Game Development with Godot

Part one

Introduction

This tutorial series will go through basics of 2d game development with the Godot engine. The goal is to help the reader get started with their own projects. We will cover basic scripting, graphics and audio and end up with a game that can work as a base for further development.

We will be using the Godot game engine because it is light, free and requires no registration. We will also take a brief look at drawing and animating in Krita and making audio with Audacity. These are also free.

A disclaimer is in order first: I am not a professional game developer. I have a few projects released in a few different engines though.

I'll be trying to go through things as they become relevant to what we're doing rather than trying to explain all the theory before putting it into practice.

Starting out there might be things you feel are difficult to grasp at first. Don't stress about this, it's completely normal if you don't have any experience with similar things.

Here are some helpful links:

https://godotengine.org The website for the Godot engine

https://docs.godotengine.org/en/stable/ Documentation pages for Godot

https://www.gdquest.com GDQuest has free and paid courses for Godot

http://kidscancode.org/godot_recipes/ Kidscancode also has Godot content

https://pixeljoint.com/forum/forum_posts.asp?TID=11299 Some basics of pixel art

Don't worry about them just now though, they will come in handy later.

Getting familiar with Godot

Godot is the most important part of this journey, so let's start by taking a look at that. And in order to take a look at it we first need to download it or run the web version (though I do recommend downloading if at all possible)

Download links are available from:

https://godotengine.org/download

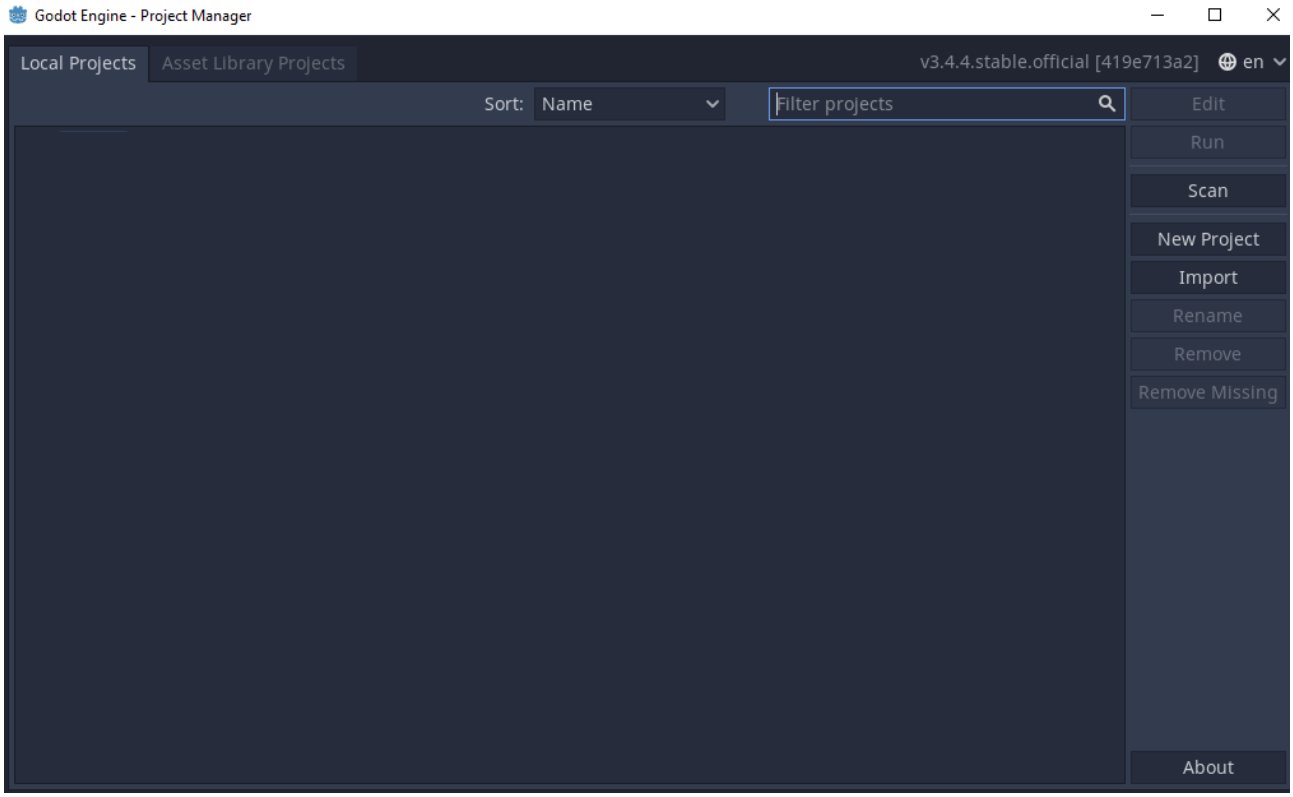
Web version is at:

https://editor.godotengine.org/releases/latest/

Keep in mind that new Godot releases might come while I'm writing this. I recommend using the latest stable version unless newer releases have some bug fix or feature you need. If you didn't catch all that, don't worry about it.

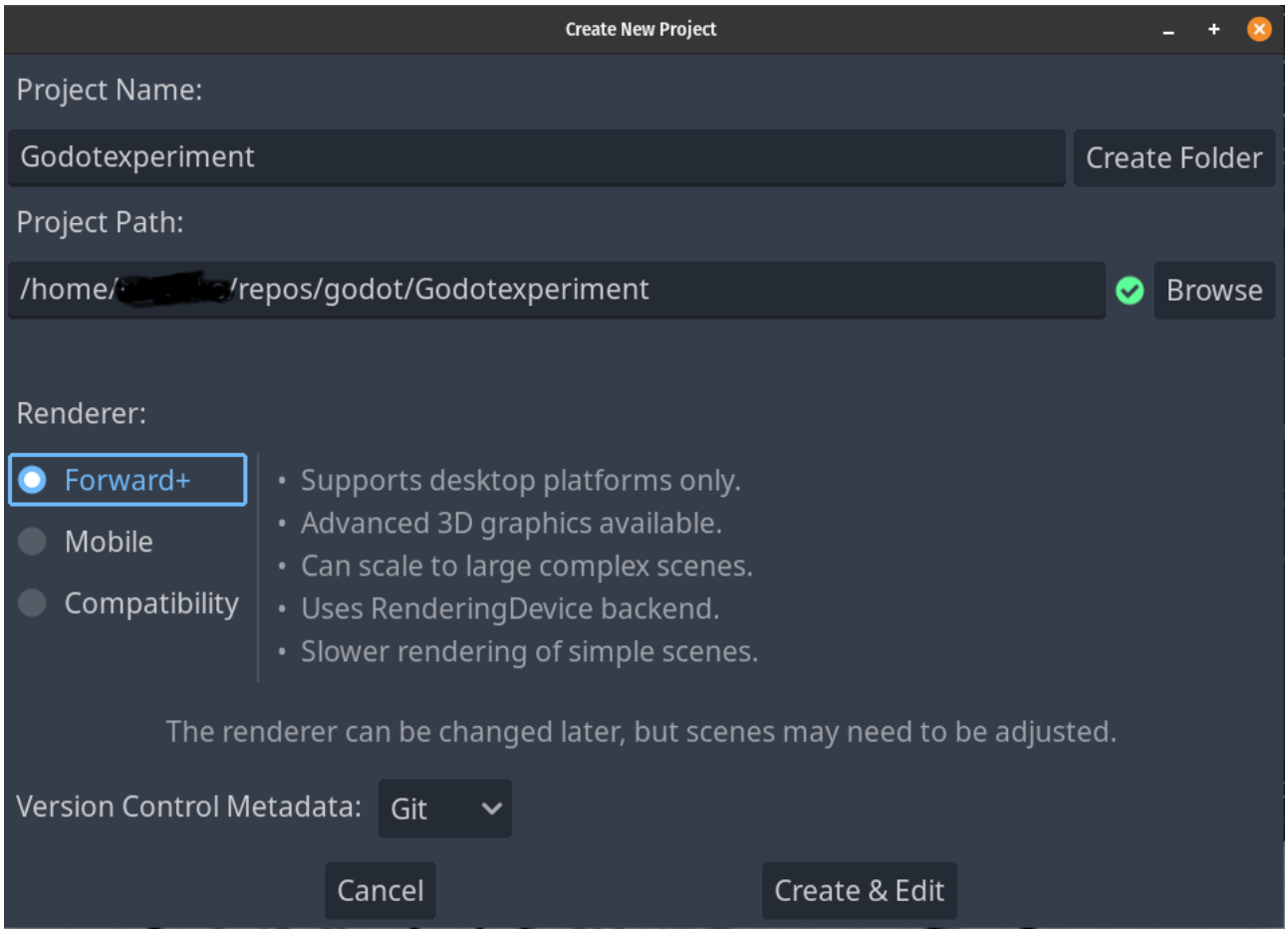
For this tutorial series I'm using the standard 64-bit version. The download will be in the form of a .zip file, simply extract the files within somewhere you can get at them, and open the .exe file. If you are running Windows you might need to give it permission to run the file. If you get a screen saying "Windows protected your PC" click on More Info and then you'll get an option to Run anyway.

Once Godot is open you'll see this screen:



This screen will list all the projects Godot knows about. You can create new projects or import existing projects.

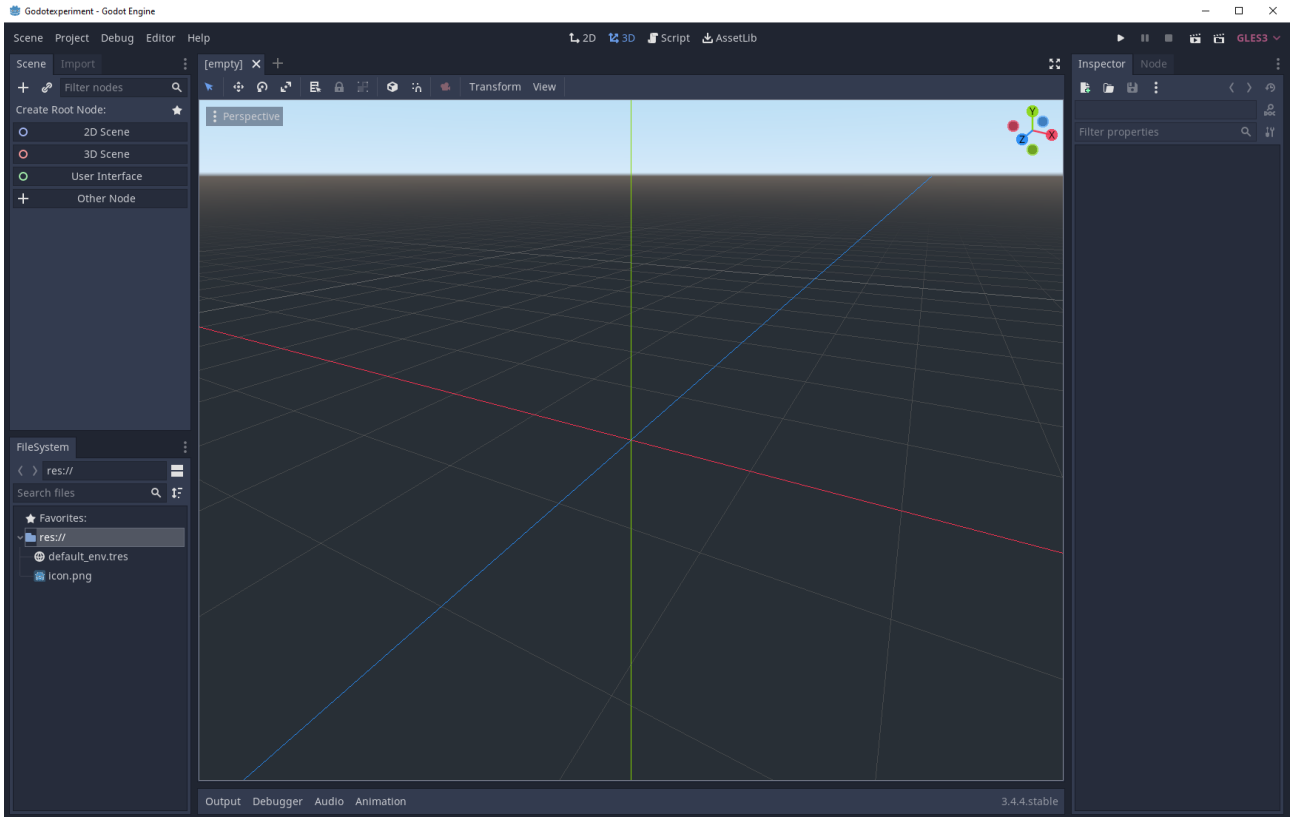
We will create a new project to play around with, so click the New Project button on the right.



Godot will require a project name and then a path to an empty folder. Click on the Browse button or enter in filepath in the Project Path field. Don't worry about the renderer at this point, once you have a folder for the project just click "Create & Edit"

I'm calling my project Godotexperiment because at this point we will do just that, we're not working on the actual game yet.

Now you should see a screen like this:

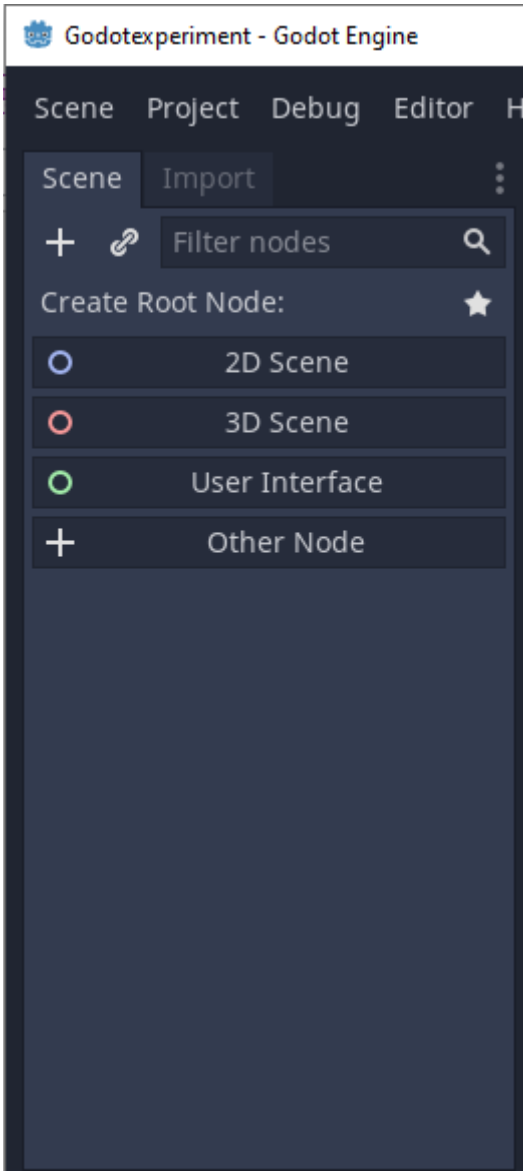


There are quite a few things to go through but we will take it one step at a time. Before we do anything else at this point I should probably talk about how Godot is designed:

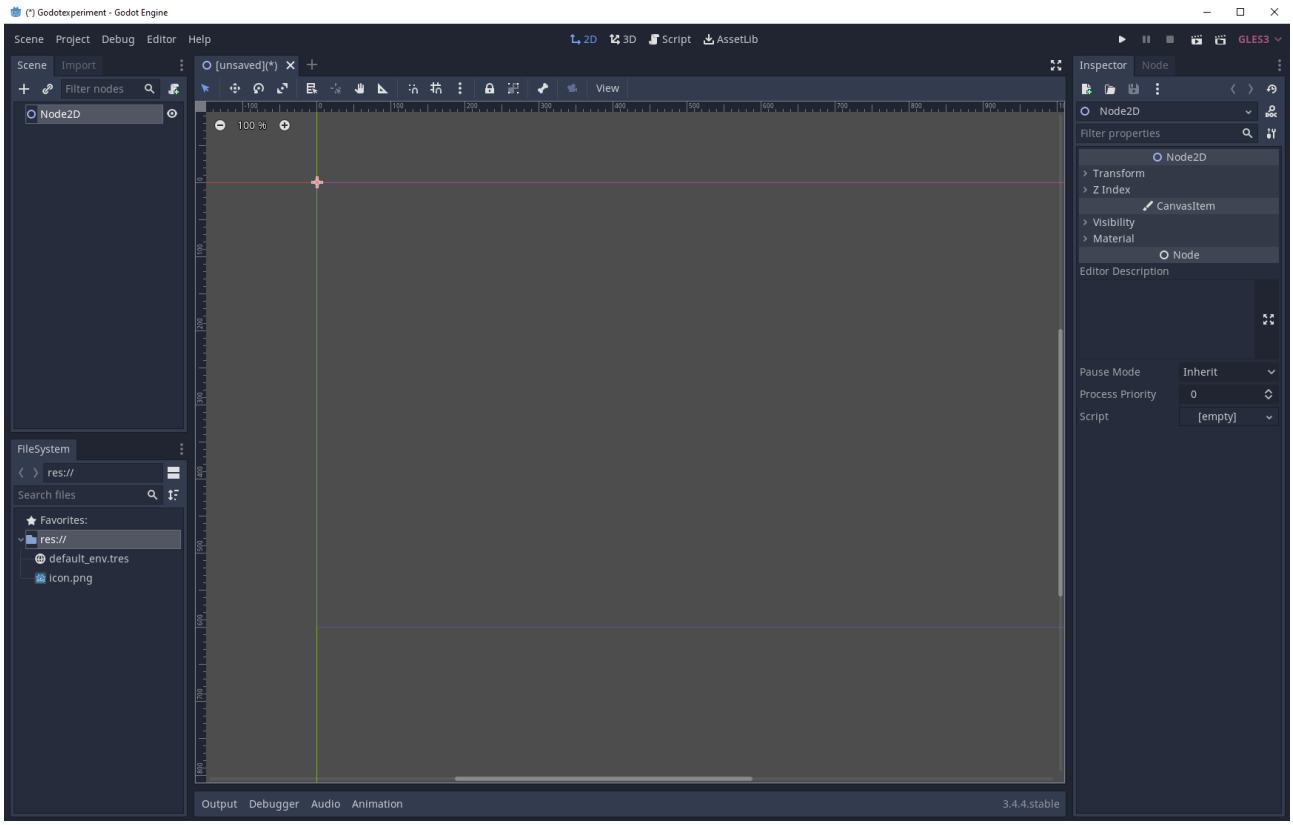
The two main components we will be working with are Scenes and Nodes. A scene is essentially a collection of nodes and nodes are components that have a function assigned to them that can be modified in the editor or in the code. Don’t worry too much if this doesn’t make sense yet, we will explore this concept quite a lot.

Meanwhile you could think of Scenes and Nodes as different objects broken down into their components. Take a door for example, we could consider that a Scene as it is constructed of different components. We have at least a handle, hinges and the actual door itself, so we could consider those the nodes.

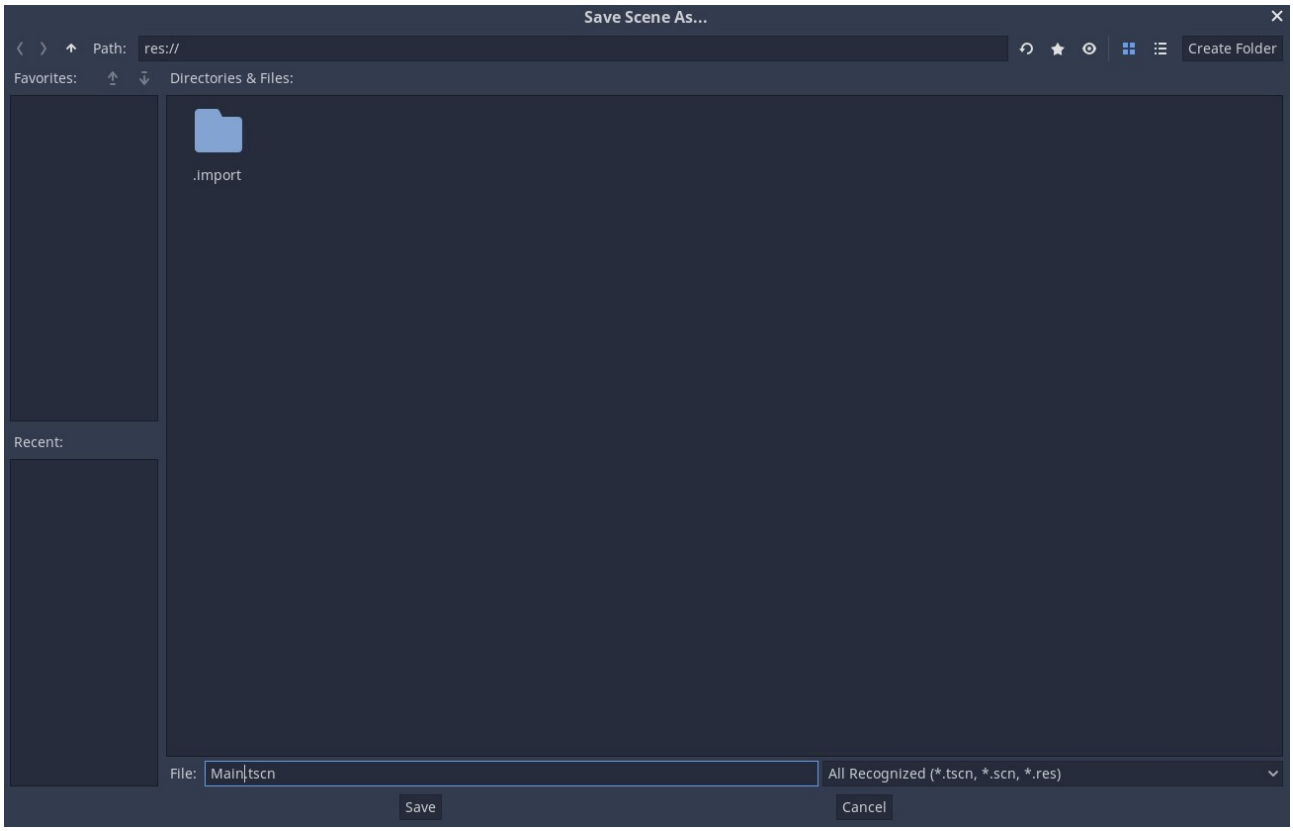
Back to Godot, we need to create a root scene and the scene needs a root node. In the left pane we see this:



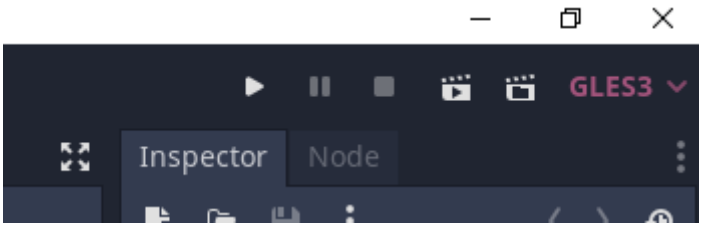
As we are creating a 2D game in this course we will be using the 2D Scene, so click on that.



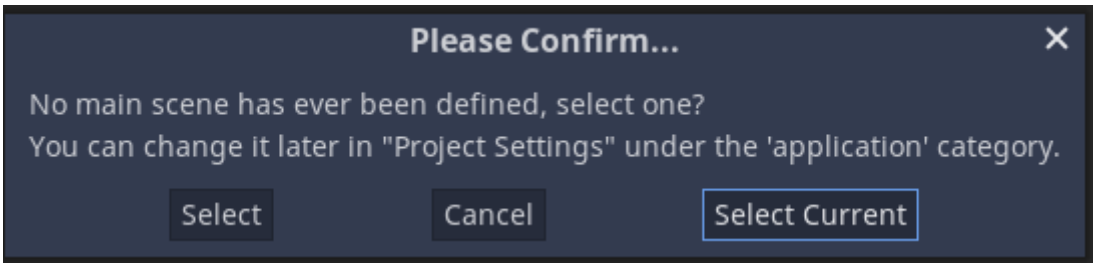
Now our view changes to 2D and we have an unnamed scene with a Node2D node. Before doing anything else I want to save this scene, you can do this by pressing the shortcut ctrl+s or by clicking Scene → Save Scene up in the left corner. I'm going to call my scene Main.tscn so I know this is the main scene.



Now we have a scene, we can test our game by pressing F5 or the play button in the top right corner:

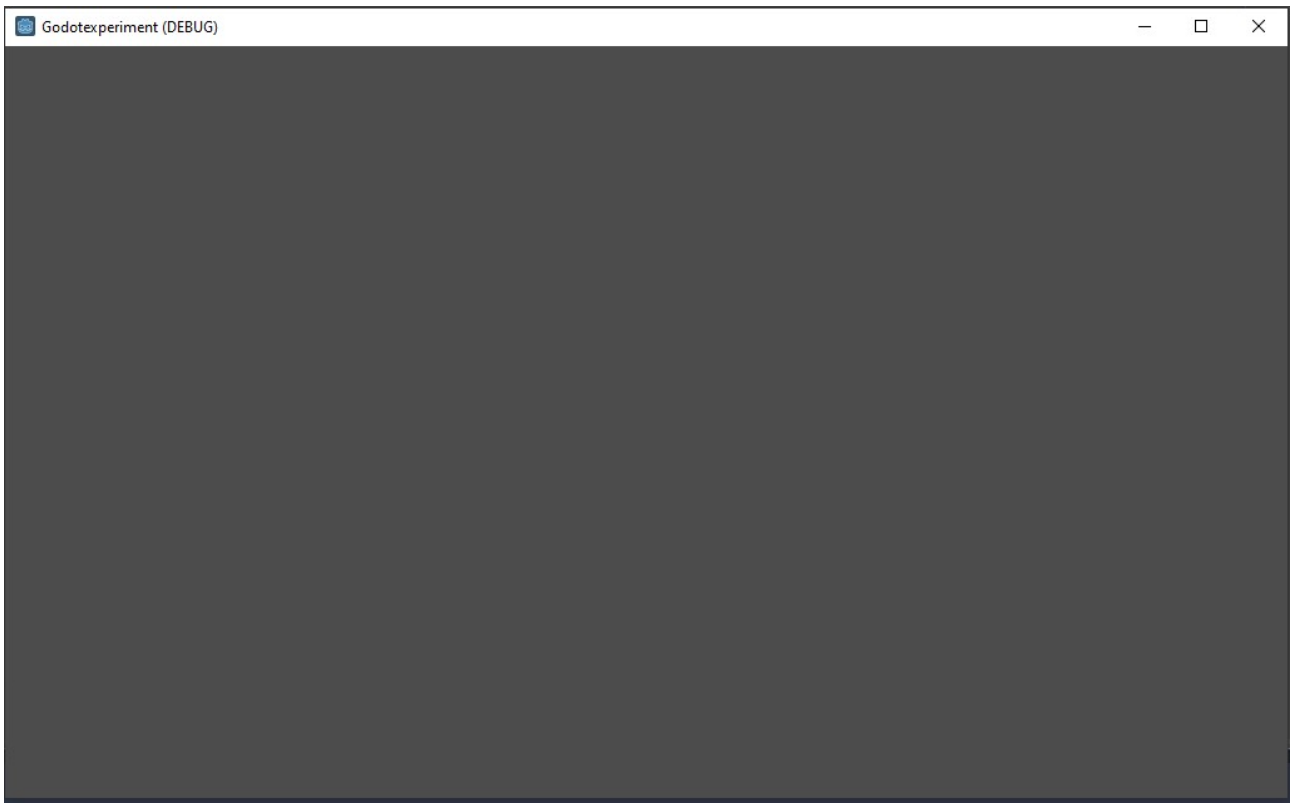


Godot will pop up a window asking us to select a main scene:

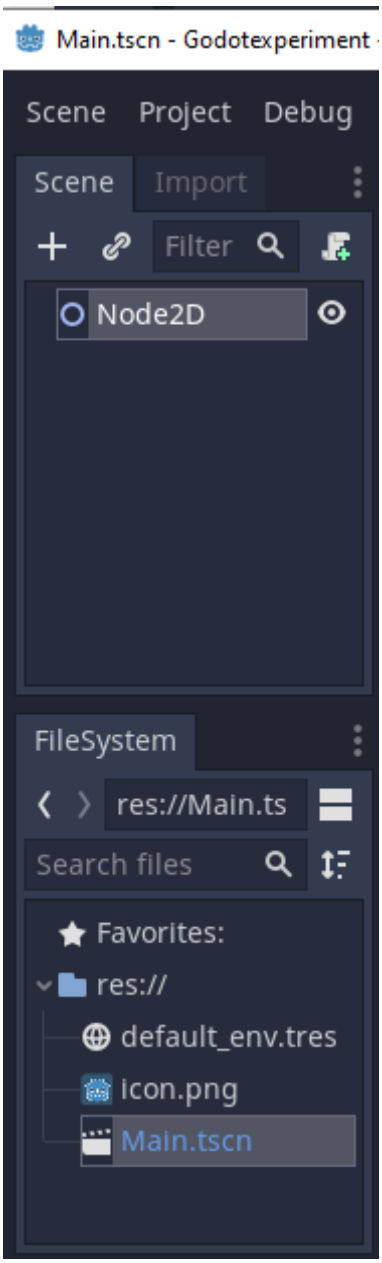


Since this will be our main scene just click Select Current.

Now the game will launch, but since everything is empty there will just be a gray window:

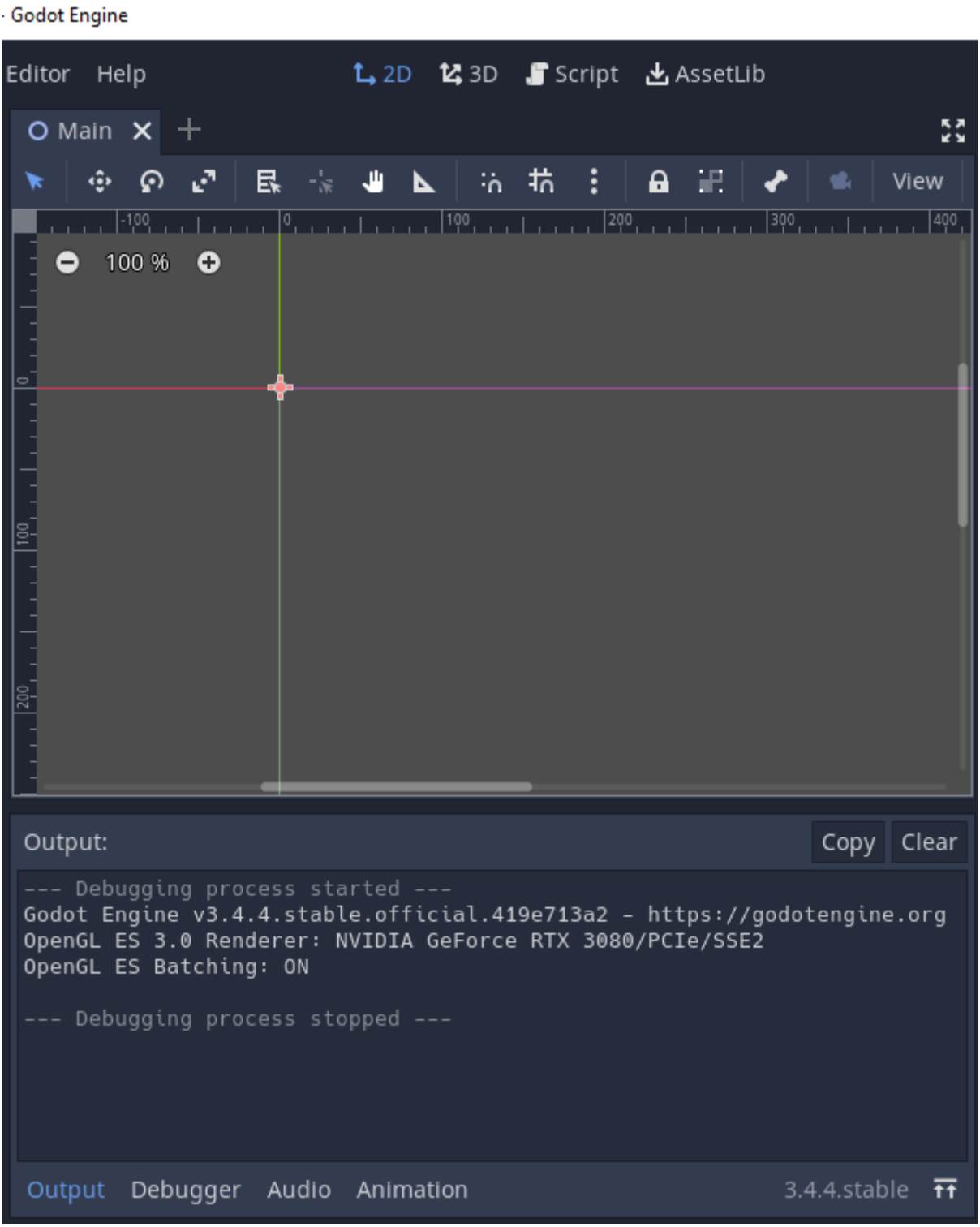


This is fine because we have no content yet. Just close this window to end the game and let’s take a very quick look at the editor.



On the left side we have the current Scene tree which right now just contains our Node2D. In the same window in another tab are the import settings for resources, but we will get back to that later.

Below the scene tree we have the FileSystem which is basically the folder structure under the project folder. We will also take a deeper look into this later on.



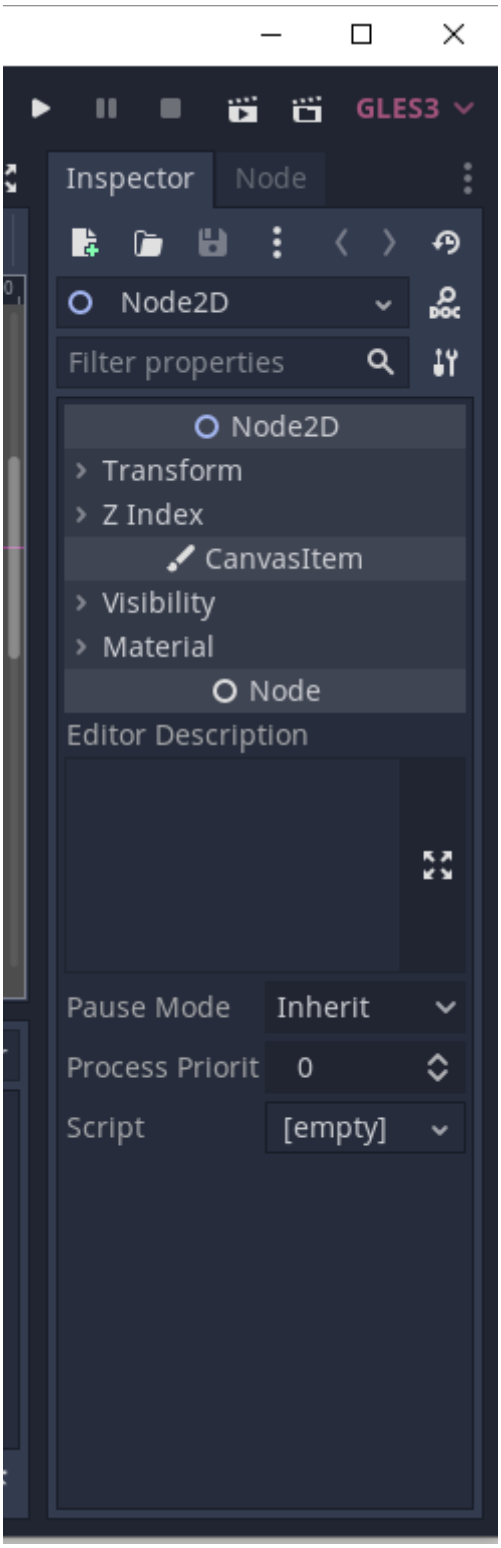
In the middle, going from top to bottom, we have our current editor view which is currently set to 2D. We won't be using 3D in this course, but if you work on 3d games you'll be using that. Then there's the script view which we will be looking at a little later. Last we have the AssetLib which we're not going to be using but can be very helpful so I recommend looking into it on your own time later on.

Below that we have our tabs of currently opened scenes and scripts, currently only Main is open.

Under that we have a toolbar with various tools that we will look at closer a little later.

Then we have the 2D view window which currently shows the location of our Node2D, on the sides are some rulers that help us determine the position of the currently selected node. If you haven't moved it, it will be at 0,0.

Below that we have a panel that can be switched between Output, Debugger, Audio and Animation. We will look at these later as well.



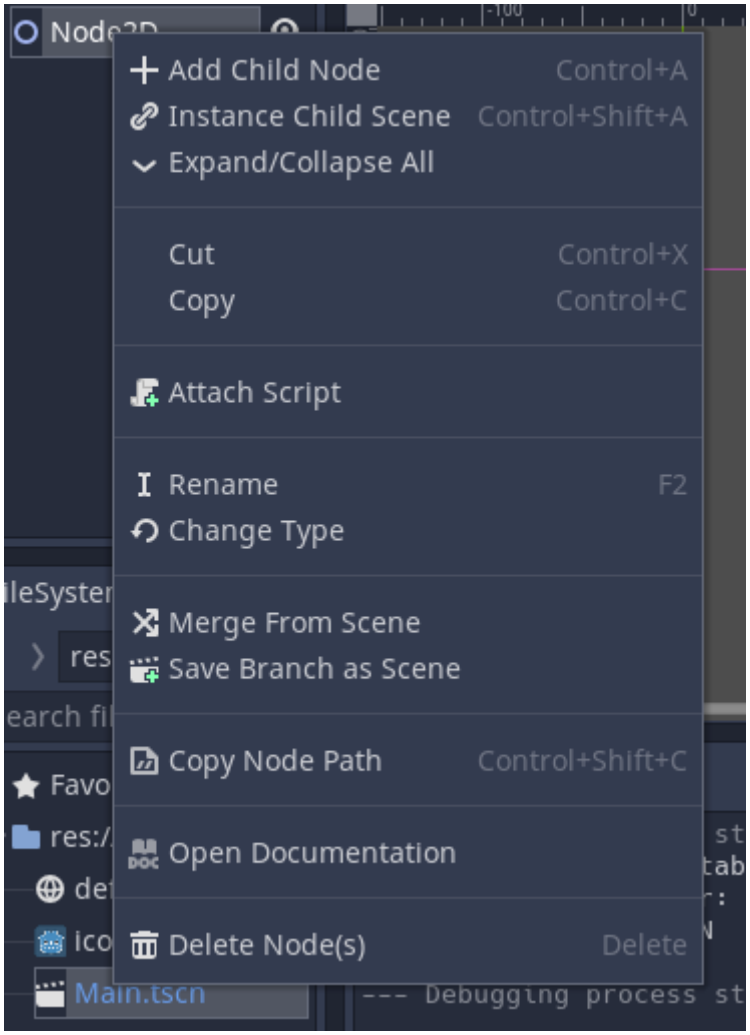
Lastly, we have the Play, Pause and Stop for the project and also the possibility to only play the current scene and to play a custom scene as well as the option to change our renderer from

Forward+. We will not be using these last two options in this course though.

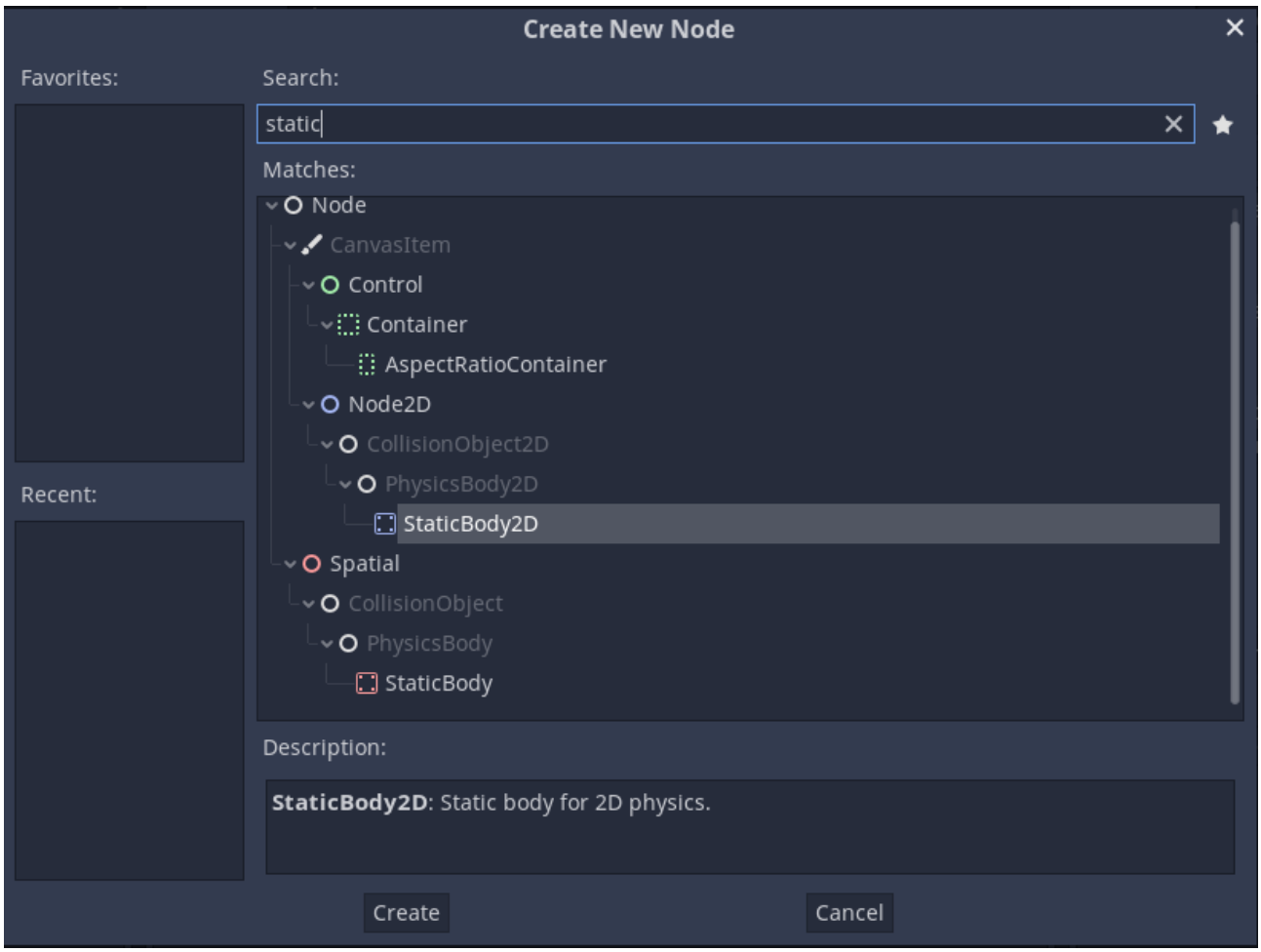
Below that we have the Inspector pane which shows us information on our currently selected item and the second tab, node, contains information on signals which, again, we will cover later.

With all that done, lets actually do something here.

Right-click on Node2D in the scene tree and click “add child node”



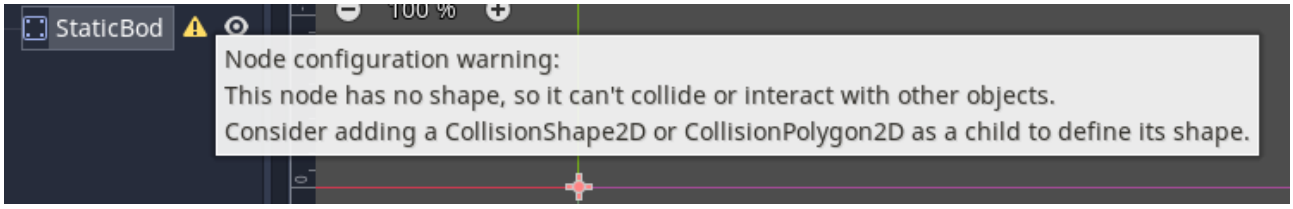
This will open up a screen where we can search through all the various nodes we can use. Let’s add a Static Body, simply type in “static” in the search bar and it should show up:



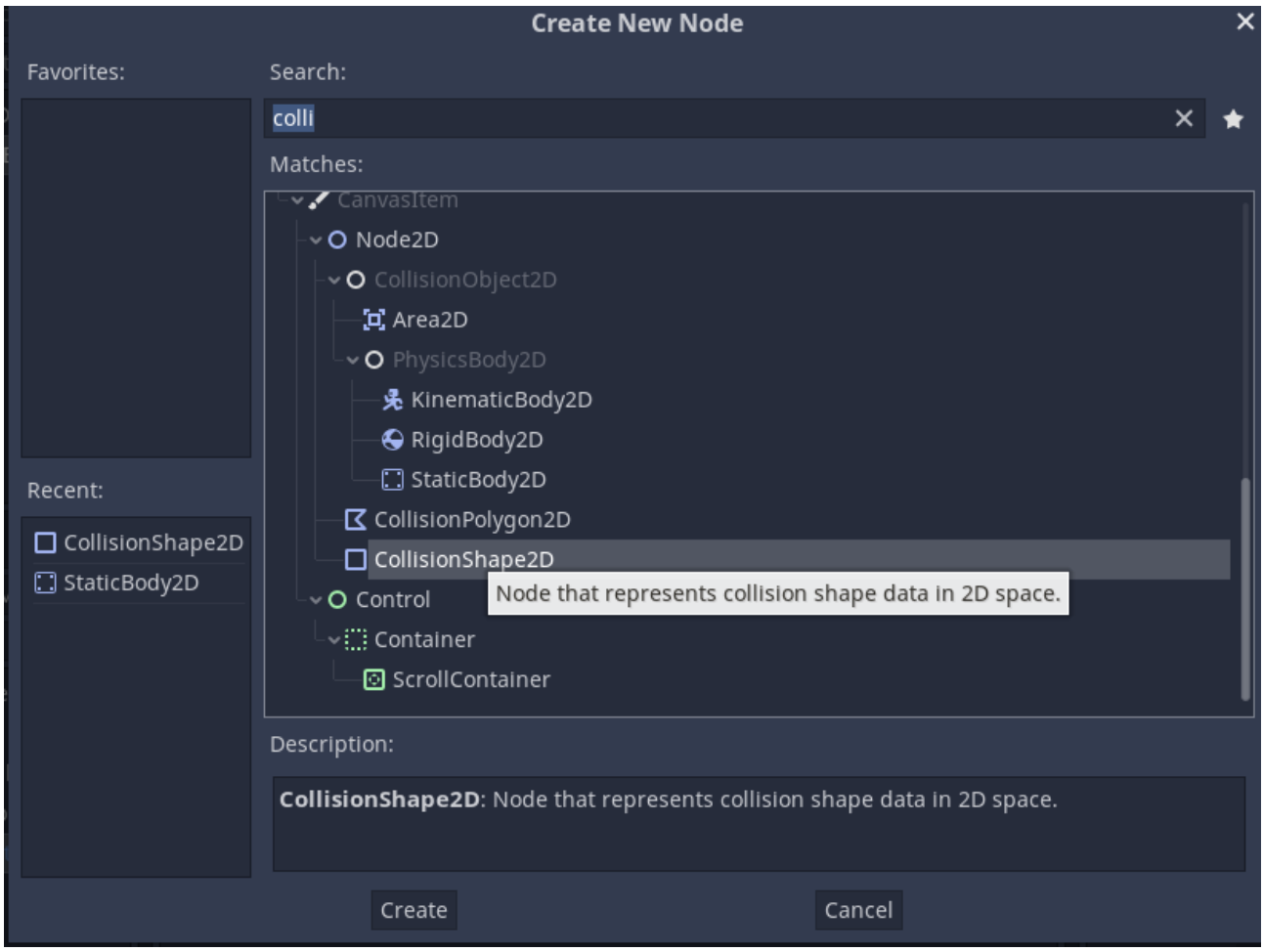
We are working in 2D so we want StaticBody2D. Select it and press “Create”. Underneath our Node2D in the scene tree we should now have a StaticBody2D:

Now what is a Static Body 2D? It is basically a physics object used to detect collision that can only move if moved by a script, though this is not recommended.

The StaticBody2D node has a warning next to it, which usually means something is wrong. By hovering our mouse over it we can see what the issue is.

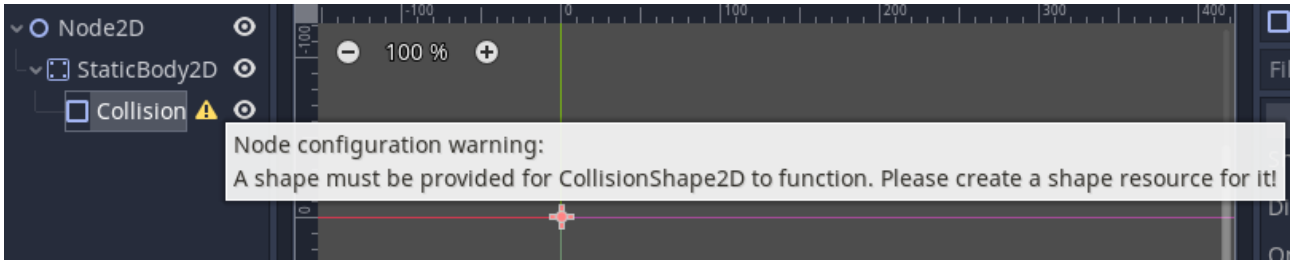


This warning is expected at this point. To fix it we need to add something that tells the node the shape of it. So right click the StaticBody2D node and click “add child node” and we will add a CollisionShape2D:



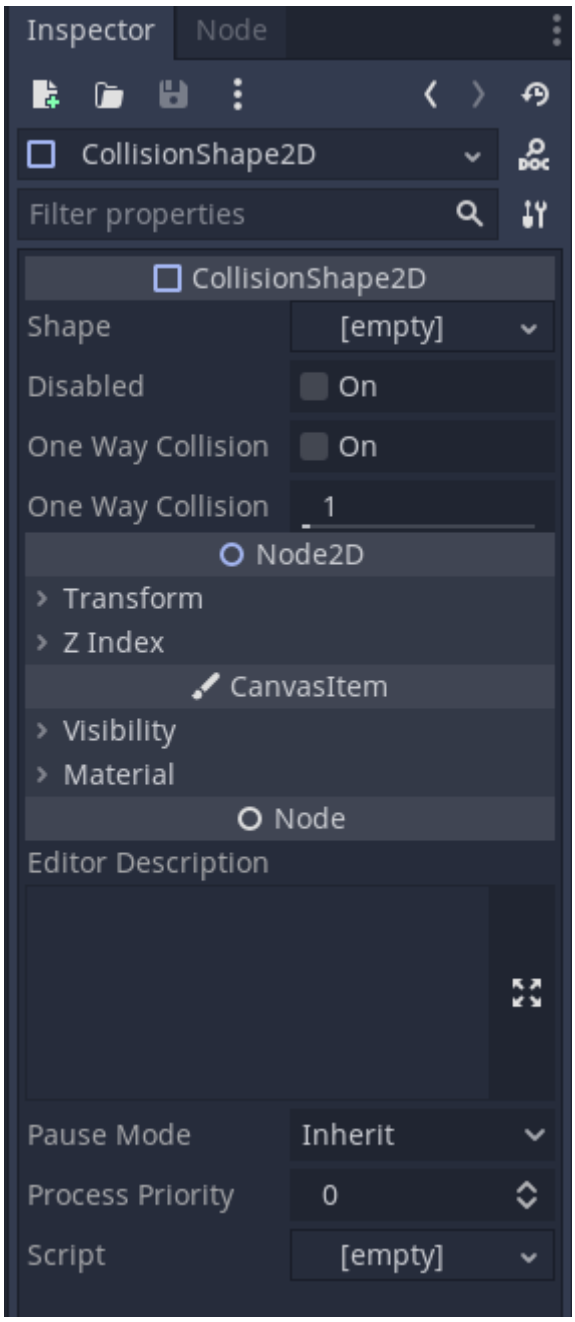
We can simply search for “colli” and it will suggest the node we want.

Now the scene tree should look like this:

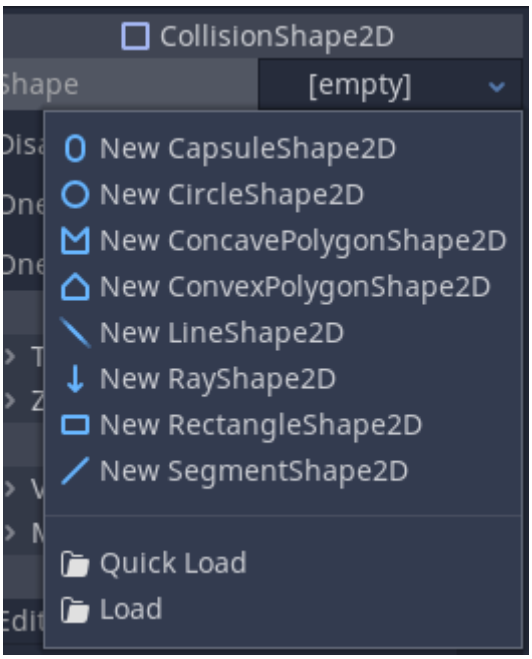


Once again there is a warning telling us something is wrong.

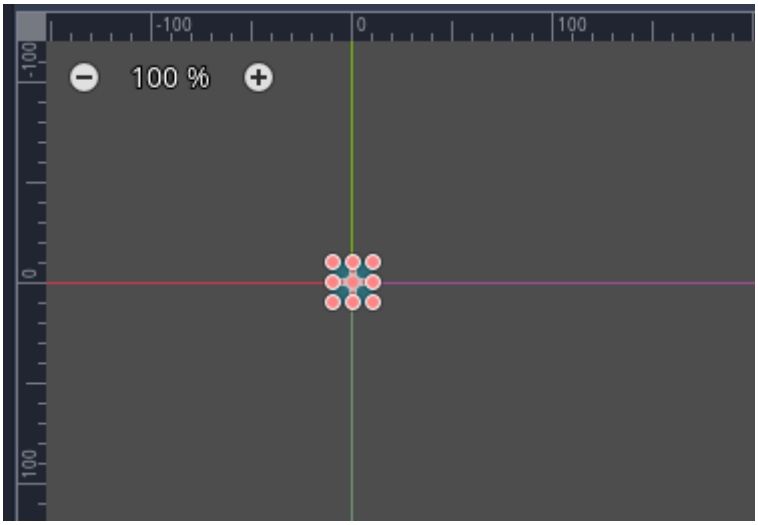
Select the CollisionShape2D in the scene tree and look at the inspector pane on the right:



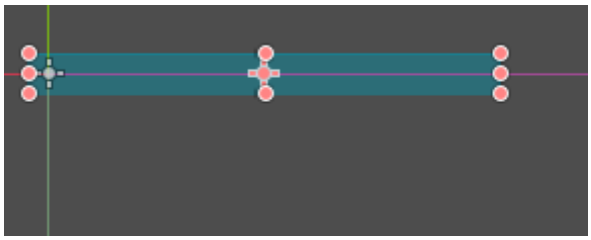
Our issue right now is that shape is empty. If we click on the empty we get a drop down menu with the option to create a new shape:



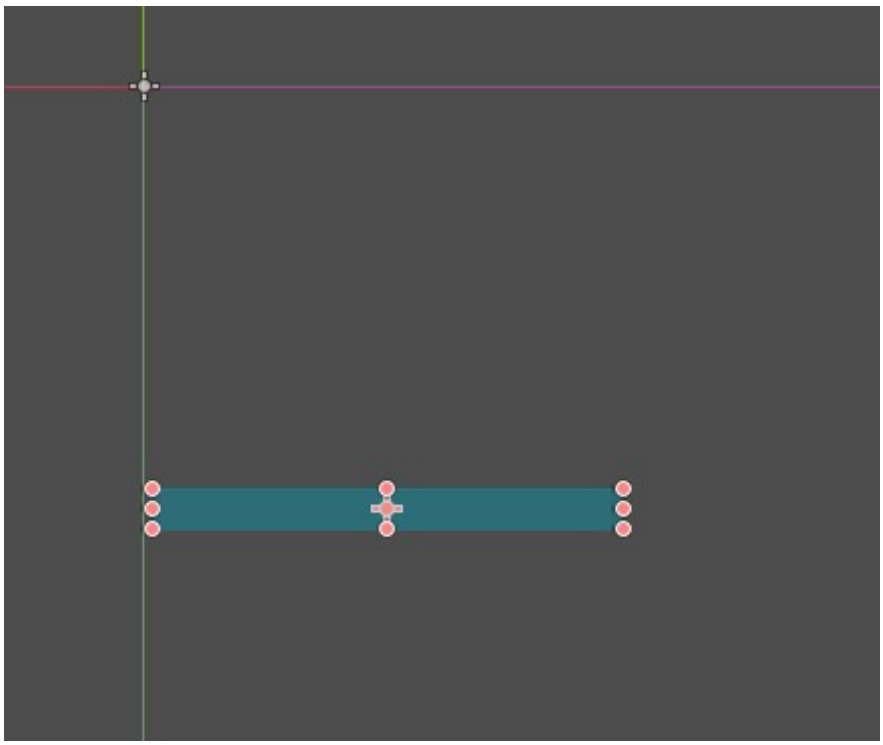
Lets make a New RectangleShape2D. Now we get a square in the main viewer:



We can change the size of this box by dragging the boxes around, lets make it a little wider:



I’m also going to move it down a little bit by dragging the box down.



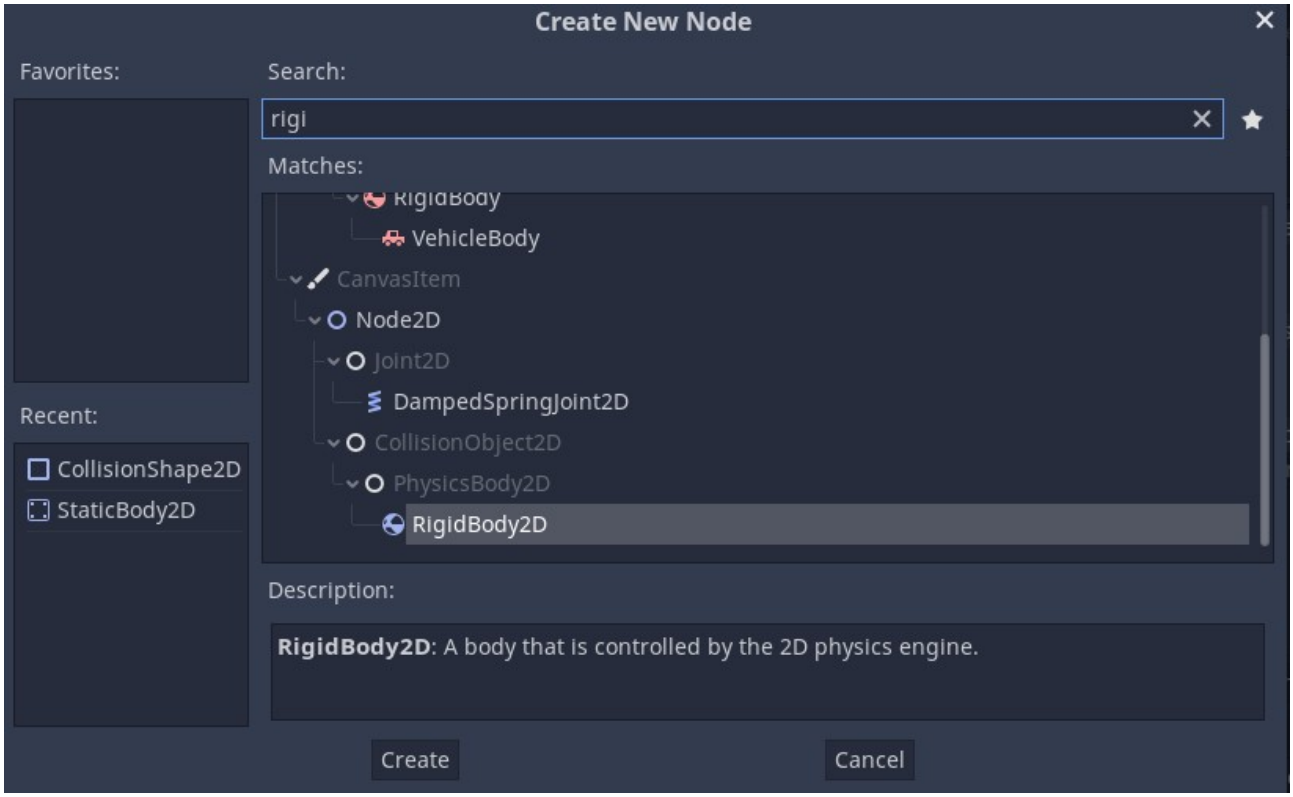
The little crosshair looking thing in the middle of the box is the boxes position, while the greyed out on in the top left of the image is the Node2D and the StaticBody2D's position, because we only dragged down the Collision shape node. If you pressed the crosshair you might have dragged something else down.

We can check the position of nodes by selecting them and checking the inspector pane. Just expand the Transform and you can see their position in the 2d coordinate system:

You can more exactly manipulate your positions from this if needed.

So now we have a box that doesn't move. Let's add something that will.

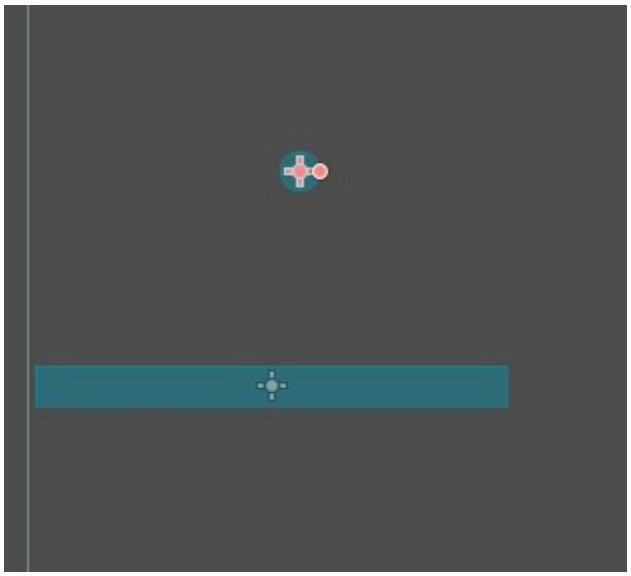
Right click on the Node2D in the scene tree and click Add Child Node, we'll add a RigidBody2D this time. It is important we right click the Node2D so the new child node gets added to the right place:



As the description says, this is a node that will be moved around by the physics engine of Godot.

We need a CollisionShape for the RigidBody2D as well, so right click on the RigidBody2D and add a CollisionShape2D. We'll make this into a ball, so once you have the CollisionShape2D click on the empty shape and select new circle shape.

I'm going to drag the new shape a bit closer to the static body:

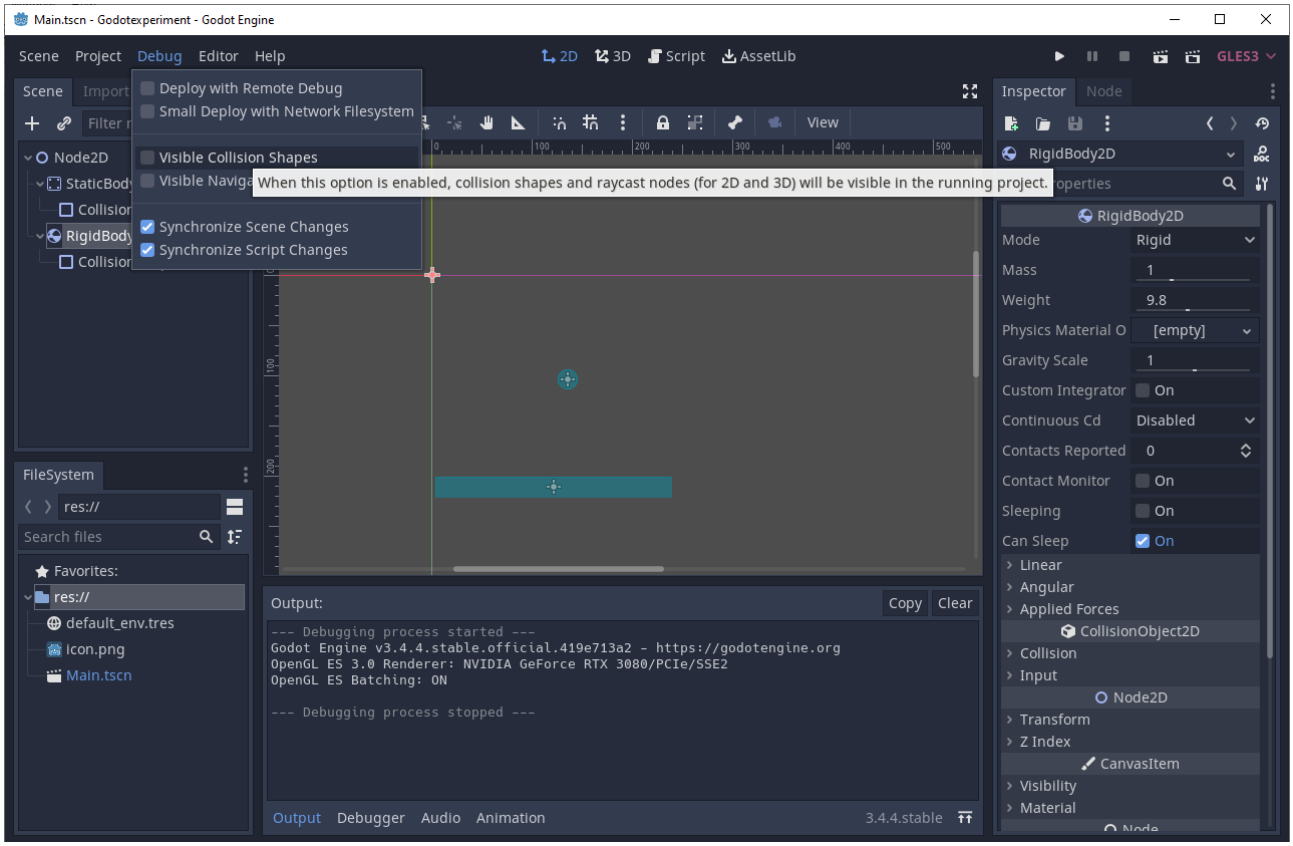


Now we can see how physics interact by pressing F5 to run the scene or by pressing the play button in the top right corner.

But hang on, it's all gray?

That’s because we don’t actually have any graphics yet. In the editor we see shapes, but these are just the shapes of the colliders. If we wanted graphics we would need to add them as child nodes to the static and rigidbody nodes.

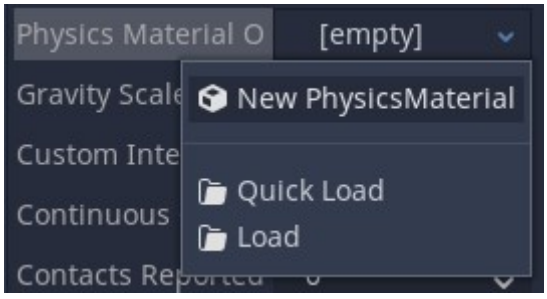
However we can, for now, circumvent this by enabling visible collision shapes in the Debug menu:



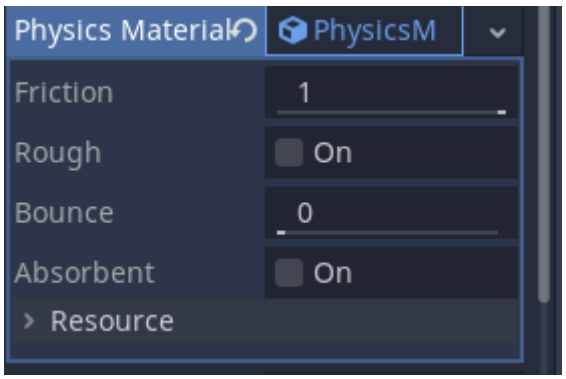
Now if we play the scene again we should see the ball fall and a red dot where it makes contact with the static body:

Now we can play around a little bit. Close the game window and select the RigidBody2D in the scene tree, then take a look at the inspector pane to the right:

We can modify a lot of different things here, the mass for example. But we can get a bit more if we add a physics material:

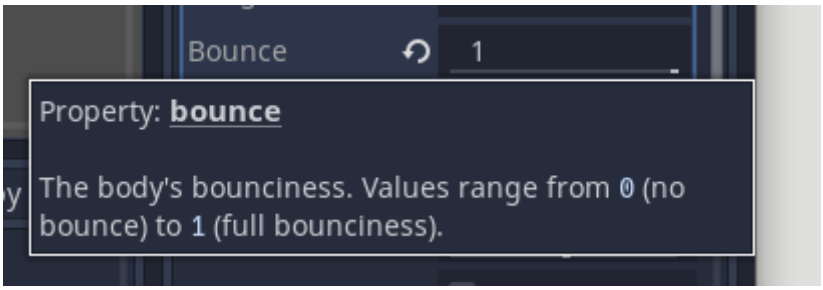


Now we need to click on the new physics material to expand its options:



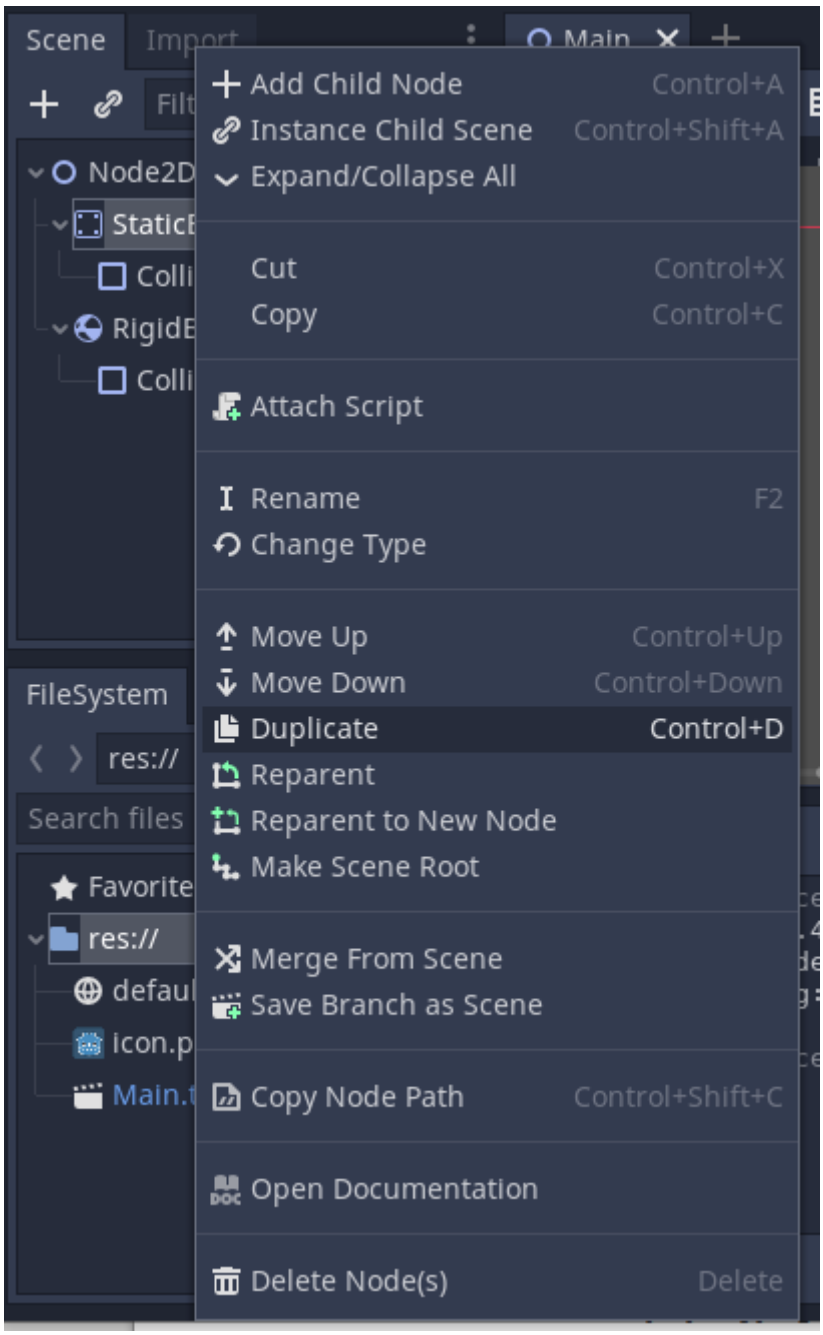
What I want to add here is Bounce, to make things a bit livelier:

One tip, you can hover the mouse over things to get a tooltip for what they are:



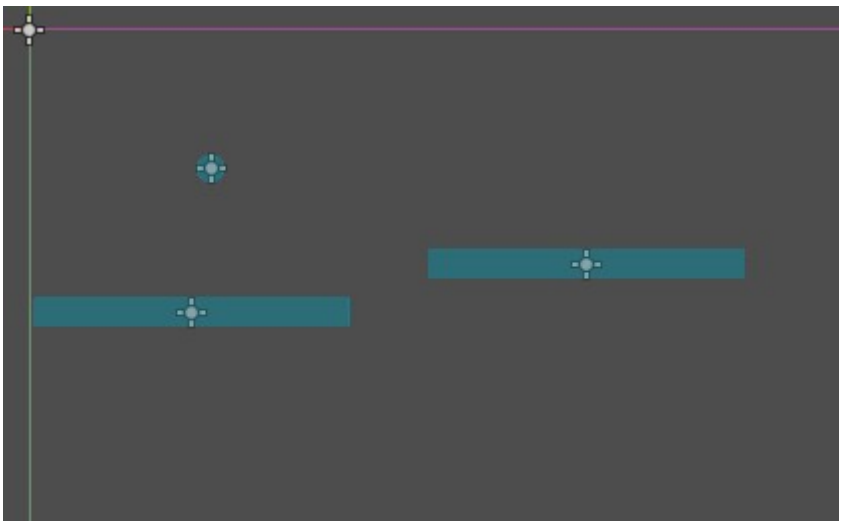
Now if you play the scene the ball should bounce depending on how much bounce you added. 1 is a bit much, but we're just having fun.

Let's add another static body by copying the existing one. Simply right-click on it in the scene tree and click duplicate:



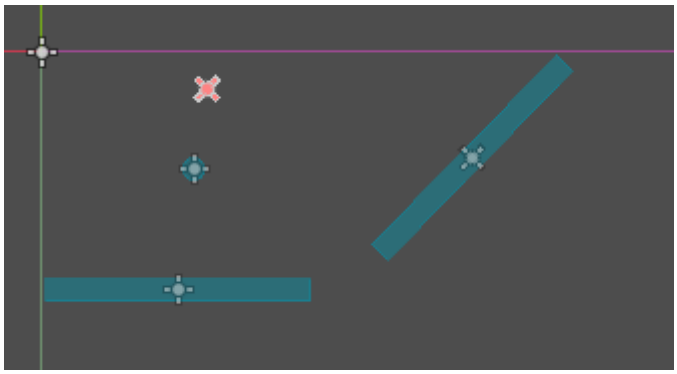
Now we can move the new static body, but lets take a look at something as we do this. On the top of the 2d window there are these tools:

We're taking a look at the first four. The cursor one is select, the second one is move, the third one is rotate and fourth one is scaling. They also have shortcut keys, QWES respectively. So I'm going to move the new static body by pressing on the move tool and moving it a bit off to the side:



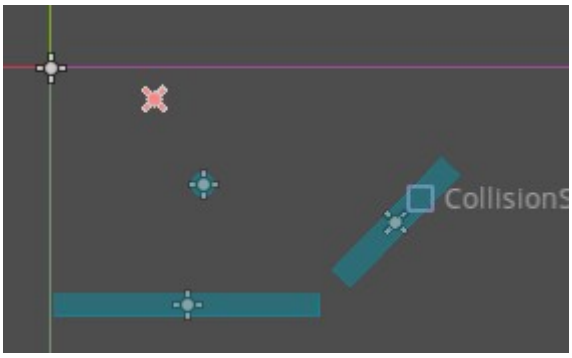
Another tip, you can move around in the 2d view by holding the middle mouse button and dragging the view around. You can also zoom in and out with the scrollwheel.

Next I want to rotate the new static body, so I'll use the rotate tool:

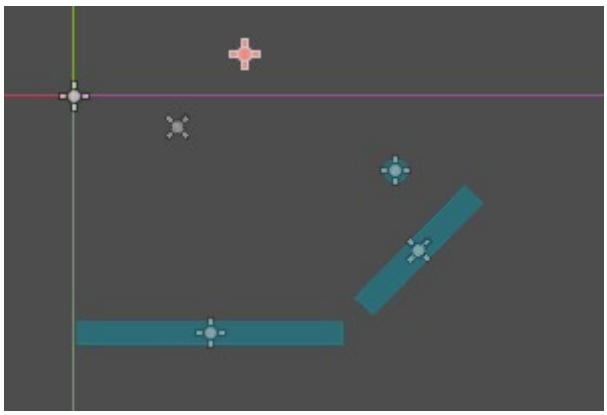


The rotate tool may not work exactly as you expect depending on where things are, but I will go through this in a bit more detail later.

Lastly there is also the scale tool, so I will use that to change the shape a bit:



Lastly I will select the rigidbody and place it over the new static body:



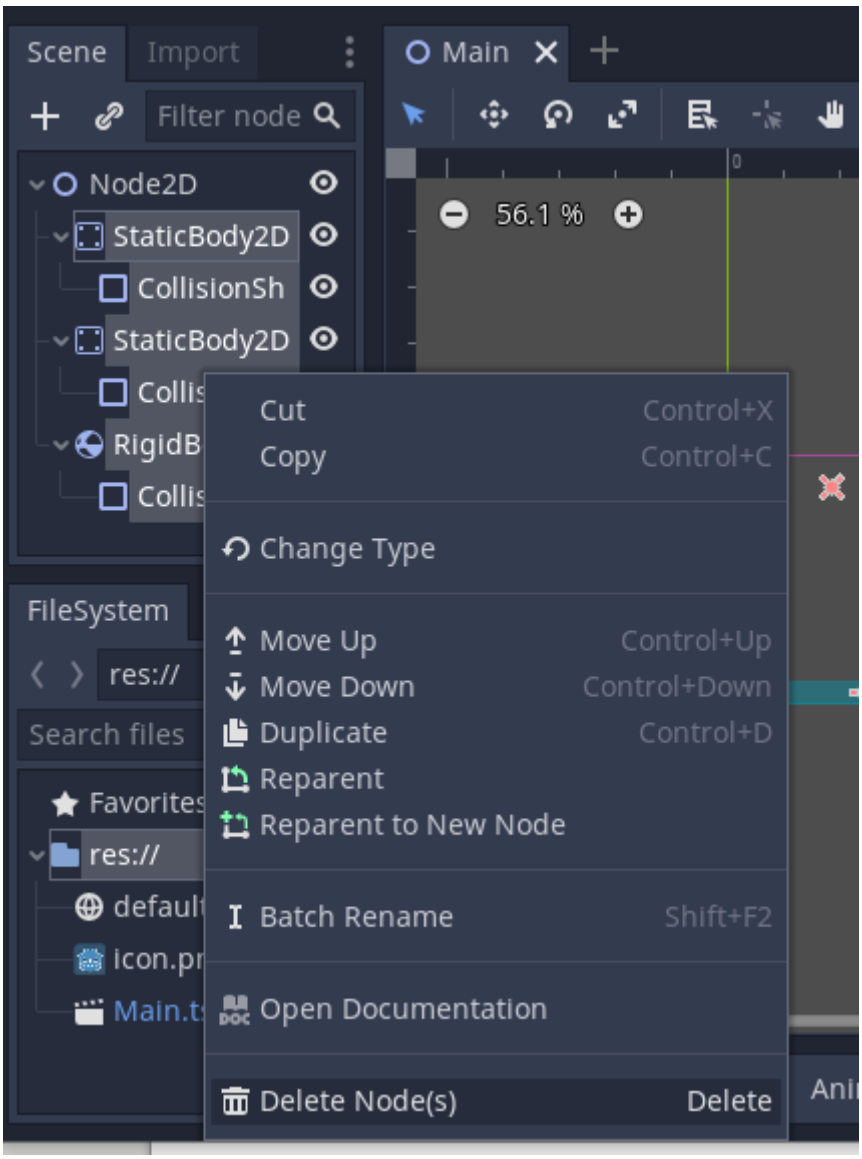
Now when we play the scene it’s a bit more dynamic.

However there are a few issues I have here. One is that the Origin of most objects are not matching where the collision shapes are. This means that scaling and rotating doesn’t work quite right as it is based on the location of the origin instead. If we instead had moved and scaled the collision shapes we wouldn’t have this issue, but things could get messy in other ways.

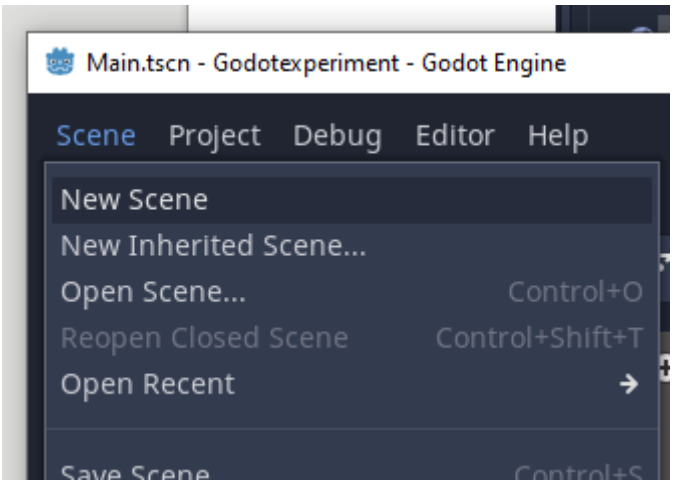
Another issue is that if I want to change a property in all of the static bodies I’d have to click through on each of them and modify it. Not a problem if we only have two of them but it can quickly become hard to manage.

So lets take a look at making Prefabs:

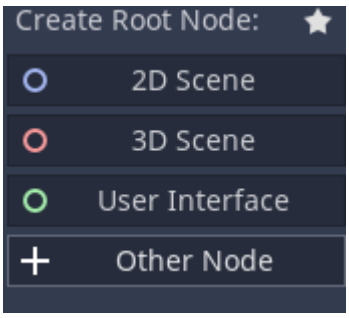
I’m going to start out by deleting everything except the root Node2D node in the scene view:



Now we have an empty scene again. This time I'm going to create a new scene, so from the Scene menu up in the left I'll select New Scene:



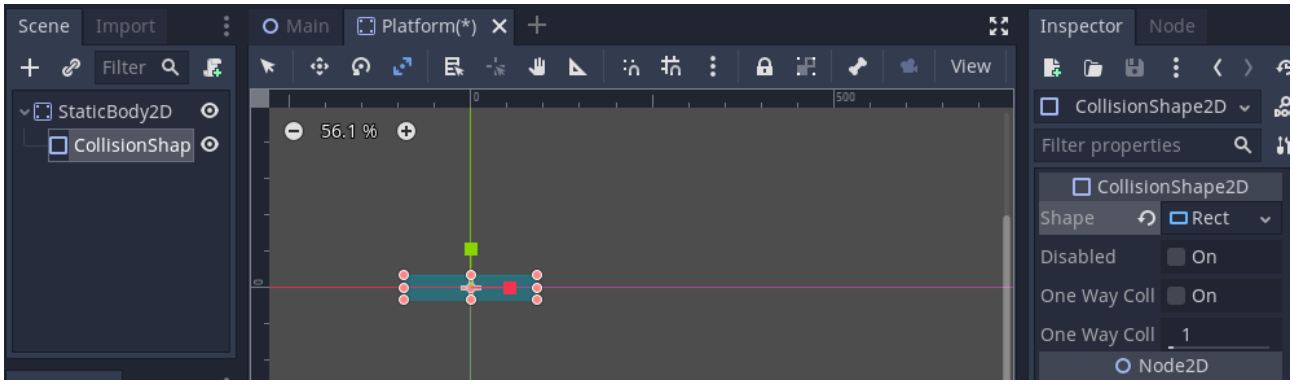
It asks us for a root node again, select Other Node:



Let’s make a StaticBody2D

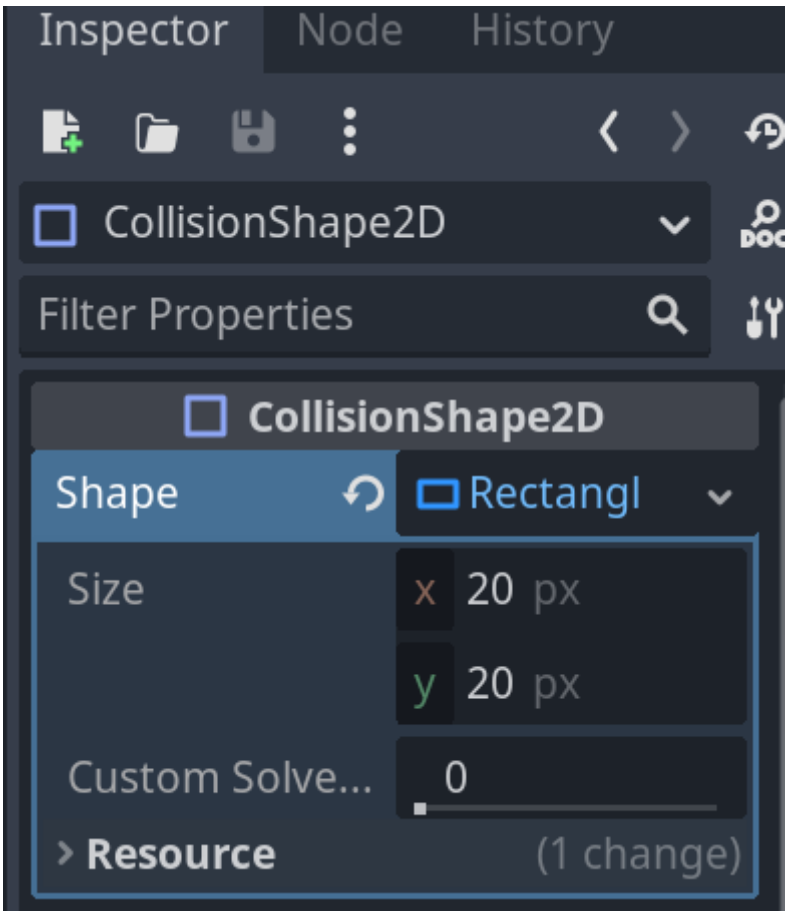
Before I do anything else I will save this as platform.tscn by going to Scene→Save scene or pressing ctrl + s

Next I’m going to add a CollisionShape2D for the static body, then add a rectangle shape to that in the inspector pane and scale it up slightly, using the scale tool on the top of the 2d view window:



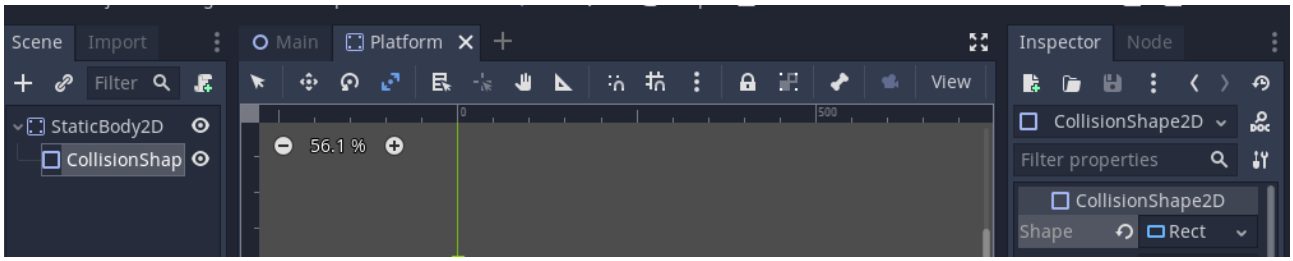
This time the center of the rectangle is in the middle of the lines that denote position 0, 0 in the editors co-ordinate system. You can check this in the inspector pane under Node2D → transform:

There is a better way to change the size of the collider though, by expanding the Shape properties under CollisionShape2D you can get precise size controls in pixels:



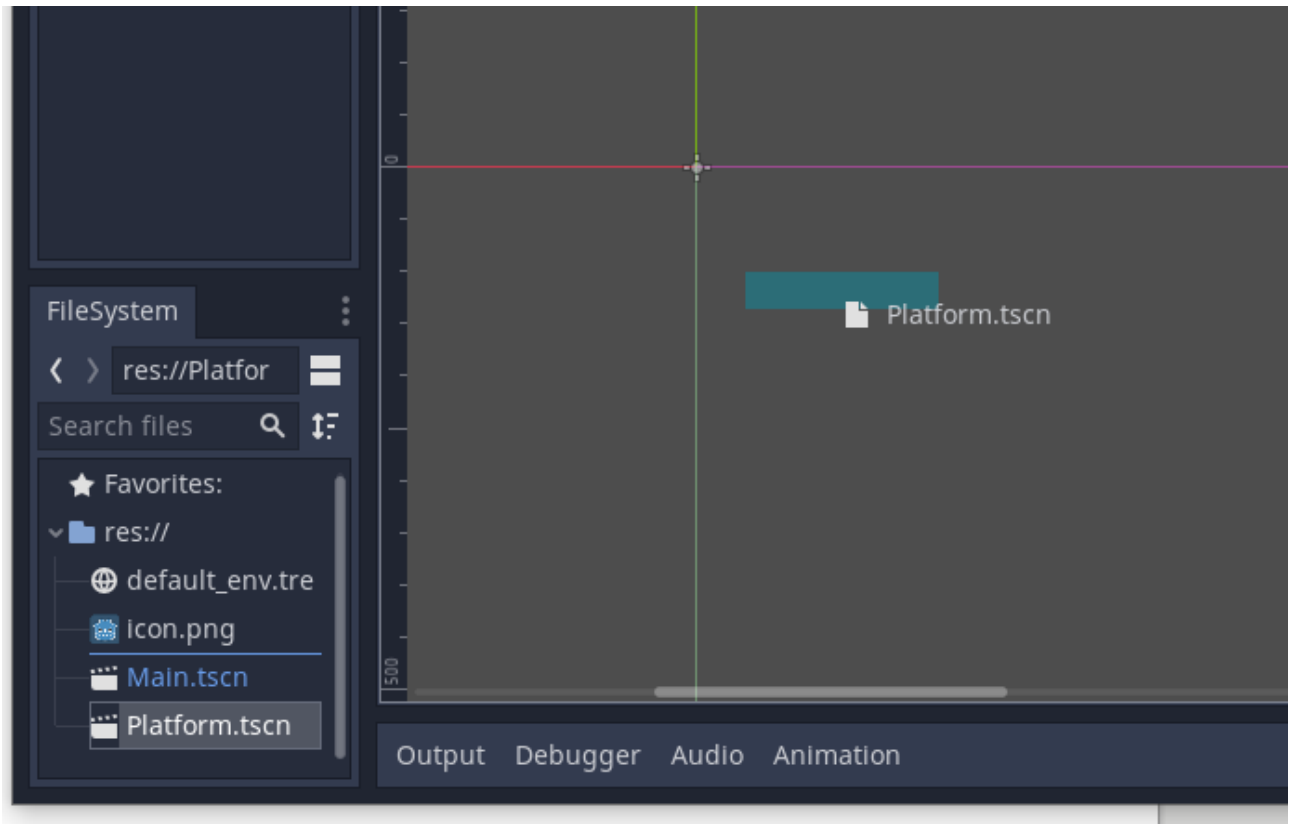
Here we can change the size to be exactly what we want it instead of messing around with scaling factors which may end up causing other issues later on. So I will reset the scale in the transform to 1 and change the size in here instead. I'm going 240 pixels on x and keeping y on 20.

I'm going to save the scene again and open up the Main scene. You might have noticed that above the 2d view window there are tabs that now contain our two currently opened scenes:



We can easily switch back to the Main scene by clicking on its tab or by double clicking on it in the FileSystem pane.

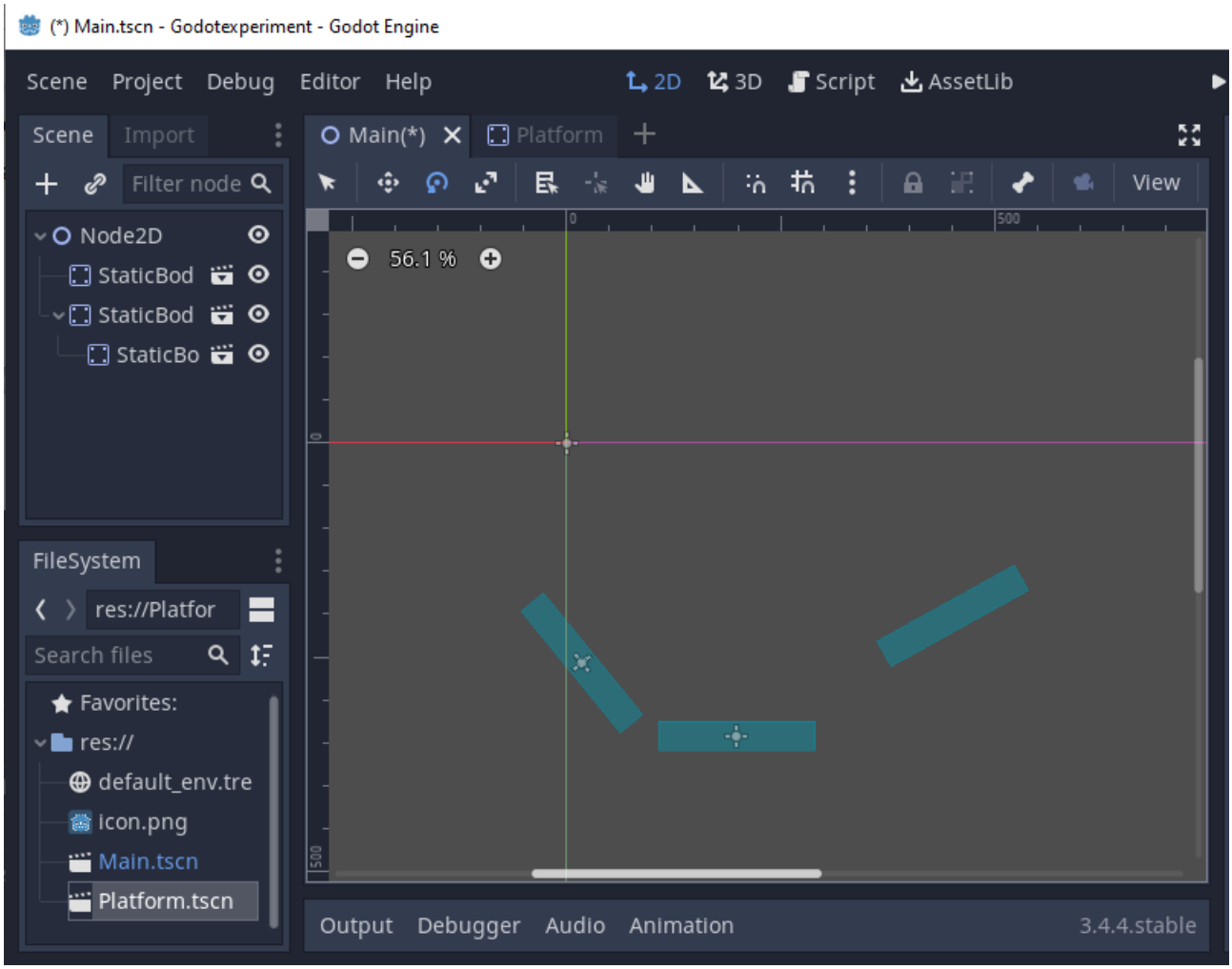
Now we can just drag the Platform scene into the main scene from the FileSystem pane like so:



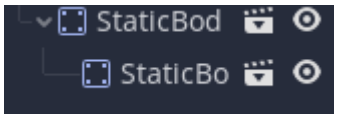
Then we can place it where ever by dragging it around.

You can also add them into the scene by dragging them into the Scene tree.

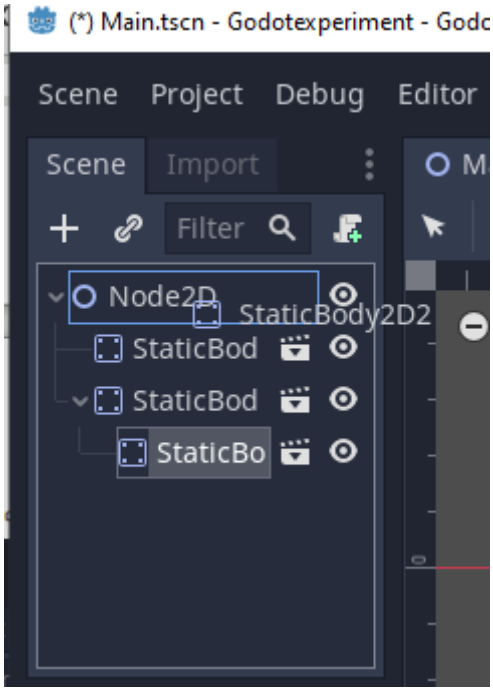
I’m going to add a few more and rotate them just for fun:



I’ve made a few problems for myself here, luckily all easily fixable. Lets start by looking at this:

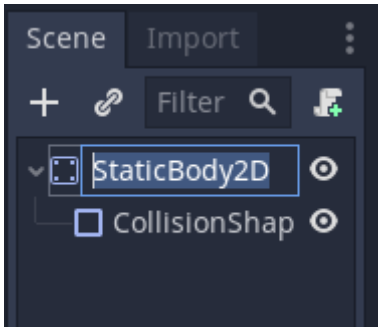


In this case the lower StaticBody2D has been placed as a child of the upper StaticBody2D. This means if I move the upper one then lower one will also move, which I don’t want. All I need to do to fix this is to drag the lower one under the Node2D in the Scene tree:



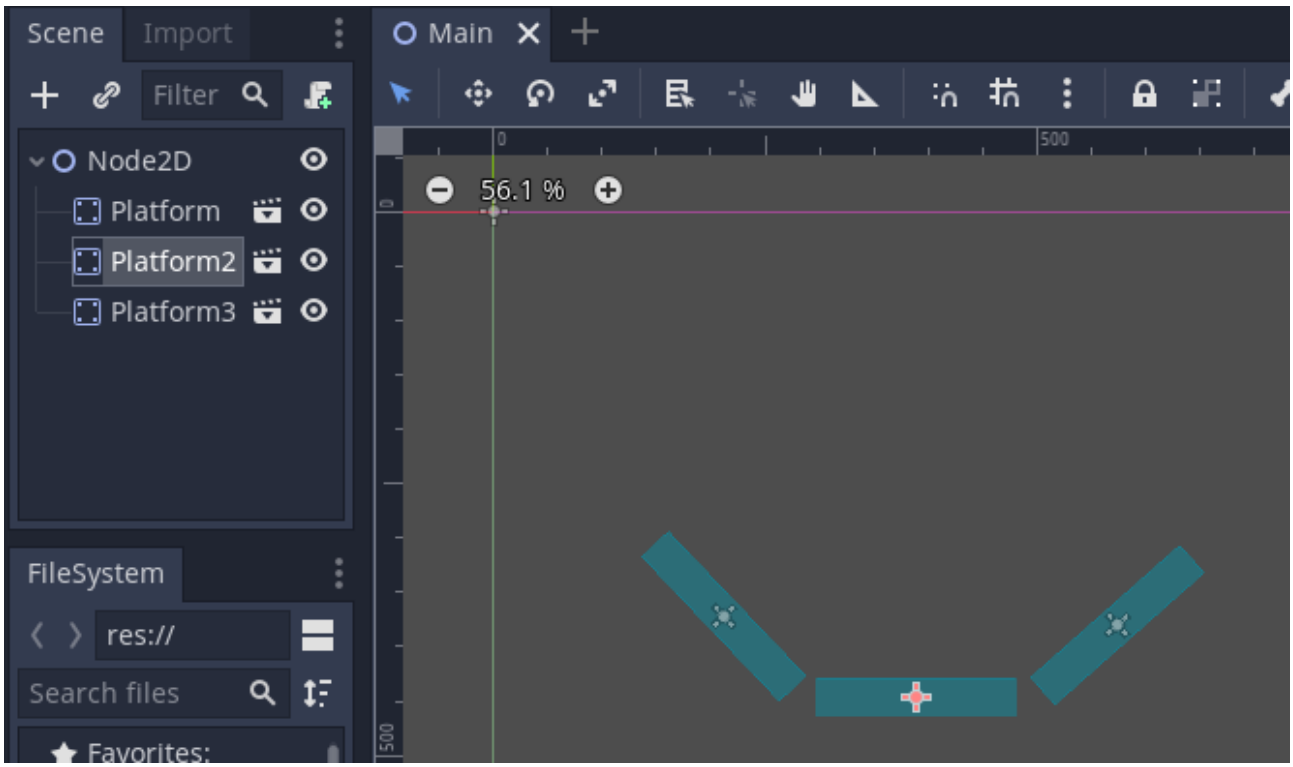
After this that issue will be fixed. The other issue is that right now the Platforms are named StaticBody2D which is a bit unclear. This is because I forgot to rename the root node in the Platform.tscn.

So I'll fix that by switching back into the Platform.tscn scene and renaming the root node by double clicking on it:



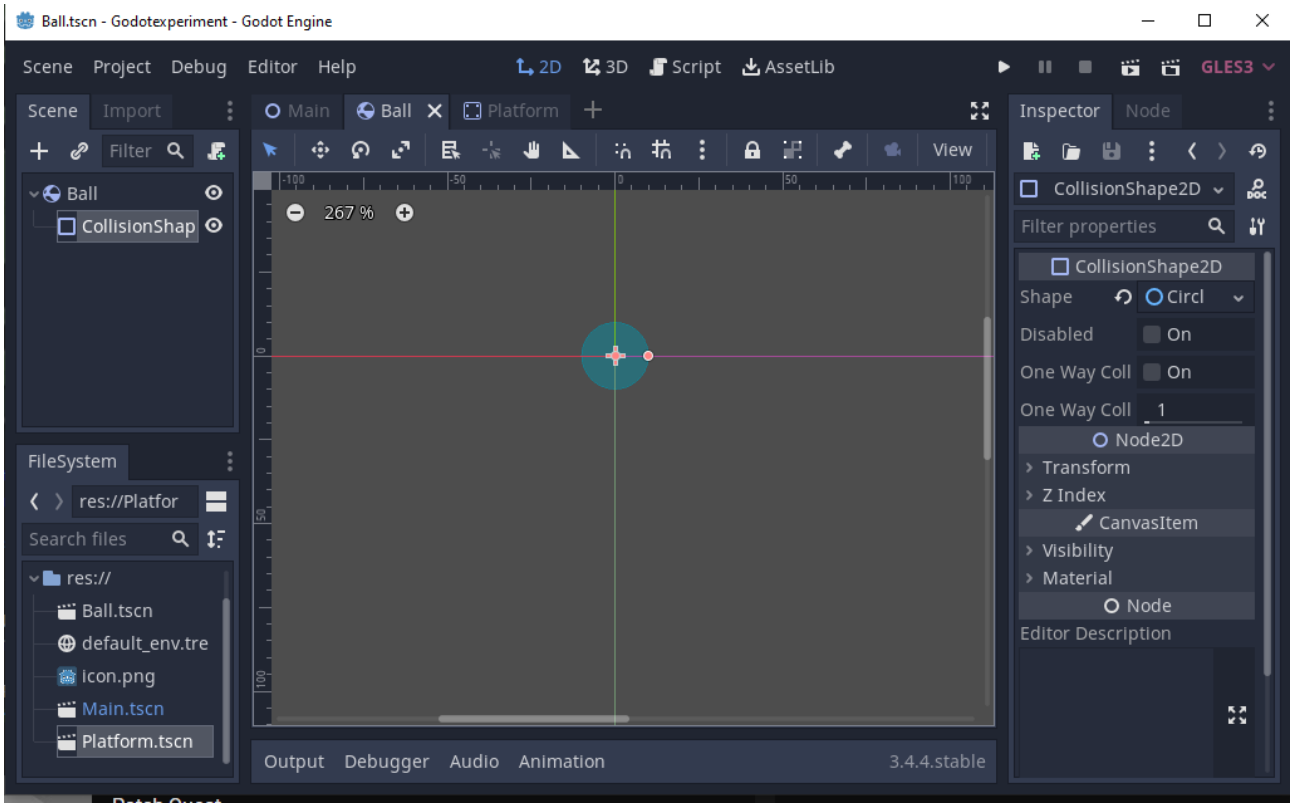
After renaming the root node to Platform any future Platform scenes we drag in. Unfortunately the currently placed ones will not have their names updated. You can either update their names manually by double clicking on the name in the Scene tree or by selecting one and changing the name in the Inspector pane.

I'm just going to delete my old ones and place in some more platforms.

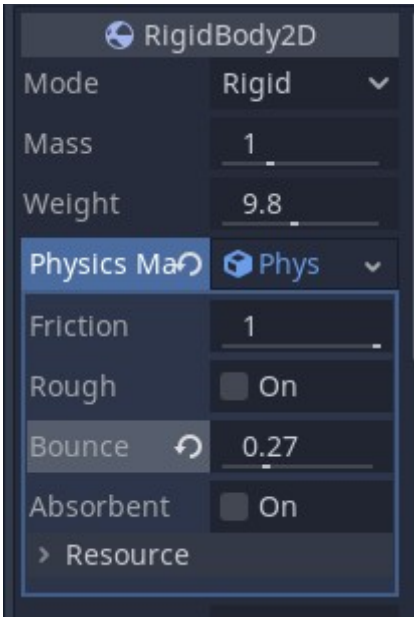


So, before moving on here, try to see if you can make a prefab for the ball. Make sure you use the correct root node!

Ok, I'll go and do this real quick. First lets make a new scene with a RigidBody2D root node, then add the CollisionShape2D and a circle shape to that. I'm also renaming the root node to Ball and saving the scene as Ball.tscn:



Next I’m also going to add a physics material to the ball root node and add a little bounce to it:

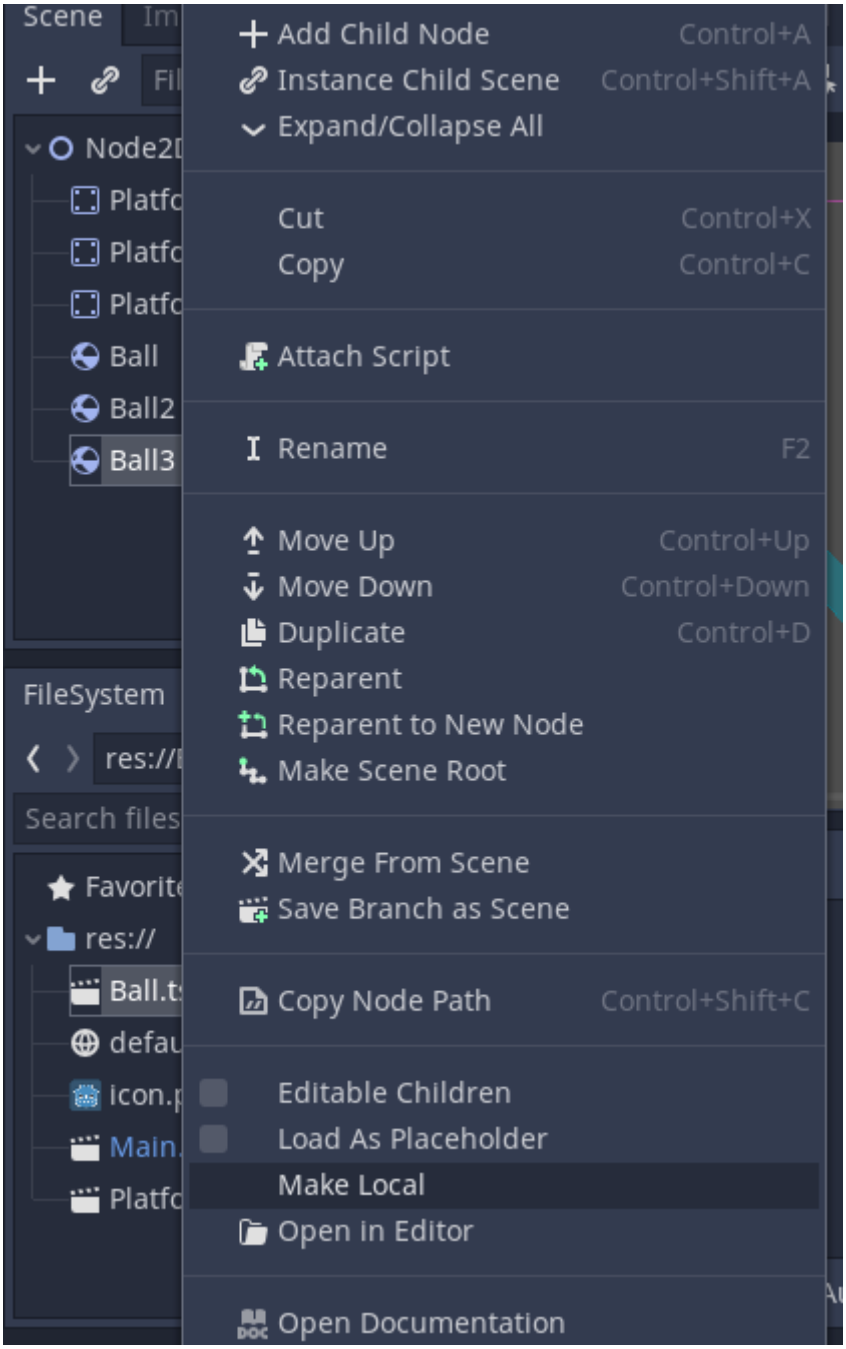


Now I’m just going to save the scene and add some balls to our main scene.

In my case after testing I feel like the balls need some adjusting, so I’ll go back to the ball scene and modify it a bit.

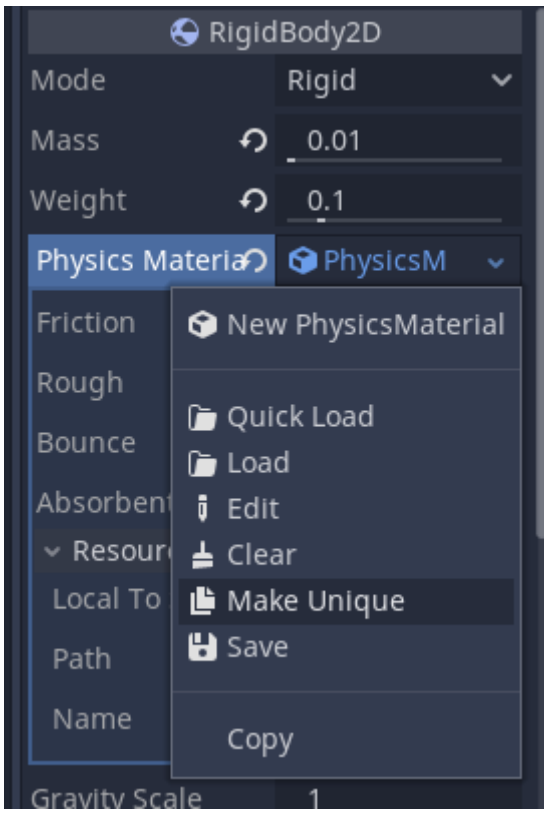
Now when modifying the prefab scene, the changes will also be applied to all the balls in the main scene. However if we want a prefab that does not change with the original prefab scene, we can

right-click on one of the balls and click make local:



This will make this scene no longer inherit from the prefabs so any changes made to the prefab will not apply to it. In our case it is however important to notice that the prefab balls and the now local made one still share the physics material, so any change made to that will apply to all objects using that material.

The material can also be made unique to the selected object however, so on our newly made local ball we can go into the physics material properties and make it unique:



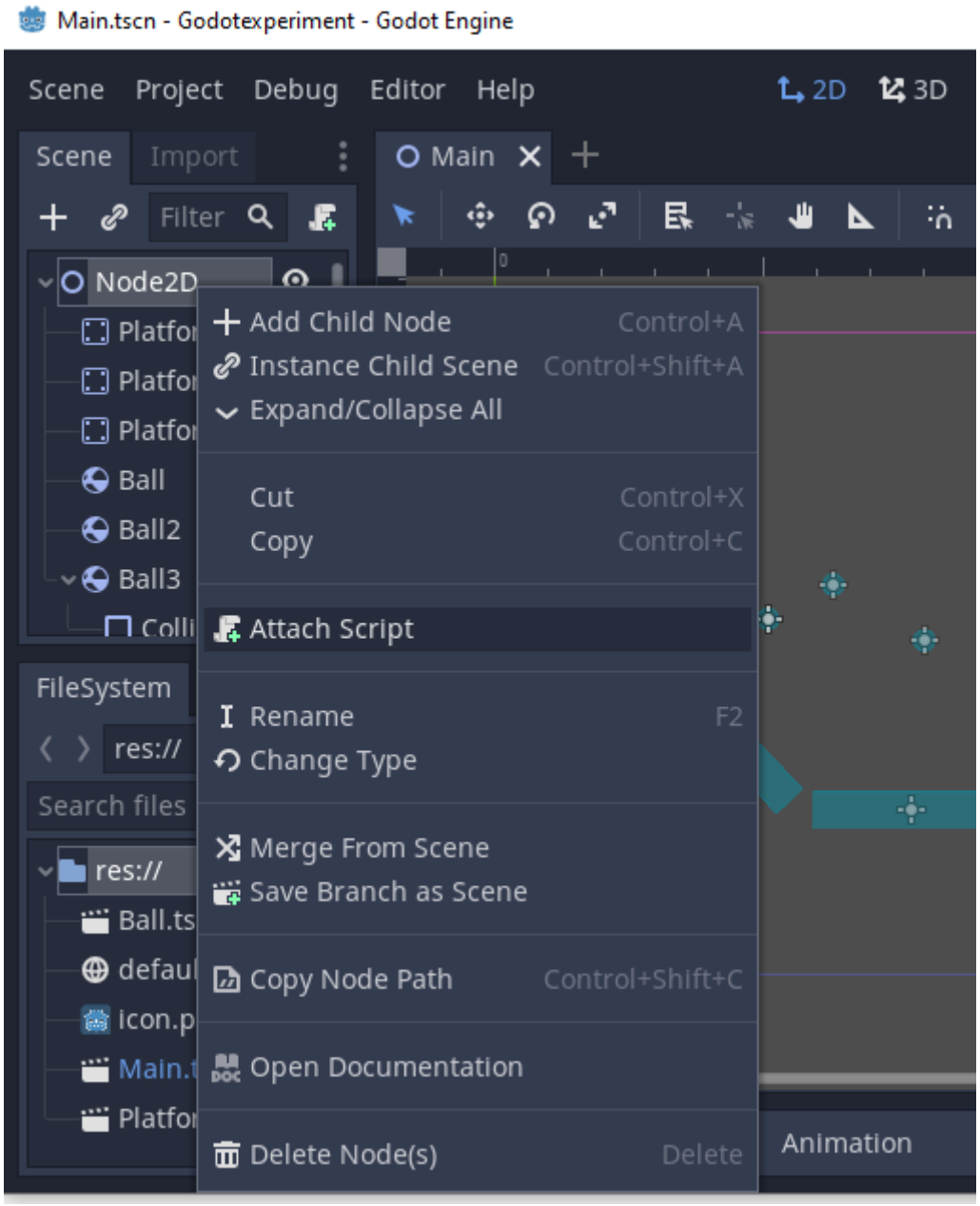
Now after this, this ball will share no properties with the prefabs. We can now do whatever we want with this ball, make it bigger, heavier, bouncier etc.

Ok, lets get into the meat of things with our next section.

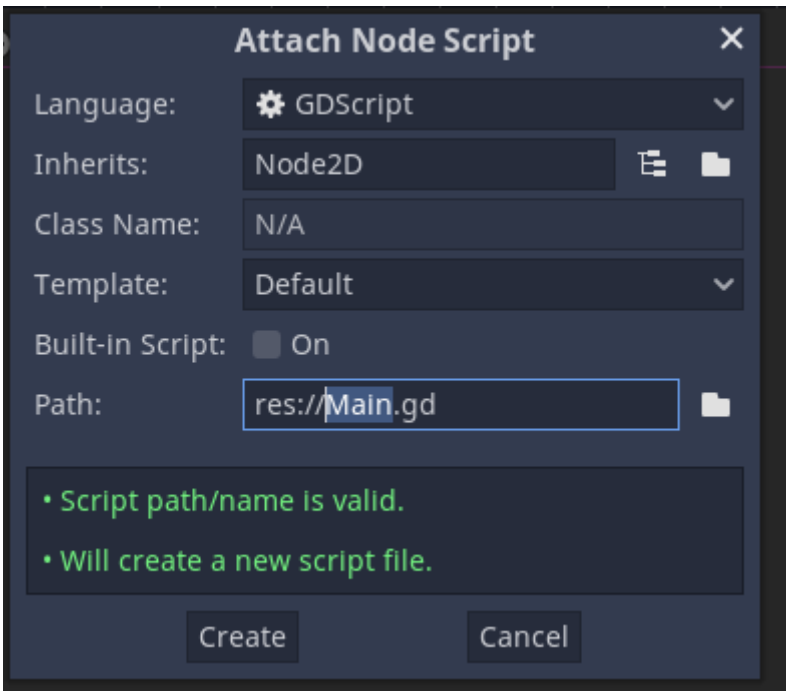
Writing Code

Let’s write our first script. To do this we’ll attach a script to the root node of our main scene. We will be writing this in GDScript. A reference for GDScript is available, this can be handy for later use: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

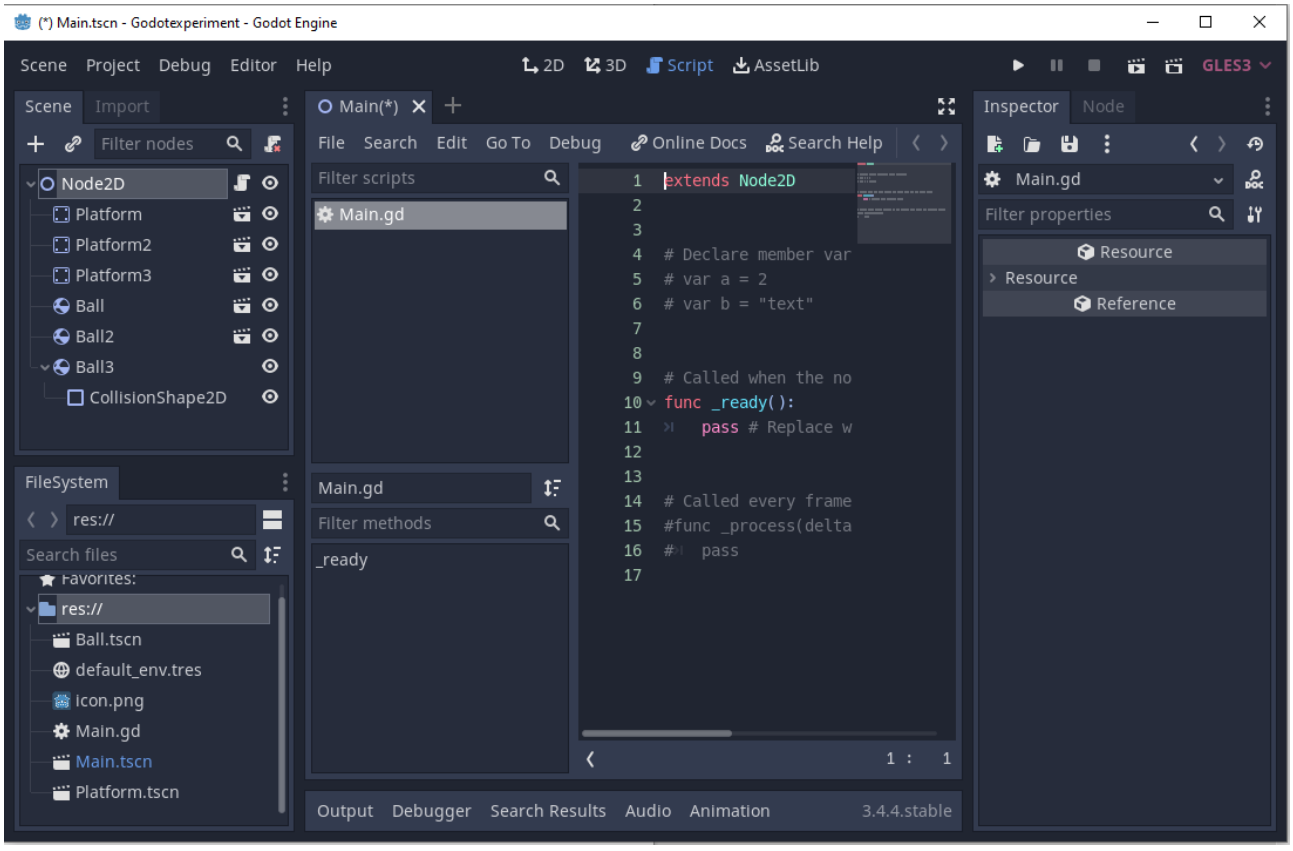
To begin right click on the root-node and click Attach Script



This will bring up the following screen:



Just press Create and you will be brought into the script editor:



The script is automatically created with the following line in it. You may get a built in script with more content, but you can delete all lines except this one:

1. `extends Node2D`

The first line here, extends Node2D tells the script what node type it is attached to, and as such

what properties and methods it has access to. These are all listed in the documentation as well:

https://docs.godotengine.org/en/stable/classes/class_node2d.html#class-node2d

Traditionally just about every tutorial I've read about programming or scripting has started with the "Hello World" example and in the interest of tradition that's what I will do here.

First, we need to add in a function called func _ready(): which is automatically run when the node the script is attached to gets loaded, so we simply write that in.

Under that we write: print("Hello World")

It is important to retain the proper indentation in the code as GDScript, which is the scripting language we're using here, takes that into account.

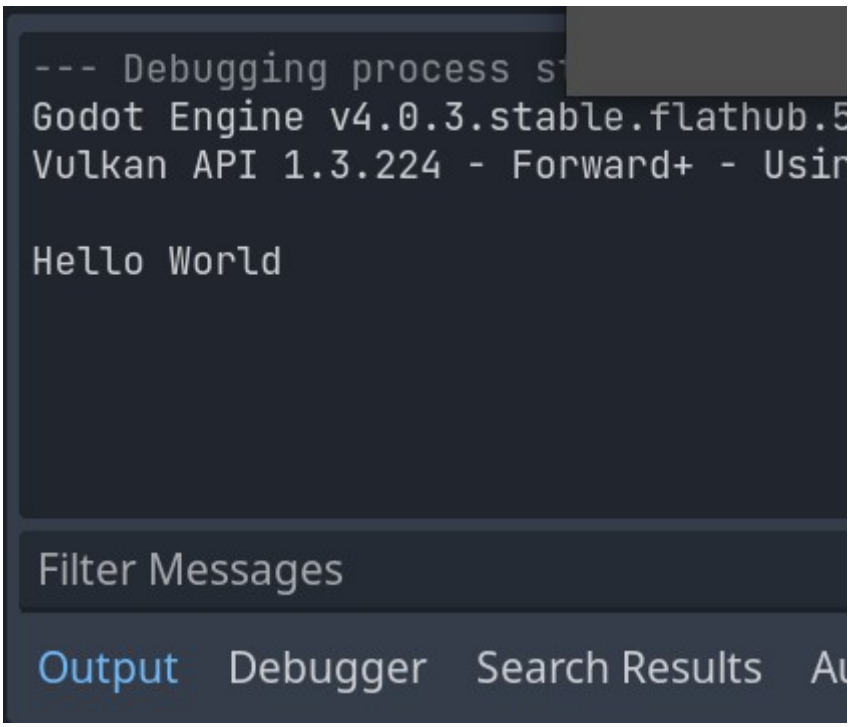
```
 9  # Called when the node enters the scene tree for the first time.
10  func _ready():
11      print("Hello world")
12
```

If I remove the tab before the print, godot will give me an error:

```
 9  # Called when the node enters the scene tree for the first time.
10  func _ready():
11  print("Hello world")
12
13
error(11,1): Indented block expected after declaration of "_ready" function.
```

Godot should automatically indent the text when pressing enter after a : but if anything is wrong the editor will let you know.

Now when we press F5 and run the project we should get a "Hello World" message in the output:



Print will let us send information to the console which is not that helpful for the player but helpful during development as it will let us test and debug different things.

We can also “comment” out code using #, this means the computer will skip this line of code, like so:

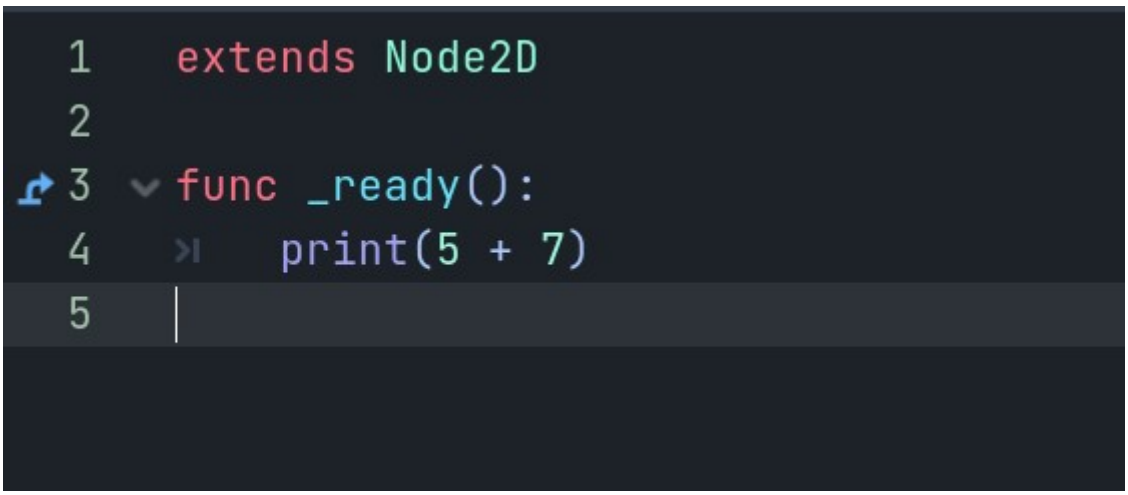
```
1  extends Node2D
2
3  func _ready():
4      print("Hello World")
5      #print("Hello world again!")
6
```

If we run the project now, only the first line will be printed, but if we remove the # both commands will be run. You can give it a try. Comments are useful for commenting code so you can keep notes on what’s happening in the code or explain it to other people.

Next lets take a look at basic math operations.

In GDScript you can do a number of mathematical operations, but we will start simple and use more advanced cases as needed. I’ll keep working in the _ready function for these.

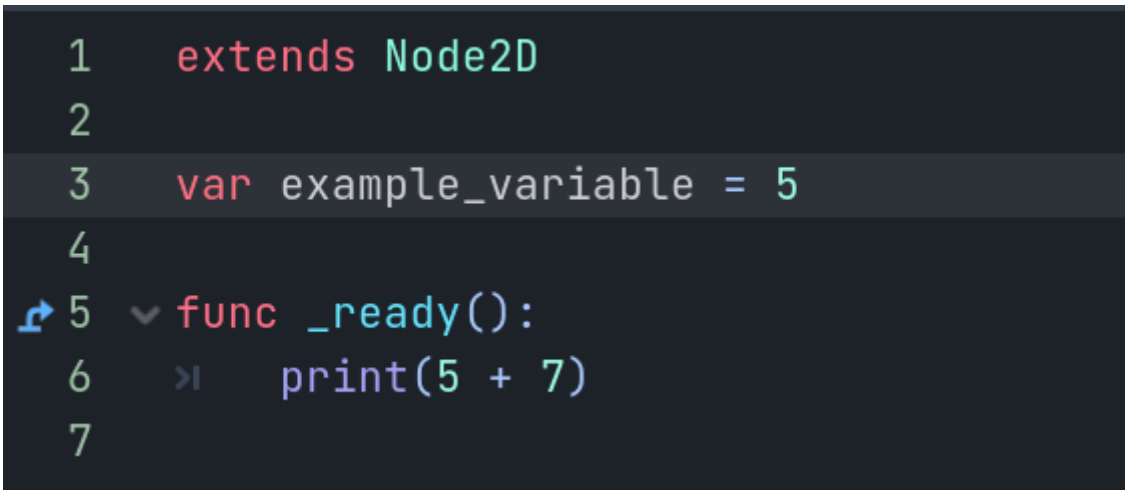
First up, addition. This is simply adding two things to each other, and we can print the result directly like so:



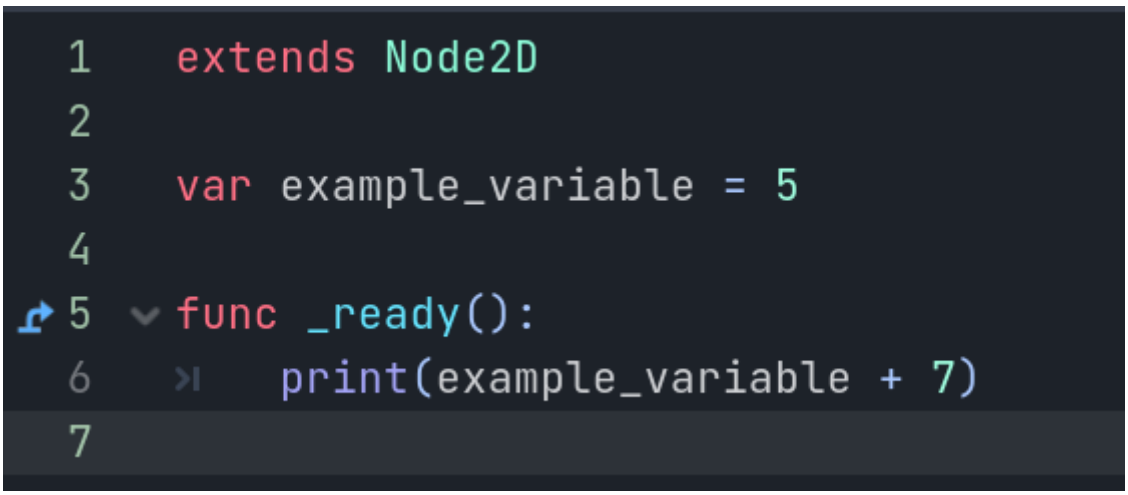
This will print 12.

Subtraction, multiplication and division works the same way, just use -, * and / respectively instead of the + and you will see the results when running the project.

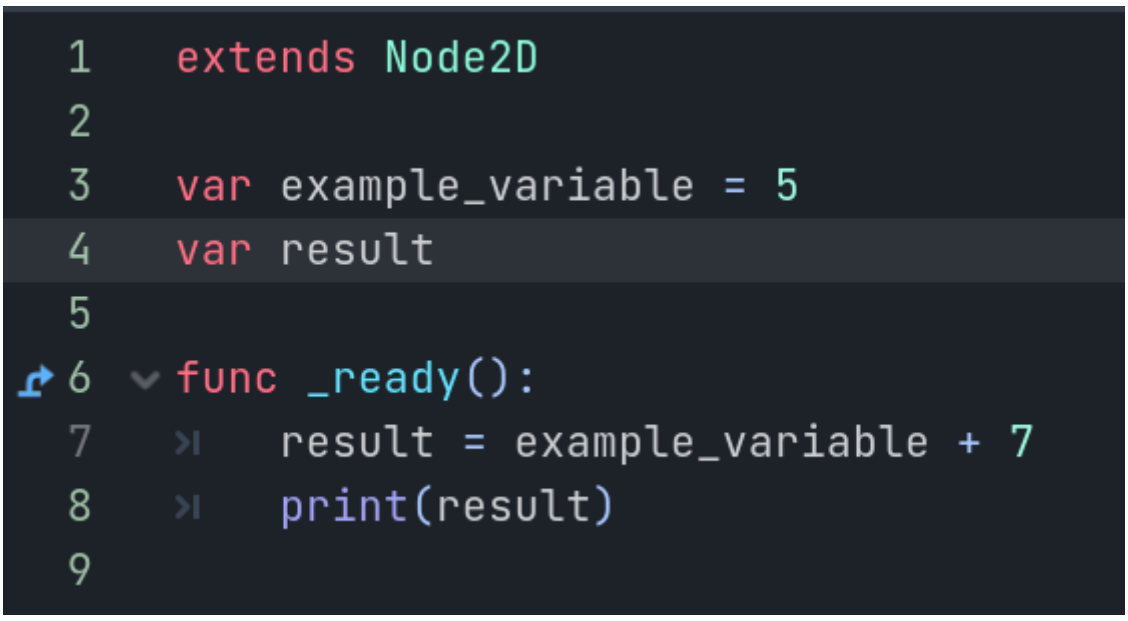
We can make this a bit more useful by using variables to store things. You can think of variables like a box that holds a certain thing or multiple things. To make a variable we first need to declare it by writing “var variable_name”. Variables can be of different types but we will go through them as we need them. For now I’m going to make a variable called “example_variable” and give it a value like so:



Now we can substitute the 5 in the print function with example_variable and get the same results as before:



This will work exactly as before. We could also make a new variable and store the result of the calculation in there, so I'll create another variable called "result", add the numbers to that and then print it like this:



Here I'm creating a variable without any value, then in the _ready function on the 7th line I'm saying result is equal to example_variable + 7 and then printing result.

This would work the same for any of the other math operations.

We can also add variables together like this:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       result = example_variable + example_variable_2
9       print(result)
10
```

Next lets move on to functions. We can create custom functions and then call on them as needed to avoid repeating written code if we find ourselves doing the same thing over and over again. To create a function it needs to be declared like the func _ready(): we already have in the script. I will create a new function called add_two like this:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       result = example_variable + example_variable_2
9       print(result)
10
11  func add_two():
12      pass
```

I'm adding in a "pass" statement here because otherwise Godot will complain about the function. Pass just tells Godot to not do anything at a certain point and can be used to return from functions.

Now lets make this function actually do something. I'll start by making it print "hello world" again and then call on the function in the _ready function. A function needs to be called to do anything, otherwise it will sit unused:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       result = example_variable + example_variable_2
9       print(result)
10      add_two()
11
12  func add_two():
13      print("Hello World")
14
```

The function is called by just writing the function name in _ready and then the ().

Now we can simply move the “result = example_variable + example_variable_2” and “print(result)” lines down to the add_two function keeping in mind the line needs to be indented by using tab:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       add_two()
9
10  func add_two():
11      result = example_variable + example_variable_2
12      print(result)
13
```

Now our _ready function is cleaner but the code will still print 12.

Now we have some very basics on variables and functions, but there is something that needs to be mentioned before we continue: Variables have a “scope” that is dependent on where they are

declared. So a variable declared outside a function like we have done so far is usable anywhere within the script, but if a variable is declared within a function it will be limited to that function. So if I create a variable called “test_variable” in the _ready function, that variable will not be usable in the “add_two” function:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       var test_variable = 9
9       add_two()
10
11  func add_two():
12      result = example_variable + example_variable_2 + test_variable
13      print(result)
14
```

In this case we get the following error:

```
Error at (12, 54): Identifier "test_variable" not declared in the current scope.
```

We can fix this by declaring the variable outside the functions or by declaring it within the function we are trying to use it in. Alternatively we could be using arguments.

Functions can take arguments if we set them up for it, and some functions require arguments to do anything. The print function we’ve been using is a good example for this. Calling on a function with parameters is done by writing the name of the function and in parenthesis the parameters separated by a comma. Declaring a function with parameters works the same way. We can rewrite the code like this:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       var test_variable = 9
9       add_two(test_variable)
10
11  func add_two(test_variable):
12      result = example_variable + example_variable_2 + test_variable
13      print(result)
14
```

We can also rename the variable in either the _ready function or the add_two function and the code will work the same, for example:

```
1   extends Node2D
2
3   var example_variable = 5
4   var example_variable_2 = 7
5   var result
6
7   func _ready():
8       var variable = 9
9       add_two(variable)
10
11  func add_two(test_variable):
12      result = example_variable + example_variable_2 + test_variable
13      print(result)
14
```

So now we have the basics of functions done. Functions are super useful and we will be seeing lots of them later on.

Let's take a quick look at how to actually affect our game with code through moving an object. First off let's delete all the extra code we have, so I'm going to leave only the first line in there:

```
1   extends Node2D
2
```

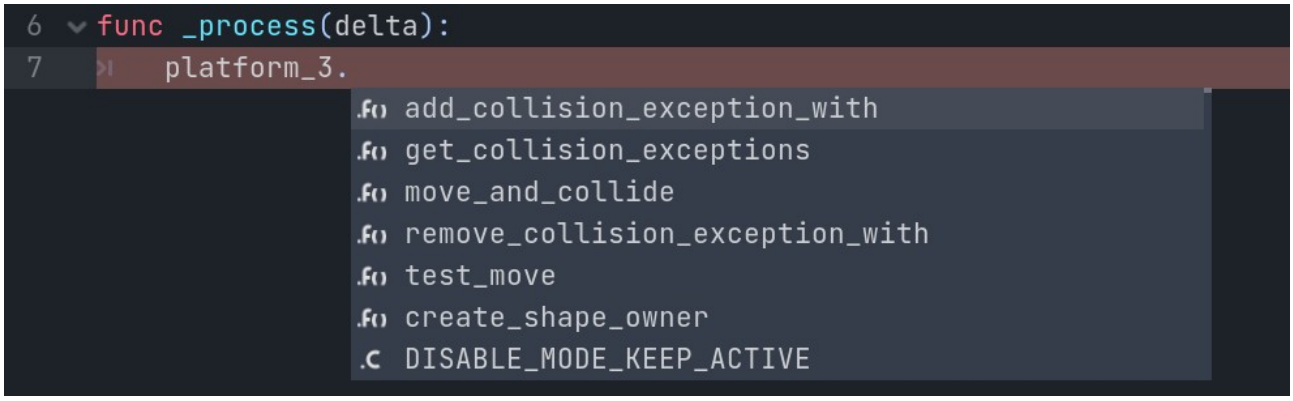
Now we're going to call on a built in function called _process(delta):

This function will run once every time the computer draws a frame. The delta parameter is a bit more complex but we will make use of it later.

Next we need a reference to what we want to move, I'm going to choose Platform3 for this. To get this reference we can drag the platform from the scene tree into the code editor while holding control. We want it outside the _process function like so:

```
1  extends Node2D
2
3  @onready var platform_3 = $Platform3
4
5
6  func _process(delta):
7      pass
```

Now we have a reference to Platform3 in the variable platform_3. This means we can access the parameters of Platform3. By simply writing platform_3. In the _process function it will give a list of things we can interact with:



To move Platform3 we want to increase its position in the x axis, to do this we can write a line of code like this:

```
6  func _process(delta):
7      platform_3.global_position.x += 5
8
```

This will access the global_position in the x axis and increase it by 5 pixels every time the computer draws a frame. This means the faster the computer can draw frames the faster the platform will move. Try it out by running the game! You should see a platform move to the right pretty quickly.

A better way to do this is by multiplying the 5 with the parameter delta. This will make it so that no matter how fast or slow the computer is, the speed of the movement is always the same. That's because delta is the difference between the current frame and the previous one, so the faster the computer the lower the number will be. To use delta simply multiply the 5 with it like so:

```
6  func _process(delta):
7      platform_3.global_position.x += 5 * delta
8
```

Now the platform should move slower!

This has been just a scratch of the surface, in the next document we will actually start to make our game and dive deeper into all the various bits and pieces needed.