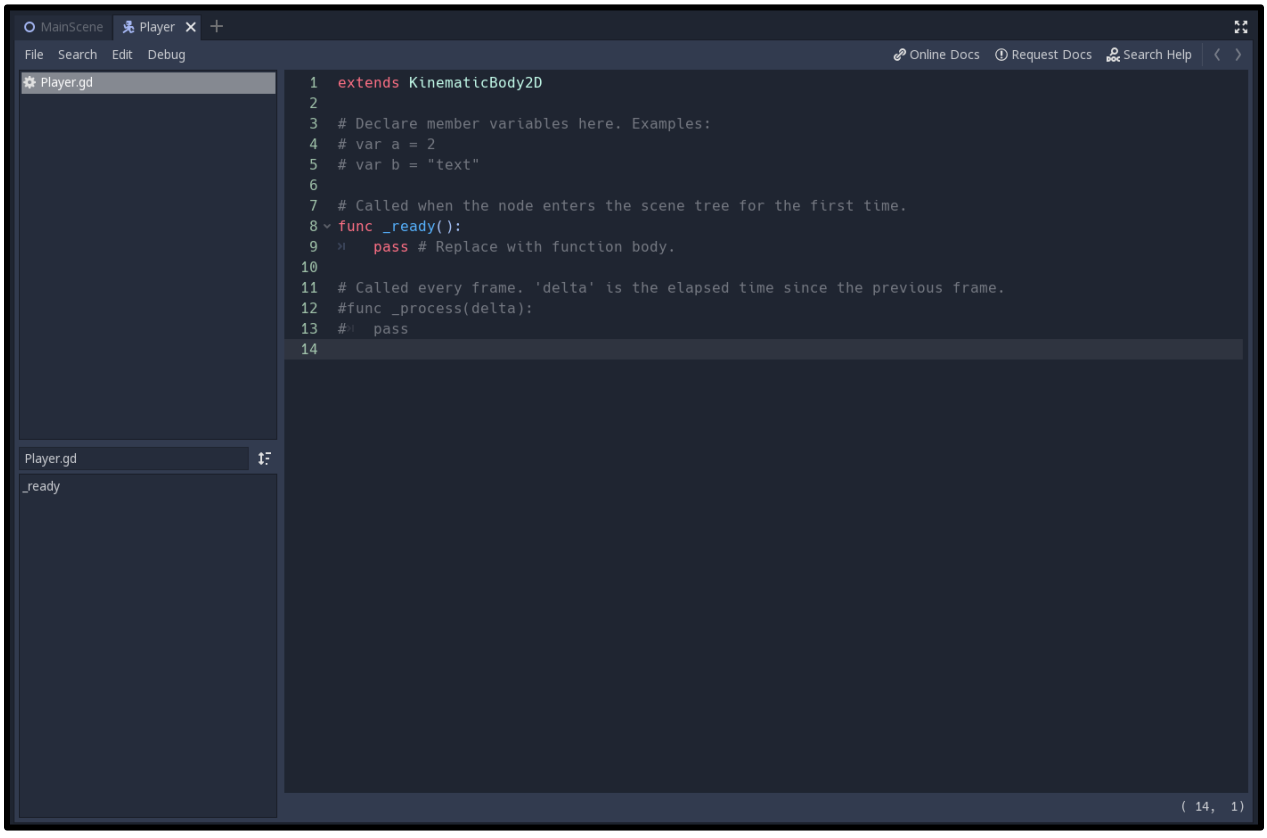
Scripting the Player

Now that we have the node structure setup, we can begin to script our player. This will involve the movement, jumping and collecting coins.

To create a script, select the parent **Player** node and in the Inspector, create a new script. This will open up another window, just hit enter.



What you'll see then is that the main window will switch from 2D to Script mode.



Godot uses its own scripting language called **GDScript**. This is similar in syntax to Python. We won't be going over every aspect of the language or the basic concepts of programming - so a basic understanding is required going forward.

Right now in the script we have two things.

```
1 extends KinematicBody2D
```

Extends is similar to using or import. We're extending from the kinematic body 2D object we're attached to, so we'll have direct access to those attributes.

```
1 func _ready ():
2     pass
```

This is a function that is basically a block of re-usable code we can call. The **_ready** function is built into Godot and gets called once when the node is initialized.

The **pass** is simply a filler line to define the function. If it's empty then an error will occur. The pass does nothing.

We're going to start by adding in some variables.

```
1  # stats
2  var score : int = 0
3
4  # physics
5  var speed : int = 200
6  var jumpForce : int = 600
7  var gravity : int = 800
8
9  var vel : Vector2 = Vector2()
10 var grounded : bool = false
```

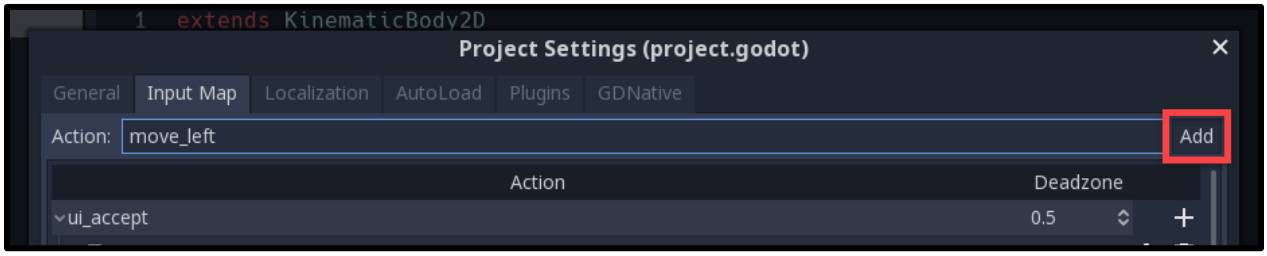
A **Vector2** defines an x and y value which can be used for position, scale, rotation, velocity. We're going to be using it to store our player's current velocity.

We need one more variable. This is a reference to the sprite component which is a child node. **onready** means that we're going to find the Sprite node when the node is initialized.

```
1 # components
2 onready var sprite = $Sprite
```

Now that we have our variables defined, let's start to get our player moving. To do this, we first want to define some key inputs. Open the **Project Settings** window (Project > Project Settings) and in there, navigate to the **Input Map** tab.

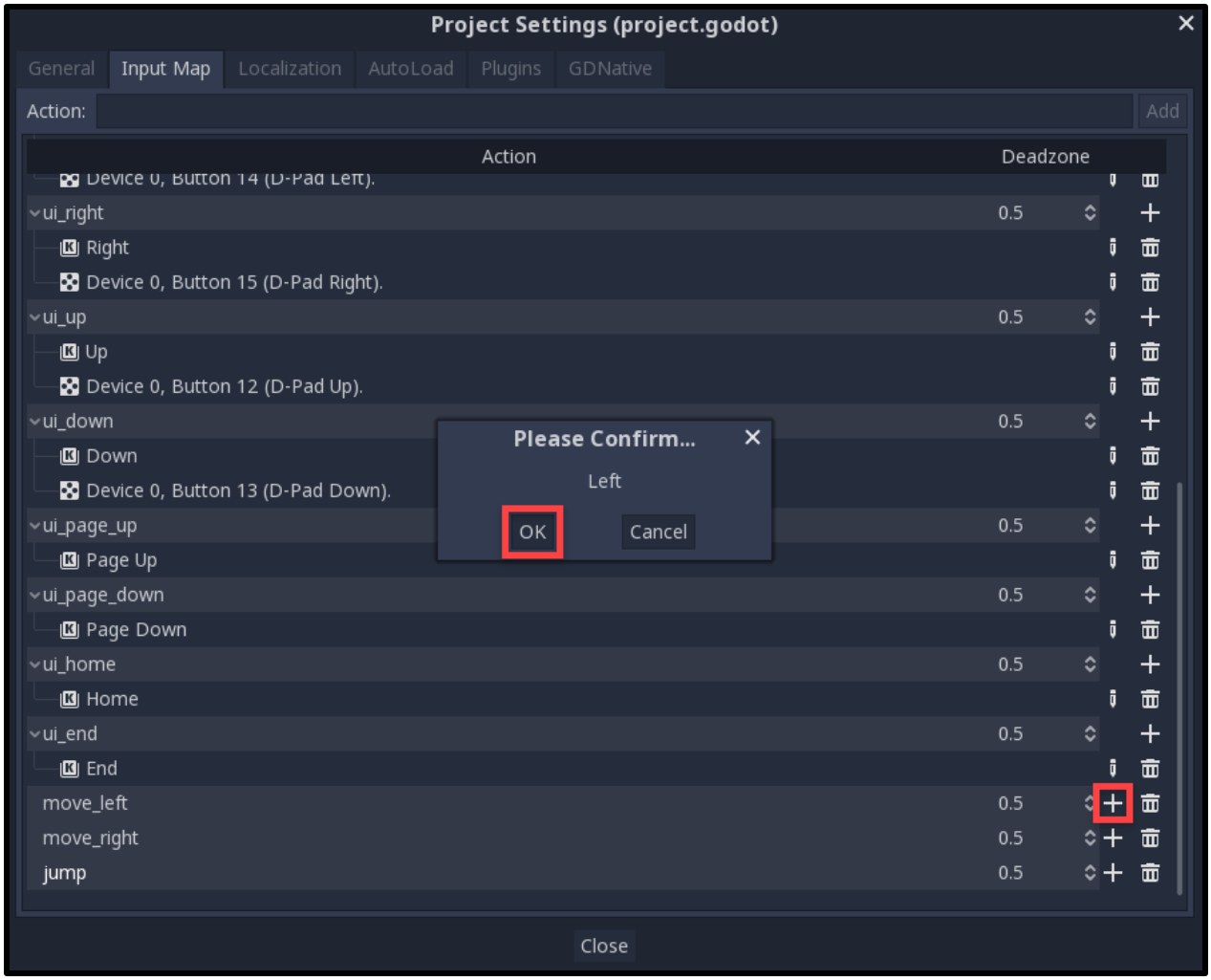
In the **Action** field, enter in a name for the input and click add.



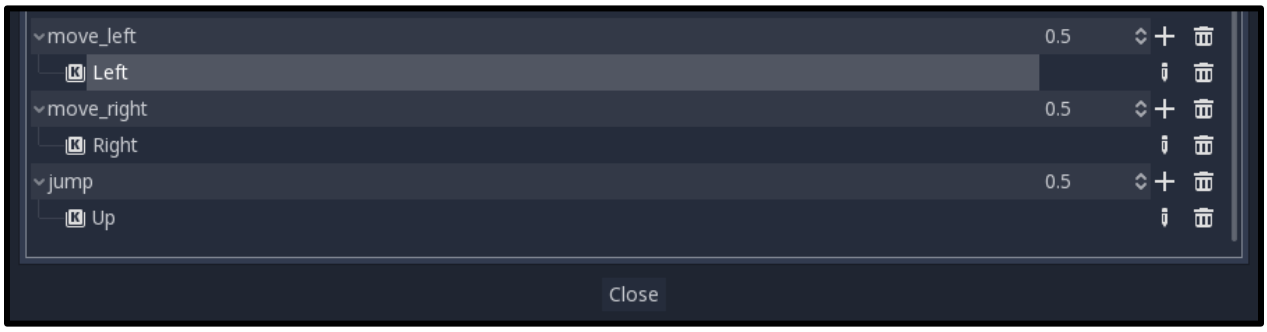
We want to create 3 new actions.

- move_left
- move_right
- jump

Next to each action is a '+' icon. Click that, then select **Key**. This will allow us to enter in a key for that action.



Enter in a new key for each of the three actions.



Once that's done, we can close the window and continue scripting.

We're now going to create a function which is built into Godot. The **_physics_process** function gets called 60 times a second and is what we use for physics calculations. We can remove the **pass** once we start filling the function in.

The **delta** parameter is the time between each frame. We can multiply our movement by this in order to move based on pixels per second, rather than pixels per frame.

```
1 func _physics_process (delta):
2     pass
```

First, we want to reset the horizontal velocity. Then check for the left and right movement key inputs. These will change the horizontal velocity.

```
1 # reset horizontal velocity
2 vel.x = 0
3
4 # movement inputs
5 if Input.is_action_pressed("move_left"):
6     vel.x -= speed
7 if Input.is_action_pressed("move_right"):
8     vel.x += speed
```

Next, we'll move the player using the **move_and_slide** function which will move along a certain velocity, detecting colliders and other things. The second parameter is the ground normal (which way is the ground pointing?).

```
1 # applying the velocity
2 vel = move_and_slide(vel, Vector2.UP)
```

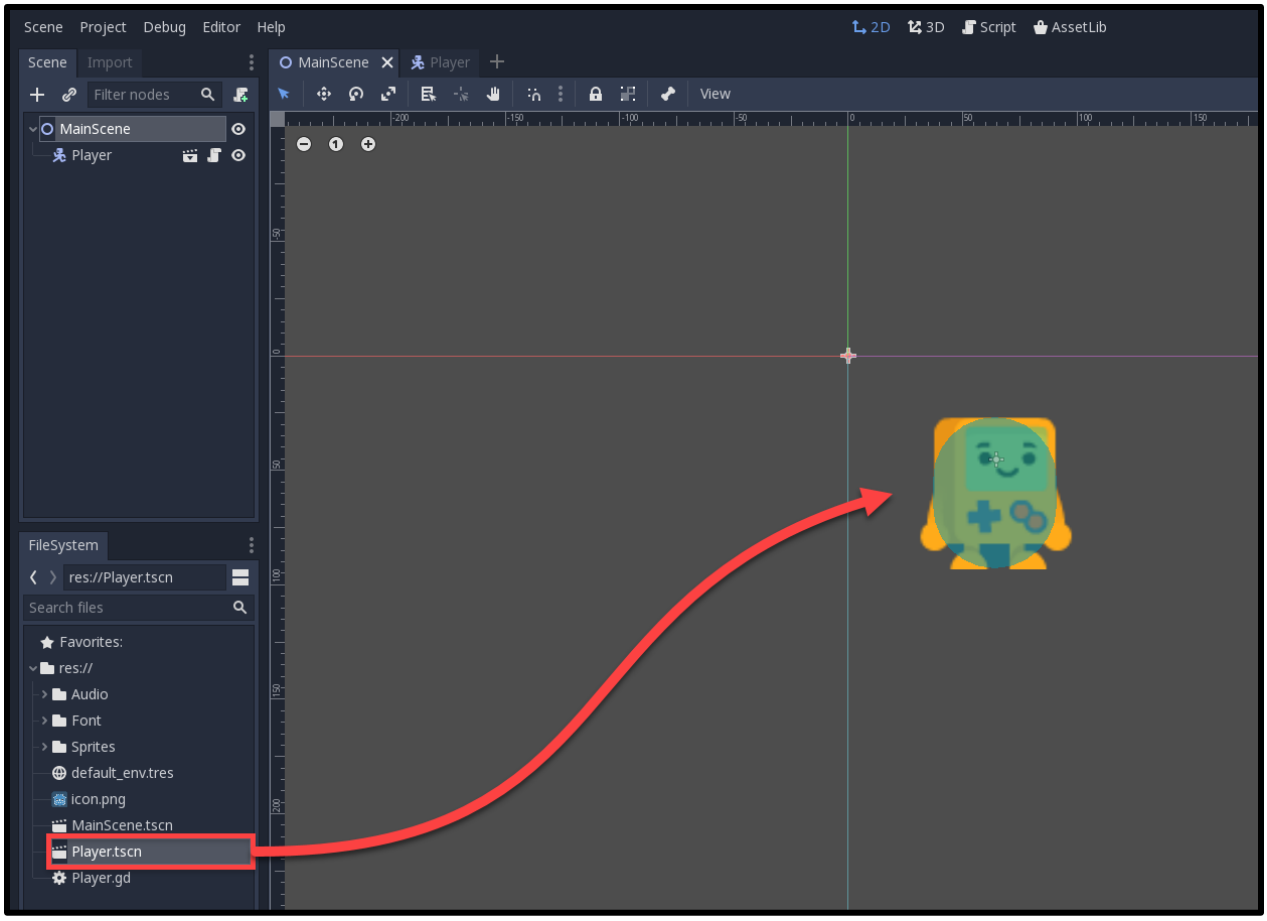
After this, we can apply gravity and check for if we're pressing the jump button and on the floor. If so, jump.

```
# gravity
vel.y += gravity * delta

# jump input
if Input.is_action_pressed("jump") and is_on_floor():
    vel.y -= jumpForce
```

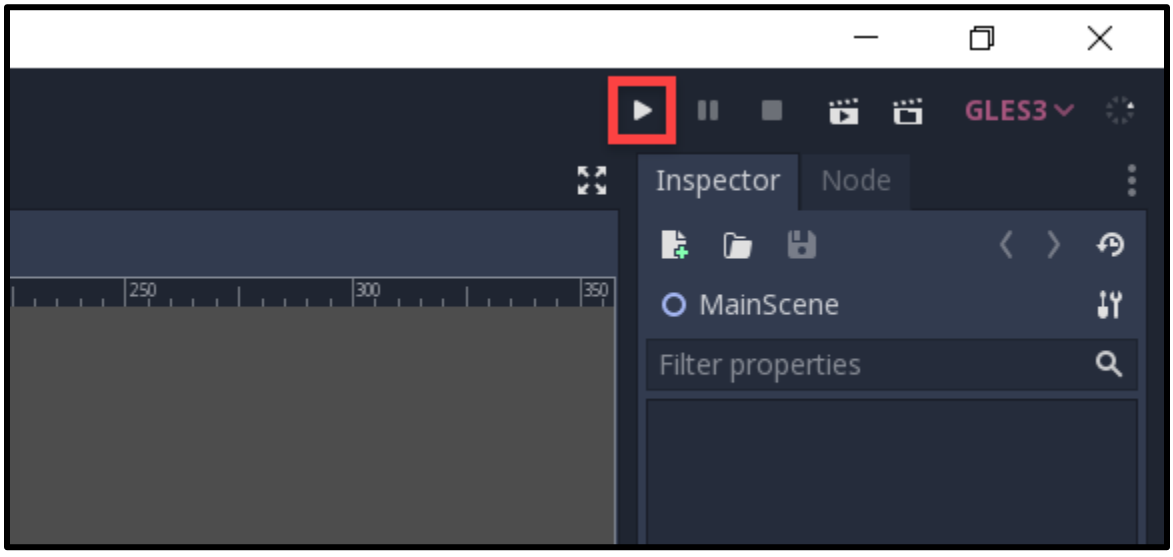
Finally, we can flip the sprite depending on which way we're moving.

```
# sprite direction
if vel.x < 0:
    sprite.flip_h = true
elif vel.x > 0:
    sprite.flip_h = false
```

We're done scripting for now, so let's switch back to **2D** mode and go to the **MainScene**. Here, we want to drag in the **Player.tscn** scene from the file system.



What we can do now is test the game out. In the top right corner of the screen, click the **Play** button.



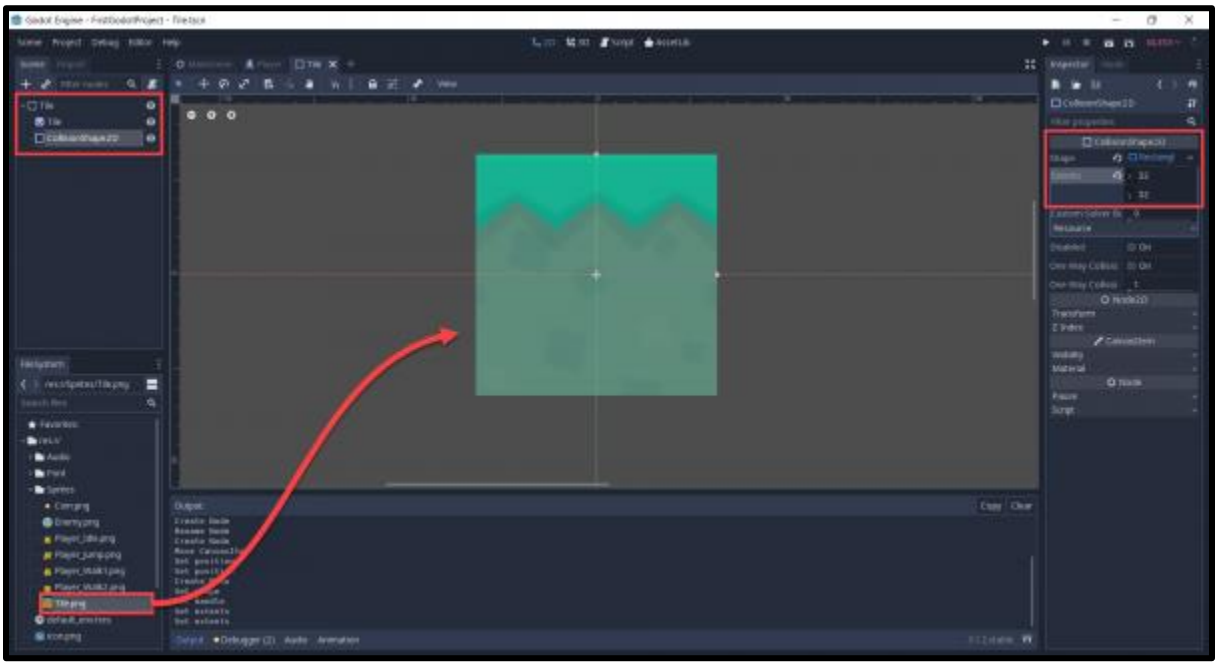
A window will pop up saying that there's no main scene selected. Click the **Select** button and select the **MainScene.tscn** scene as the default one.

The game should then open up but we'll quickly fall off-screen due to gravity. What we need to do is create a tile scene which we can then duplicate in the main scene.

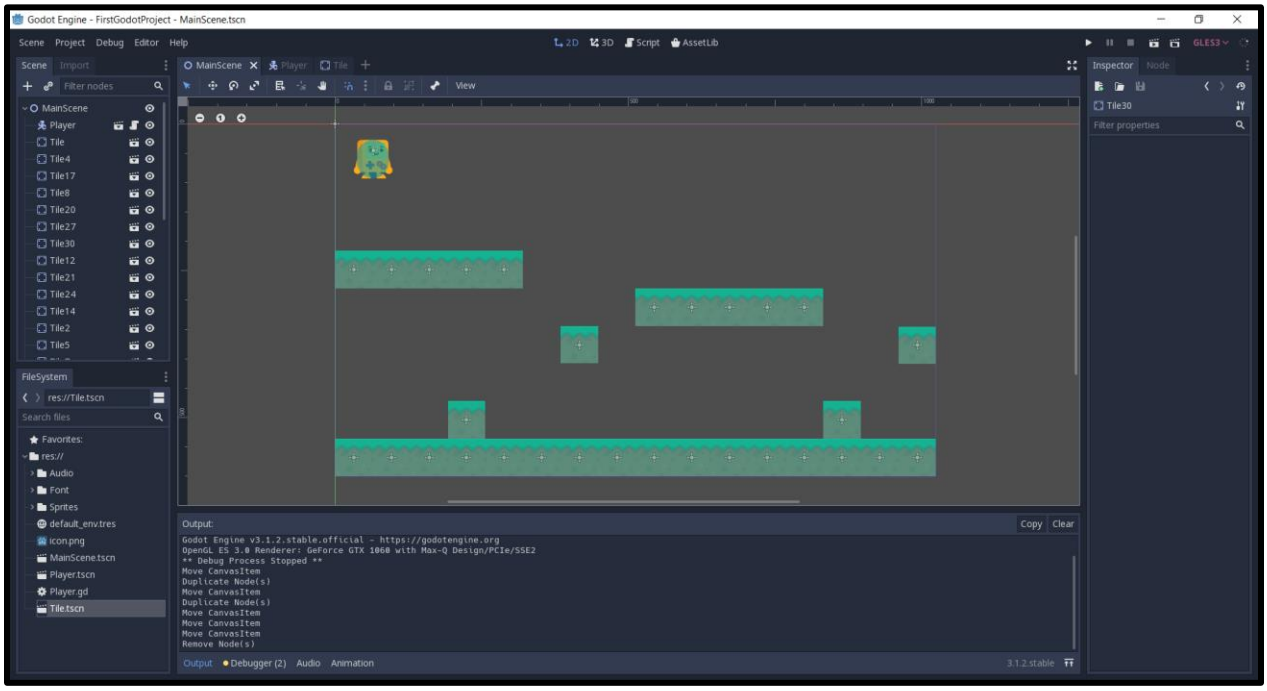
Creating a Tile

Let's now create a new scene (Scene > New Scene), select **Custom Node** and search for the **StaticBody2D** node. This is a physics node which is static so it doesn't move or do anything but detect collisions.

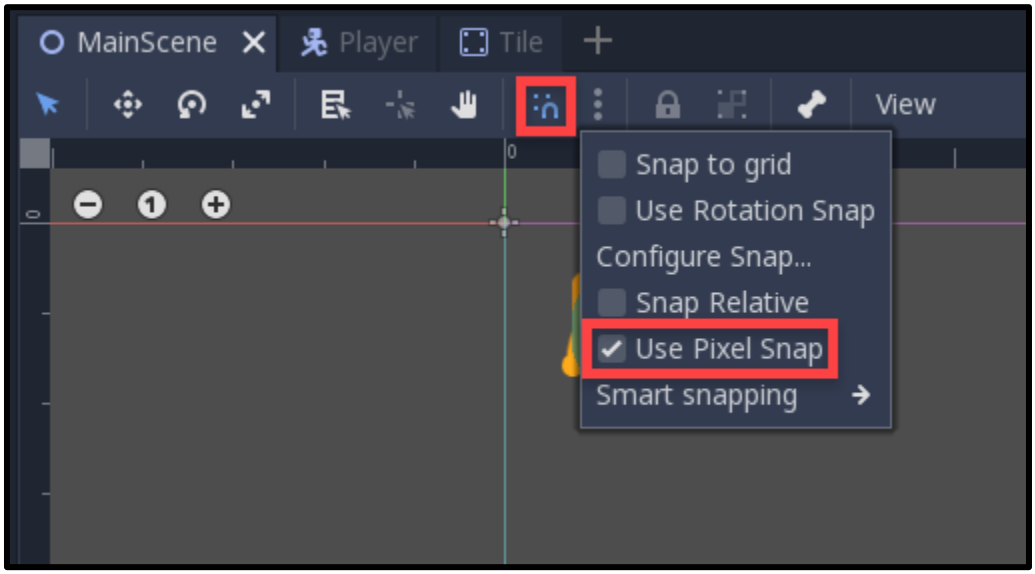
1. Rename the node to **Tile**
2. Attach a new CollisionShape2D child node
3. Set the **Shape** to Rectangle
4. Set the **Extents** to 32, 32
5. Drag in the **Tile** image from the file system to create a new sprite node
6. Rename it to **Tile**
7. Set the **Position** to 0, 0
8. Save the scene as **Tile.tscn**



Back in the MainScene, we can drag the tile scene in. Then duplicate it with **Ctrl + D** and move it around.



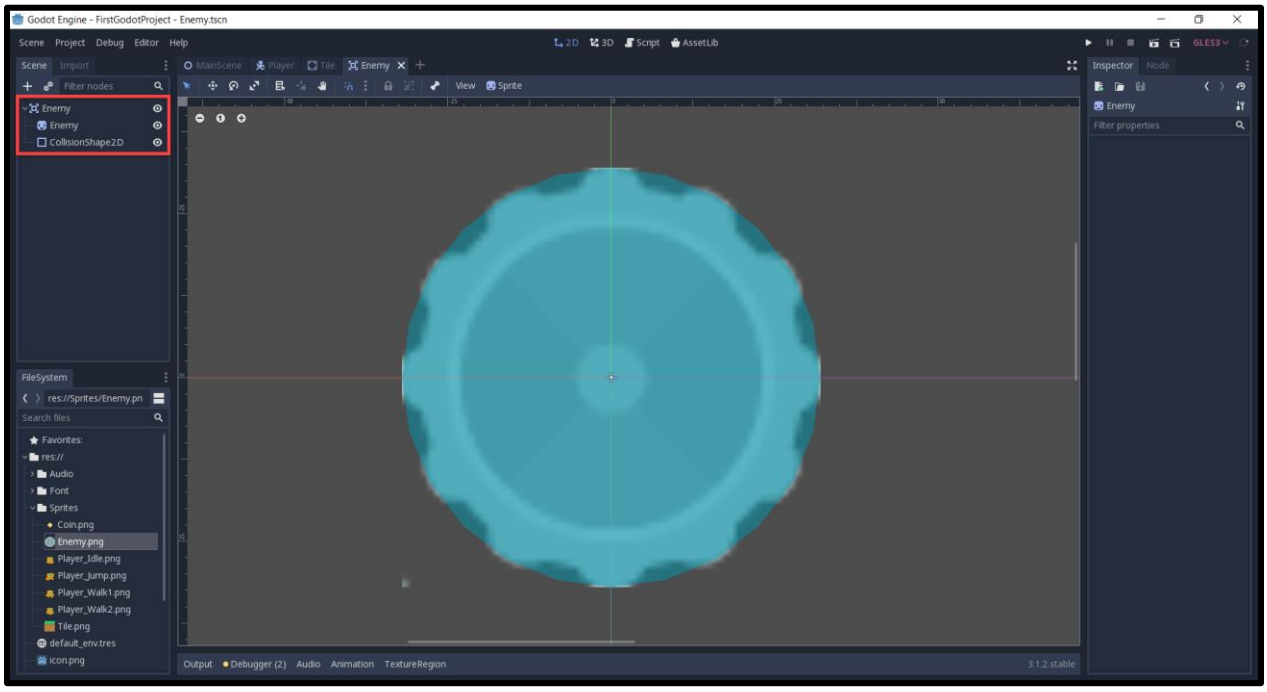
To make it easier to position the tiles, we can turn on snapping. At the top left of the scene panel, click on the snap button to toggle it. Then click on the three dots to bring up the snap settings. Make sure only **Use Pixel Snap** is selected. Now when we move a tile around, it will snap based on the pixels.



Now we can press play and test it out on our new level.

Creating an Enemy

Create a new scene with a root node of **Area2D**. As a child attach a collision shape 2D node in the shape of a circle and drag the **Enemy** sprite in too (make sure to center it). Then save the scene as **Enemy.tscn**.



Select the Area2D root node and create a new script called **Enemy**.

First, we want to create the variables.

```
1  export var speed : int = 100
2  export var moveDist : int = 100
3
4  onready var startX : float = position.x
5  onready var targetX : float = position.x + moveDist
```

export means that this variable is exposed in the inspector, allowing us to change the property on the instance of the scene.

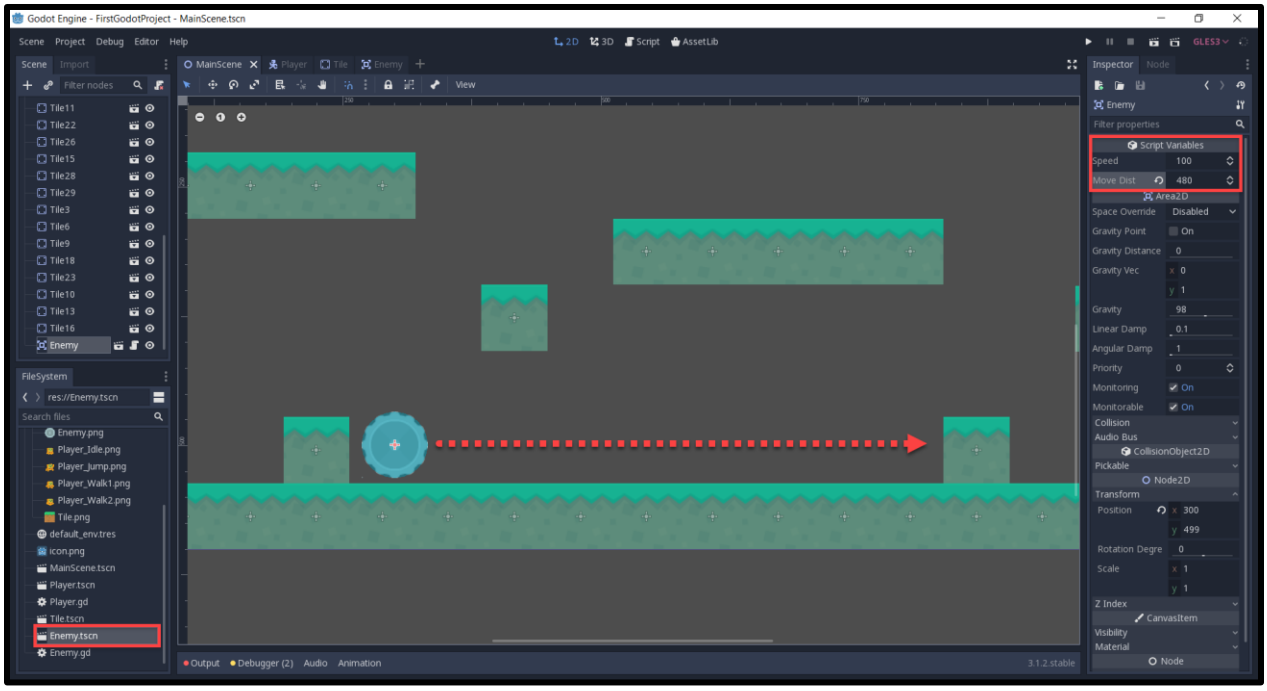
Then in the **_physics_process** function, we're going to be moving between the startX and targetX positions. The **move_to** function is our own custom function we'll make after.

```
1  func _physics_process (delta):
2
3      # move to the "targetX" position
4      position.x = move_to(position.x, targetX, speed * delta)
5
6      # if we're at our target, move in the other direction
7      if position.x == targetX:
8          if targetX == startX:
9              targetX = position.x + moveDist
10         else:
11             targetX = startX
```

The **move_to** function translates a given value to a target based on a given step.

```gdscript
# moves "current" towards "to" in an increment of "step"
func move_to (current, to, step):

    var new = current

    # are we moving positive?
    if new < to:
        new += step

        if new > to:
            new = to
    # are we moving negative?
    else:
        new -= step

        if new < to:
            new = to

    return new
```

Back in the **MainScene**, drag the **Enemy.tscn** into the scene. Figure out the distance between the enemy and where you want to move to and set that in the **Move Dist** property.

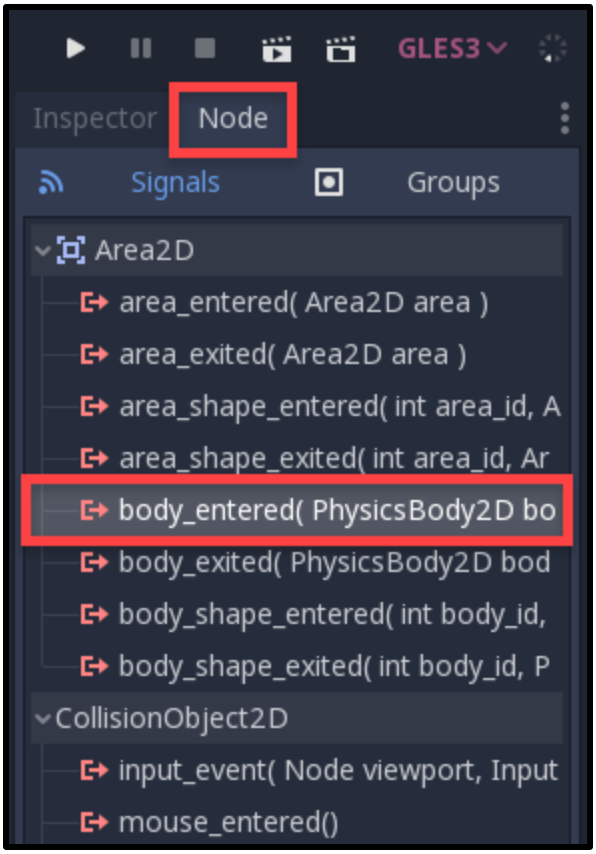


We can now press play and see the enemy in action. You may notice that it's quite slow, so let's set the **Speed** to 500.

Colliding With the Enemy

Right now, the player can walk through the enemy with no issue. What we want, is for the game to reload if we hit the enemy. To do this, we need to work with signals. These are events that can be called when a certain thing happens to a node.

In the **Enemy** scene, select the root Area2D node and in the Inspector, switch to the **Node** tab. Here, we can see all the signals that the node can emit. We want to double click on the **body_entered** signal. Press enter and it should create a new function in the **Enemy** script.



In this function, we're going to check if the body we entered was called **Player**. If so, call the **die** function on that node.

```
1  # called when we collide with a physics body
2  func _on_Enemy_body_entered (body):
3
4      if body.name == "Player":
5          body.die()
```

Let's now go to the **Player** script and create the **die** function.

```
# called when we hit an enemy
func die ():

    get_tree().reload_current_scene()
```

Now when the player hits the enemy, the scene will reload.

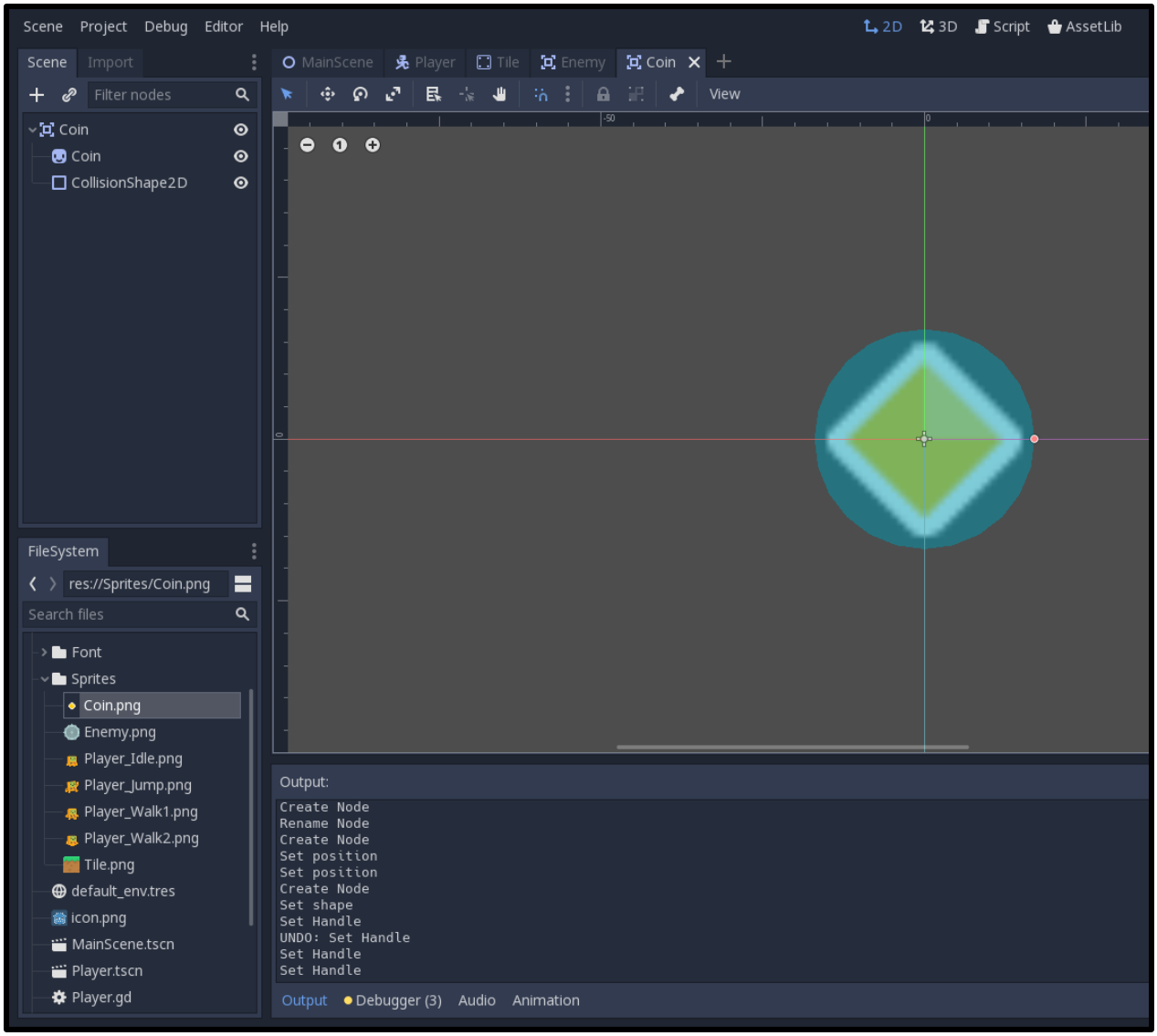
Collecting Coins

We want the player to achieve a certain goal, so let's add in some coins. Before we create the scene, let's go and set up the scripting side of things over in the **Player** script.

The **collect_coin** function gets called when we run into a coin and increments our score by the given parameter value.

```
# called when we run into a coin
func collect_coin (value):

    score += value
```

Next, create a new scene and make the root node an **Area2D**. Give it a collision shape node as a child with the shape being a circle. Then drag in the **Coin** sprite and center it.



On the Area2D root node, create a new script called **Coin**. This is just going to detect when a player has entered the collider, then call the collect_coin function.

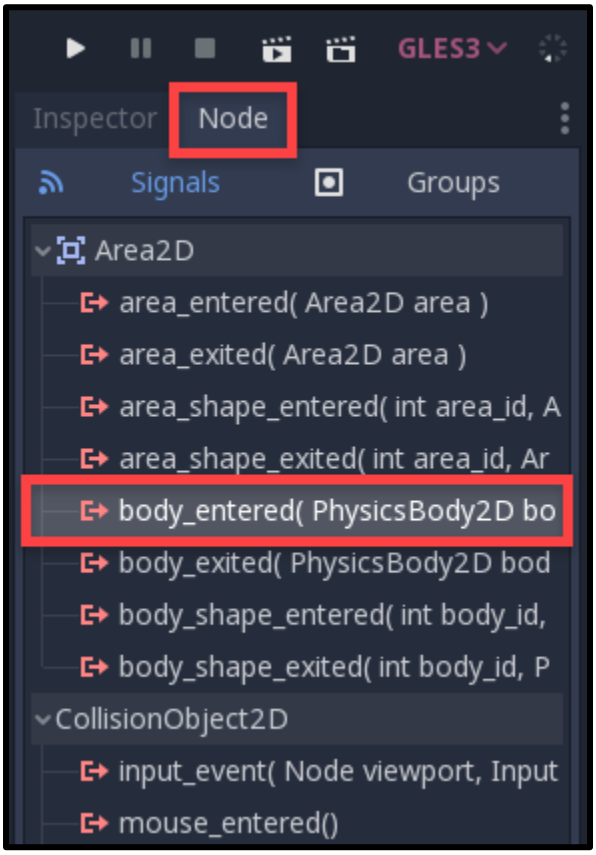
We'll start with our only variable which is just the value of the coin.

```
1 export var value = 1
```

The _**process** function is built into Godot and gets called every frame. We're going to use it in order to rotate the coin over time.

```
1  func _process (delta):
2
3      # rotate 90 degrees a second
4      rotation_degrees += 90 * delta
```

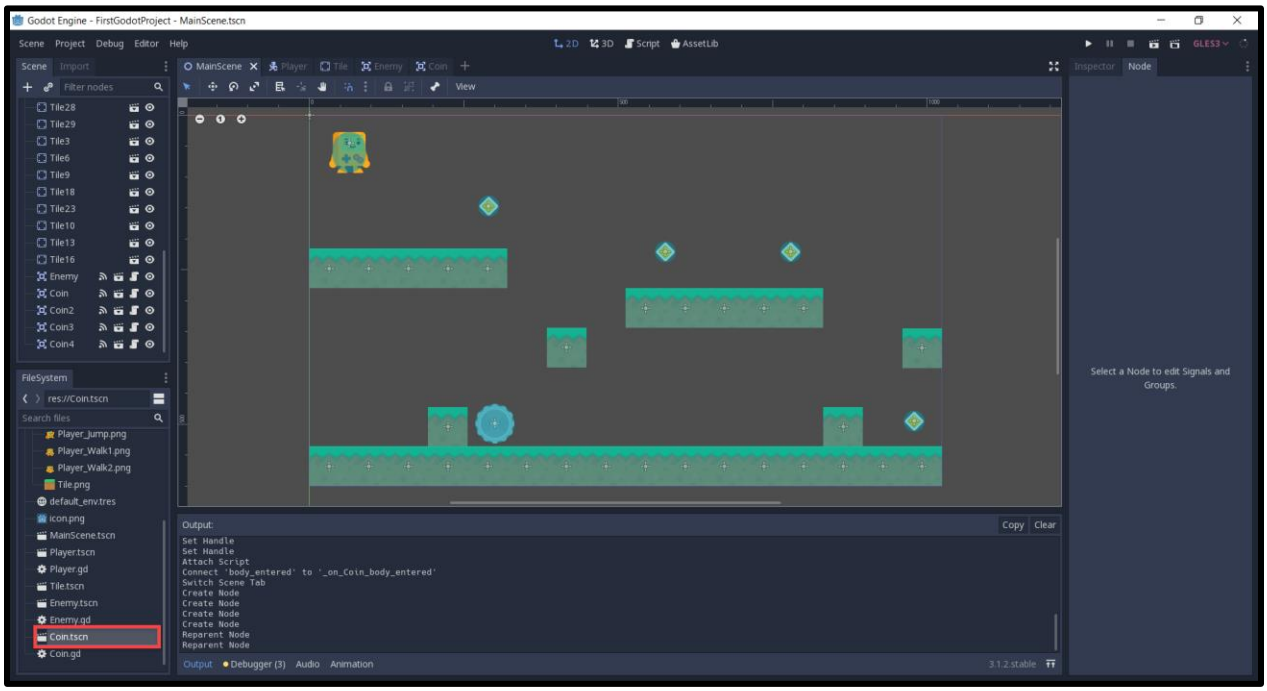
Next, we need to select the Area2D node, in the Inspector go to the **Node** tab and double-click on the **body_entered** signal to attach it to the script.



Back in the script, we can fill in the function to check if the entering body is the player. If so call the collect_coin function and destroy the coin.

```gdscript
# called when something collides with us
func _on_Coin_body_entered (body):

    # was it the player?
    if body.name == "Player":
        body.collect_coin(value)
        queue_free()
```

In the main scene, we can now drag in the coins to populate the level.

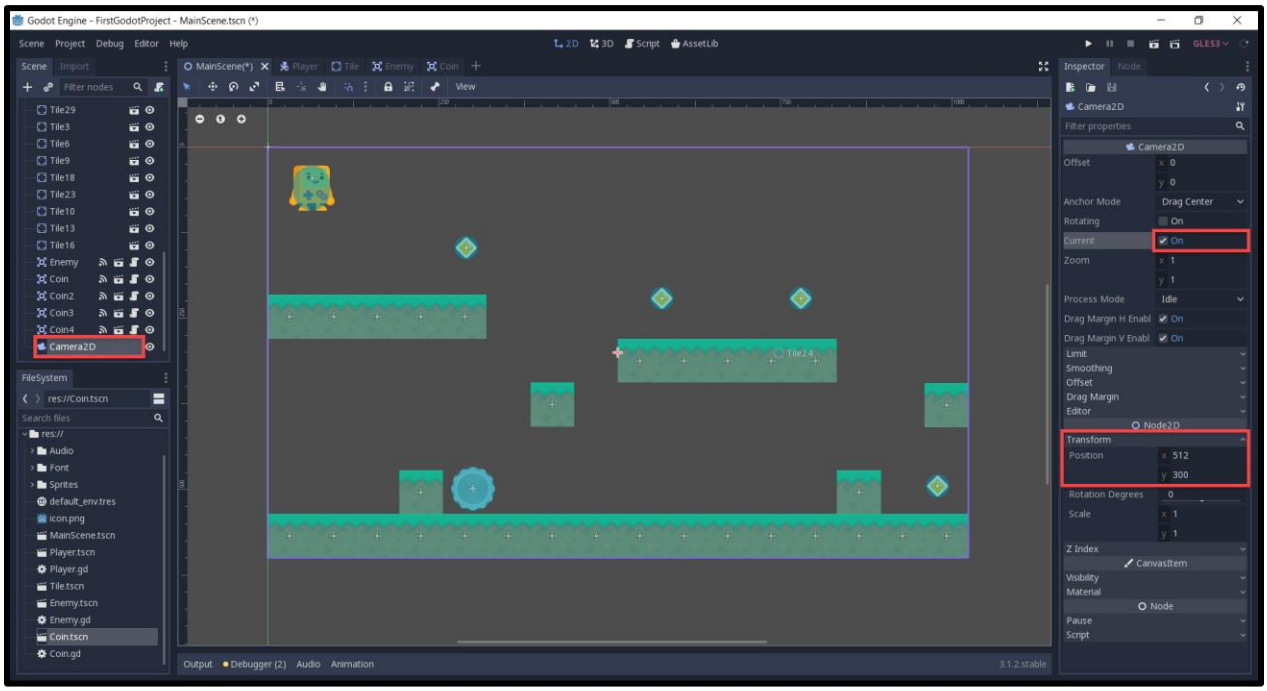


Tracking Camera

Right now our camera is static, so let's implement the ability for the camera to move horizontally.

The camera is a new node we'll create as right now, the game is being rendered just based on the position of the nodes and pixels, not the position of the camera. So in the **MainScene**, right click on the parent node and create a new child node of type **Camera2D**.

- Enable **Current** so that the camera will be active
- Move the camera to cover the level



Now we can create a new script on the camera node called **CameraController**.

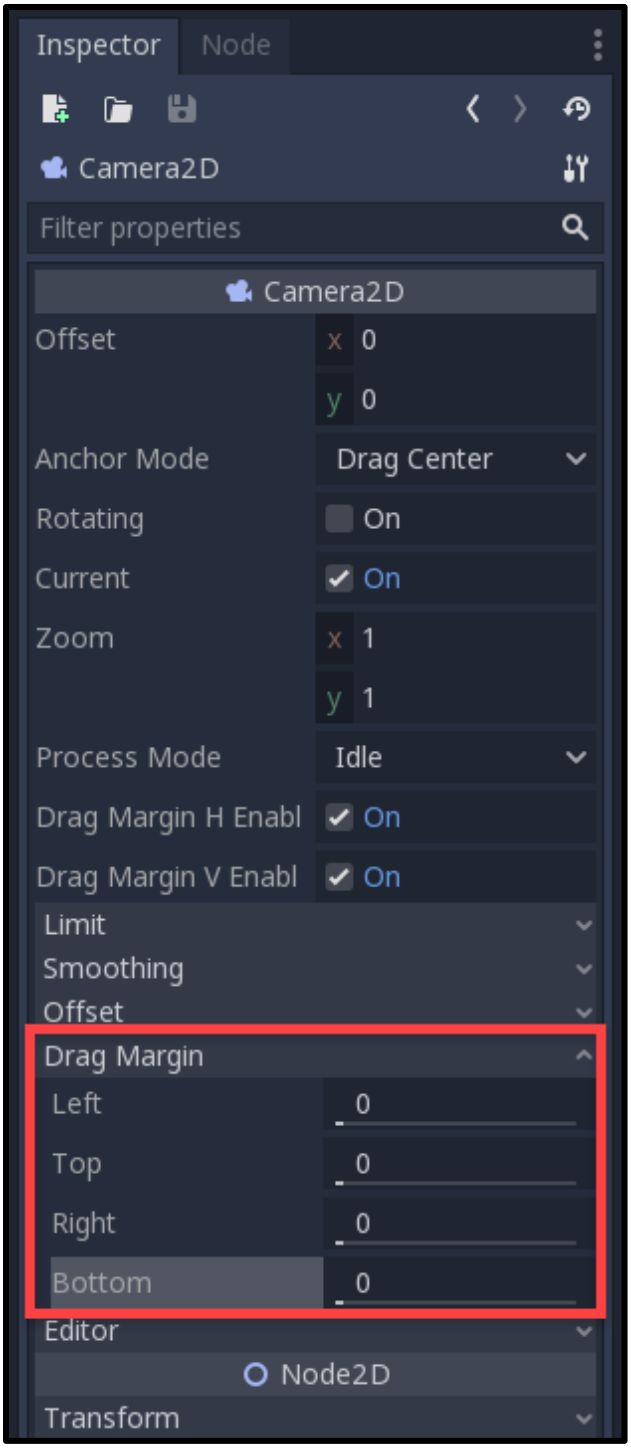
First, we want to create a variable to store a reference to the player in order to track them. **get_node** will search for a node based on the given path.

```
1 onready var player = get_node("/root/MainScene/Player")
```

Then in the **_process** function (called every frame), we can set our X position to be the same as the player's.

```
1 # tracks the player along the X axis
2 func _process (delta):
3
4     position.x = player.position.x
```

Now when we press play, you'll see that the camera follows along the player's X position. Except that it does act a bit weird. Select the camera and in the Inspector, set the **Drag Margins** to 0, 0, 0, 0. This means that there's no margin for us to move around freely, the camera will always be tracking our X position.



Press play and you should see that it's all working fine now.

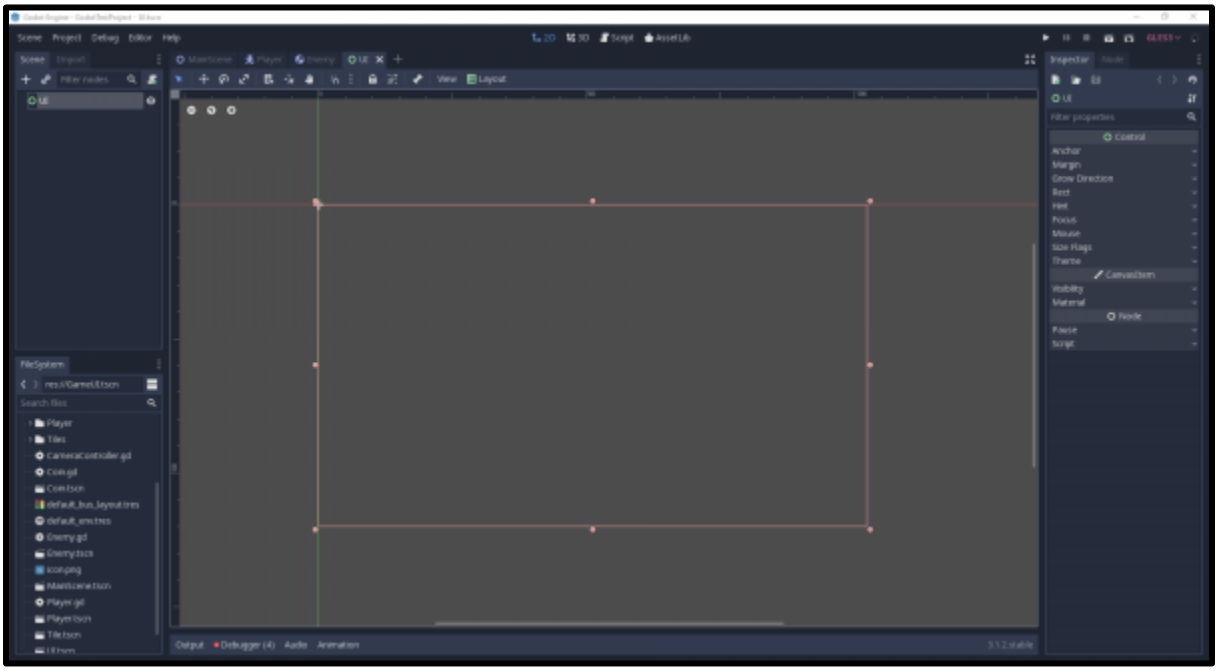
UI

Now that we have the game pretty much done, all we need to do is add in some sort of user interface to show us our current score.

To begin, create a new scene with the root node being **User Interface** (control).

1. Rename the node to **UI**
2. Save the scene

You'll see that there is a colored rectangle. This is the bounds of the control node which is how we build our UI elements.

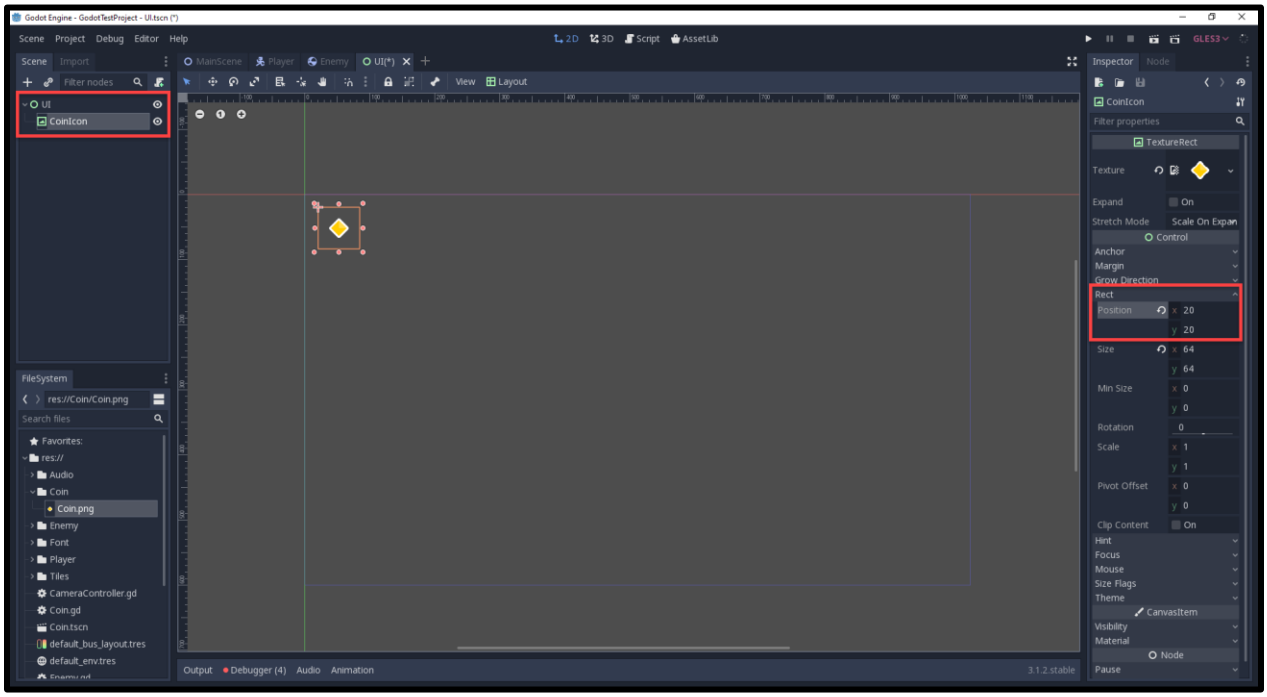


This rect right now is covering the whole screen. Let's add in two new things.

- A coin icon
- A text box to show our current score

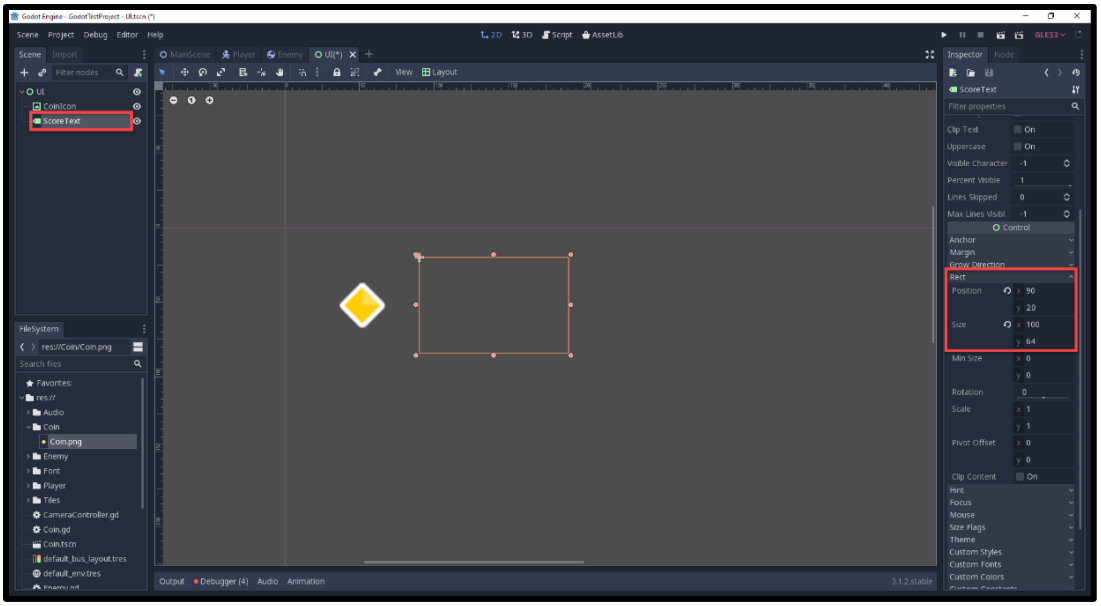
Let's start with the icon. Create a new child node of type **TextureRect**. This is a control node where we can show an image. In our file system, find the coin image and drag that into the **Texture** field in the inspector.

Also set the **Position** to 20, 20.



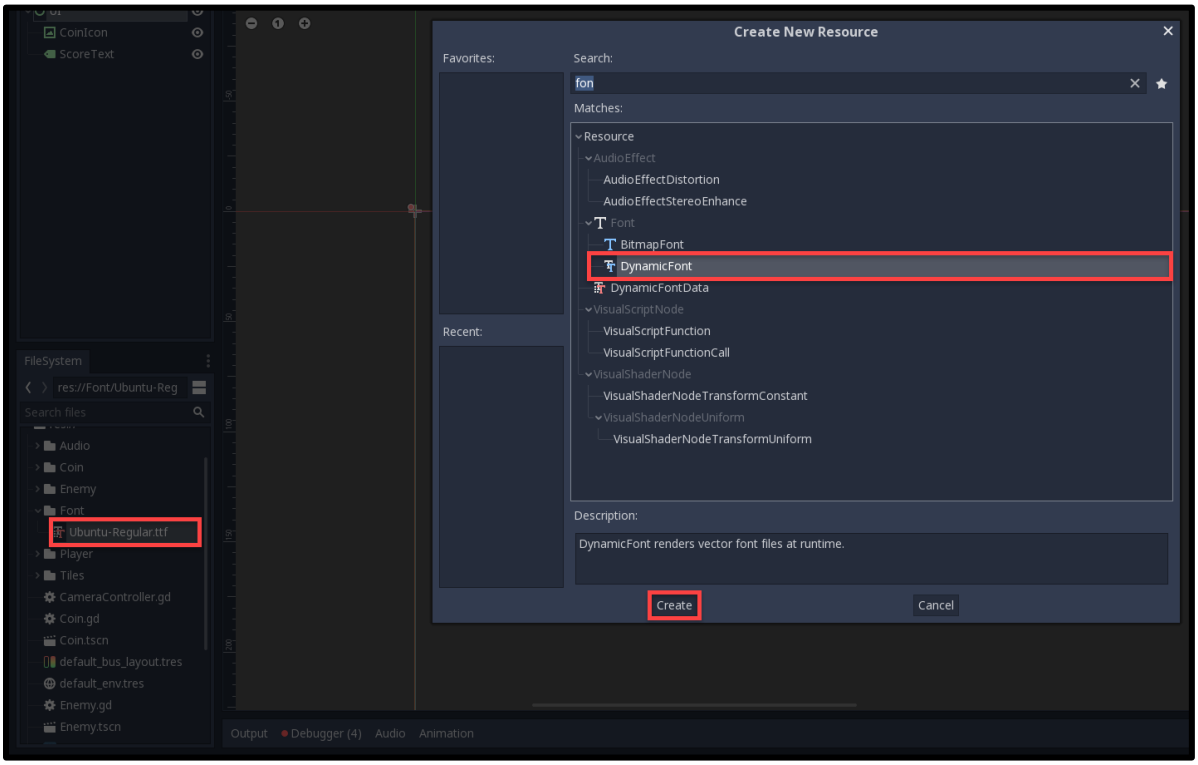
Then for the text, we want to create a new child node of type **Label**. This is a control node where we can display text.

1. Rename it to **ScoreText**
2. Set the **Position** to 90, 20
3. Set the **Size** to 100, 64

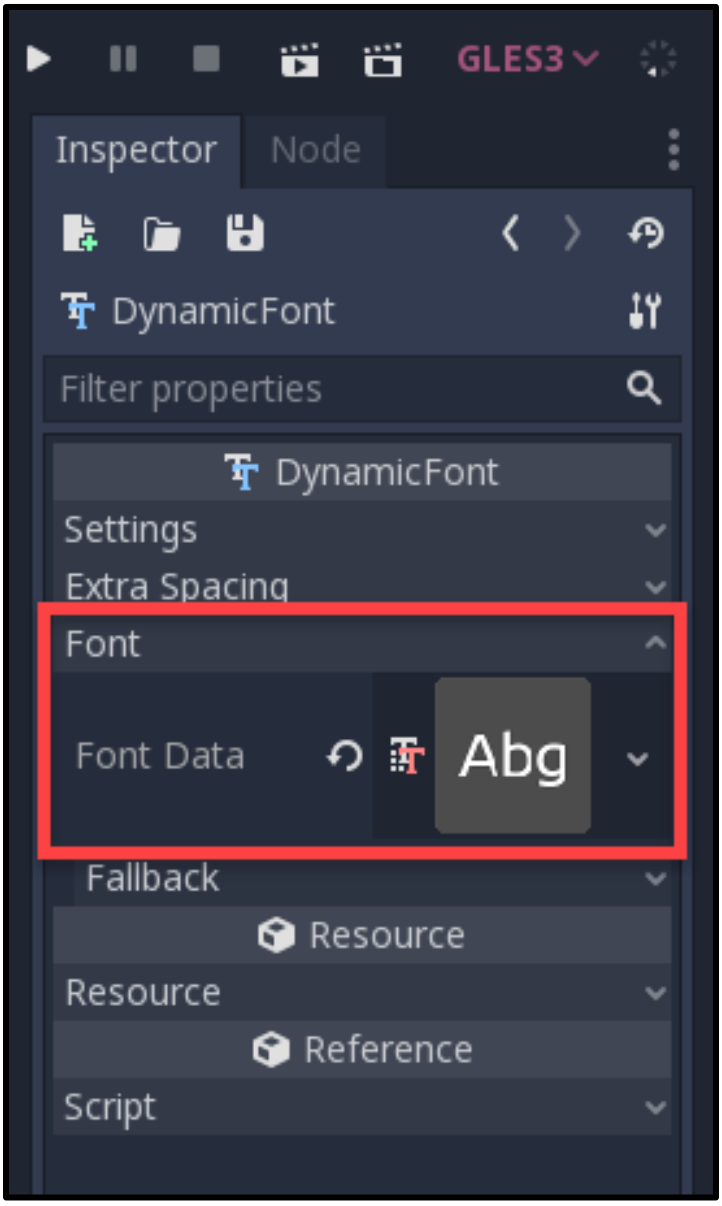


Right now we can't see our text, and that's because we need a font. There is a default font but it's quite limiting so we'll setup our own.

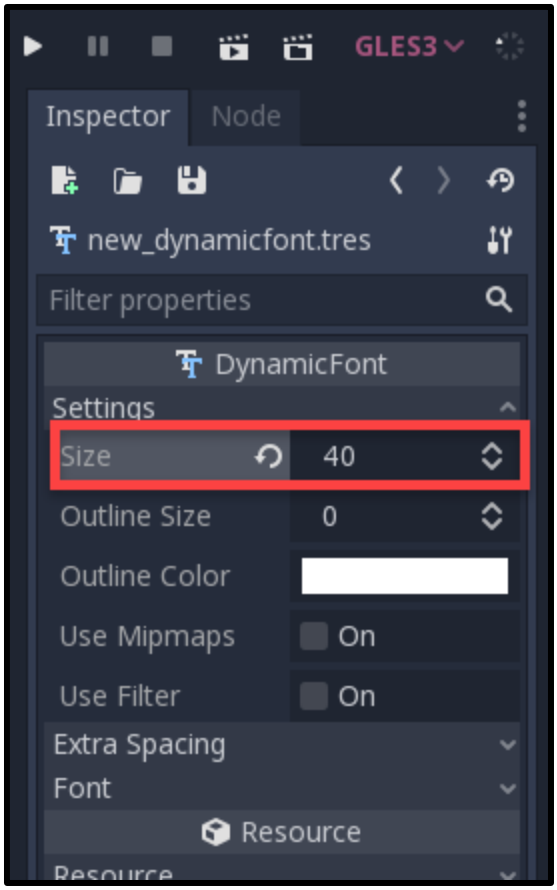
In the file system, open the Font folder, right click on the font file we have and select **New Resource...** Find the **DynamicFont** resource and create that.



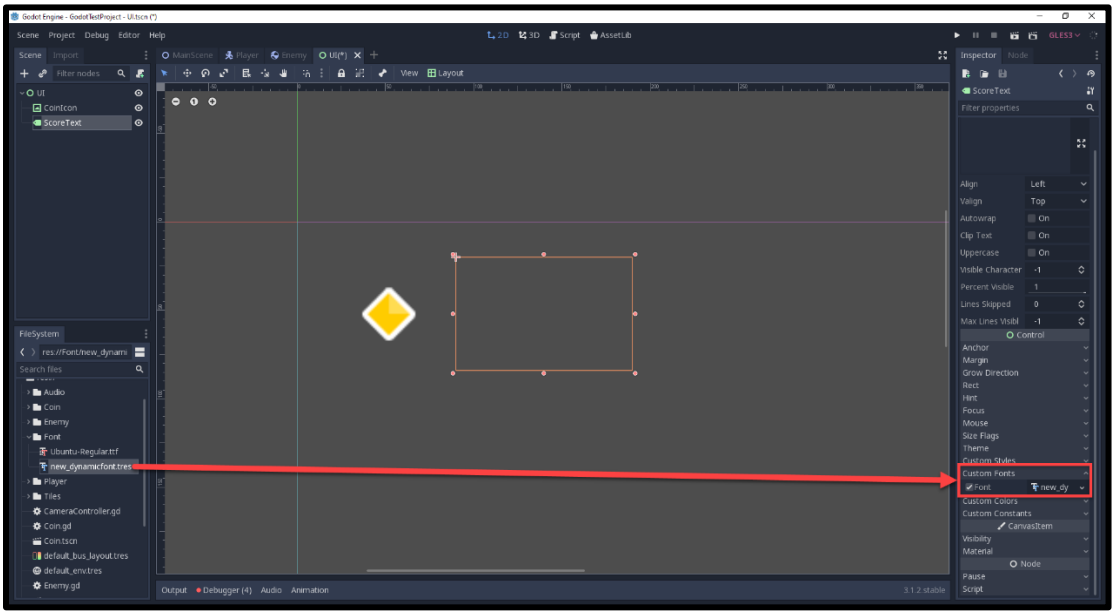
Select our new resource and in the inspector, drag the .ttf font file into the **Font Data** property to assign it.



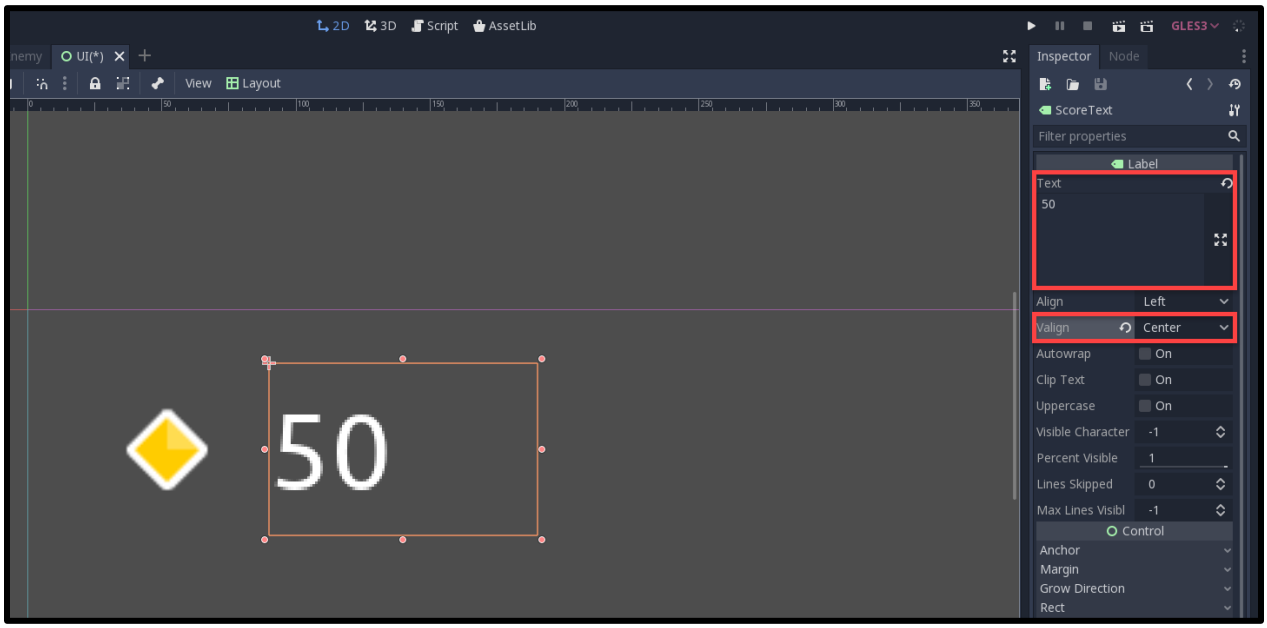
Under the Settings tab, set the **Size** to 40.



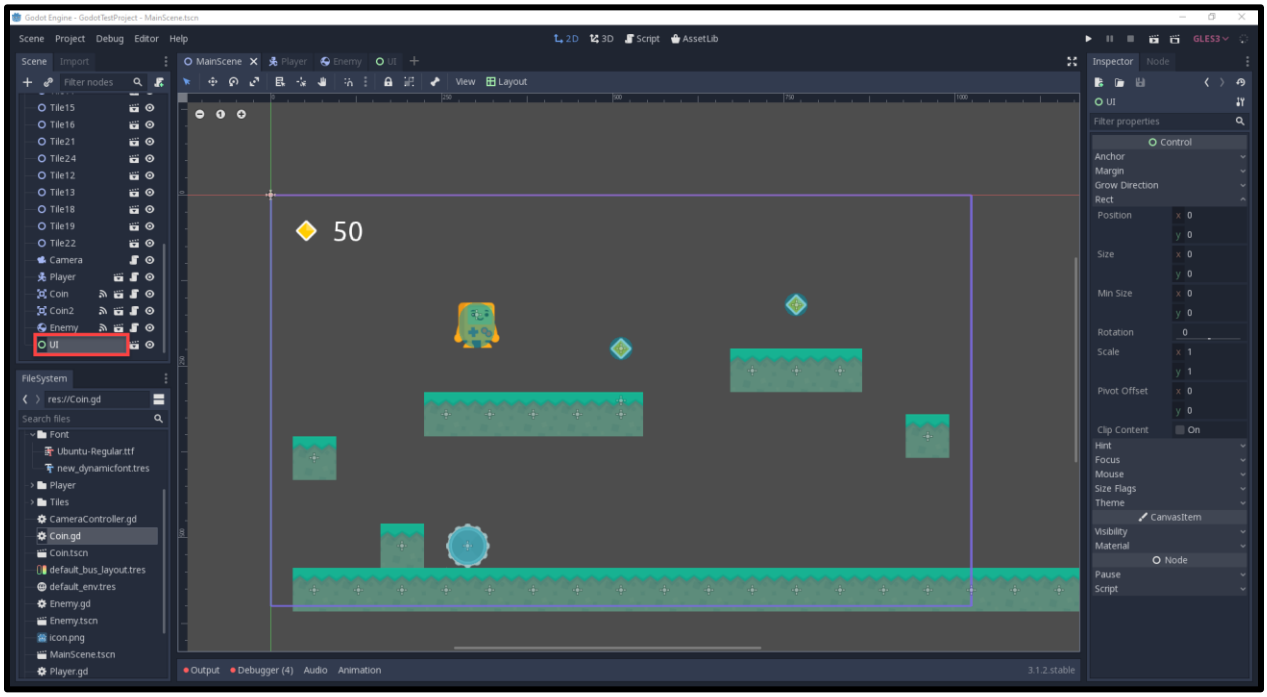
Select the **ScoreText** and under the Custom Fonts tab, drag in our new dynamic font resource.



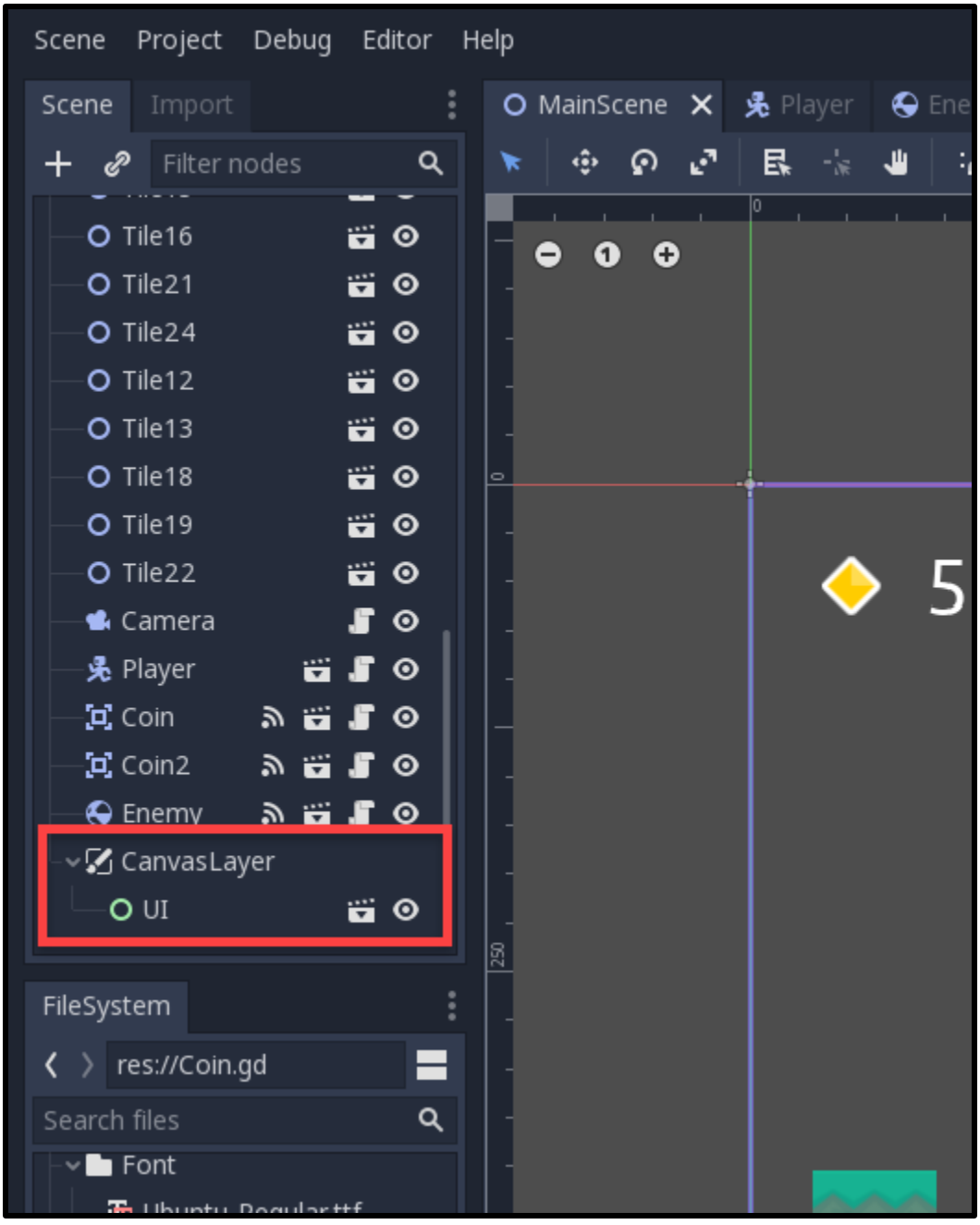
We can now type in a placeholder number up in the **Text** property. To make the text centered vertically, set the **Valign** property to Center.



Alright, we've got our UI setup. If we go to the **MainScene** we can drag the UI scene into the node list and it should appear.



If you press play though, you'll see that it doesn't follow the camera. To fix this, we need to make this UI scene a child of a new node called **CanvasLayer**. This node defines what gets rendered onto the screen as UI.



Scripting the UI

All that's left to do is connect the UI to a script. In the UI scene, create a new script called UI and attach it to the parent node.

First, we need to create a variable which will get the score text node upon starting the game.

```
1 onready var scoreText = get_node("ScoreText")
```

Then in our only function, we'll set the score text to whatever the parameter sends over.

```
1 func set_score_text (score):
2
3     scoreText.text = str(score)
```

Over in the **Player** script, let's create a variable to reference the UI node.

```
1 onready var ui = get_node("/root/MainScene/CanvasLayer/UI")
```

Then down in the **collect_coin** function, let's add a line of code to set the UI text.

```
1 ui.set_score_text(score)
```

If we press play now the score text will update when we collect a coin. Let's also set it up so that the text is set initially when the game begins. In the **UI** script...

```
1 func _ready ():
2
3     scoreText.text = "0"
```

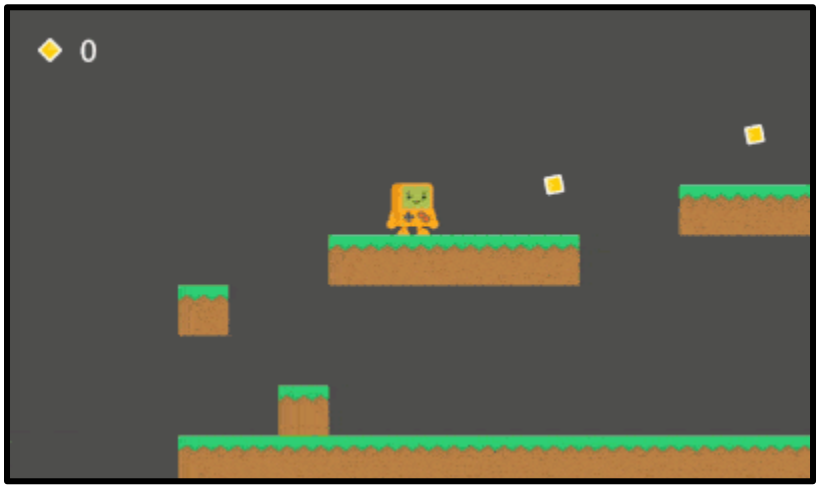
Now the text should be set to 0 when we start the game.

Conclusion

And that's it! If you hit Play on your Godot project now, everything should work. Congratulations on completing your first game!

We now have a functional 2D platformer with a playable character, coins, UI, and a dangerous enemy to obstruct our path. While you can certainly improve upon this framework to add even more challenges, you've no doubt seen how the foundations of the framework taught here will provide a useful starting point as you increase your game development skills further. The sky is truly the limit, and you can expand as needed to create new game projects with these fundamentals.

Either way, you've taken the first crucial step and created your first game! We wish you luck in the future, and we can't wait to see the sorts of games you create!

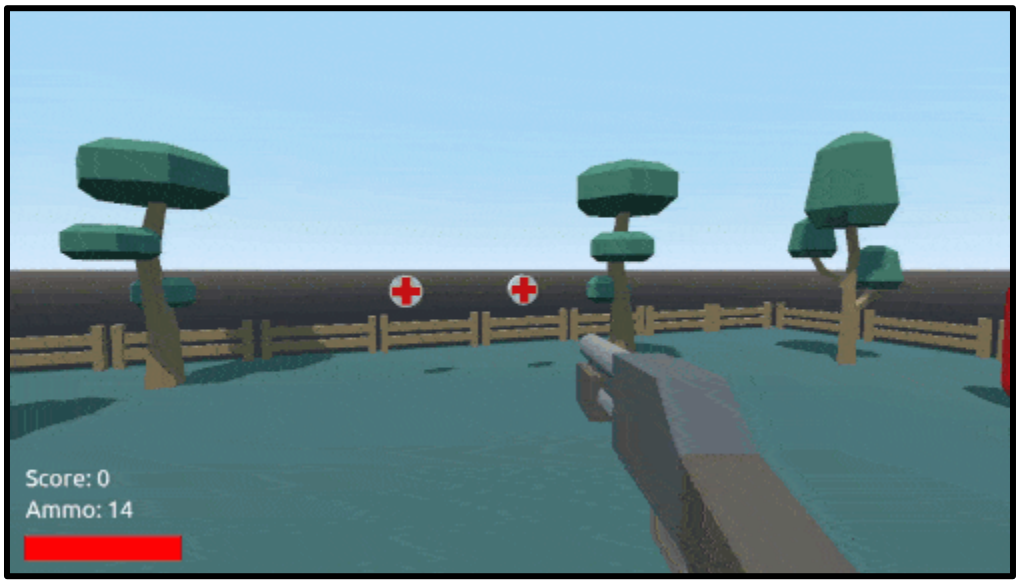


Create a First-Person Shooter in Godot - Part 1

Introduction

Welcome to the tutorial, and get ready to learn some in-demand skills for creating first-person shooter games! Throughout this tutorial, we will work with the 3D aspects of the Godot game engine to enhance your foundations when it comes to action game development. Additionally, we'll be creating a first-person shooter from scratch, which will show you how to set up the FPS player and camera, enemies, efficient shooting mechanics, health and ammo pickups, and more.

Before we begin, it's important to know that a basic understanding of the Godot engine is required. If this is your first time using Godot, I recommend you read through the How to Make First Game with Godot chapter first. It will teach you how to install Godot, and the basics of the engine and GDScript.



Project Files

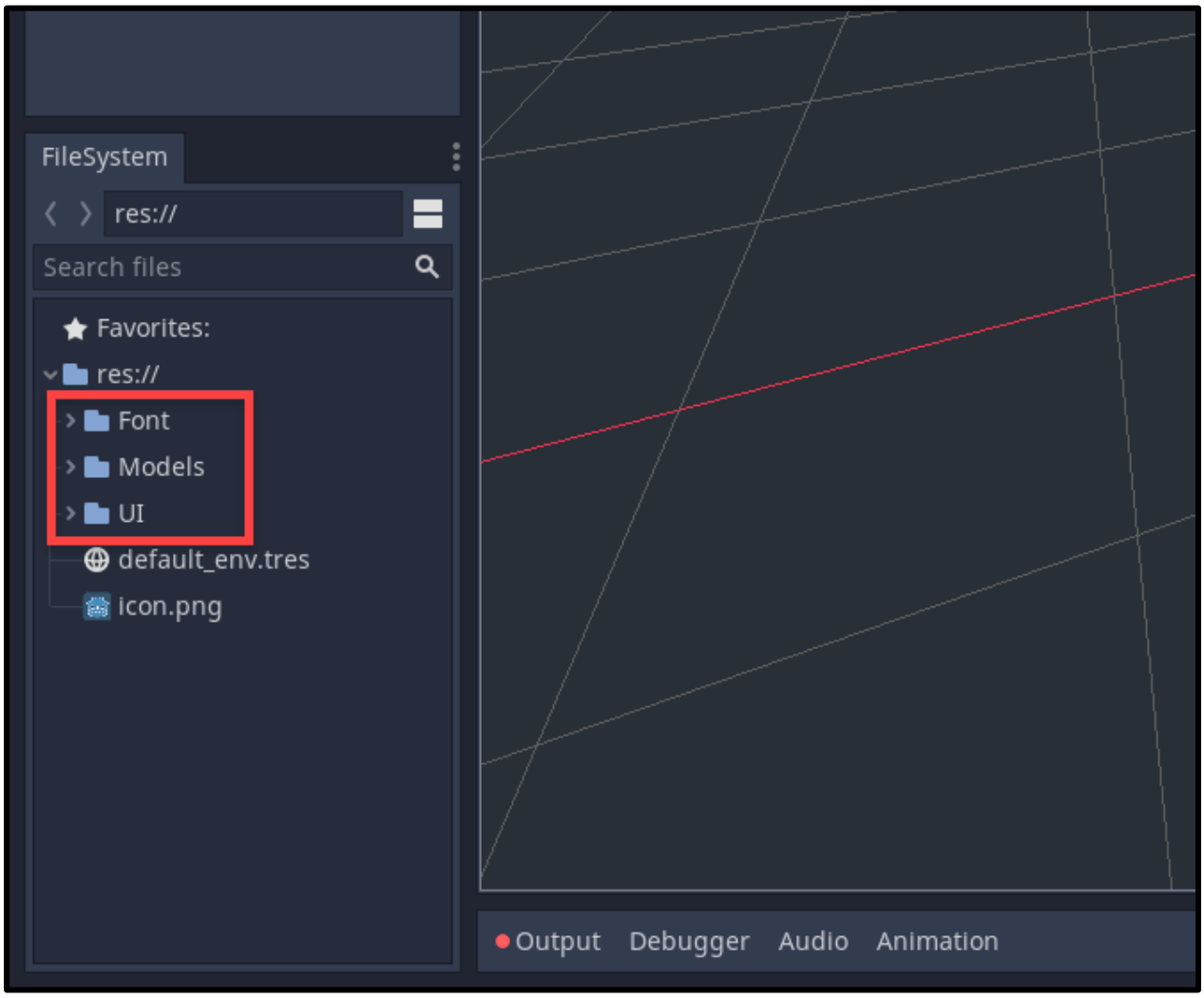
For this project, we'll be using a handful of pre-made assets such as models and textures. Some of these are custom-built, while others are from kenney.nl, a website for public domain game assets.

The assets can be downloaded here.

You can also download the completed Godot project via the same link!

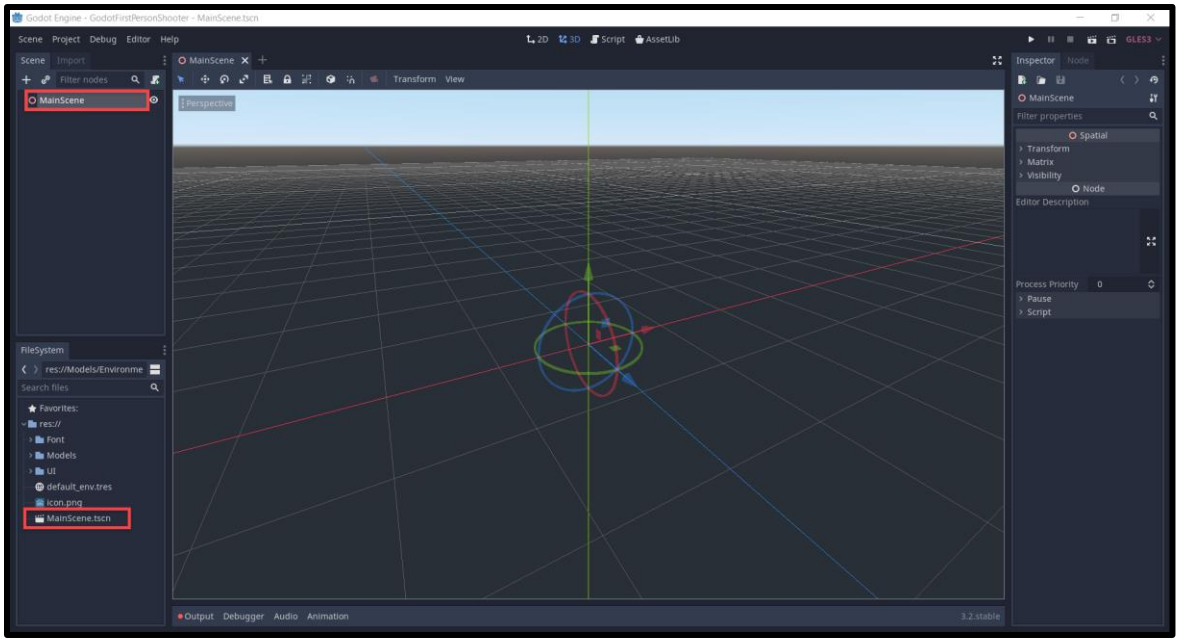
Let's Begin

To begin, create a new Godot project. We have a few assets such as models and textures we'll be using (download above), so let's drag the **Font**, **Models** and **UI** folders into the *FileSystem*.



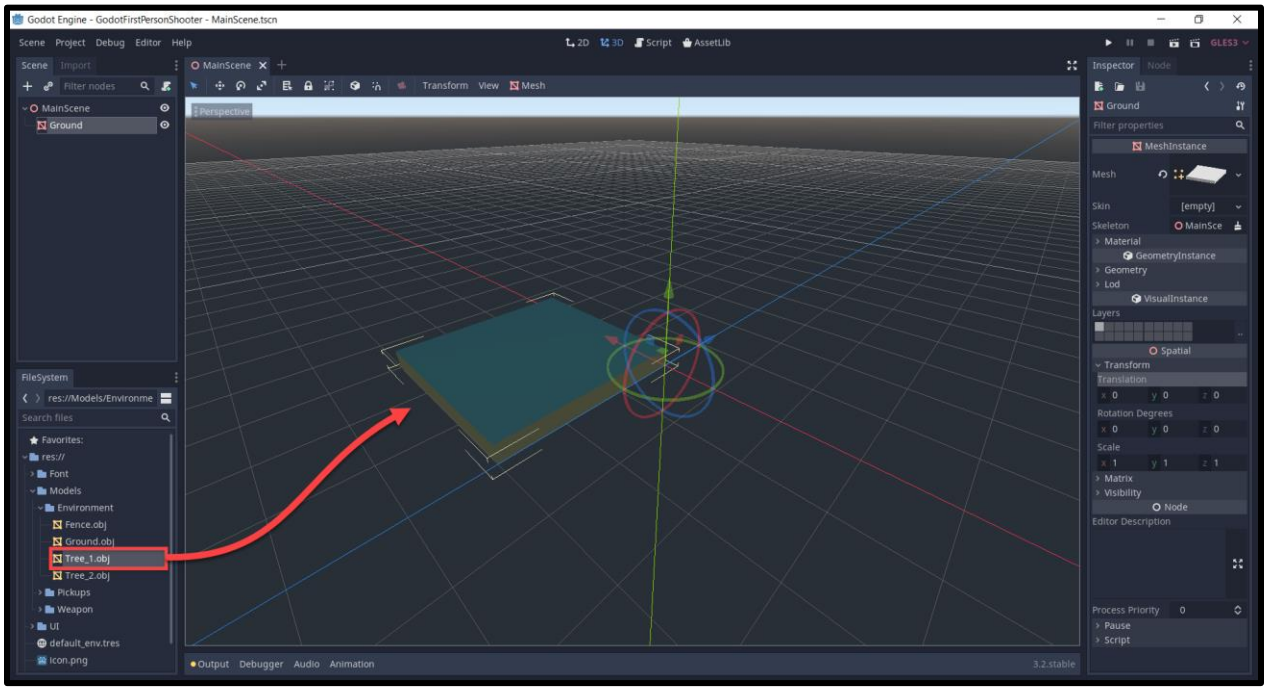
Now that we have our assets, let's create our first scene by going to the scene panel and selecting the **3D Scene** option.

1. Rename the node to **MainScene.**
2. Save the main scene to the file system.



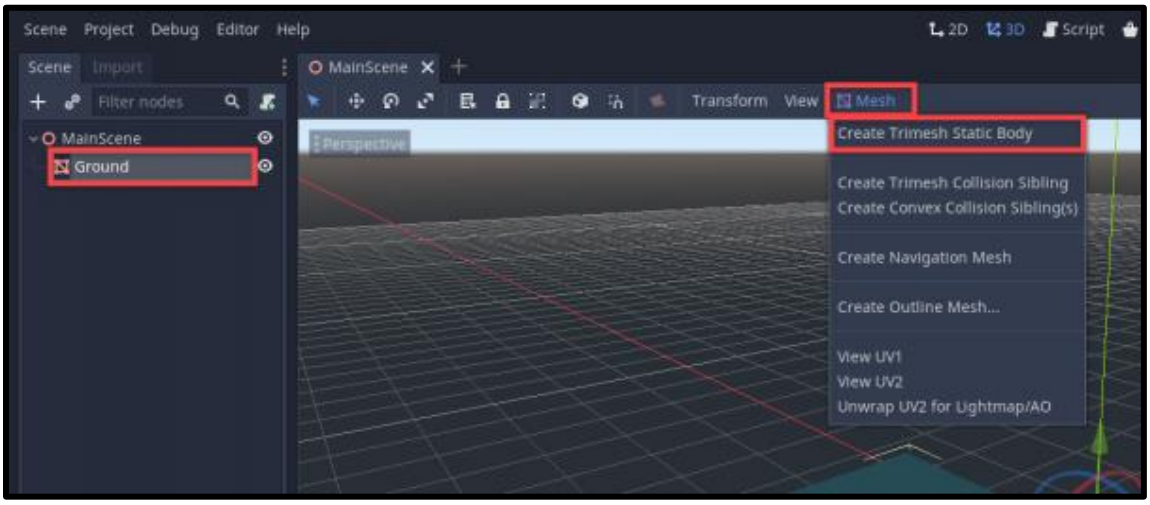
Building Our Environment

The first thing we're going to do is create our environment so we have somewhere for the player to exist. In our Models/Environment folder, we have a **Ground** model. Drag that into the scene window to create a new **MeshInstance** node. Set the position to 0.



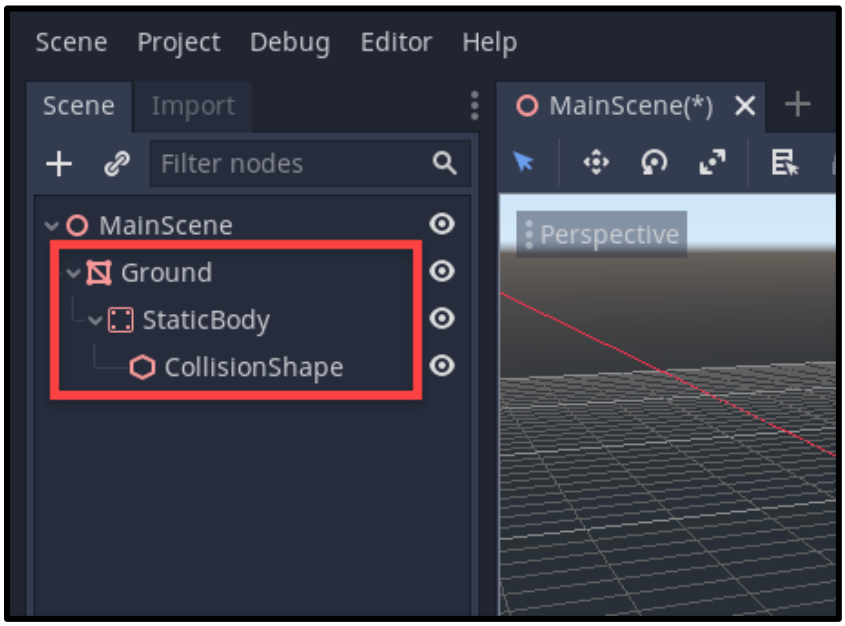
The MeshInstance node will render a model, but that's it. In our case, we want the player to be able to collide with the ground, otherwise they'd sink through.

So to do this, we can select the ground node and in the scene panel, select *Mesh > Create Trimesh Static Body*.

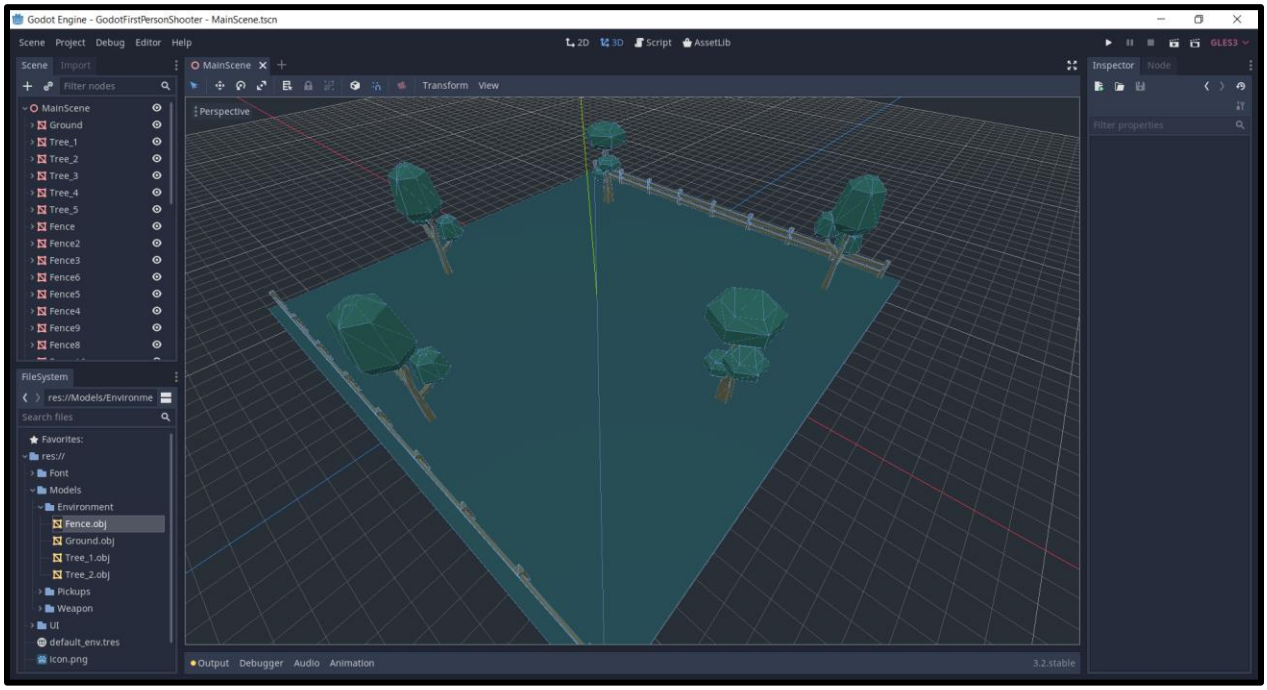


This will create two children nodes.

- **StaticBody**: static physics object which can detect collisions
- **CollisionShape**: defines the shape and dimensions of the object's collider

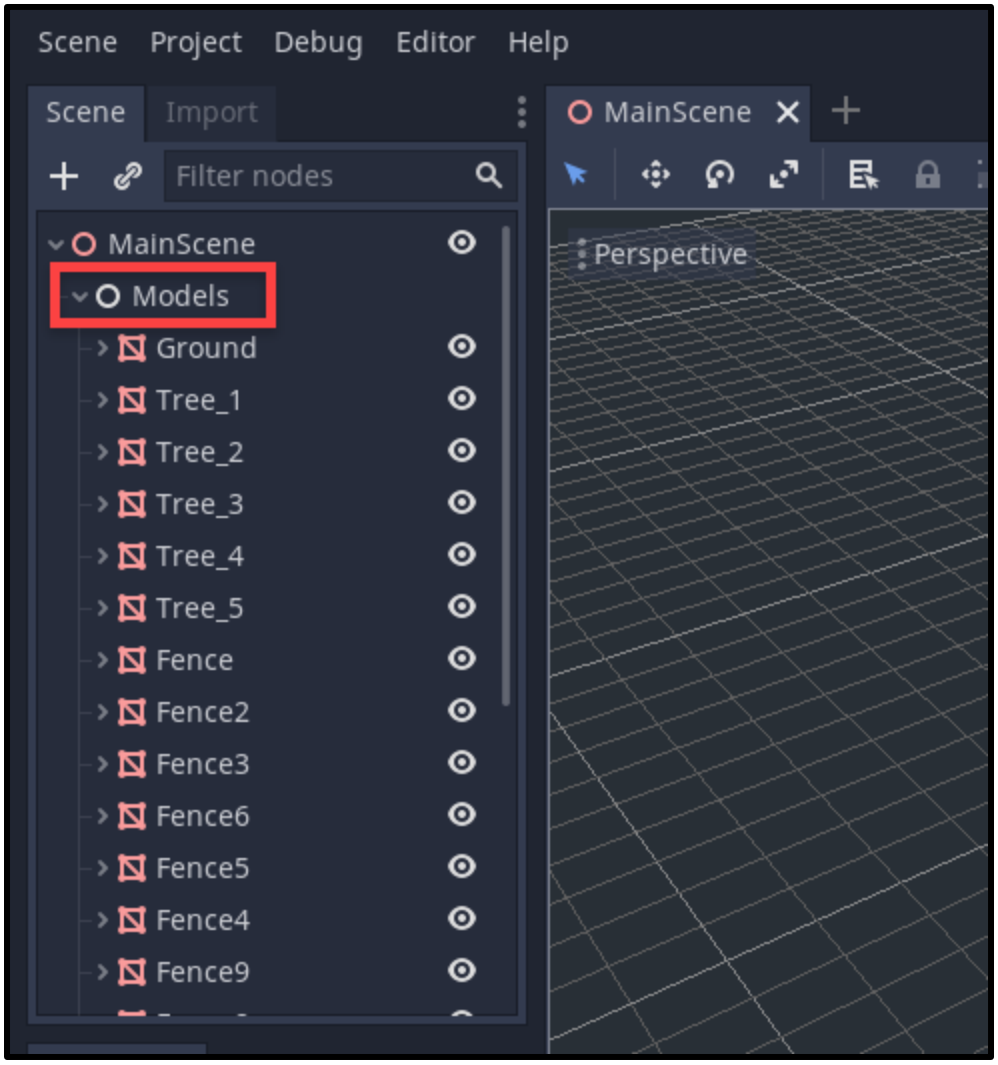


With this node, we can change the position, scale and rotation. Go ahead and create an environment.



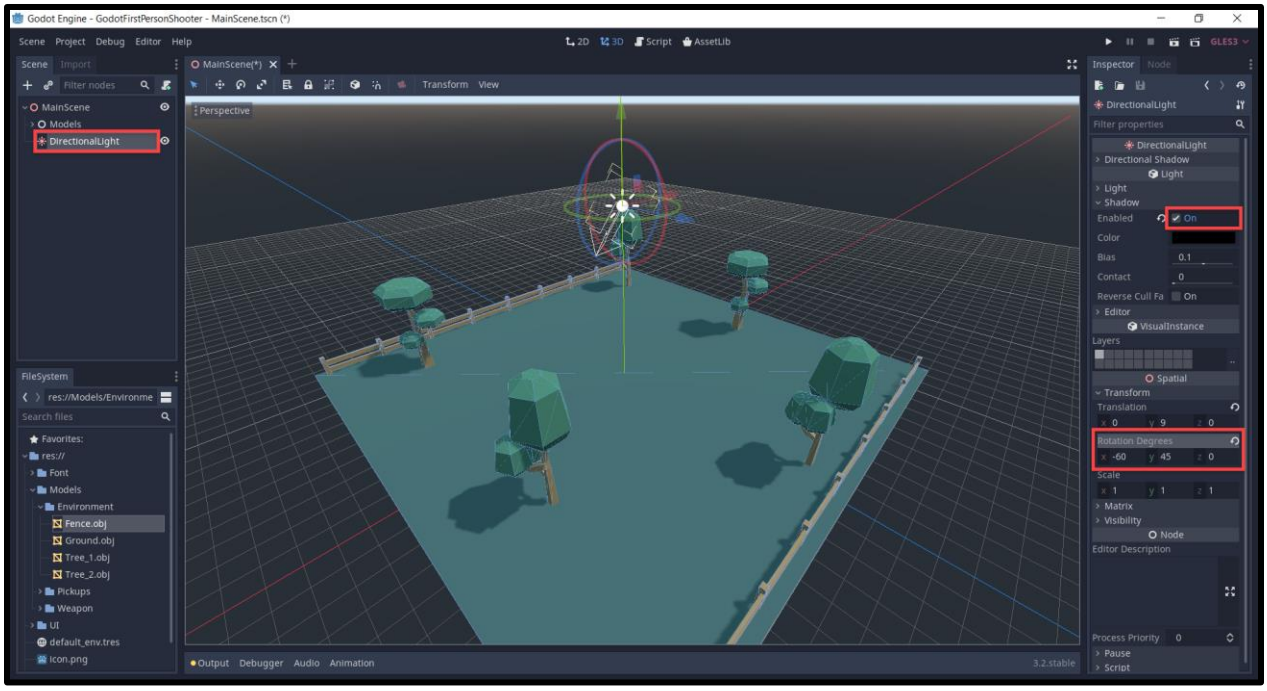
One thing you may notice, is that the node structure is starting to get quite long and convoluted since we have a lot of models. To fix this, we can create a new node as a container and store the models in as children.

Create a new node of type **Node** as a child of the MainScene. Drag the model nodes inside to make them children. Now we can visually open or close the node container.



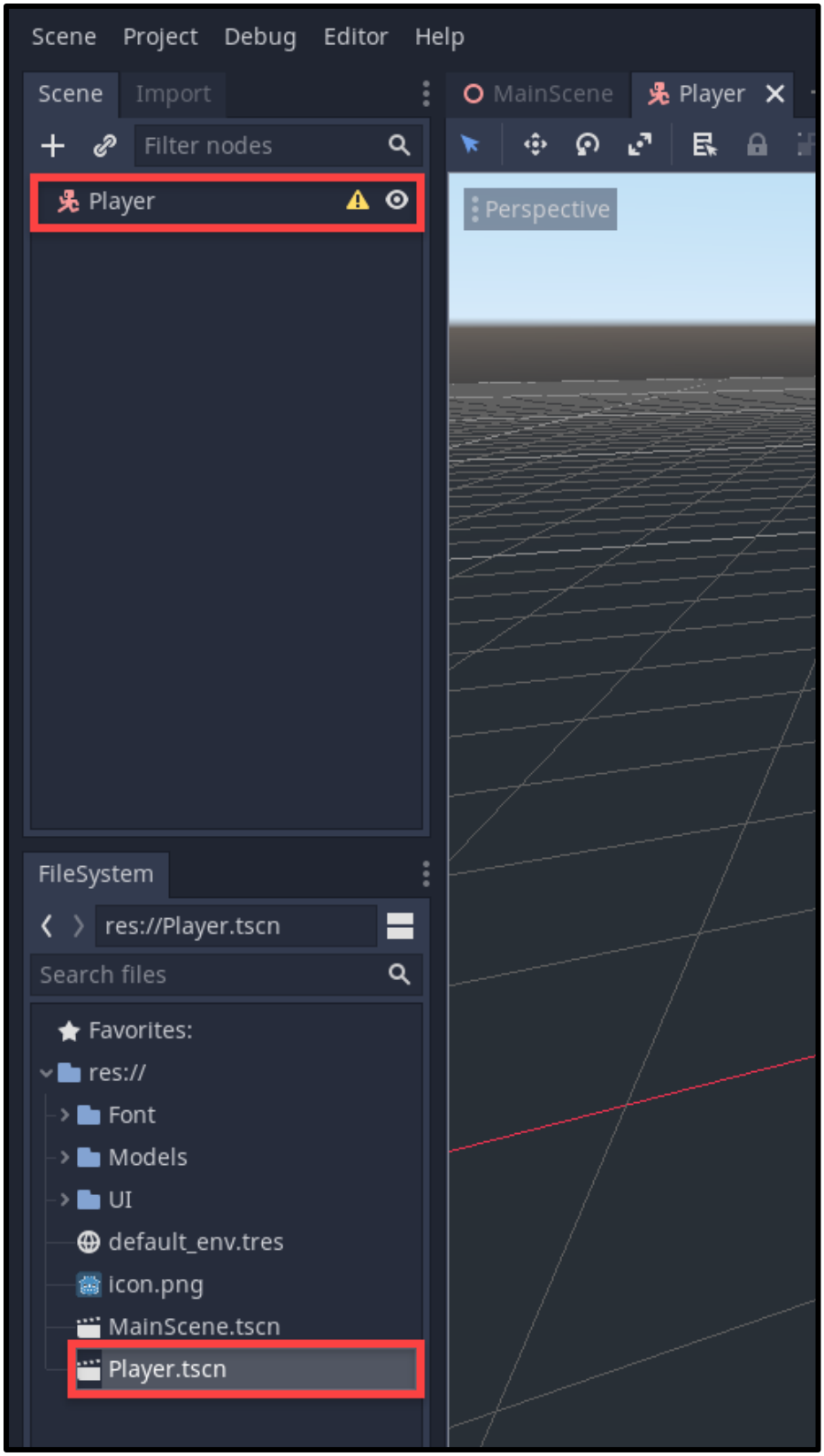
Another thing you might notice is that the scene is quite dark. To fix this, we can add in a directional light which will act like a sun.

1. Create a new node of type **DirectionalLight**.
2. Enable shadows.
3. Set the **Rotation Degrees** to -60, 45, 0.



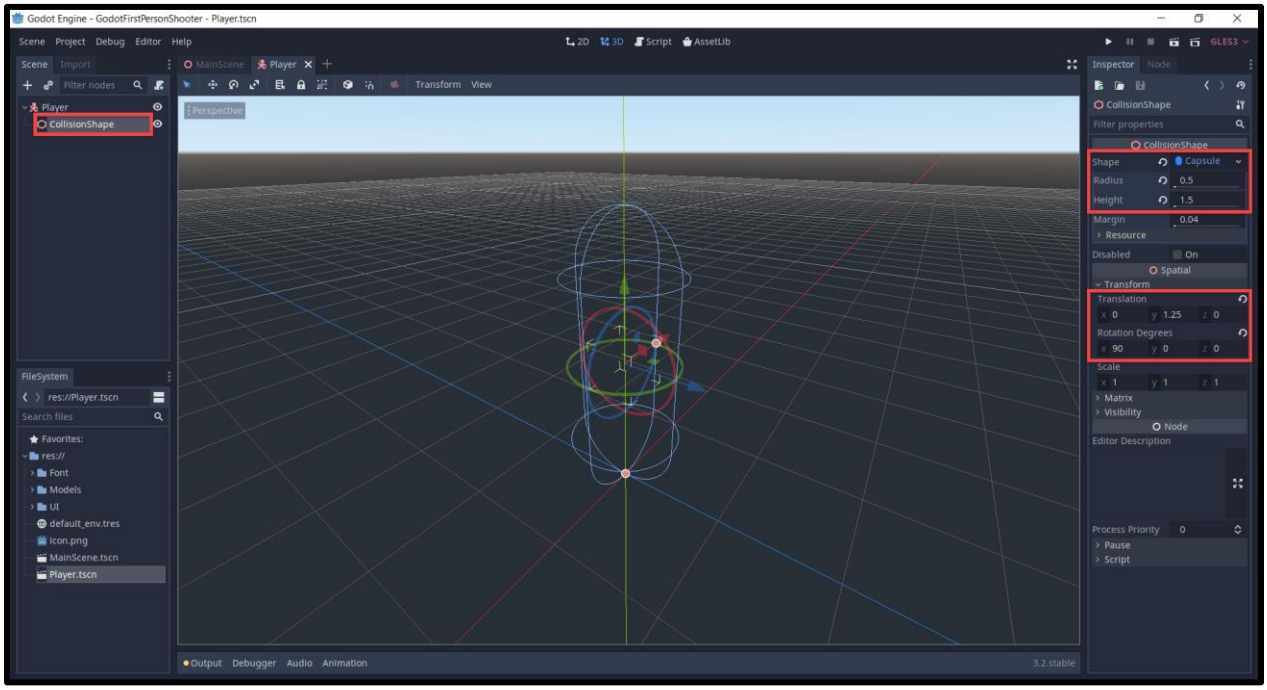
Creating the Player

Now we can create the player. Create a new scene with a root node of **KinematicBody** and rename it to **Player**. We can then save the scene.



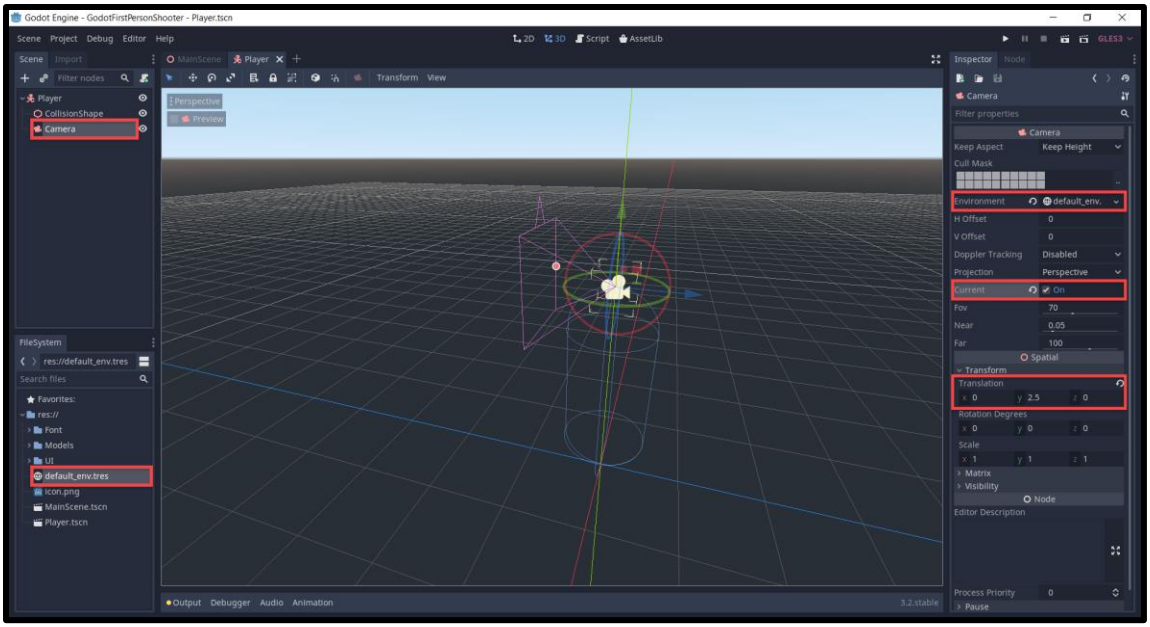
Let's add a **CollisionShape** node to detect collisions.

1. Set the **Shape** to Capsule
2. Set the **Radius** to 0.5
3. Set the **Height** to 1.5
4. Set the **Translation** to 0, 1.25, 0
5. Set the **Rotation** to 90, 0, 0



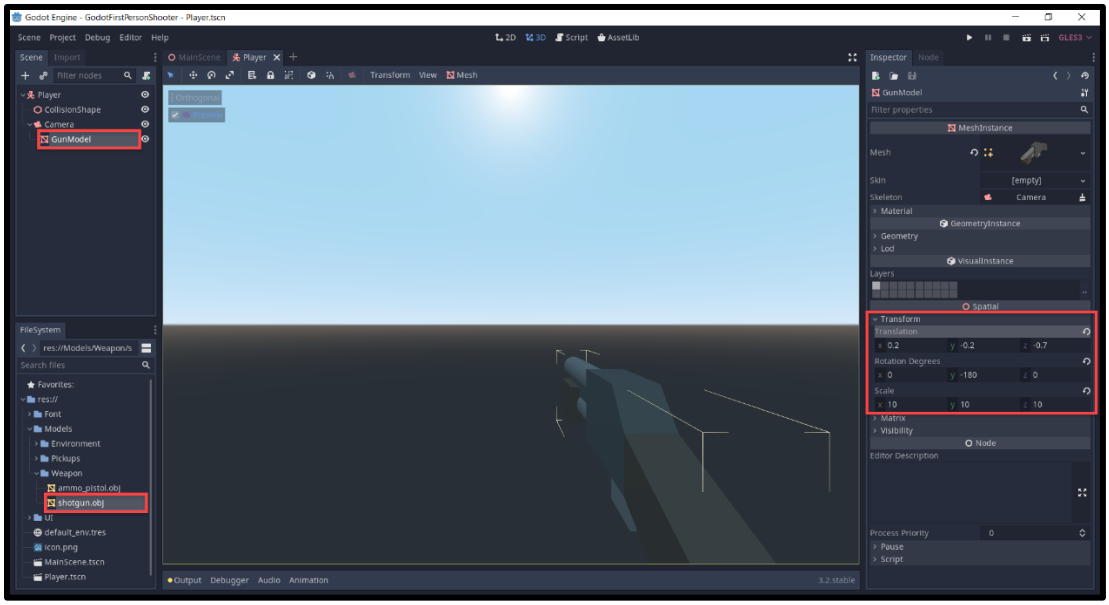
Next, add a new **Camera** node. This will allow us to see through the player's eyes.

1. Set the **Environment** to default_env.tres (found in file system)
2. Enable **Current**
3. Set the **Translation** to 0, 0.25, 0



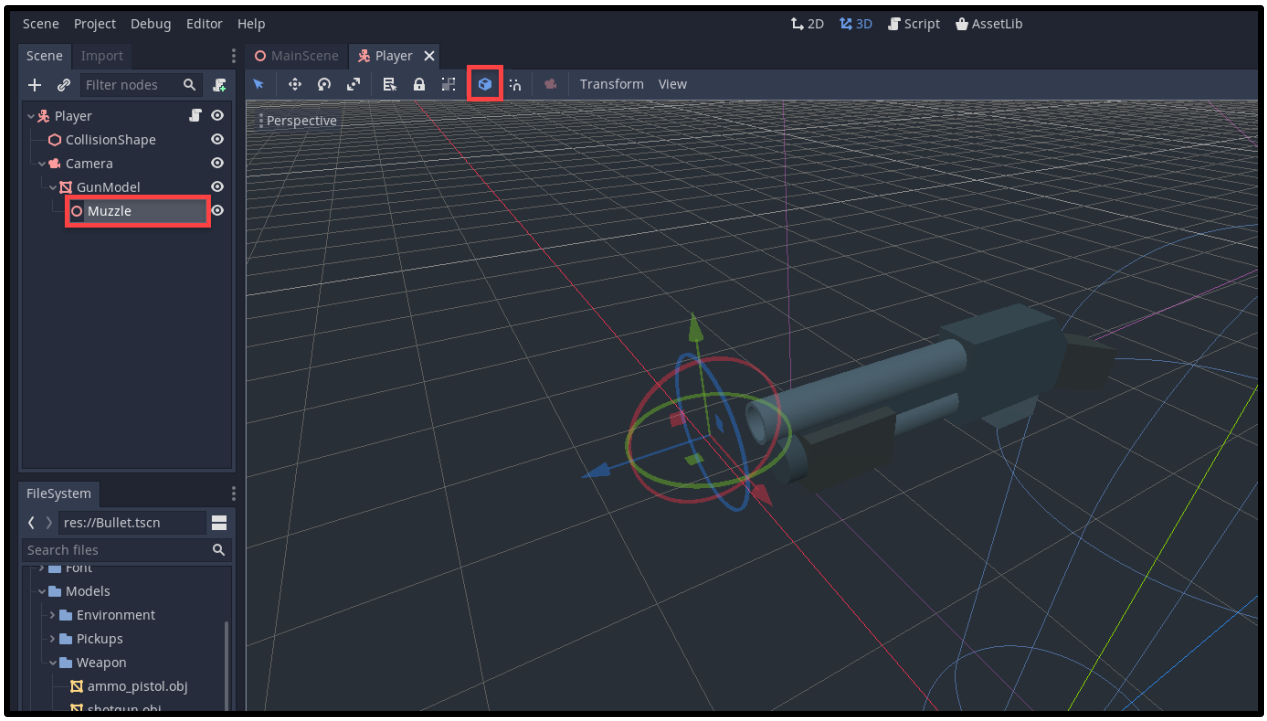
For the gun, drag the **shotgun.obj** model into the scene and make it a child of Camera. To see through the camera's perspective, select the camera and in the scene view toggle the **Preview** option (top left of the window).

1. Rename the node to **GunModel**
2. Set the **Translation** to 0.2, -0.2, -0.7
3. Set the **Rotation Degrees** to 0, -180, 0
4. Set the **Scale** to 10, 10, 10

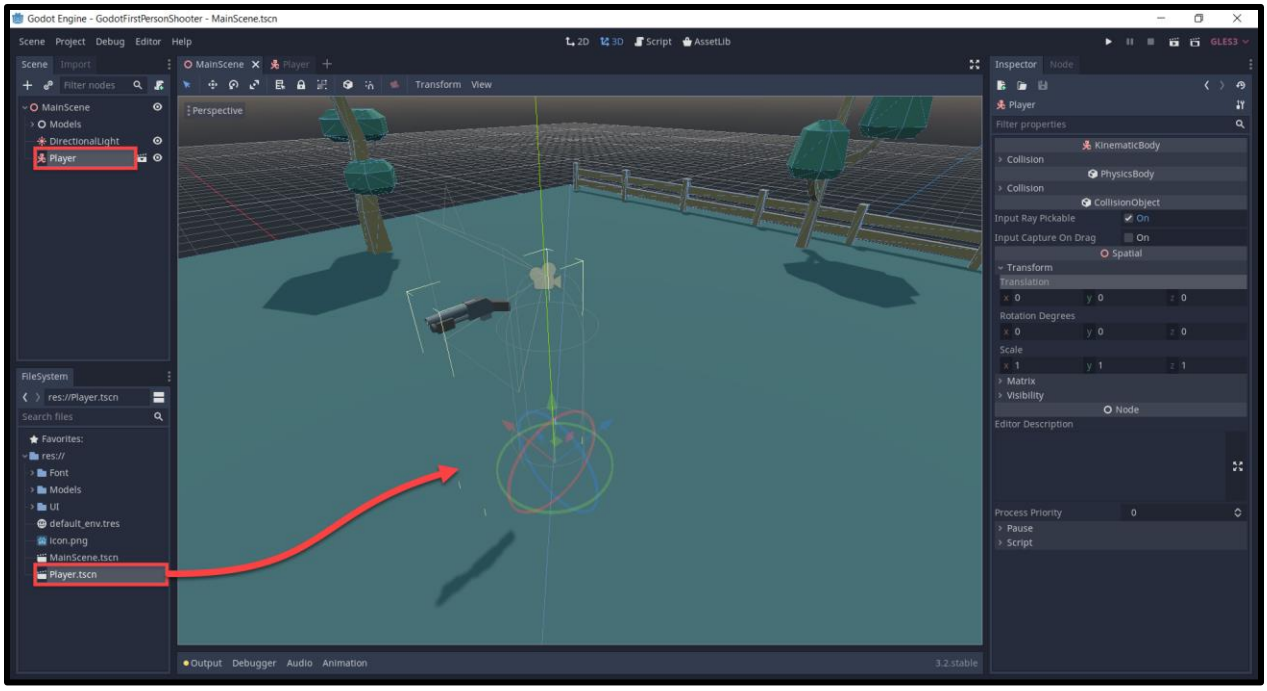


The gun also needs a muzzle so we know where to spawn the bullets. Create a new **Spatial** node as a child of **GunModel**.

1. Enable **Use Local Space** (shortcut = **T**) so that the gizmos are orientated in local space
2. Rename the node to **Muzzle**
3. Position the node in front of the gun



Back in the MainScene, let's drag in the **Player.tscn** to create a new instance of the Player scene.



With this here, we can press the play button (make the MainScene the main scene) and you should be looking through the player's eyes.

Finally, before we begin scripting, let's set up our input actions. Open the **Project Settings** window (Project > Project Settings) and go to the **Input Map** tab. Create a new action for:

- move_forward (W key)
- move_backward (S key)
- move_left (A key)
- move_right (D key)
- jump (Space key)
- shoot (Left Mouse Button)

Scripting the Player

Now that we have our player scene setup, let's get to scripting the player movement, jumping, gravity, shooting, etc.

In the **Player** scene, create a new script on the **Player** node. By default, it should be called Player.gd. We'll begin by filling in our variables.

```gdscript
# stats
var curHp : int = 10
var maxHp : int = 10
var ammo : int = 15
var score : int = 0

# physics
var moveSpeed : float = 5.0
var jumpForce : float = 5.0
var gravity : float = 12.0

# cam look
var minLookAngle : float = -90.0
var maxLookAngle : float = 90.0
var lookSensitivity : float = 0.5

# vectors
var vel : Vector3 = Vector3()
var mouseDelta : Vector2 = Vector2()

# player components
onready var camera = get_node("Camera")
```

First, we can begin by setting up the ability to look around. The **_Input** function is built into Godot and gets called whenever an input is detected (keyboard, mouse click and movement). What we're doing here, is getting the direction and length that our mouse moved.

```
# called when an input is detected
func _input (event):

    # did the mouse move?
    if event is InputEventMouseMotion:
        mouseDelta = event.relative
```

Next, we'll create the _**process** function. This is built into Godot and gets called every frame. Here, we're going to use that mouseDelta vector and apply that to the camera and player rotations.

```
# called every frame
func _process (delta):

    # rotate camera along X axis
    camera.rotation_degrees -= Vector3(rad2deg(mouseDelta.y), 0, 0) * lookSensitivity * delta

    # clamp the vertical camera rotation
    camera.rotation_degrees.x = clamp(camera.rotation_degrees.x, minLookAngle, maxLookAngle)

    # rotate player along Y axis
    rotation_degrees -= Vector3(0, rad2deg(mouseDelta.x), 0) * lookSensitivity * delta

    # reset the mouse delta vector
    mouseDelta = Vector2()
```

We can now press **Play** and see that we look around when we move the mouse.

Next up is movement. This will all be done in the _**physics_process** function. It's built into the KinematicBody node and gets called 60 times a second (good for physics).

```
# called every physics step
func _physics_process (Delta):
```

First, we're going to check for our keyboard inputs.

```gdscript
# reset the x and z velocity
vel.x = 0
vel.z = 0

var input = Vector2()

# movement inputs
if Input.is_action_pressed("move_forward"):
    input.y -= 1
if Input.is_action_pressed("move_backward"):
    input.y += 1
if Input.is_action_pressed("move_left"):
    input.x -= 1
if Input.is_action_pressed("move_right"):
    input.x += 1

# normalize the input so we can't move faster diagonally
input = input.normalized()
```

Next, we need to get the forward and right direction of our player, so we know which way we're facing and can apply that to our velocity.

```gdscript
# get our forward and right directions
var forward = global_transform.basis.z
var right = global_transform.basis.x
```

We can then set our velocity, apply gravity and move the player!

```
# set the velocity
vel.z = (forward * input.y + right * input.x).z * moveSpeed
vel.x = (forward * input.y + right * input.x).x * moveSpeed

# apply gravity
vel.y -= gravity * delta

# move the player
vel = move_and_slide(vel, Vector3.UP)
```

Finally, we're going to check for the jump action and change the Y velocity when that happens.

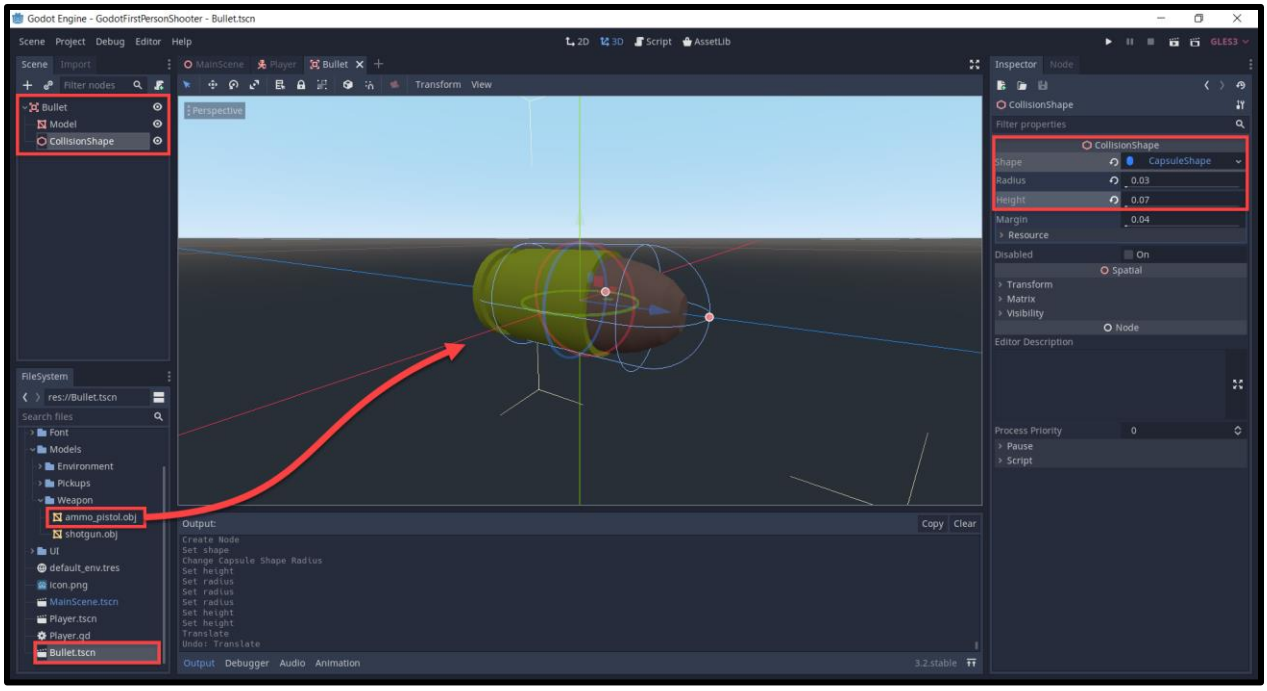
```
# jump if we press the jump button and are standing on the floor
if Input.is_action_pressed("jump") and is_on_floor():
    vel.y = jumpForce
```

If we press the play button now, you'll be able to move, look around and jump. That's how to create a basic first-person controller in Godot. Now let's get working on the rest of the game.

Creating the Bullet Scene

Before we start to script the ability to shoot, we need to create a scene for the bullet. Create a new scene with a root node of **Area**. This is a node which can detect physics collisions.

1. Save the scene to the *FileSystem*.
2. Drag the **ammo_pistol.obj** model into the scene as a child of Bullet.
3. Set the model's **Scale** to 10, 10, 10. (not in image)
4. Set the model's **Rotation Degrees** to 90, 0, 0. (not in image)
5. Attach a new **CollisionShape** node.
6. Set the **Shape** to Capsule
7. Set the **Radius** to 0.03
8. Set the **Height** to 0.07



On the **Bullet** node, create a new script called **Bullet.gd**. We'll start of with the variables.

```
1 var speed : float = 30.0
2 var damage : int = 1
```

Inside of the _**process** function, we'll move the bullet forwards at a rate of *speed* units per second.

```
1 # called every frame
2 func _process (delta):
3
4     # move the bullet forwards
5     translation += global_transform.basis.z * speed * delta
```

Now it's time to detect if we're hitting something. Select the **Bullet** node and over in the inspector, click on the **Node** tab to see all of the signals that the node emits. A signal is like an event which can call a function when something happens. In our case, we want to connect

the **body_entered** signal to our script. This gets called when the collider enters another node's collider.

Double click on that signal and hit enter. A new function should have appeared in the bullet script.

We're going to check if the body we hit has a function called **take_damage**. We haven't created this yet, but we will do it soon. If so, deal the damage and destroy the bullet by calling the **destroy** function. This is a custom function we're going to create right after this one.

```
1  # called when we enter the collider of another body
2  func _on_Bullet_body_entered (body):
3
4      # does this body have a 'take_damage' function?
5      # if so, deal damage and destroy the bullet
6      if body.has_method("take_damage"):
7          body.take_damage(damage)
8          destroy()
```

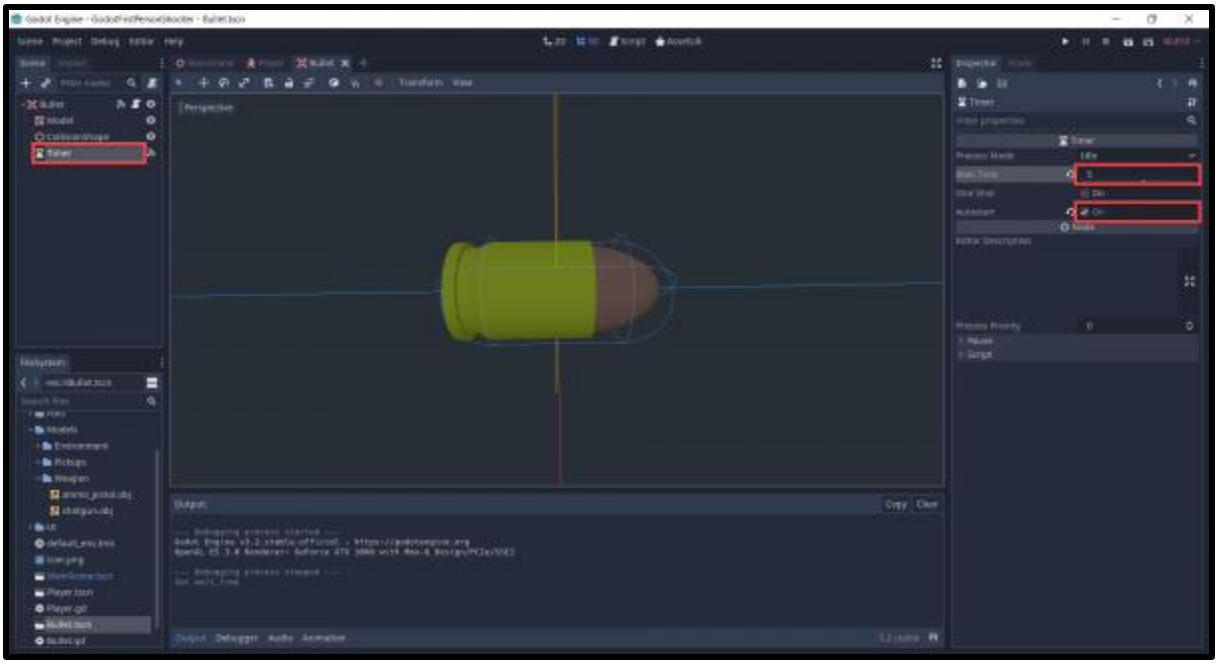
The **destroy** function will simply destroy the bullet instance.

```
1  # destroys the bullet
2  func destroy ():
3
4      queue_free()
```

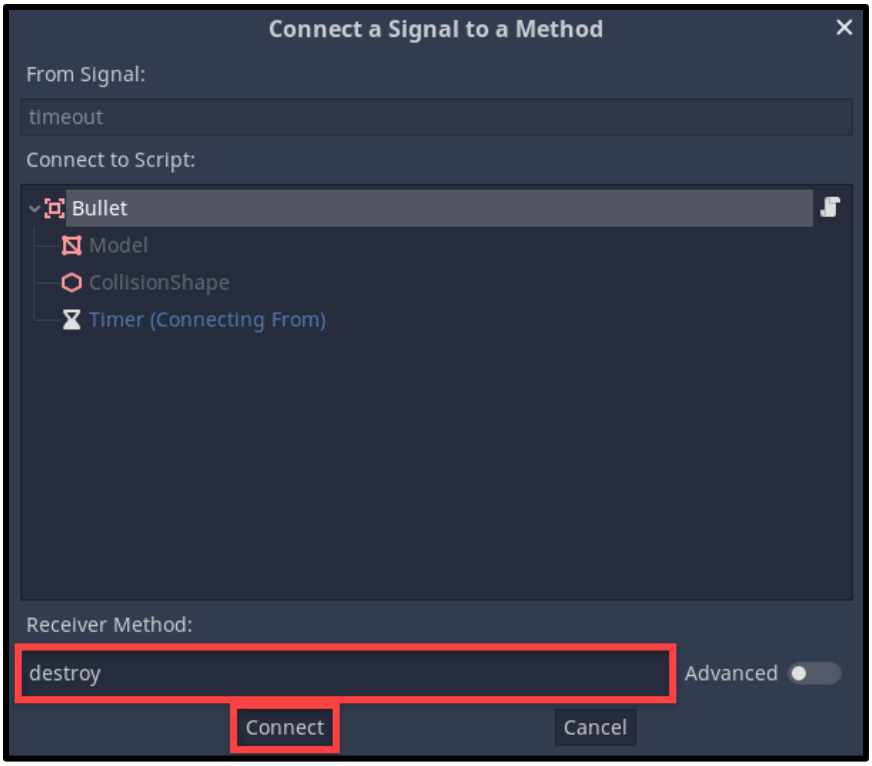
We now have a bullet which will move forward and damage whatever it hits (yet to be added). Let's also add in something so that the bullet can't travel forever if it misses its target.

In the **Bullet** scene, create a new node of type Timer. This node will count up and when it hits a certain amount, we'll call the destroy function.

1. Set the **Wait Time** to 5
2. Enable **Autostart**



Over in the node tab, let's connect the **timeout** signal to our script. This gets called once the wait time has been reached. Double click on the **timeout** signal and a window will pop-up. Change the **Receiver Method** to *destroy*. This will connect the signal to our existing function.



Shooting Bullets

Now that we have our bullet, let's implement the ability to shoot it. Over in the **Player** script, let's create some new variables to keep track of our components.

```
onready var muzzle = get_node("Camera/GunModel/Muzzle")
onready var bulletScene = preload("res://Bullet.tscn")
```

Down in the **_process** function, let's check for when the shoot action is pressed.

```
# check to see if we're shooting
if Input.is_action_just_pressed("shoot"):
    shoot()
```

The **shoot** function will spawn a new instance of the bullet scene and position it.

```
# called when we press the shoot button - spawn a new bullet
func shoot ():

    var bullet = bulletScene.instance()
    get_node("/root/MainScene").add_child(bullet)

    bullet.global_transform = muzzle.global_transform
    bullet.scale = Vector3.ONE

    ammo -= 1
```

If you want the mouse to be locked when we're playing the game, we can do so in the **_ready** function which gets called when the node is initialized.

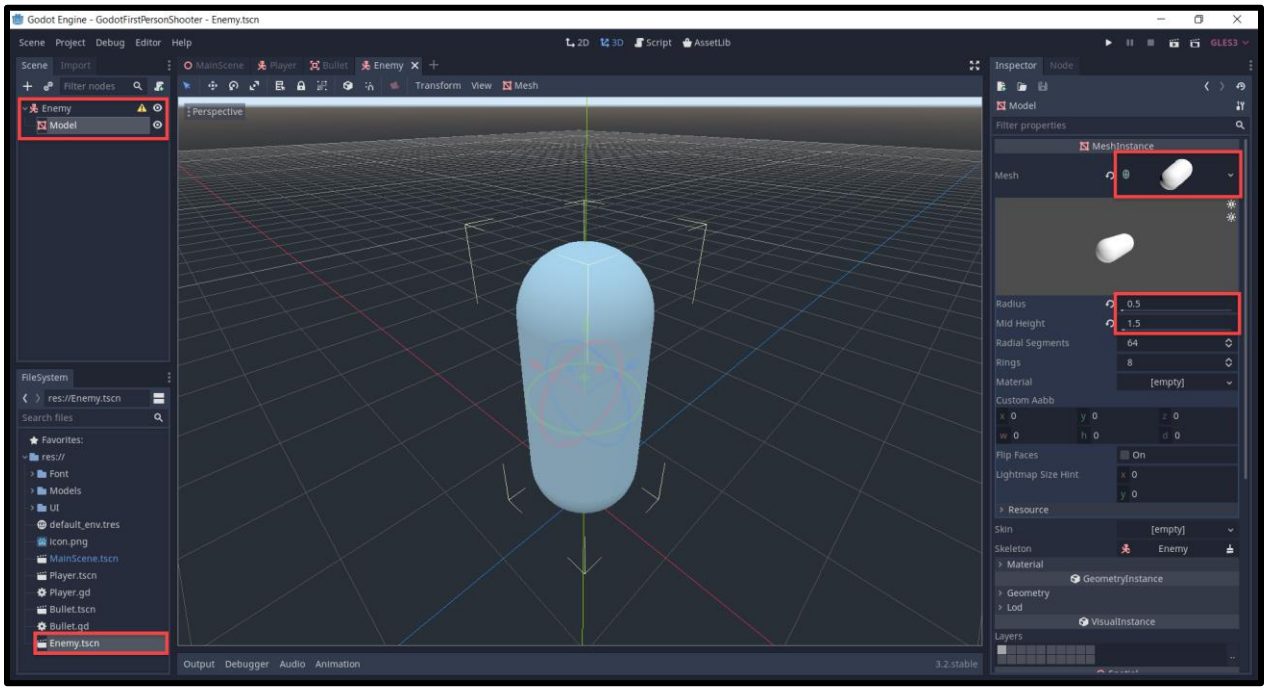
```gdscript
func _ready ():

    # hide and lock the mouse cursor
    Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
```

We can now press play and test it out.

Creating the Enemy

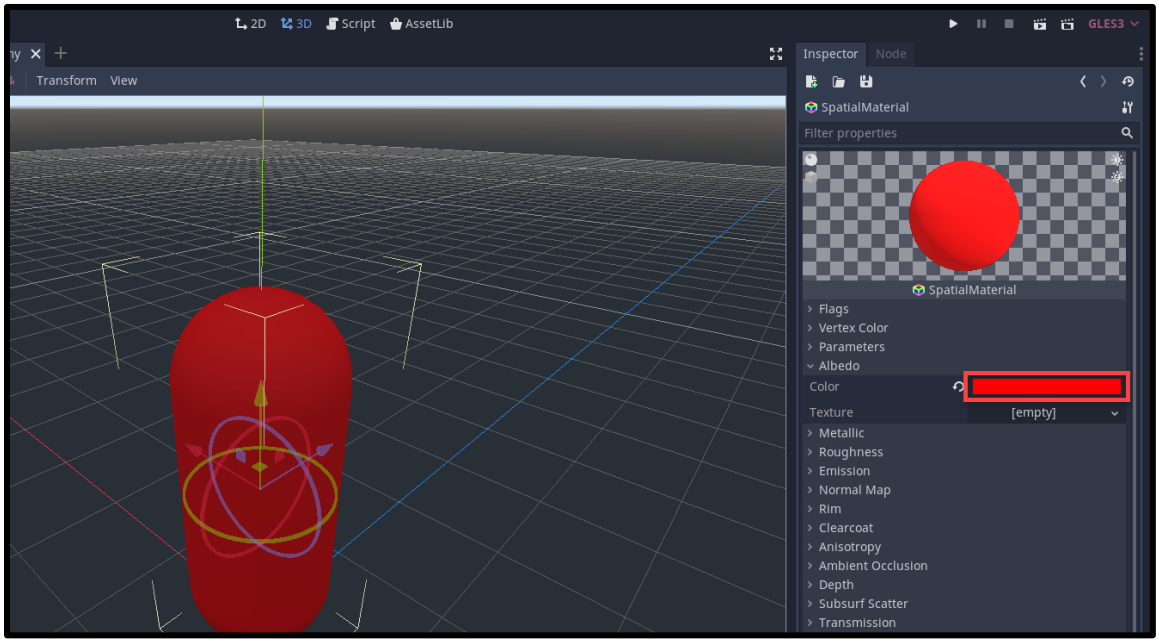
Create a new scene with a root node of **KinematicBody**.

1. Rename it to **Enemy**.
2. Save the scene.
3. Attach a new node of type **MeshInstance**.
4. Rename it to **Model**.
5. Set the **Mesh** to *capsule*.
6. Set the **Radius** to 0.5.
7. Set the **Mid Height** to 1.5.
8. Rotate and re-position the mesh so that it is orientated like below.

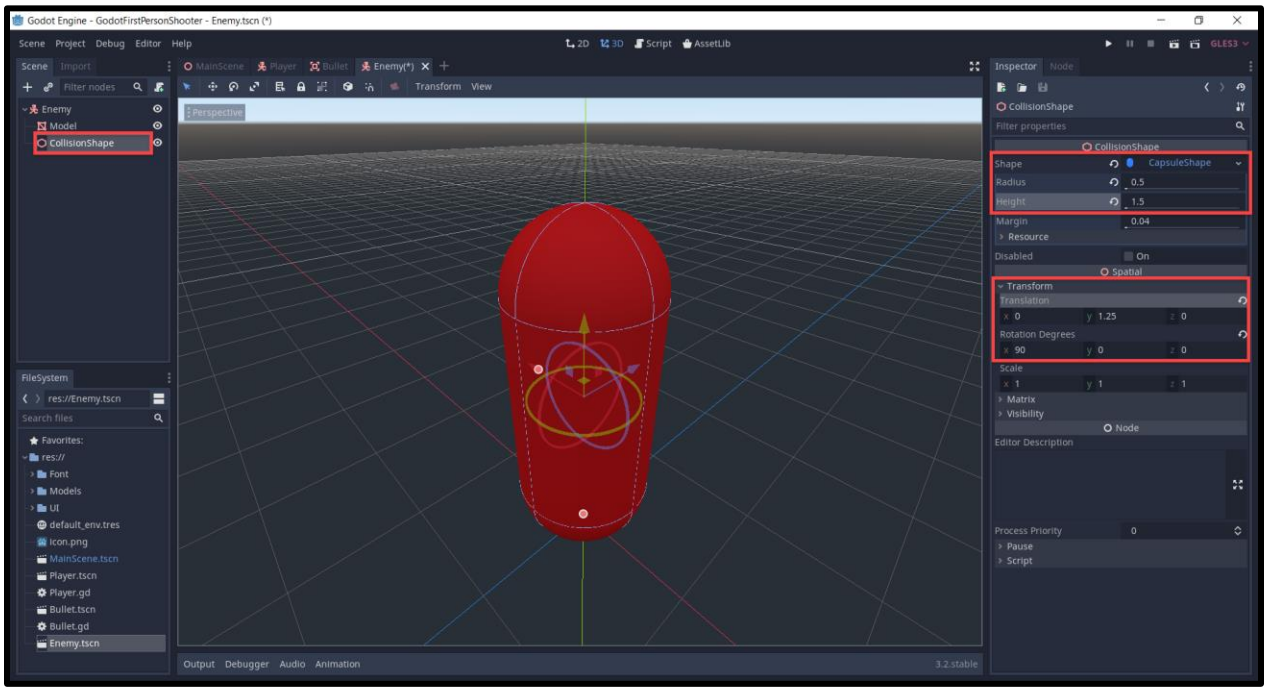


In the **MeshInstance** node, you'll see a **Material** property which is empty. Select it and click **New SpatialMaterial**.

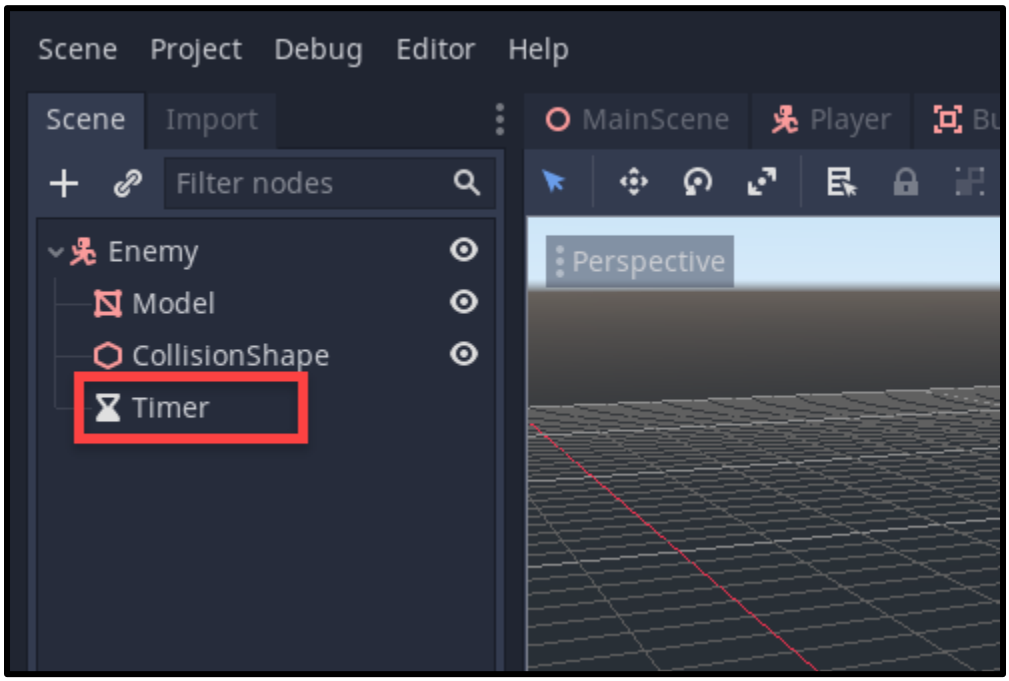
1. Set the **Albedo Color** to *red*.



Let's then attach a new **CollisionShape** node and make it match the mesh.



Finally, we'll attach a **Timer** node to the enemy. This is going to be used in order to attack the player.



Continued Part 2

And that's where we'll conclude for the time being!

At this point, we've learned several essential foundations for setting up a first-person shooter in Godot. First, we established our arena where all the action will take place. We then created a player, along with a camera, which is scripted to follow the mouse movements as we move the player around. To this player, we also added a gun model that can shoot bullets - which we also created with performance in mind. Lastly, we set up our enemy model and learned a bit about materials in the process.

Of course, there is still more needed to complete our FPS! In Part 2 of this tutorial, we will finish up building our Godot FPS by implementing the enemies' movement, setting up ammo and health pickups, and polishing everything off with a nifty UI! We hope to see you there.

Create a First-Person Shooter in Godot - Part 2

Introduction

Welcome back, and I hope you're ready to finish creating our Godot FPS tutorial.

In Part 1, we set up our arena, our player character, our FPS camera, our gun, our bullets, and even our red enemies. However, while we certainly implemented the shooting mechanics, our enemies can't yet damage players, get damaged themselves, or move. In addition, we have no pickups to speak of, let alone a UI to give the player essential health and ammo information. As such, in this tutorial, we will be jumping into setting those up.

By the end, you will have not only learned a lot about 3D game development, but also have a nifty FPS to add to your portfolio!

Project Files

For this project, we'll be using a handful of pre-made assets such as models and textures. Some of these are custom-built, while others are from kenney.nl, a website for public domain game assets.

The assets and source code can be downloaded here.

Scripting the Enemy

Create a new script on the **Enemy** node. Let's begin with the variables.

```gdscript
# stats
var health : int = 5
var moveSpeed : float = 1.0

# attacking
var damage : int = 1
var attackRate : float = 1.0
var attackDist : float = 2.0

var scoreToGive : int = 10

# components
onready var player : Node = get_node("/root/MainScene/Player")
onready var timer : Timer = get_node("Timer")
```

In the _**ready** function, we'll set up the timer to timeout every *attackRate* seconds.

```gdscript
func _ready ():

    # setup the timer
    timer.set_wait_time(attackRate)
    timer.start()
```

If we select the **Timer** node, we can connect the **timeout** signal to the script. This will create the _**on_Timer_timeout** function. We'll be working on this later on.

```gdscript
func _on_Timer_timeout ():
    pass
```

In the _**physics_process** function, we'll move towards the player.

```gdscript
func _physics_process ():

    # calculate direction to the player
    var dir = (player.translation - translation).normalized()
    dir.y = 0

    # move the enemy towards the player
    move_and_slide(dir * moveSpeed, Vector3.UP)
```

The **take_damage** function gets called when we get damaged by the player's bullets.

```gdscript
# called when we get damaged by the player
func take_damage (damage):

    health -= damage

    # if we've ran out of health - die
    if health <= 0:
        die()
```

The **die** function gets called when our health reaches 0. The **add_score** function for the player will be added soon.

```gdscript
# called when our health reaches 0
func die ():

    player.add_score(scoreToGive)
    queue_free()
```

The last function to add is the **Attack** function. We'll be creating the player's **take_damage** function soon.

```
# deals damage to the player
func attack ():

    player.take_damage(damage)
```

Finally in the **_on_Timer_timeout** function, we can check the distance to the player and try to attack them.

```
# called every 'attackRate' seconds
func _on_Timer_timeout ():

    # if we're at the right distance, attack the player
    if translation.distance_to(player.translation) <= attackDist:
        attack()
```

Player Functions

In the **Player** script, we're going to add in a number of functions which we need right now and in the future. The **die** function will be filled in later once we have our UI setup.

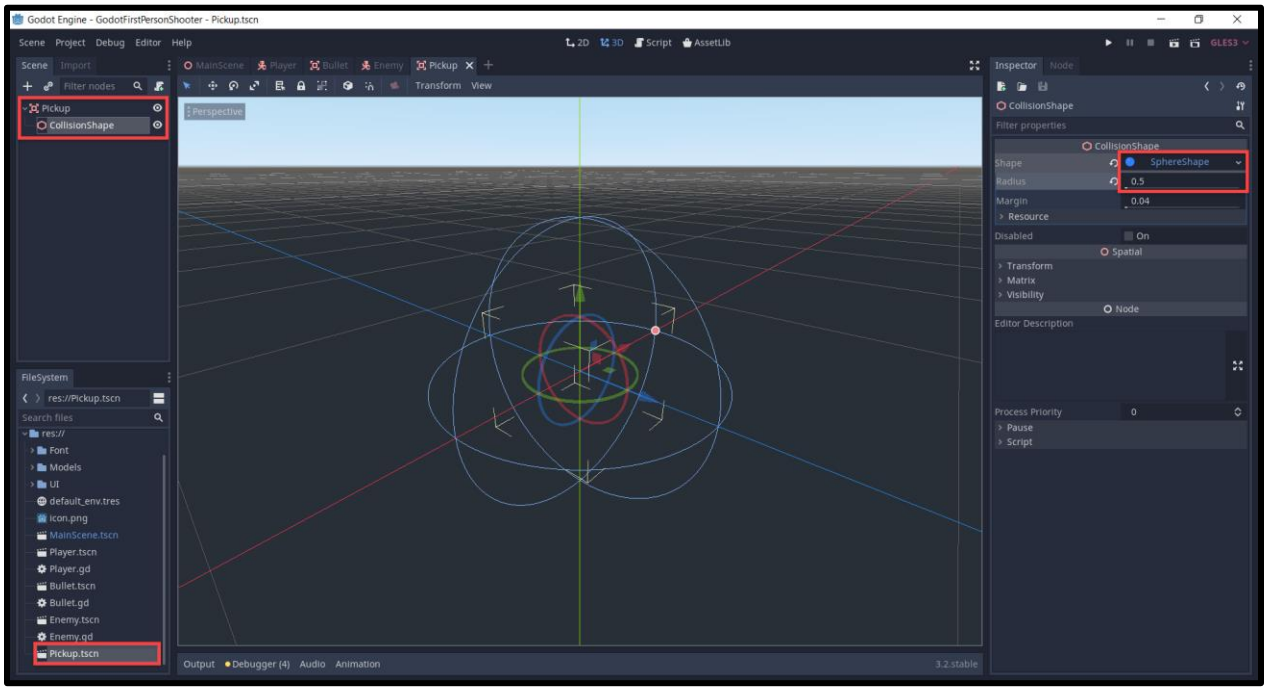
```gdscript
# called when an enemy damages us
func take_damage (damage):

    curHp -= damage

    if curHp <= 0:
        die()

# called when our health reaches 0
func die ():

    pass

# called when we kill an enemy
func add_score (amount):

    score += amount

# adds an amount of health to the player
func add_health (amount):

    curHp = clamp(curHp + amount, 0, maxHp)

# adds an amount of ammo to the player
func add_ammo (amount):

    ammo += amount
```

Now we can go to the **MainScene** and drag the **Enemy** scene into the scene window to create a new instance of the enemy. Press play and test it out.

Pickups

For our pickups, we're going to create one template scene which the health and ammo pack will inherit from. Create a new scene with a root node of **Area**.

2. Rename it to **Pickup**.
3. Save the scene.
4. Attach a child node of type **CollisionShape**.
5. Set the **Shape** to Sphere.
6. Set the **Radius** to 0.5.



Next, create a script on the **Pickup** node. First, we'll create an *enumerator* which is a custom data type that contains different options.

```
1  enum PickupType {
2      Health,
3      Ammo
4  }
```

Then for our variables.

```
# stats
export(PickupType) var type = PickupType.Health
export var amount : int = 10

# bobbing
onready var startYPos : float = translation.y
var bobHeight : float = 1.0
var bobSpeed : float = 1.0
var bobbingUp : bool = true
```

In the _**process** function, we're going to make the pickup bob up and down.

```
func _process (delta):

    # move us up or down
    translation.y += (bobSpeed if bobbingUp else -bobSpeed) * delta

    # if we're at the top, start moving downwards
    if bobbingUp and translation.y > startYPos + bobHeight:
        bobbingUp = false
    # if we're at the bottom, start moving up
    elif !bobbingUp and translation.y < startYPos:
        bobbingUp = true
```

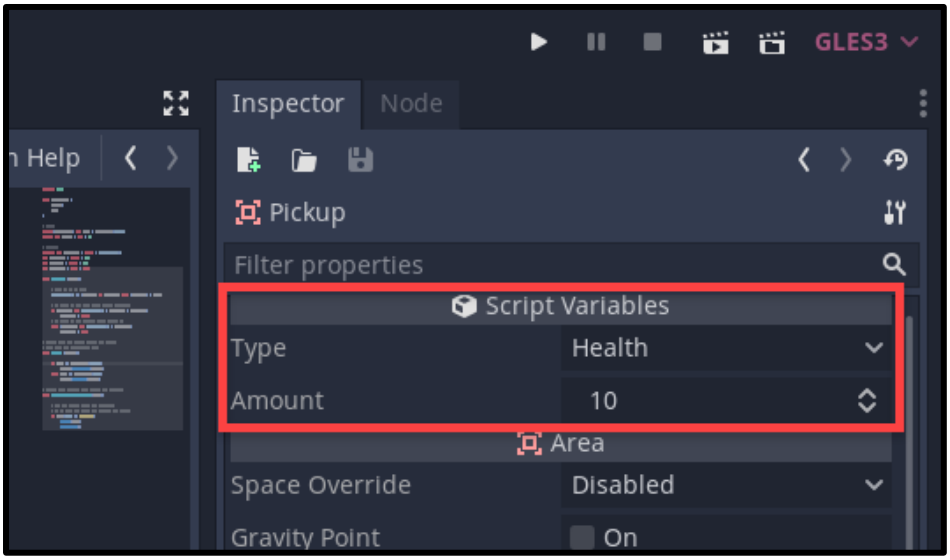
Select the **Pickup** node and connect the **body_entered** node to the script.

```
# called when another body enters our collider
func _on_Pickup_body_entered (body):

    # did the player enter our collider?
    # if so give the stats and destroy the pickup
    if body.name == "Player":
        pickup(body)
        queue_free()
```

The **pickup** function will give the player the appropriate stat increase.

```
# called when the player enters the pickup
# give them the appropriate stat
func pickup (player):

    if type == PickupType.Health:
        player.add_health(amount)
    elif type == PickupType.Ammo:
        player.add_ammo(amount)
```

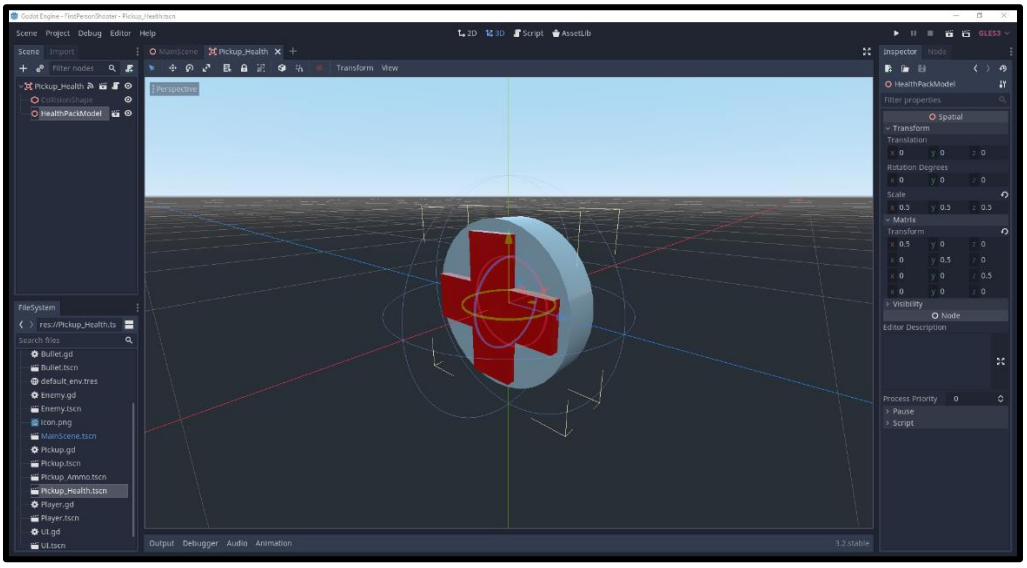
Now that we've finished the script, let's go back to the scene and you'll see that the **Pickup** node now has two exposed variables.



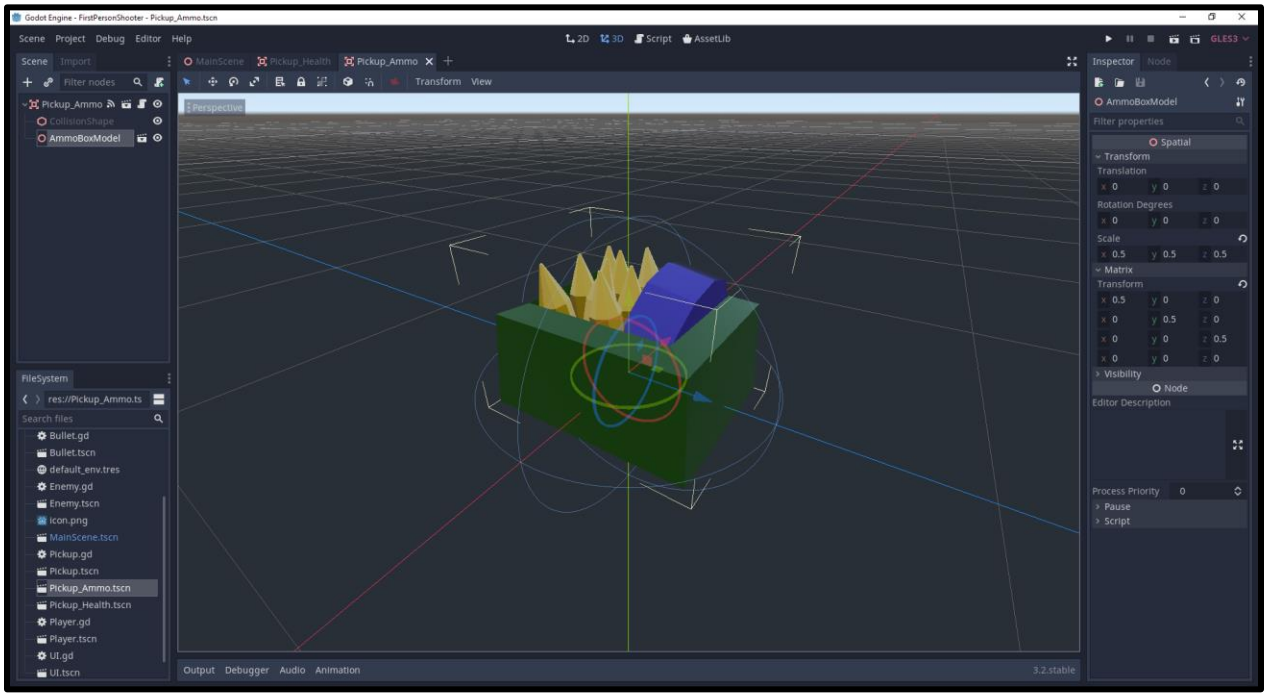
We're now going to create two inherited scenes from this original **Pickup** one. Go *to Scene > New Inherited Scene*... and a window will pop up asking to select a base scene. Select the **Pickup.tscn** and a new scene should be created for you. You'll see that there's already the area and collider nodes there since they are a parent. This means any changes to the original **Pickup** scene, those changes will also be applied to the inherited scenes.

All we need to do here is...

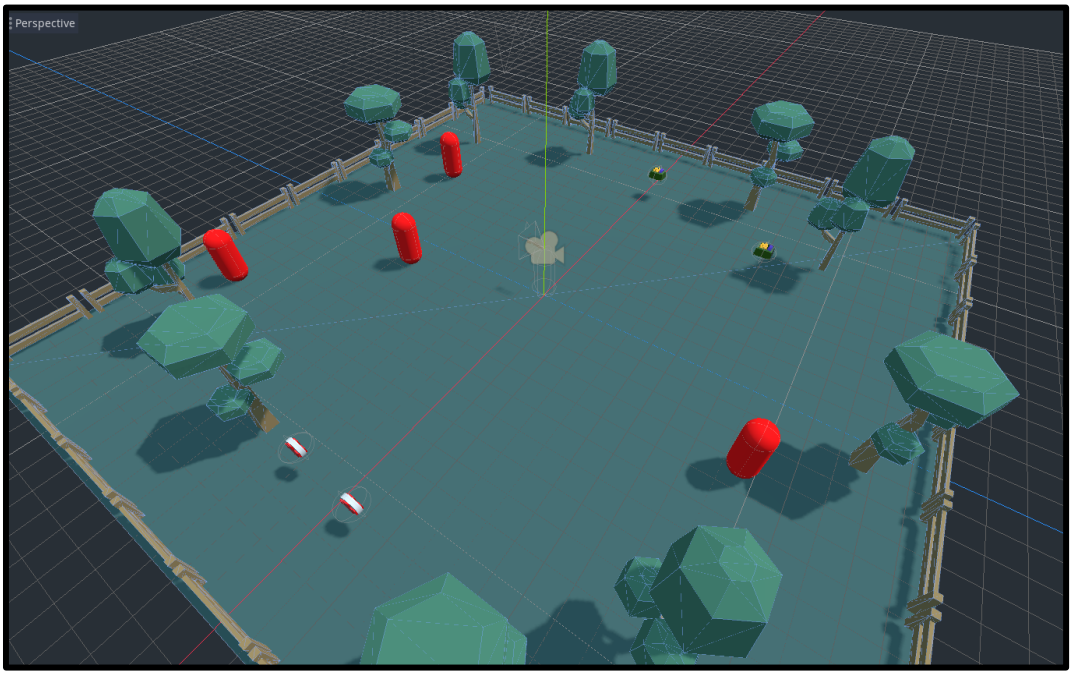
- Rename the area node to **Pickup_Health**
- Set the pickup type to **Health**
- Drag in the health pack model



We also want to do the same for the ammo pickup.



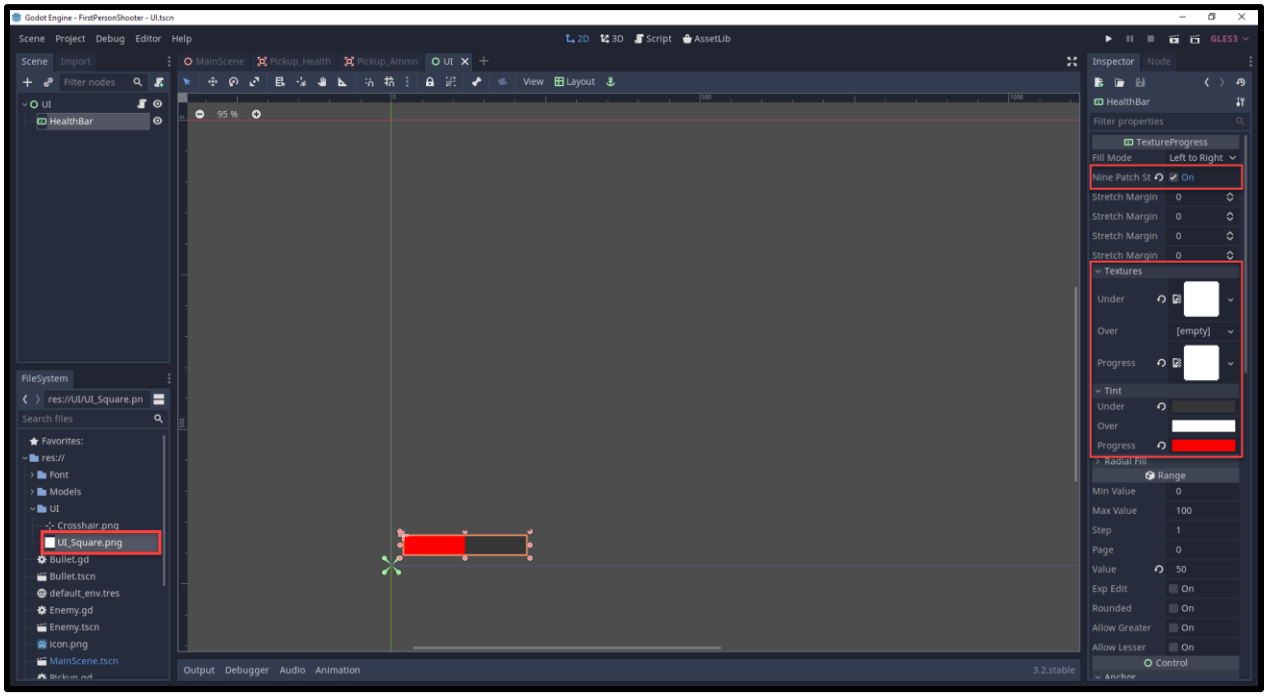
Back in the **MainScene**, we can drag in the Enemy, Pickup_Health and Pickup_Ammo scenes and place them around.



UI

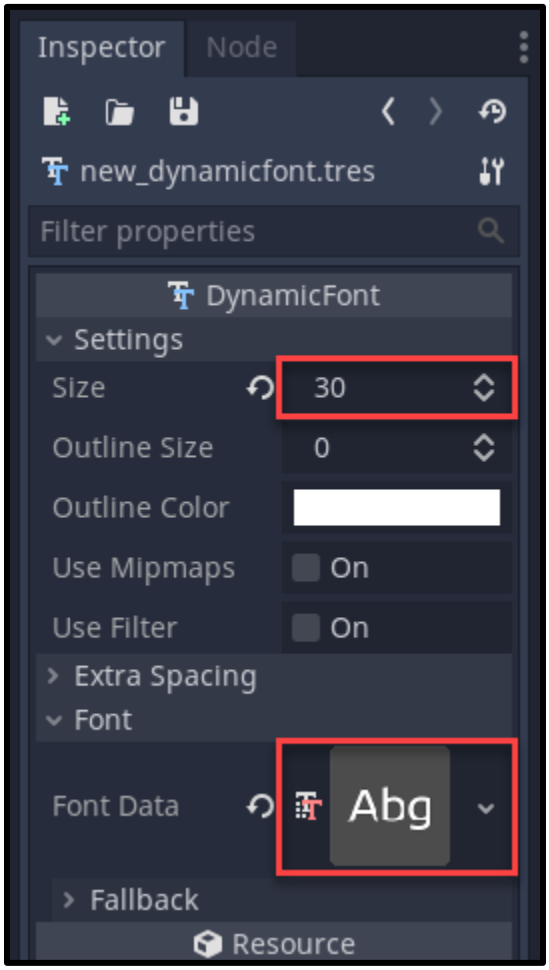
Now it's time to create the UI which will display our health, ammo, and score. Create a new scene with the root node being **User Interface** (control node).

- Rename the node to **UI**
- Create a new child node of type **TextureProgress**
- Enable **Nine Patch Stretch**
- Rename it to **HealthBar**
- Move the health bar to the bottom left of the screen and re-size it
- Drag the 4 anchor points (green pins) to the bottom left of the screen
- Set the **Under** and **Progress** textures to the UI_Square.png image
- Set the **Tints** as seen in the image.



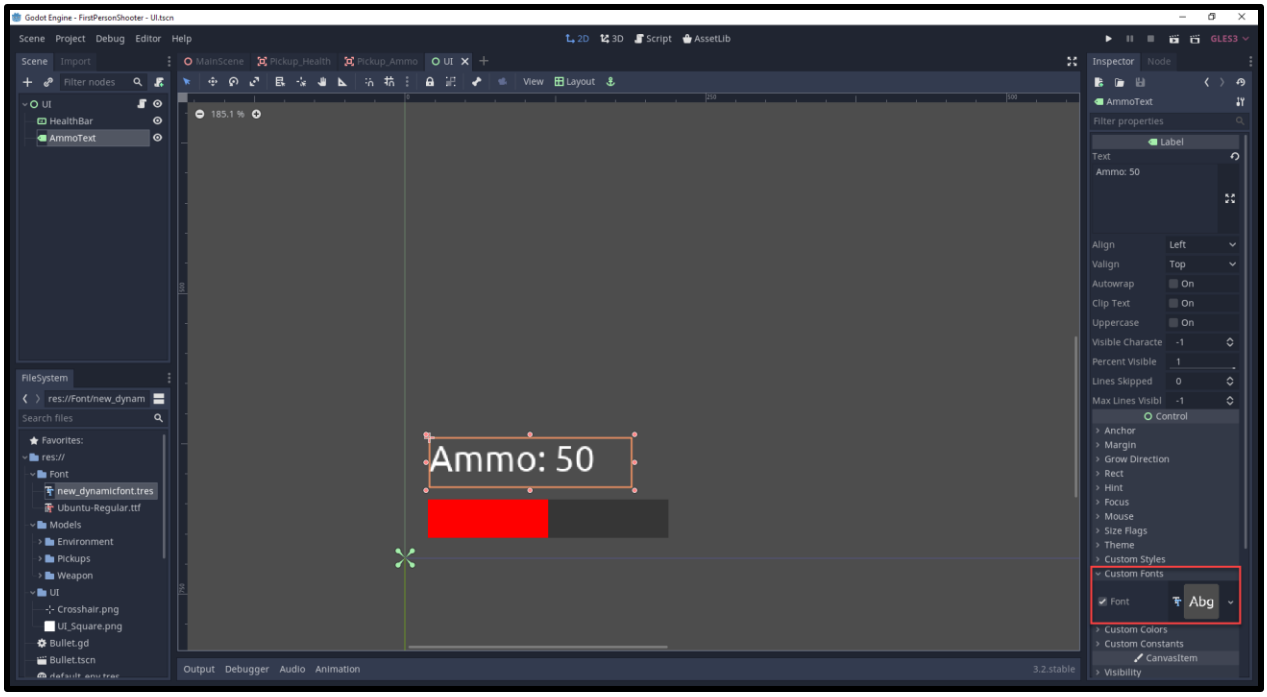
For the text, we need to create a new dynamic font resource. In the file system, find the **Ubuntu-Regular.ttf** file - right click it and select **New Resource**...

- Search for and create a **DynamicFont**
- In the inspector, set the **Font Data** to the ubuntu font file
- Set the **Size** to 30



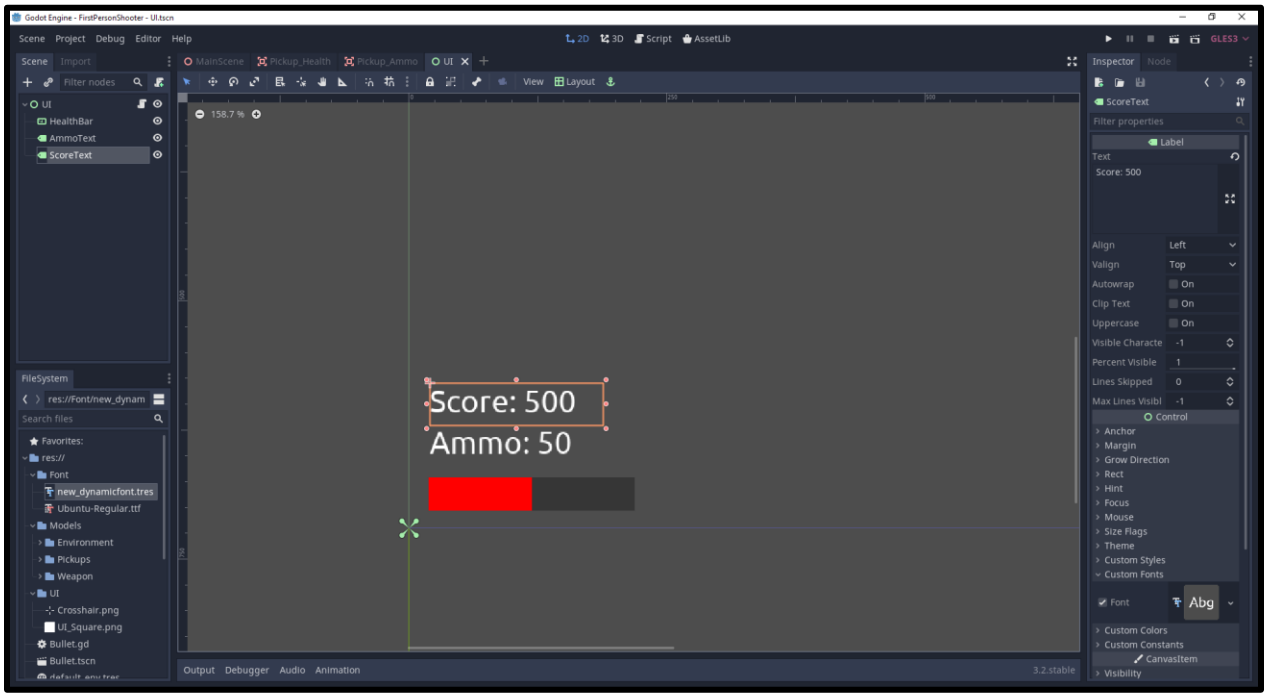
Now we can create the text elements. Create a new **Label** node and call it **AmmoText**.

- Resize and position it like in the image below
- Set the **Custom Font** to the new dynamic font file
- Move the anchor points down to the bottom left



With the node selected, press **Ctrl + D** to duplicate the text.

- Rename it to **ScoreText**
- Move it above the ammo text



Scripting the UI

Now that we have our UI elements, let's create a new script attached to the UI node called **UI**.

First, we can create our variables.

```
onready var healthBar : TextureProgress = get_node("HealthBar")
onready var ammoText : Label = get_node("AmmoText")
onready var scoreText : Label = get_node("ScoreText")
```

Then we're going to have three functions which will each update their respective UI element.

```
func update_health_bar (curHp, maxHp):

    healthBar.max_value = maxHp
    healthBar.value = curHp

func update_ammo_text (ammo):

    ammoText.text = "Ammo: " + str(ammo)

func update_score_text (score):

    scoreText.text = "Score: " + str(score)
```

So we got the functions to update the UI nodes. Let's now connect this to the **Player** script. We'll start by creating a variable to reference the UI node.

```
onready var ui : Node = get_node("/root/MainScene/CanvasLayer/UI")
```

Then in the _**ready** function, we can initialize the UI.

```
1  func _ready ():
2
3      # set the UI
4      ui.update_health_bar(curHp, maxHp)
5      ui.update_ammo_text(ammo)
6      ui.update_score_text(score)
```

We want to update the ammo text in both the **shoot** and **add_ammo** functions.

```
1  ui.update_ammo_text(ammo)
```

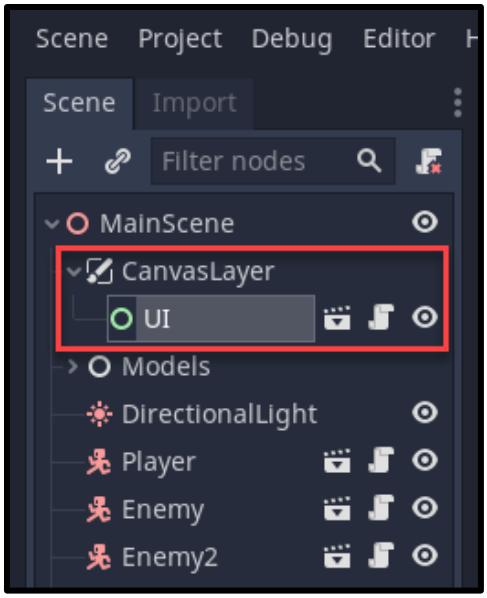
We want to update the health bar in both the **take_damage** and **add_health** functions.

```
1  ui.update_health_bar(curHp, maxHp)
```

We want to update the score text in the **add_score** function.

```
1  ui.update_score_text(score)
```

And now we can go back to the **MainScene** and create a new node called **CanvasLayer**. Whatever is a child of this, gets rendered to the screen so let's now drag in the **UI** scene as a child of this node.



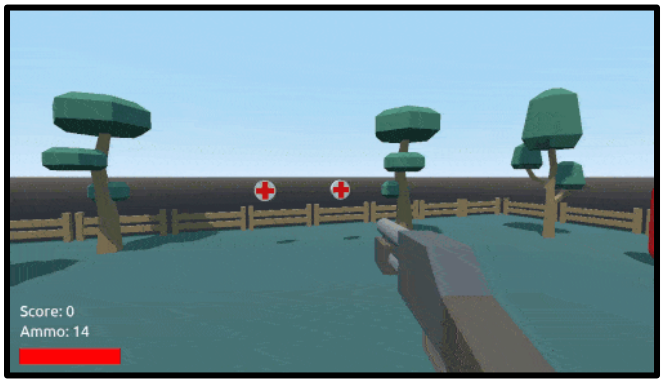
Now we can press play and see that the UI is on-screen and updates when we take damage, collect pickups, and kill enemies.

Conclusion

Congratulations on completing the tutorial!

If you've been following along, you should now have a complete Godot FPS game at your fingertips. Players will be able to fire on enemies, gather pickups for ammo and health, and even potentially be defeated by our menacing red capsules! Not only that, but you also have boosted your own knowledge in how Godot's 3D engine works and how you can utilize 3D's unique features in a number of ways. Of course, from here, you can expand upon the FPS game we created in any way you please - adding in new systems, models, sound, etc.

Thanks for following along, and I hope to see you in the next Godot tutorial.



Build a 2D RPG in Godot - Part 1

Introduction

Welcome, everyone! Even after all the years games have existed, RPGs are still one of the most popular genres that aspiring developers want to learn how to make. However, given how feature-rich RPGs can be, it can be hard to know where to start. Thankfully, though, with Godot and some programming knowledge at hand, you will have all the tools you need to start on your very own dream RPG!

In this RPG tutorial series, we'll be creating a 2D RPG inside of the Godot game engine together. It will feature a top-down player controller, enemies, combat, loot, and even a leveling system. For Part 1, though, we will be covering a number of new systems inside of Godot as we start to build our RPG, including tilemaps, sprite animations, and raycasting. This course will require a basic understanding of the Godot game engine and the GDScript scripting language. If you're new to Godot, you can view our introductory tutorial here.

Otherwise, though, we hope you strap in and are ready to learn how to make 2D RPGs with Godot.

If you'd like to jump straight in making enemies and loot, check out Part 2 instead!

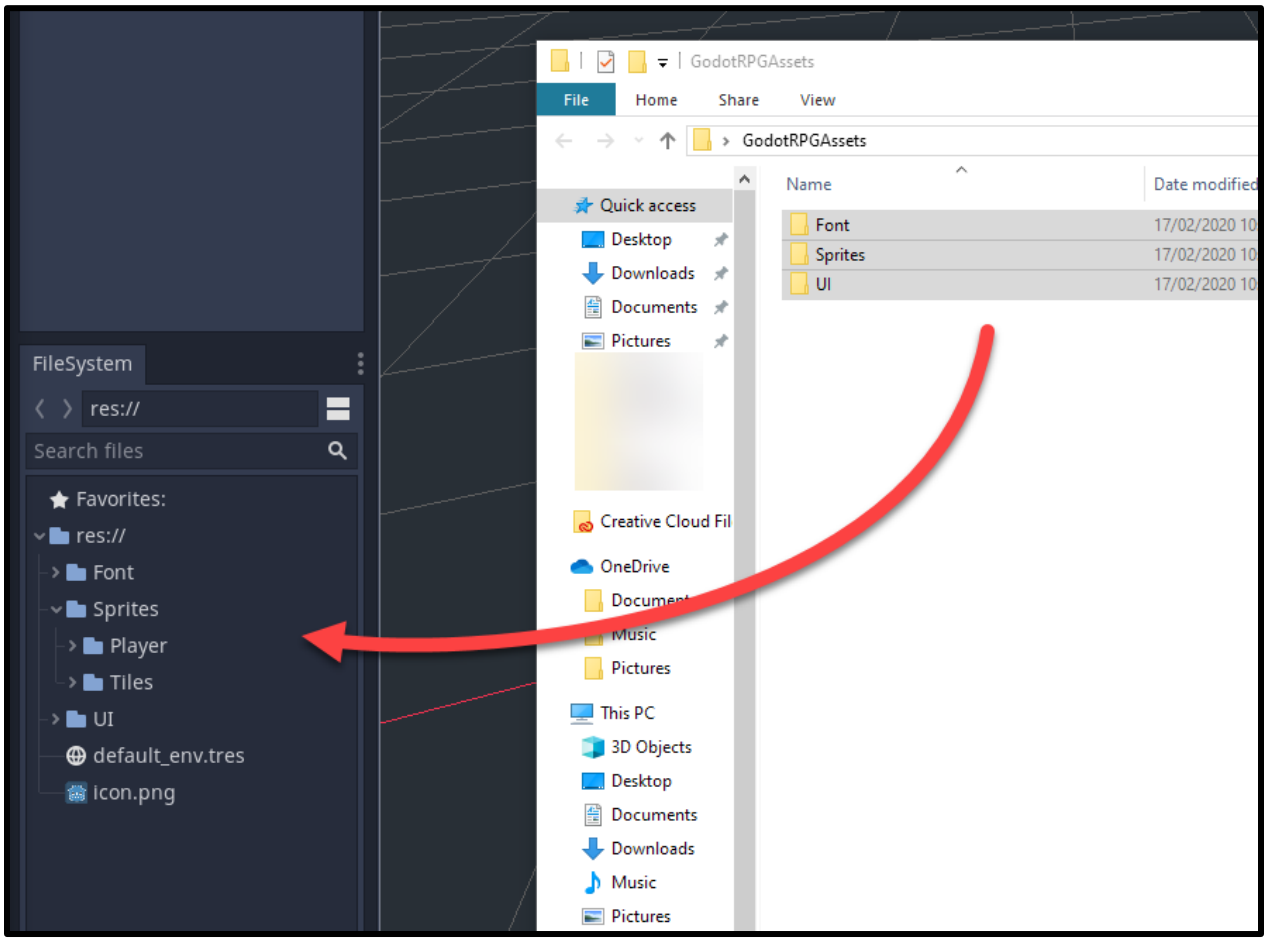
Project Files

For this project, we'll be needing some assets such as sprites and a font. These will be sourced from kenney.nl and Google Fonts. You can, of course, choose to use your own assets, but for this course we'll be using these.

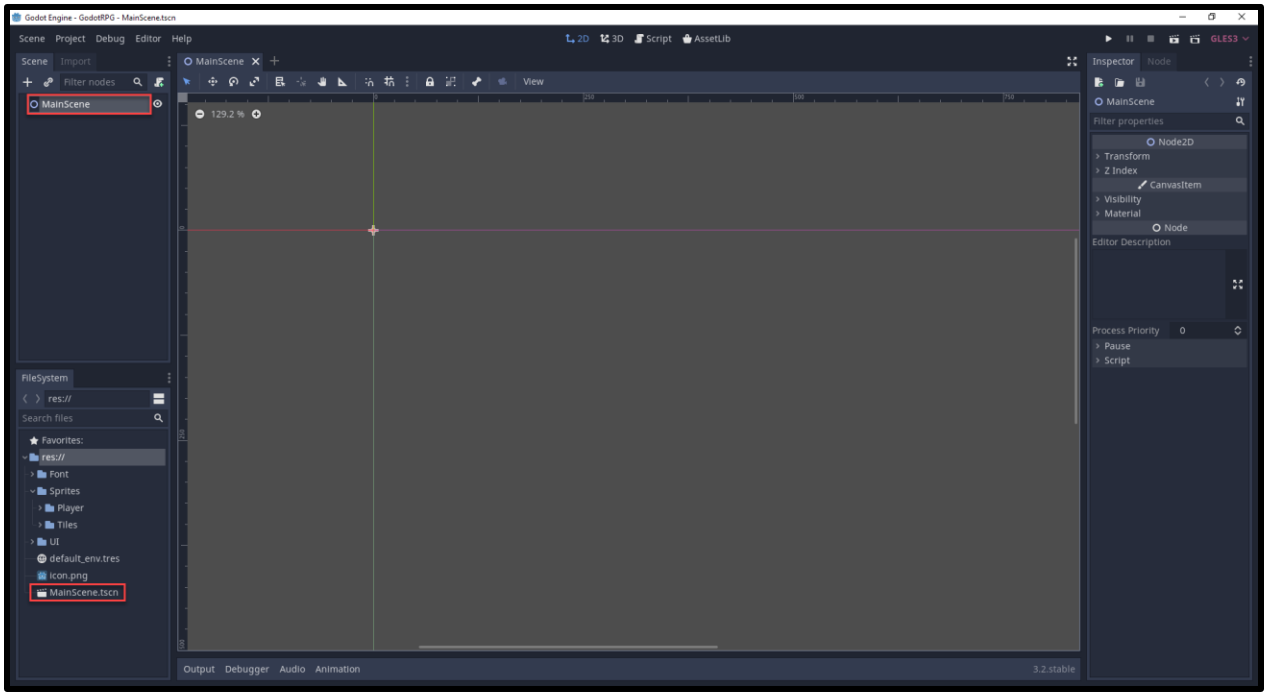
The assets and source code can be downloaded here.

Setting Up The Project

To begin, let's create a new Godot project. Inside of the editor, let's then import the assets we'll be needing.



Once we have the assets, let's create our first scene by selecting **2D Scene** in the scene panel. Rename this node to **MainScene**, then save the scene to the file system.



Creating the Player

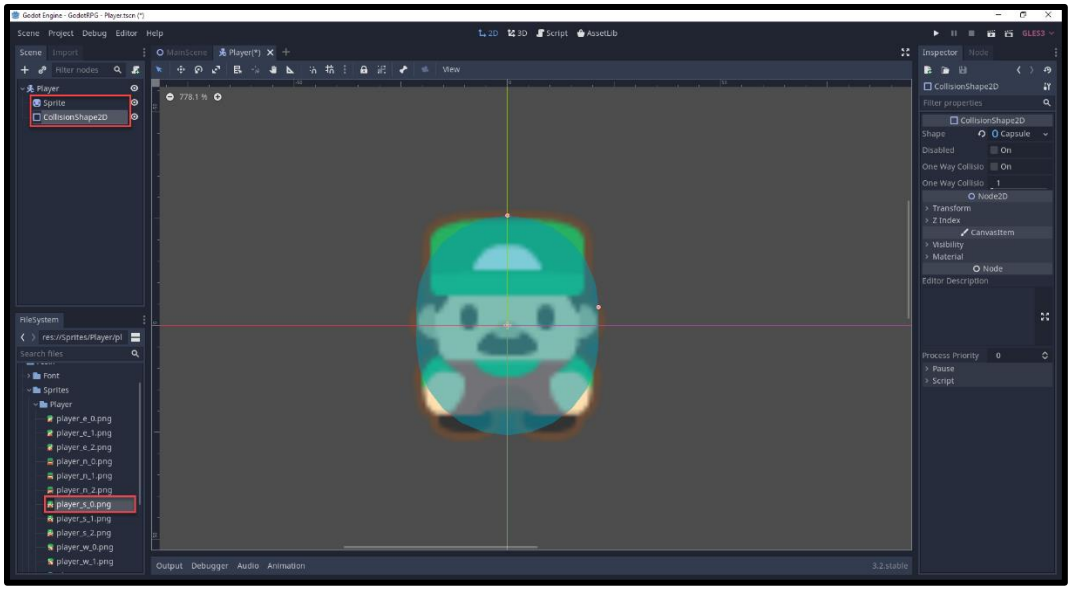
Now that we have the main scene, let's create a new scene (Scene > New Scene) of type **KinematicBody2D**.

- Rename the node to **Player**
- Save the scene to the file system

As a child of this node, we want to...

- Drag in the **player_s_0.png** sprite in to create a new sprite node
- Set the position to 0, 0
- Create a child node of type **CollisionShape2D**
- Set the **Shape** to Capsule
- Resize it to fit the sprite

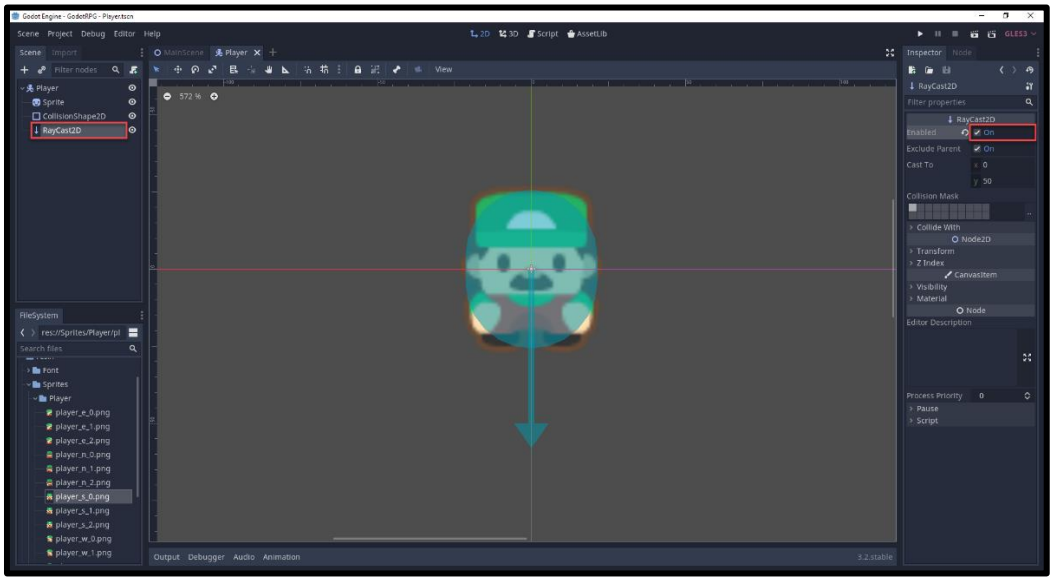
Right now, we're using just a sprite node to allow us to see the player. Later on, we'll be replacing this with an **AnimatedSprite** node.



Finally, for our player, let's create a **RayCast2D** node. You'll see that in the scene view, an arrow will appear to come out of the player. This is a raycast, which is a concept in game development that is used in many different cases. You give a raycast an origin position and a direction. Then it will shoot a point from that origin in the given direction. If that point ever hits a collider it will return that object and some other information.

In our case, the raycast has a max distance and this will be used to detect interactable objects and enemies.

- Make sure to enable **Enabled**



Scripting the Player Movement

Now that we have the player scene, let's create a new script attached to the kinematic body node called **Player**. We can start with our variables.

```
1  var curHp : int = 10
2  var maxHp : int = 10
3  var moveSpeed : int = 250
4  var damage : int = 1
5
6  var gold : int = 0
7
8  var curLevel : int = 0
9  var curXp : int = 0
10 var xpToNextLevel : int = 50
11 var xpToLevelIncreaseRate : float = 1.2
12
13 var interactDist : int = 70
14
15 var vel = Vector2()
16 var facingDir = Vector2()
17
18 onready var rayCast = $RayCast2D
```

Later on, we'll be adding in a couple more variables, but for most of our systems, this will do. Now, in order to move the player around, we'll need to know which buttons do what. Go to the **Project Settings** window (*Project > Project Settings*...) and click on the **Input Map** tab. Here, we want to create 5 new actions and assign a keyboard key to each of them.

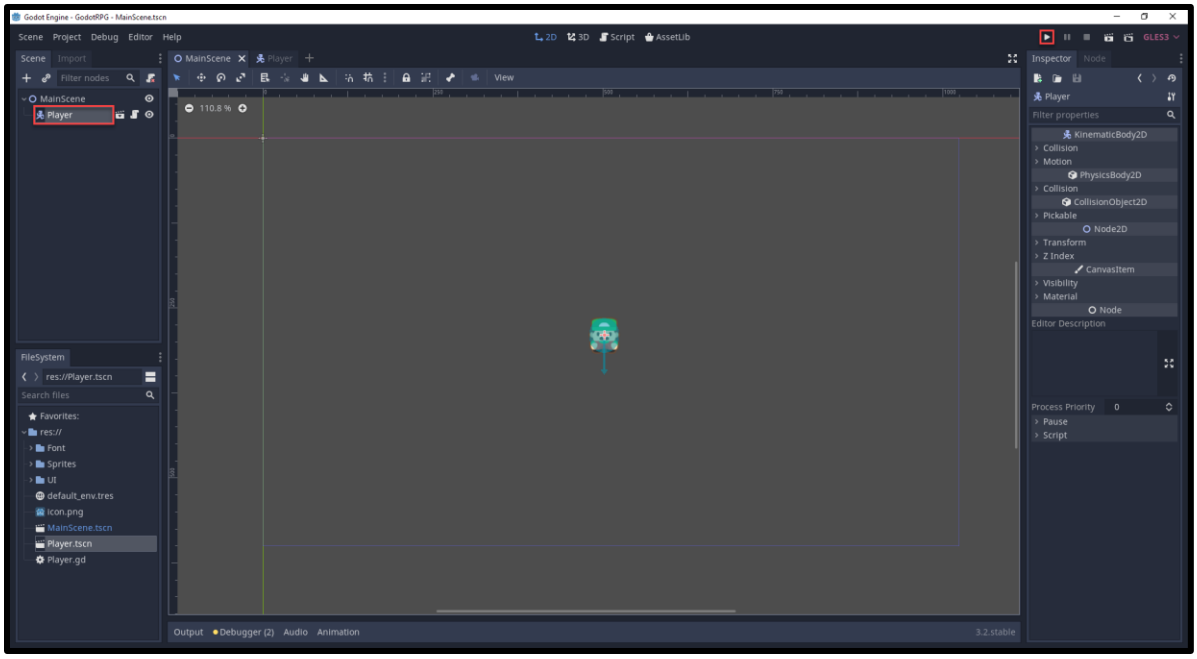
- move_left - **left arrow key**
- move_right - **right arrow key**
- move_up - **up arrow key**
- move_down - **down arrow key**
- interact – **space**

Back in the script, we can create the **_physics_process** function. A built-in function which runs at 60 FPS (good for physics). What we want to do here, is a few things.

1. Reset **vel** (vector containing our velocity)
2. Detect the 4 direction keys to change the velocity and facing direction (used for animations later)
3. Normalize the velocity vector to prevent faster diagonal movement
4. Move the player based on the velocity using the KinematicBody2D node function

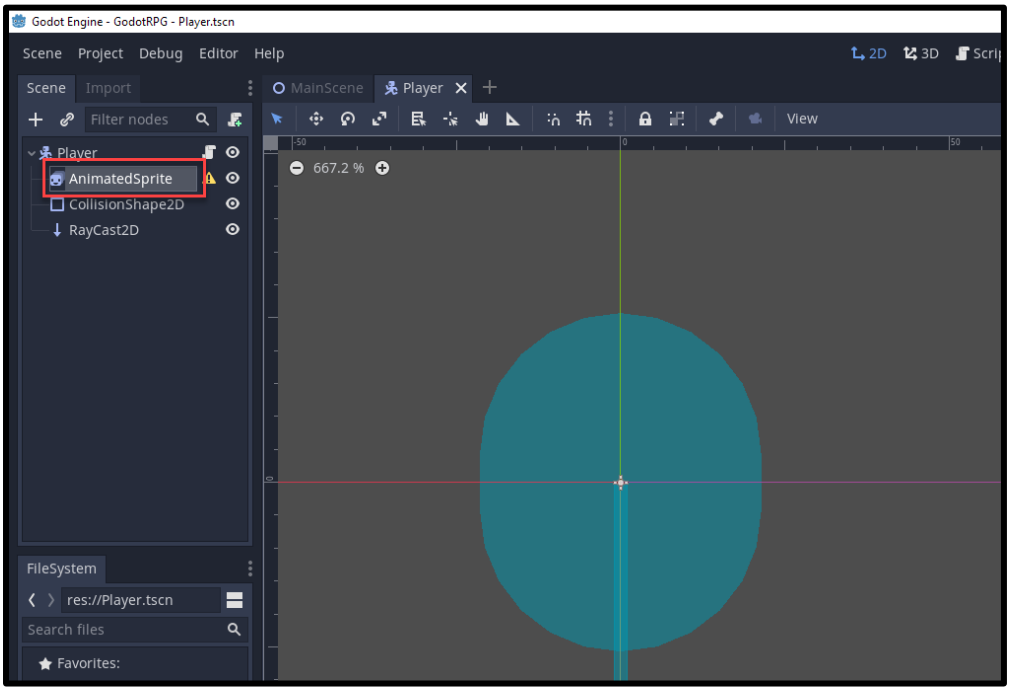
```gdscript
func _physics_process (delta):

    vel = Vector2()

    # inputs
    if Input.is_action_pressed("move_up"):
        vel.y -= 1
        facingDir = Vector2(0, -1)
    if Input.is_action_pressed("move_down"):
        vel.y += 1
        facingDir = Vector2(0, 1)
    if Input.is_action_pressed("move_left"):
        vel.x -= 1
        facingDir = Vector2(-1, 0)
    if Input.is_action_pressed("move_right"):
        vel.x += 1
        facingDir = Vector2(1, 0)

    # normalize the velocity to prevent faster diagonal movement
    vel = vel.normalized()

    # move the player
    move_and_slide(vel * moveSpeed, Vector2.ZERO)
```

Let's hop over to the **MainScene** and test the movement out. In the file system, drag the player scene into the scene window to create a new instance of it. Then click the **Play** button, choose the main scene to be the base scene and see if the movement works.

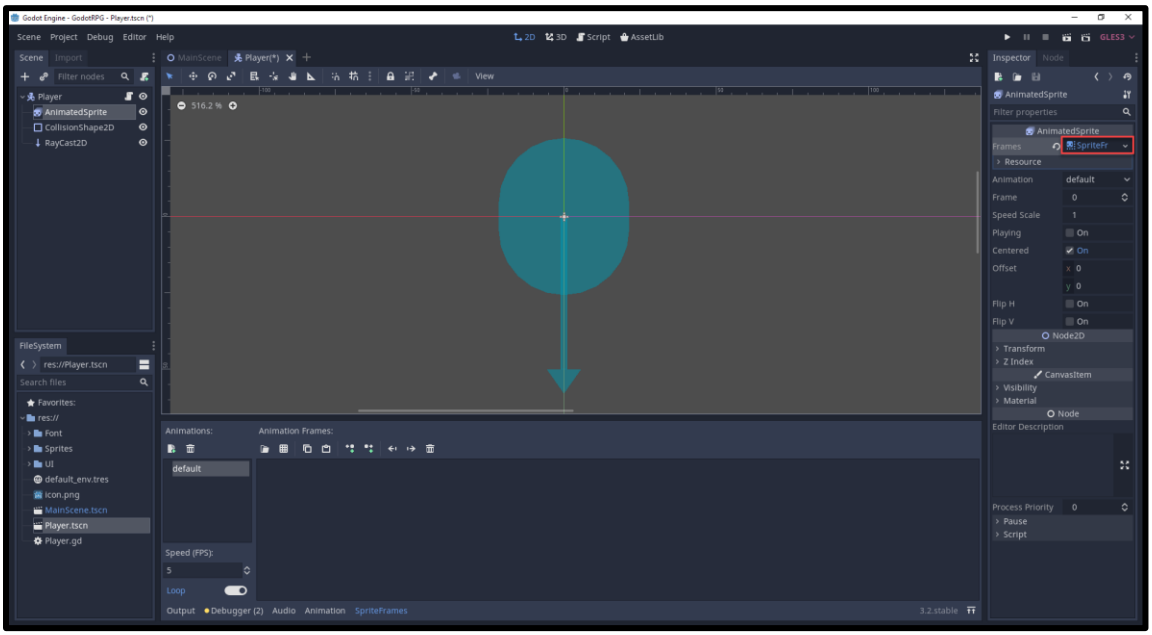


Animating the Player

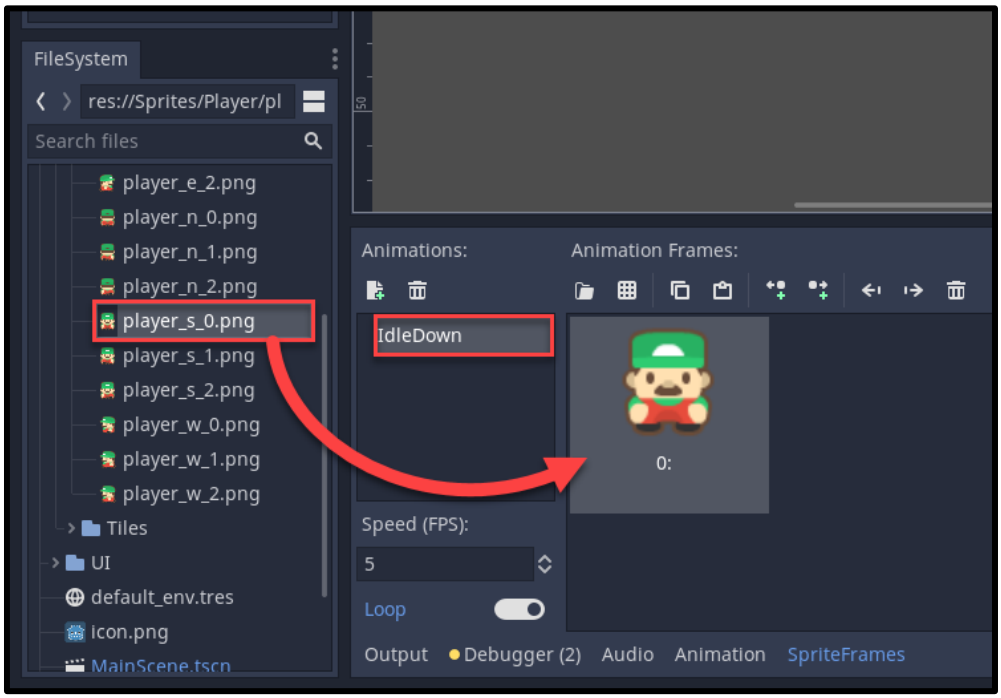
You'll see that we can move around, but it's pretty bland. Let's now implement sprite animations. In the **Player** scene, delete the **Sprite** node and replace it with an **AnimatedSprite** node.



In the inspector, create a new sprite frames resource. The **SpriteFrames** window will then pop-up.



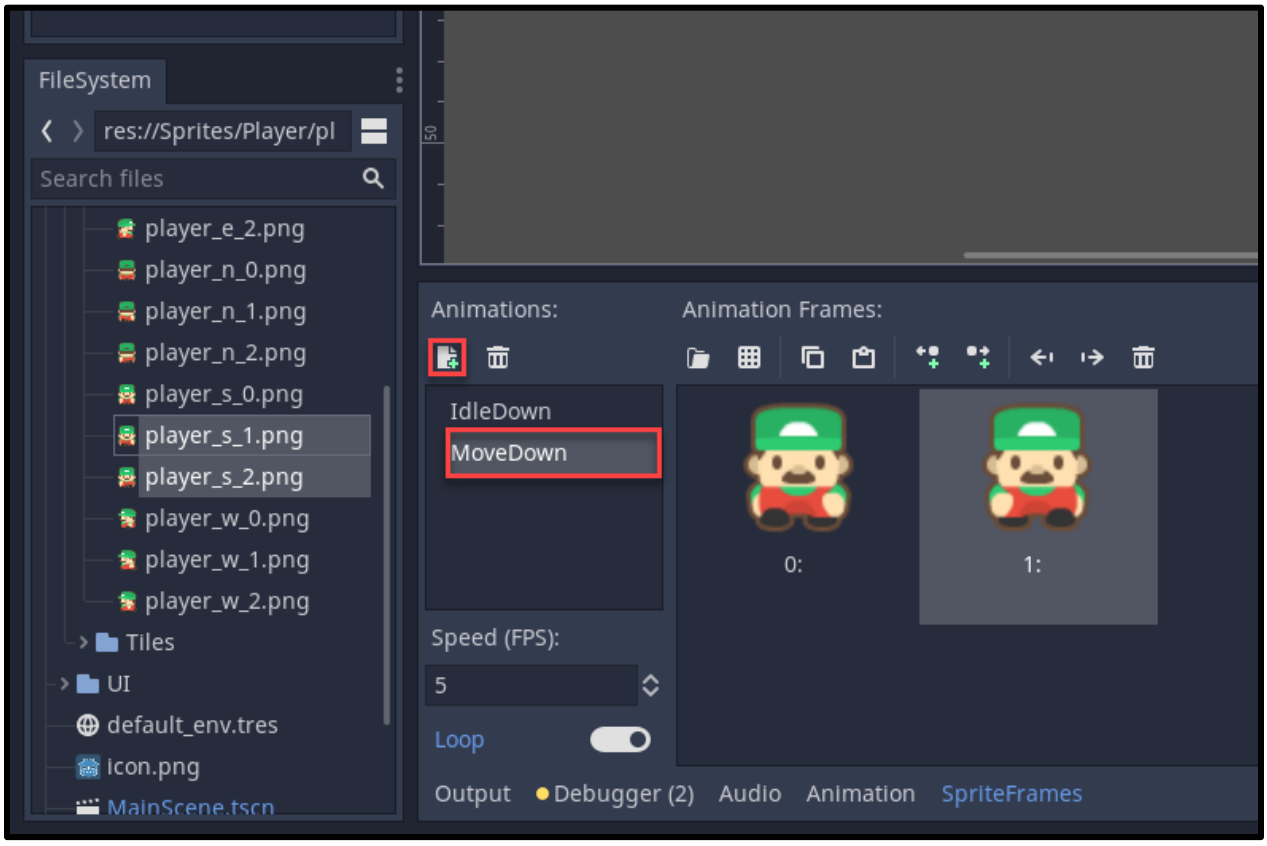
In the animations list there will be an animation called **default**. Double click on it and rename it to **IdleDown**. For this animation, we're going to drag in the **player_s_0** sprite.



From here, we can create the rest of the animations. These will be:

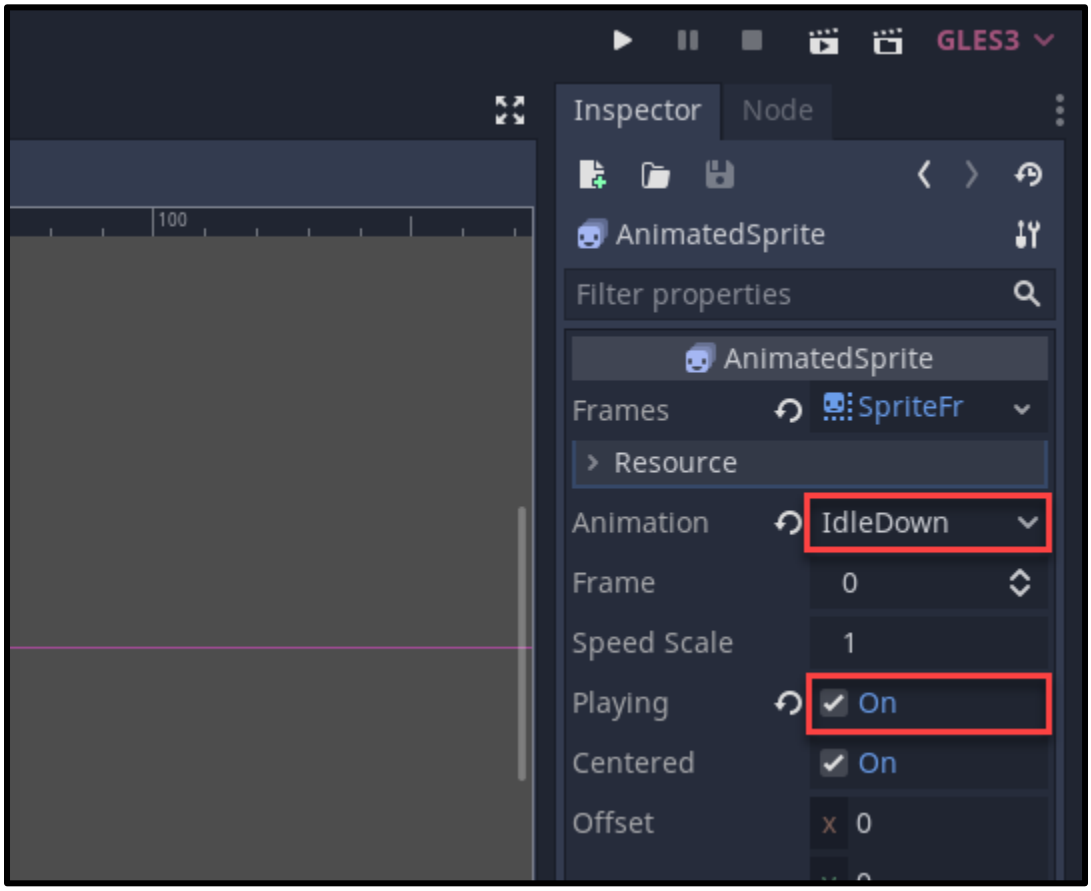
- IdleDown (already created)
- IdleUp
- IdleLeft
- IdleRight
- MoveDown
- MoveUp
- MoveLeft
- MoveRight

Here's what the **MoveDown** animation will look like. Click on the **New Animation** button to create a new animation.



Go through now and create all of the idle and move animations. Once that's done, select the **AnimatedSprite** node and...

- Set the **Animation** to *IdleDown*
- Enable **Playing**



Back in the **Player** script, let's create a new variable to reference the animated sprite node.

```
1 onready var anim = $AnimatedSprite
```

Then we can create the **manage_animations** function which will check the velocity and facing directions to determine which animation to play.

```
func manage_animations ():

    if vel.x > 0:
        play_animation("MoveRight")
    elif vel.x < 0:
        play_animation("MoveLeft")
    elif vel.y < 0:
        play_animation("MoveUp")
    elif vel.y > 0:
        play_animation("MoveDown")
    elif facingDir.x == 1:
        play_animation("IdleRight")
    elif facingDir.x == -1:
        play_animation("IdleLeft")
    elif facingDir.y == -1:
        play_animation("IdleUp")
    elif facingDir.y == 1:
        play_animation("IdleDown")
```

The **play_animation** function will take in an animation name and play that.

```
func play_animation (anim_name):

    if anim.animation != anim_name:
        anim.play(anim_name)
```

We can call the manage_animations function every frame at the end of the **_physics_process** function.

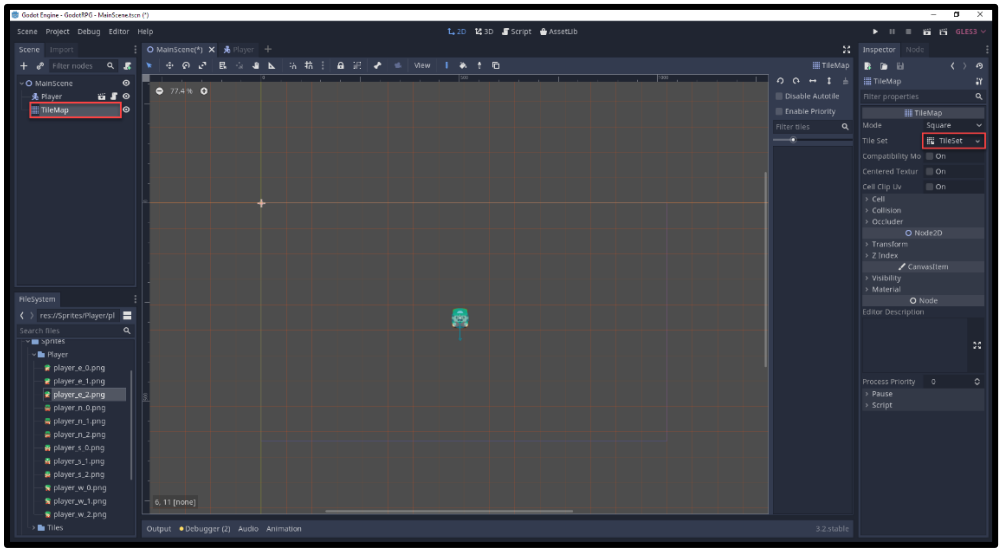
```
manage_animations()
```

Now when we press play, you should see that the animations play when we move around.

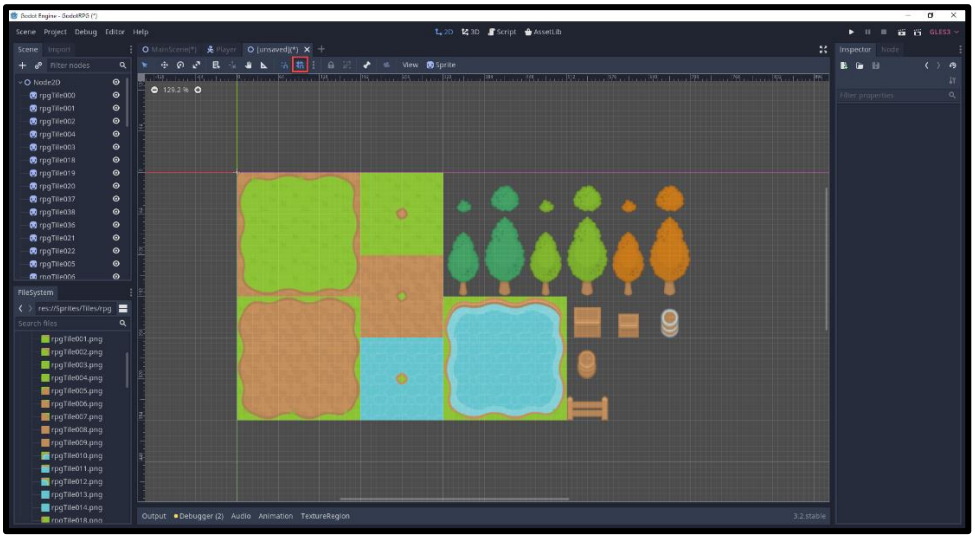
Creating the Tilemap

Now that we've got a player who can walk around, let's create our world.

In the **MainScene**, create a new child node of type **TileMap**. Then in the inspector, create a new tileset.

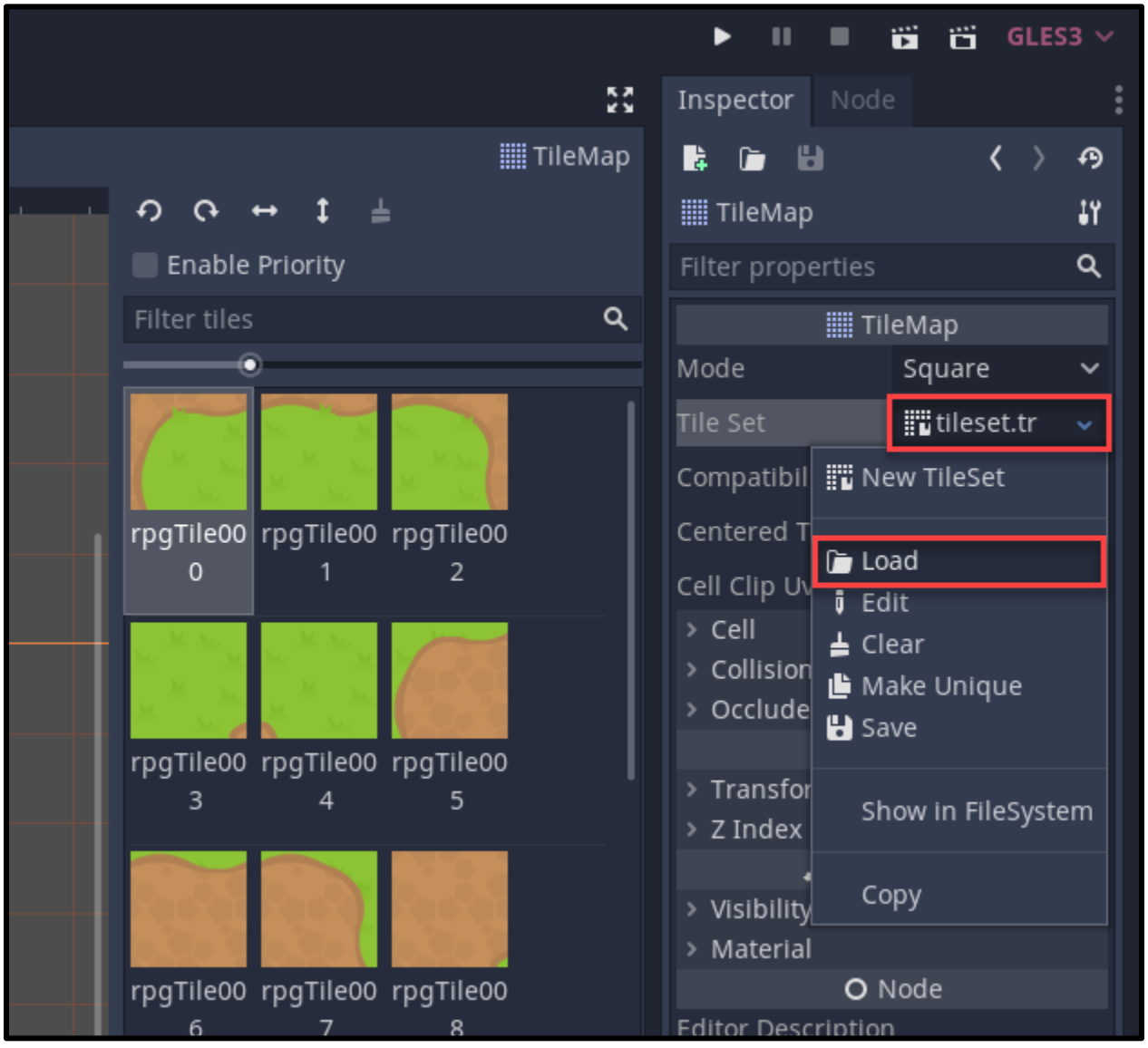


These tile sets require tiles, so to do this, we need to create a new scene with all the available tiles in it.



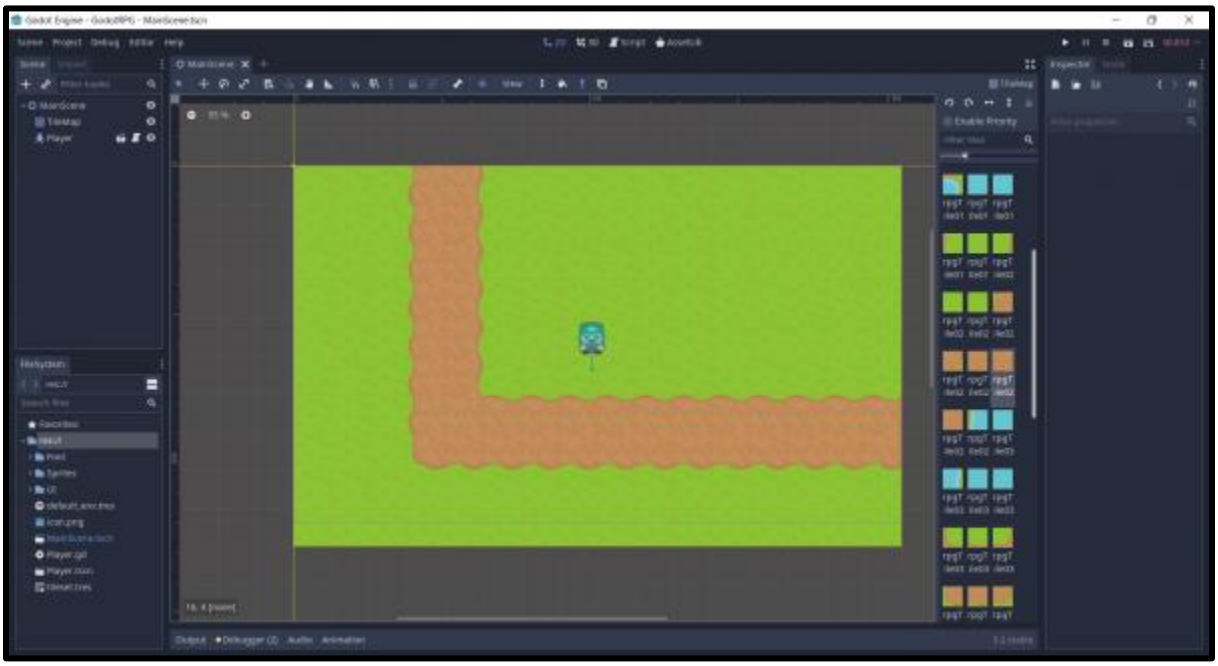
Once all the tiles are in the new scene, go to *Scene > Convert To... > Tile Set* and save it as **tileset.tres**.

Back in the **MainScene**, select the TileMap node and in the inspector, select the tileset property and choose load. Select the resource we just made and the tiles should appear in the panel to the right.



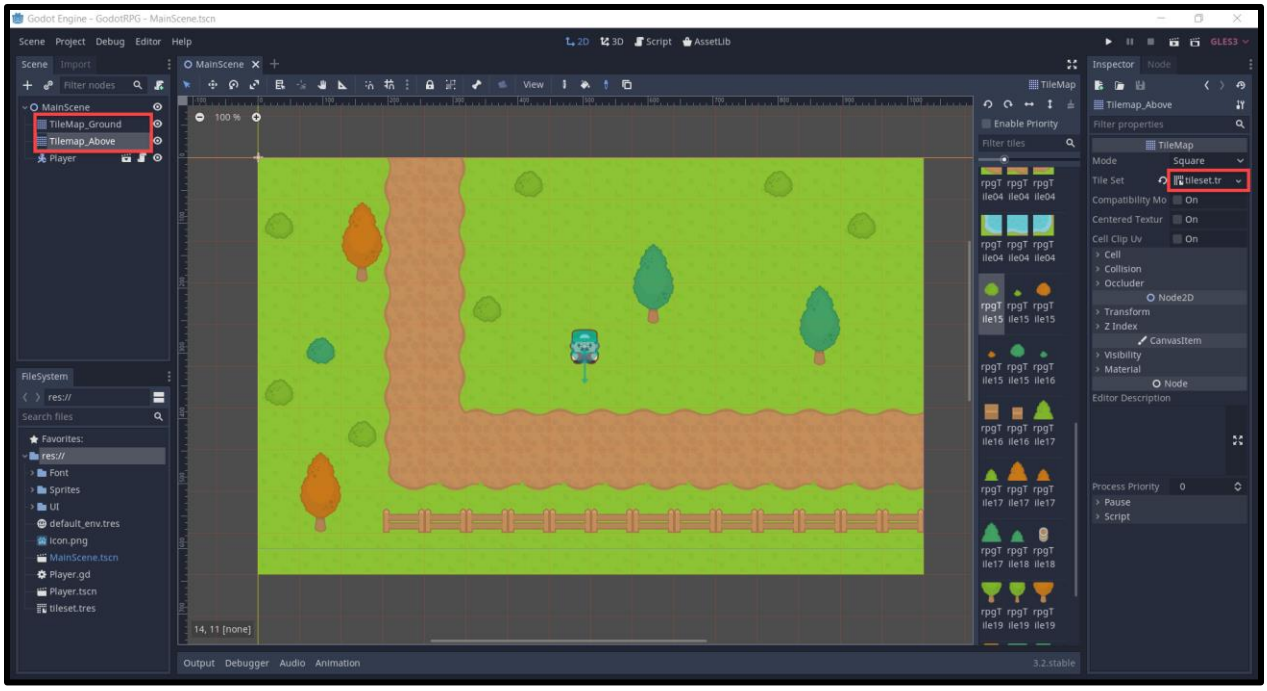
We can now select a tile and paint it in the scene view. Here are some keys for building your tilemap:

- Place tile - **Left Mouse**
- Delete tile - **Right Mouse**
- Draw line - **Shift + Left Mouse**
- Draw rectangle - **Shift + Ctrl + Left Mouse**
- Delete line and delete rectangle are the same but with right mouse



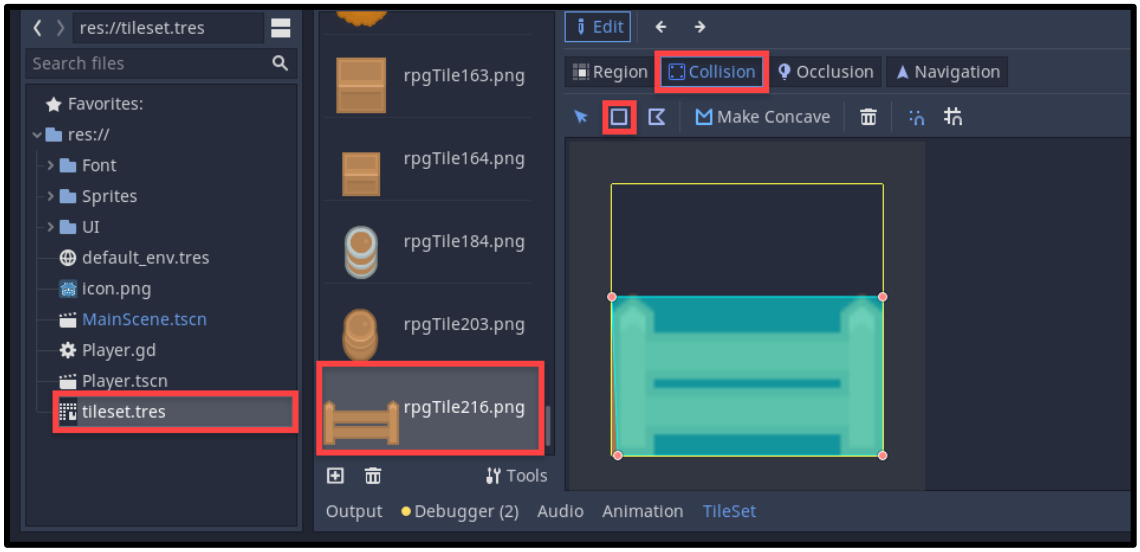
Now if we try to place a tree, you'll see that it replaces the tile rather than placing it on top. To fix this, we can create a second tilemap.

- Call the existing tilemap **TileMap_Ground**
- Create a new tilemap called **TileMap_Above**
- Set that tileset to the same tileset
- We can now draw tiles on-top of the existing tilemap

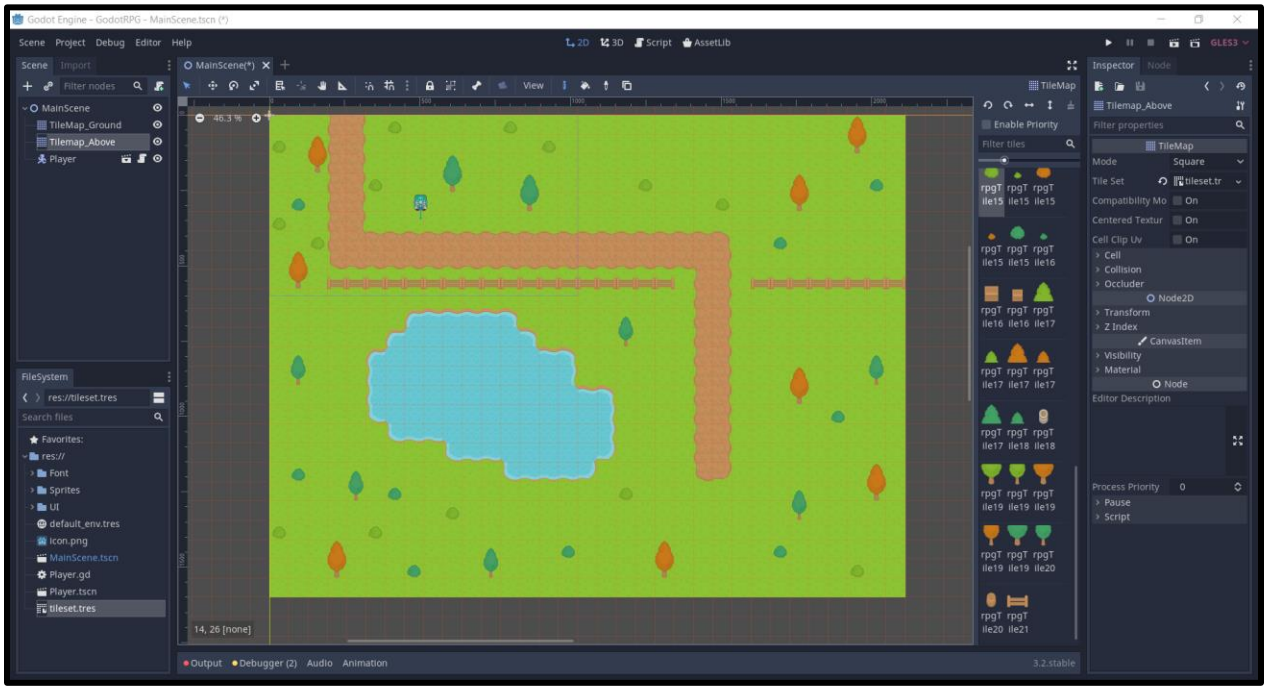


Something you might also want is a collision with some tiles. To implement this, select the tilemap in the file system and the TileSet panel should pop up.

- Select a tile
- Click on the sprite that pops up (a bunch of settings should then appear)
- Select **Collision**
- Select the **Rectangle** tool
- Draw the collider on the sprite



Let's now create a larger scene with our two tilemaps.



Continued in Part 2

With the tilemap created and player set up, we now have the basis for a spectacular 2D RPG in Godot! Over the course of part 1, we've covered not only setting up sprite animations for your player character in Godot, but also setting up raycasts and tilemaps with appropriate collisions. Fortunately, with just these foundations, you can expand into other games as well, or create even more complicated RPG maps.

Nevertheless, our RPG certainly isn't complete! In Part 2, we'll cover the rest of our RPG creation process with enemies, loot, UI systems, and more! Hope to see you then!

Build a 2D RPG in Godot - Part 2

Introduction

Welcome back everyone! It's time to finish our 2D RPG in Godot!

For Part 1 of this tutorial series, we started setting up our 2D RPG project in the open-source and free Godot game engine. Some of the things we learned about include: tilemaps, player controllers, raycasting, sprite animations, and more! All in all, we have a great base to work with and expand on!

However, what's an RPG without some enemies or items to loot? In this second part of our 2D RPG tutorial for Godot, we'll finish by adding these elements, as well as adding the ability for our camera to follow the player.

So sit back, and prepare yourself to finish your first RPG in Godot!

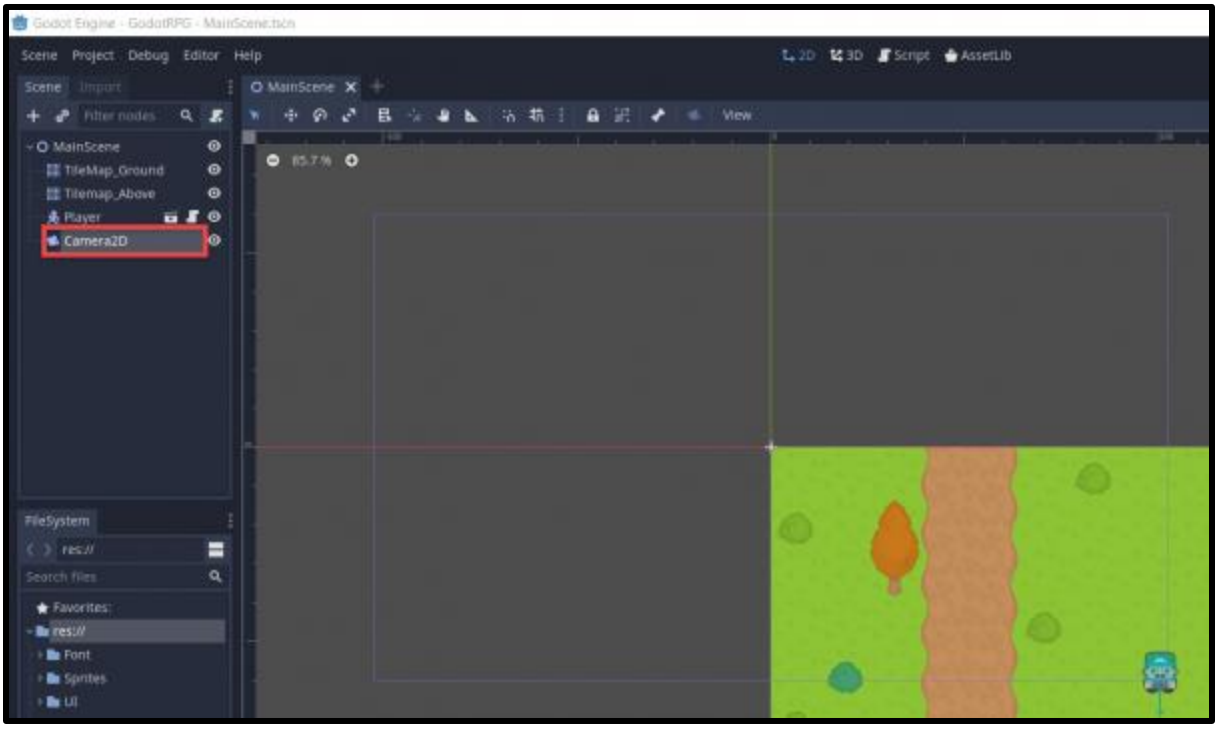
Project Files

For this project, we'll be needing some assets such as sprites and a font. These will be sourced from kenney.nl and Google Fonts. You can, of course, choose to use your own assets, but for this course we'll be using these.

The assets and source code can be downloaded here.

Camera Follow

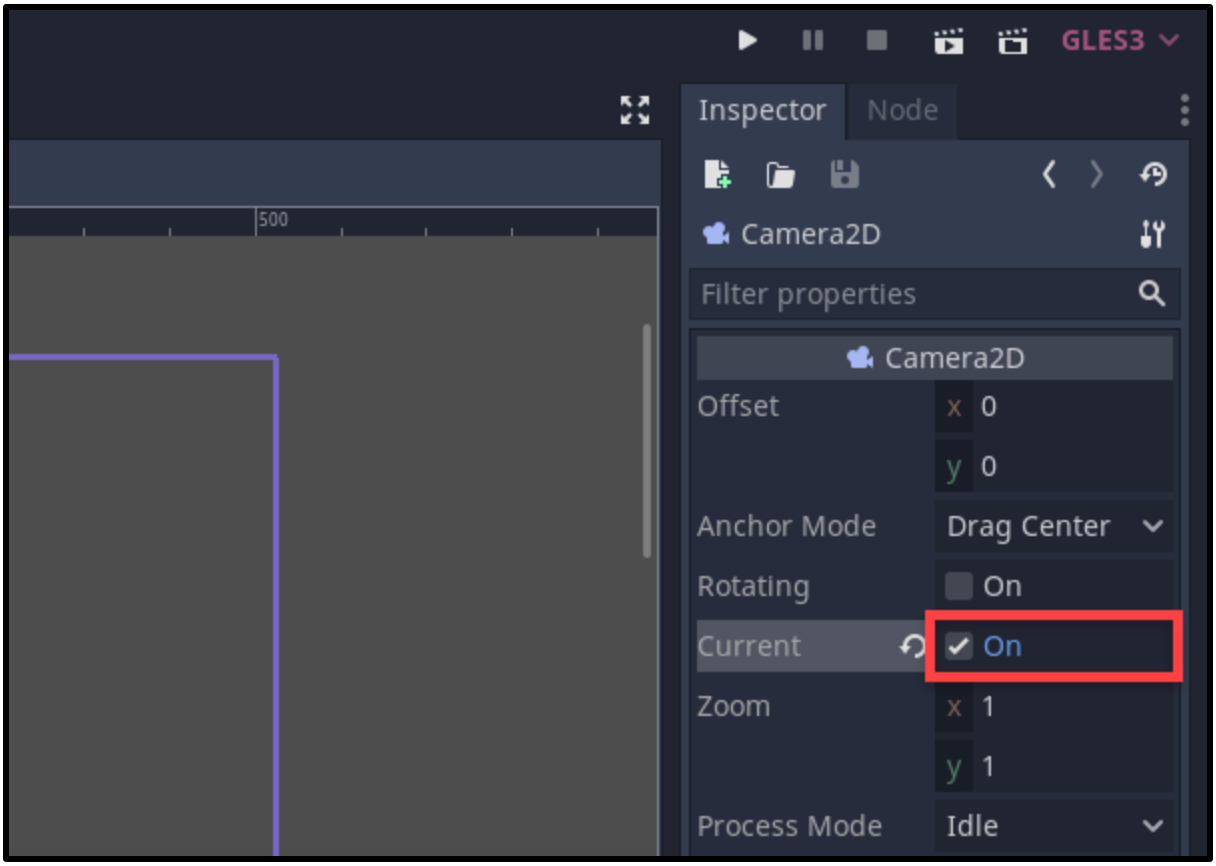
With our big scene, there's one problem which is the fact that the camera doesn't move. To fix this, let's create a new **Camera2D** node in the MainScene.



Then we can create a new script attached to the camera node. Call it **CameraFollow**. All we're going to do here is find the player and move towards them every frame.

```
1  onready var target = get_node("/root/MainScene/Player")
2
3  func _process (delta):
4      position = target.position
```

Finally, in the inspector make sure to enable **Current** so that the camera will be the one we look through.

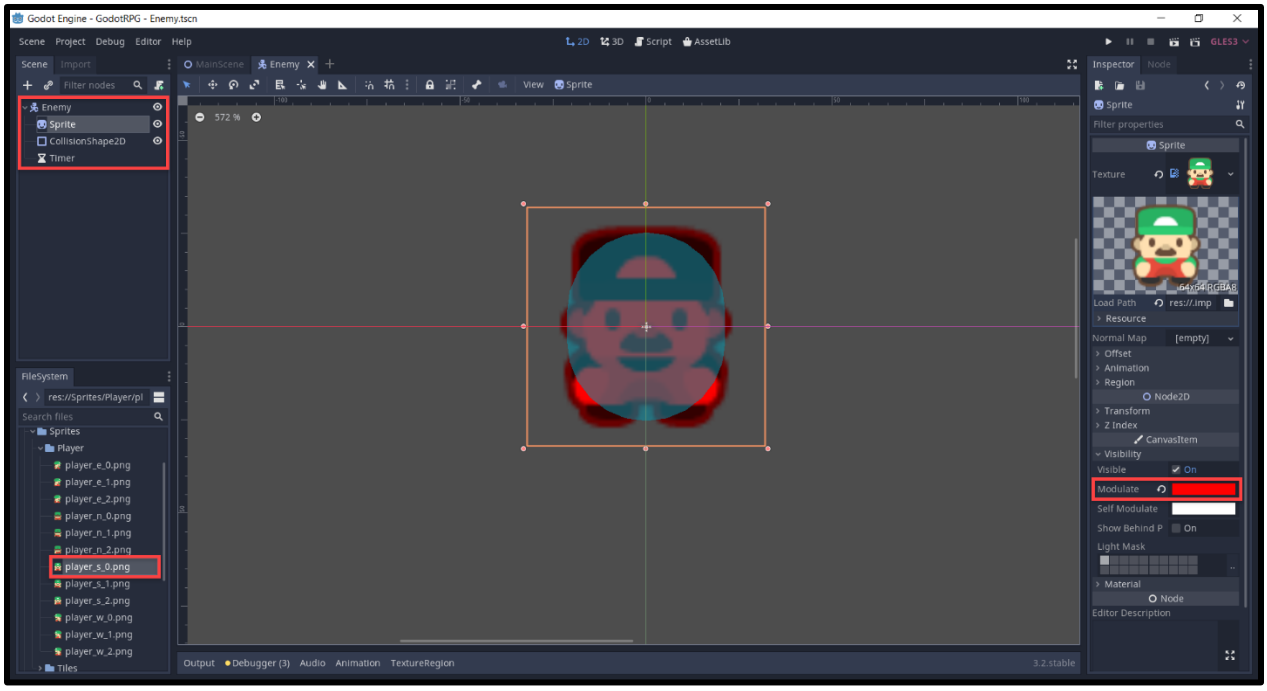


Now if we press play, you'll see that the camera follows the player.

Creating an Enemy

Let's now create an enemy scene. These will chase after the player and attack them when in range.

1. Create a new scene with a root node of **KinematicBody2D**
2. Rename it to **Enemy** and save the scene
3. Drag in the **player_s_0.png** sprite to create a sprite node
4. Set the **Modulate** to *red*
5. Create a **CollisionShape2D** node with a capsule shape and resize it
6. Create a **Timer** node - this will be used for checking for when we can attack



Scripting the Enemy

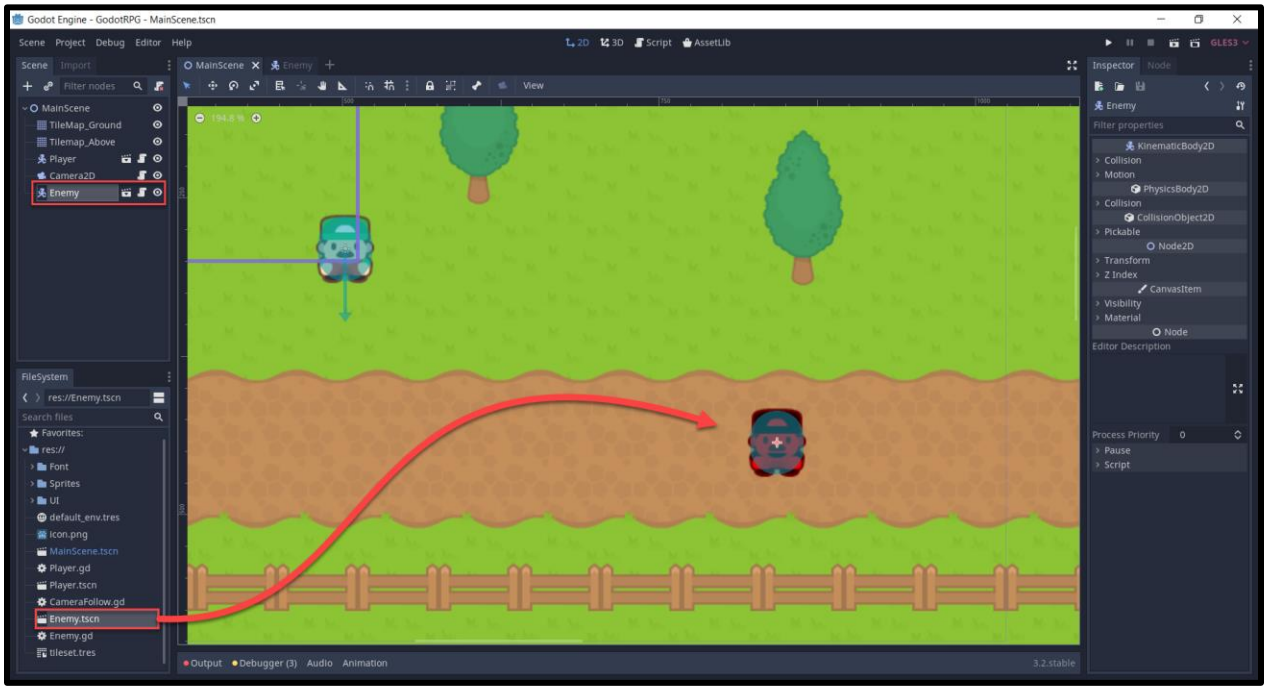
On the enemy node, create a new script called Enemy. We can start with the variables.

```
1  var curHp : int = 5
2  var maxHp : int = 5
3
4  var moveSpeed : int = 150
5  var xpToGive : int = 30
6
7  var damage : int = 1
8  var attackRate : float = 1.0
9  var attackDist : int = 80
10 var chaseDist : int = 400
11
12 onready var timer = $Timer
13 onready var target = get_node("/root/MainScene/Player")
```

Inside of the **_physics_process** function (gets called 60 times per second) we want to move towards the target if we're further than the attack distance but closer than the chase distance.

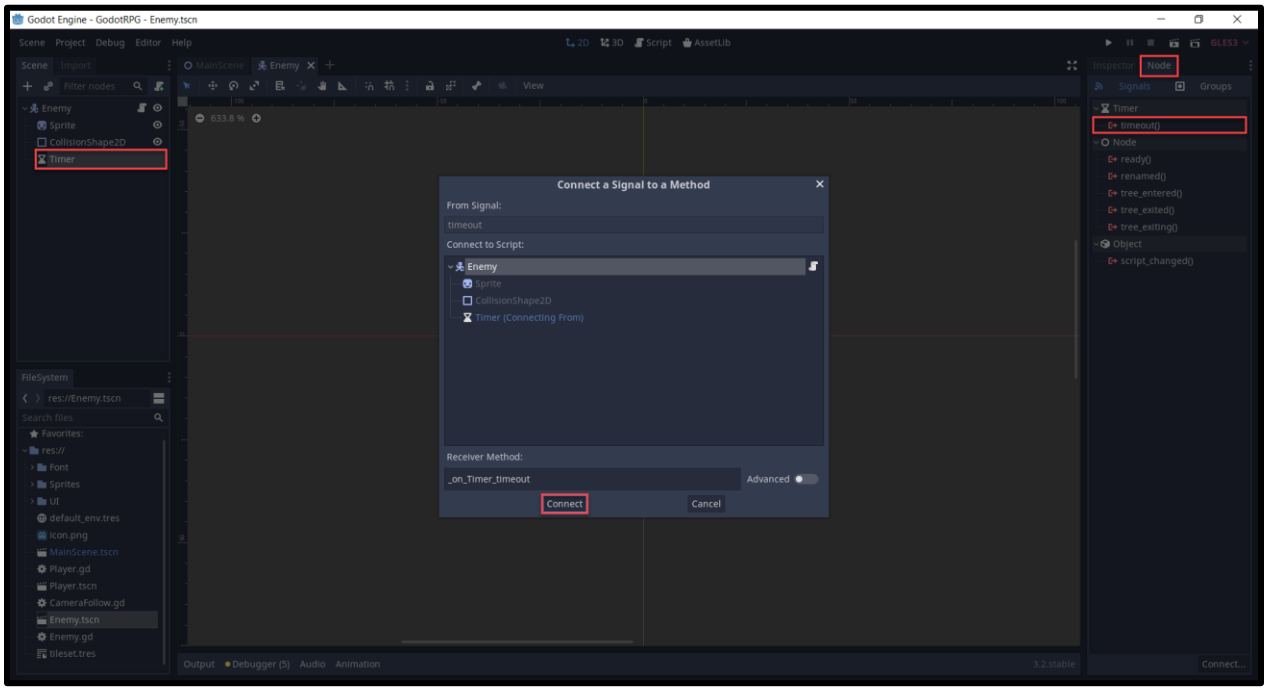
```
1  func _physics_process (delta):
2
3      var dist = position.distance_to(target.position)
4
5      if dist > attackDist and dist < chaseDist:
6          var vel = (target.position - position).normalized()
7
8          move_and_slide(vel * moveSpeed)
```

Back in the **MainScene** let's drag in the **Enemy** scene to create a new instance. Now we can press play and test it out. The enemy should move towards us.



Now that the enemy moves towards us, let's have it attack us.

1. Go to the **Enemy** scene
2. Select the **Timer** node
3. In the Inspector panel, click on the **Node** tab
4. Double click on the **timeout()** signal
5. Click **Connect**



This will create the **_on_Timer_timeout** function. Here, we want to check if we're within the attack distance and then attack the target. We'll be creating the **take_damage** function on the player soon.

```
1  func _on_Timer_timeout ():
2
3      if position.distance_to(target.position) <= attackDist:
4          target.take_damage(damage)
```

In the **_ready** function which gets called once the node is initialized, let's setup the timer node.

```
1  func _ready ():
2
3      timer.wait_time = attackRate
4      timer.start()
```

Finally for the enemy, let's create the **take_damage** and **die** functions. The **target.give_xp** function will be called over on the player script later on.

```
1  func take_damage (dmgToTake):
2
3      curHp -= dmgToTake
4
5      if curHp <= 0:
6          die()
7
8  func die ():
9
10     target.give_xp(xpToGive)
11     queue_free()
```

Player Functions

Now we want to go over to the **Player** script and add in some new functions to work with the enemies. Let's start with the **give_gold** function which will add gold to the player.

```
1  func give_gold (amount):
2
3      gold += amount
```

For the levelling system, let's create the **give_xp** and **level_up** functions.

```
func give_xp (amount):

    curXp += amount

    if curXp >= xpToNextLevel:
        level_up()

func level_up ():

    var overflowXp = curXp - xpToNextLevel

    xpToNextLevel *= xpToLevelIncreaseRate
    curXp = overflowXp
    curLevel += 1
```

The **take_damage** and **die** functions will be called when an enemy attacks us.

```
func take_damage (dmgToTake):

    curHp -= dmgToTake

    if curHp <= 0:
        die()

func die ():

    get_tree().reload_current_scene()
```

Now if we press play, you'll see that the enemy can attack and kill us - which will reload the scene.

Player Interaction

With our player, we want the ability to interact with chests and enemies. In the **Player** script, let's create the _**process** function (called every frame) and check for when we're pressing the interact button.

```
1  func _process (delta):
2
3      if Input.is_action_just_pressed("interact"):
4          try_interact()
```

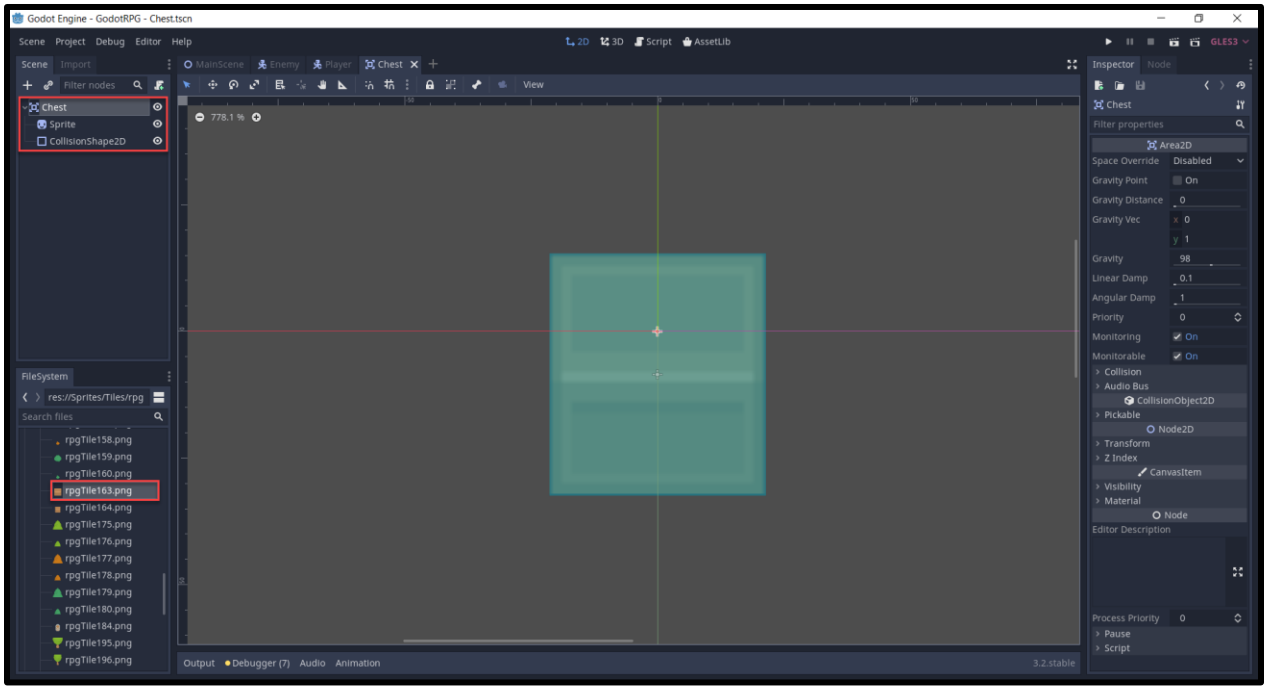
The **try_interact** function will check to see if the raycast is hitting anything. If it's an enemy, damage them but if it's not - call the **on_interact** function if they have it.

```
1  func try_interact ():
2
3      rayCast.cast_to = facingDir * interactDist
4
5      if rayCast.is_colliding():
6          if rayCast.get_collider() is KinematicBody2D:
7              rayCast.get_collider().take_damage(damage)
8          elif rayCast.get_collider().has_method("on_interact"):
9              rayCast.get_collider().on_interact(self)
```

Chest Interactable

Now we need something to interact with. This is going to be a chest which will give the player some gold. Create a new scene with a root node of **Area2D**.

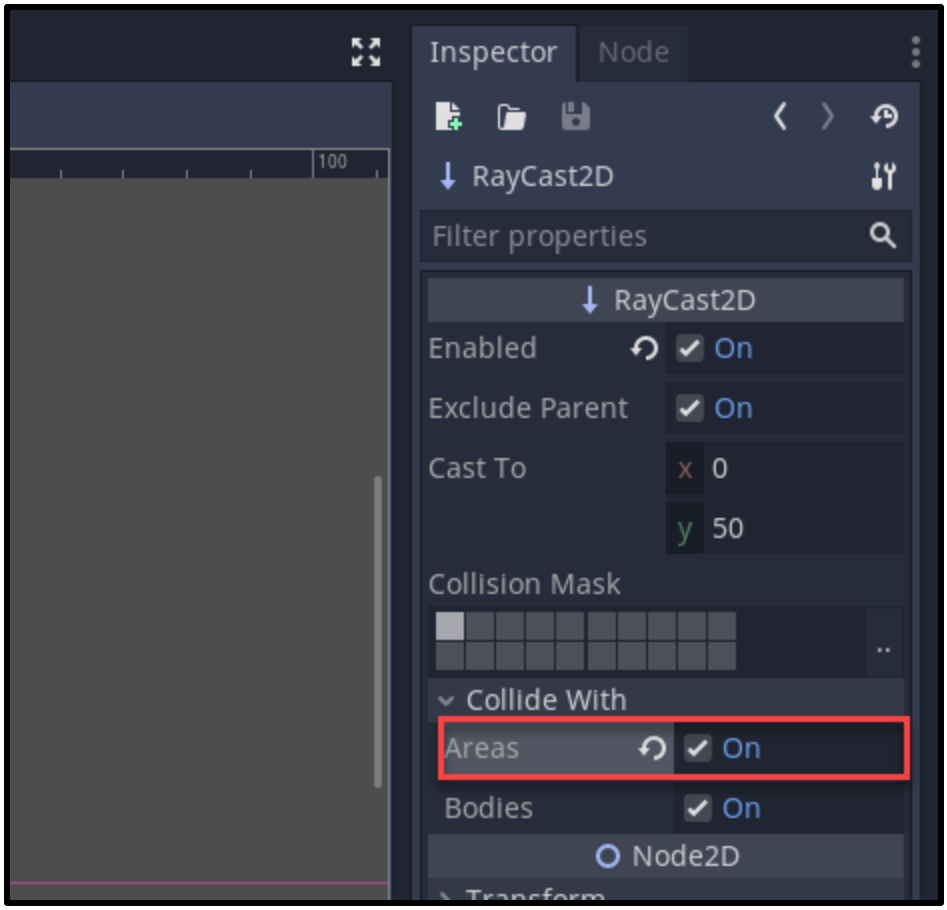
1. Rename the node to **Chest**
2. Save the node
3. Drag in the **rpgTile163.png** to create a Sprite node
4. Create a **CollisionShape2D** node
5. Set the shape to a *Rectangle* and resize it to fit the sprite



Next, create a new script called **Chest** attached to the Area2D node. All we're going to do here, is have the **on_interact** function which gives the player gold then destroys the node.

```
1  export var goldToGive : int = 5
2
3  func on_interact (player):
4
5      player.give_gold(goldToGive)
6      queue_free()
```

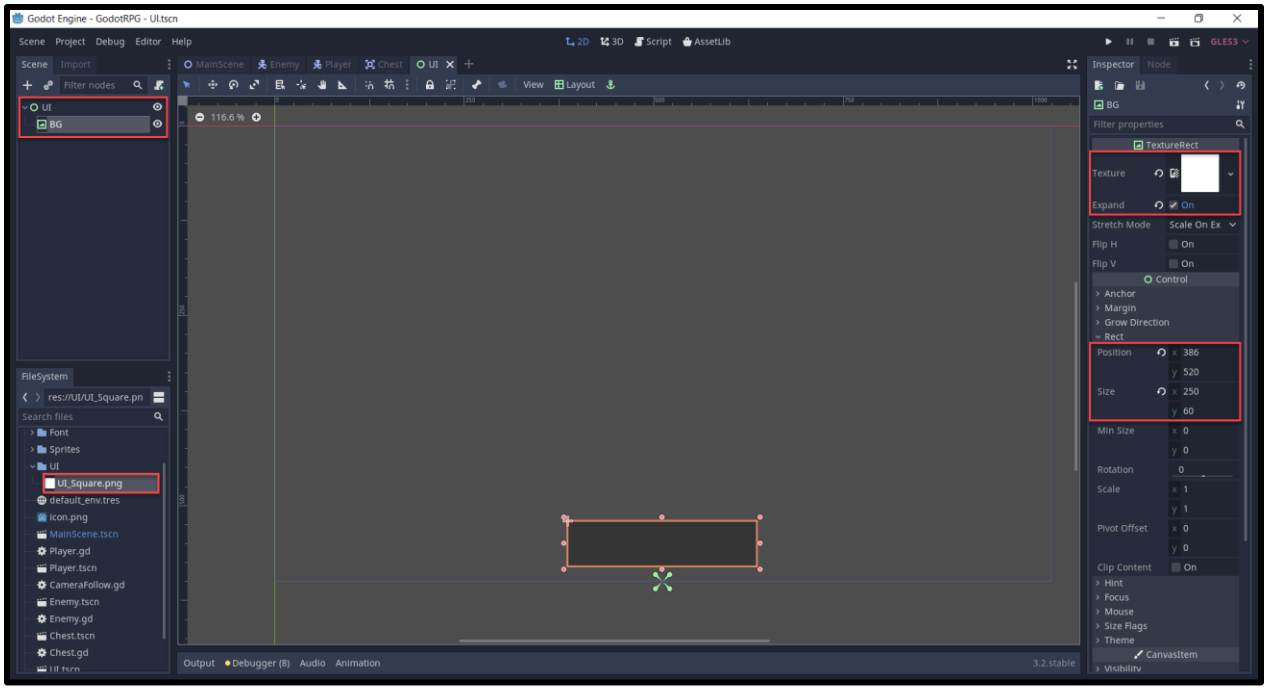
You might notice that interacting with the chest wont work but attacking the enemy does work. So in the **Player** scene, select the ray cast node and enable **Collide With Areas**.



Creating the UI

Create a new scene with a root node of **Control**.

1. Rename the node to **UI**
2. Save the scene
3. Create a new **TextureRect** node and call it **BG**
4. Set the **Texure** to *UI_Square.png*
5. Enable **Expand**
6. Drag the 4 anchor points down to the bottom center
7. Set the **Position** to *386, 520*
8. Set the **Size** to *250, 60*
9. Set the **Visibility > Self Modulate** to *dark grey*

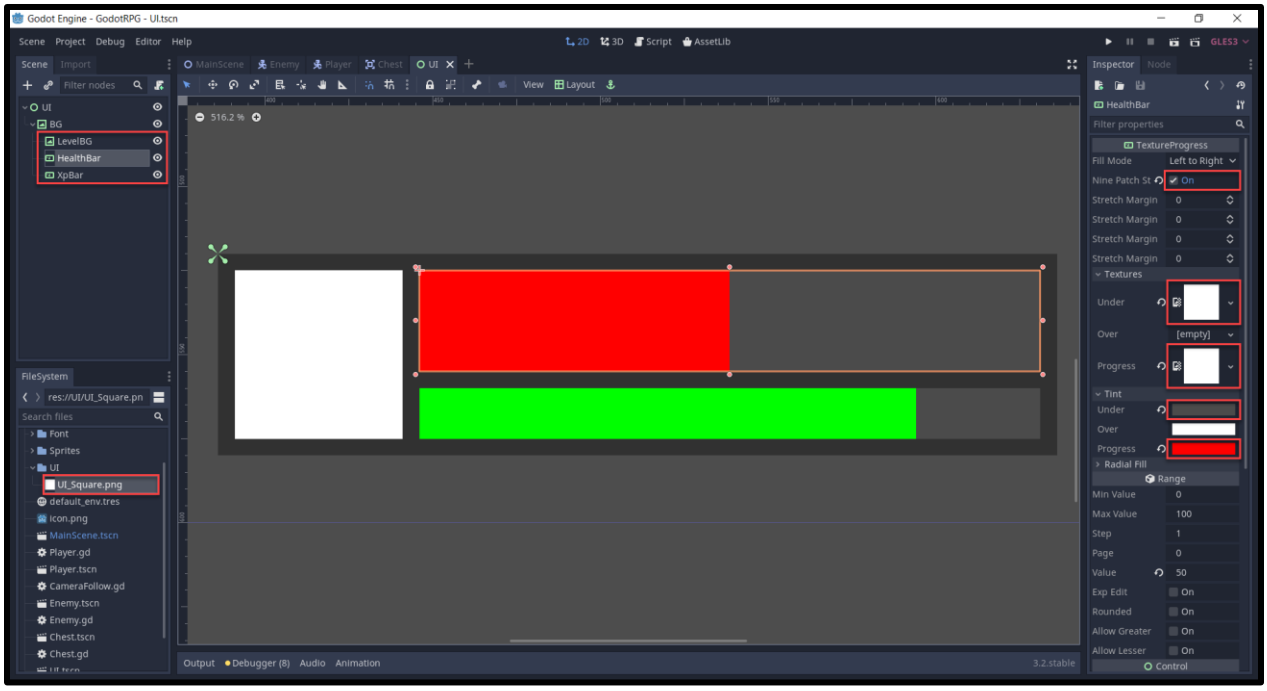


As a child of this **BG** node, we want to create three new nodes.

- LevelBG - **TextureRect**
- HealthBar - **TextureProgress**
- XpBar - **TextureProgress**

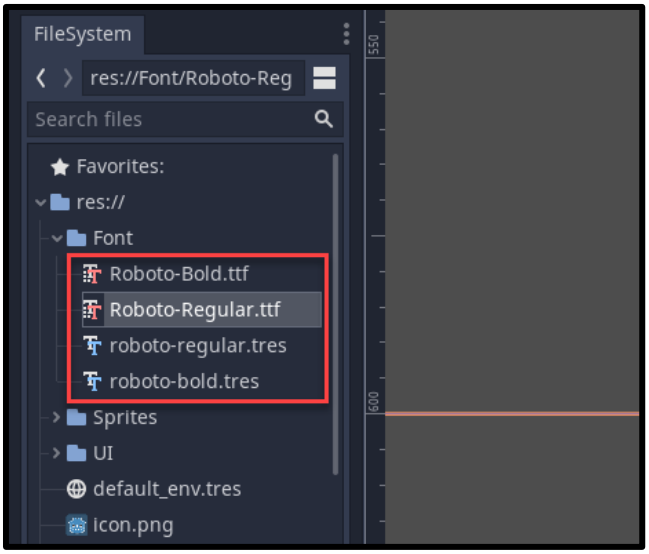
For the **TextureProgress** nodes...

1. Enable **Nine Patch Stretch**
2. Set the **Under** and **Progress** textures to *UI_Square.png*
3. Set the **Under** tint to *dark grey*
4. Set the **Progress** tint to *red* or *green*



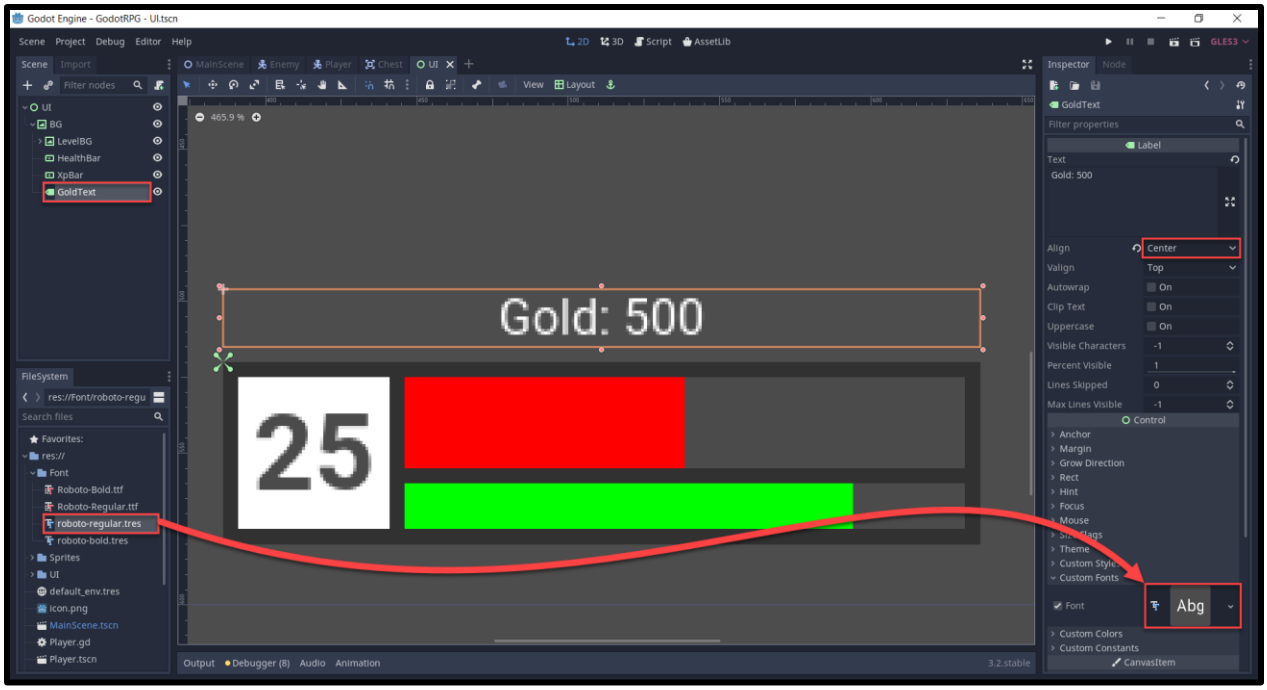
Now we need to setup our fonts. With the assets we imported into the project, we have two font files. Right now, these are just .ttf files which need to be converted in a format that Godot can read. For each font file...

1. Right click it and select **New Resource...**
2. Select the **DynamicFont** resource
3. Save this to the Font folder
4. Double click on the new resource and drag the .ttf file into the **Font > Font Data** property



As a child of the **LevelBG** node, create a new **Label** node and rename it to **LevelText**.

- Resize the rect box to fit the level BG square
- Set **Align** and **Valign** to *Center*
- Drag the bold dynamic font resource into the **Custom Fonts** property
- Set the font **Size** to *35*
- Set the **Visibility > Self Modulate** to *grey*



Create a new **Label** node called **GoldText.**

1. Move and resize it to look like below
2. Set **Align** to *Center*
3. Set the **Custom Font** to the regular dynamic font resource

Scripting the UI

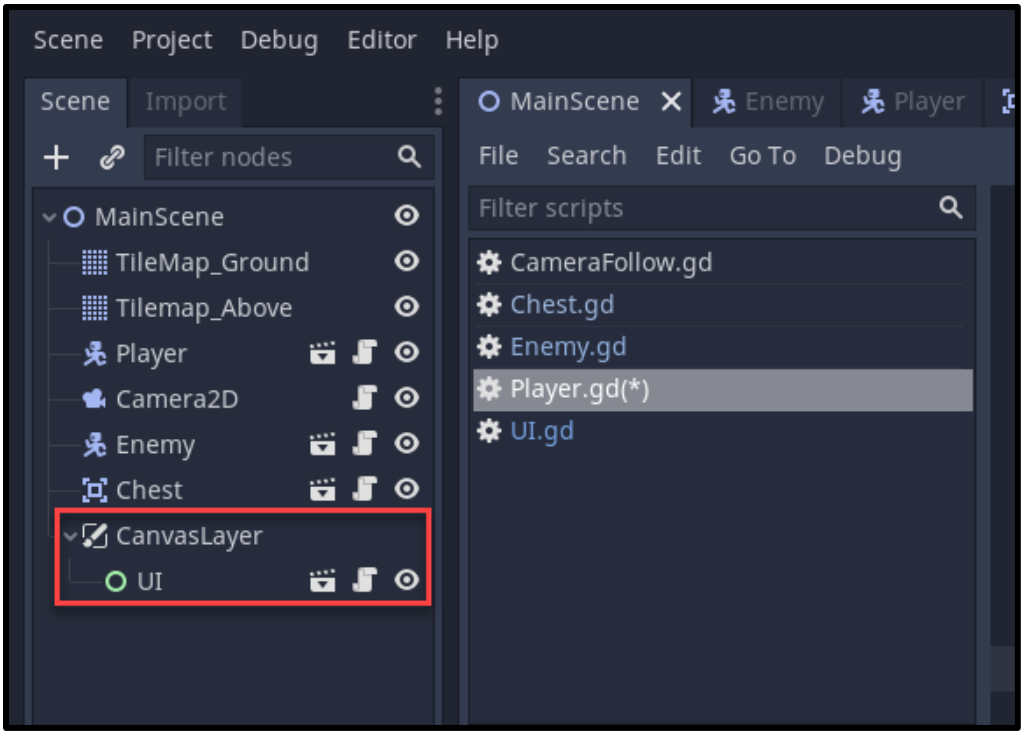
Next up, let's create a new script attached to the **UI** node, called **UI**. We'll start with the variables to reference the children nodes.

```gdscript
onready var levelText : Label = get_node("BG/LevelBG/LevelText")
onready var healthBar : TextureProgress = get_node("BG/HealthBar")
onready var xpBar : TextureProgress = get_node("BG/XpBar")
onready var goldText : Label = get_node("BG/GoldText")
```

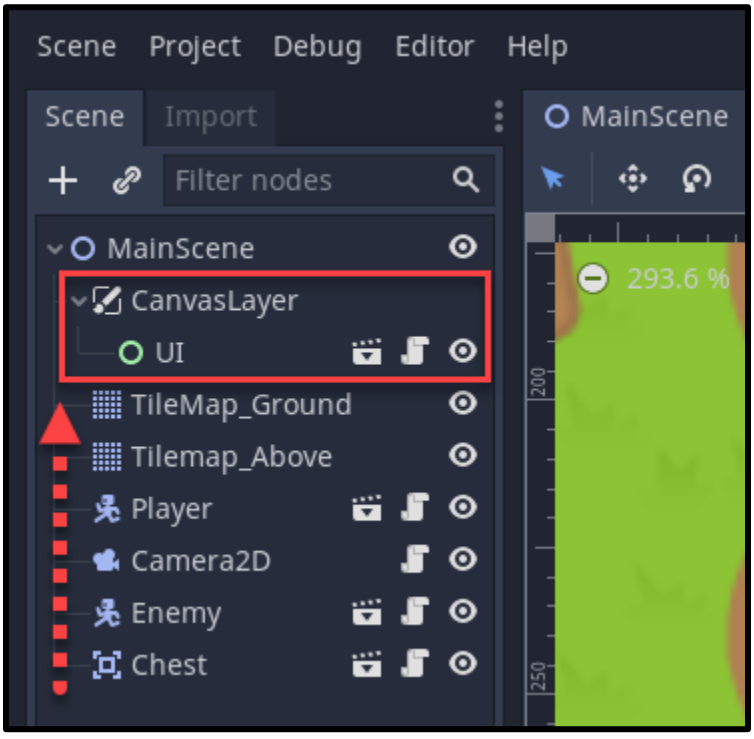
Then we have our functions which update each of these nodes.

```gdscript
# updates the level text Label node
func update_level_text (level):

    levelText.text = str(level)

# updates the health bar TextureProgress value
func update_health_bar (curHp, maxHp):

    healthBar.value = (100 / maxHp) * curHp

# updates the xp bar TextureProgress value
func update_xp_bar (curXp, xpToNextLevel):

    xpBar.value = (100 / xpToNextLevel) * curXp

# updates the gold text Label node
func update_gold_text (gold):

    goldText.text = "Gold: " + str(gold)
```

Now that we have the **UI** script, let's go over to the **MainScene** and drag the **UI** scene in. To make it render to the screen, we need to create a **CanvasLayer** node and make UI a child of it.



In order to be able to access the UI script for the player, we need to move the canvas layer up to the top of the hierarchy.



Over in the **Player** script, let's create a variable to reference the UI script.

```
onready var ui = get_node("/root/MainScene/CanvasLayer/UI")
```

Create the _**ready** function which gets called once the node is initialized. In there, we'll be initializing the UI.

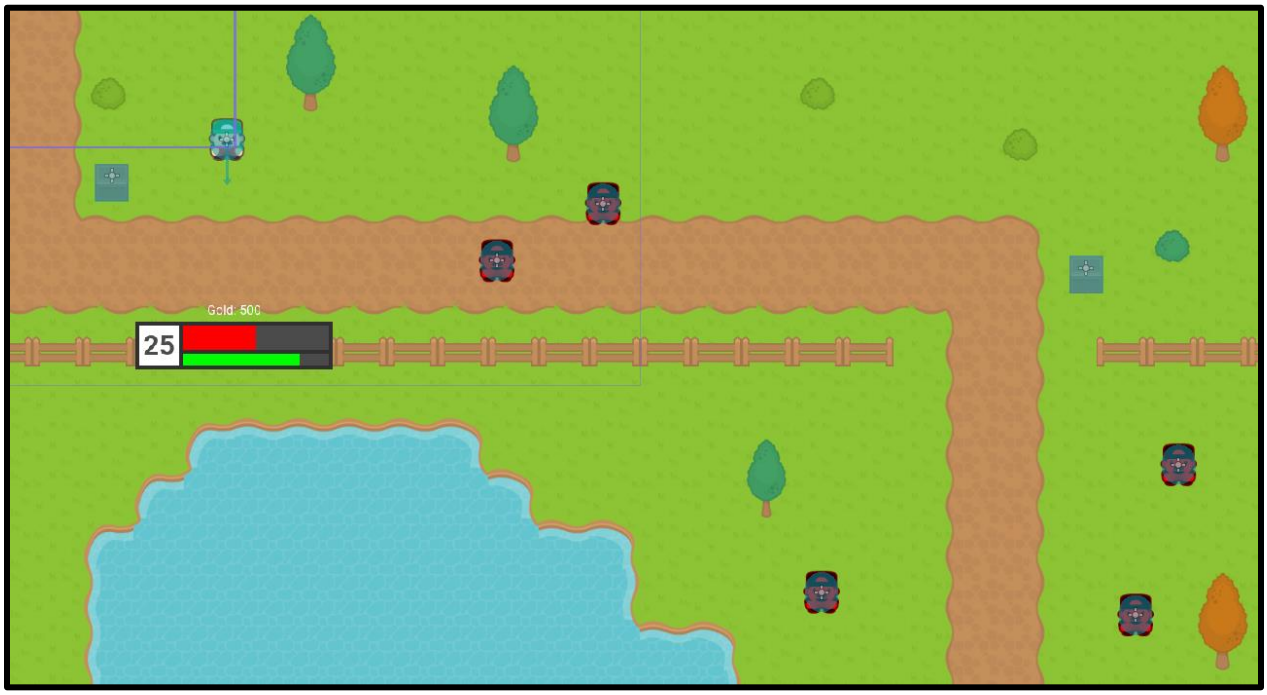
```
func _ready ():

    ui.update_level_text(curLevel)
    ui.update_health_bar(curHp, maxHp)
    ui.update_xp_bar(curXp, xpToNextLevel)
    ui.update_gold_text(gold)
```

Now we need to go through a few functions and call the UI functions at the end of them. They go as following:

```
# give_gold function
ui.update_gold_text(gold)

# give_xp function
ui.update_xp_bar(curXp, xpToNextLevel)

# level_up function
ui.update_level_text(curLevel)
ui.update_xp_bar(curXp, xpToNextLevel)

# take_damage function
ui.update_health_bar(curHp, maxHp)
```

We can now press play and see that the UI is set at the start of the game and changes as we get gold, take damage, get xp and level up.

With all of this, we're finished with our RPG game in Godot. Let's now go back to the **MainScene** and place some enemies and chests around the place.



Conclusion

Congratulations on completing the tutorial! Through perseverance, hard work, and a little bit of know-how, we just created a 2D RPG in Godot with a number of different features. Over the course of Part 1 and Part 2, we've shown you how to set up:

- A top-down 2D player controller
- Enemies who can attack and chase the player
- Chests we can interact with to get gold
- A UI to show our health, XP, level, and gold

And with that, you should now possess the fundamental skills to create even more RPGs in the future! Of course, you can expand upon the game created here - adding in new features, fleshing out current ones, or even just improving certain aspects you felt were lacking! There are endless directions to go from here, but with this solid foundation, you'll have quite the headstart!

Thank you very much for following along, and I wish you the best of luck with your future games.

Develop a 3D Action RPG in Godot - Part 1

Introduction

Ready to create your own game with some action-oriented gameplay?

In some of our previous tutorials, we covered making a 3D FPS and a 2D RPG. Both are well-loved genres in the gaming industry and have a lot to offer players in terms of entertaining experiences. However, what if you wanted a 3D RPG or maybe something with a little more spice to it? For this tutorial, we're going to delve into just that and show you how to make a 3D, action RPG in the Godot game engine.

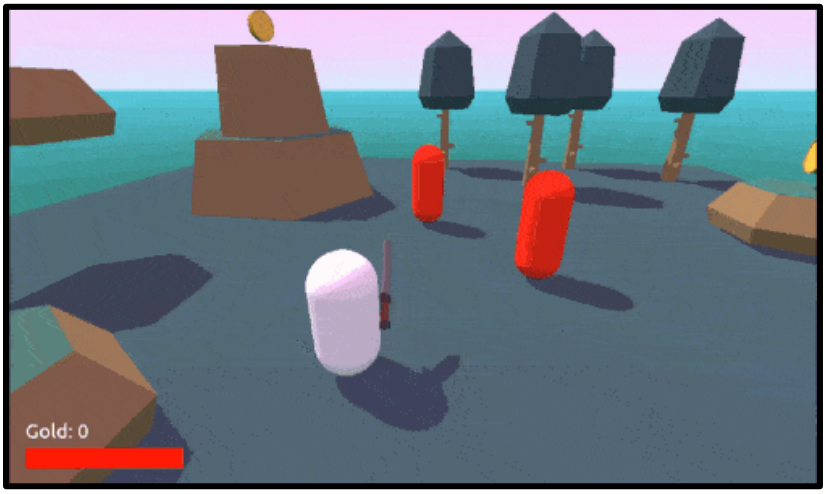
This 3D action RPG tutorial will cover a lot of ground, including how to make:

- A third-person player controller
- Enemies who follow and attack the player
- Melee combat system
- Collectible coins
- UI to show our health and gold

If that sounds great to you, we hope you sit back and are eager to create your own action RPG from scratch!

Before starting, please note that this tutorial won't be going over the basics of Godot. If this is your first time learning the engine, make sure to check out the introductory tutorial first.

Want to jump straight to making enemies and combat? You can try out Part 2 instead!



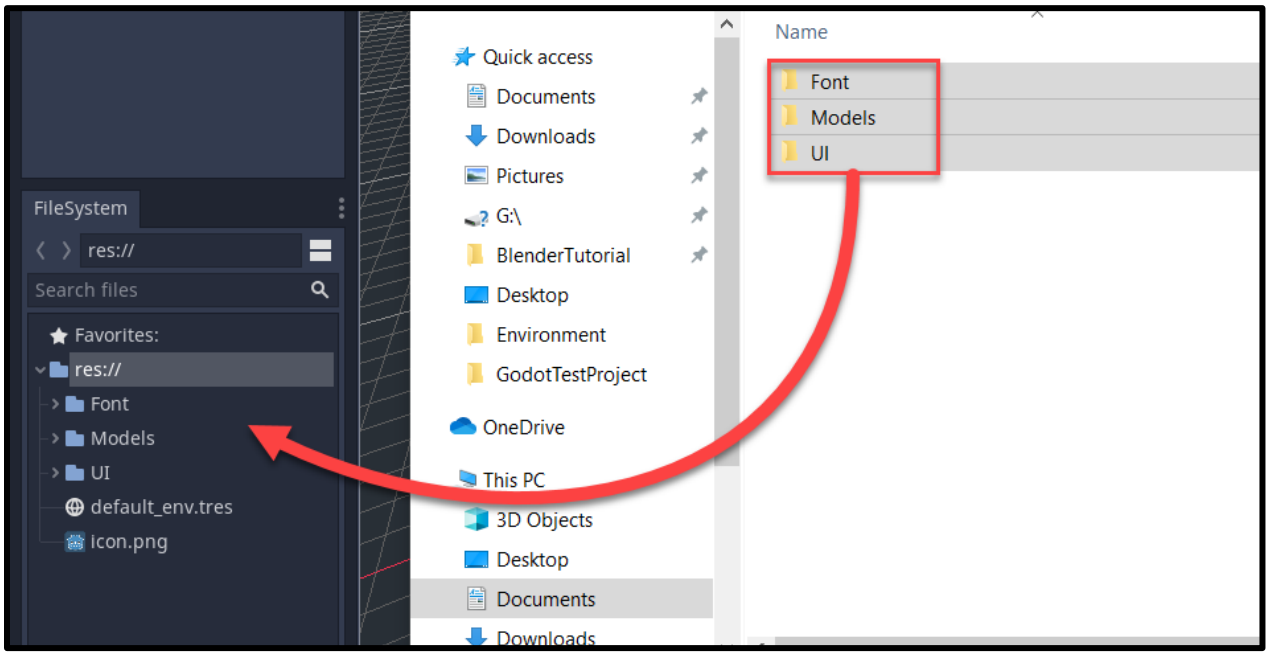
Project Files

In this course, we're going to be using a few models and a font to make the game look nice. You can, of course, choose to use your own, but we're going to be designing the game with these specific ones in mind. The models are from kenney.nl, a good resource for public-domain game assets. Then we're getting our font from Google Fonts.

You can download the source code and assets here.

Project Setup

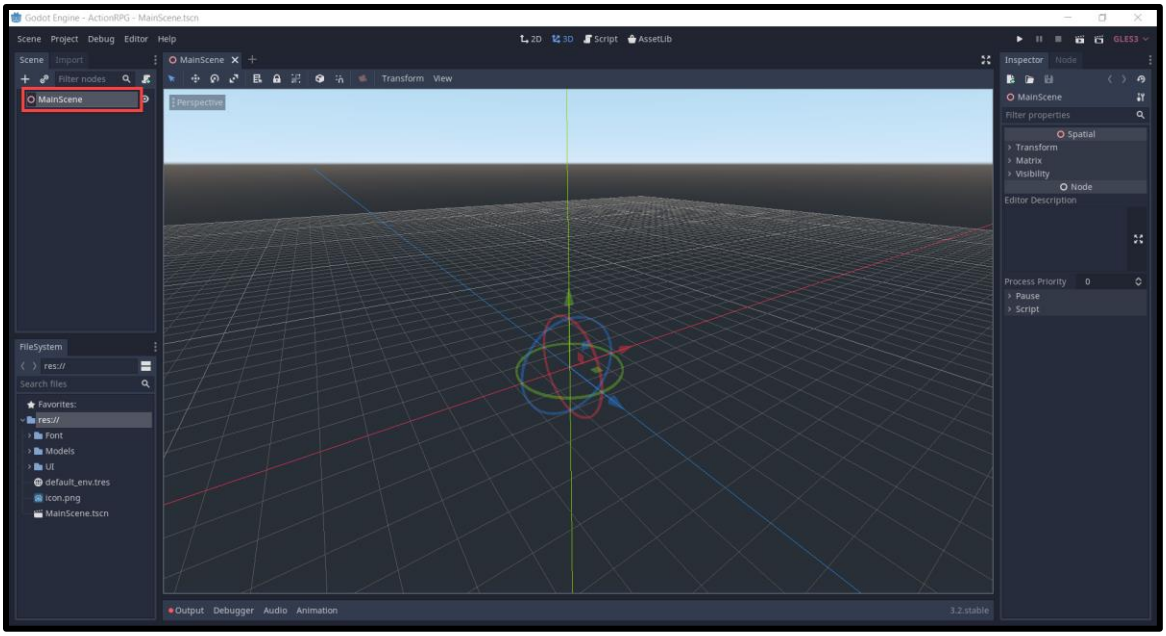
To begin, let's create a new Godot project. First, we're going to import the assets we'll need.



In the scene panel, select **3D Scene** as our first root node. Rename the node to **MainScene** and save it to the file system.

You'll see that we're in 3D mode here. In 3D, we have **Spatial** nodes. These are like **Node2D**'s, but allow us to position, rotate and scale them in 3D space. Below you'll see we have a few different colored lines.

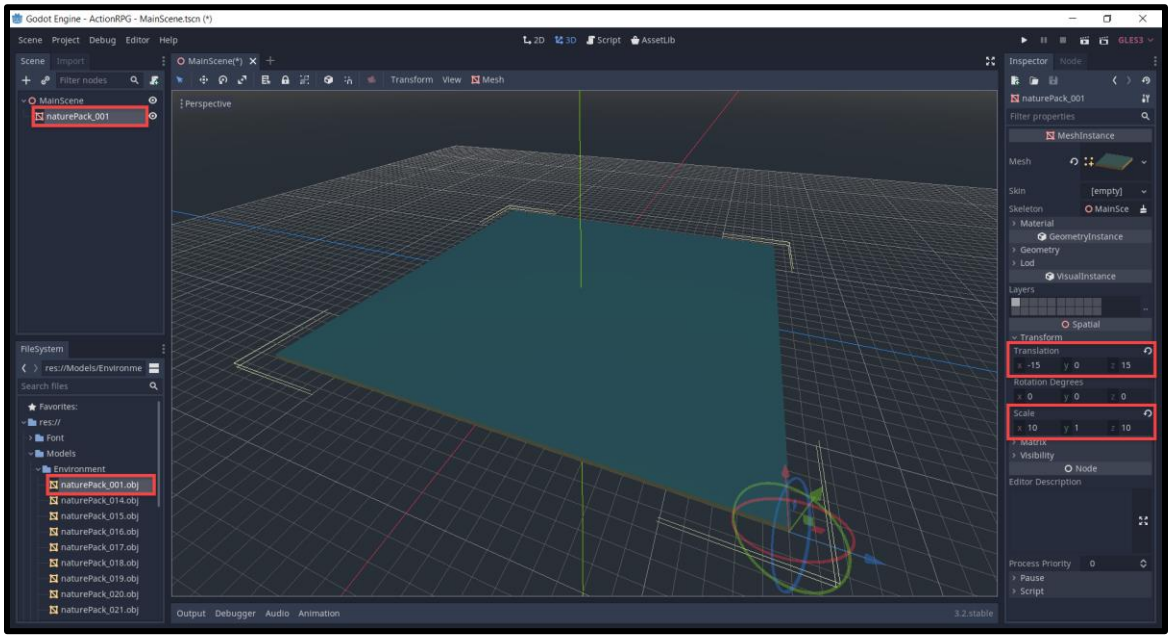
- Blue line = **Z axis**
- Red line = **X axis**
- Green line = **Y axis**



Creating Our Environment

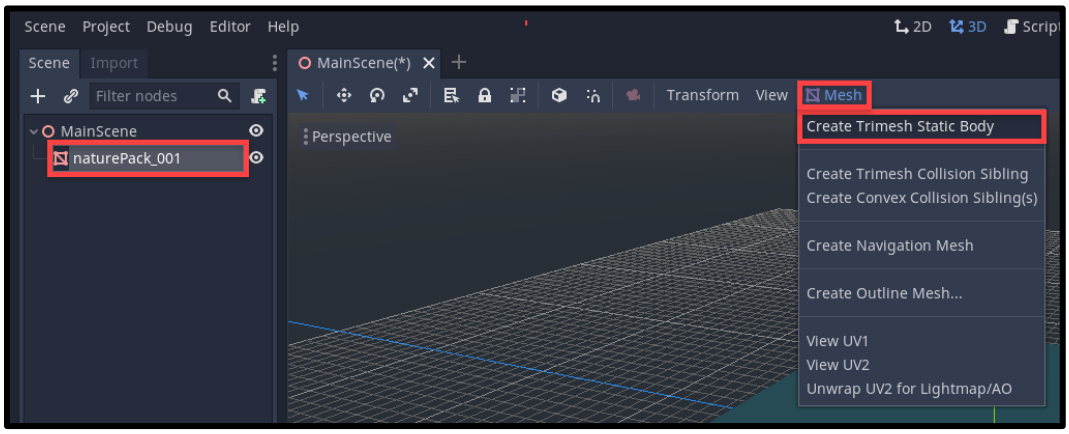
Here in the **MainScene**, we're going to start off by creating our environment.

1. Drag in the **naturePack_001.obj** model to create a new **MeshInstance** node
2. Set the **Translation** to *-15, 0, 15*
3. Set the **Scale** to *10, 1, 10*



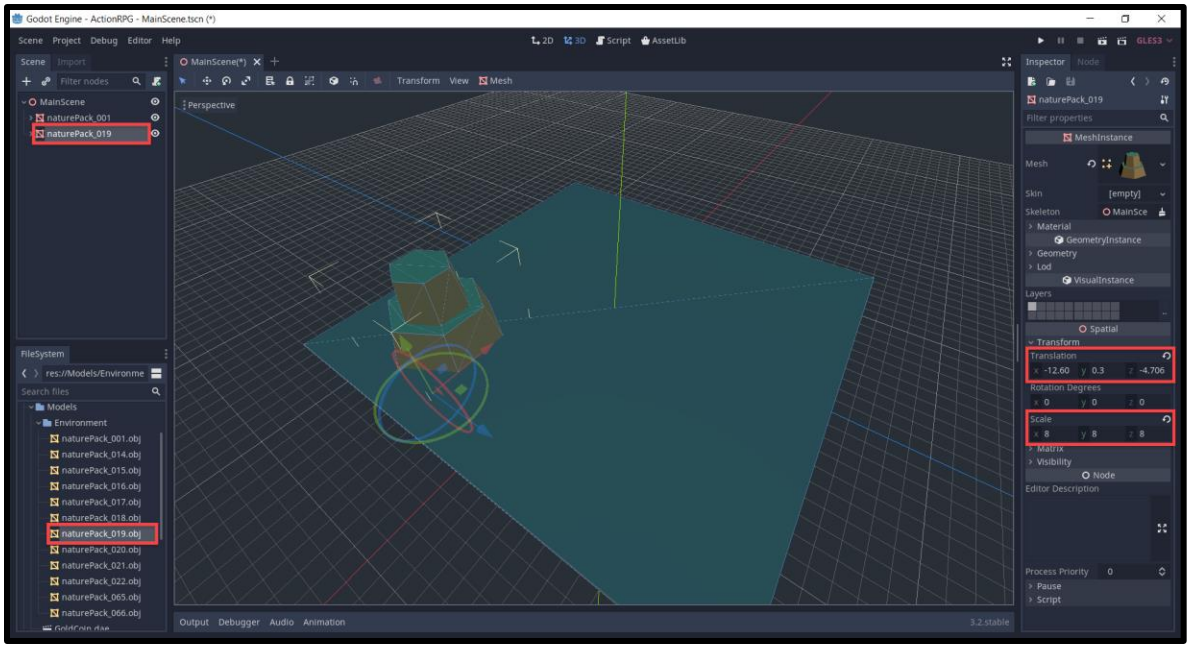
This is going to be our ground but we have one problem. There's no collider on the model so we're going to fall through it. To do this quickly, we can...

- Select the node
- Select **Mesh > Create Trimesh Static Body**

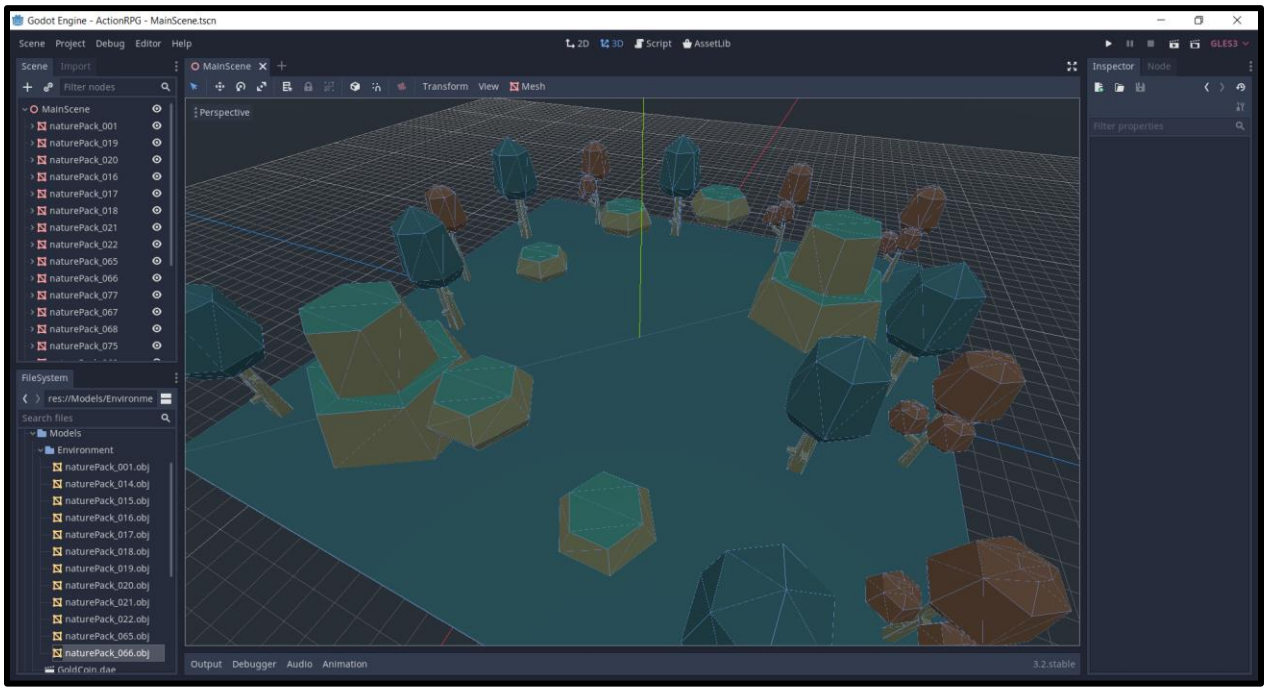


Now that we have the ground, let's drag in the **naturePack_019.obj** model.

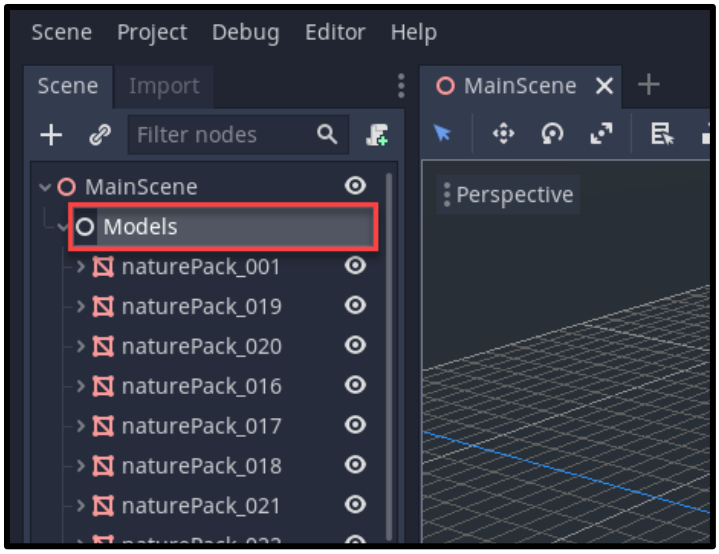
- Give it a collider **Mesh > Create Trimesh Static Body**
- Set the **Translation** to *-12.6, 0.3, -4.7*
- Set the **Scale** to *8, 8, 8*



We can then drag in more models, assign colliders to them and scale/position the nodes to create a nice looking environment.



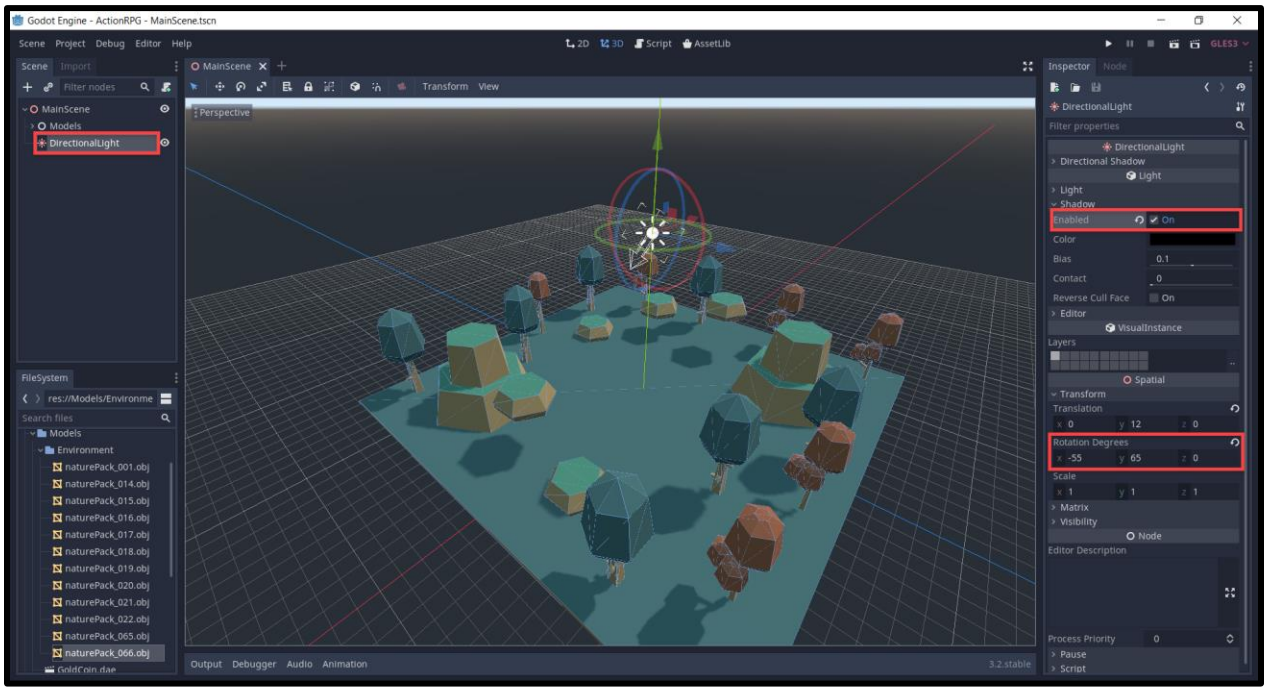
One thing you might notice is that the hierarchy is looking quite cluttered with all of these models. To fix this, we can create a new **Node** node and drag the models in as a child. This is the most simplistic type of node and is good for containers. We can then retract and expand the node when we need to.



Another thing you may notice is that it's quite dark. To fix this, we can create a **DirectionalLight** node which acts as our sun.

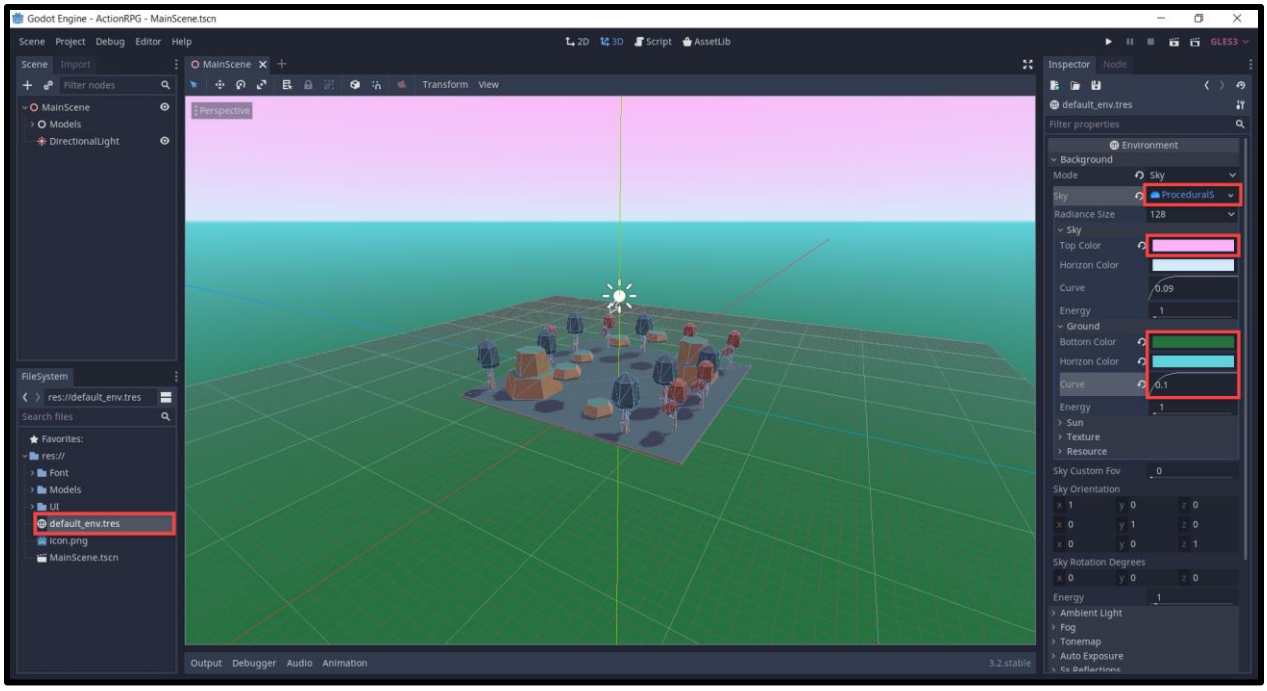
1. Set the **Rotation Degrees** to -55, 65, 0
2. Enable **Shadow > Enabled**

Now we have light.



Along with this, let's make the skybox look a bit nicer. Double click on the **default_env.tres** resource in the FileSystem to open up the options in the inspector.

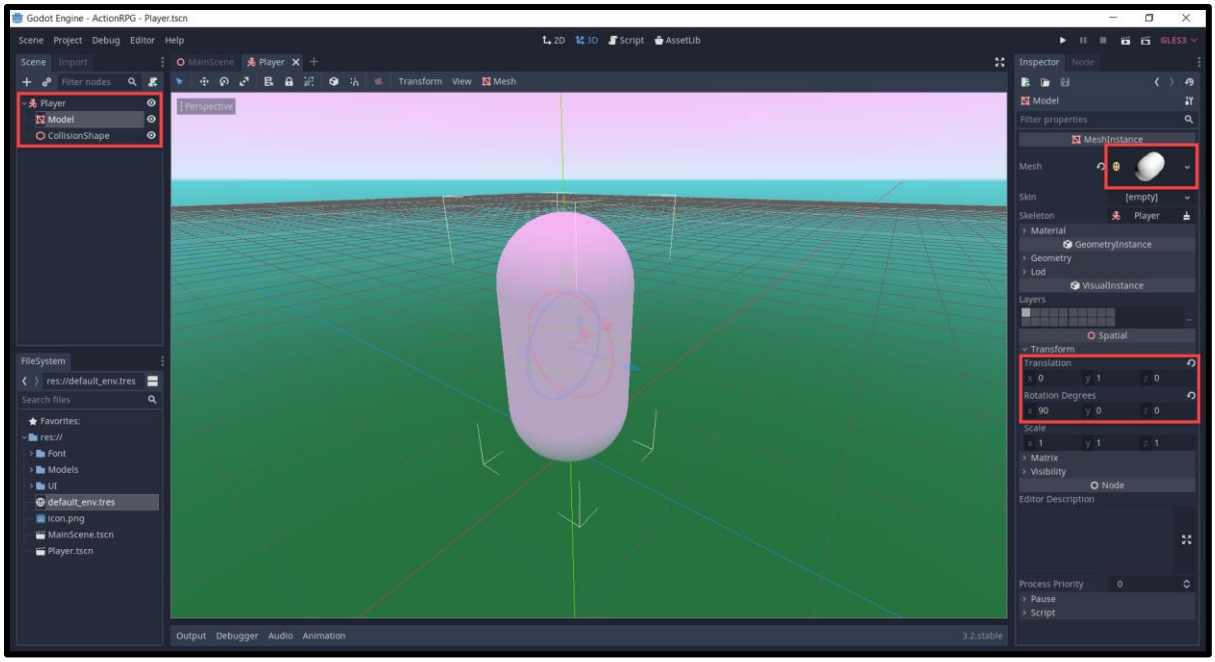
- Click on the **Sky** property to edit the skybox
- Set the **Top Color** to *pink*
- Set the **Bottom Color** to *green*
- Set the **Horizon Color** to *blue*
- Set the **Curve** to *0.1*



Creating the Player

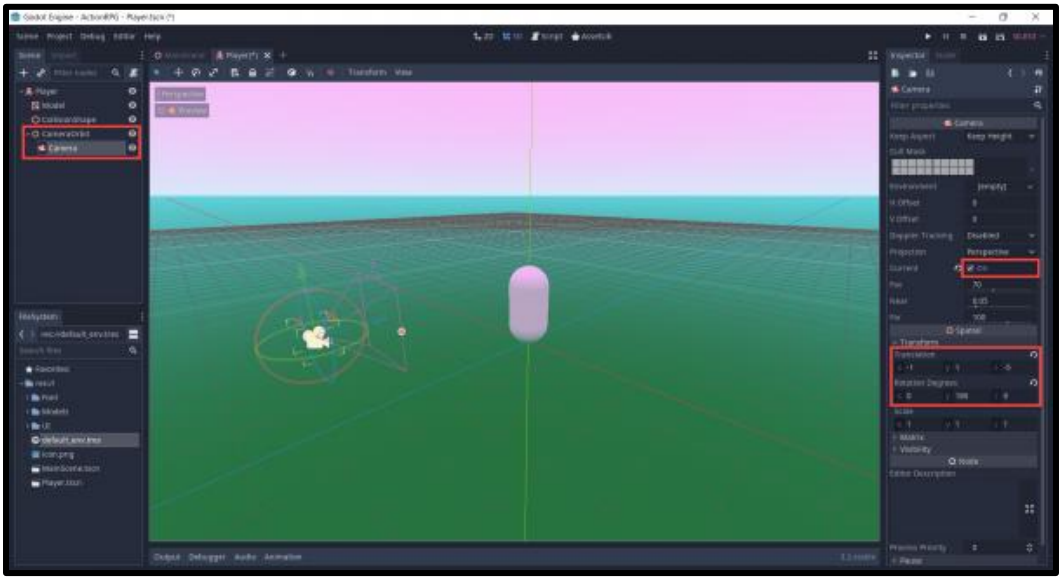
Create a new scene with a root node of **KinematicBody**. This is a node for physics objects you want to walk around like a player or enemy.

- As a child, create a new **MeshInstance** node
    - Set the **Mesh** to a *Capsule*
    - Set the mesh **Radius** to *0.5*
    - Set the **Translation** to *0, 1, 0*
    - Set the **Rotation Degrees** to *90, 0, 0*
- Another child is the **CollisionShape** node
    - Have the same properties as the mesh node



For the camera, we're going to have a center node with the camera as the child.

- Create a new Spatial node and rename it to **CameraOrbit**
- Set the **Translation** to *0, 1, 0*
- As a child of that, create a new **Camera** node
- Enable **Current**
- Set the **Translation** to *-1, 1, -5*
- Set the **Rotation Degrees** to *0, 180, 0*

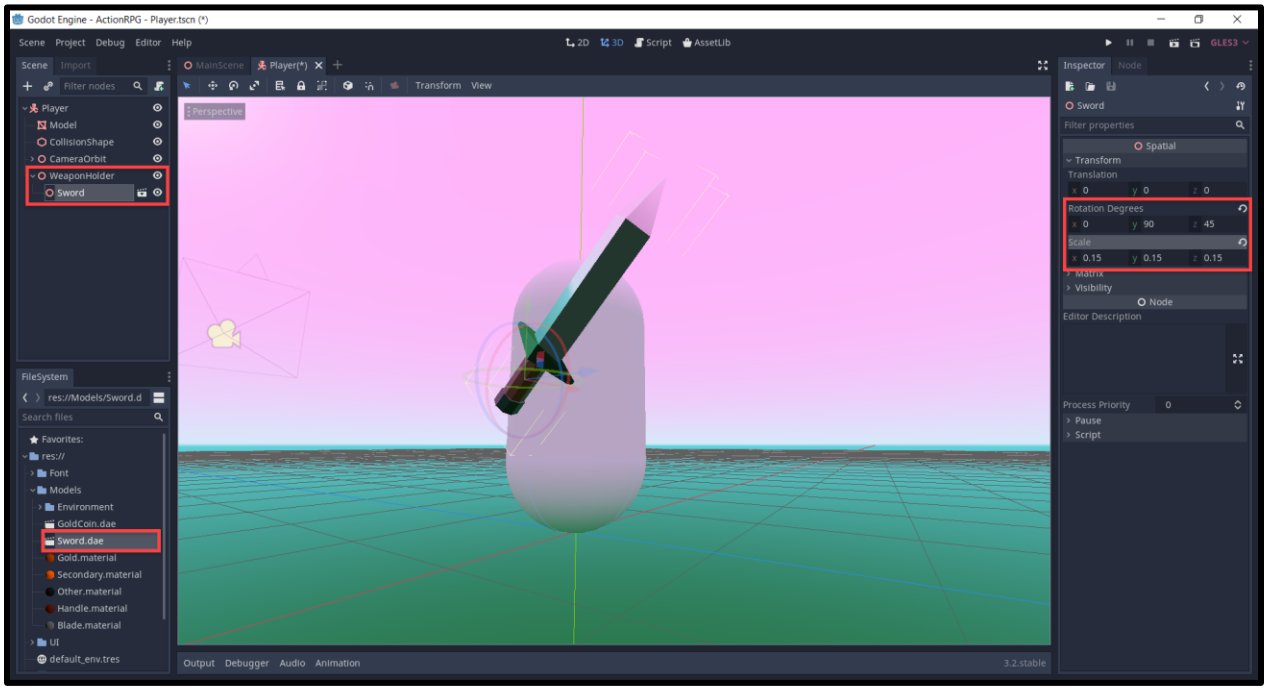


For the sword, we're going to create a Spatial node called **WeaponHolder**. This will hold the sword model.

- Set the **Translation** to *-0.58, 1, 0.035*

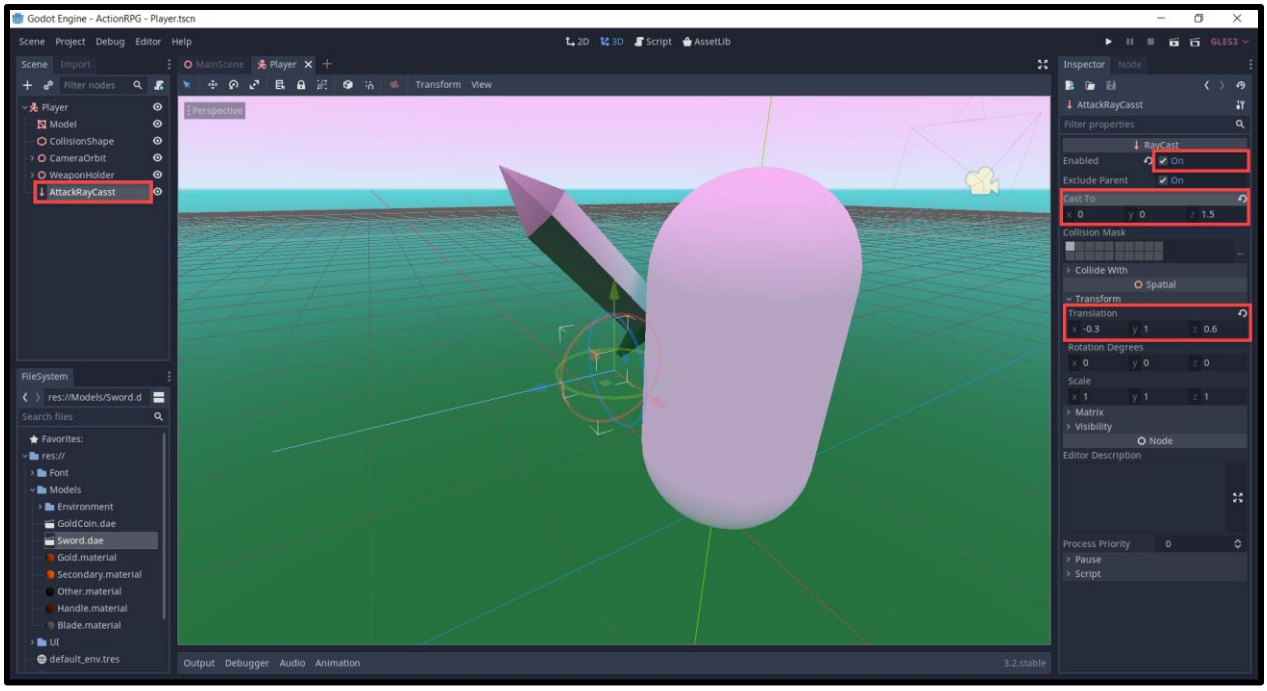
As a child of this node, drag in the **Sword.dae** model.

- Set the **Rotation Degrees** to *0, 90, 45*
- Set the **Scale** to *0.15, 0.15, 0.15*

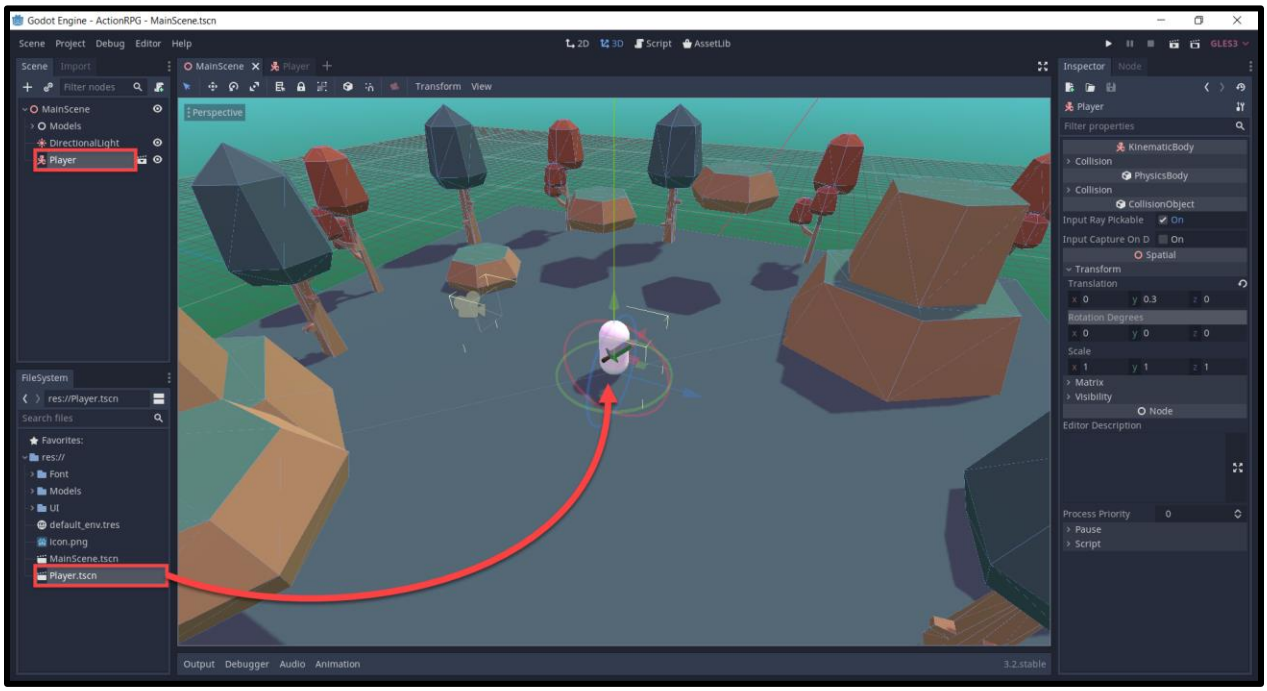


For when we want to attack enemies, we'll need to create a **RayCast** node. Rename it to **AttackRayCast**. This shoots a point from a certain position in a direction and we can get info on what it hits.

- Enable **Enabled** so that the ray will work
- Set the **Cast To** to *0, 0, 1.5*
- Set the **Translation** to *-0.3, 1, 0.6*



Back in the **MainScene**, let's drag in the **Player** scene.



Camera Look

Now that we've got our player object, let's start to script the camera look system. In the **Player** scene, select the **CameraOrbit** node and create a new script called **CameraOrbit**. We can start with the variables.

```
1  # look stats
2  var lookSensitivity : float = 15.0
3  var minLookAngle : float = -20.0
4  var maxLookAngle : float = 75.0
5
6  # vectors
7  var mouseDelta = Vector2()
8
9  # components
10 onready var player = get_parent()
```

The _**input** function gets called when any input is detected (keyboard, mouse key, mouse movement, etc). We want to check if the mouse moved, and if so - set the mouse delta.

```
1  # called when an input is detected
2  func _input (event):
3
4      # set "mouseDelta" when we move our mouse
5      if event is InputEventMouseMotion:
6          mouseDelta = event.relative
```

Inside of the _**process** function (called every frame), we're going to rotate the camera vertically and rotate the player horizontally. Rotating the player separately makes it easier later on to figure out movement directions.

```gdscript
# called every frame
func _process (delta):

    # get the rotation to apply to the camera and player
    var rot = Vector3(mouseDelta.y, mouseDelta.x, 0) * lookSensitivity * delta

    # camera vertical rotation
    rotation_degrees.x += rot.x
    rotation_degrees.x = clamp(rotation_degrees.x, minLookAngle, maxLookAngle)

    # player horizontal rotation
    player.rotation_degrees.y -= rot.y

    # clear the mouse movement vector
    mouseDelta = Vector2()
```

Finally, we can lock and hide the mouse cursor in the _**ready** function.

```gdscript
# called when the node is initialized
func _ready ():

    # hide the mouse cursor
    Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
```

Now if we press play, it's going to ask us to select a base scene. Choose the **MainScene** then we can test out the camera orbit system.

Scripting the Player

Let's now get working on our player. Create a new script called **Player** attached to the Player node. We can start with the variables.

```gdscript
# stats
var curHp : int = 10
var maxHp : int = 10
var damage : int = 1

var gold : int = 0

var attackRate : float = 0.3
var lastAttackTime : int = 0

# physics
var moveSpeed : float = 5.0
var jumpForce : float = 10.0
var gravity : float = 15.0

var vel = Vector3()

# components
onready var camera = get_node("CameraOrbit")
onready var attackCast = get_node("AttackRayCast")
```

Now we're going to be detecting keyboard inputs and basing the movement on that. In order to get these inputs though, we need to create actions. Let's go to the **Project Settings** window *(Project > Project Settings...)* then go to the **Input Map** tab.

Here, we want to enter in an action name and add it. Then assign a key to that action. We need 6 different actions.

- move_forwards
  - Key = **W**
- move_backwards
  - Key = **S**

- move_left
    - Key = **A**
- move_right
    - Key = **D**
- jump
    - Key = **Space**
- attack
    - Mouse Button = **Left Button**

Back in our script, we can create the **_physics_process** function. This is built into Godot and gets called 60 times a second - this is good for doing physics calls.

```
1 # called every physics step (60 times a second)
2 func _physics_process (delta):
```

Inside of this function, we're first going to reset the x and z axis' of our velocity vector and create a new vector to store our inputs.

```
vel.x = 0
vel.z = 0

var input = Vector3()
```

Then we can detect our movement actions and modify the input vector.

```
# movement inputs
if Input.is_action_pressed("move_forwards"):
    input.z += 1
if Input.is_action_pressed("move_backwards"):
    input.z -= 1
if Input.is_action_pressed("move_left"):
    input.x += 1
if Input.is_action_pressed("move_right"):
    input.x -= 1
```

With our input vector, we want to normalize it. This means resizing the vector to a magnitude (x + y + z = magnitude) of 1. Because when we move forward, our input vector is (0, 0, 1) with a magnitude of 1. Yet when we move diagonally (1, 0, 1) we have a magnitude of 2. We move faster diagonally, so reducing our diagonal input vector to something like (0.5, 0, 0.5) will maintain a magnitude of 1.

```
# normalize the input vector to prevent increased diagonal speed
input = input.normalized()
```

Now since we can look around and rotate, we don't want to move along the global direction. We need to find our local direction.

```
# get the relative direction
var dir = (transform.basis.z * input.z + transform.basis.x * input.x)
```

Next, we can apply this to our velocity vector.

```
# apply the direction to our velocity
vel.x = dir.x * moveSpeed
vel.z = dir.z * moveSpeed
```

So we can move horizontally, but what about gravity?

```
# gravity
vel.y -= gravity * delta
```

Along with gravity, we also want the ability to jump.

```
# jump input
if Input.is_action_pressed("jump") and is_on_floor():
    vel.y = jumpForce
```

Finally with everything calculated, let's actually move the player.

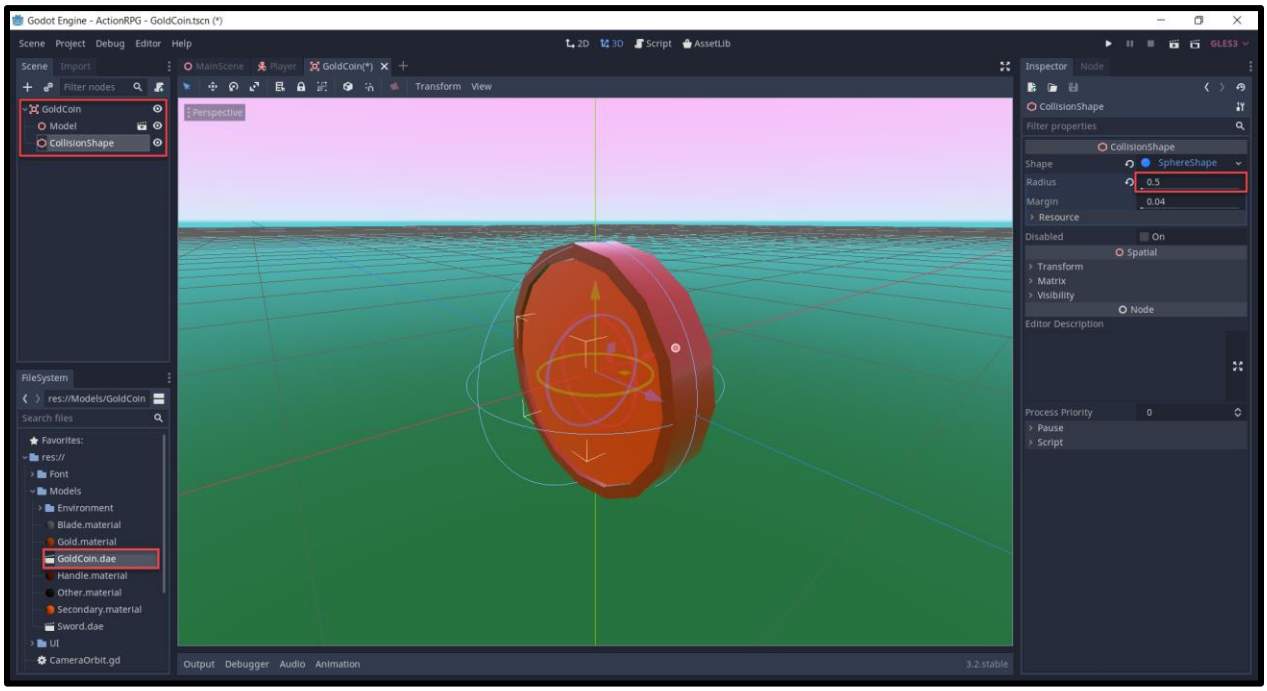
```
# move along the current velocity
vel = move_and_slide(vel, Vector3.UP)
```

We can now press play and test it out!

Gold Coins

Next up, we're going to create a coin object which the player can collect.

1. Create a new scene with a root node of **Area**
2. Rename the node to **GoldCoin**
3. Save it to the file system
4. Drag in the **GoldCoin** model as a child
5. Set the **Scale** to *0.5, 0.5, 0.5*
6. Create a new **CollisionShape** node
7. Set the **Radius** to *0.5*



Attached to the Area node, create a new script called **GoldCoin**. First, we can work on our two variables.

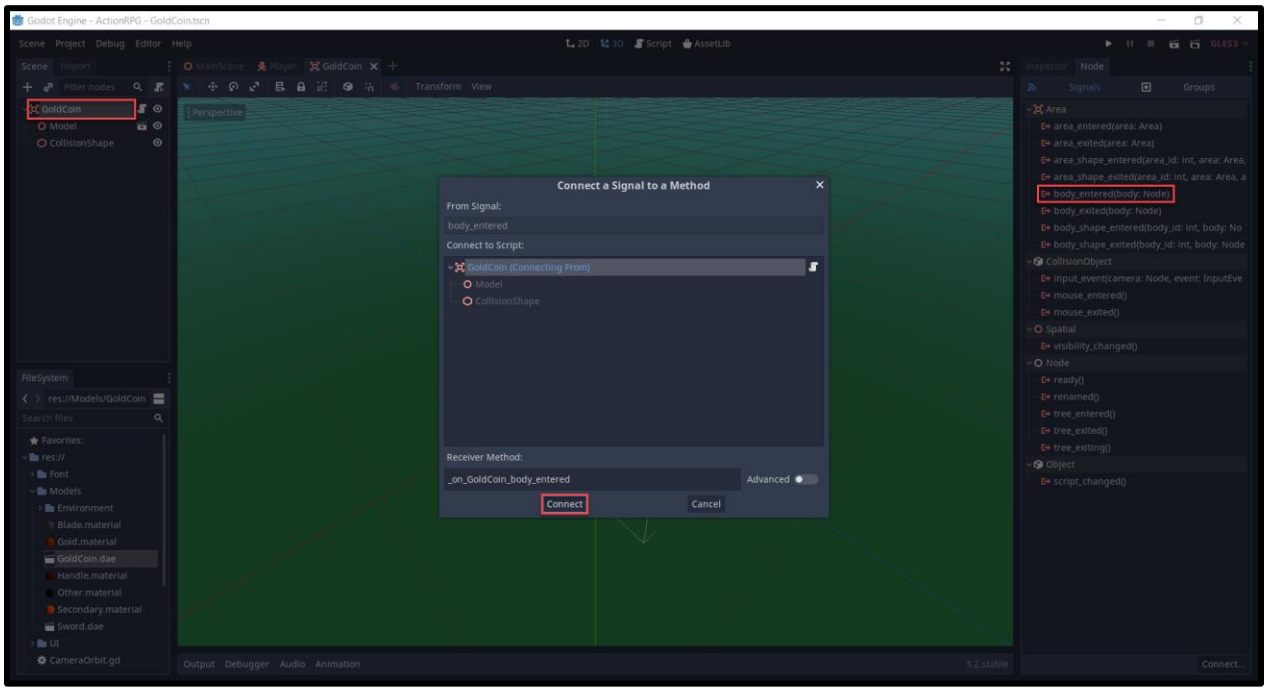
```
1 export var goldToGive : int = 1
2 var rotateSpeed : float = 5.0
```

Inside of the _**process** function which gets called every frame, we'll rotate the coin along its Y axis.

```
1  # called every frame
2  func _process (delta):
3
4      # rotate along the Y axis
5      rotate_y(rotateSpeed * delta)
```

In order to detect when the player has entered the coin, we need to connect to a signal.

1. Select the GoldCoin node
2. In the **Node** tab, double click on the **body_entered** signal
3. Click **Connect** and you should see that a new function is created in the script



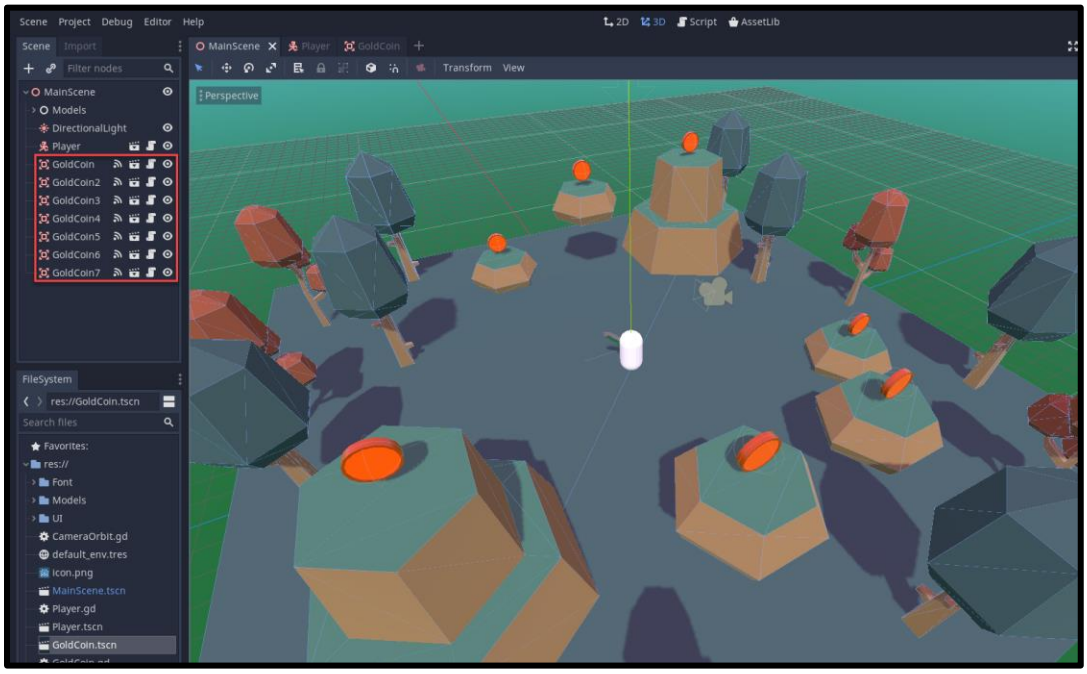
This function gets called when another body enters the coin collider. What we want to do is check to see if the body is the player. If so, call the **give_gold** function (which we'll create after) and then destroy the node with **queue_free**.

```gdscript
# called when a body enters the coin collider
func _on_GoldCoin_body_entered (body):

    # is this the player? If so give them gold
    if body.name == "Player":
        body.give_gold(goldToGive)
        queue_free()
```

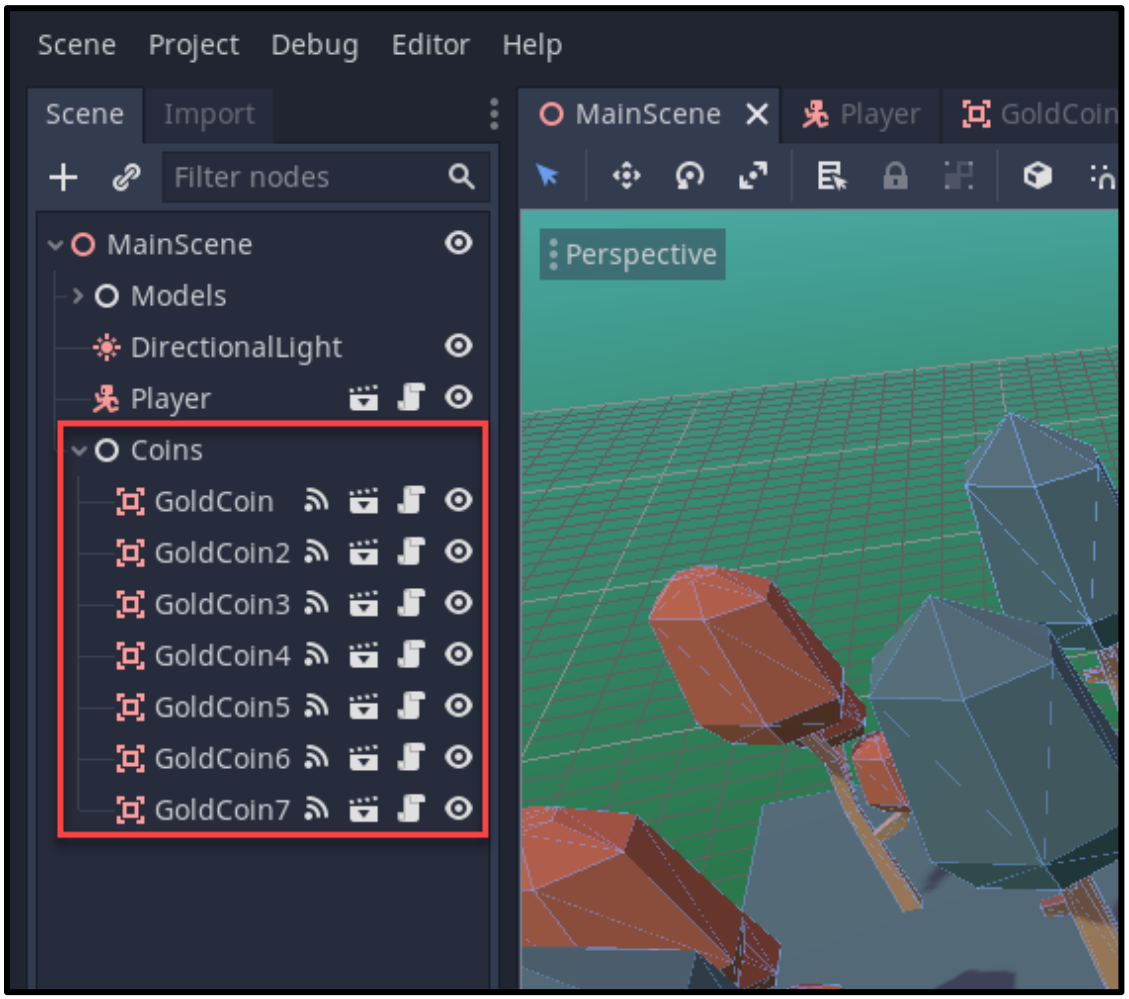
Let's now create the **give_gold** function over in the **Player** script.

```gdscript
# called when we collect a coin
func give_gold (amount):

    gold += amount
```

Back in our **MainScene**, let's drag in the GoldCoin scene. Duplicate it multiple times to create a number of different instances and position them around.



Just like with the models, let's create a new empty **Node** and make the coins a child of it. This will act as a sort of container to make the scene hierarchy less cluttered.



Continued in Part 2

So far, we've created a player with a sword, scripted our player to move, and even set up our various coins that the player will try to collect. To boot, we also set up our camera so that it can follow the player. While some aspects certainly are playable in this state, our 3D action RPG is definitely not complete! After all, what is an RPG without some enemies and obstacles to face off against?

In Part 2, we'll finish up this 3D action RPG project by setting up our enemies, animating our sword for combat, and implementing our UI. In so doing, we'll have a solid foundation for an RPG that can be expanded upon as you wish! Until the next part!

Develop a 3D Action RPG in Godot - Part 2

Introduction

Welcome back to this tutorial series where we're building a 3D action RPG from scratch with Godot. In Part 1, we covered a lot of ground with how to make an action RPG with Godot, including:

- Creating the project
- Setting up the environment
- Implementing the third-person player controller
- Building collectible gold coins

While these elements certainly create a great foundation for our game, we'll want to add just a bit more to create that true, action RPG feel. Thus, in Part 2, we'll be finishing our project by adding enemies and combat mechanics - including a nifty animation for our sword. To boot, we'll also dive into the Godot UI so that you can set up a simple way to track the player's health and gold.

We hope you're ready to dive in and create this impressive piece for your portfolio!

Project Files

In this course, we're going to be using a few models and a font to make the game look nice. You can, of course, choose to use your own, but we're going to be designing the game with these specific ones in mind. The models are from kenney.nl, a good resource for public-domain game assets. Then we're getting our font from Google Fonts.

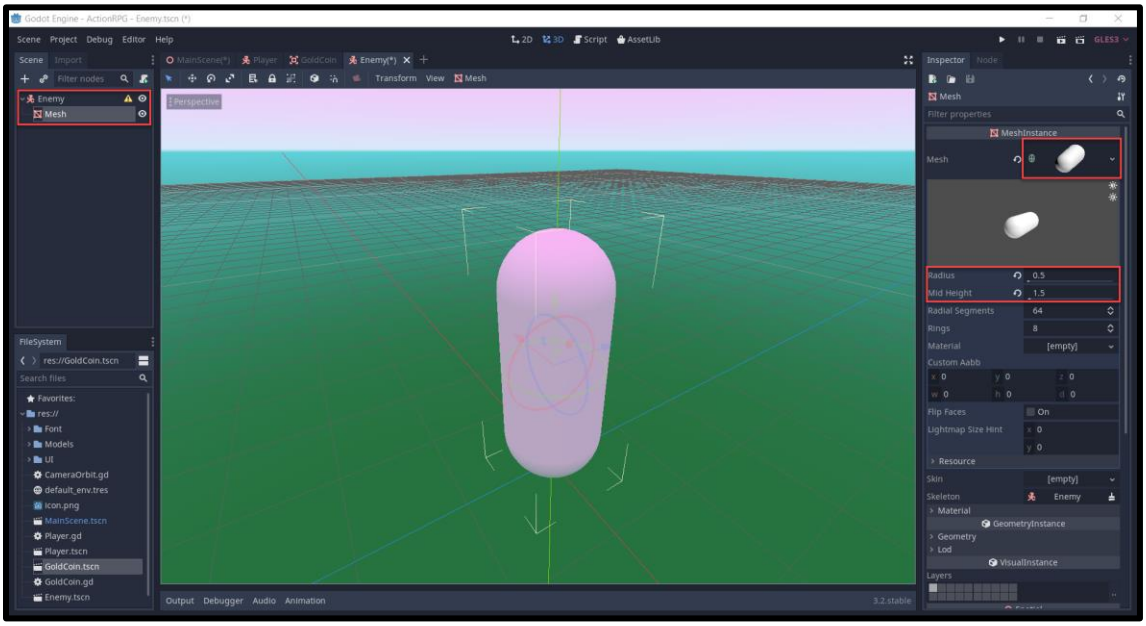
You can download the source code and assets here.

Creating the Enemy

Let's now create the enemy. They will chase after the player and damage them at a specified distance. To begin, let's create a new scene with a root node of **KinematicBody**.

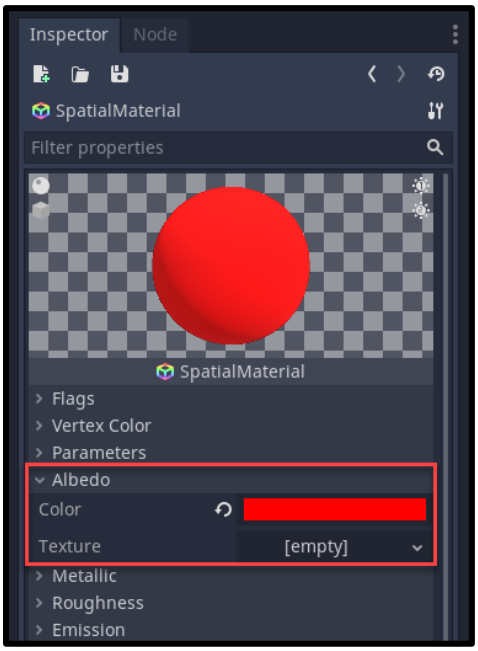
1. Rename the node to **Enemy**
2. Save the scene
3. As a child, create a new **MeshInstance** node (rename it to **Mesh**)
4. Set the **Mesh** to *CapsuleShape*
5. Set the **Radius** to *0.5*
6. Set the **Mid Height** to *1.5*

7. Set the **Transform > Translation** to *0, 1.25, 0*
8. Set the **Transform > Rotation Degrees** to *90, 0, 0*



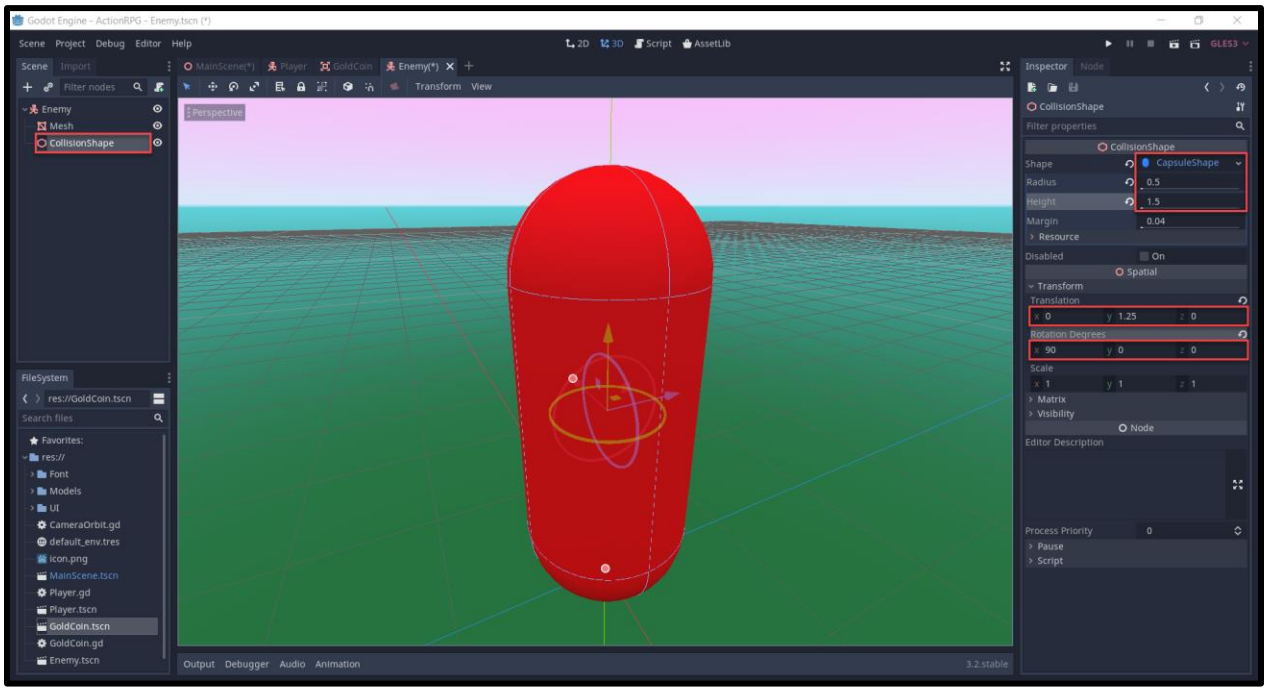
Let's differ this capsule from our player. Just under where we set the radius. we can create a new **SpatialMaterial**. Click on that to open up the material properties.

- Set the **Albedo > Color** to *red*

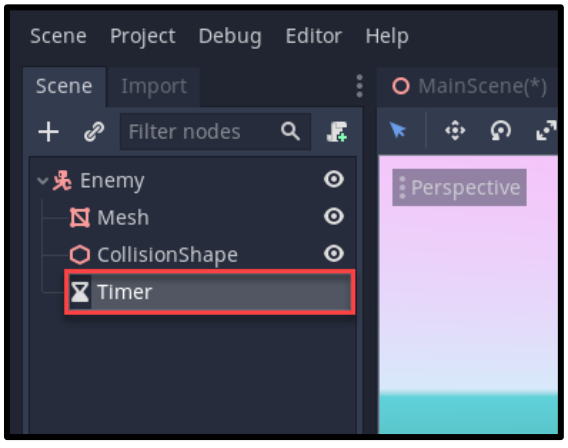


Next, we can create a **CollisionShape** node with the same properties as the mesh.

1. Set the **Shape** to *CapsuleShape*
2. Set the **Radius** to *0.5*
3. Set the **Height** to *1.5*
4. Set the **Translation** to *0, 1.25, 0*
5. Set the **Rotation Degrees** to *90, 0, 0*



Finally, we're going to create a **Timer** node which can send out a signal every certain amount of seconds. This will be setup in-script.



Scripting the Enemy

Now, we're going to create the enemy script. Select the **Enemy** node and create a new script called **Enemy**. First, we want to enter in our variables.

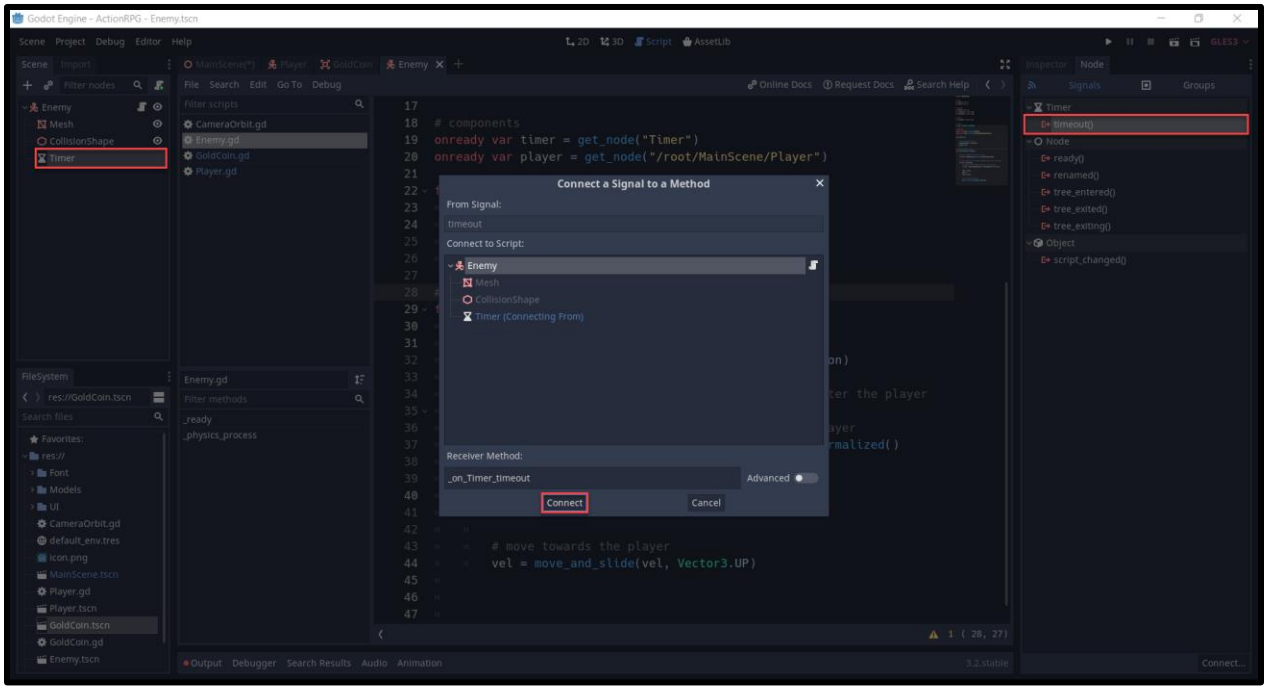
```
# stats
var curHp : int = 3
var maxHp : int = 3

# attacking
var damage : int = 1
var attackDist : float = 1.5
var attackRate : float = 1.0

# physics
var moveSpeed : float = 2.5

# vectors
var vel : Vector3 = Vector3()

# components
onready var timer = get_node("Timer")
onready var player = get_node("/root/MainScene/Player")
```

First, we're going to create the _**ready** function which gets called once the node is initialized. Inside of here, we're going to set the timer wait time and start it.

```gdscript
func _ready ():

    # set the timer wait time
    timer.wait_time = attackRate
    timer.start()
```

But nothing will happen right now because the timer isn't connected to anything. So select the node and in the **Node** tab, we want to connect the **timeout()** signal to the script.

1. Double click the **timeout** signal
2. Click **Connect**
3. Now you should see that there's a new function in the enemy script



With the new function, let's check our distance to the player and damage them. We'll create the **take_damage** function on the player soon.

```gdscript
# called every "attackRate" seconds
func _on_Timer_timeout ():

    # if we're within the attack distance - attack the player
    if translation.distance_to(player.translation) <= attackDist:
        player.take_damage(damage)
```

Next, let's implement the ability for the enemy to chase after the player when further than the attack distance.

```gdscript
# called 60 times a second
func _physics_process (delta):

    # get the distance from us to the player
    var dist = translation.distance_to(player.translation)

    # if we're outside of the attack distance, chase after the player
    if dist > attackDist:
        # calculate the direction between us and the player
        var dir = (player.translation - translation).normalized()

        vel.x = dir.x
        vel.y = 0
        vel.z = dir.z

        # move towards the player
        vel = move_and_slide(vel, Vector3.UP)
```

Let's finish of the enemy script with the **take_damage** and **die** functions. The take damage function will be called when the enemy is attacked by the player.

```gdscript
# called when the player deals damage to us
func take_damage (damageToTake):

    curHp -= damageToTake

    # if our health reaches 0 - die
    if curHp <= 0:
        die()

# called when our health reaches 0
func die ():

    # destroy the node
    queue_free()
```

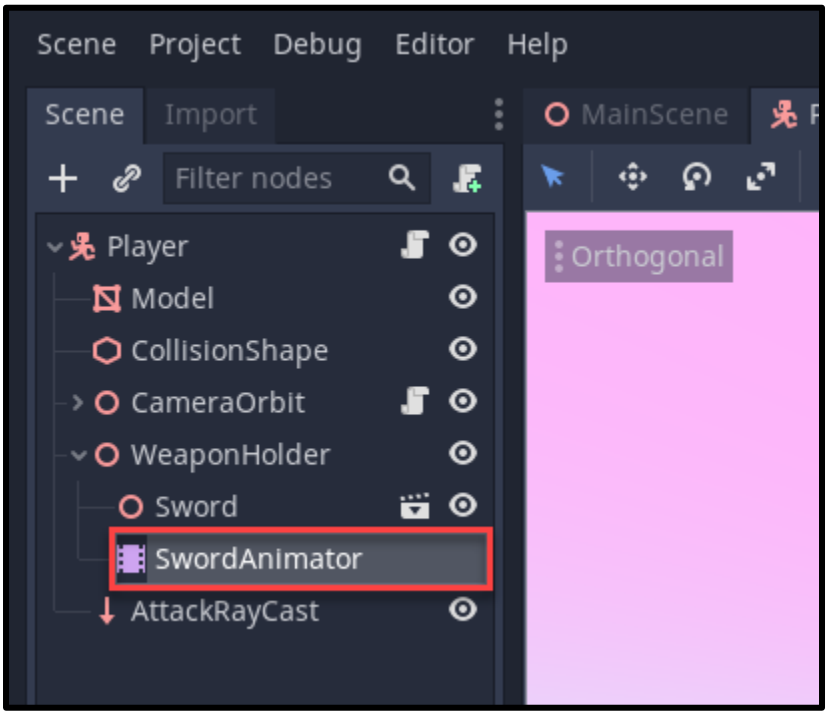
Our enemy is now finished. We've got all the functions setup, so let's now go over to the **Player** script and create the functions which get called by an attacking enemy.

```gdscript
# called when an enemy deals damage to us
func take_damage (damageToTake):

    curHp -= damageToTake

    # if our health is 0, die
    if curHp <= 0:
        die()

# called when our health reaches 0
func die ():

    # reload the scene
    get_tree().reload_current_scene()
```

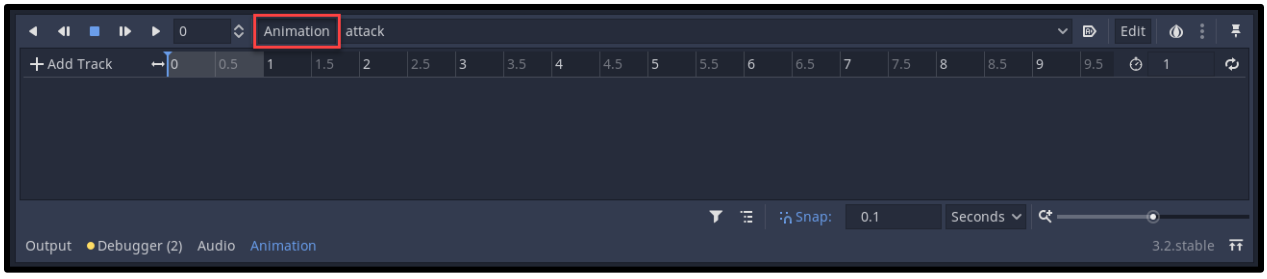
We can now go to the **MainScene** and drag the enemy scene in to create a new instance. Press play and the enemy should chase after you.

Sword Animation

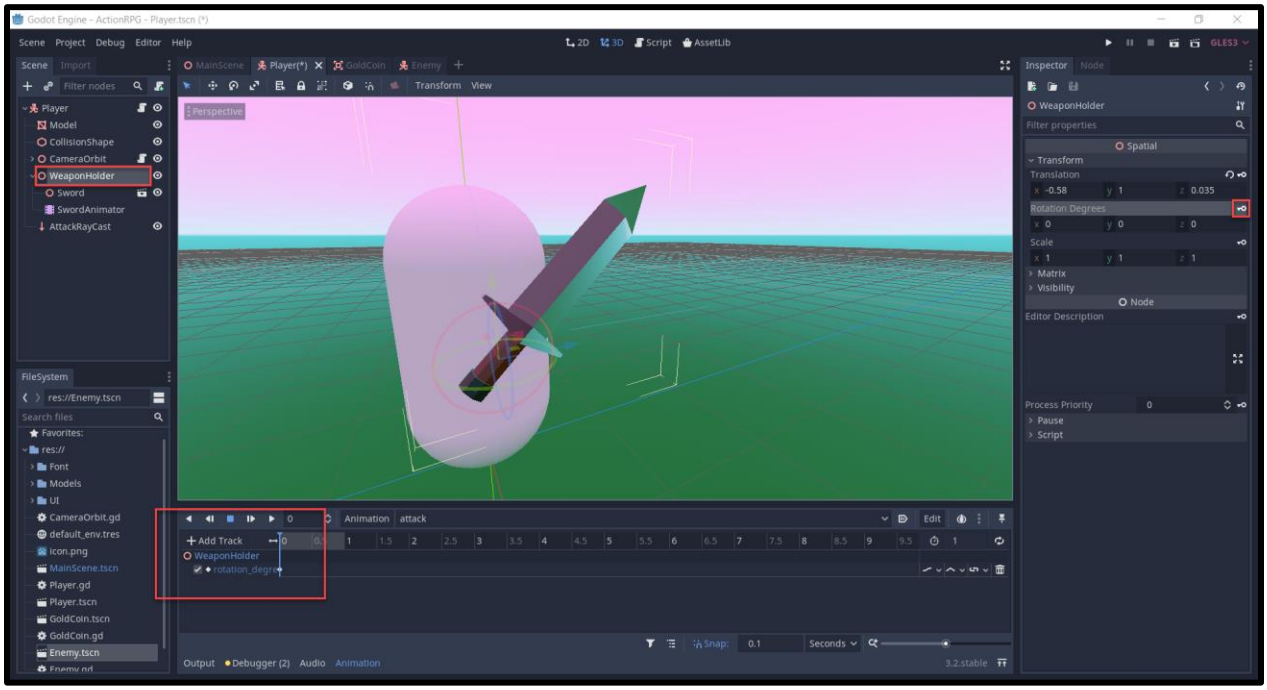
Now its time to implement the ability to attack enemies. Before we jump into scripting, let's create an animation for the player's sword. In the **Player** scene, right click on the **WeaponHolder** and create a new child node of type **AnimationPlayer**. Rename it to **SwordAnimator**.



Selecting the node should toggle the **Animation** panel at the bottom of the screen. Here, we want to click on the **Animation** button and create a new animation called **attack**.

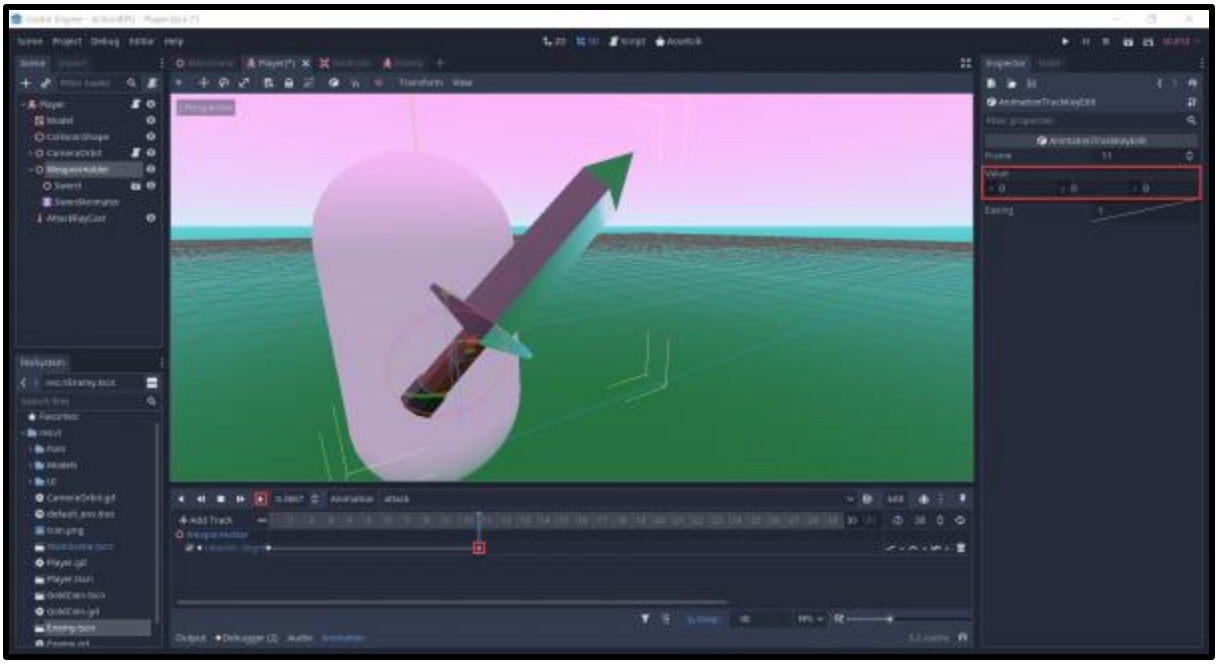


With the animator, we can choose which properties we want to animate. Select the **WeaponHolder**, and in the inspector select the key icon next to **Rotation Degrees**. Create that new track.

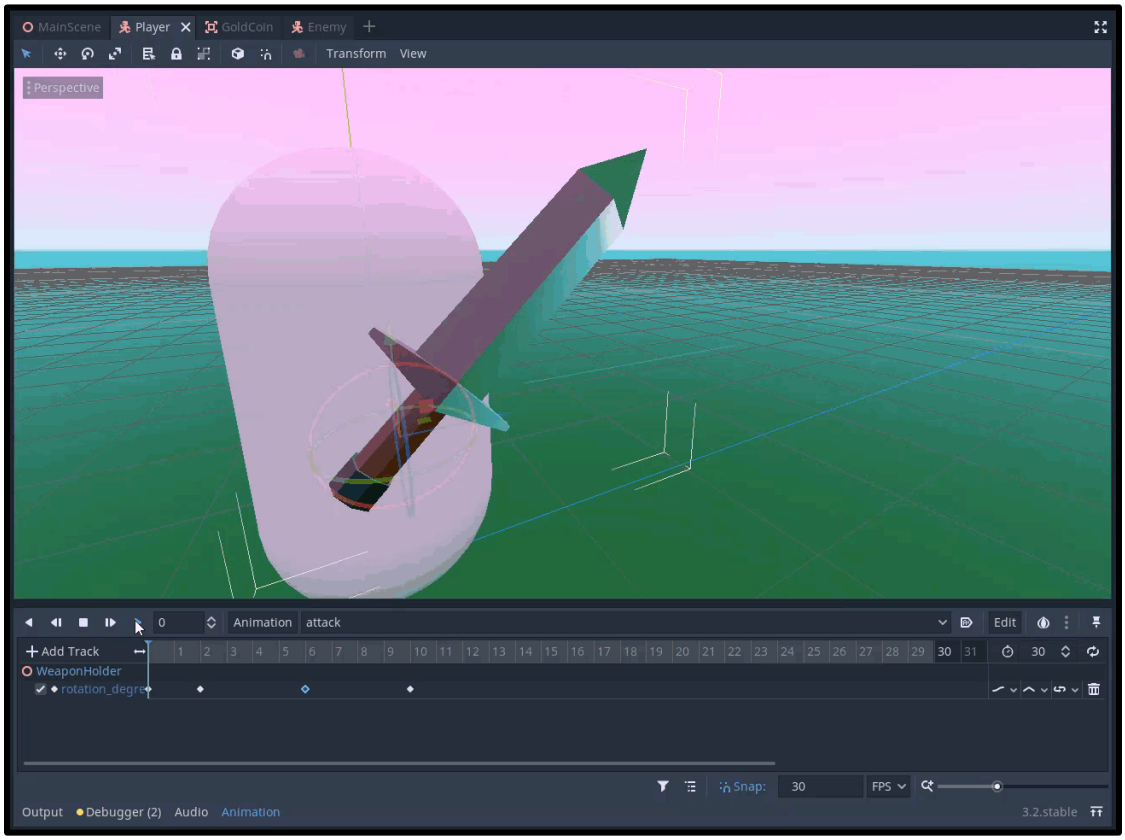


By default the FPS of the animation is quite low, so select the **Time** drop-down and change that to **FPS**. Set the FPS to **30**.

Now we can right-click on the tack and insert a new key. Selecting the key, we can change the rotation with the **Value** property.



Here's the attack animation I made.



Attacking the Enemy

Now that we have the attack animation, let's go ahead and start scripting it. In the **Player** script, let's create the _**process** function which gets called every frame.

```
1  func _process (delta):
2
3      # attack input
4      if Input.is_action_just_pressed("attack"):
5          try_attack()
```

The **try_attack** function will check to see if we can attack and if so, damage the enemy.

```
1  # called when we press the attack button
2  func try_attack ():
```

First, we want to see if we can attack based on the attack rate. We're getting the current time elapsed in milliseconds. That's why we're multiplying attackRate by 1000.

```
1  # if we're not ready to attack, return
2  if OS.get_ticks_msec() - lastAttackTime < attackRate * 1000:
3      return
```

Then we want to set the last attack time and play the animation.

```
1  # set the last attack time to now
2  lastAttackTime = OS.get_ticks_msec()
3
4  # play the animation
5  swordAnim.stop()
6  swordAnim.play("attack")
```

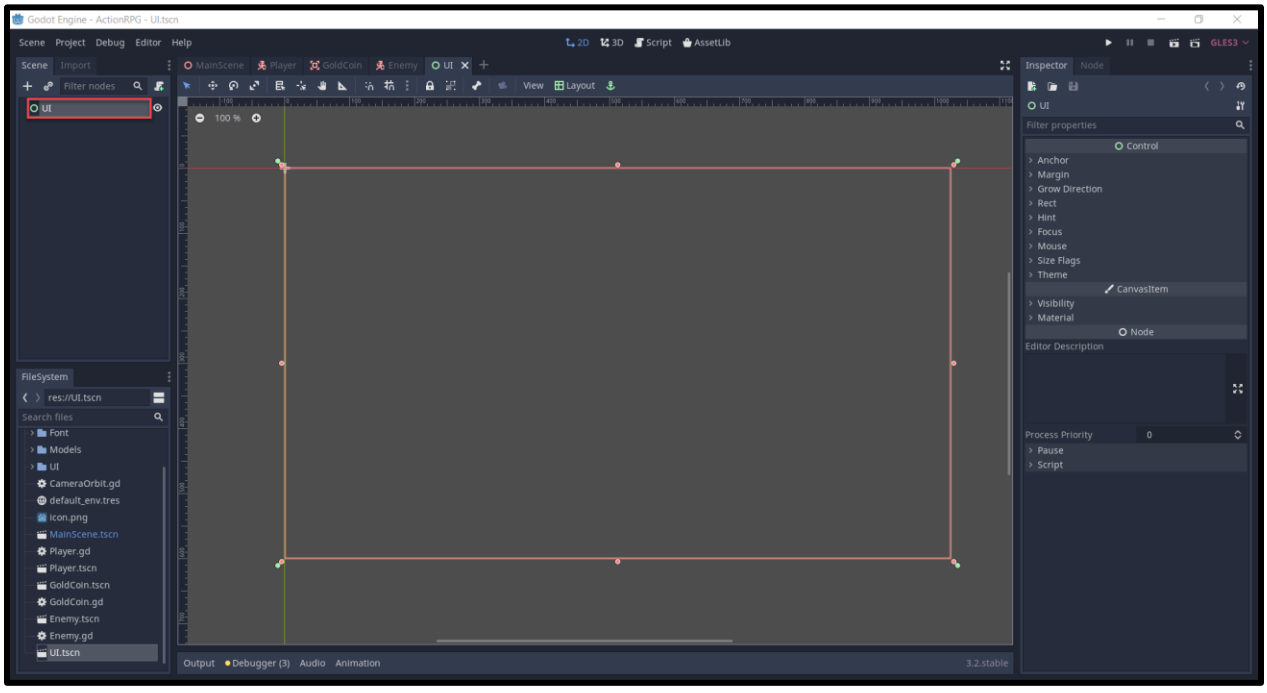
To actually attack the enemy, we need to check to see if the raycast is detecting an enemy.

```
# is the ray cast colliding with an enemy?
if attackCast.is_colliding():
    if attackCast.get_collider().has_method("take_damage"):
        attackCast.get_collider().take_damage(damage)
```

We can now press play and test it out. Try attacking the enemy and eventually they should disappear, showing that our system works.

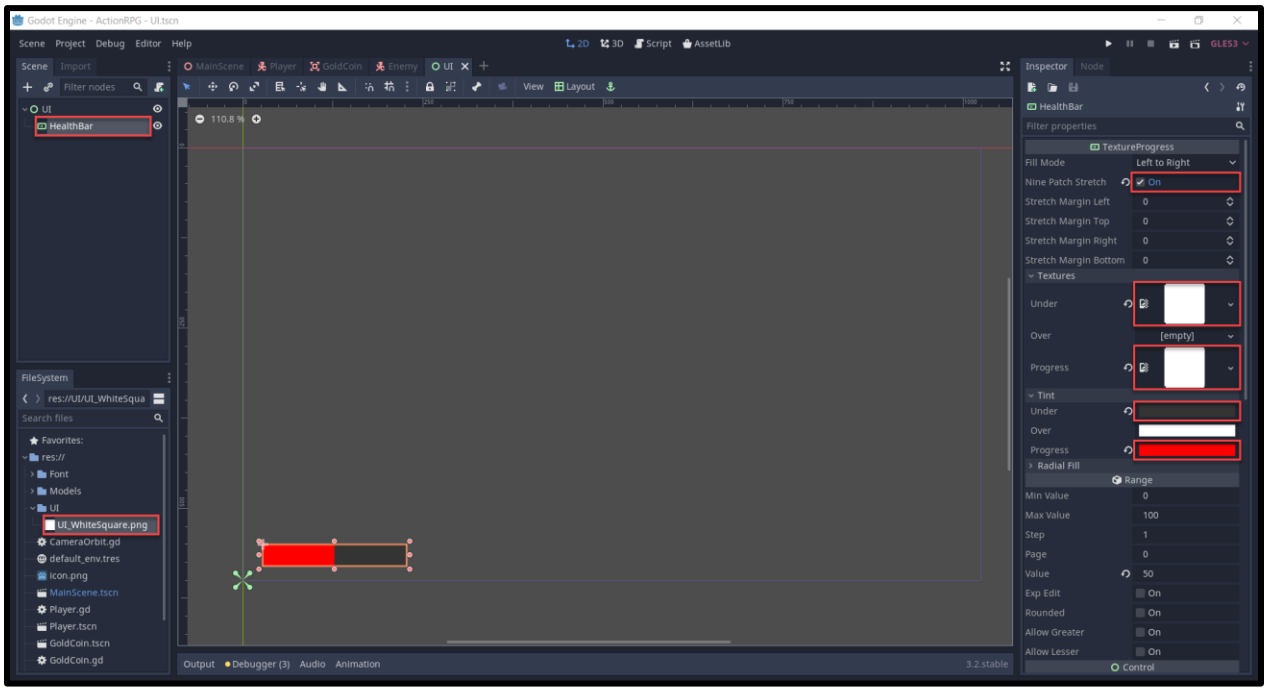
Creating the UI

Now that we have all of the systems in place, the final thing to do is implement the user interface. This will include a health bar and gold text. So to begin, let's create a new scene of type **user interface** (Control node). Rename it to **UI** and save the scene.



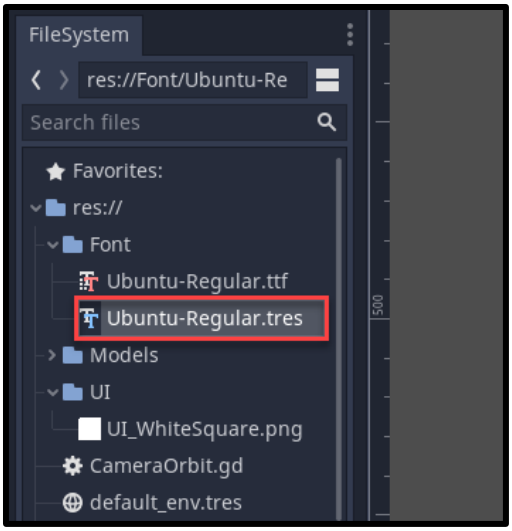
As a child, create a new **TextureProgress** node. This is a texture which can resize like a progress bar.

1. Rename the node to **HealthBar**
2. Enable **Nine Patch Stretch**
3. Set the **Under** and **Progress** textures to *UI_WhiteSquare.png*
4. Set the **Under** tint to *dark grey*
5. Set the **Progress** tint to *red*
6. Bring the anchor points (4 green pins) down to the bottom left of the screen
7. Drag the health bar down to the bottom left and resize it (drag on the circles to resize)



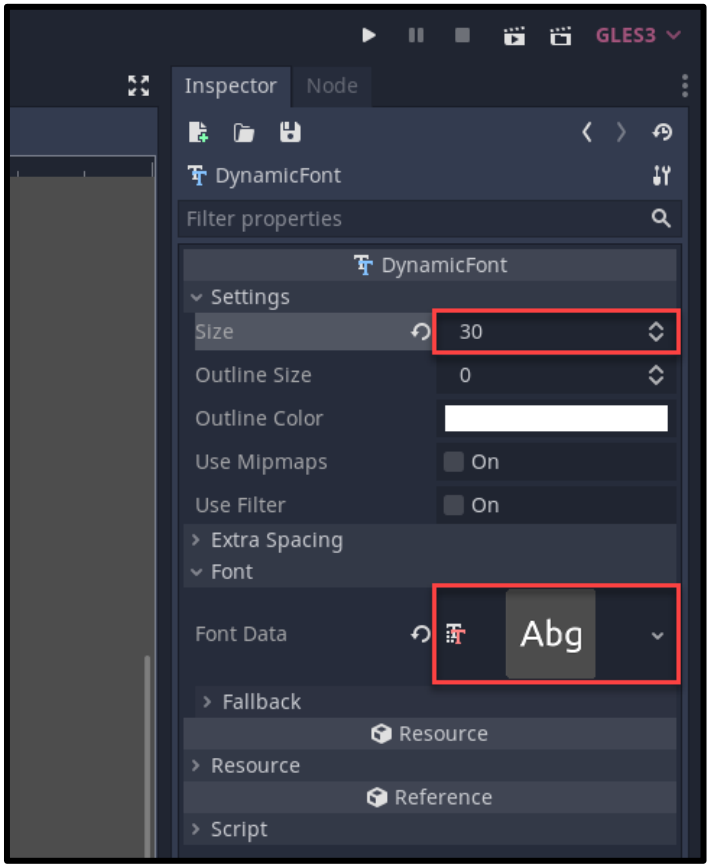
For our gold text, we're going to need a font as the default one isn't that versitile.

1. Right click on the .ttf file in the file system and select **New Resource...**
2. Search for and select **DyanamicFont**
3. This will create a new dynamic font resource



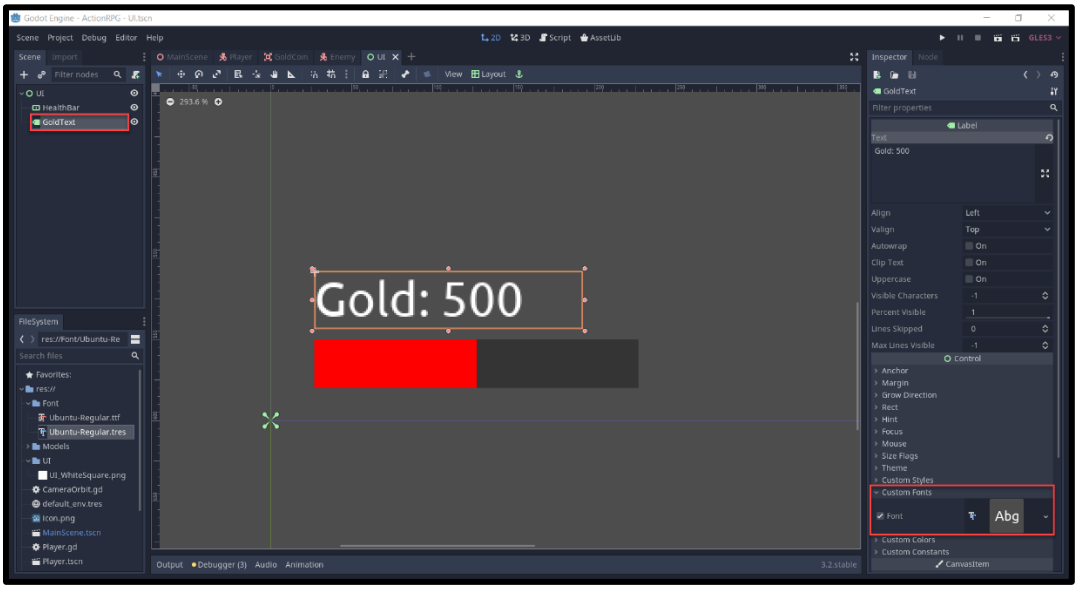
Double click the dynamic font, and in the inspector...

1. Set the **Size** to *30*
2. Drag the .ttf file into the **Font Data** property

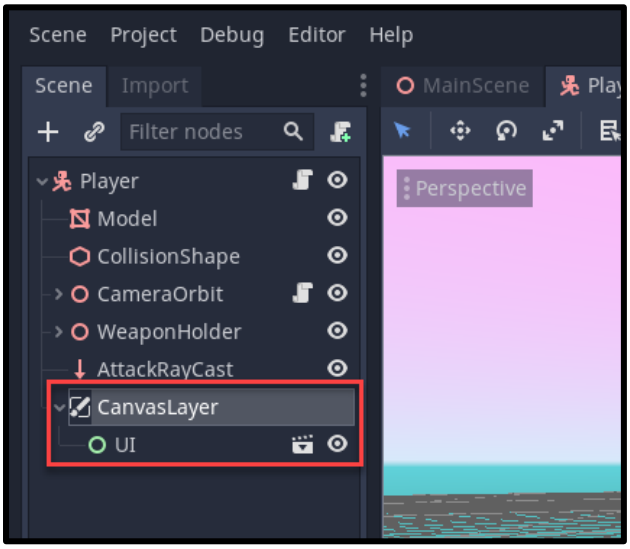


Now we can create a **Label** node and rename it to **GoldText**. This node will display text on-screen.

1. Drag it down above the health bar
2. Set the anchor points to be bottom-left of the screen
3. Drag the dynamic font asset into the **Custom Font** property on the label



Now that we have our UI, let's go to the **Player** scene and create a child node called **CanvasLayer**. This is a node which renders the control node children to the screen. So as a child of the canvas layer, drag in the **UI** scene.



If we press play, you'll see that the UI is rendering on the screen.

Scripting the UI

We've got our UI displaying on-screen, but it doesn't do anything just yet. In the **UI** scene, create a new script attached to the UI node called **UI**. We can start with the variables.

```
onready var healthBar = get_node("HealthBar")
onready var goldText = get_node("GoldText")
```

Then we can create the **update_health_bar** function. This will set the value of the texture progress.

```
# called when we take damage
func update_health_bar (curHp, maxHp):

    healthBar.value = (100 / maxHp) * curHp
```

Finally, the **update_gold_text** function will update the text of the gold text label.

```
# called when our gold changes
func update_gold_text (gold):

    goldText.text = "Gold: " + str(gold)
```

Now that we've got the **UI** script created, let's go to the **Player** script and hook these functions up.

Let's create a variable to reference the UI node.

```
onready var ui = get_node("CanvasLayer/UI")
```

Then inside of the _**ready** function (called when the node is initialized), we want to initialize the UI.

```
# called when the node is initialized
func _ready ():

    # initialize the UI
    ui.update_health_bar(curHp, maxHp)
    ui.update_gold_text(gold)
```

Inside of the **give_gold** function, we can update the gold text.

```
# update the UI
ui.update_gold_text(gold)
```

Inside of the **take_damage** function, we can update the health bar.

```
# update the UI
ui.update_health_bar(curHp, maxHp)
```

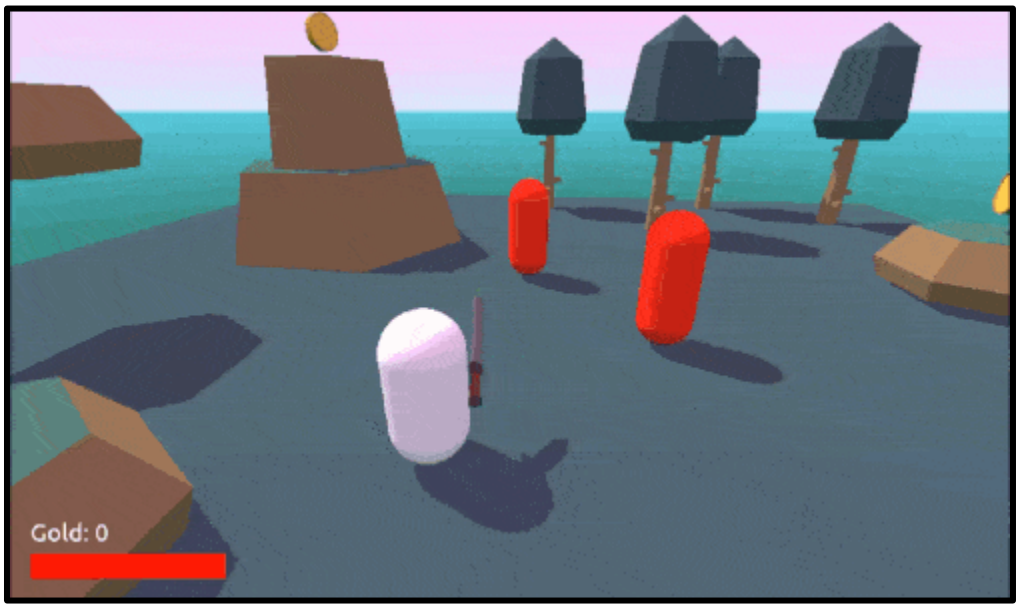
And there we go. You can press play now and see that the UI will update when we take damage and collect coins.

Conclusion

Congratulations on completing the tutorial! If you've been following along, you should now have a 3D action RPG playable within your Godot editor! Over the course of both Part 1 and Part 2, we learned how to create:

- A third-person player controller
- Enemies who chase and attack the player
- Player combat
- Collectible coins
- User interface

With that, you now have a fantastic project that you can share with your friends, add to your portfolio, or use as a base for your own projects. As you can imagine, these systems can be expanded in numerous ways, such as adding more collectibles, adding different enemies, and beyond. We encourage you to explore what Godot is capable of - but for now, we hope you enjoy this action RPG you've created with your own two hands!



Make a Strategy Game in Godot - Part 1

Introduction

Whether turn-based, real-time, or resource management based, strategy games remain cemented in the world as one of the most popular genres. This is why numerous beginning and experienced developers alike dream of creating their own strategy games and building a balanced, challenging gameplay experience. So, what if we told you that making your own strategy game was completely possible with Godot?

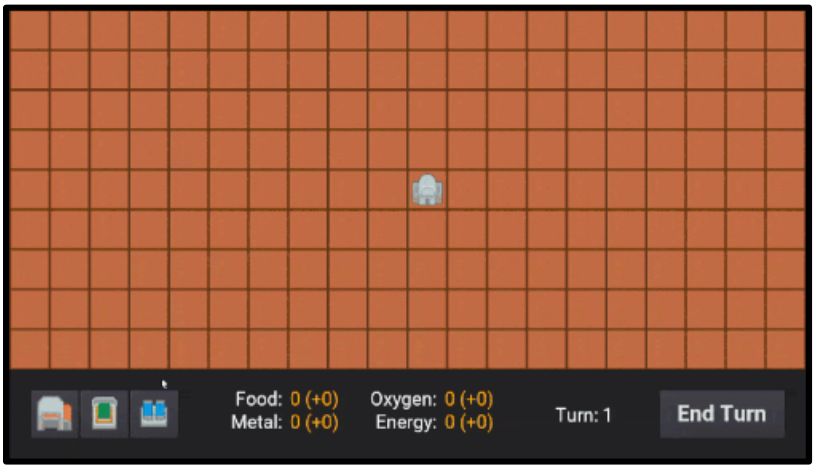
If that sounds exciting, then this is the tutorial for you! Through this tutorial, we're going to show you how to make a resource-based city building game with ease - all within the Godot game engine. You will learn several features common to strategy games such as:

- Making a turn-based system
- Having three different buildings which product resources
- Setting up a grid-based placement system
- Using a UI to show resources, turn, etc.

So, if you're ready, buckle down, turn on your computer, and get pumped to make a strategy game project with Godot.

Before we begin though, it's important to know that this tutorial will not go over the basics of Godot. If you're new to the engine, we recommend you read through our introductory tutorial on the subject!

Already know all this and want to jump into building placement and turn-based game flow? Try out Part 2 instead!



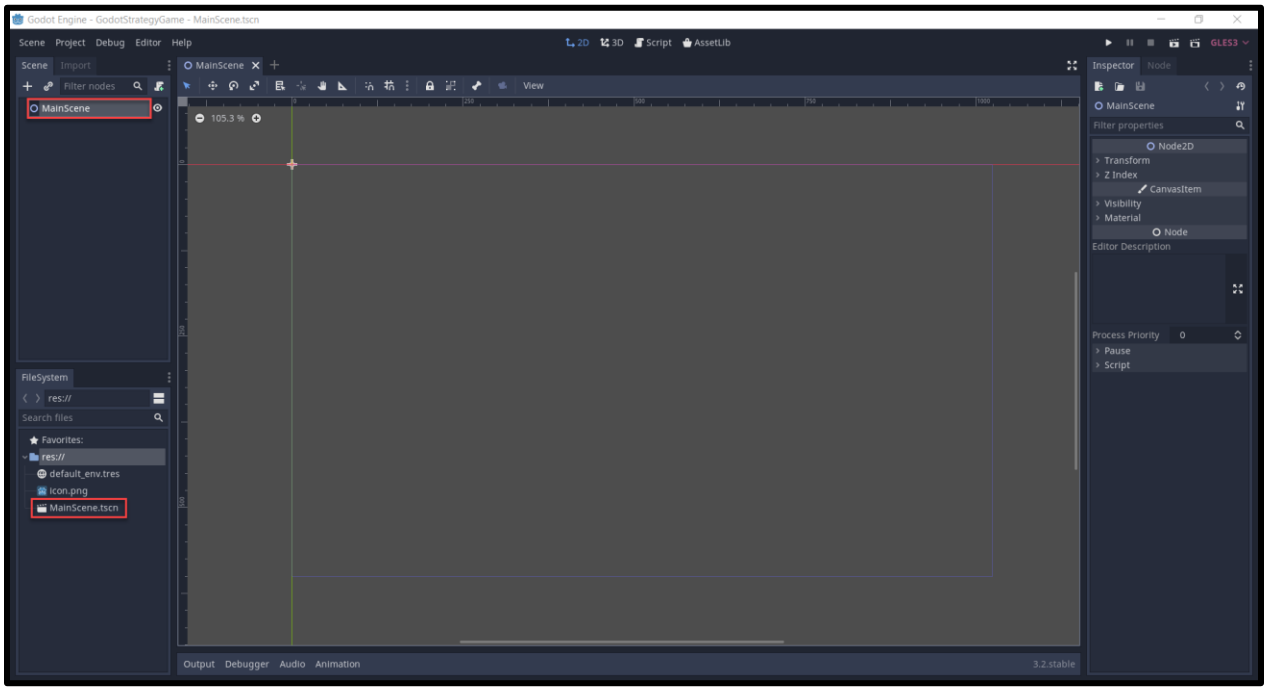
Project Files

In this tutorial, we'll be using some sprites from the kenney.nl website (an open domain game asset website) and fonts from Google Fonts. You can of course choose to use your own assets, but we'll be designing the game around these:

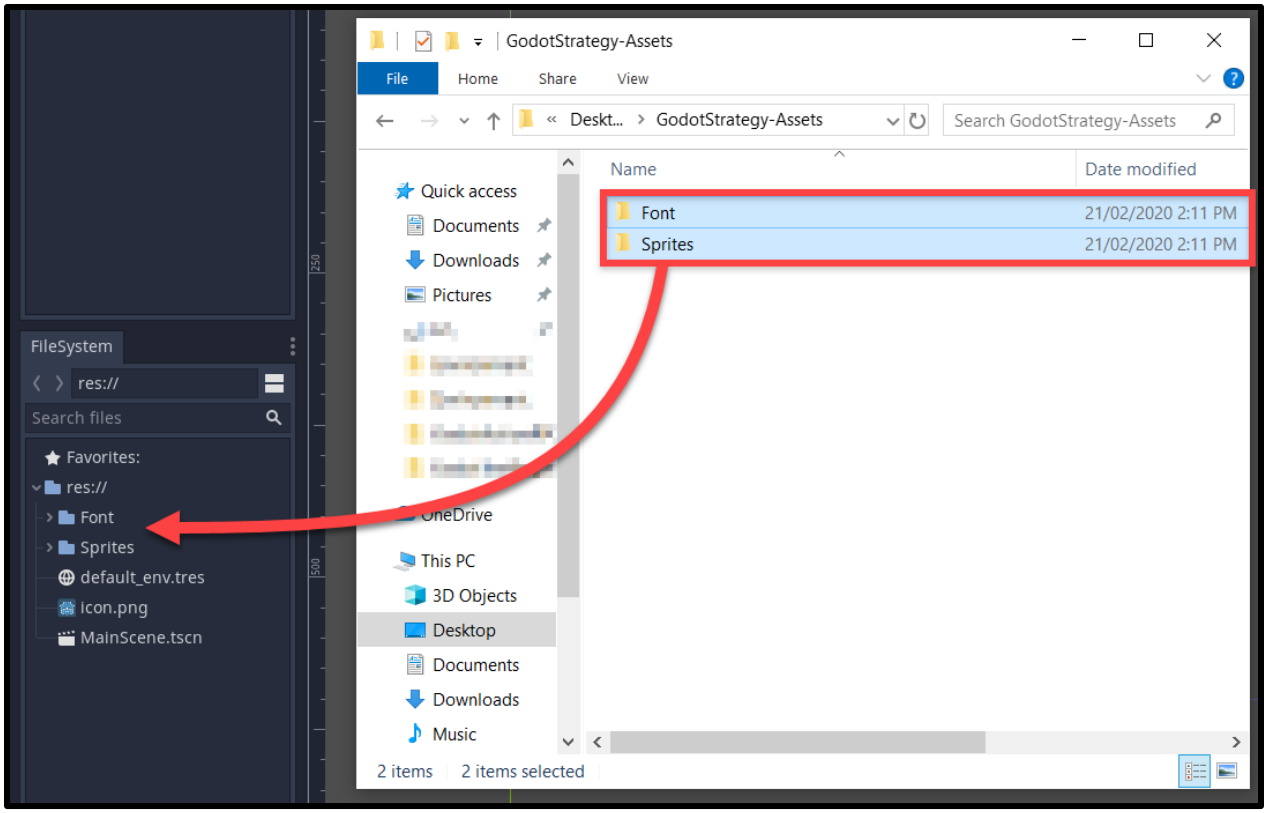
You can download the source code and assets here.

Setting up the Project

Create a new Godot project and for our first scene, we're going to choose **2D Scene** (Node2D node). Rename it to **MainScene** and save the scene to the file system.



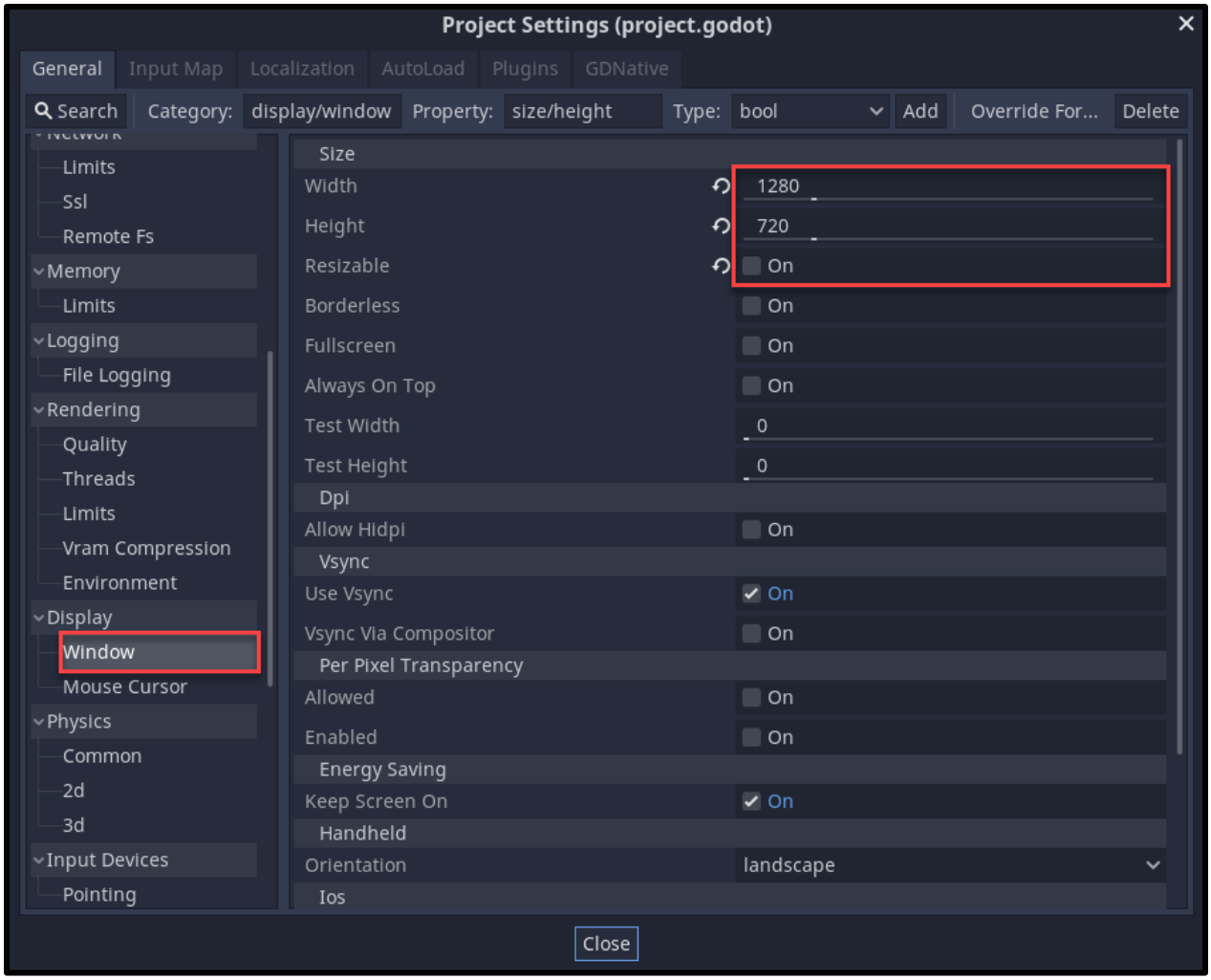
Let's also import our assets by dragging the two folders into the file system.



Something we also want to do is change the resolution of the screen. When we press play (you'll be asked to choose a base scene - select the MainScene), you'll see that the screen is pretty small. Due to our sprite size and the amount of tiles we're going to have, let's set the resolution.

Open the **Project Settings** window (*Project > Project Settings...*) and go down to the **Display > Window** tab.

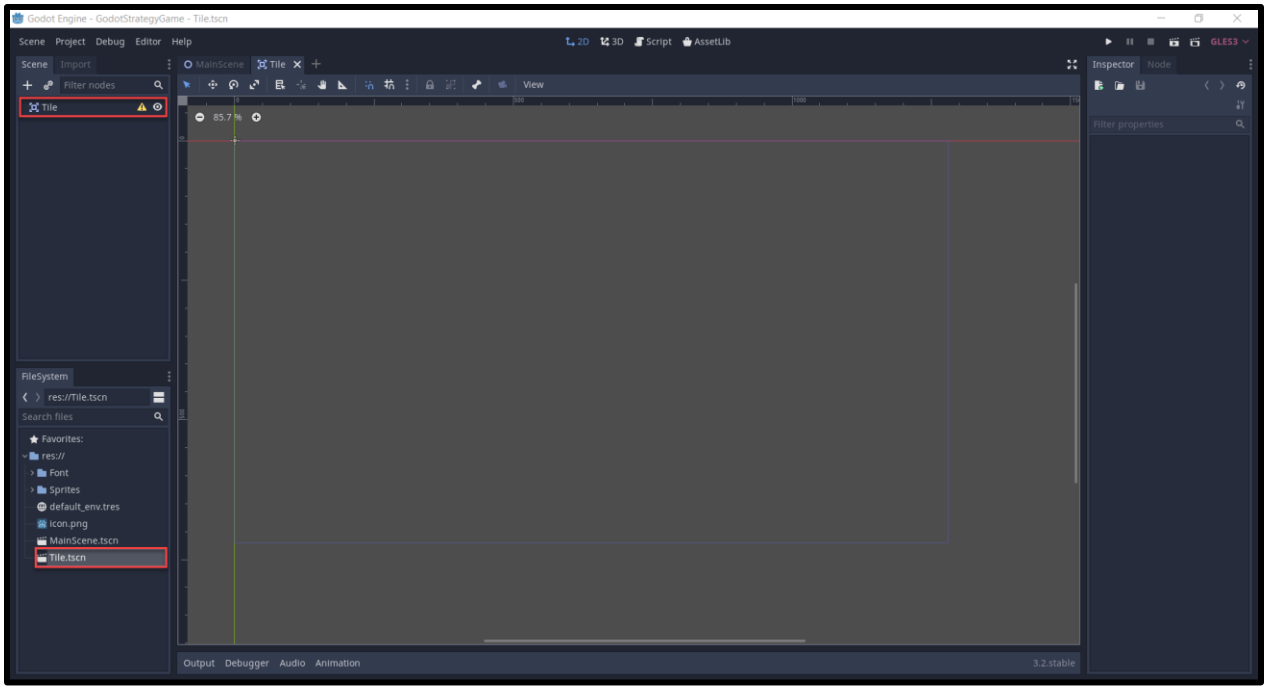
- Set the **Width** to *1280*
- Set the **Height** to *720*
- Disable **Resizable**



Creating the Tiles

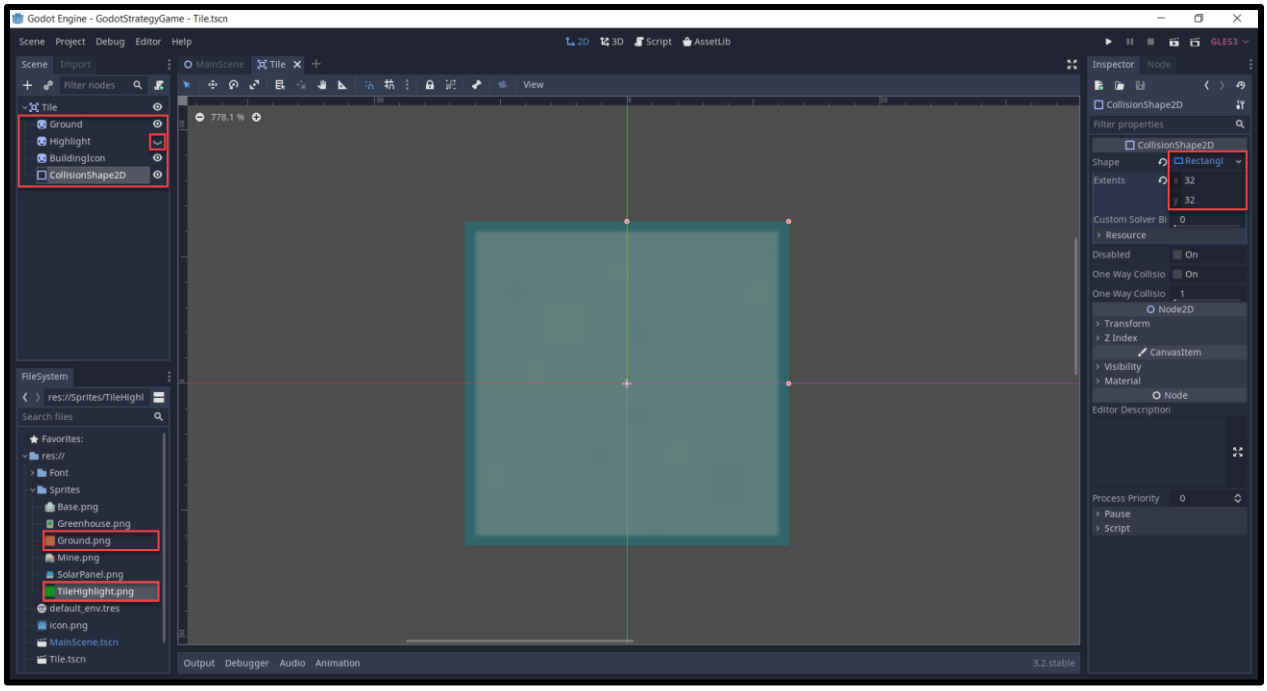
We're going to start by creating the tile scene. This will contain the sprite, highlight, collider, script, etc. So create a new scene with a root node of **Area2D**. An Area2D node can detect collisions which is needed for selecting tiles later on.

1. Rename the node to **Tile**
2. Save the scene to the file system

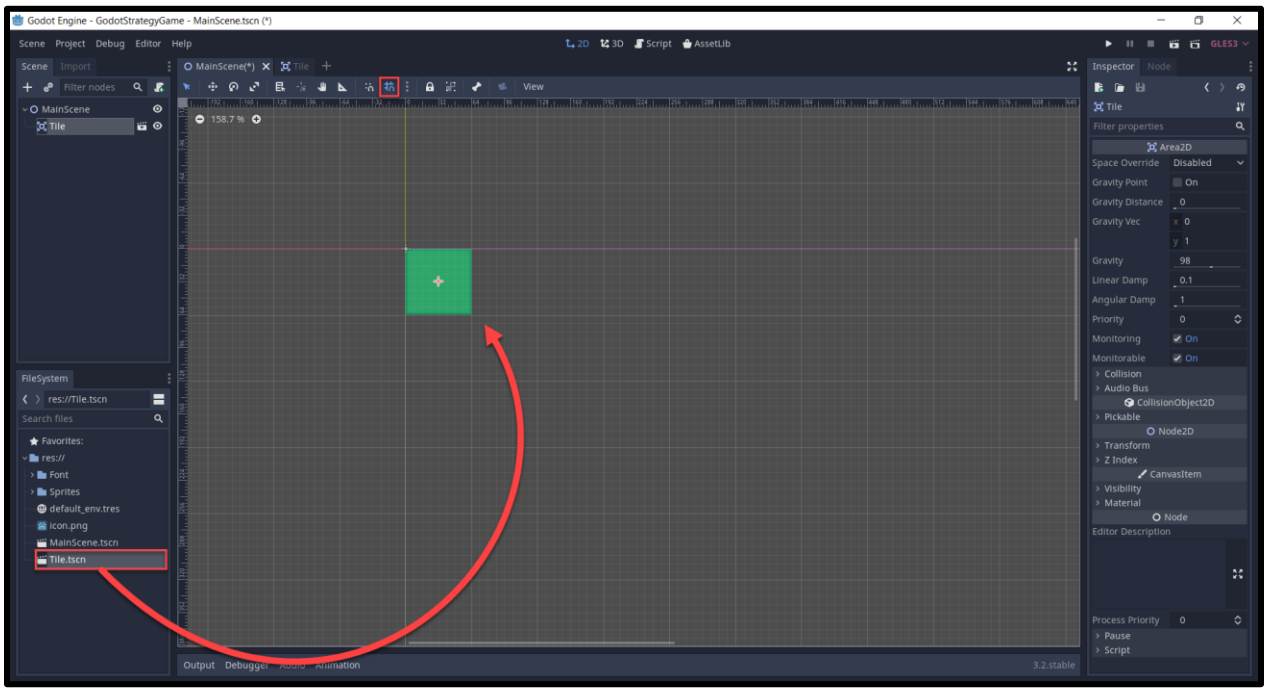


We then want to create a few more nodes as a child of the main node.

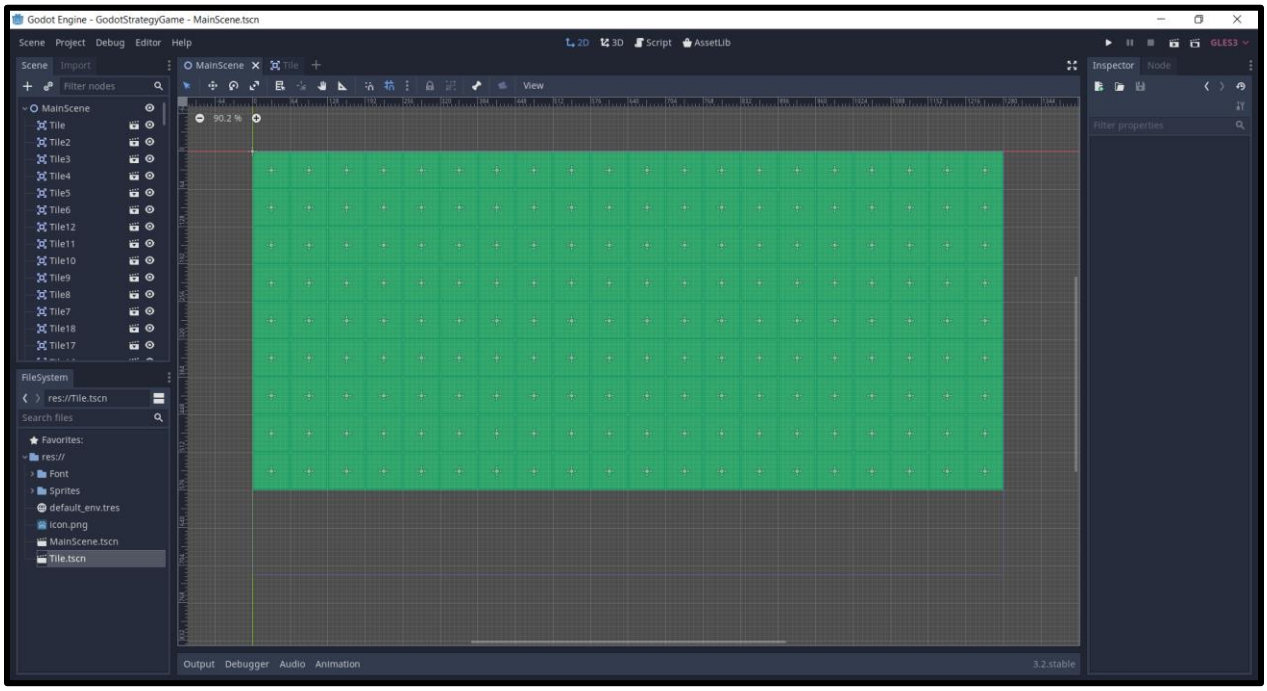
1. Drag in the **Ground.png** image to create a new Sprite node (make sure it's at a position of *0, 0*)
2. Drag in the **TileHighlight.png** image to create a new Sprite node
    a. Rename it to **Highlight**
    b. Set the **Scale** to *6.4, 6.4*
    c. Click the eye to disable it by default
3. Create a new **Sprite** node
    a. Rename it to **BuildingIcon**
4. Create a new **CollisionShape2D** node
    a. Set the **Shape** to *RectangleShape2D*
    b. Set the **Extents** to *32, 32*



Back in the **MainScene**, let's drag in the Tile scene to create a new instance. To make it easier to position, enable **Grid Snapping**.

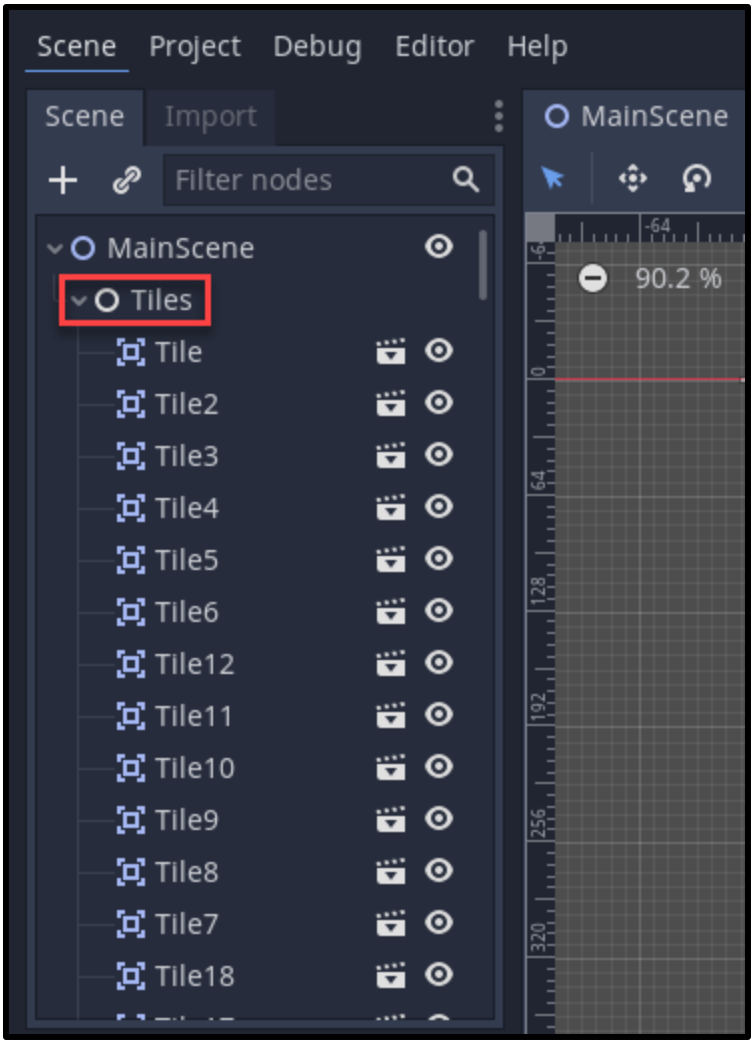


With the tile node selected, press **Ctrl + D** to duplicate it. Fill in a *20* by *9* area with tiles. This is going to be where we play the game and make sure that it's all contained within the blue box (the window).



You may notice that the scene hierarchy is pretty cluttered. We have over 100 nodes which will make it harder to find other things that aren't tiles. In order to fix this, we can...

1. Create a **Node** node (most basic node we'll use as a container)
2. Rename it to **Tiles**
3. Drag all the tile nodes in as a child
4. With this, we can retract the children to hide them in the scene hierarchy

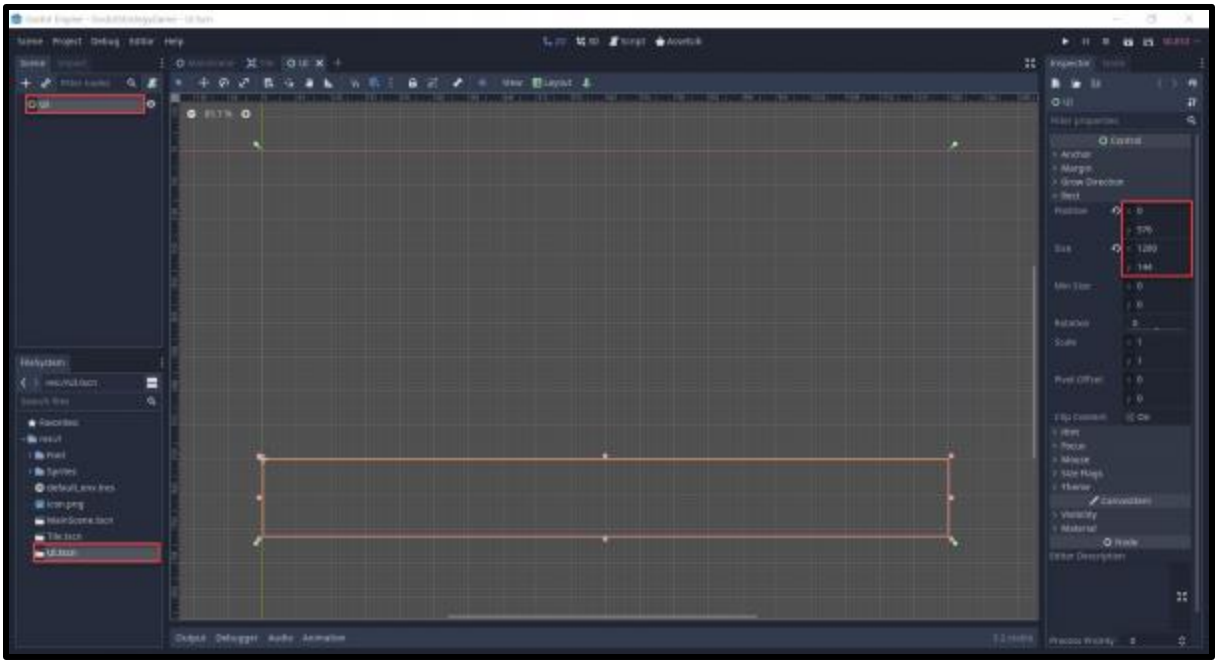


We haven't created the script for the tile yet, but next up is the UI.

Creating the UI

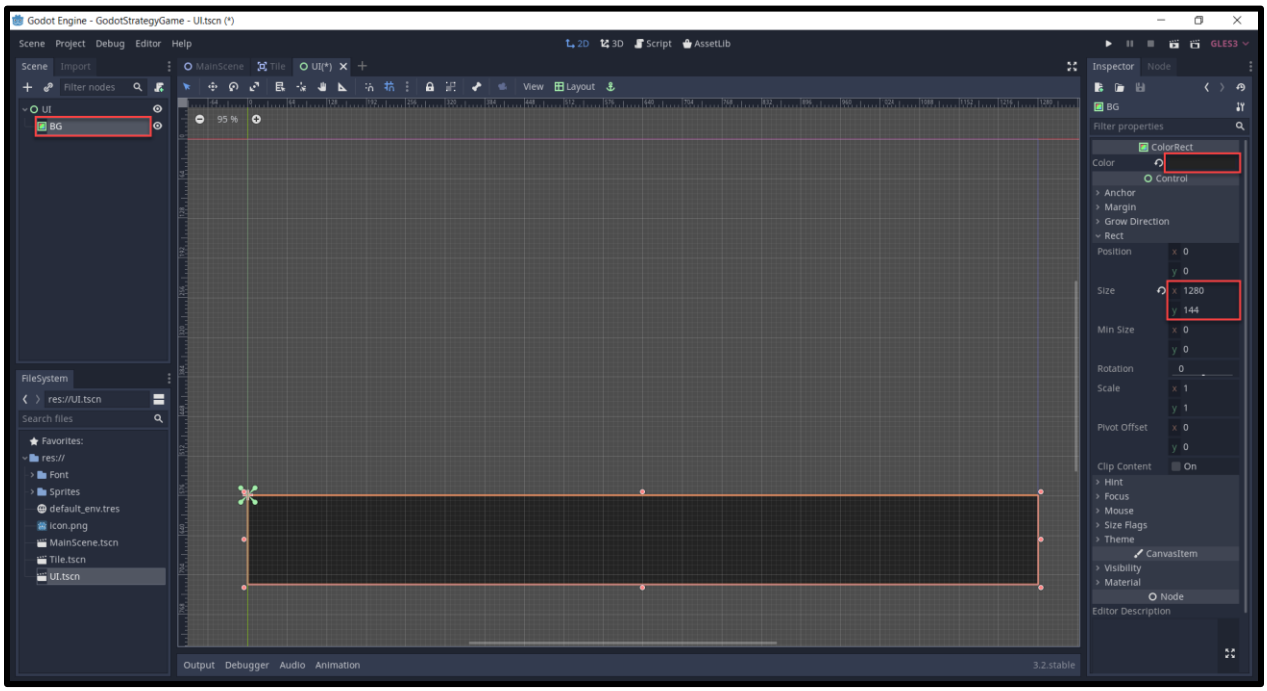
Our UI is going to be a bar at the bottom of the screen displaying our resources, turn and buttons. Create a new scene with a root node of type User Interface (**Control** node).

1. Rename the node to **UI**
2. Save the scene to the file system
3. Set the **Rect > Position** to *0, 576*
4. Set the **Rect > Size** to *1280, 144*



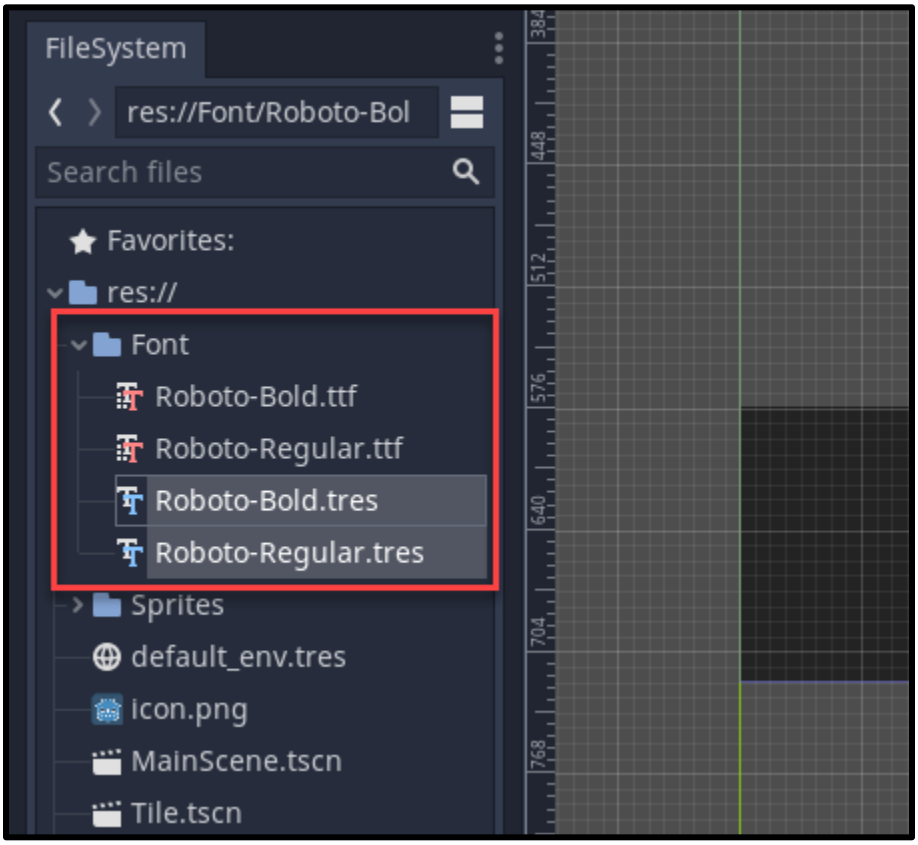
As a child of the UI node, create a **ColorRect** node. This is a control node that renders a certain color.

1. Set the **Color** to *dark grey*
2. Set the **Size** to *1280, 144*



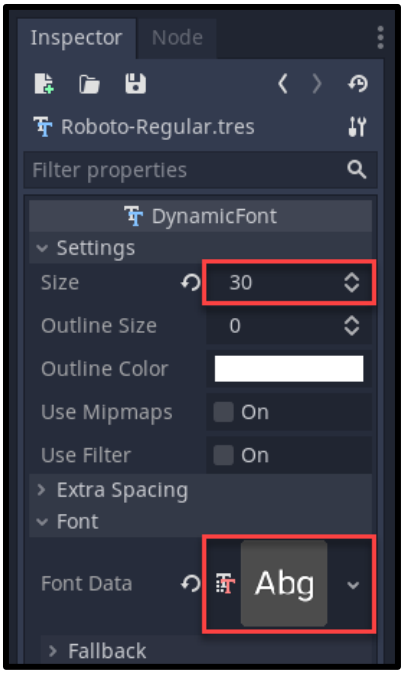
Before we continue, let's setup out fonts. In the **Font** folder, we have two .ttf files. For each font file...

1. Right click the .ttf file and select **New Resource...**
2. Create a new **DynamicFont** resource



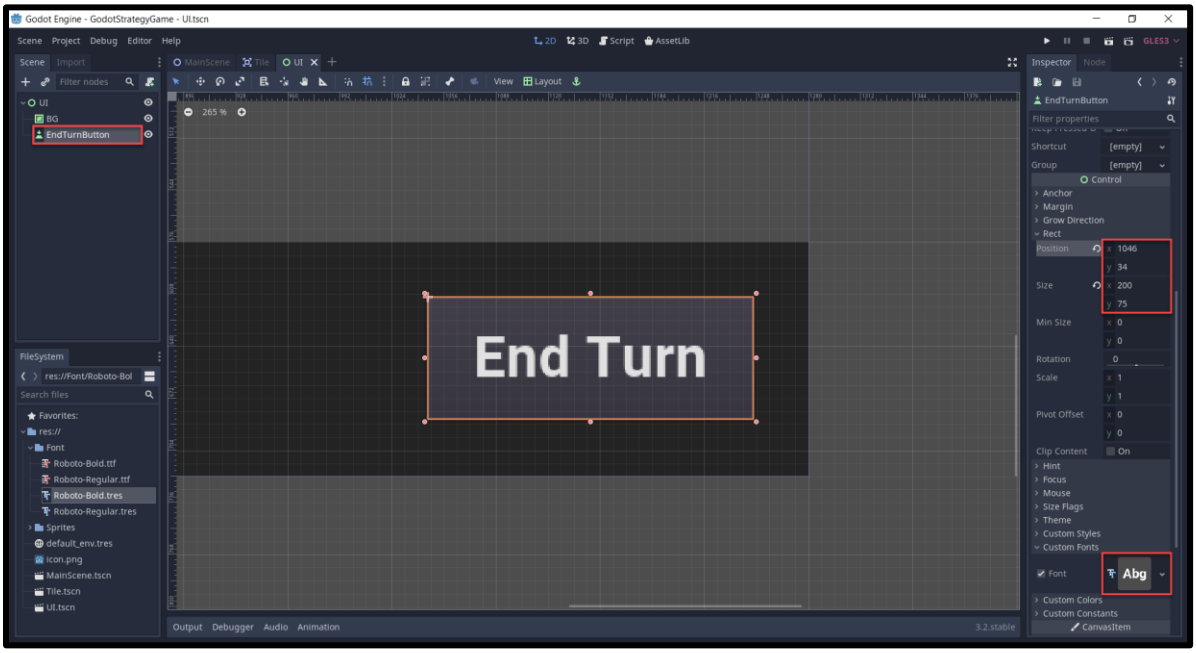
For each dynamic font resource...

1. Double click on the **DynamicFont** resource
2. Set the **Size** to...
    a. Regular = *30*
    b. Bold = *35*
3. Drag the respective .ttf file into the **Font Data** property



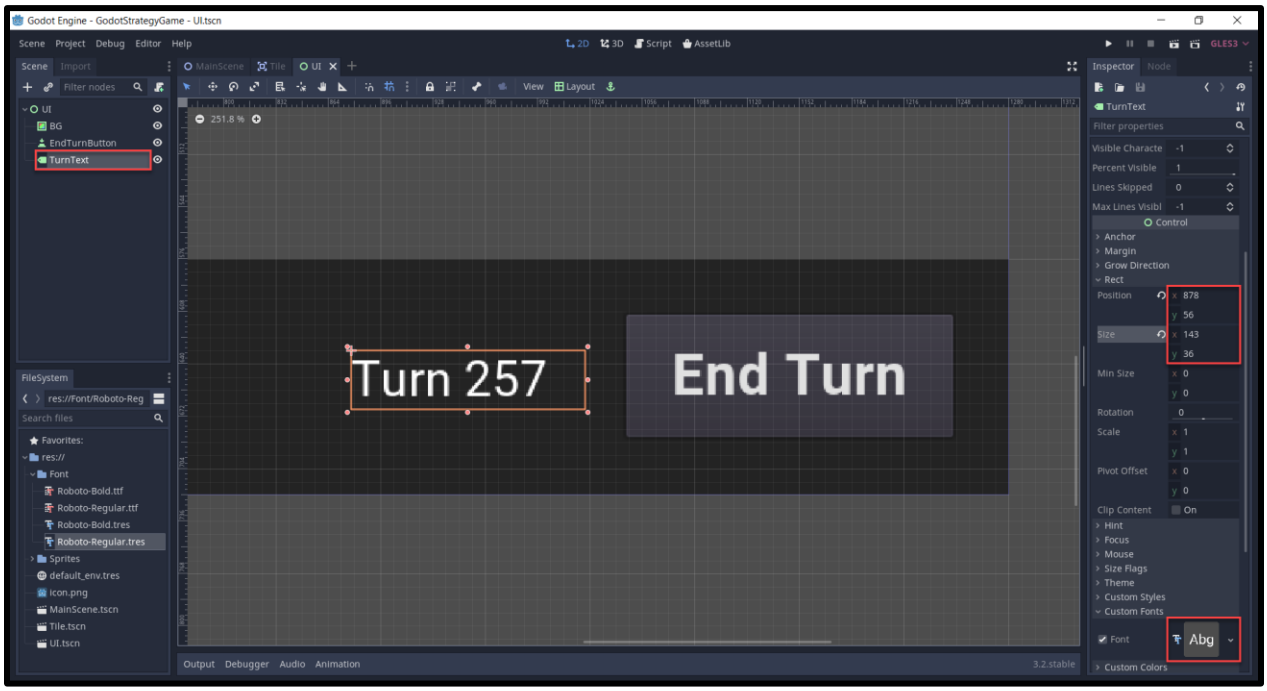
Next, create a **Button** node and rename it as **EndTurnButton**.

1. Set the **Text** to *End Turn*
2. Set the **Position** to *1046, 34*
3. Set the **Size** to *200, 75*
4. Drag the bold dynamic font resource into the **Custom Fonts > Font** property



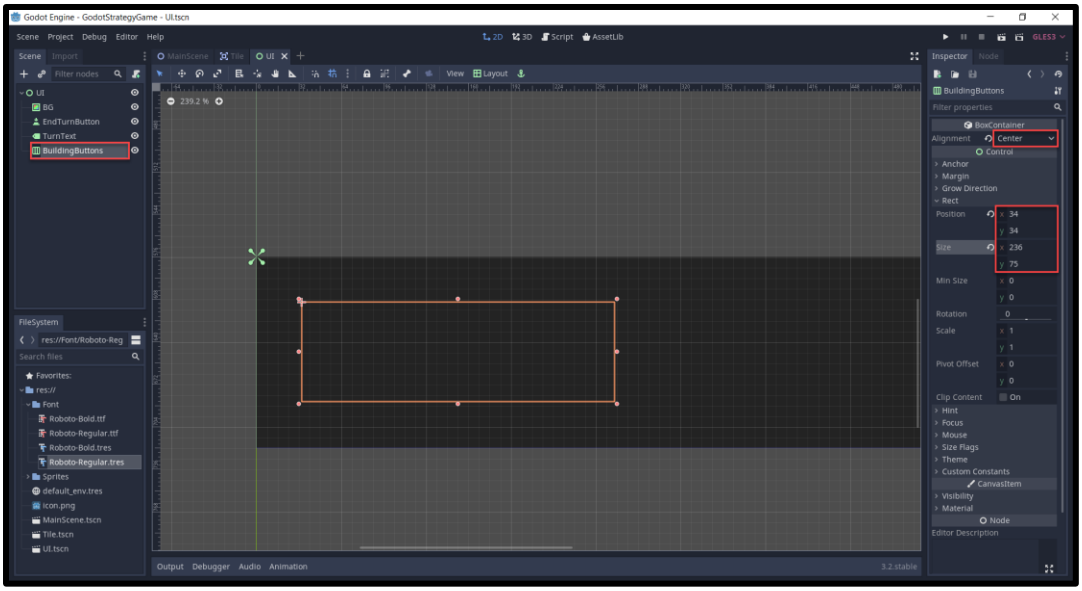
Create a new **Label** node and rename it to **TurnText**.

1. Set the **Text** to a placeholder of *Turn 257*
2. Set the **Position** to *878, 56*
3. Set the **Size** to *143, 36*
4. Drag the regular dynamic font into the **Custom Fonts > Font** property



Next, we'll be creating the three building buttons. We're going to store these inside of a node which can automatically resize its children. Create a new **HBoxContainer** node and rename it to **BuildingButtons**.

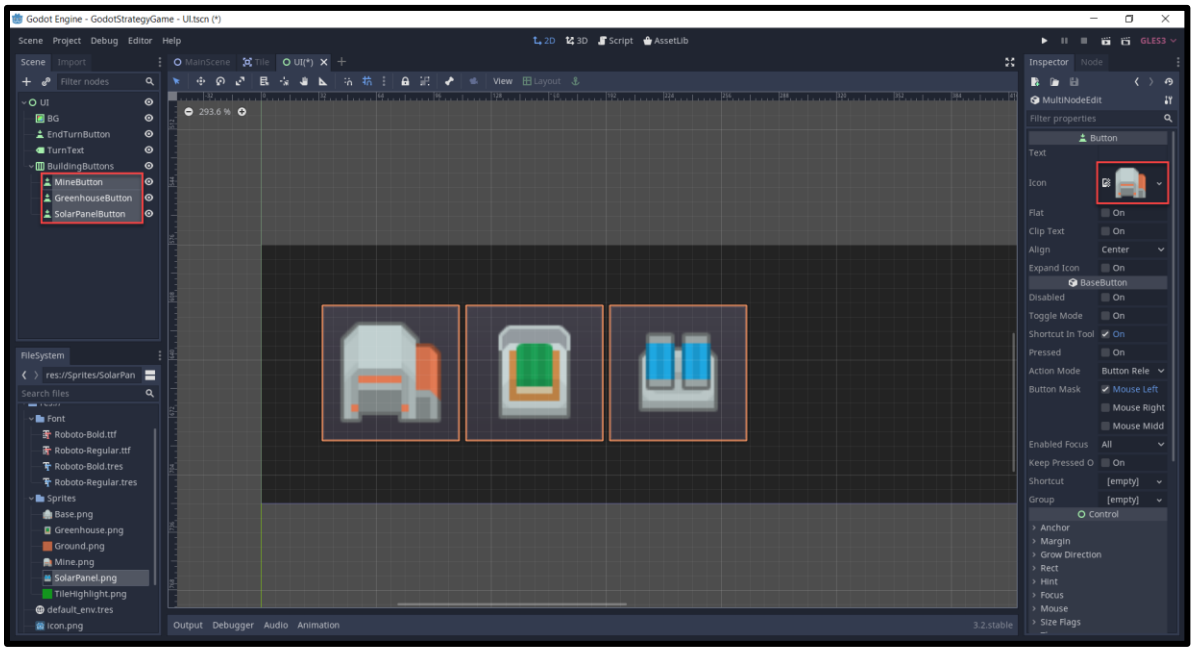
1. Set the **Alignment** to *Center*
2. Set the **Position** to *34, 34*
3. Set the **Size** to *236, 75*



As a child of the **BuildingButtons** node, create three **Button** nodes. You'll see that they automatically resize to fit the bounds of the BuildingButtons node.

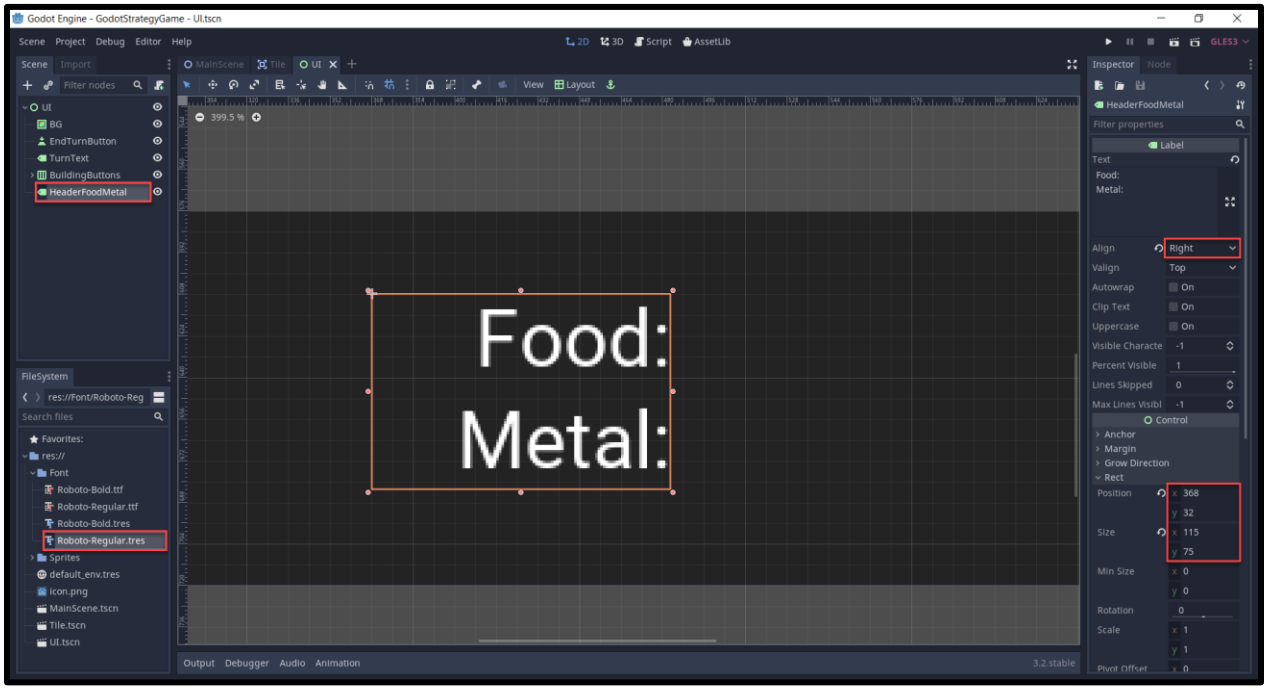
- MineButton
- GreenhouseButton
- SolarPanelButton

For each button, set their **Icon** texture to their respective texture in the **Sprites** folder.



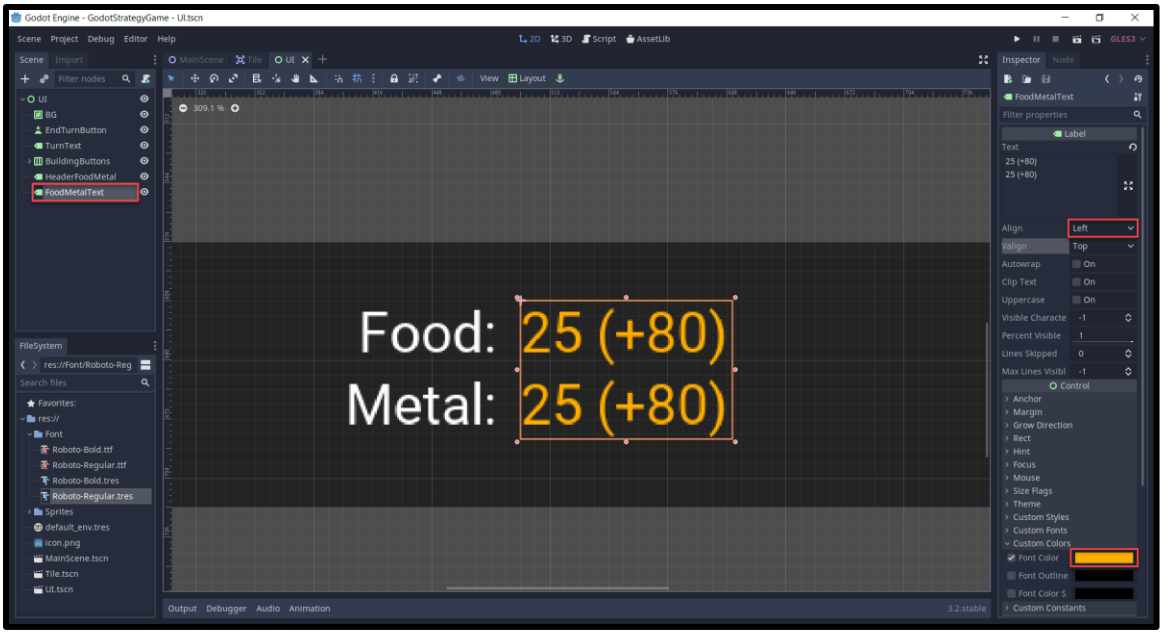
Create a new **Label** node and rename it to **HeaderFoodMetal**.

1. Set the **Text** to *Food: Metal*:
2. Set the **Align** to *Right*
3. Set the **Position** to *368, 32*
4. Set the **Size** to *115, 75*
5. Set the **Custom Font > Font** property to the regular dynamic font



Duplicate the node and rename it to **FoodMetalText**.

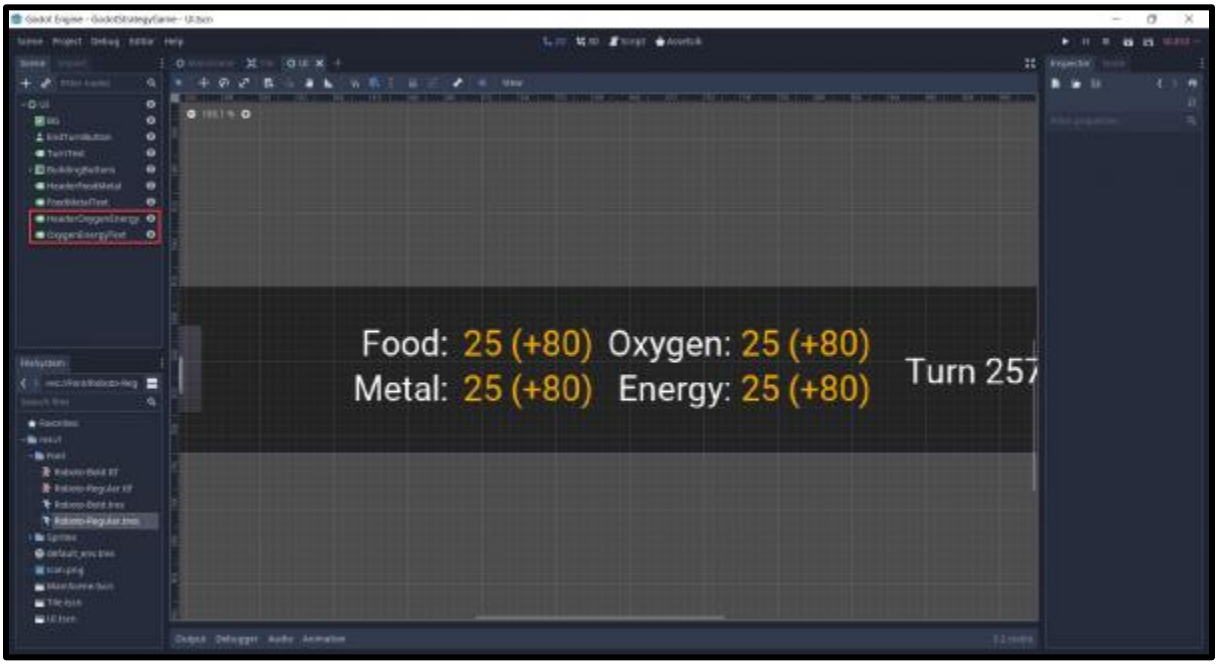
1. Move it over to the right like in the image below
2. Set the **Text** to a placeholder of *25 (+80) 25 (+80)*
3. Set the **Align** to *Left*
4. Set the **Custom Colors > Font Color** to *yellow/orange*



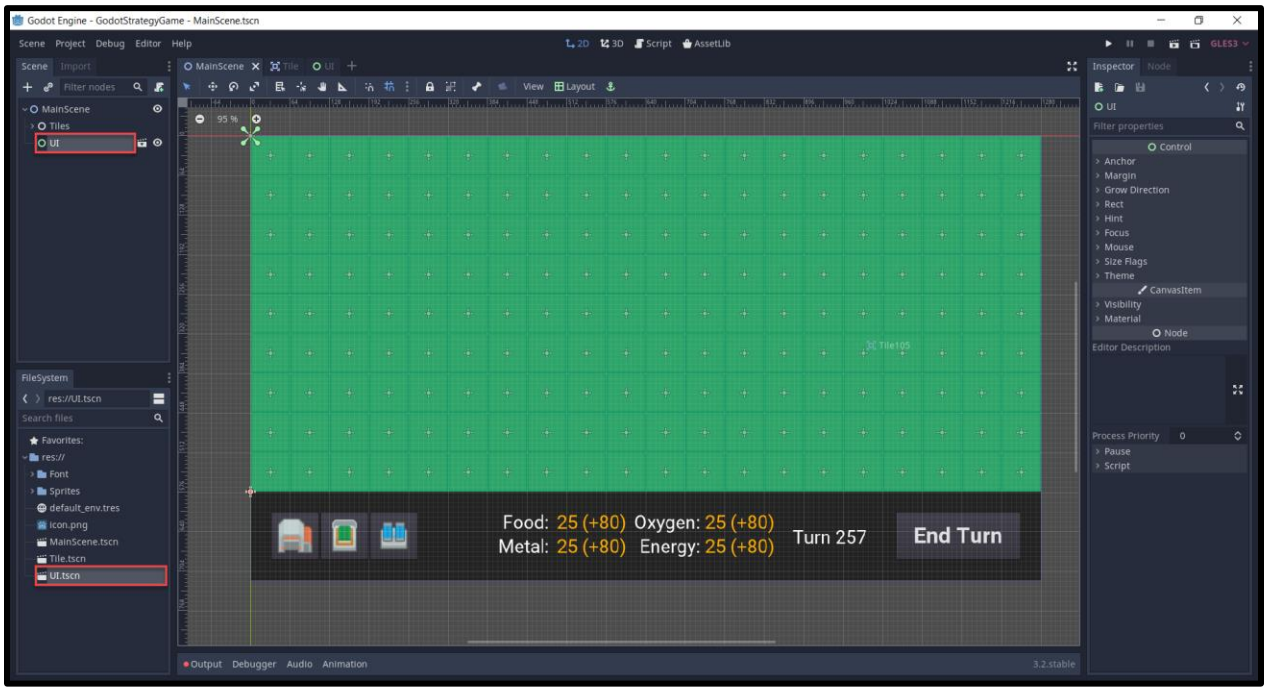
Select the two labels and duplicate them. Rename them respectively to:

- HeaderOyxgenEnergy
- OxygenEnergyText

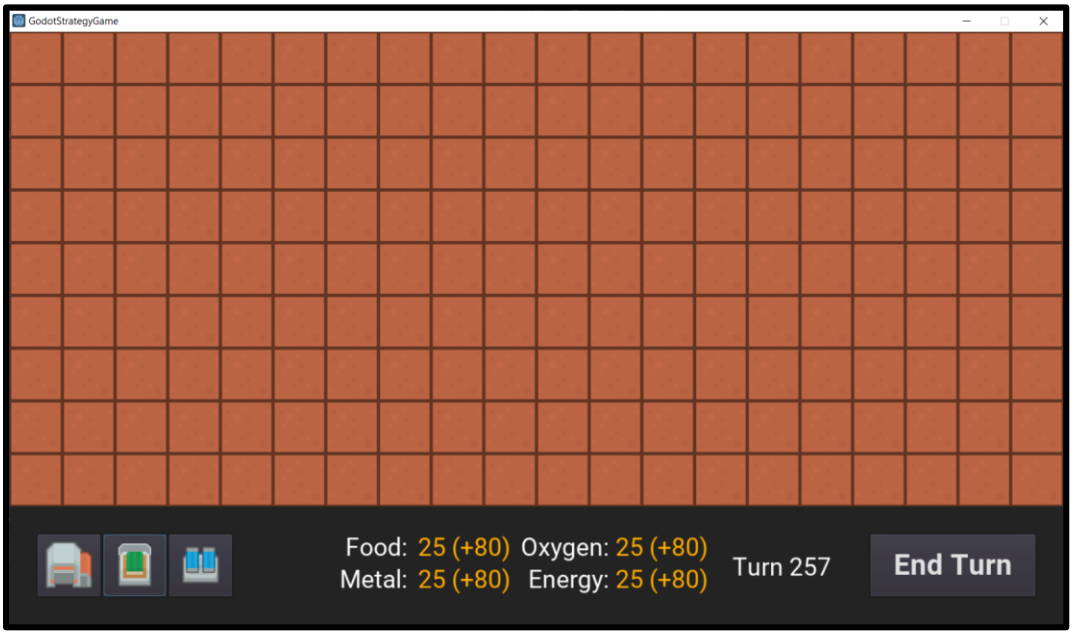
Move them over to the right. Change the text of the header label to '*Oxygen: Energy:*'.



There we go. Our UI is now complete, so let's go back to the **MainScene** and drag in the **UI** scene.



Now if we press play, you'll see basically what the final game will look like - just without the functionality.



Tile Script

To begin scripting, we'll create the **Tile** script. This will hold data and functions relative to each tile. In the **Tile** scene, select the Tile node and create a new script attached to it called **Tile**. We'll start with our variables.

```
# is this the starting tile?
# a Base building will be placed here at the start of the game
export var startTile = false

# do we have a building on this tile?
var hasBuilding : bool = false

# can we place a building on this tile?
var canPlaceBuilding : bool = false

# components
onready var highlight : Sprite = get_node("Highlight")
onready var buildingIcon : Sprite = get_node("BuildingIcon")
```

Next, let's create the _**ready** function - this gets called when the node is initialized. Here, we're going to add the tile to the "Tiles" group. A group in Godot can allow us to easily access and modify a group of nodes.

```
# called once when the node is initialized
func _ready ():

    # add the tile to the "Tiles" group when the node is initialized
    add_to_group("Tiles")
```

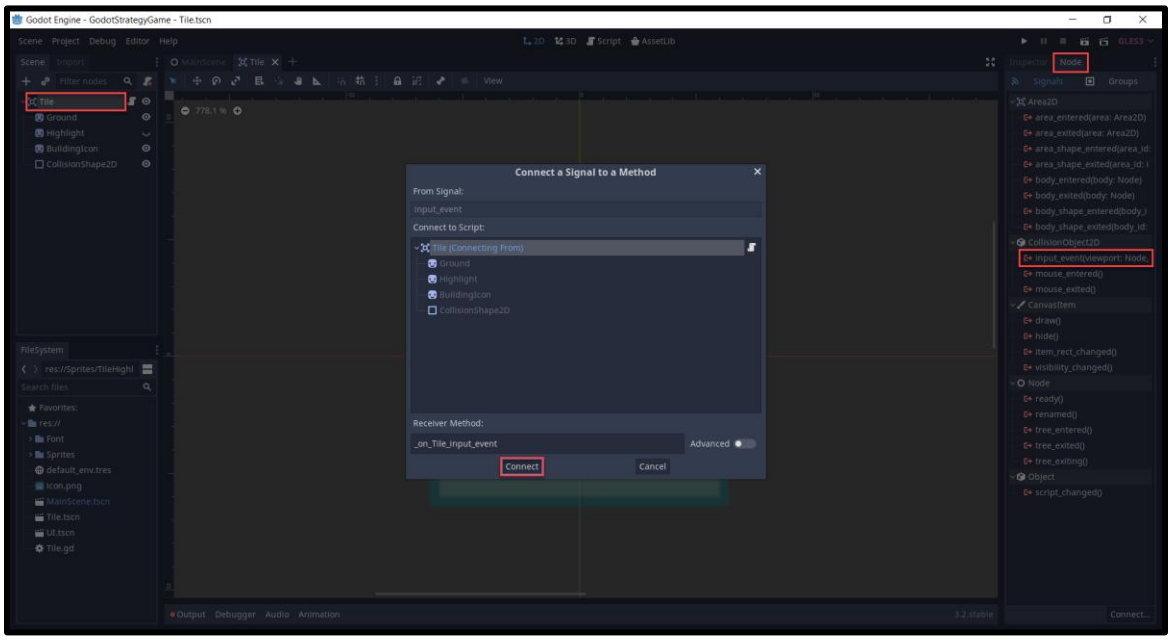
The **toggle_highlight** function will toggle the highlight visual on or off, also toggling whether or not a building can be placed.

```
1  # turns on or off the green highlight
2  func toggle_highlight (toggle):
3
4      highlight.visible = toggle
5      canPlaceBuilding = toggle
```

The **place_building** function gets called when we place a building on the tile.

```
1  # called when a building is placed on the tile
2  # sets the tile's building texture to display it
3  func place_building (buildingTexture):
4
5      hasBuilding = true
6      buildingIcon.texture = buildingTexture
```

Next, we need to connect a signal to the script. The **input_event** signal gets emitted once an input is detected in the collider like a mouse click. So with the **Tile** node selected, go to the **Node** panel and double click the signal. Then connect it to the script.



This will create the **_on_Tile_input_event** function in the script. We need some other things implemented first, so let's just add **pass** to the function which means it's empty.

```
# called when an input event takes place on the tile
func _on_Tile_input_event (viewport, event, shape_idx):

    pass
```

Map Script

The **Map** script will reference all our tiles and have the ability to highlight the available ones, get a tile at a given position, etc. In the **MainScene**, select the Tiles node and attach a new script called **Map**. We can start with our variables.

```
# all the tiles in the game
var allTiles : Array

# all the tiles which have buildings on them
var tilesWithBuildings : Array

# size of a tile
var tileSize : float = 64.0
```

The **get_tile_at_position** function will return the tile that is at the given position.

```
# returns a tile at the given position - returns null if no tile is found
func get_tile_at_position (position):

    # loop through all of the tiles
    for x in range(allTiles.size()):
        # if the tile matches our given position, return it
        if allTiles[x].position == position and allTiles[x].hasBuilding == false:
            return allTiles[x]

    return null
```

The **disable_tile_highlights** function loops through all the tiles and disables their highlight visual.

```
# disables all of the tile highlights
func disable_tile_highlights ():

    for x in range(allTiles.size()):
        allTiles[x].toggle_highlight(false)
```

The **highlight_available_tiles** function will highlight all of the tiles which can have a building placed on. This is the surrounding tiles of all buildings. First, we're going to loop through all tiles that have a building on it.

```
# highlights the tiles we can place buildings on
func highlight_available_tiles ():

    # loop through all of the tiles with buildings
    for x in range(tilesWithBuildings.size()):
```

Inside the for loop, we're first going to get the north, south, east and west tile relative to the current one.

```
# get the tile north, south, east and west of this one
var northTile = get_tile_at_position(tilesWithBuildings[x].position + Vector2(0, tileSize))
var southTile = get_tile_at_position(tilesWithBuildings[x].position + Vector2(0, -tileSize))
var eastTile = get_tile_at_position(tilesWithBuildings[x].position + Vector2(tileSize, 0))
var westTile = get_tile_at_position(tilesWithBuildings[x].position + Vector2(-tileSize, 0))
```

After this, we're going to check each of the tiles. If they're not null, enable the highlight.

```gdscript
# if the directional tiles aren't null, toggle their highlight - allowing us to build
if northTile != null:
    northTile.toggle_highlight(true)
if southTile != null:
    southTile.toggle_highlight(true)
if eastTile != null:
    eastTile.toggle_highlight(true)
if westTile != null:
    westTile.toggle_highlight(true)
```

We're not finished with the **Map** script yet, but we do need to work on a few other things first.

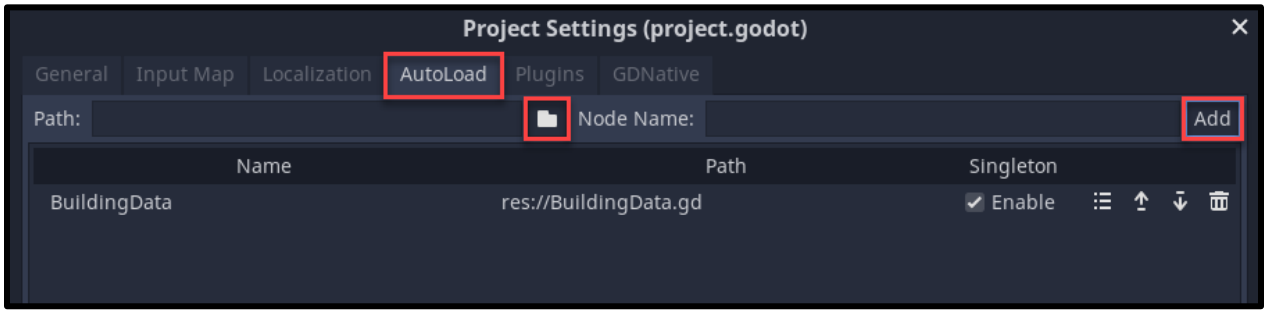
BuildingData Script

The **BuildingData** script isn't a script which is attached to a node. This is going to be a global class which can be accessed anywhere in the project. In the **Script** tab, go *File > New Script...* and create a new script called **BuildingData**.

Then we need to go to the **Project Settings** (*Project > Project Settings...*) and click on the **AutoLoad** tab.

1. Select the file button and find the BuildingData.gd script
2. Click the **Add** button

This will create a new singleton for the **BuildingData** script.



Back in the **BuildingData** script, let's begin by creating a new class called **Building**. This is going to be an object to hold the data for each building.

```gdscript
class Building:

    # building type
    var type : int

    # building texture
    var iconTexture : Texture

    # resource the building produces
    var prodResource : int = 0
    var prodResourceAmount : int

    # resource the building needs to be maintained
    var upkeepResource : int = 0
    var upkeepResourceAmount : int

    func _init (type, iconTexture, prodResource, prodResourceAmount,
                upkeepResource, upkeepResourceAmount):
        self.type = type
        self.iconTexture = iconTexture
        self.prodResource = prodResource
        self.prodResourceAmount = prodResourceAmount
        self.upkeepResource = upkeepResource
        self.upkeepResourceAmount = upkeepResourceAmount
```

We're using integers for the building type and resources. These integers refer to a specific thing.

Building Types

- Base = 0
- Mine = 1
- Greenhouse = 2
- Solar Panel = 3

Resources

- None = 0
- Food = 1
- Metal = 2
- Oxygen = 3
- Energy = 4

Outside of the Building class, we can create 4 variables for each of our 4 buildings (Base, Mine, Greenhouse, Solar Panel).

```
var base = Building.new(0, preload("res://Sprites/Base.png"), 0, 0, 0, 0)
var mine = Building.new(1, preload("res://Sprites/Mine.png"), 2, 1, 4, 1)
var greenhouse = Building.new(2, preload("res://Sprites/Greenhouse.png"), 1, 1, 0, 0)
var solarpanel = Building.new(3, preload("res://Sprites/SolarPanel.png"), 4, 1, 0, 0)
```

We'll be using these variables to know what each building produces, takes, the texture and other values.

Continued in Part 2

While there are certainly more elements to add, we've gone on quite the journey here so far! As the basis of our game, we've already set up a grid-based tile system that will allow us to place buildings later and highlight tiles where applicable buildings can be played. Additionally, we've also set up the buildings themselves which will provide the crucial resource management functionality to our game (along with an accompanying UI).

While this is a great start in building a strategy game with Godot, we have a bit more to go! In Part 2, we're going to finish off our map, create the ability to place the buildings, create the turn-based gameplay flow, and beyond to connect all our features up to work as one, coherent game project! See you then!

Make a Strategy Game in Godot - Part 2

Introduction

Welcome back to Part 2 of creating a strategy game in the Godot game engine!

In Part 1, we began showing you how to build your first strategy game in Godot by setting up highlight-able tiles, buildings, and the basic UI which will tell the players crucial information about their resource management. All in all, we have a great foundation to work with so we can finish our project and add it to our portfolio!

Of course, with this being Part 2, there is still more go. We need to finish up our map system, set up our UI properly, give ourselves the ability to place buildings, and even implement the overall turn-based gameplay flow. So, if you're prepared, and let's finish this resource management game and become master Godot developers!

Project Files

In this tutorial, we'll be using some sprites from the kenney.nl website (an open domain game asset website) and fonts from Google Fonts. You can of course choose to use your own assets, but we'll be designing the game around these:

You can download the source code and assets here.

Finishing the Map Script

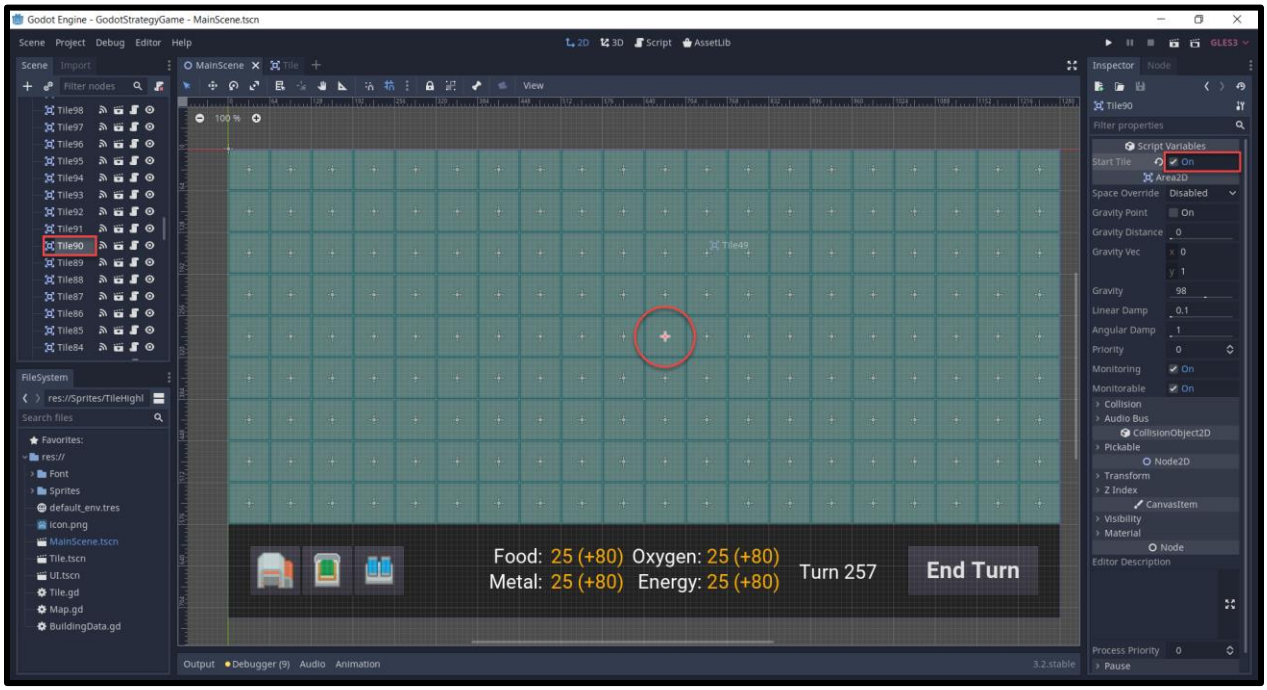
To begin, let's go back to our **Map** script. The **place_building** function gets called when we want to place down a building on a tile.

```
1  # places down a building on the map
2  func place_building (tile, texture):
3
4      tilesWithBuildings.append(tile)
5      tile.place_building(texture)
6
7      disable_tile_highlights()
```

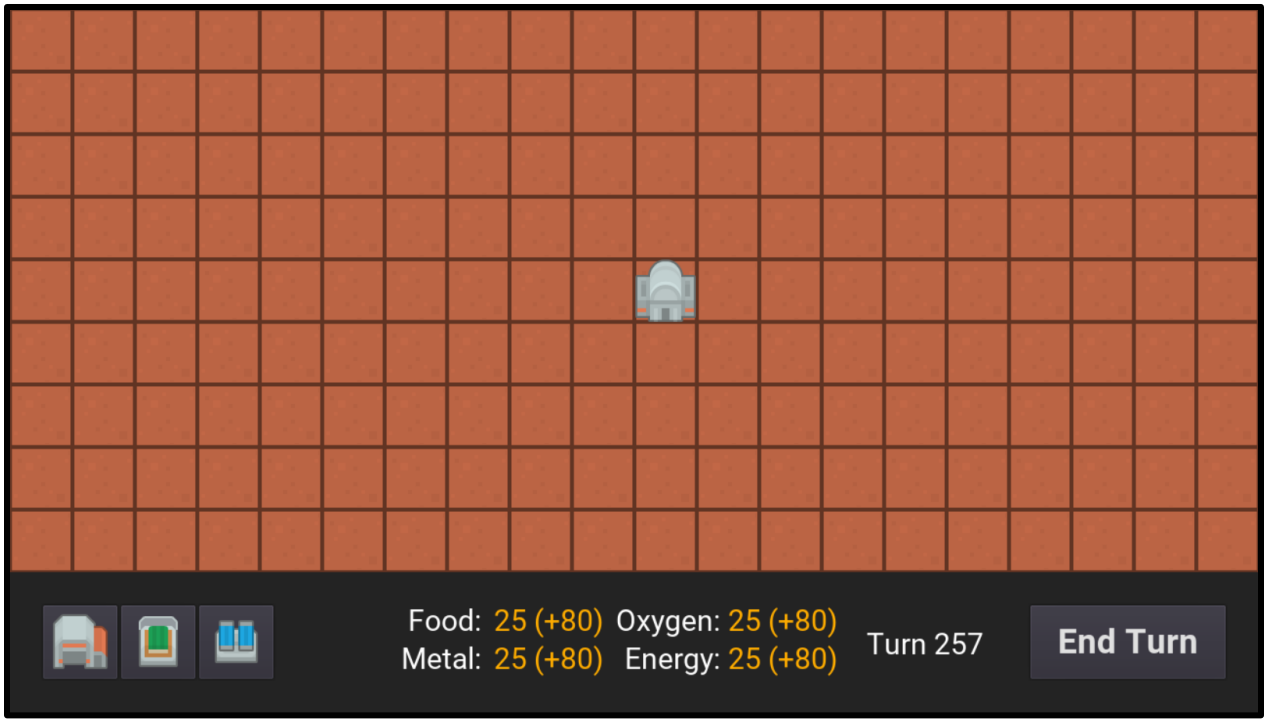
Finally, the **_ready** function gets called when the node is initialized. Here, we want to get all of the tiles in the "Tiles" group and setup the initial base building.

```
1  func _ready ():
2
3      # when we're initialized, get all of the tiles
4      allTiles = get_tree().get_nodes_in_group("Tiles")
5
6      # find the start tile and place the Base building
7      for x in range(allTiles.size()):
8          if allTiles[x].startTile == true:
9              place_building(allTiles[x], BuildingData.base.iconTexture)
```

Back in the **MainScene**, let's select the center tile and enable **Start Tile**.



Now if we press play, you should see that the center tile has a Base building on it.



GameManager Script

The **GameManager** script is what's going to manage our resources and states. Go to the **MainScene** and select the **MainScene** node. Create a new script attached to it called **GameManager**. We can start with our variables.

```gdscript
# current amount of each resource we have
var curFood : int = 0
var curMetal : int = 0
var curOxygen : int = 0
var curEnergy : int = 0

# amount of each resource we get each turn
var foodPerTurn : int = 0
var metalPerTurn : int = 0
var oxygenPerTurn : int = 0
var energyPerTurn : int = 0

var curTurn : int = 1

# are we currently placing down a building?
var currentlyPlacingBuilding : bool = false

# type of building we're currently placing
var buildingToPlace : int

# components
onready var ui : Node = get_node("UI")
onready var map : Node = get_node("Tiles")
```

The **on_select_building** function gets called when we press one of the three building UI buttons. This will be hooked up later on when we create the **UI** script.

```gdscript
# called when we've selected a building to place
func on_select_building (buildingType):

    currentlyPlacingBuilding = true
    buildingToPlace = buildingType

    # highlight the tiles we can place a building on
    map.highlight_available_tiles()
```

The **add_to_resource_per_turn** function adds the given *amount* to the given *resource* per turn.

```gdscript
# adds an amount to a certain resource per turn
func add_to_resource_per_turn (resource, amount):

    # resource 0 means none, so return
    if resource == 0:
        return
    elif resource == 1:
        foodPerTurn += amount
    elif resource == 2:
        metalPerTurn += amount
    elif resource == 3:
        oxygenPerTurn += amount
    elif resource == 4:
        energyPerTurn += amount
```

The **place_building** function will be called when we place down a tile on the grid.

```gdscript
# called when we place a building down on the grid
func place_building (tileToPlaceOn):

    currentlyPlacingBuilding = false

    var texture : Texture

    # are we placing down a Mine?
    if buildingToPlace == 1:
        texture = BuildingData.mine.iconTexture

        add_to_resource_per_turn(BuildingData.mine.prodResource,
          BuildingData.mine.prodResourceAmount)
        add_to_resource_per_turn(BuildingData.mine.upkeepResource,
          -BuildingData.mine.upkeepResourceAmount)

    # are we placing down a Greenhouse?
    if buildingToPlace == 2:
        texture = BuildingData.greenhouse.iconTexture

        add_to_resource_per_turn(BuildingData.greenhouse.prodResource,
          BuildingData.greenhouse.prodResourceAmount)
        add_to_resource_per_turn(BuildingData.greenhouse.upkeepResource,
          -BuildingData.greenhouse.upkeepResourceAmount)

    # are we placing down a Solar Panel?
    if buildingToPlace == 3:
        texture = BuildingData.solarpanel.iconTexture

        add_to_resource_per_turn(BuildingData.solarpanel.prodResource,
          BuildingData.solarpanel.prodResourceAmount)
        add_to_resource_per_turn(BuildingData.solarpanel.upkeepResource,
          -BuildingData.solarpanel.upkeepResourceAmount)

    # place the building on the map
    map.place_building(tileToPlaceOn, texture)
```

Finally, we have the **end_turn** function which gets called when we press the end turn button.

```
# called when the player ends the turn
func end_turn ():

    # update our current resource amounts
    curFood += foodPerTurn
    curMetal += metalPerTurn
    curOxygen += oxygenPerTurn
    curEnergy += energyPerTurn

    # increase current turn
    curTurn += 1
```

Okay so we've got our GameManager class all setup but there's no real way for it to function. In order to connect everything together, we need to create a **UI** script.

UI Script

In the **UI** scene, select the UI node and create a new script called **UI**. Let's start with our variables.

```
# container holding the building buttons
onready var buildingButtons : Node = get_node("BuildingButtons")

# text displaying the food and metal resources
onready var foodMetalText : Label = get_node("FoodMetalText")

# text displaying the oxygen and energy resources
onready var oxygenEnergyText : Label = get_node("OxygenEnergyText")

# text showing our current turn
onready var curTurnText : Label = get_node("TurnText")

# game manager object in order to access those functions and values
onready var gameManager : Node = get_node("/root/MainScene")
```

First, we have the **on_end_turn** function. This gets called when a turn is over, so we're going to reset the UI.

```
# called when a turn is over - resets the UI
func on_end_turn ():

    # updates the cur turn text and enable the building buttons
    curTurnText.text = "Turn: " + str(gameManager.curTurn)
    buildingButtons.visible = true
```

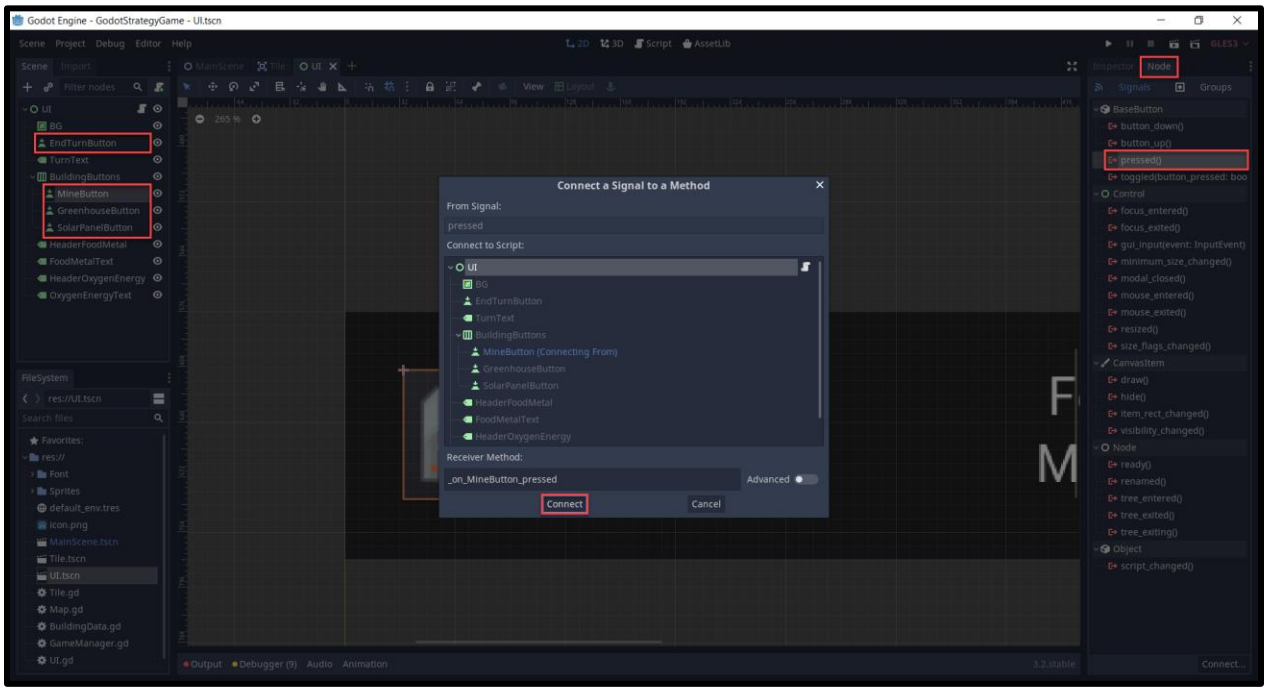
The we have the **update_resource_text** function which updates the two resource labels to show the player's current resource values.

```
# updates the resource text to show the current values
func update_resource_text ():

    # set the food and metal text
    var foodMetal = ""

    # sets the text, e.g. "13 (+5)"
    foodMetal += str(gameManager.curFood) +
      " (" + ("+" if gameManager.foodPerTurn >= 0 else "") +
      str(gameManager.foodPerTurn) + ")"
    foodMetal += "\n"
    foodMetal += str(gameManager.curMetal) +
      " (" + ("+" if gameManager.metalPerTurn >= 0 else "") +
      str(gameManager.metalPerTurn) + ")"

    foodMetalText.text = foodMetal

    # set the oxygen and energy text
    var oxygenEnergy = ""

```

```
21      # set the text, e.g. "13 (+5)"
22      oxygenEnergy += str(gameManager.curOxygen) +
23        " (" + ("+" if gameManager.oxygenPerTurn >= 0 else "") +
24        str(gameManager.oxygenPerTurn) + ")"
25      oxygenEnergy += "\n"
26      oxygenEnergy += str(gameManager.curEnergy) +
27        " (" + ("+" if gameManager.energyPerTurn >= 0 else "") +
28        str(gameManager.energyPerTurn) + ")"
29
30      oxygenEnergyText.text = oxygenEnergy
```

Now we need to connect the buttons. In the **UI** scene, do the following for the EndTurnButton, MineButton, GreenhouseButton and SolarPanelButton...

1. Select the button node
2. Double click the **pressed** signal (called when we press the button)
3. Connect that to the UI script



So back in our script, we'll have 4 new functions. Let's start with the three building buttons.

```gdscript
# called when the Mine building button is pressed
func _on_MineButton_pressed ():

    buildingButtons.visible = false
    gameManager.on_select_building(1)

# called when the Greenhouse building button is pressed
func _on_GreenhouseButton_pressed ():

    buildingbuttons.visible = false
    gameManager.on_select_building(2)

# called when the Solar Panel building button is pressed
func _on_SolarPanelButton_pressed

    buildingButtons.visible = false
    gameManager.on_select_building(3)
```

Then we have the end turn button function.

```gdscript
# called when the "End Turn" button is pressed
func _on_EndTurnButton_pressed ():

    gameManager.end_turn()
```

Connecting Everything Together

Now that we have our UI script, let's go back to the **Tile** script and fill in the **_on_Tile_input_event** function.

```gdscript
# called when an input event takes place on the tile
func _on_Tile_input_event (viewport, event, shape_idx):

    # did we click on this tile with our mouse?
    if event is InputEventMouseButton and event.pressed:
        var gameManager = get_node("/root/MainScene")

        # if we can place a building down on this tile, then do so
        if gameManager.currentlyPlacingBuilding and canPlaceBuilding:
            gameManager.place_building(self)
```

Next, let's hop into the **GameManager** script and create the _**ready** function. Here, we're going to initialize the UI.

```gdscript
func _ready ():

    # updates the UI when the game starts
    ui.update_resource_text()
    ui.on_end_turn()
```

At the end of the **end_turn** function, let's also update the UI.

```gdscript
# update the UI
ui.update_resource_text()
ui.on_end_turn()
```

Finally, at the bottom of the **place_building** function, we can update the resource text UI.

```gdscript
# update the UI to show changes immediately
ui.update_resource_text()
```

Now we can press play and test out the game!

Conclusion

Congratulations on completing the tutorial!

You just created a 2D, turn-based strategy game in Godot. Through this journey, we've covered a wide array of topics, from setting up objects that give and take resources, to creating a tile-based map that provides visual clues about where buildings can be placed. Further, with turn-based gameplay mechanics also introduced, we've tackled a key component for many other sorts of strategy games as well!

From here, you can expand upon what you've learned to add more systems, work on existing ones we touched on here, or even start a new strategy game project with Godot. Regardless, thank you very much for following along with the tutorial, and we wish you the best of luck with your future Godot games.