In this course, we will be expanding our knowledge of the Godot Game Engine by looking at different aspects of the engine that will help us create our games. We will be doing this by creating four mini-projects, each of which will cover a certain Godot feature or collection of features.

Course Requirements

In order to get the most out of this course, there are some requirements:

- Have the Godot engine installed.
- Have a basic understanding of the Godot engine.
- Have a foundational understanding of the GD Script programming language. You need to know what a variable or a function is, however, we will go finely through each line of code in this course.

Balloon Popper Mini-game

The first game we will be creating is a Balloon Popper mini-game. In this game, spheres will appear on the screen, when you click them they will grow in size and when they reach a given threshold they will pop and increase our score. Using this game we will cover a number of Godot systems that include:

- Detecting clicks on spheres.
- Setting up a scoring script.
- Displaying our score on the screen.

Physics Mini-game

The second thing we will explore is physics in the Godot Engine. We will do this by creating a game where you push a player around the scene and crash through boxes which will react dynamically to the collisions. Using this game we will explore multiple systems, that include:

- Using a rigid body and its properties to create the physics we want for our game.
- A script that pushes the player around the level depending on where you click.

Programming Loops in GD Script.

Loops are an incredibly important part of programming, especially in game development where you may want to repeat one action many times. In our third mini-game, we will use loops to create a randomly generated start system. We will set this up to be very customizable, and also look at other ways we can use loops in our GD Script code.

Skiing Mini-game

In the final mini-game, we will be focussing on collision detection inside of the Godot Engine. We will do this by connecting a signal to our player so that when they hit another collider it will reset the scene. Collision detection is used in nearly every game, so it is definitely an important feature of Godot to cover.

About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the

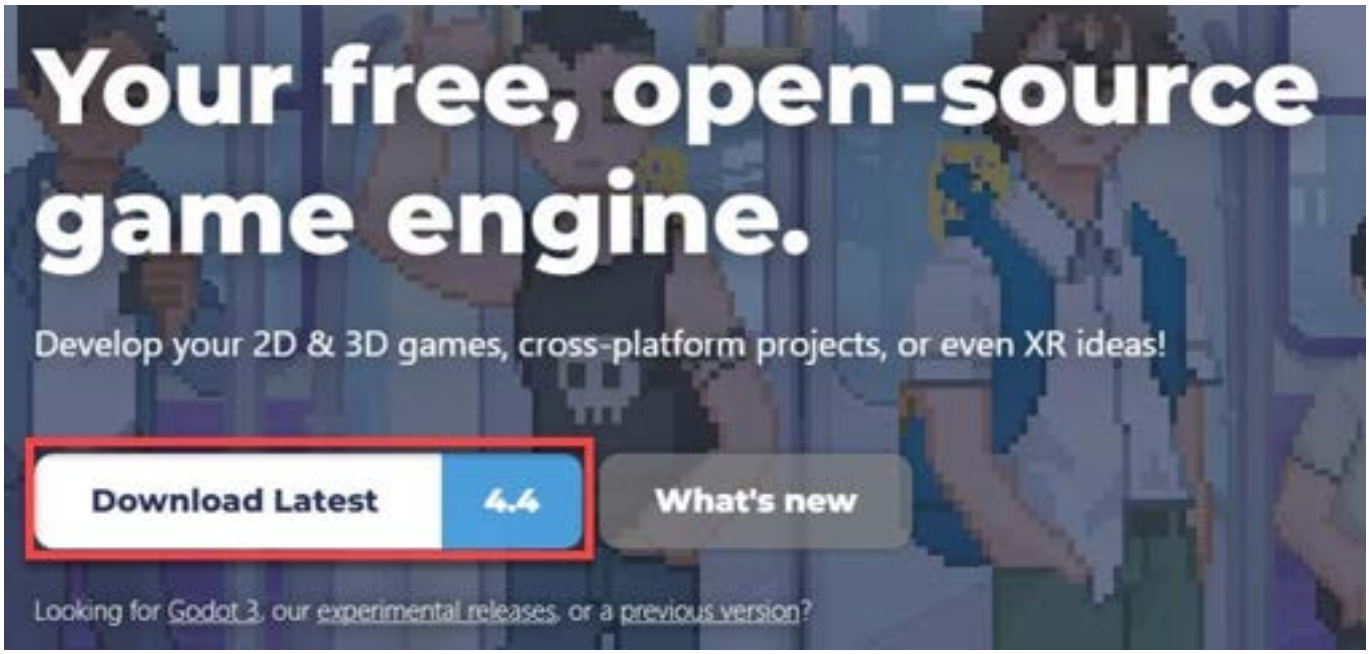
included course files.

Course Updated to Godot 4.4

We've updated the project files to Godot version 4.4 for this course – the latest stable release.

How to Install Version 4.4

You can download the most recent version of Godot by heading to the Godot website (https://godotengine.org/) and clicking the "Download Latest" button on the front page. This will automatically take you to the download page for your operating system.



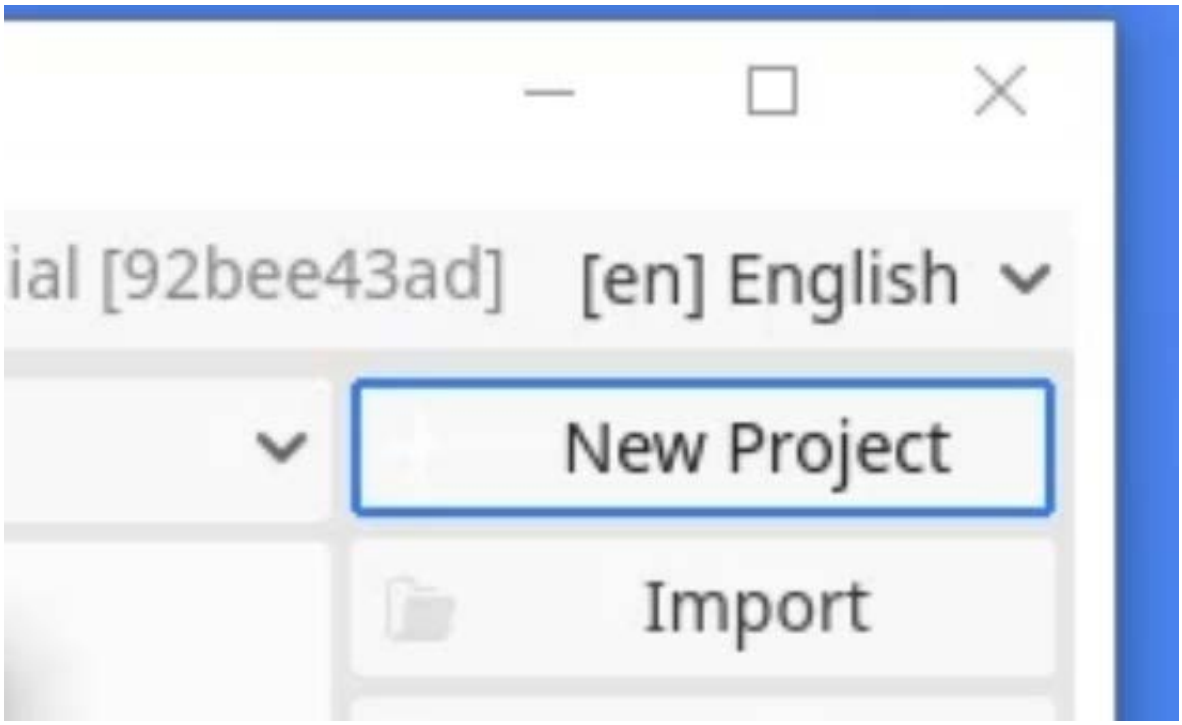
Once on the download page, just click the option for **Godot 4.4.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application launcher – no further installation steps are required!

If Godot fails to automatically select the correct operating system for you, you can scroll down to the "Supported platforms" section on the download page to manually select the download version you would like to install:

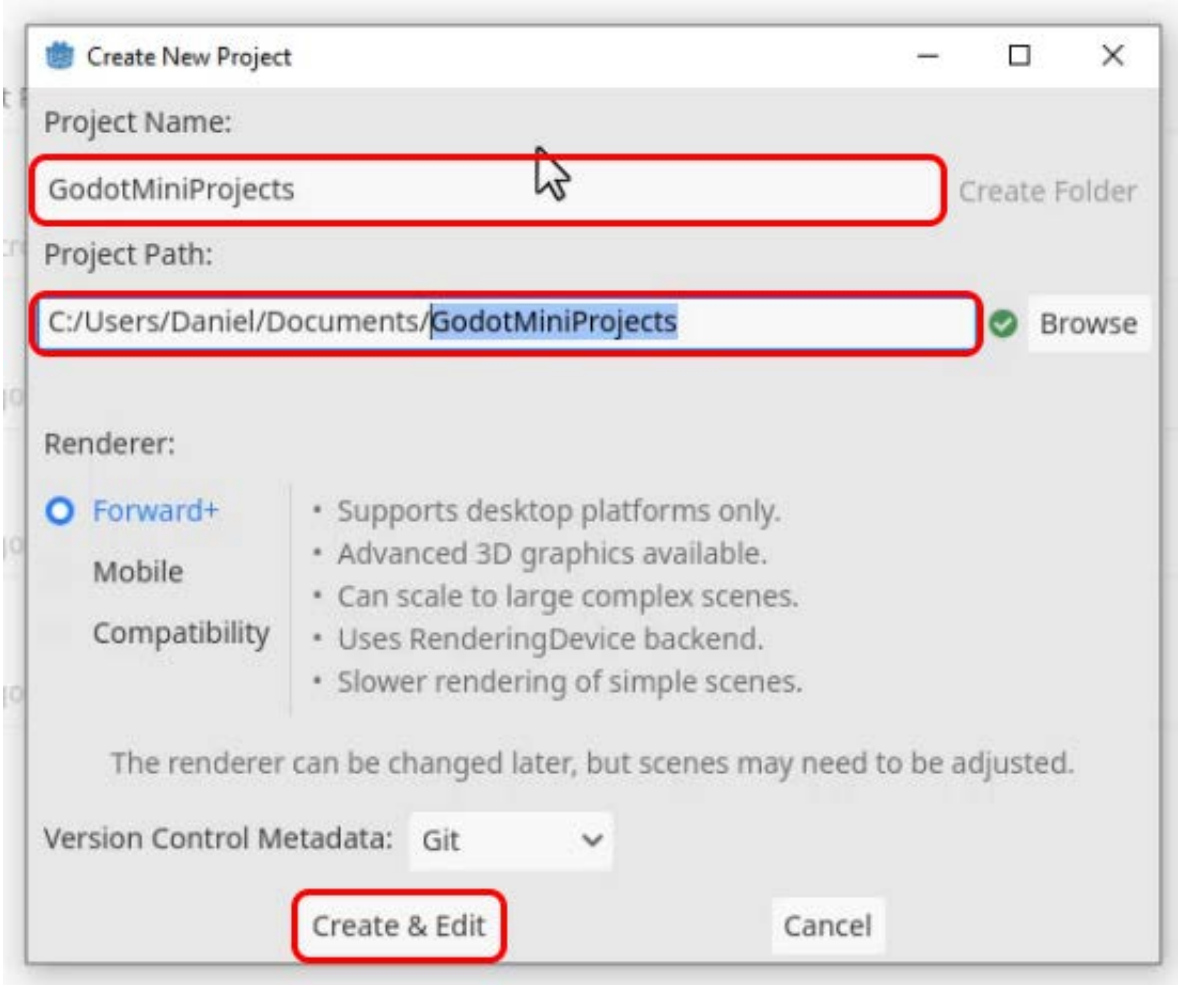
In this lesson, we will begin our mini-projects course by creating the *Balloon Popper* mini-game that we described in the introduction. This game will have a number of different balloons on the screen that can be clicked on to increase their size. Once they reach a certain size, they will pop which will increase our score. The score will then be displayed on the screen using Godot's *UI* system.

Creating the Project

We will begin this mini-game in the *Project Manager* that appears when you launch the *Godot Engine.* For this mini-game, we are going to create a **New Project**.

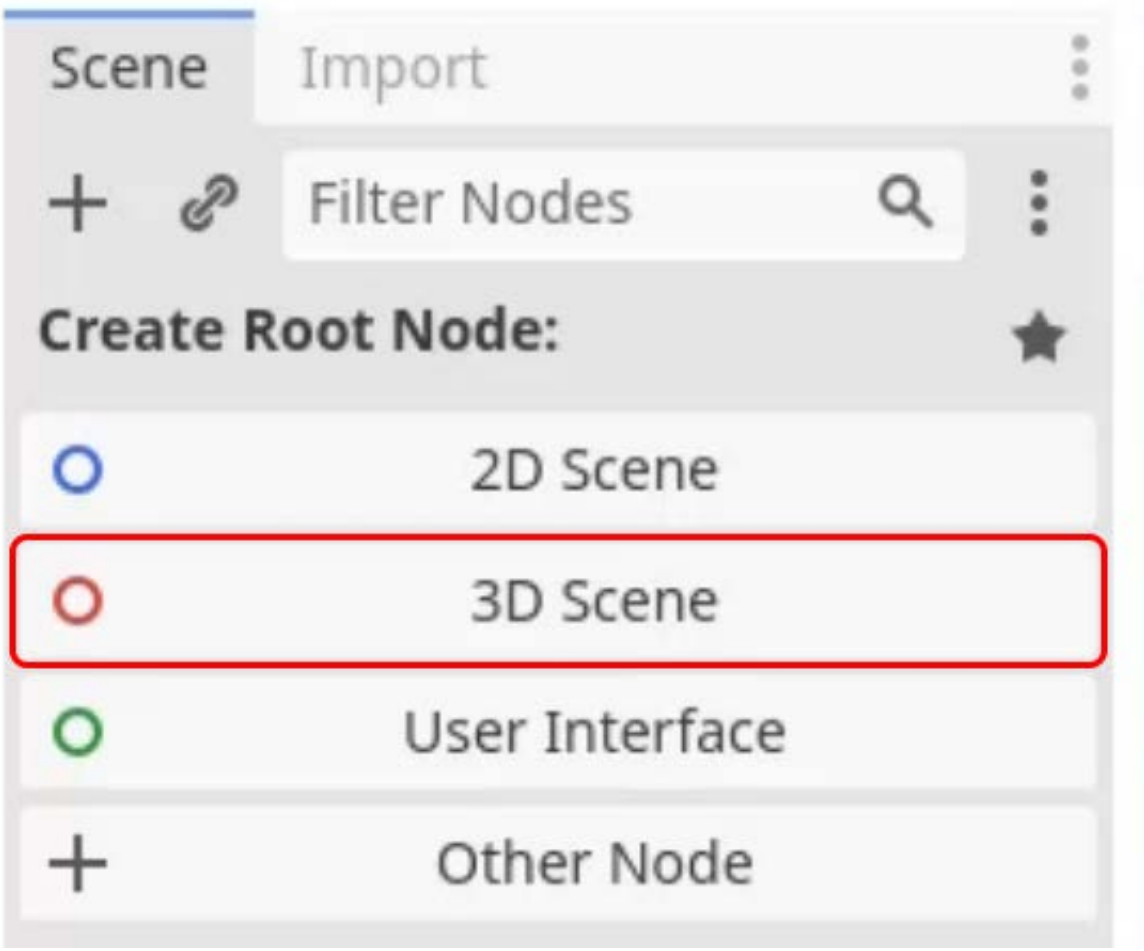


We then want to name this *GodotMiniProjects* and choose the *Project Path*. Remember to press the **Create Folder** button to create a new folder for your project to go into. We can then press the **Create & Edit** button to load the new project in the Godot Editor.

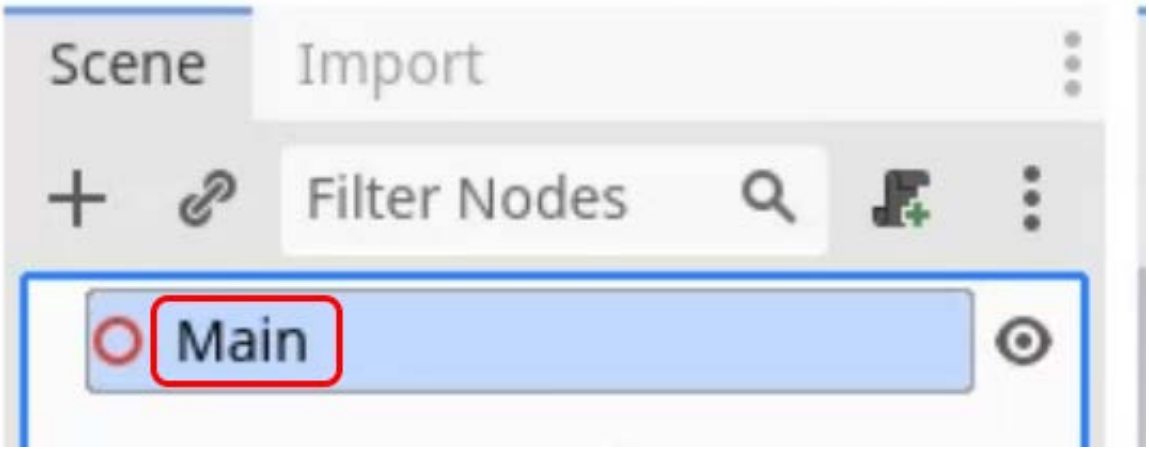


Creating the Root Node

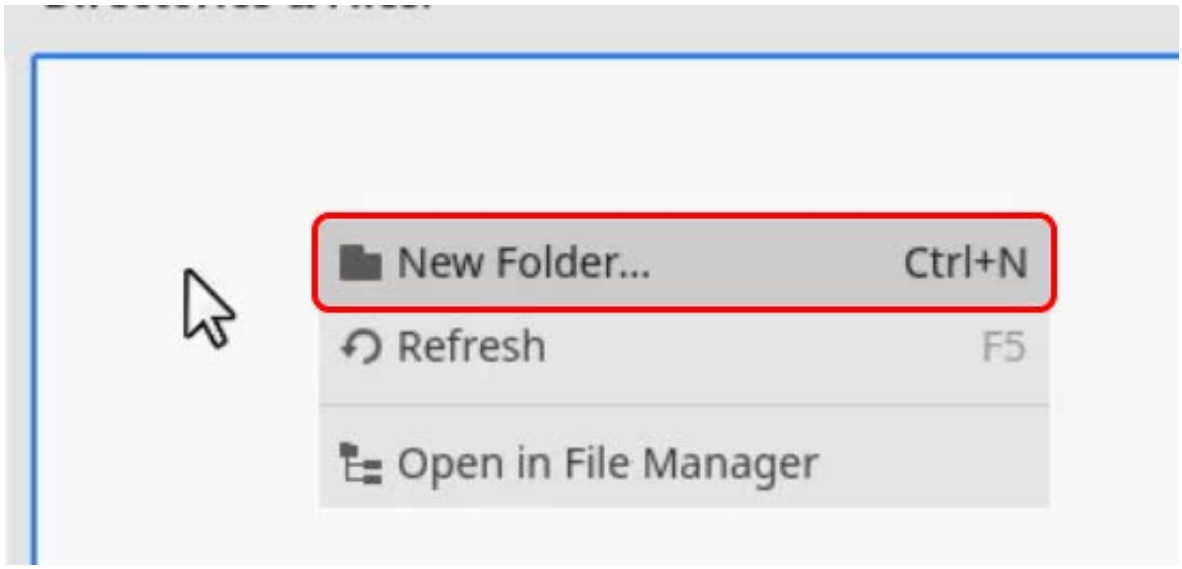
The first step is to create a *root* node. As a reminder, the *root* node is an empty node that acts as a container for all other nodes in the level. For this game, we will be using a 3D view, so we will choose the **3D Scene** option.



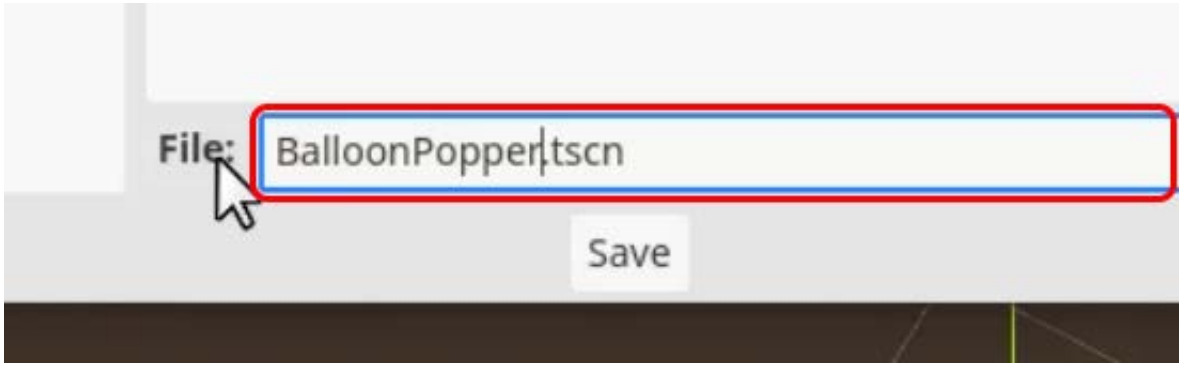
We can then rename this node to *Main* to keep the Scene hierarchy organized.



We will then create a **new folder** called *Balloon Popper* to save our scene in, so that we can easily keep track of which files are for which mini-game.



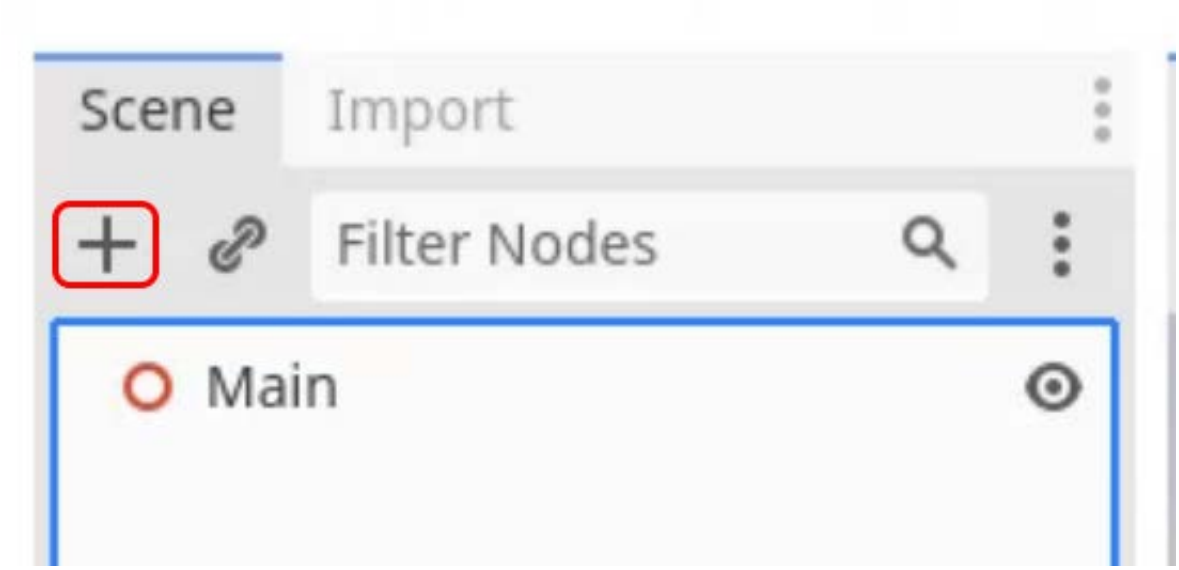
We can then save the scene inside this folder, using the name *BalloonPopper.tscn.*



You will notice in the *FileSystem* window you can now find the scene inside the *Balloon Popper* folder as planned. We will store all our assets for each mini-game in their respective folder to keep the *FileSystem* organized.

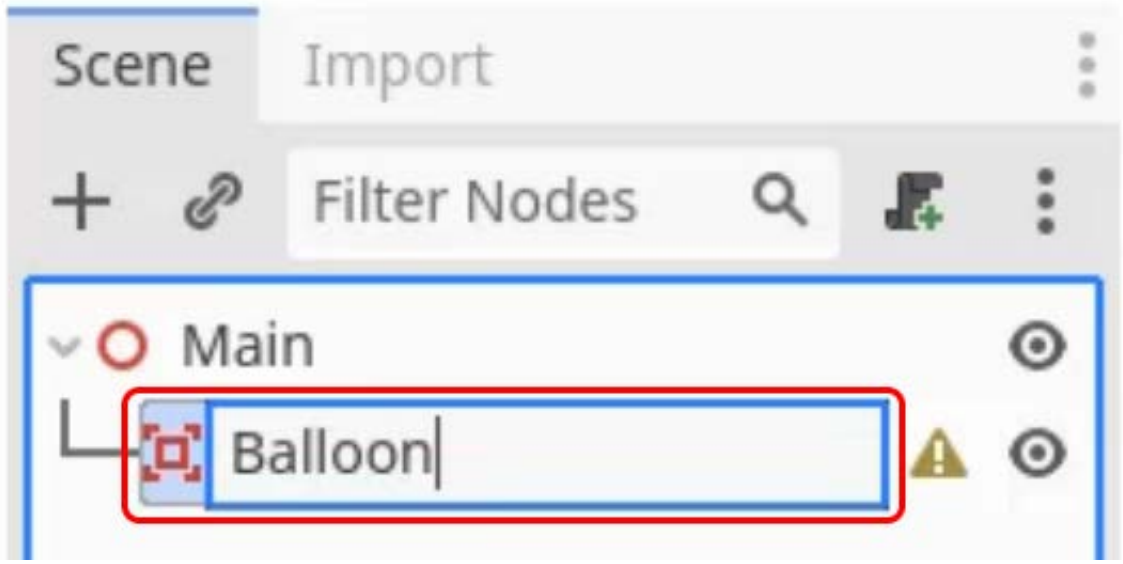
Creating the Balloon

Next, we will create the balloon that will be clicked on. To do this, we will first press the **plus** symbol in the *Scene* window, to open the *Create New Node* window.

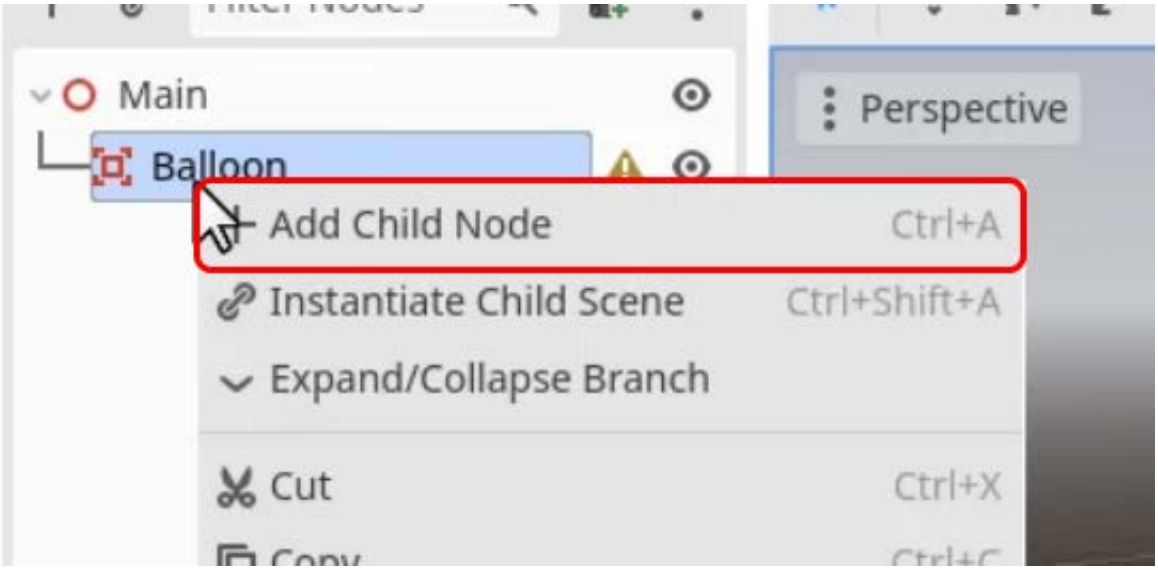


We will then create an **Area3D** node, which will be able to detect collisions and physics interactions. This will allow us to detect when we are clicking on the balloons with our mouse.

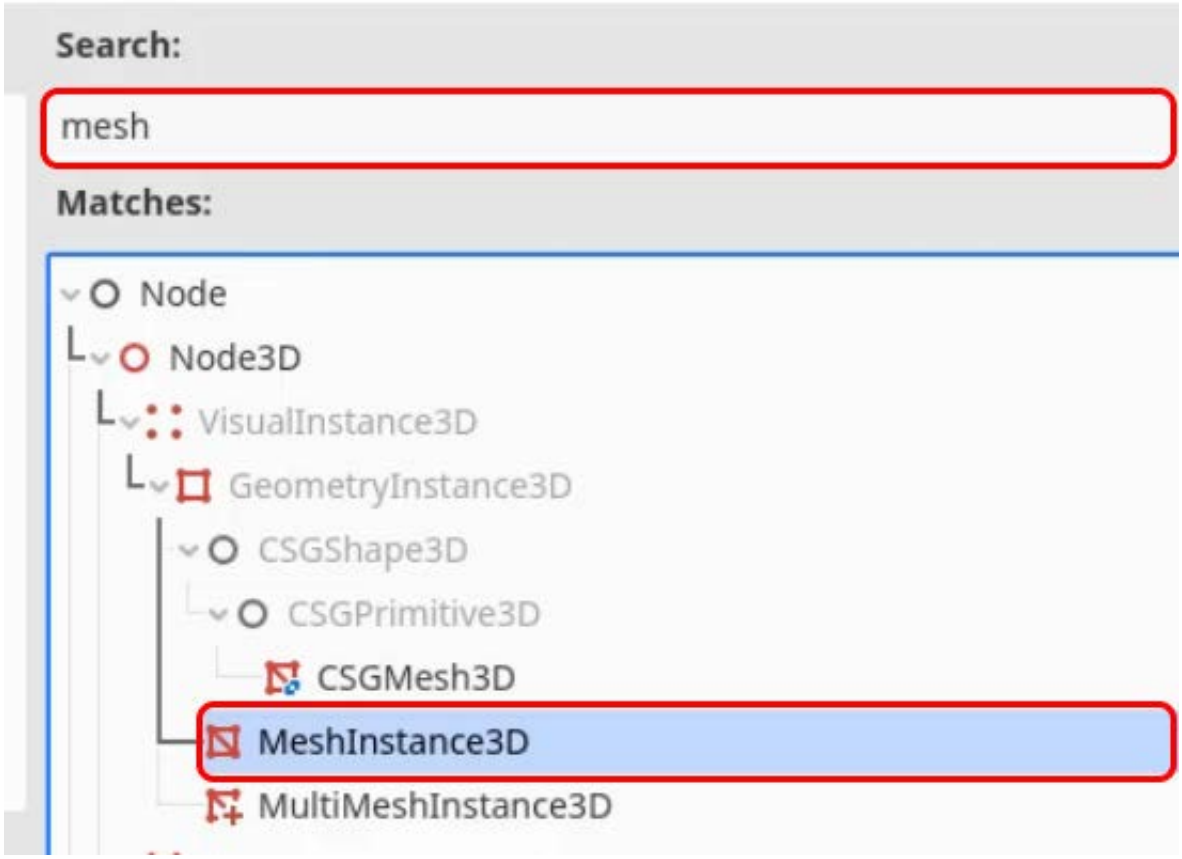
We can then rename this new node to *Balloon.*



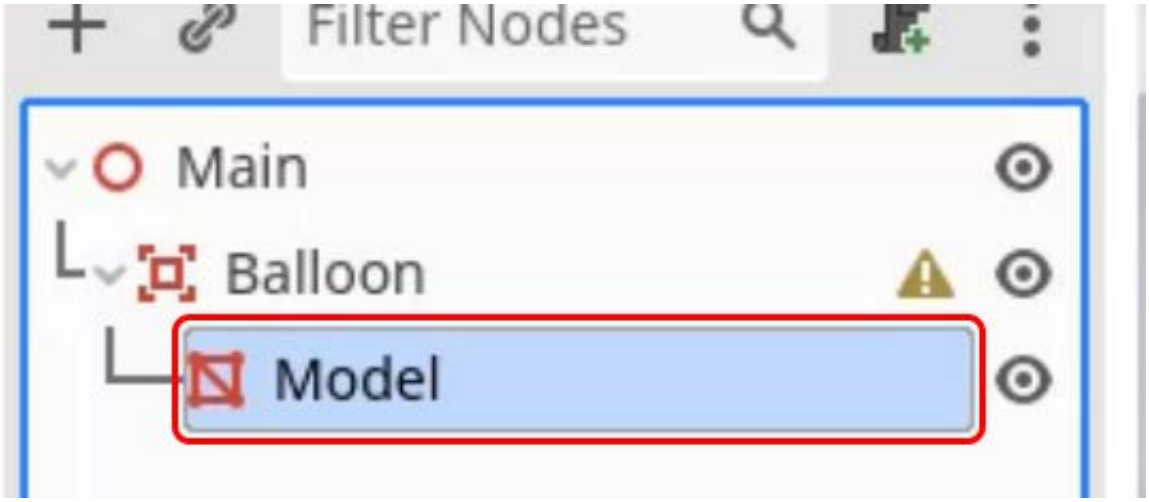
We will then add two child nodes to the *Balloon* Area3D node. To do this, we will *right-click* the *Balloon* node and select **Add Child Node**.



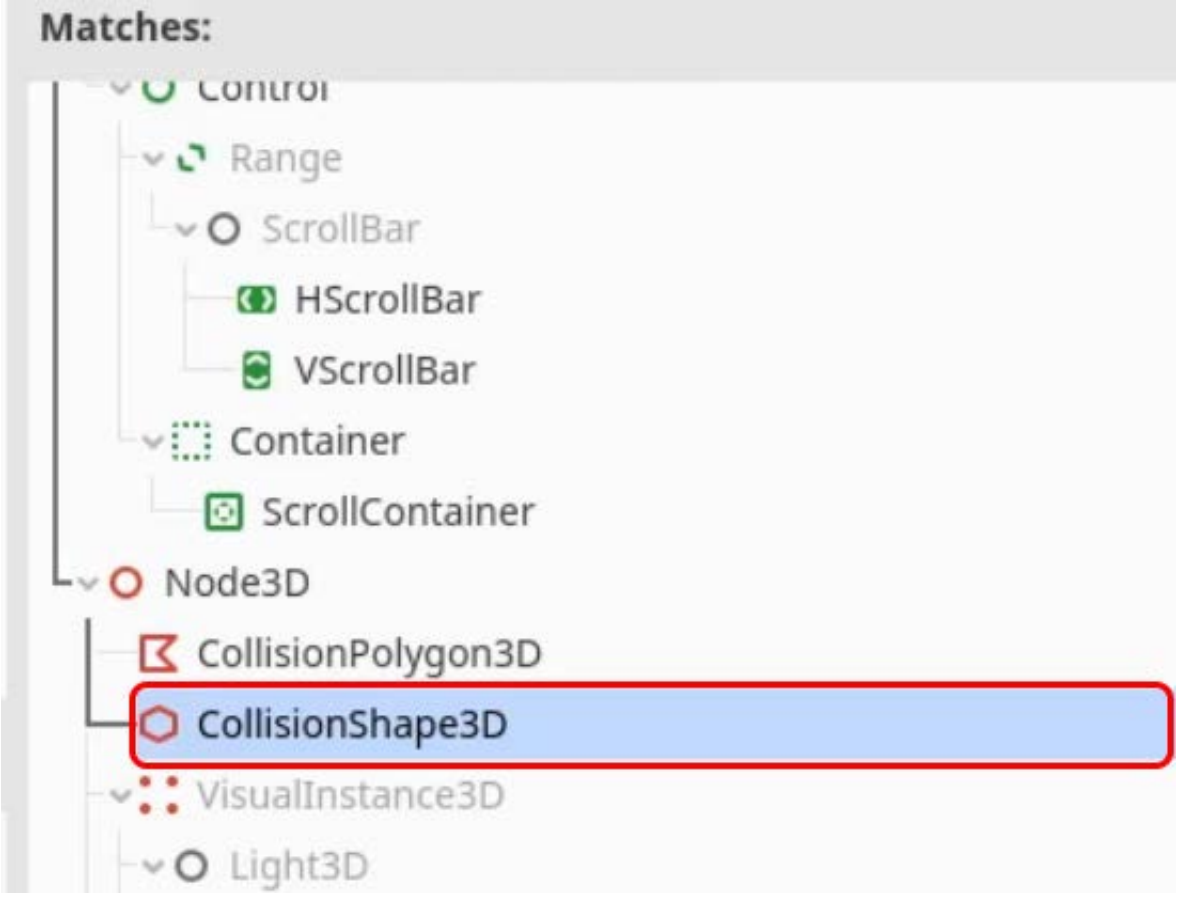
The first node we will add is a **MeshInstance3D** which we will use to see our *Balloon* in the level.



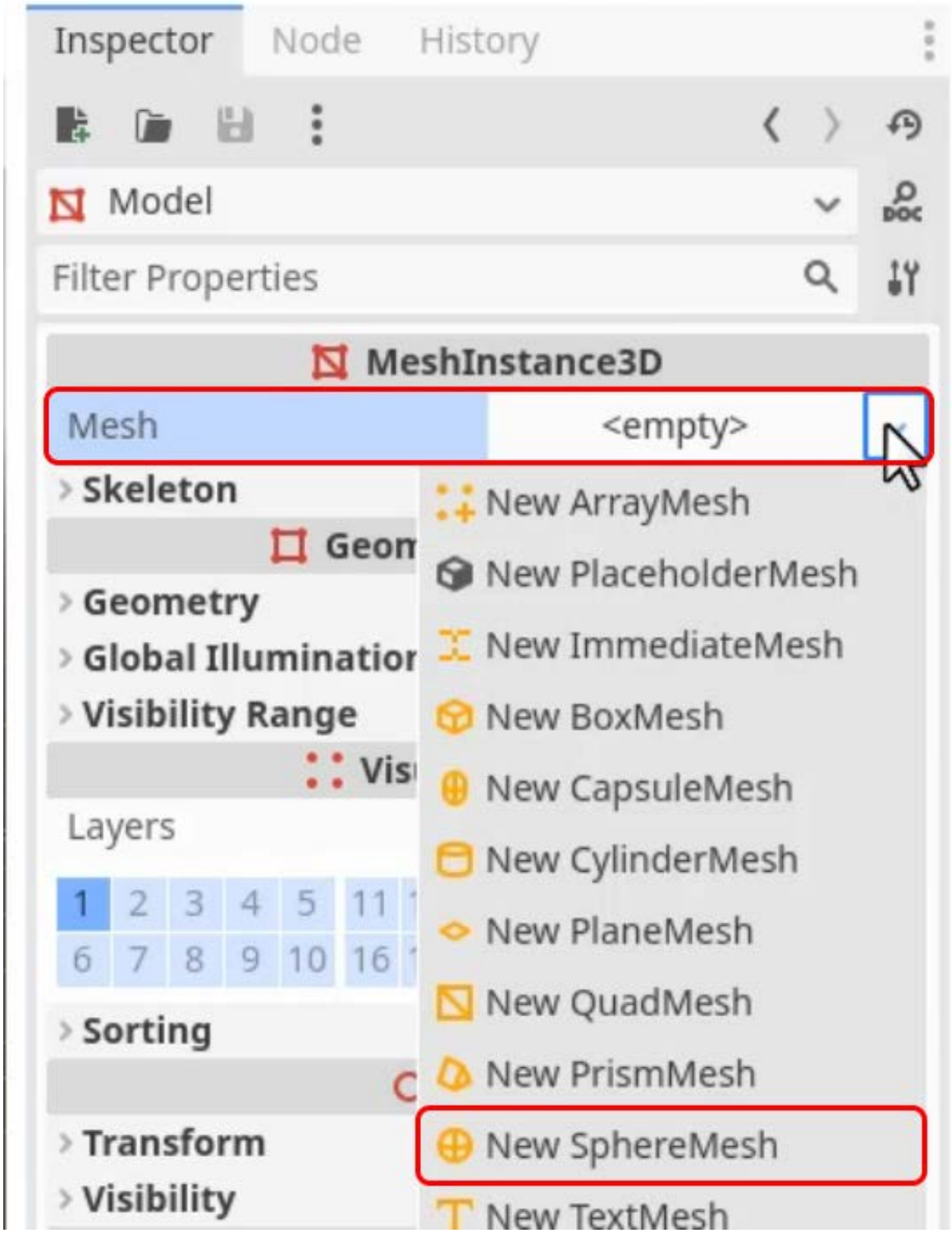
We will then rename this to *Model* so that it has a more recognizable name.



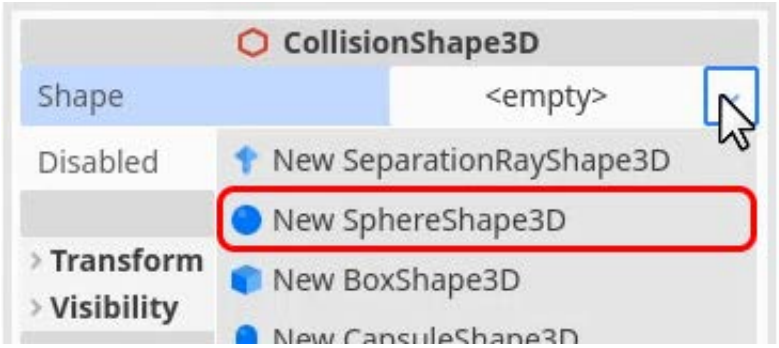
The second node we will add will be a **CollisionShape3D**.



We can then set *Model's* **Mesh** property to a **New SphereMesh**.

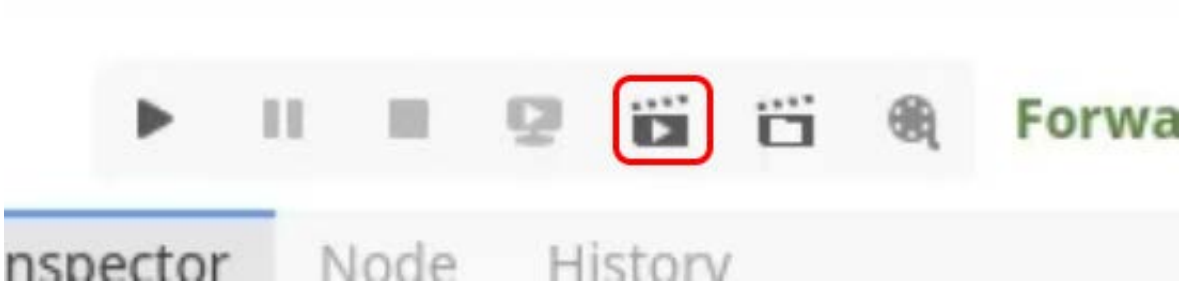


We will also add a **New SphereShape3D** to the **Shape** property of our new *CollisionShape3D* node to match our *Model*.

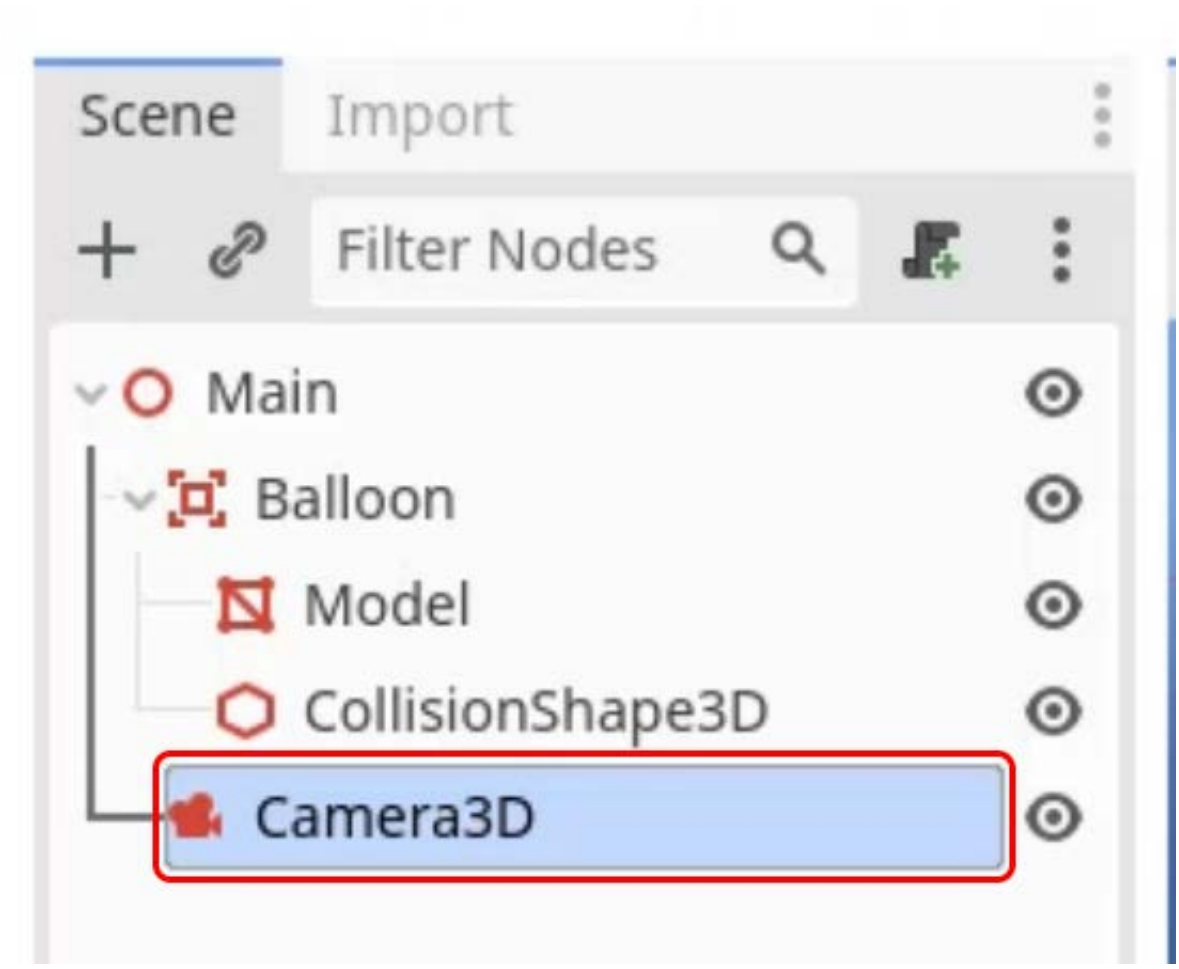


Adding a Camera and Lighting

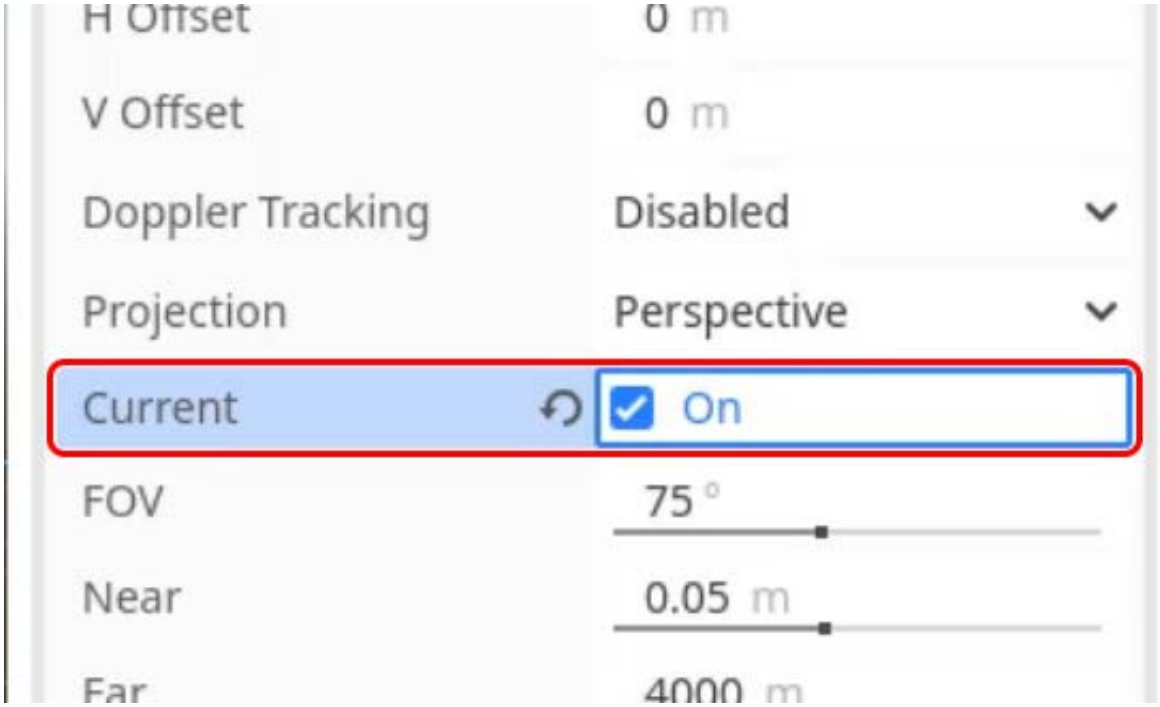
To test the current scene we will use the **Run Current Scene** play button, which is to the right of the normal play button.



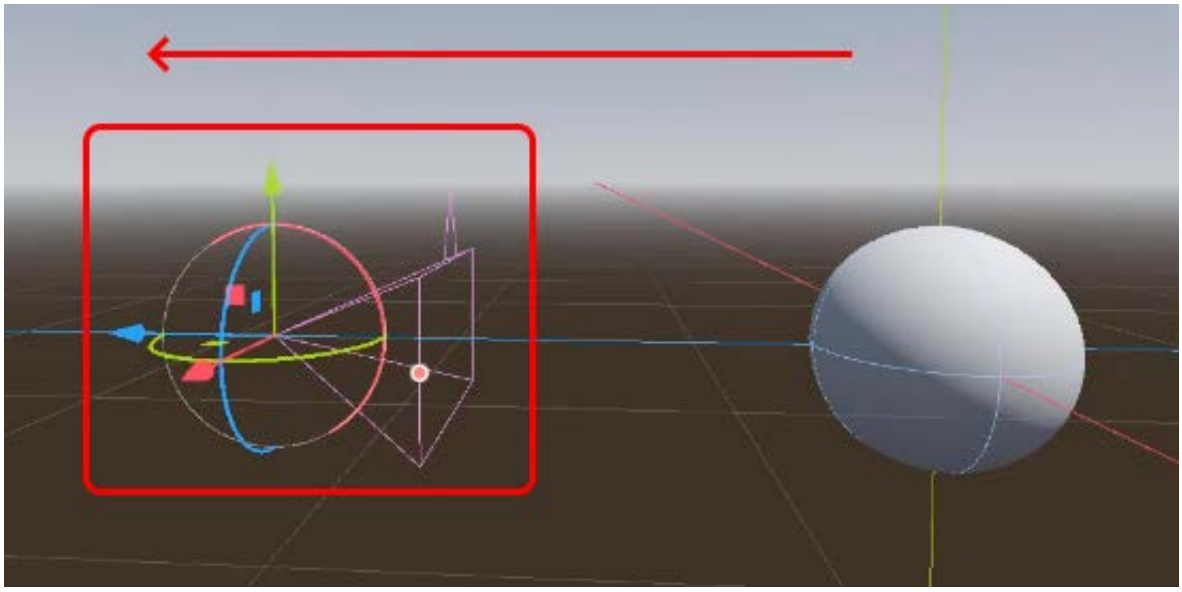
However, you will notice that currently there is nothing to see in our scene. This is because we don't have a *Camera* in our scene, which in 3D is required for Godot to know where and in what direction we want to render our screen. We can therefore fix this, by adding a **Camera3D** into the scene.



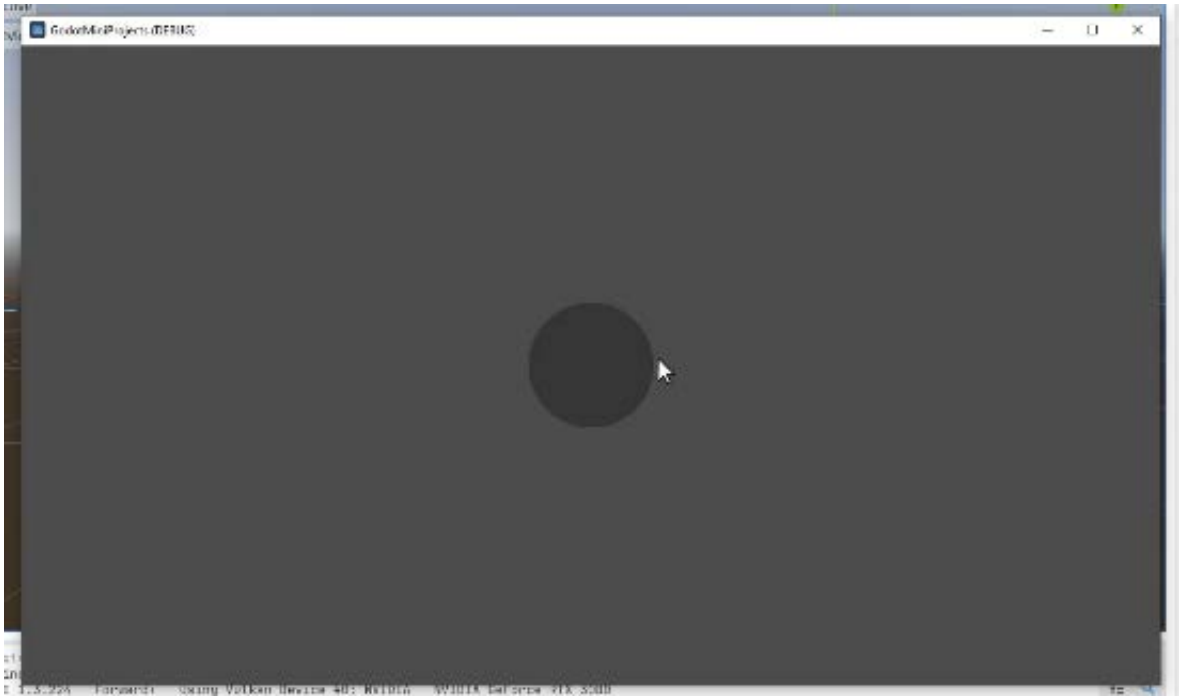
We will then enable the **Current** property of the new *Camera3D* node. This can also be found in the *Inspector* window.



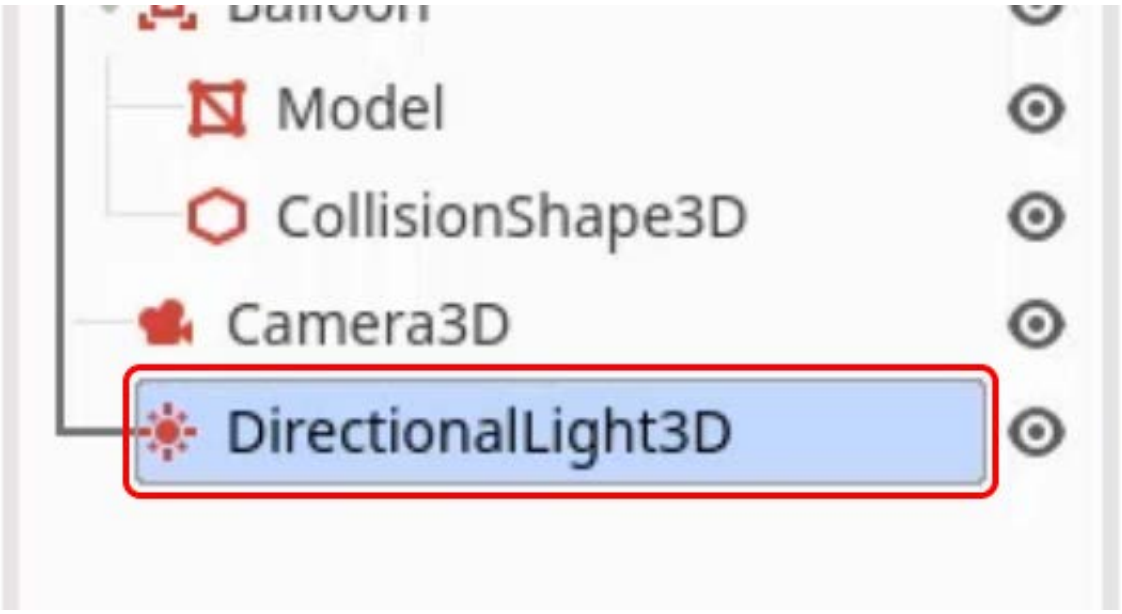
We can then bring the *Camera3D* node backward using the move tool in the viewport so that we can see our sphere in its view.



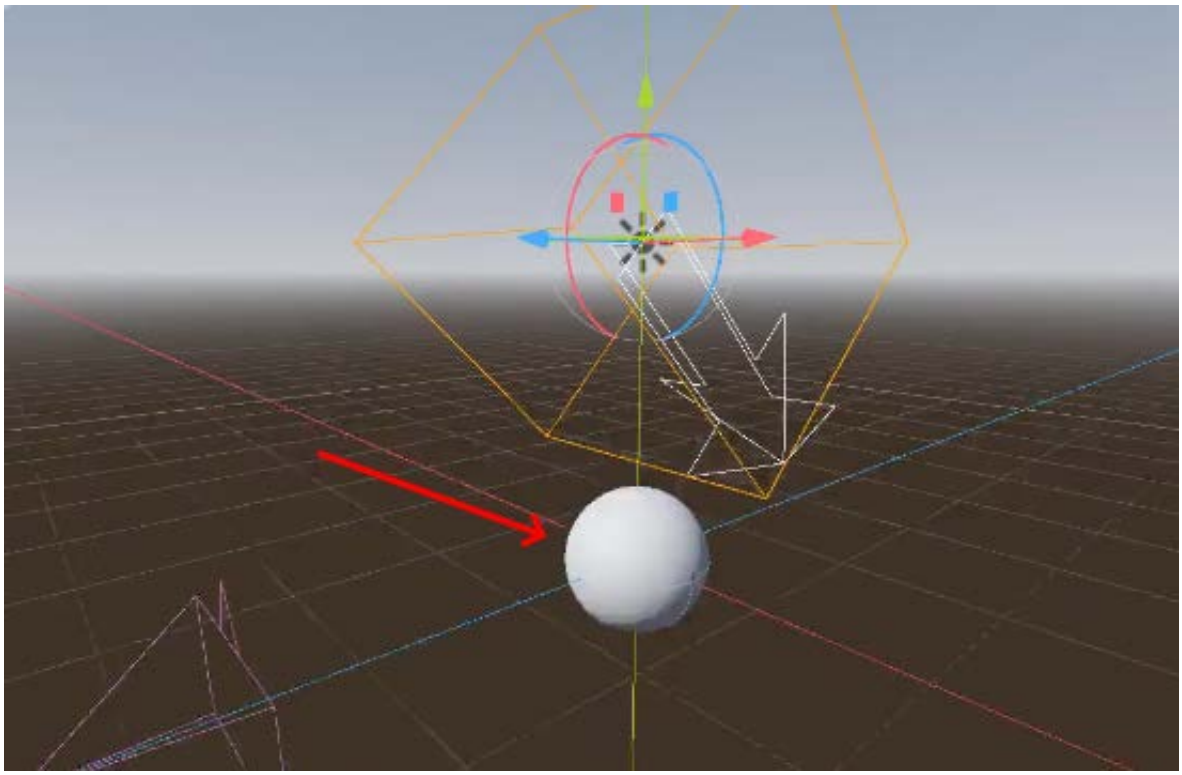
If you now press **Play** you will see our sphere in the game view.



However, you will notice the sphere is really dark compared to the *Editor's* view. This is because the lighting is not correct. We can fix this by adding a **DirectionalLight3D** node into the scene.



We then want to aim the white arrow of the *DirectionalLight3D,* by rotating it, so that the light is hitting the front of our sphere to light it properly.



Now if you press **Play** you will see the sphere is shaded properly.

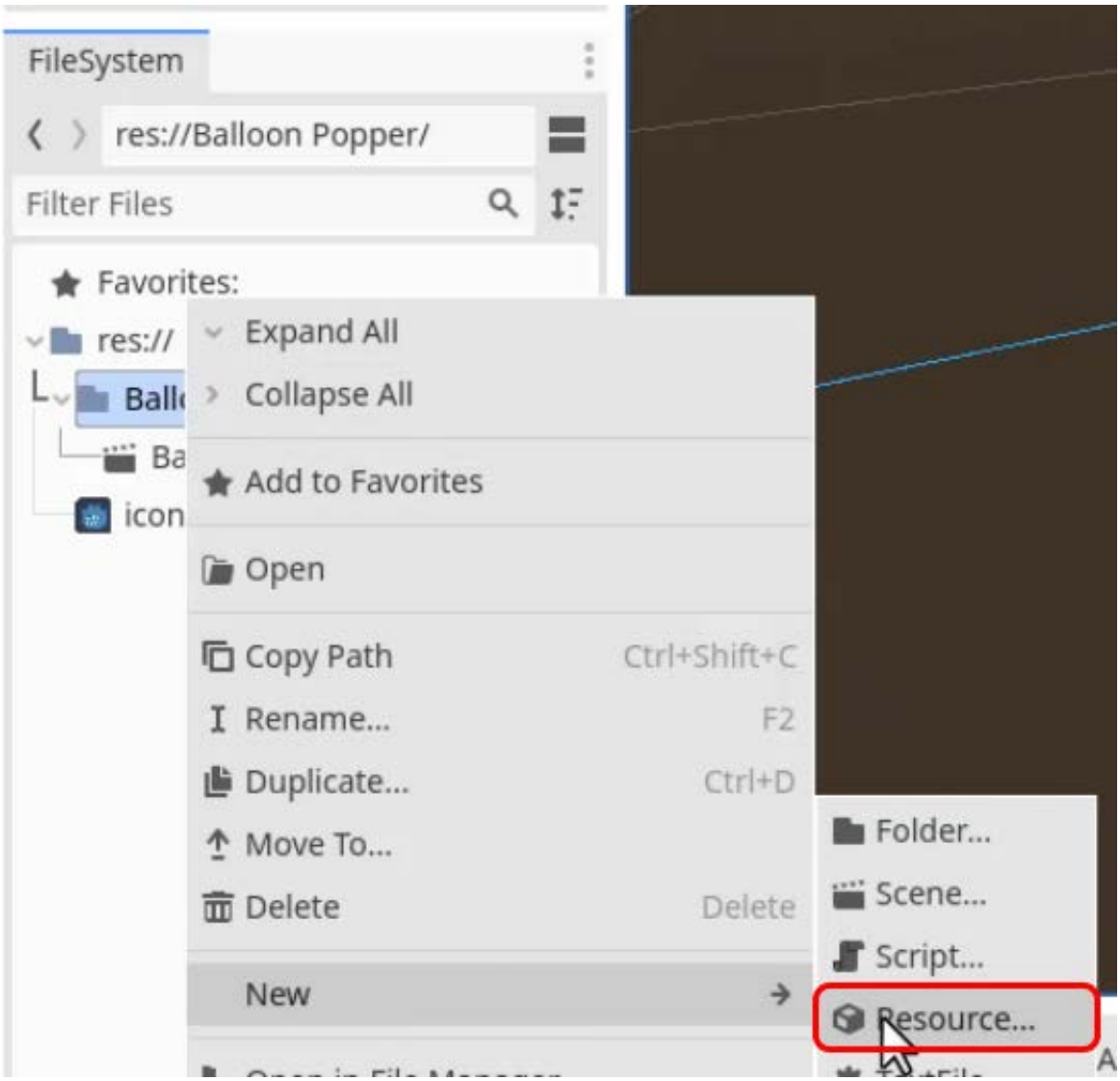
Challenge Tasks

Before the next lesson, you should experiment with the camera and create a material for the sphere. You can change the camera’s field of view and position, and create a material for the sphere that has a nice, bright color and a shiny surface. This will make the balloon look more realistic.

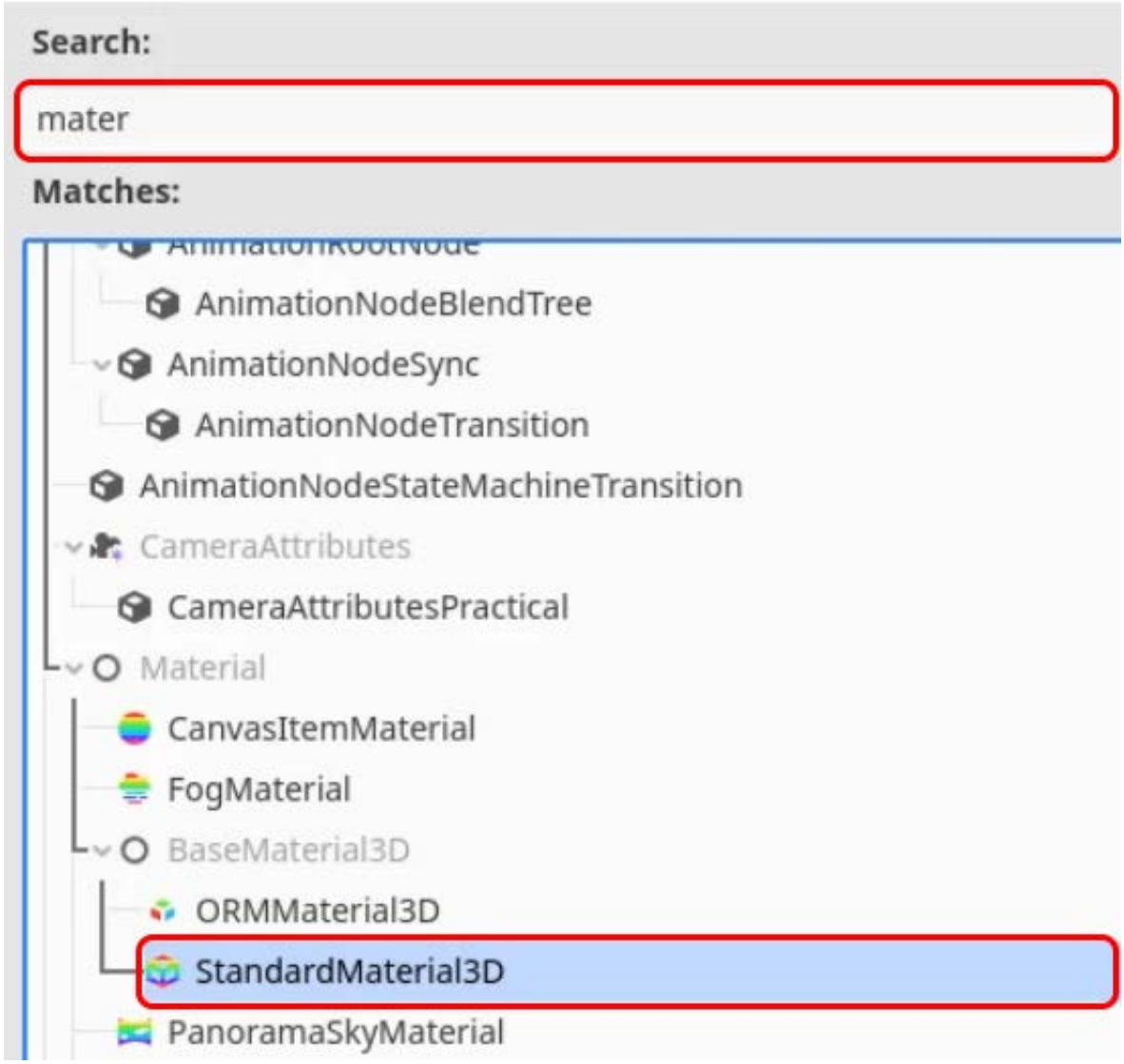
In this lesson, we will set up the systems to detect mouse clicks on the balloon. We also discussed how to increase the size of the balloon and set up a scoring system.

Creating a Material for the Balloon

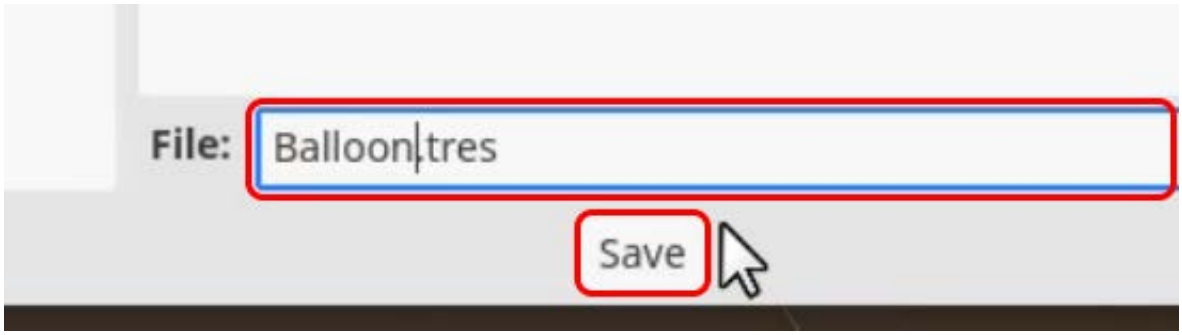
In the previous lesson, we set a challenge to create a material for the *Balloon* node. To do this, we will start by right-clicking on the *Balloon Popper* folder in the *FileSystem*, and choosing **New Resource**.



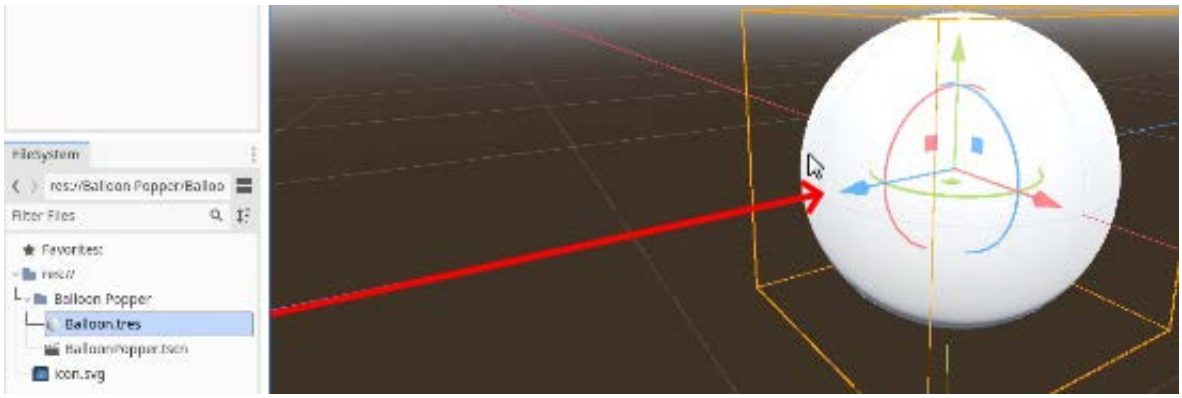
We will then select **StandardMaterial3D** from the list.



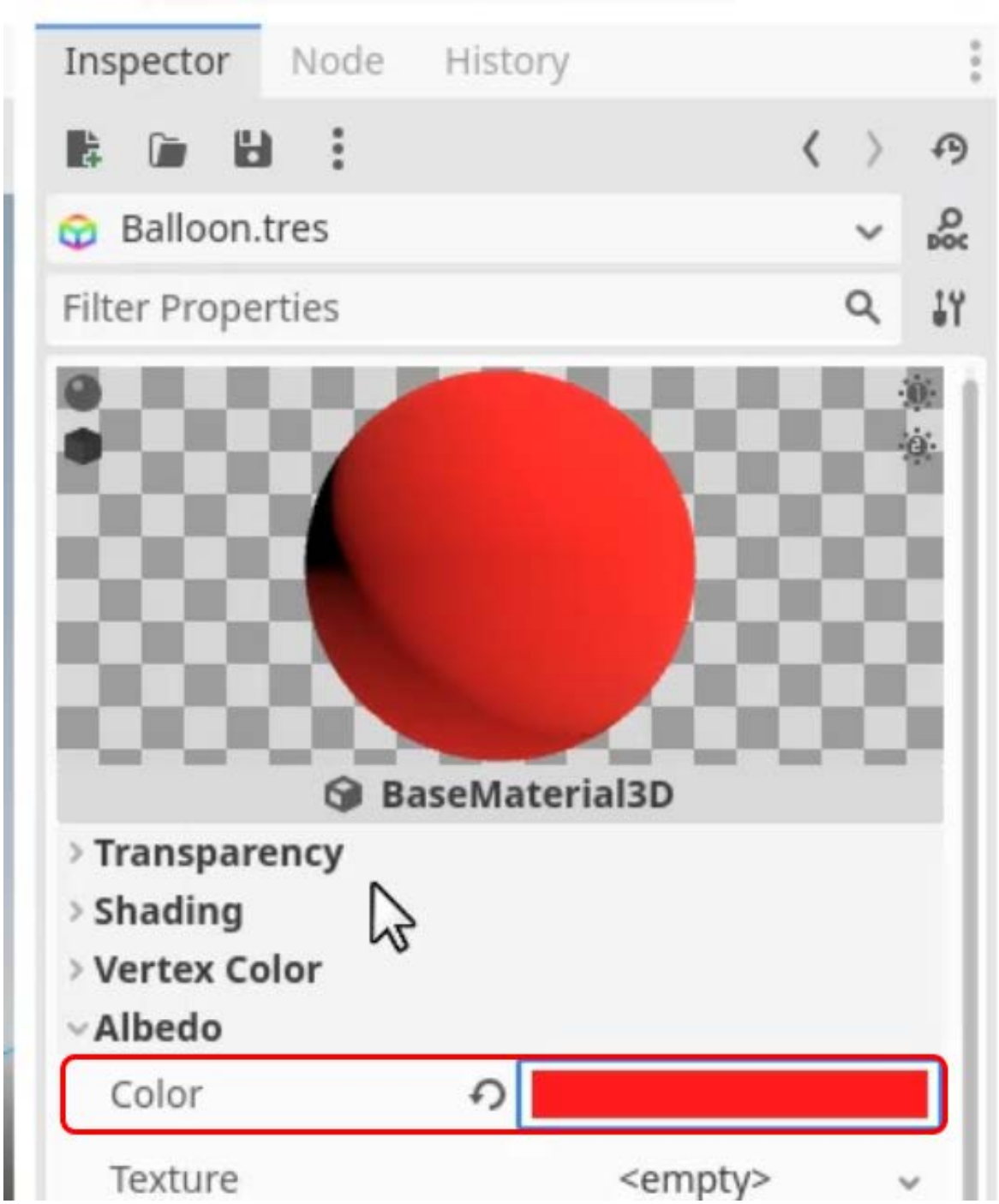
Once created, we can then name it *Balloon.tres* so that we can easily identify it as the material for our *Balloon's Model* node.



We can then click and drag the material onto the *Model* node to apply it.

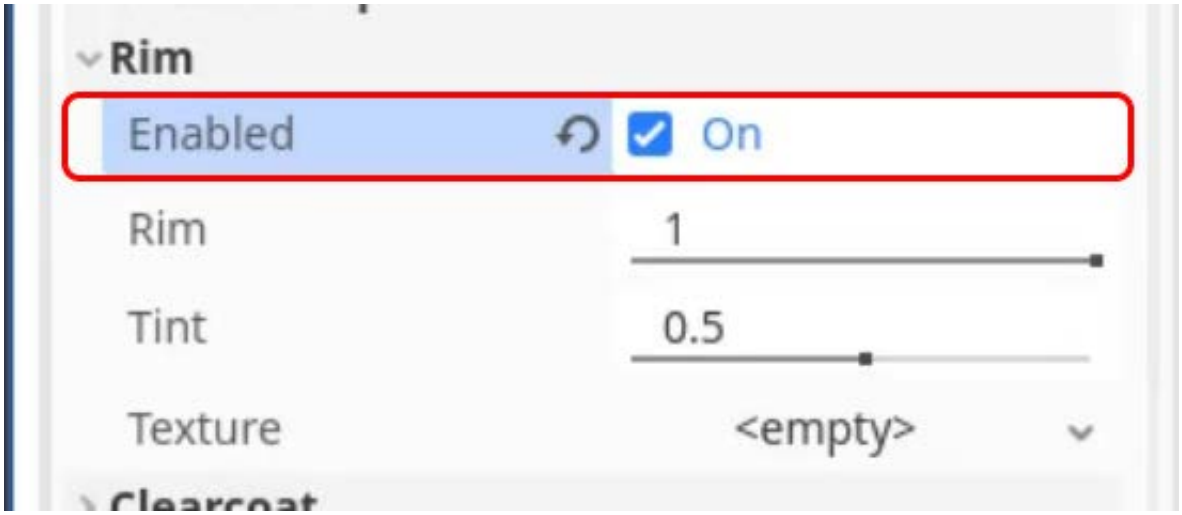


With the material selected, we can then change the **Color** property to a vibrant red in the *Inspector*.

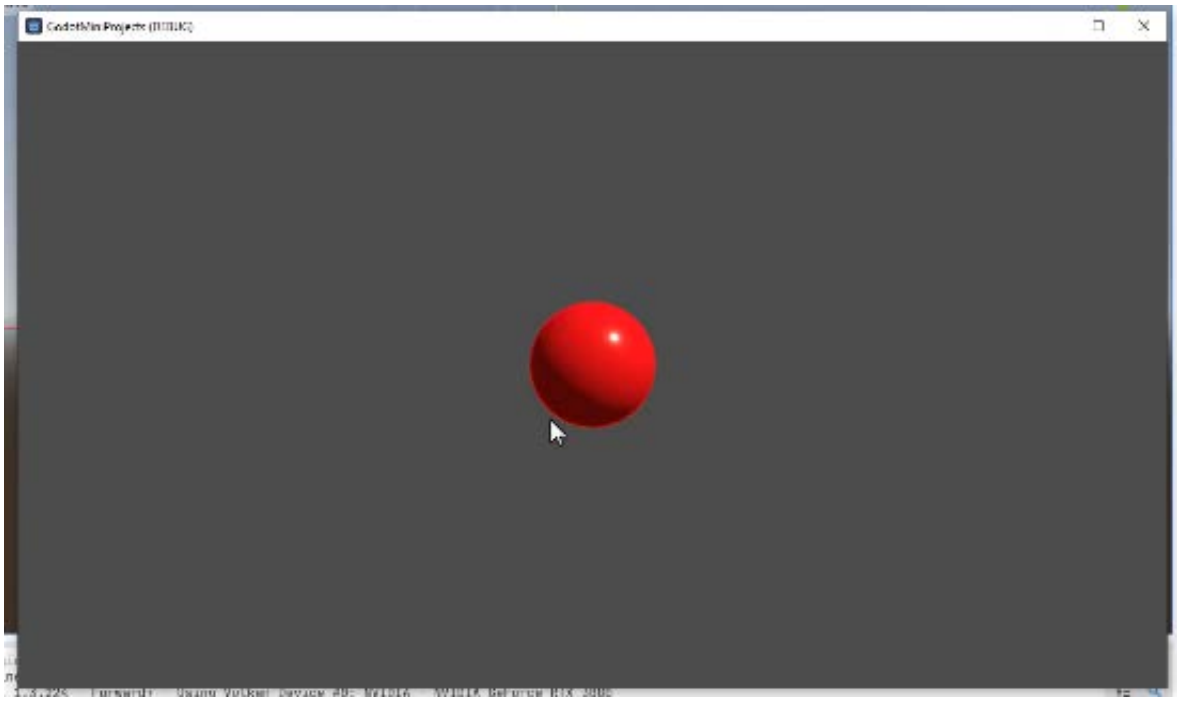


We can then adjust the **Roughness** to create a specular highlight on the material.

Finally, we can enable **Rim** to make the highlights around the edges pop.

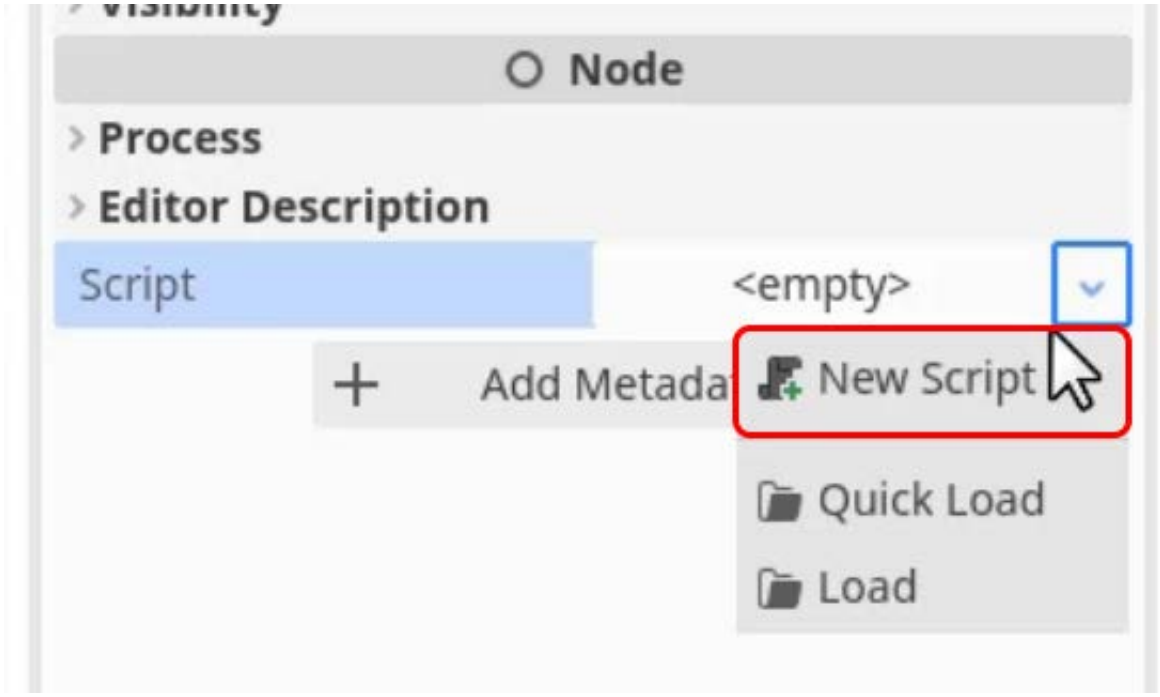


If we then **Play** the level we can see what the material will look like in our game.

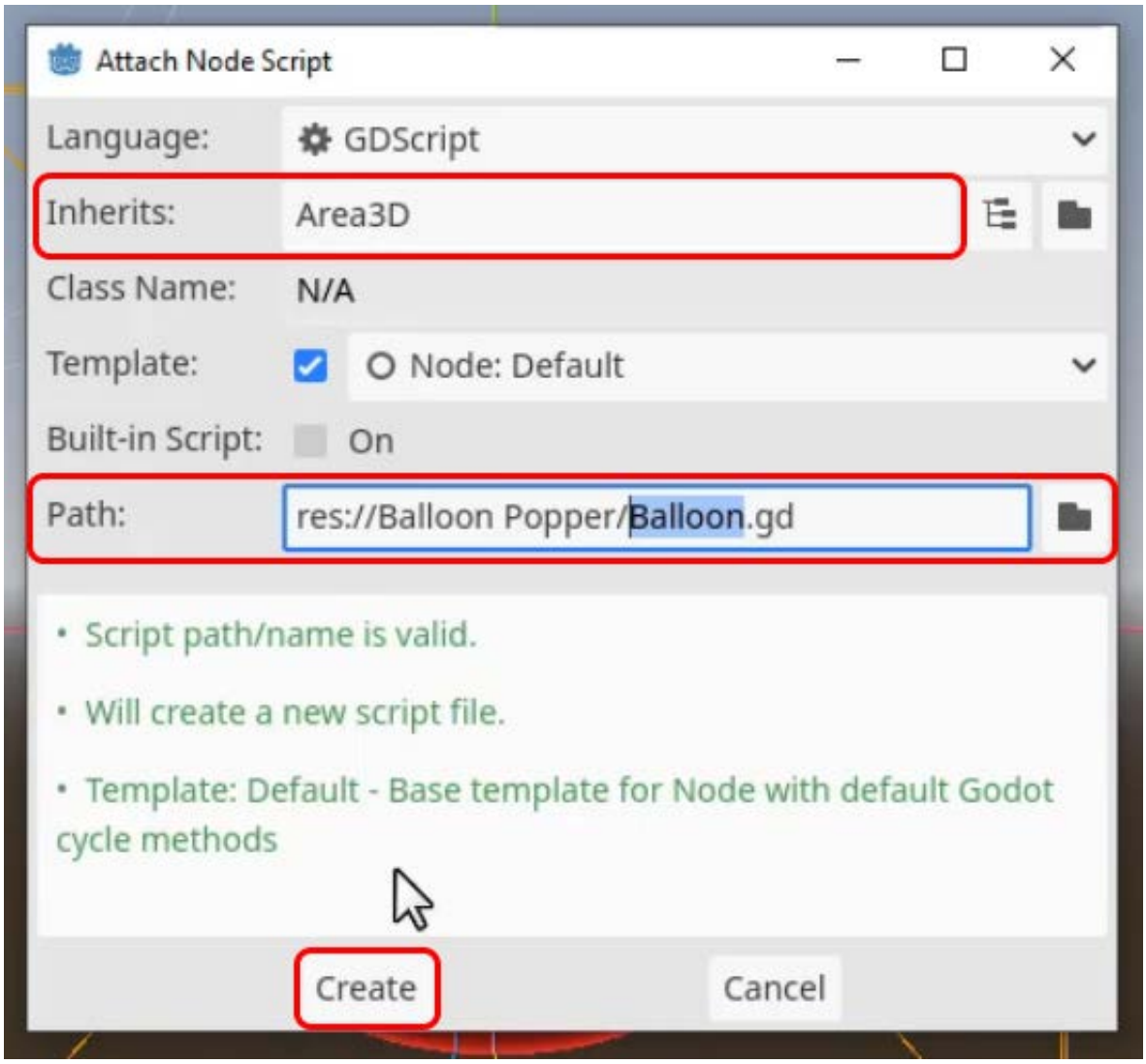


Creating our Balloon Script

To start adding gameplay to our mini-game we want to add the systems to detect mouse clicks on our balloon. To do this we will select our *Balloon* node and create a **New Script** in the *Inspector*.



We will call this script *Balloon.gd* and make sure to save it inside the *Balloon Popper* folder. The script should also inherit from *Area3D* by default, as our *Balloon* node is an *Area3D* node.



This will then open the **Script Editor** and create the **Balloon.gd** script asset in the *FileSystem*. For our script, we won't be using the *_ready* or the *_process* functions so those can be deleted, which will leave you with only the following line:

```
extends Area3D
```

Before adding our mouse click signal, we first need to create some variables. The first will be an **int** variable called **clicks_to_pop** with a default value of **3**. This will be the number of times we need to press a balloon before it pops. We will create these variables at the top of the script, but under the *extends* line.

```
var clicks_to_pop : int = 3
```

We will create another variable called **size_increase** which will be of type **float** and equal to **0.2** by default. This will be how much our *Balloon* increases in size each time we click it.

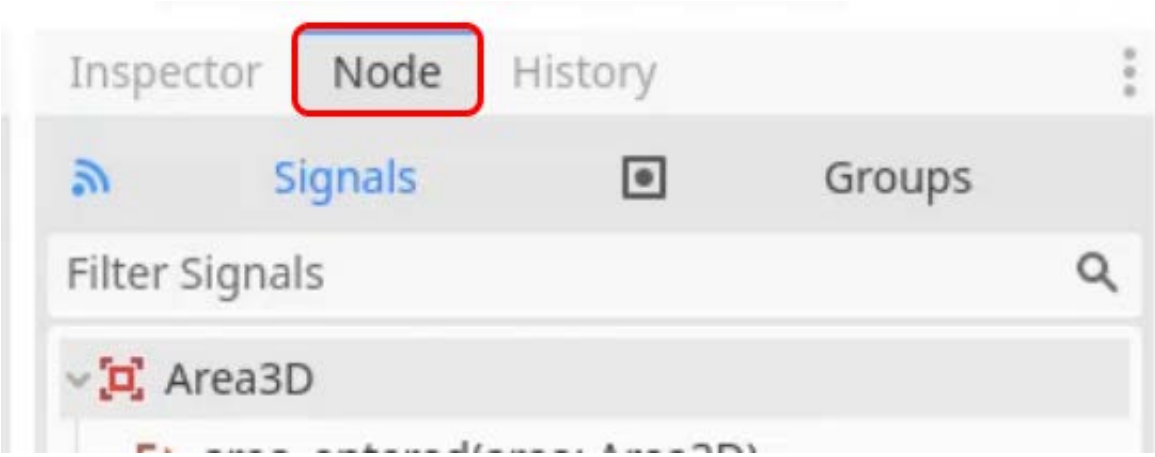
```
var size_increase : float = 0.2
```


Our final variable will be another **int** variable called **score_to_give**. This will be the amount we increase the score by every time the balloon is popped, for now, we will set it to **1**.

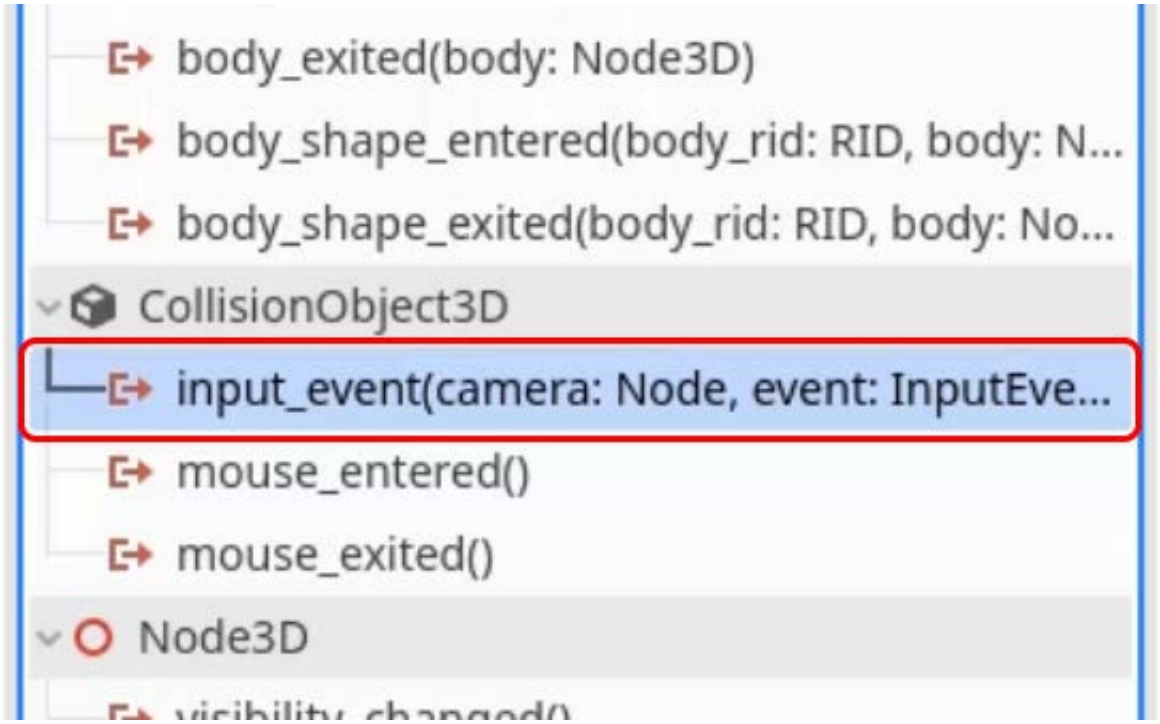
```
var score_to_give : int = 1
```

Detecting Clicks on our Balloon

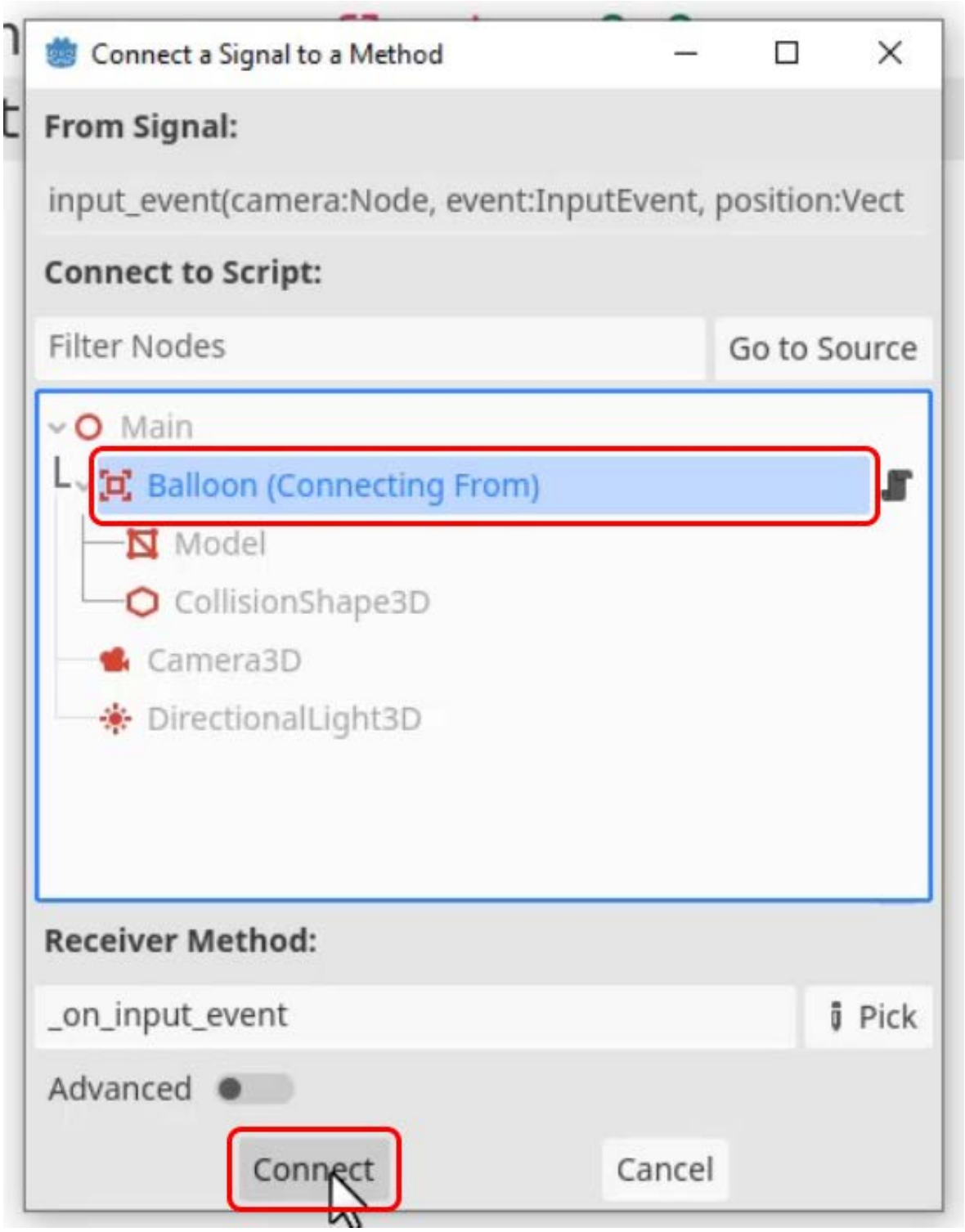
To detect mouse clicks on the balloon, we need to connect a signal to say the balloon has been clicked. To do this we will select the *Balloon* node in the *Scene* window and then go to the **Node Window**.



For our script, we will choose the **input_event** signal, which will call a function when there is an input on our *Balloon* node.



This will bring up the *Connect a Signal to a Method* window, where we can make sure the *Balloon* node is selected and then press **Connect**.



In the *Script Editor*, you will now see a new *_on_input_event* function that we have just connected to the *Input Event* signal. This function has lots of parameters to do with the input, however we will only be using the *event* parameter for this function. We first want to make sure that our function only runs when a mouse click event occurs.

```
func _on_input_event(camera, event, position, normal, shape_idx):
```



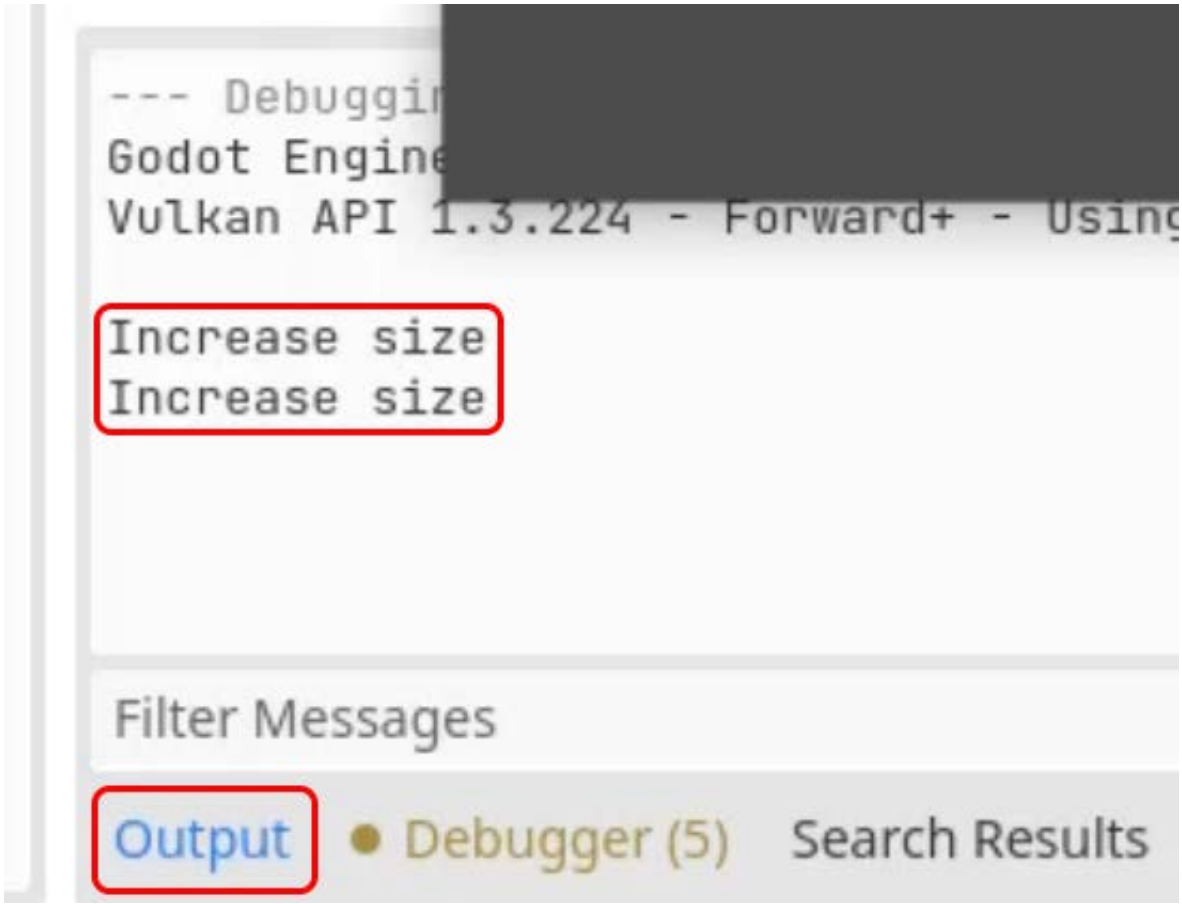
```
    if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and
 event.pressed:
```

This long if statement will check if the event is a mouse button event. If it is, it will then check that the mouse button is the left button, and if so, it will then check if the event is due to the button being pressed. These three sections are separated by the *and* keyword in the if statement, and will only run if each section returns true.

As a test to see if this if statement works, we will print out the string *"Increase size"* to the *Ouptut* window if the user clicks on the *Balloon*.

```
func _on_input_event(camera, event, position, normal, shape_idx):
    if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and
 event.pressed:
        print("Increase size")
```

You can then **save** the script and press **Play**. You will now see if you click the *Balloon* our string will be printed to the *Output* window.



In the next lesson, we will look at increasing the size of the *Balloon* each time it is clicked, along with setting up our scoring system.

In this lesson, we are going to look at making our *Balloon* increase in size each time we click it, along with adding the first part of the scoring system.

Increasing the Scale

In the last lesson, we added the systems required to detect if the player has clicked on the *Balloon* node. We will now build upon this to increase the scale of the *Balloon* and detect if it is big enough to pop, instead of just printing a message to the screen. The first step will then be to increase the scale, to do this we will use this line of code:

```
func _on_input_event(camera, event, position, normal, shape_idx):
    if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and
 event.pressed:
        scale += Vector3.ONE * size_increase
```

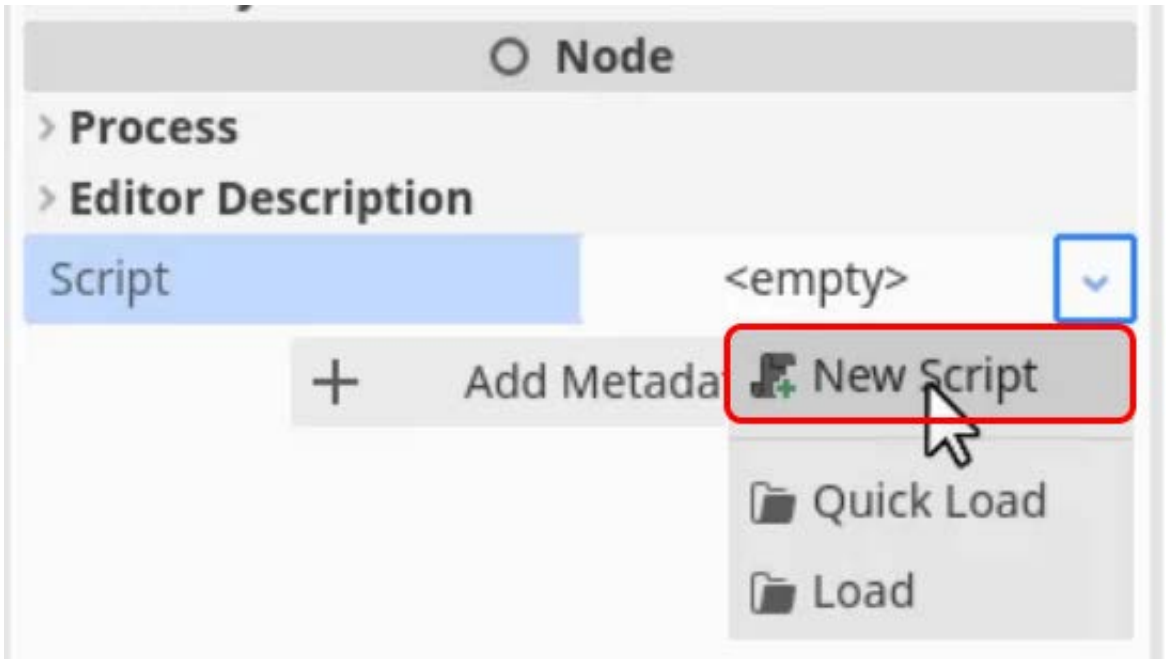
In this case, we use *Vector3.ONE* to increase the scale uniformly across all axes. *Vector3.ONE* is essentially a normal vector, with 1 in every axis (1, 1, 1), so when we multiply it by our *size_increase* value, we change the 1s to the value of the *size_increase* variable. If you now **save** and press **Play** you will see that if you click the *Balloon* it will increase in size each time. To limit the number of times we can click the *Balloon*, we will decrease *clicks_to_pop* each time, and then we will use an if statement to check if *clicks_to_pop* is equal to 0, and if so, pop (destroy) the *Balloon* node. To destroy the node, we will use the *queue_free* method.

```
func _on_input_event(camera, event, position, normal, shape_idx):
    if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and
 event.pressed:
        ...
        clicks_to_pop -= 1

        if clicks_to_pop == 0:
            queue_free()
```

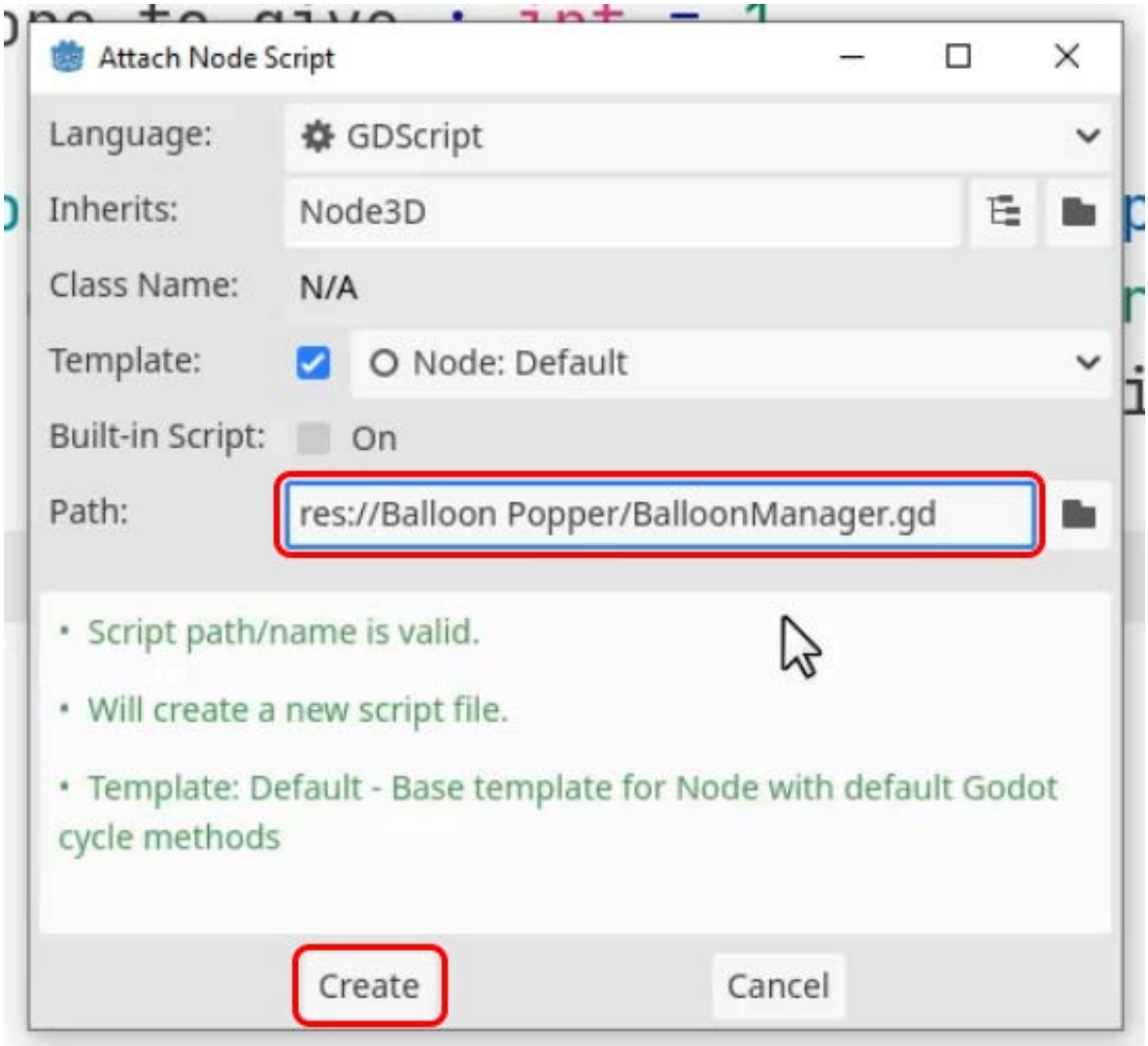
Now when you **save** and **Play** you will see that you can click the *Balloon* three times, with it popping on the third click. You can change the default value of the *click_to_pop* variable to change the number of times we can click the *Balloon*.

Tracking the Score

To keep track of our score, we will need to create a new script called *Balloon Manager*, as we don't want to track the score in a script that will be destroyed when the *Balloon* is popped. We will be using the **root node** (*Main*) to hold our **New Script** as this won't be destroyed during the game.



We will name this script *BalloonManager.gd* and save it in the *BalloonPopper* folder.



As with the *Balloon.gd* script we won't be using the *_ready* or *_process* functions, so we can delete them leaving us with only the *extends Node3D* line. We will then make a variable called **score** of type **int** with a default value of **0**.

```
extends Node3D

var score : int = 0
```

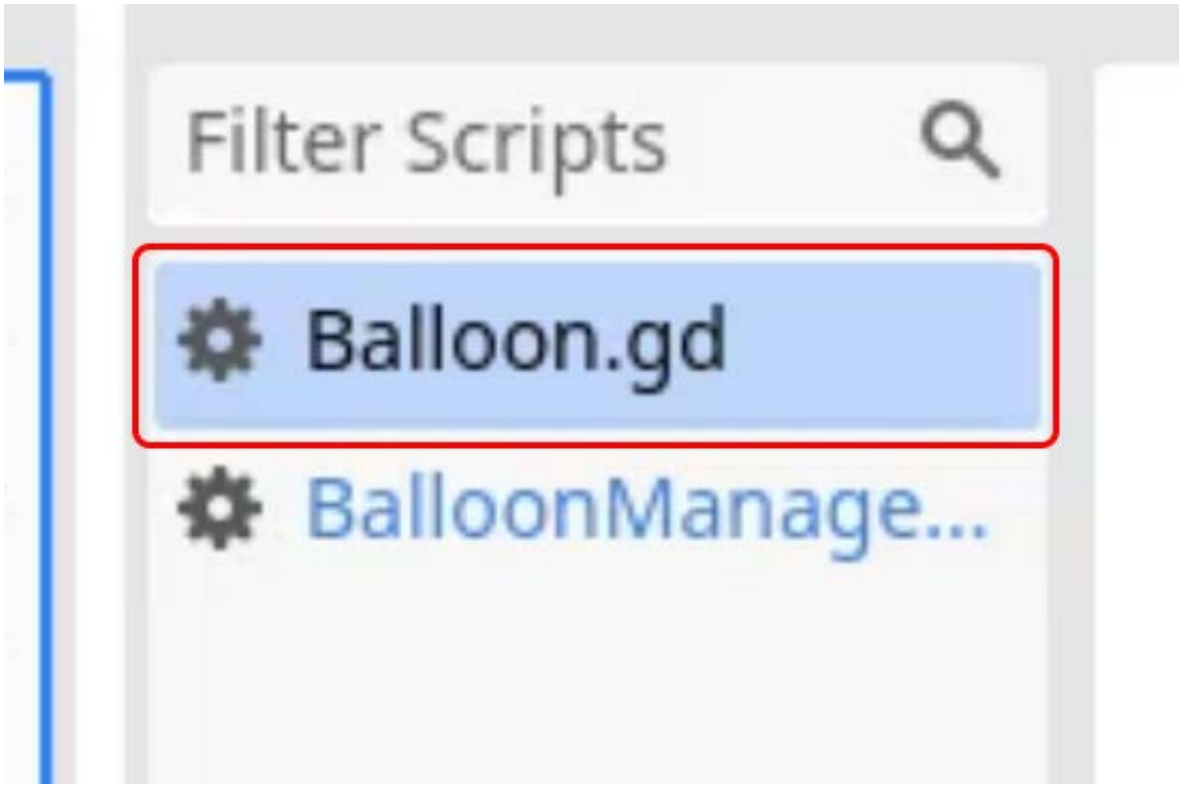
From here we will create a **new function** called *increase_score* with a parameter called *amount*. In this function, we will then add *amount* to our *score* variable.

```
func increase_score (amount):
    score += amount
```

We will also print our *score* value so that we can see changes to the variable.

```
func increase_score (amount):
    ...
    print(score)
```

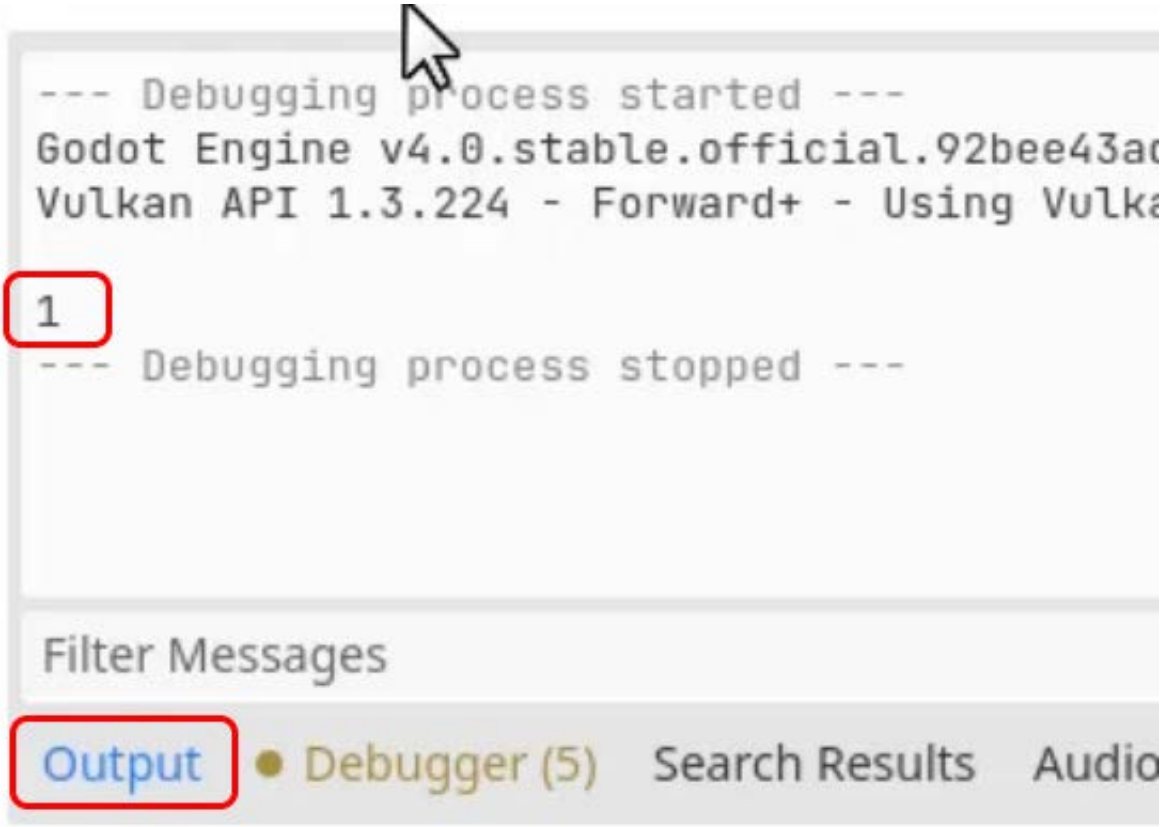
We can then return to the **Balloon.gd** script in the *Script Editor* to make use of the *increase_score* method.



In the *_on_input_event* function we will get the *root* node (as this has our *BalloonManager* script on it) and call the new function before the *Balloon* is destroyed.

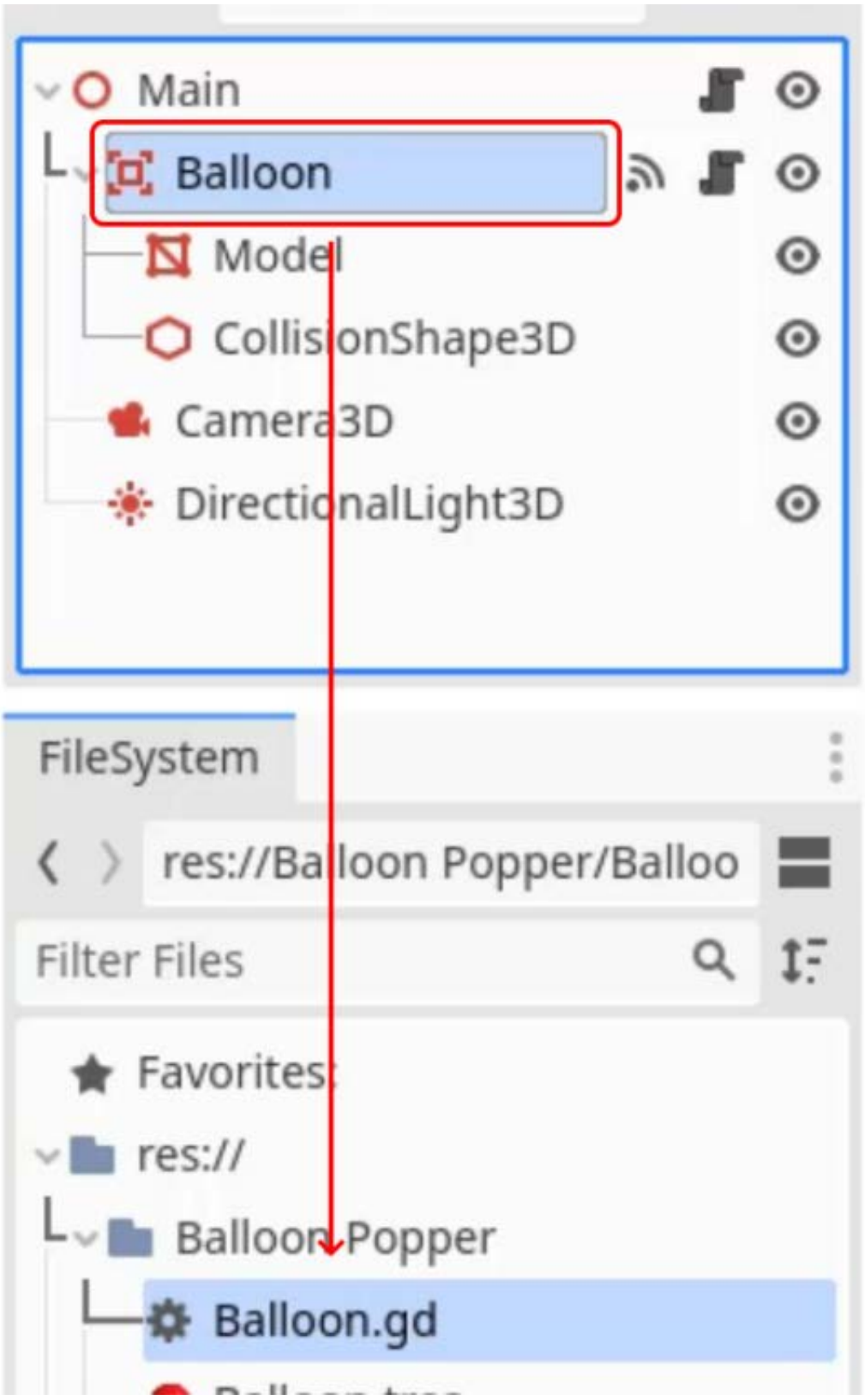

```
func _on_input_event(camera, event, position, normal, shape_idx):
    ...

    if clicks_to_pop == 0:
        get_node("/root/Main").increase_score(score_to_give)

        queue_free()
```

Now if you **save** both scripts and press **Play**, you will see that if you pop the *Balloon* a score of 1 will be printed in the *Output* window.

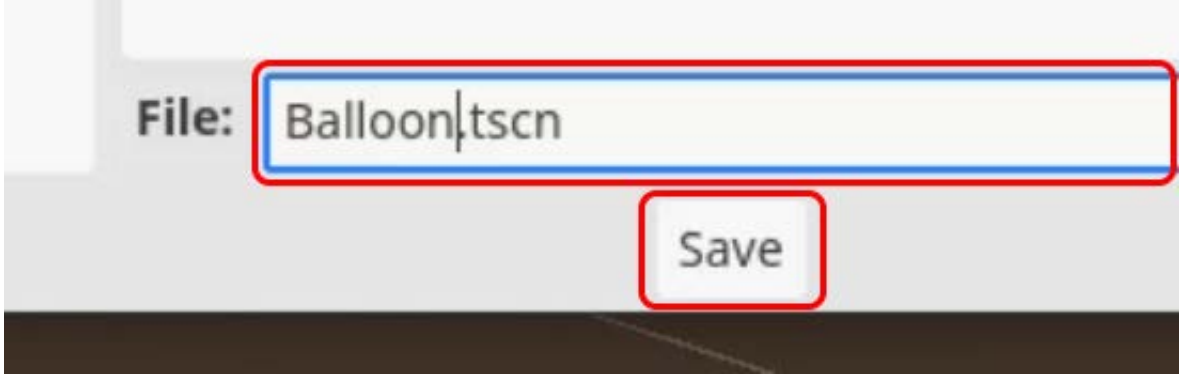


Adding Multiple Balloons

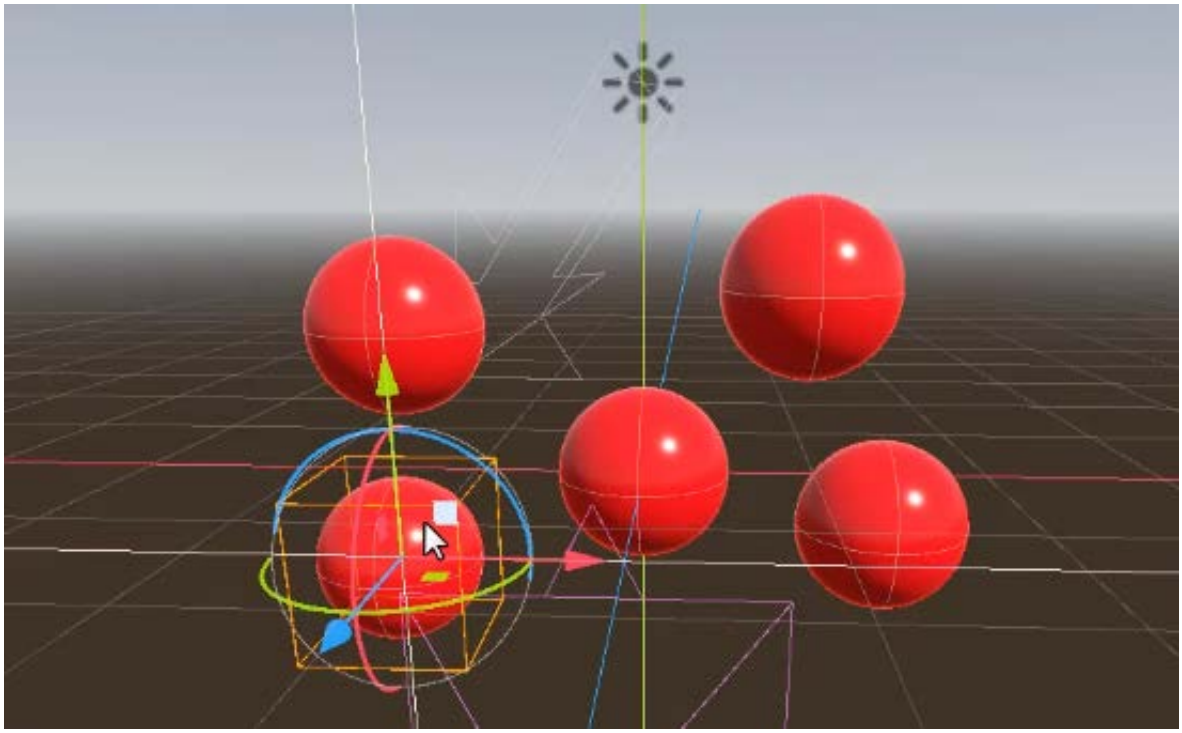
We now want to turn our *Balloon* node into a scene, so that we can make any changes to the scene and they will change across all instances of it. To do this, we will drag the **Balloon** node into the **FileSystem**.



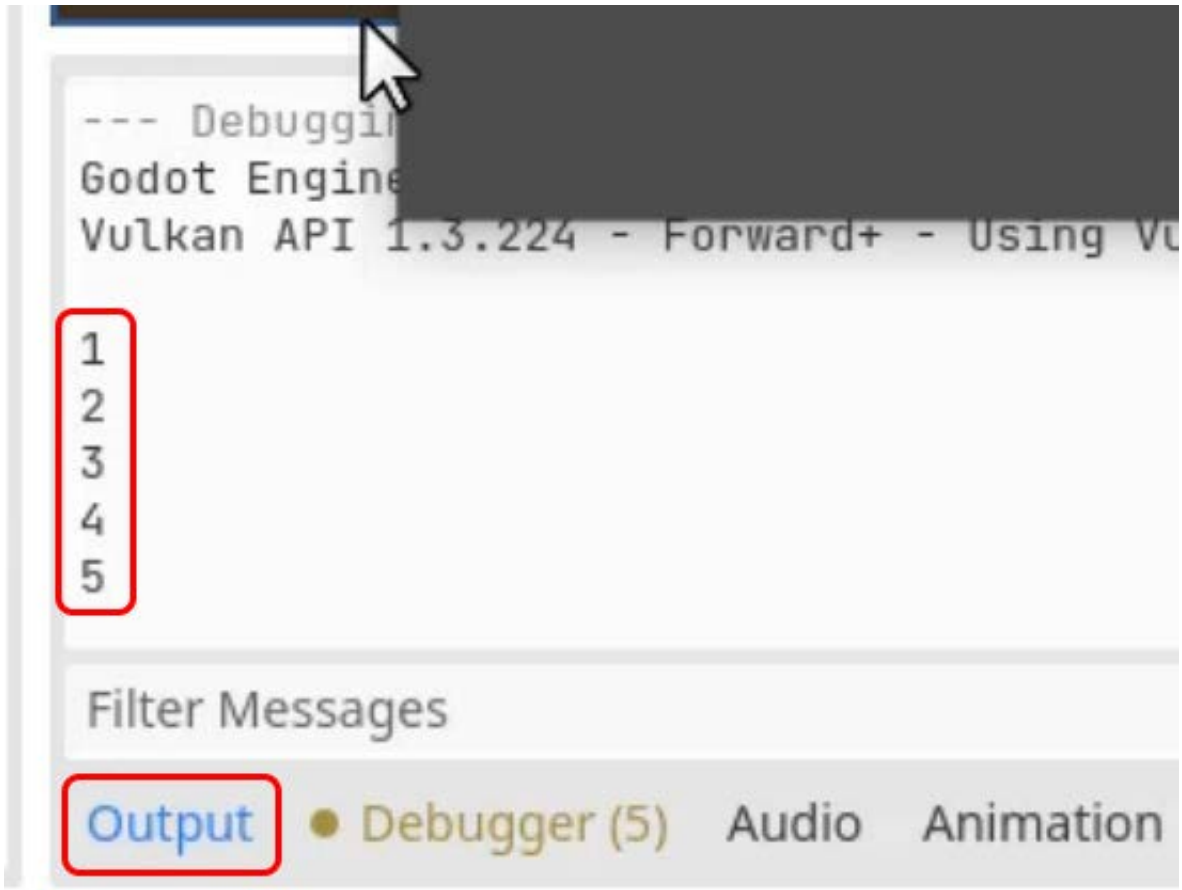
From here, we will name the scene *Balloon.tscn* and press **save**.



You can now **duplicate** (**CTRL+D**) the *Balloon* scene around the level to create multiple balloons to pop.



Now if you press **Play** you will see you can pop the balloons and increase the score, which can be seen in the *Output* window.



This isn’t a good way of checking the score, however, so in the next lesson we will look at showing the *score* value as some text in the top left of the screen. Along with this, we will also look at having different values for the variables in each instance of the *Balloon* scene.

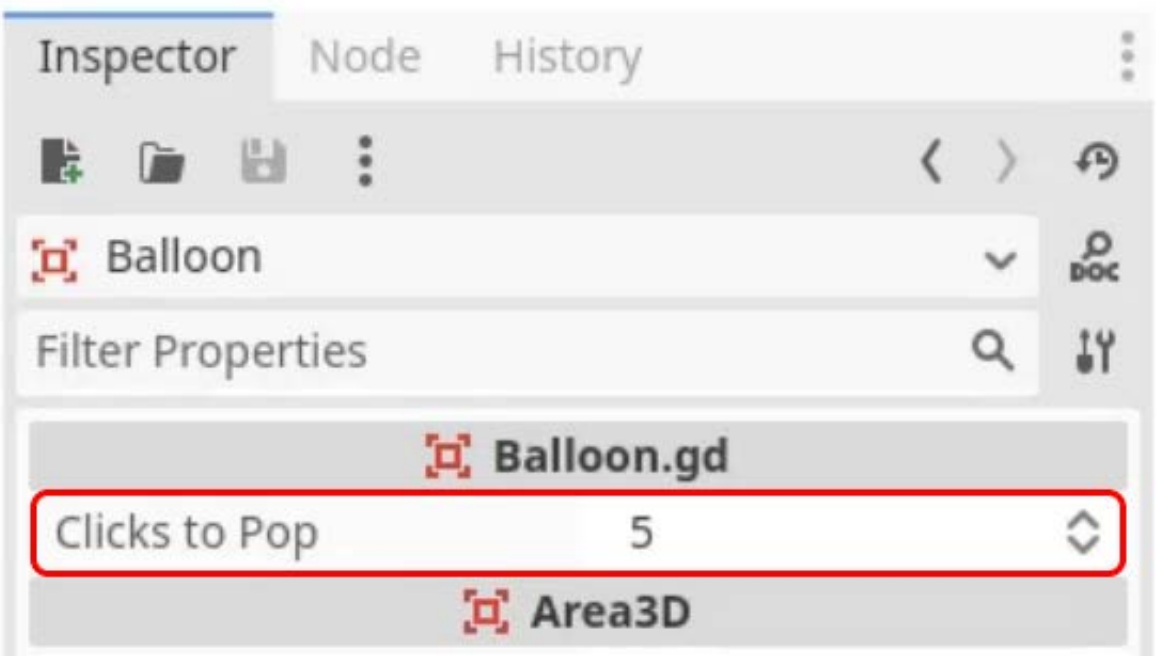
In this lesson, we are going to finish our *Balloon Popper* mini-game by adding the final touches.

Different Values for each Balloon

Currently, each balloon has the same values for the variables, such as *clicks_to_pop*. What we want to do is allow each *Balloon* instance to have its own versions of these variables, so that each balloon can pop after a different number of clicks. To do this, we need to add the keyword **export** before the variable definition in our *Balloon.gd* script.

```
@export var clicks_to_pop: int = 5
```

This will make the *clicks_to_pop* variable available in the inspector when any of our *Balloon* nodes are selected. Here we can change the value of the variable for the selected instance of the *Balloon* scene. You can try changing the values around, and then press **Play** and you will see each *Balloon* you changed takes a different amount of clicks to pop. If you don't change a value, it will be set to the default value we set in our script, in this case, that is 5.



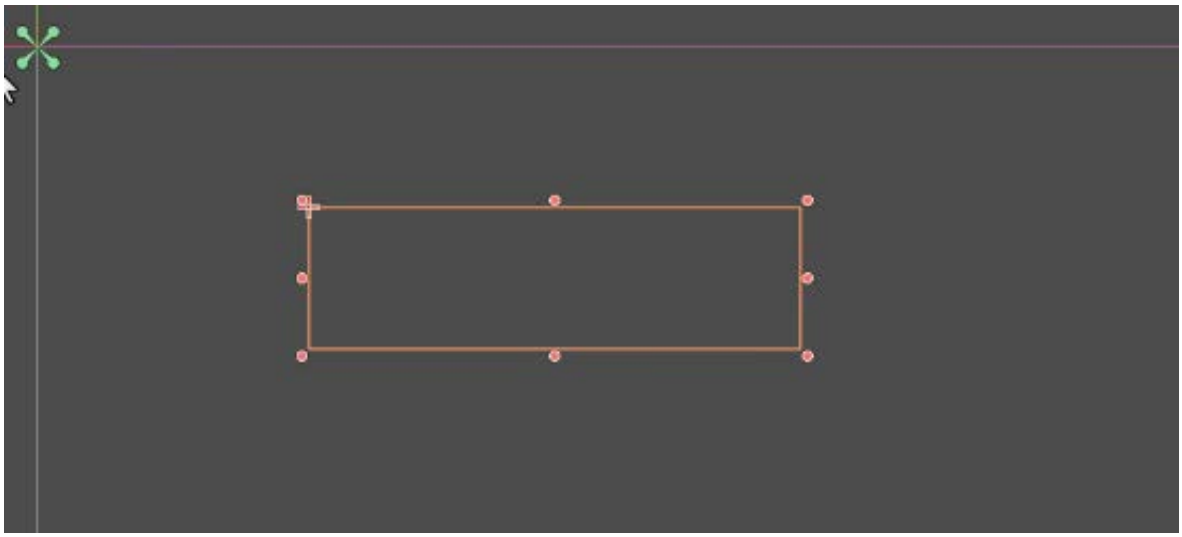
We can also add the *export* keyword to each of our other variables, to expose them in the inspector.

```
@export var clicks_to_pop: int = 5
@export var size_increase : float = 0.2
@export var score_to_give : int = 1
```

Creating a Score Label

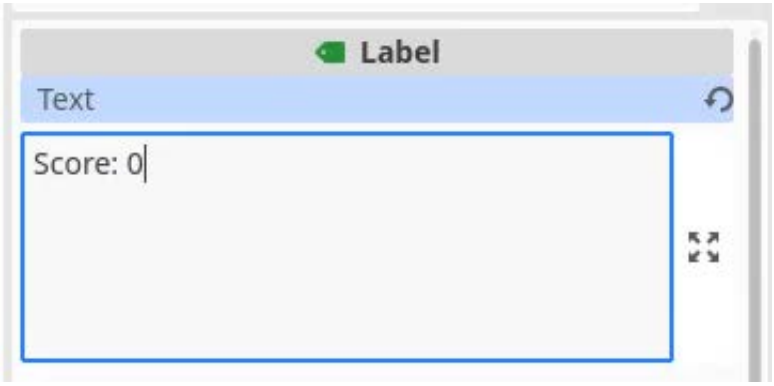
We also want to set up some UI text to display our current score. To do this, we create a new **Label** node.

You will notice that when you create the *Label* it will automatically change to *2D* mode. This is due to *UI* items being 2D objects as they apply directly to the player's screen. You will notice an orange box with orange circles on the edges, in the viewport window, this is our *Label.* We can use this box to move it around, by clicking and dragging it, along with the orange circles to scale the box to a new size.

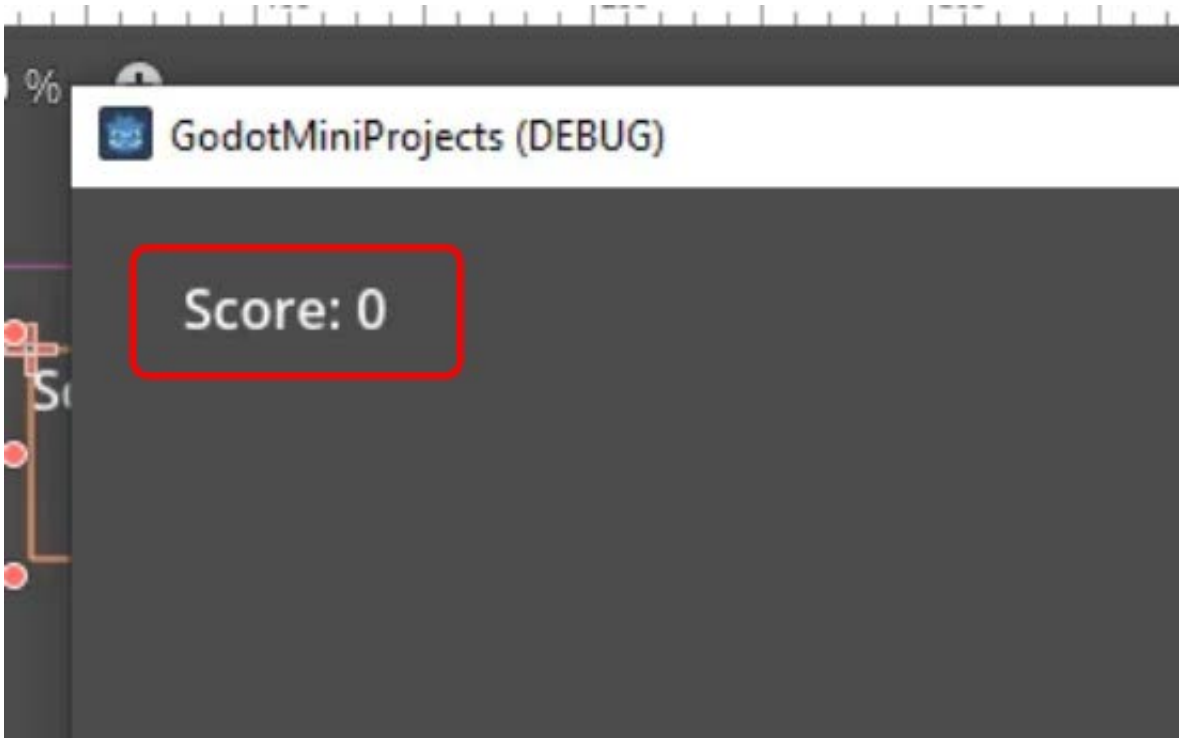


You will also notice a thin-blue rectangle in the viewport. This represents the size of the player's screen, so you can place the label anywhere in this box and it will be the same size and position on the screen, that it is in the blue rectangle. As this is our score text, we will leave it in the top left corner of the screen.

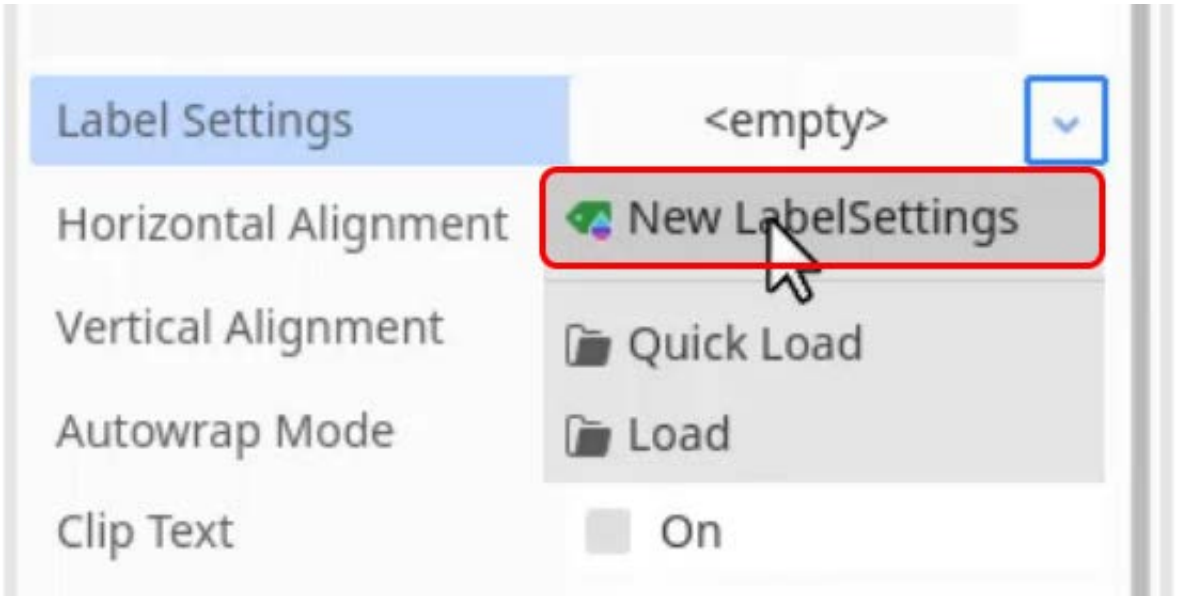
If you type something in the **Text** property of the *Inspector*, you will see that it appears in the *Label*. So we will set the *Text* as "*Score: 0*" to represent how it will look in our game.



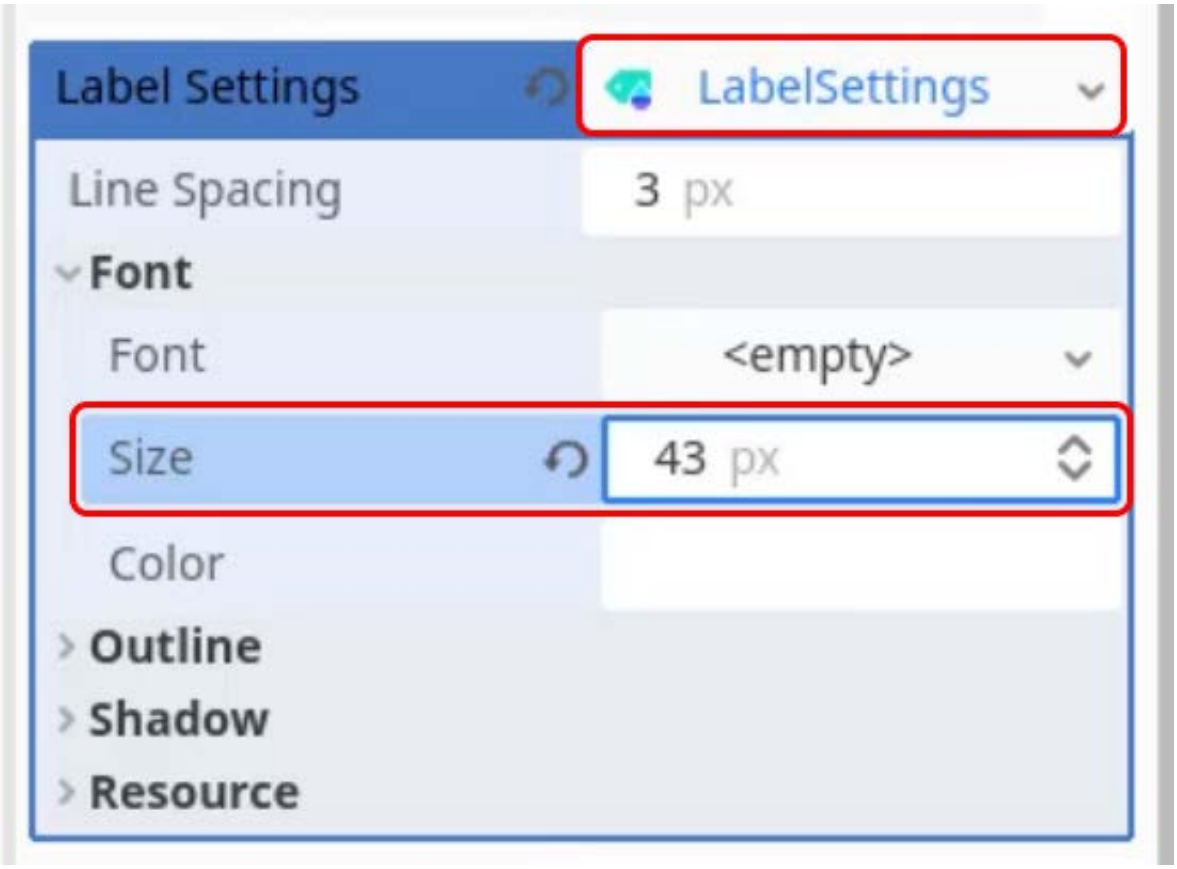
If you now press **Play** you will see the *Label* being displayed in the correct place on the screen.



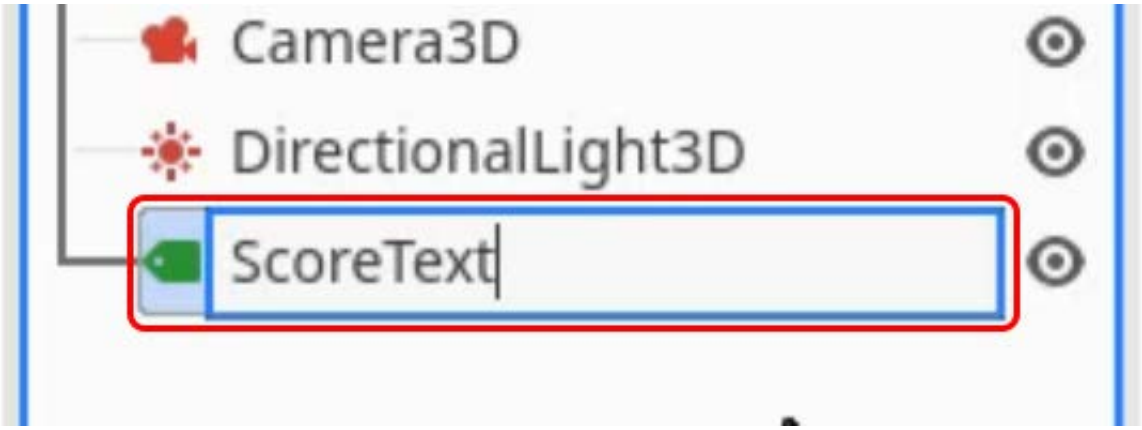
However, the text is a bit small, so to fix this we will create a **New LabelSettings** property for our *Label* node.



You can then click it to expand the *LabelSettings* and open the **Font** dropdown, where we can increase the **Size** property of our text.



Finally, we want to rename the *Label* to *ScoreText* so that we can easily identify it.



Modifying the Score Label

To change our *ScoreText* label to represent our *score* variable while we are playing, we will need to add some code to the **BalloonManager.gd** script. The first step will be to store our *ScoreText* node as a variable. We will be doing this using the *@export* tag which will allow us to assign the variable in the inspector.

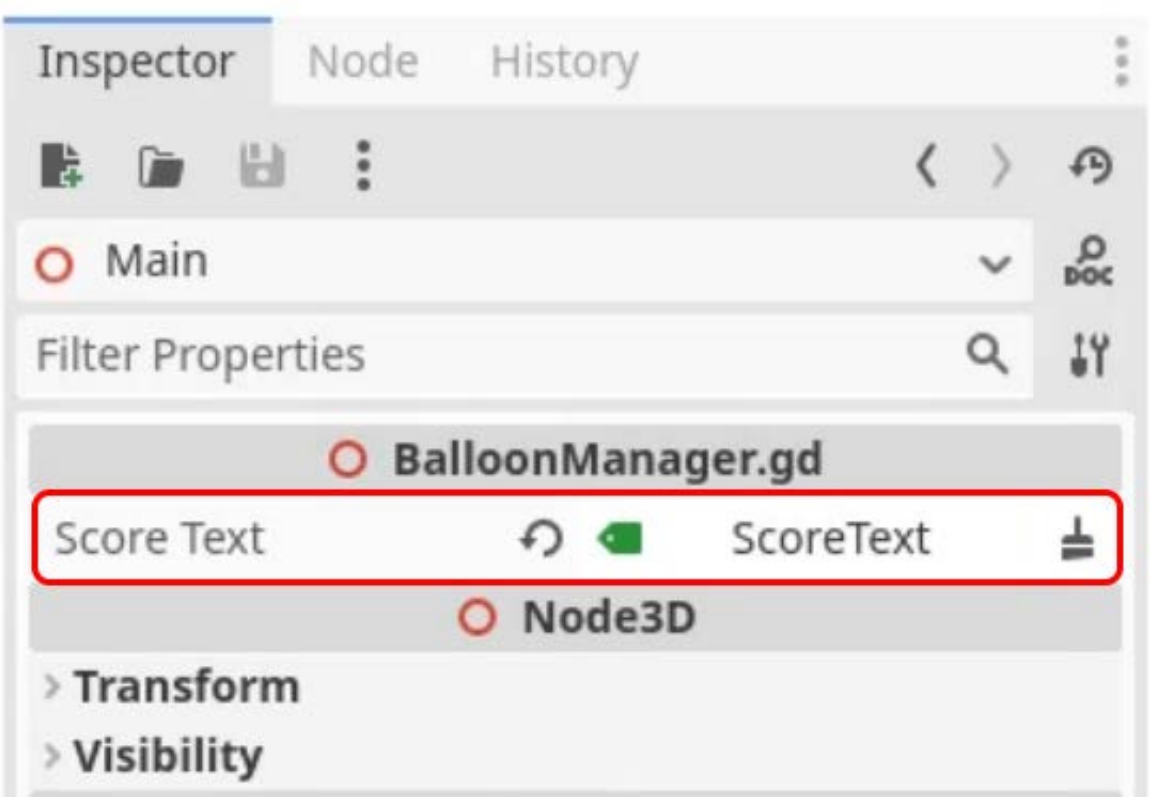
```
@export var score_text : Label
```

Then in the *increase_score* function, we will replace the *print* statement with a new line of code that

changes our *score_text*'s text value to be our score.

```
func increase_score (amount):
    ...
    score_text.text = str("Score: ", score)
```

Here we use the *str* function which essentially combines multiple variables together to create one larger string. To finish our script, we will select the **Main** node in the *Inspector* and assign our *Score Text* property to the *ScoreText* node that we created earlier.



Now when you press **Play** you will notice the score label increases with each *Balloon* you pop.

With the score completed, that is our first mini-game complete! In the next lesson, we will begin looking at the physics-based mini-game.

Let's look back on our balloon popper project and go over some aspects in a bit more detail, as there were quite a few things introduced in these lessons.

Input Event

On our Balloon node, we are connecting to the **input_event** signal, which gets emitted once an input has taken place on the balloon.

We are connecting this signal to the **_on_input_event** function in the Balloon script.

```
func _on_input_event(camera, event, position, normal, shape_idx):
    if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and
 event.pressed:
        # run code
```

A lot of data gets sent over when this signal is called, but we are only checking for three things in our if statement:

1. Is the event related to a mouse button?
2. Is the button that was pressed the left mouse button?
3. Has the event been pressed down?

If all those criteria have been met, then we run the respective code of increasing the balloon's size, etc.

String Formatting

In our **BalloonManager** script, you will notice what we are calling the **str()** function. This is what's known as a string formatting function. It can take a string, multiple strings, integers, floats, booleans, etc – and combine them together in one single string.

```
score_text.text = str("Score: ", score)
```

We are combining the string "Score: " with the integer number of our variable **score**. If score was 15, the resulting string would look like this, "Score: 15".

Additional Resources

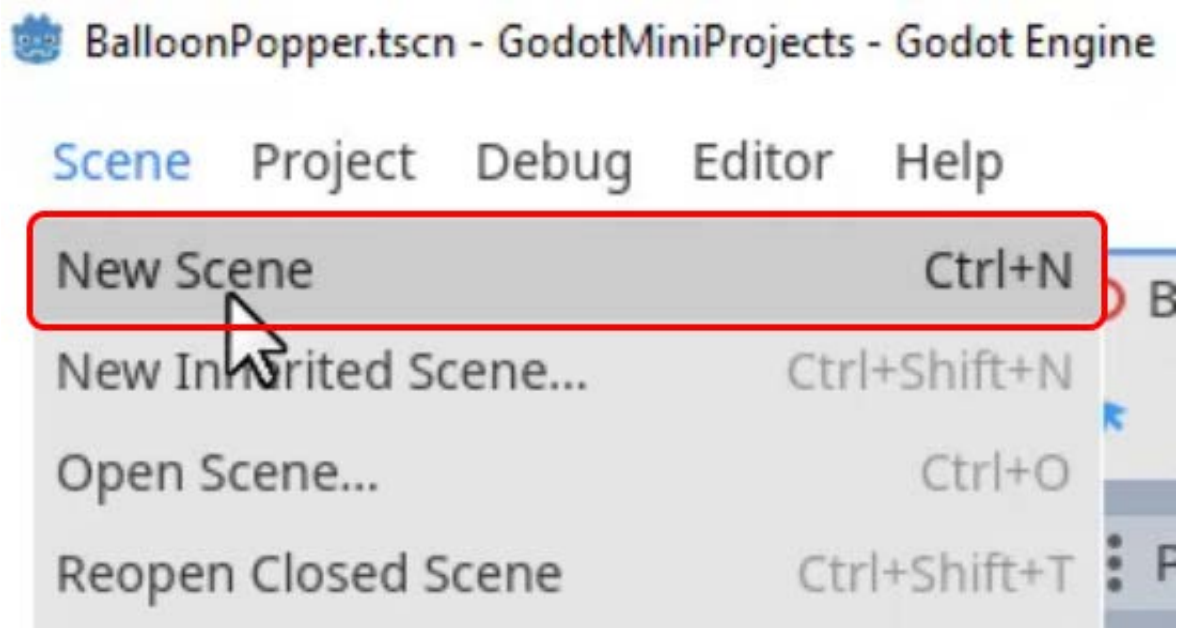
If you wish to learn more, then you can refer here to the Godot documentation.

- Input Events
- Formatting Strings

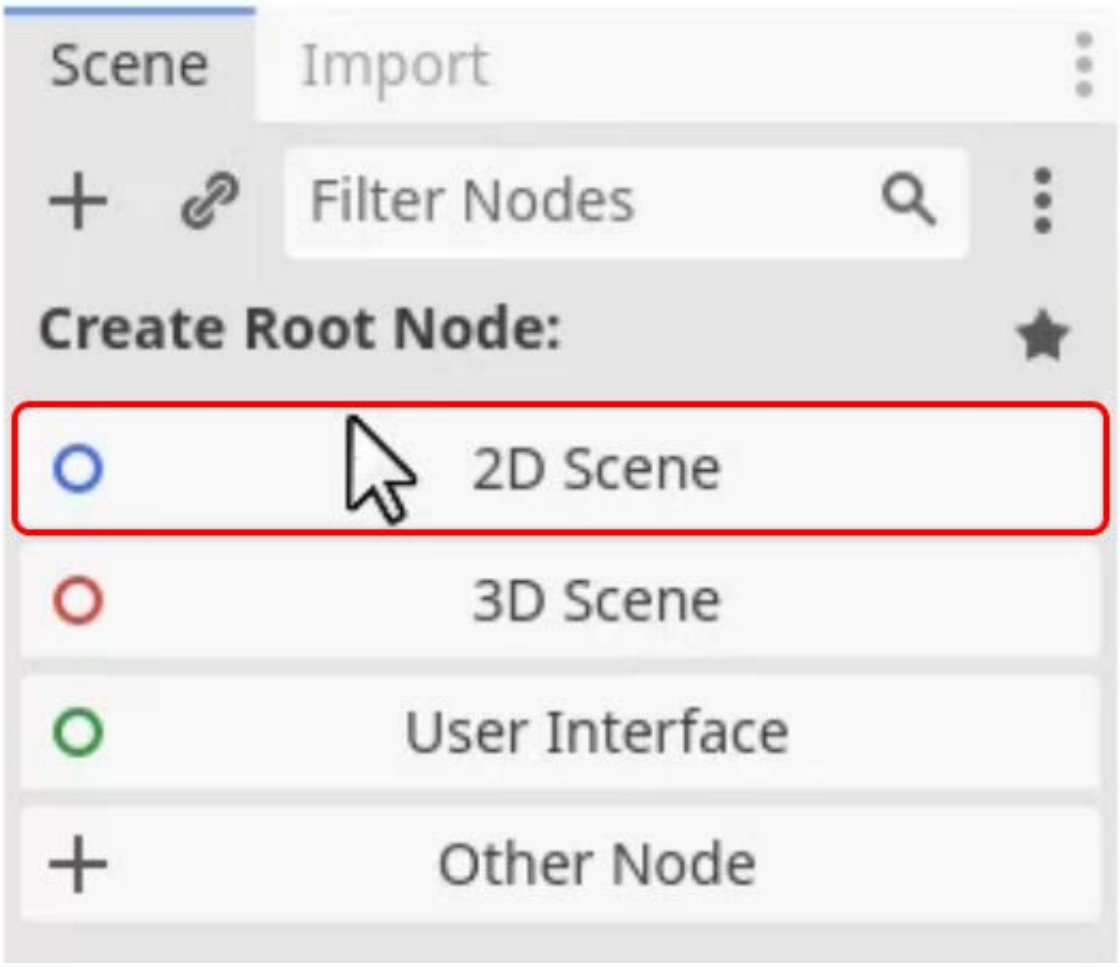
In this lesson, we are going to begin looking at our physics-based mini-game. We are going to use this mini-game to learn Godot's physics system along with how to push a player around based on mouse clicks.

Creating a New Scene

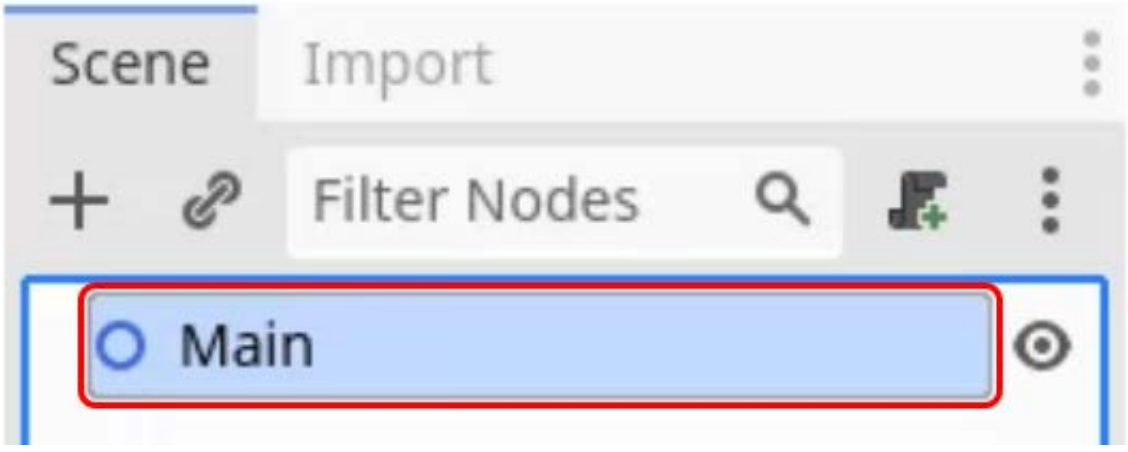
To start with, we want to create a new scene for our mini-game to take place in.



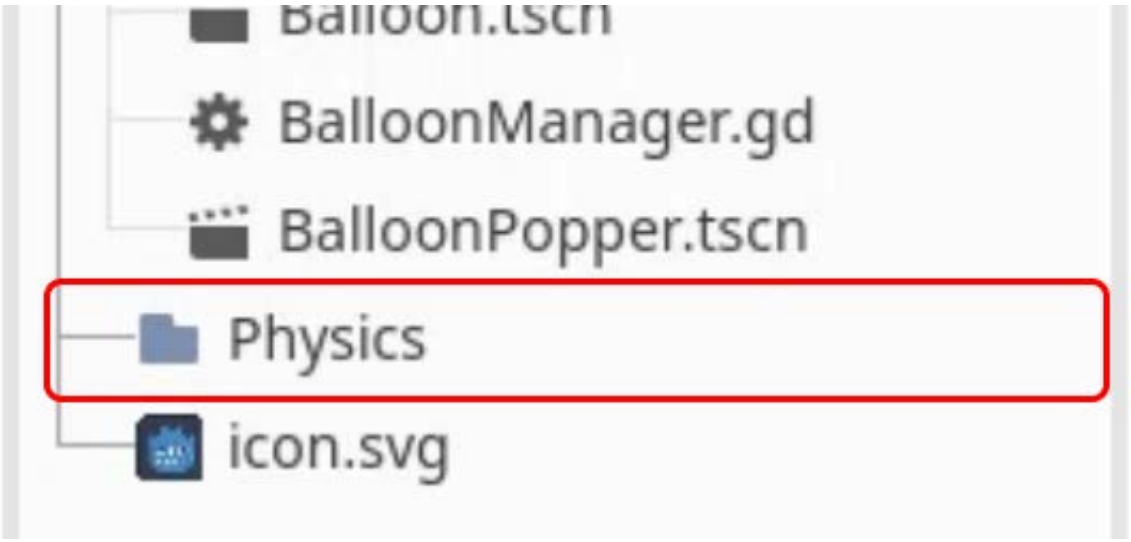
We will make this a **2D** project, and use the *2D Scene* button to select the root node.



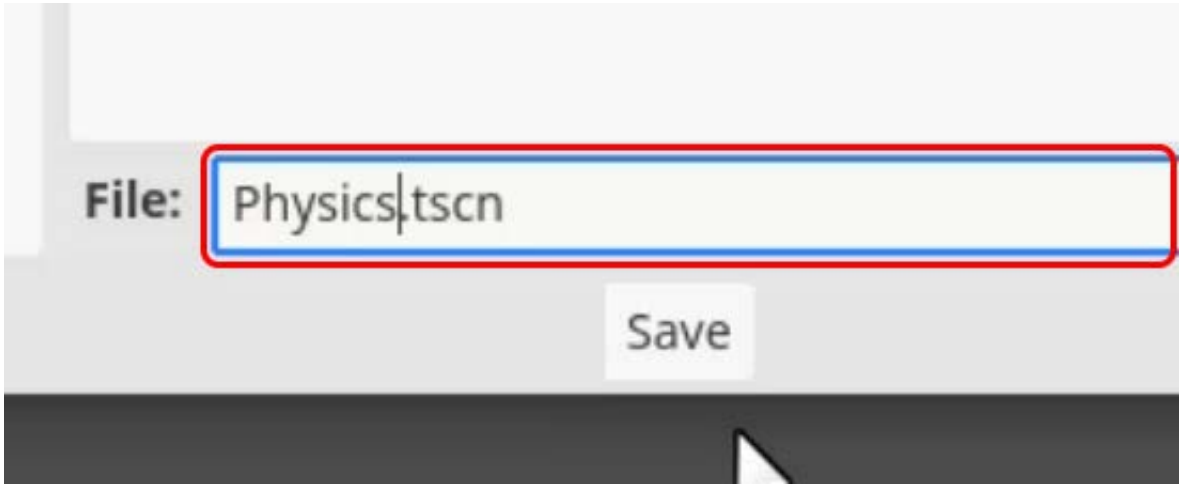
We will also rename the root node to *Main.*



Before saving our scene, we want to create a new folder in the *FileSystem* which we call *Physics*.

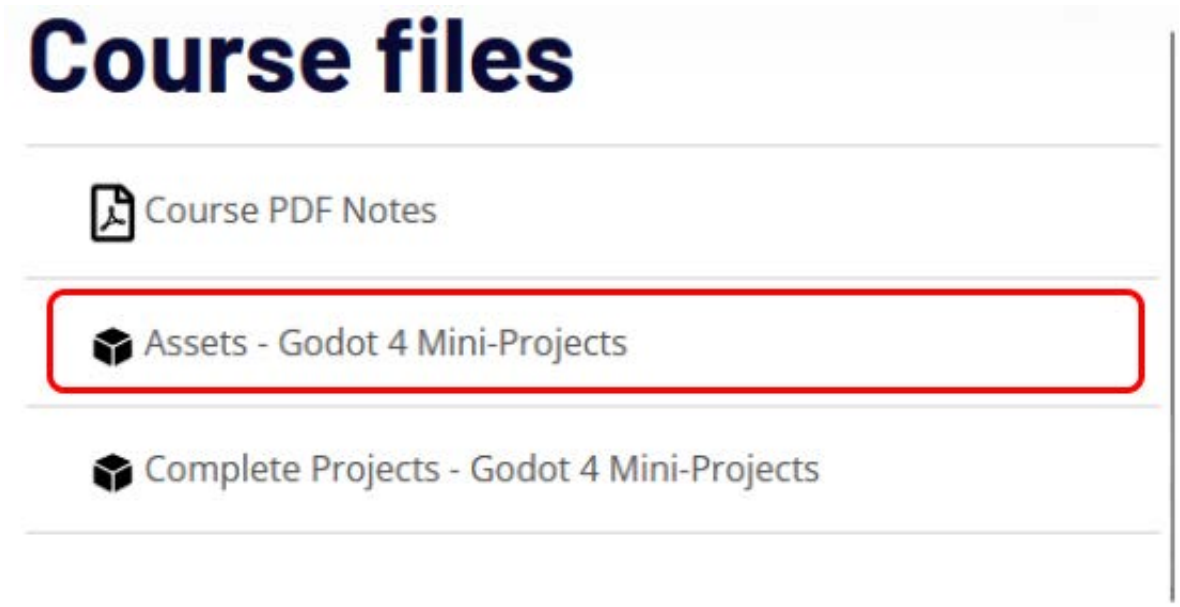


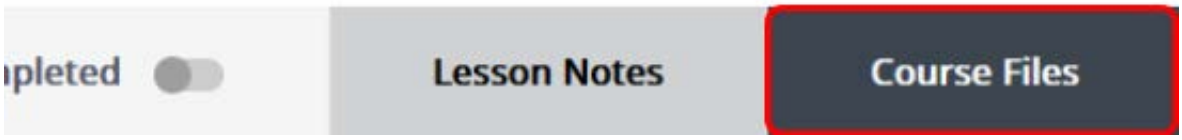
You can then **Save The Scene** in the new folder and call it *Physics.tscn*.



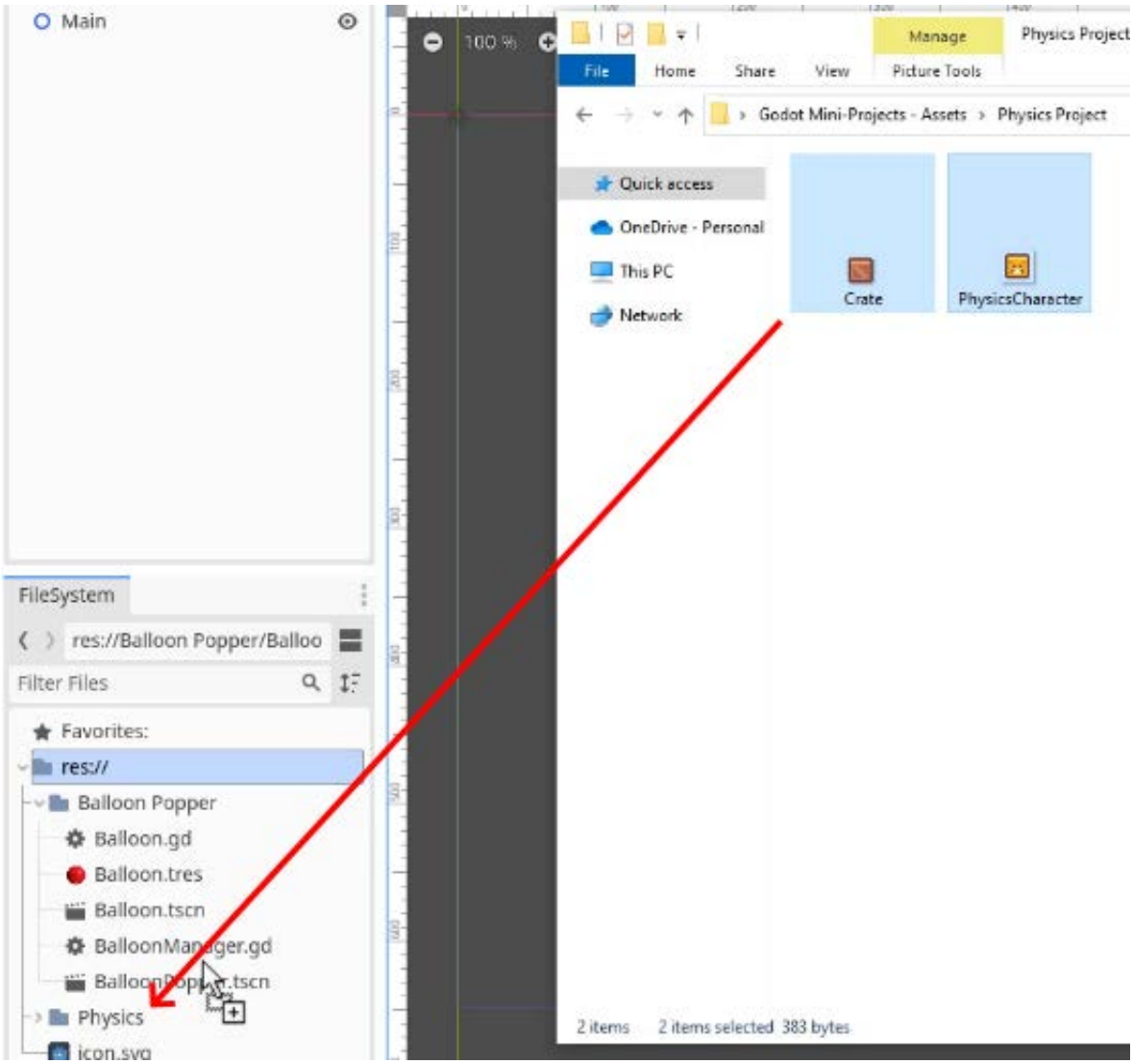
Creating our Player Nodes

The first thing we need to create for our game is the player character. To do this we will need a sprite. As part of this course, there is a downloadable asset pack you can use for the sprites of our player, although of course, you can use your own assets if you would prefer. You can find the files in the *Course Files* section under the name *Assets – Godot 4 Mini-Projects*.



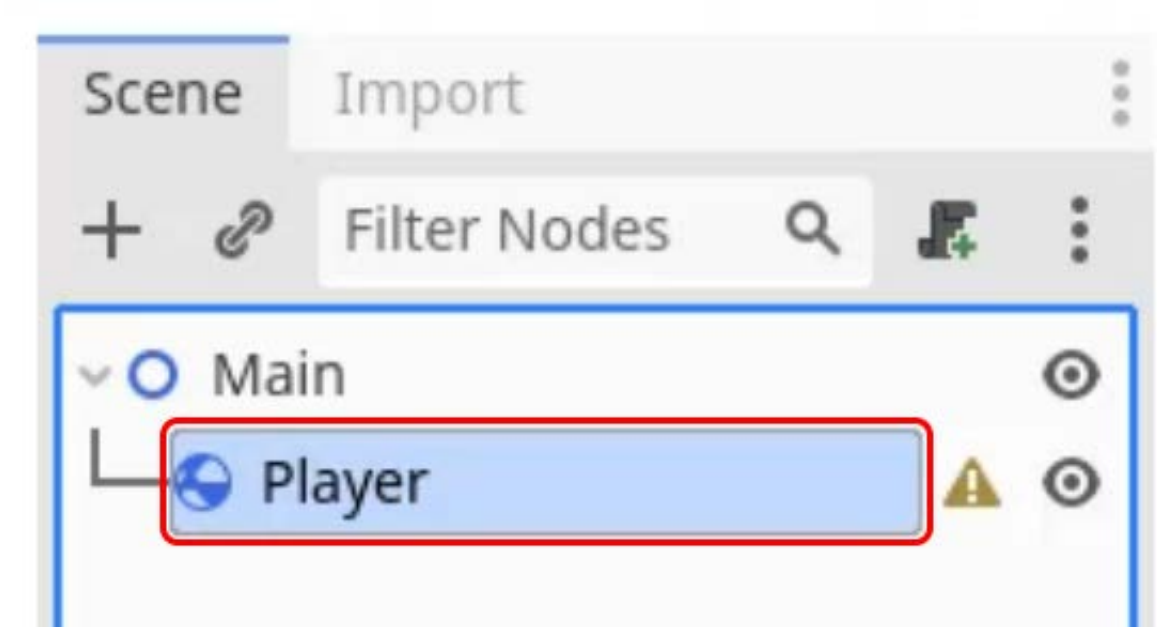


When you download and extract the files you will find two folders inside, we will be using the two sprites found inside the **Physics Project** folder. To use them in Godot we will drag the files inside the *Physics* folder in the *FileSystem*.

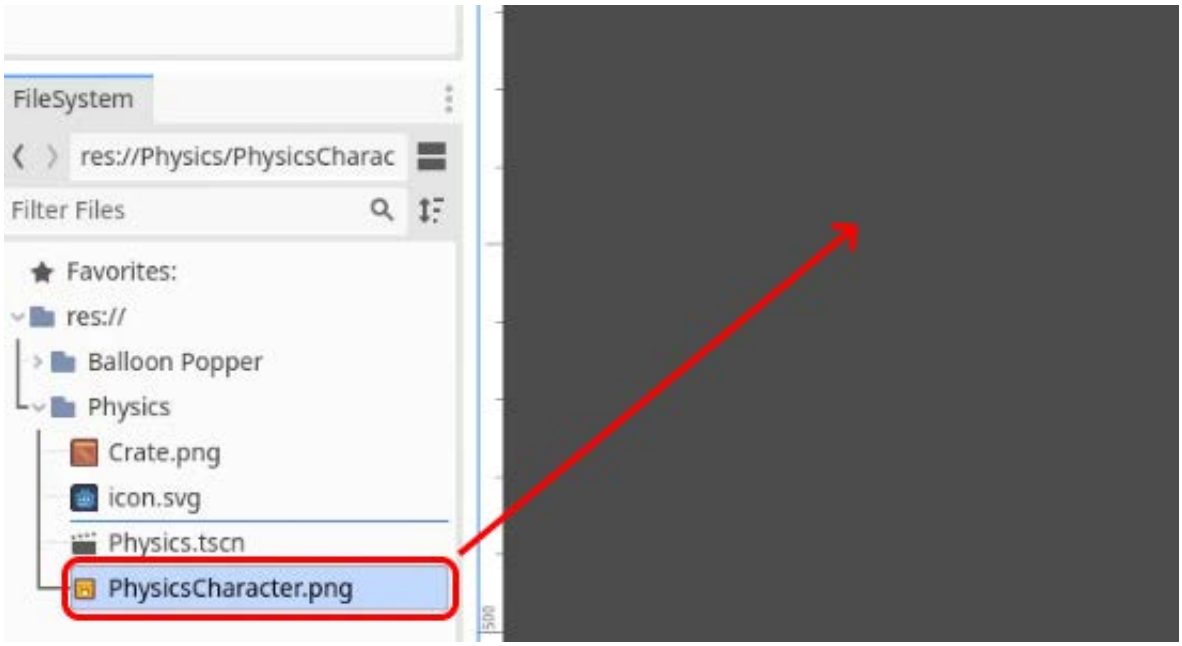


For the root node of our *Player,* we will use a **RigidBody2D** which will allow us to have the player be affected by the *Godot Physics System*.

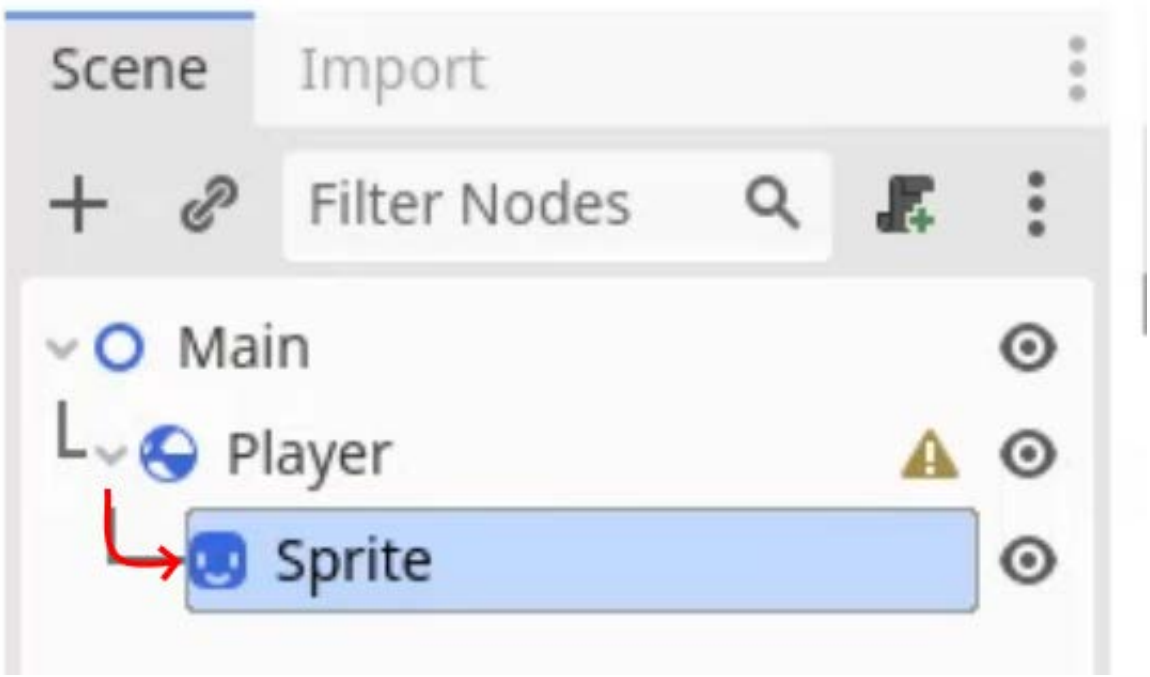
We can rename this to *Player* to easily identify it.



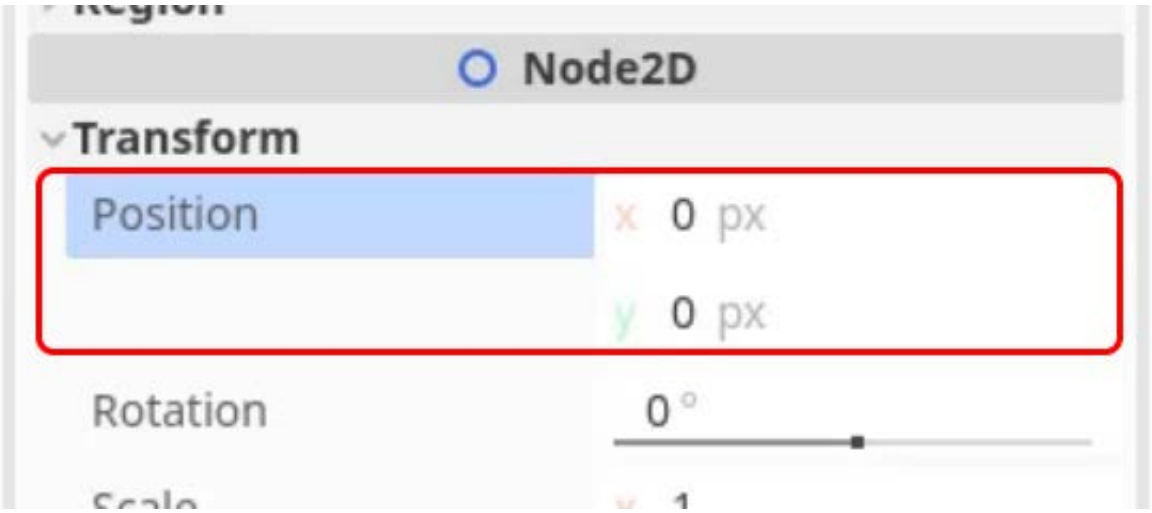
We can then drag the *PhysicsCharacter.png* asset into our viewport to instantiate it as a *Sprite* node.



Next, rename the node to *Sprite* and make it a child node of our *Player* rigid body node.

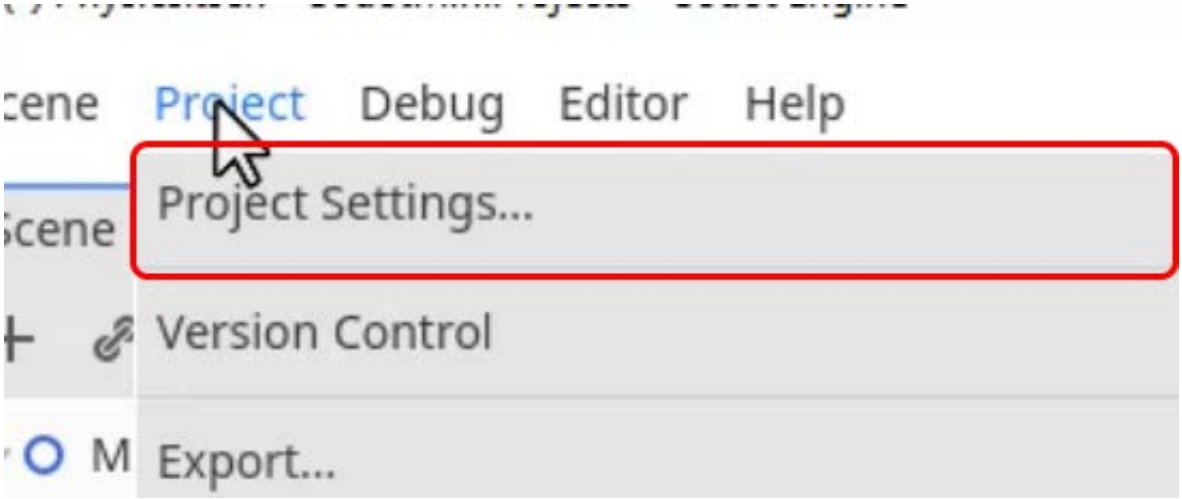


Finally, set the **Position** property of the *Sprite* to **(0,0)** to center it on the *Player* node.

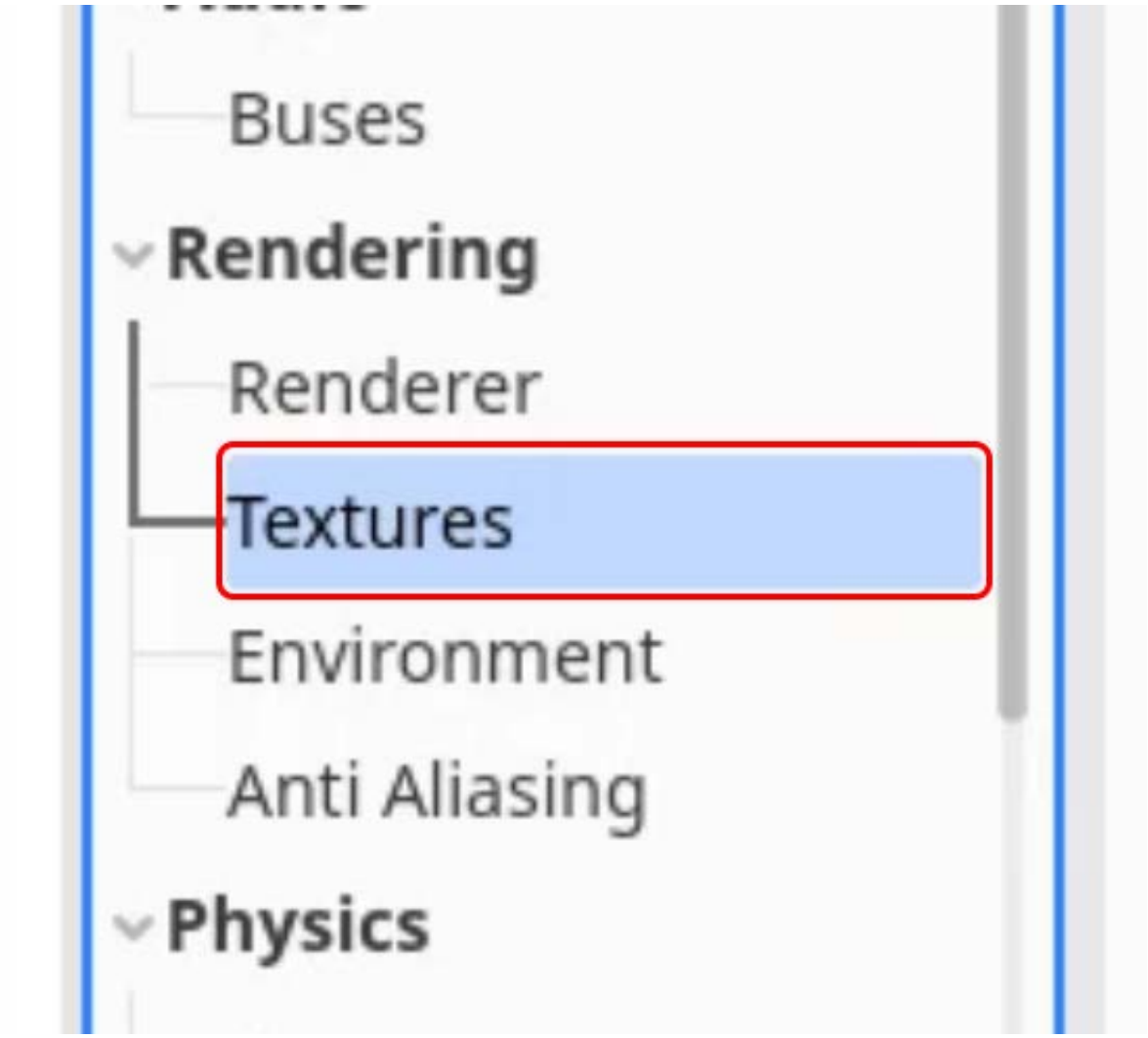


Fixing Blurry Pixel Art

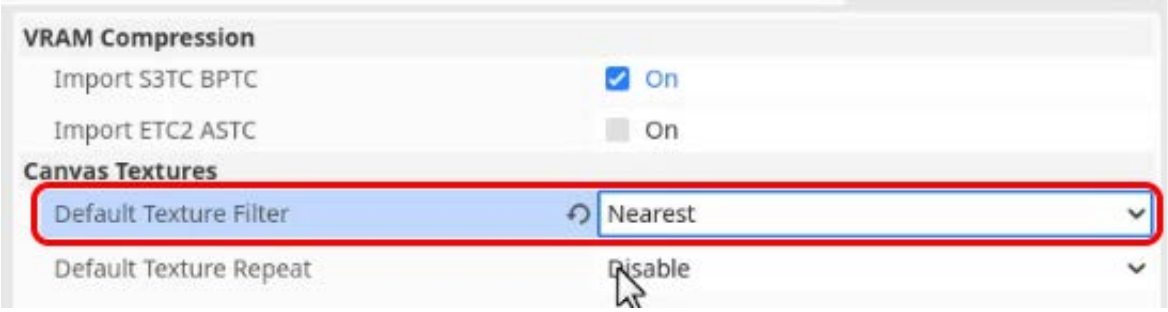
You will notice that our pixel art appears blurry in Godot by default. To fix this we need to change the *filtering mode* of our textures. We can do this by opening the **Project Settings** window.



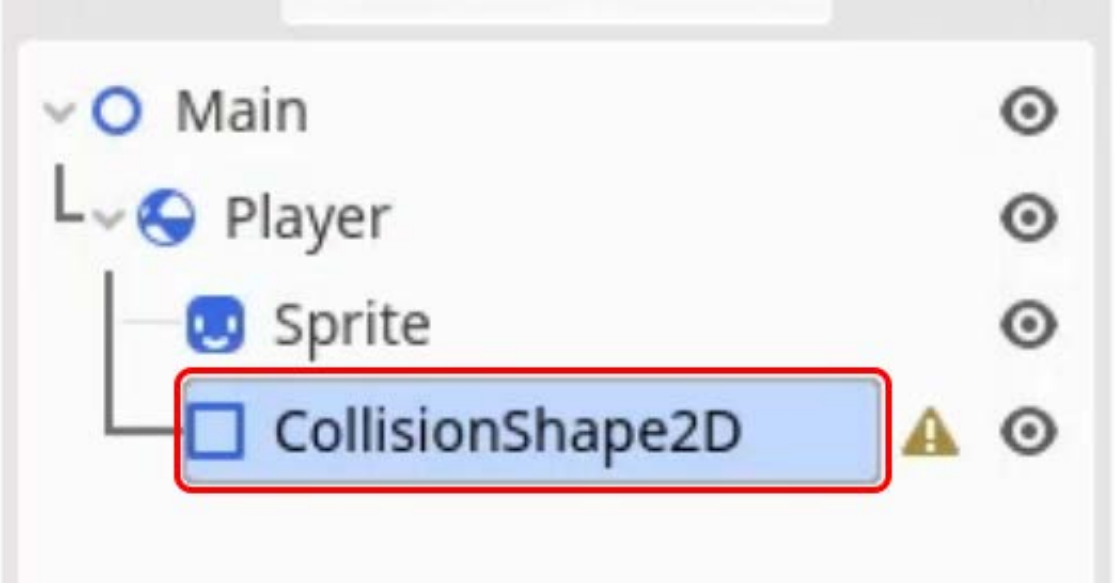
Then select the **Textures** tab.



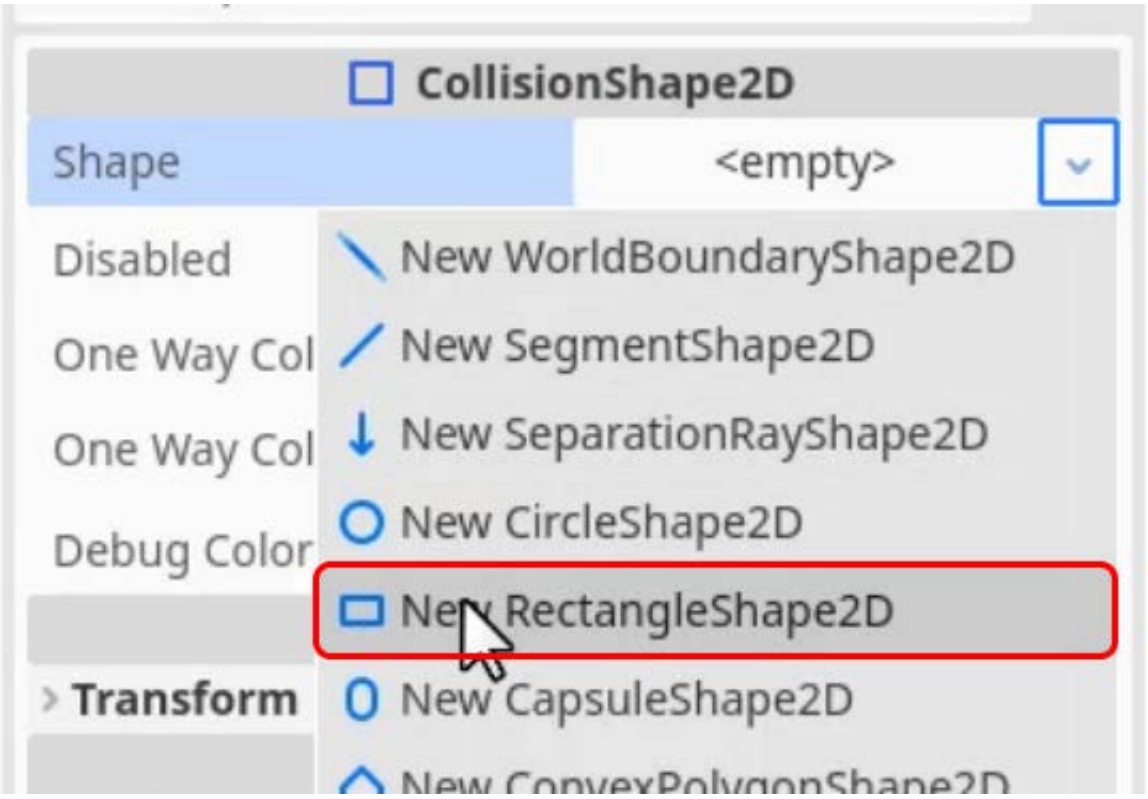
Finally, change the **Default Texture Filter** value to **Nearest**. This will make our pixel art sprites appear much sharper.



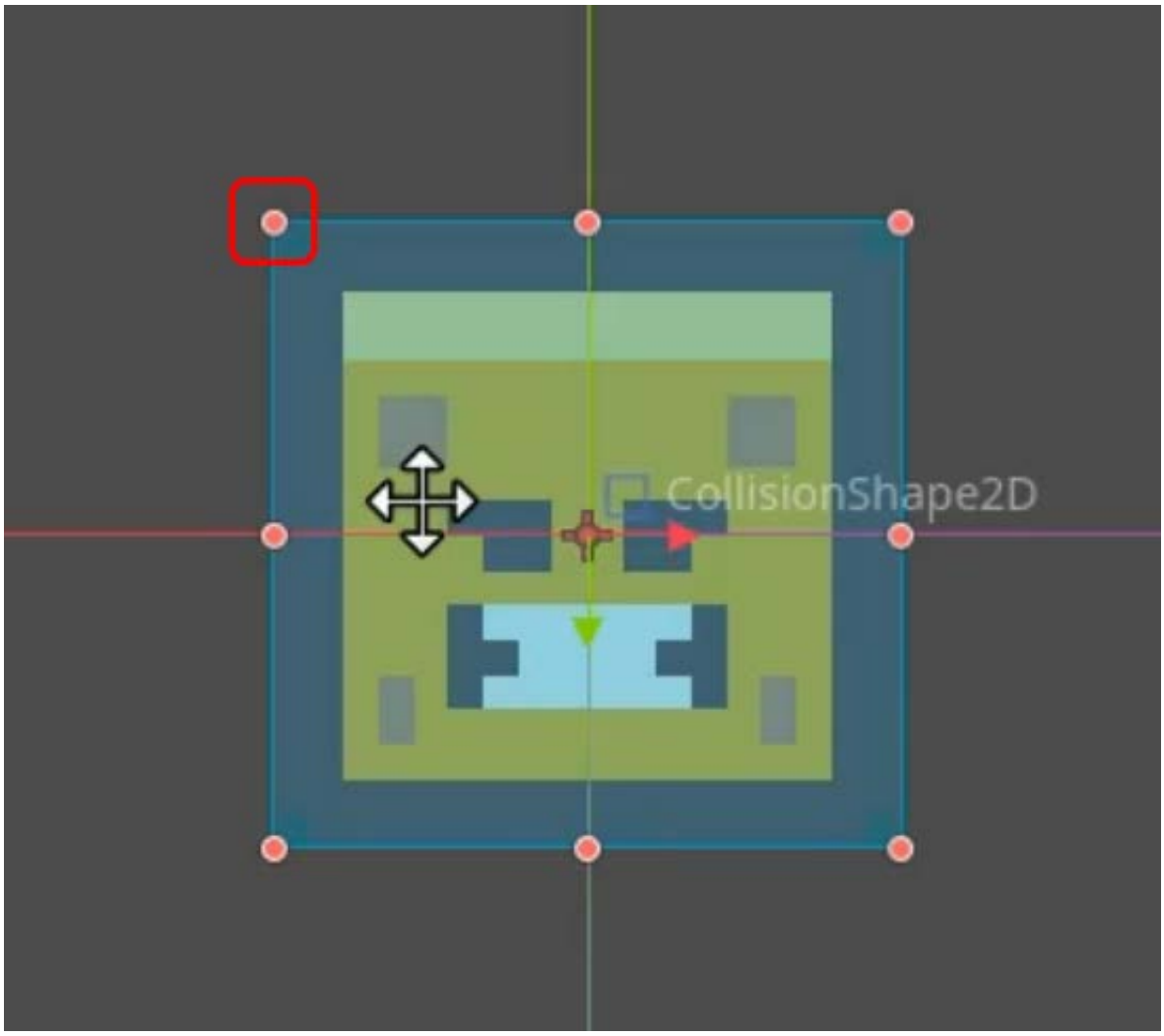
We also need a collider for our rigid body to work with the physics systems. To do this we will add a new **CollisionShape2D** node as the child node of our *Player* node.



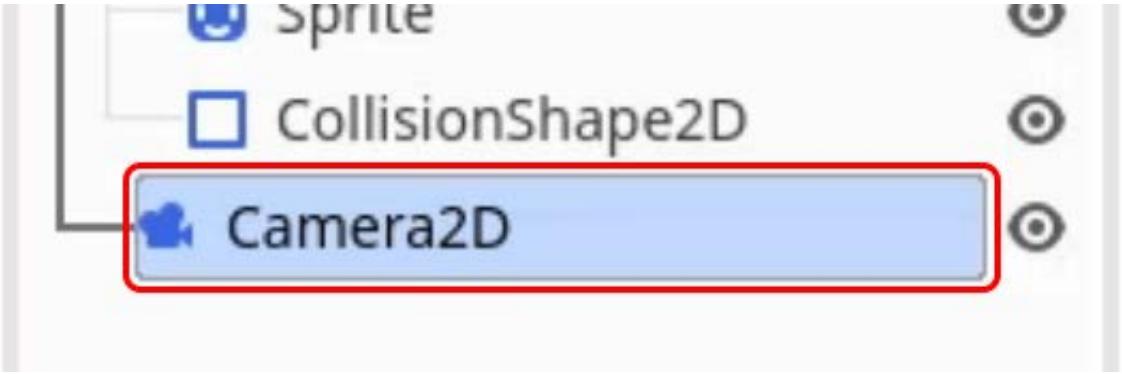
We will set the **Shape** property to a **New RectangleShape2D**.



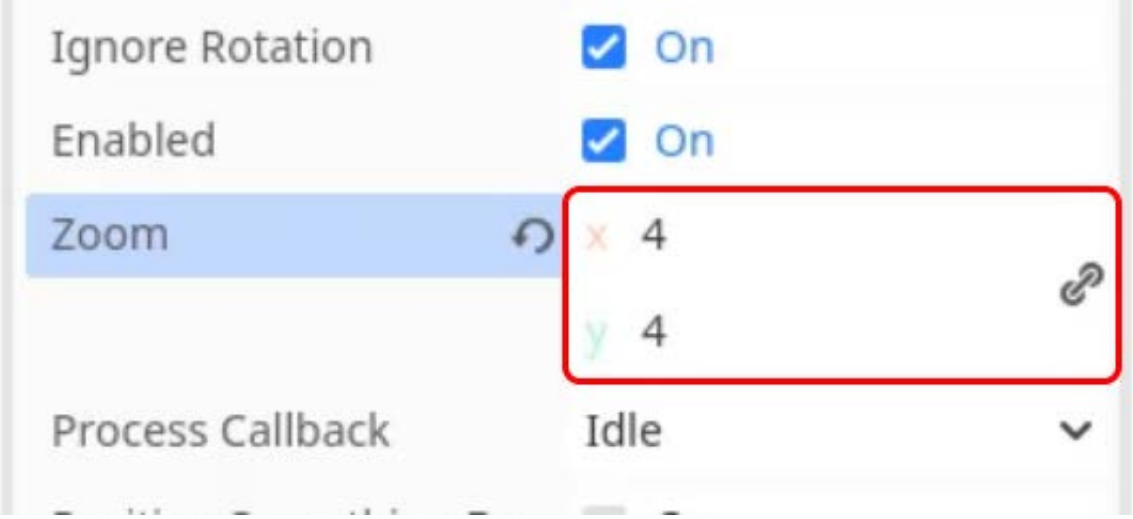
Finally, use the **orange circles** to resize the collider to the same size as our *Sprite*.



We will also add a **Camera2D** node so that we can see our *Player*.

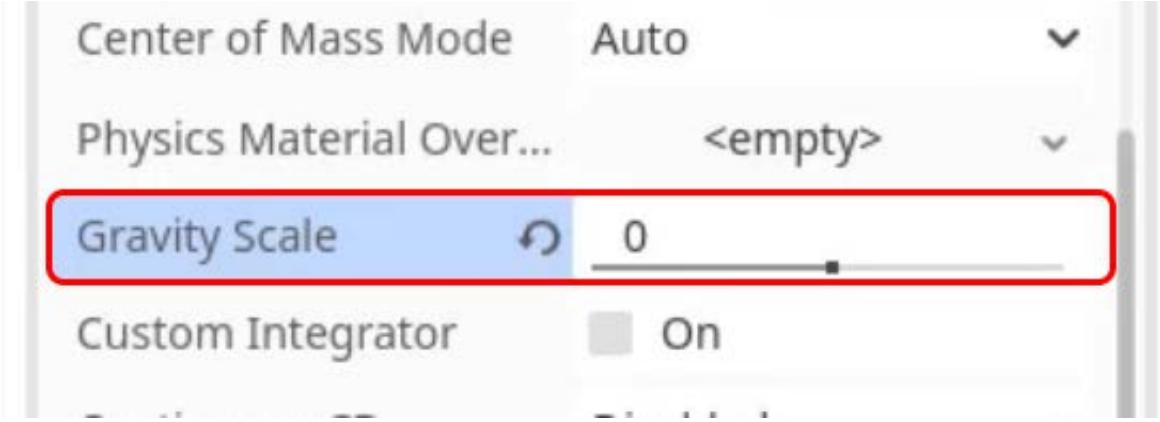


We can zoom in the camera by setting the **Zoom** property to **(4,4)**.



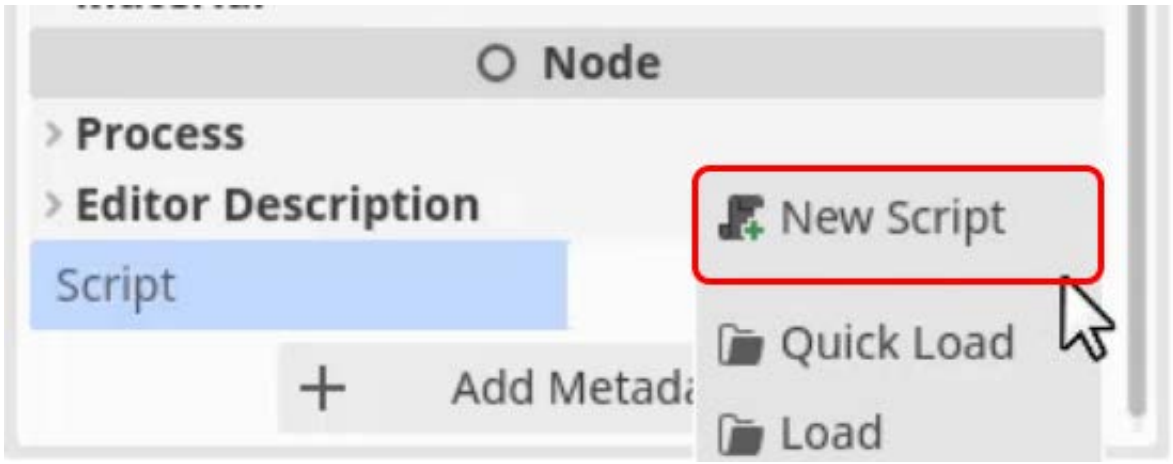
If you now press **Play** you will see our *Player* node falls due to gravity, which is the result of the *Rigidbody2D* node. In the *Inspector* we can modify lots of settings, such as the *mass* which will affect the weight of forces applied to the object. The *Gravity Scale* will determine how much gravity is applied to the node. There are lots of other settings such as *Linear* and *Angular* drag which can also be modified to change other behaviors of the rigid body.

We will change the **Gravity Scale** to a value of **0** to stop our *Player* node from being affected by gravity, as we don't want that for this game.

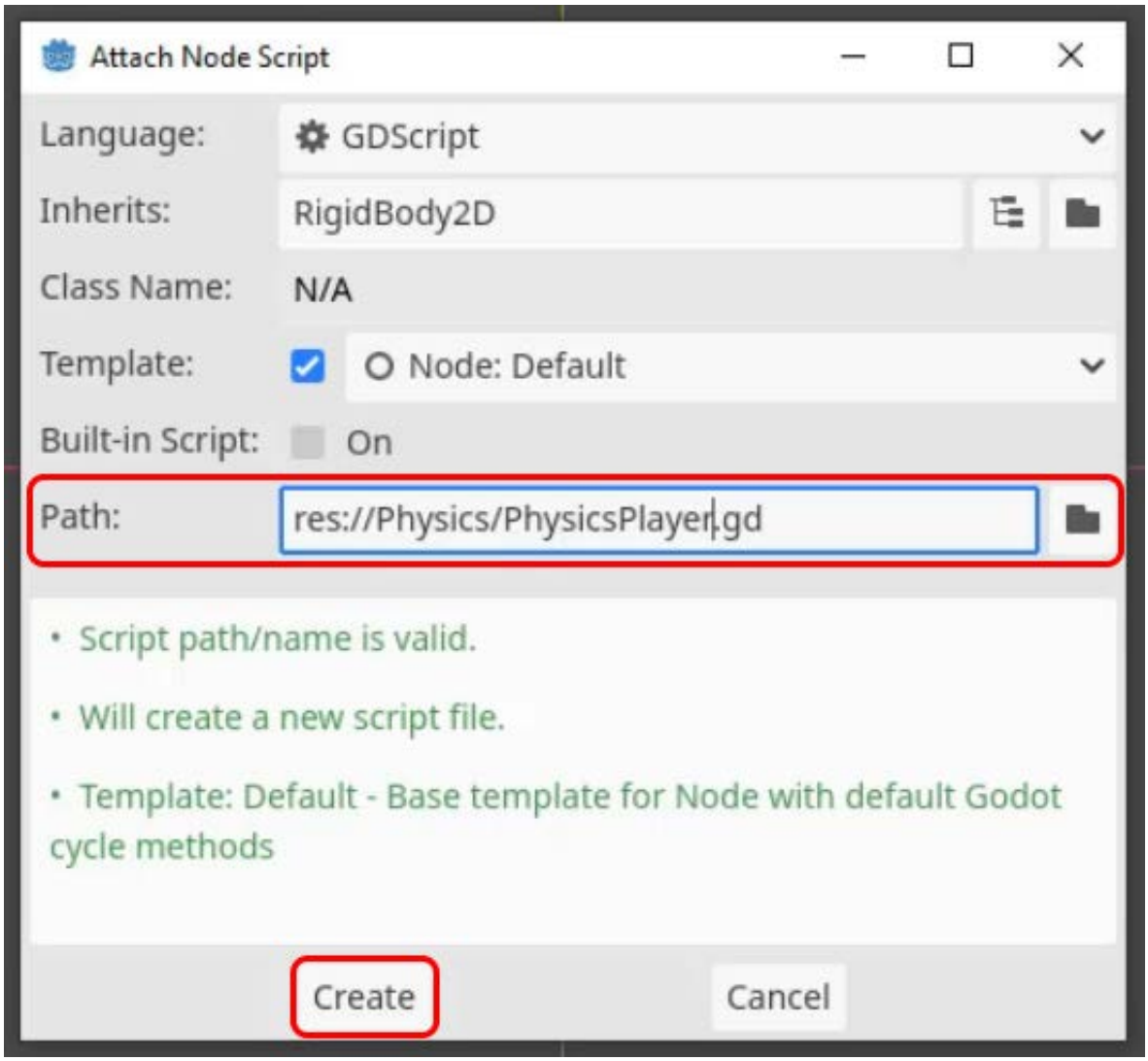


Adding Movement to the Player

The next step will be to add a **New Script** to our *Player* rigid body node.



We will name this script *PhysicsPlayer.gd* and save it in our *Physics* folder.



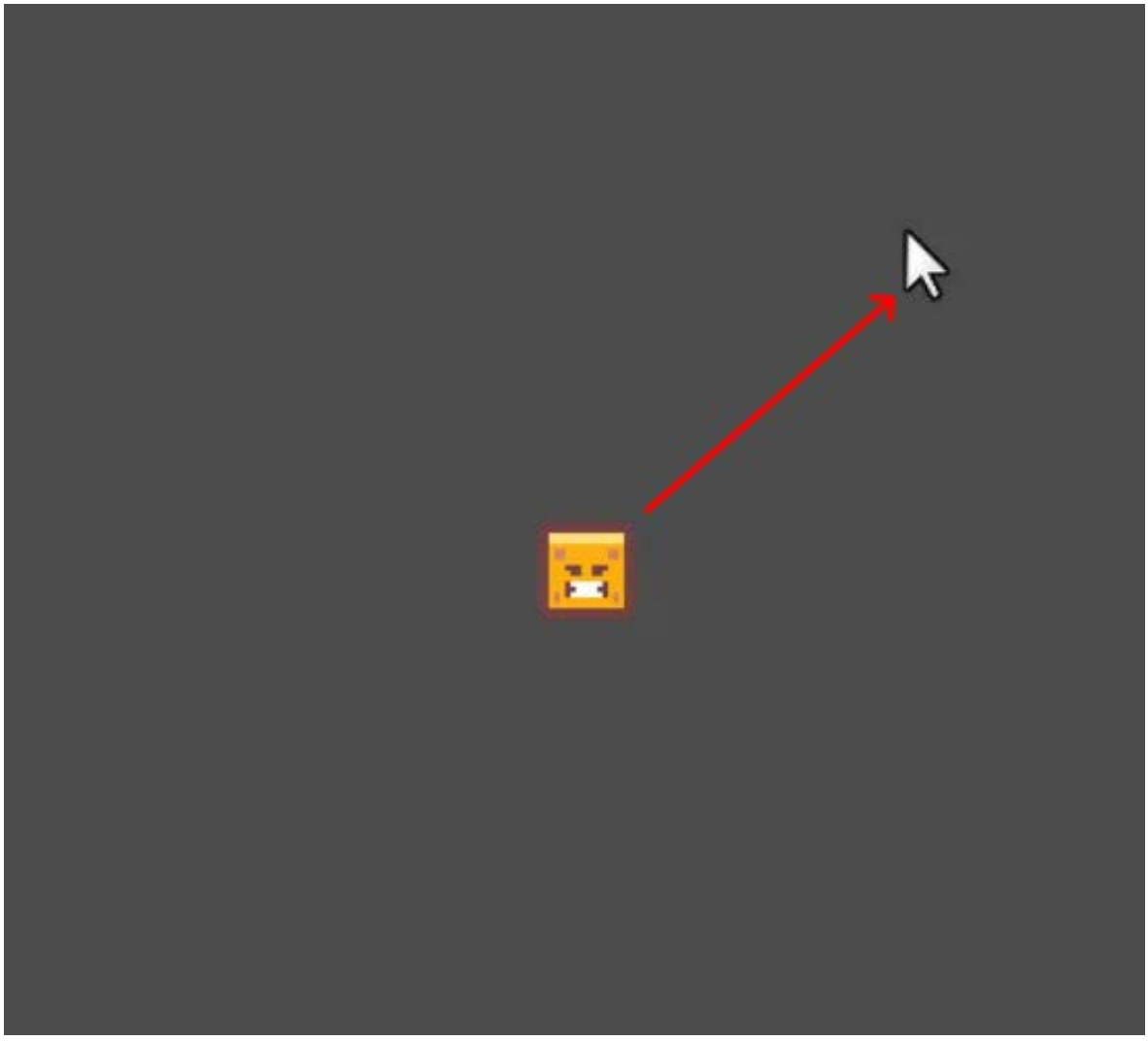
We can delete the *_ready* function as we won't be using them in this script, but we will use the *_process* function. We will start this script by creating a variable called **hit_force** using the type of **float** and set it equal to **50.0** by default.

```
var hit_force : float = 50.0
```


We will use this variable as the force applied when the player clicks on the screen. To detect if the left mouse button is pressed we will use the following if statement in the *_process* function.

```
func _process(_delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
```

From here we will get the direction we want to use. This will be the direction from the *Player* node to the mouse position when the user clicks.



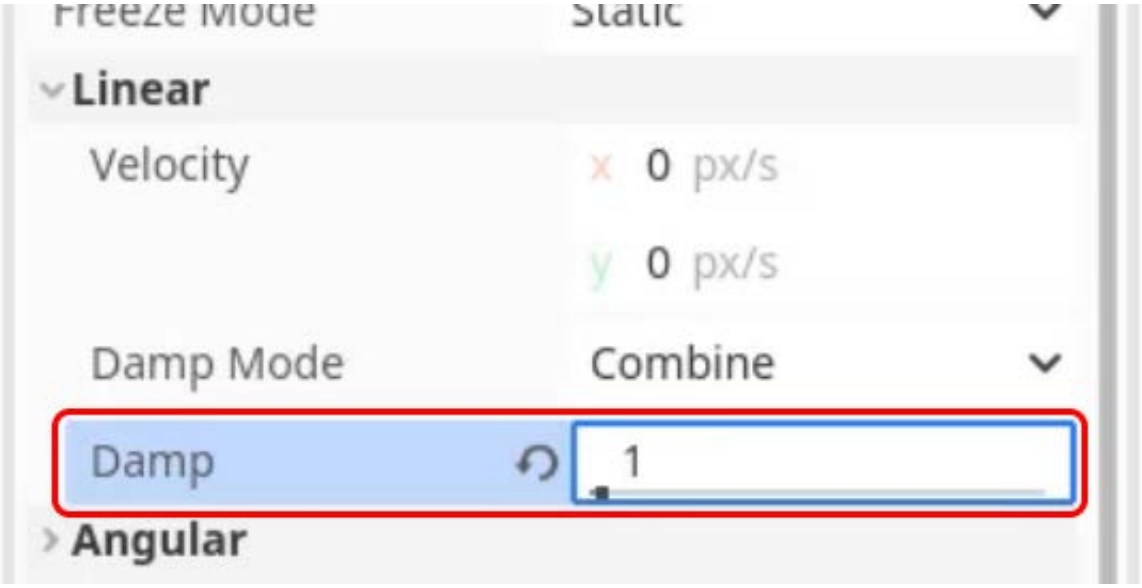
We will store this direction in a variable named *dir* and use the *direction_to* function to get the direction from our *Player's global_position* to the mouse position.

```
func _process(_delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
        var dir = global_position.direction_to(get_global_mouse_position())
```


We can then use the *apply_impulse* function to apply a force in that direction.

```
func _process(_delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
        ...
        apply_impulse(dir * hit_force)
```

Now if you **save** the script and press **Play** you will see the player moves in the direction you click. However, it is a little hard to control as the *Player* rigid body doesn't have any drag. We can increase the drag by increasing the **Damp** value under the **Linear** section in the *Inspector* tab.



You can change this to any value you like to get the feeling you want, along with changing the mass which will affect how much force is applied.

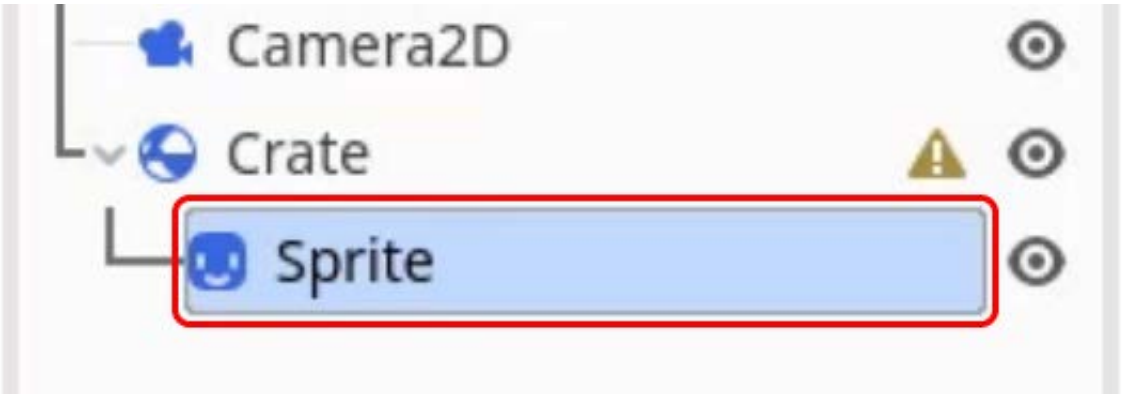
In the next lesson, we will be adding our crates to the scene which will also be physics objects to finish this mini-project.

In this lesson, we will be finishing our physics-based mini-game in Godot. We will first be looking at how to create a crate, add a collision shape, and adjust the gravity and drag.

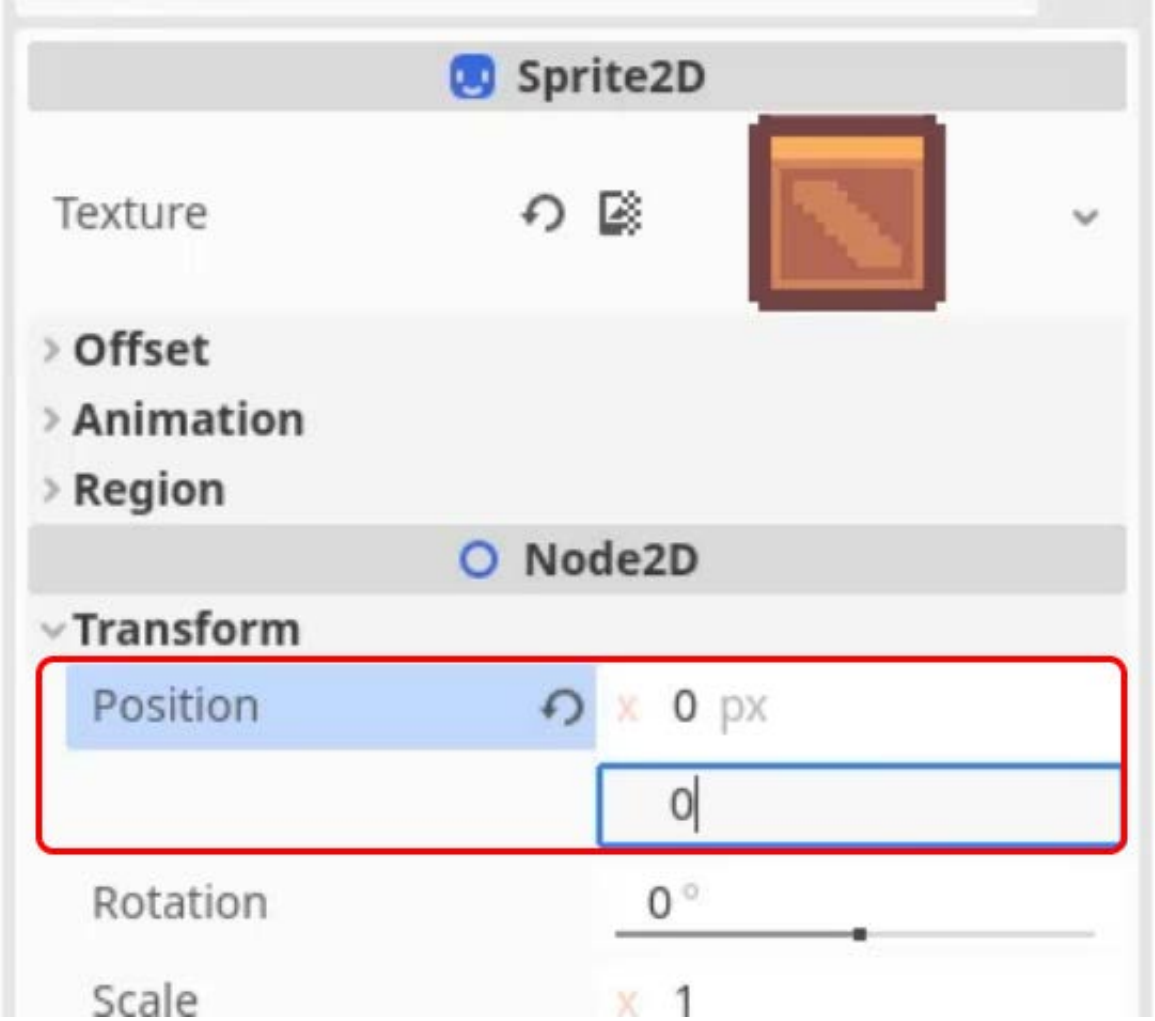
Creating the Crates

The first step is to create a new **Rigidbody2D** node which we will rename to *Crate*.

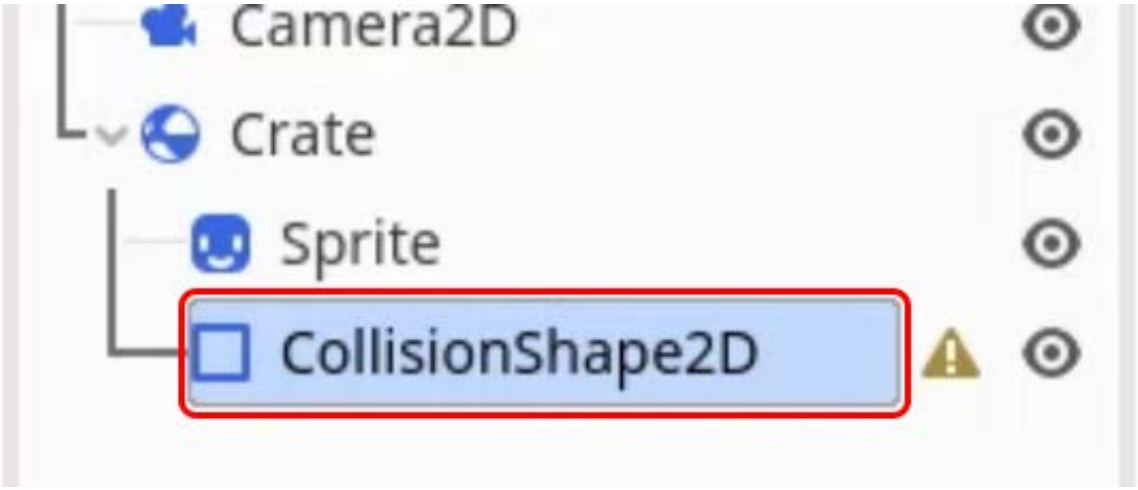
From here we will use the **Crate.png** asset as a Sprite for the *Crate* node tree.



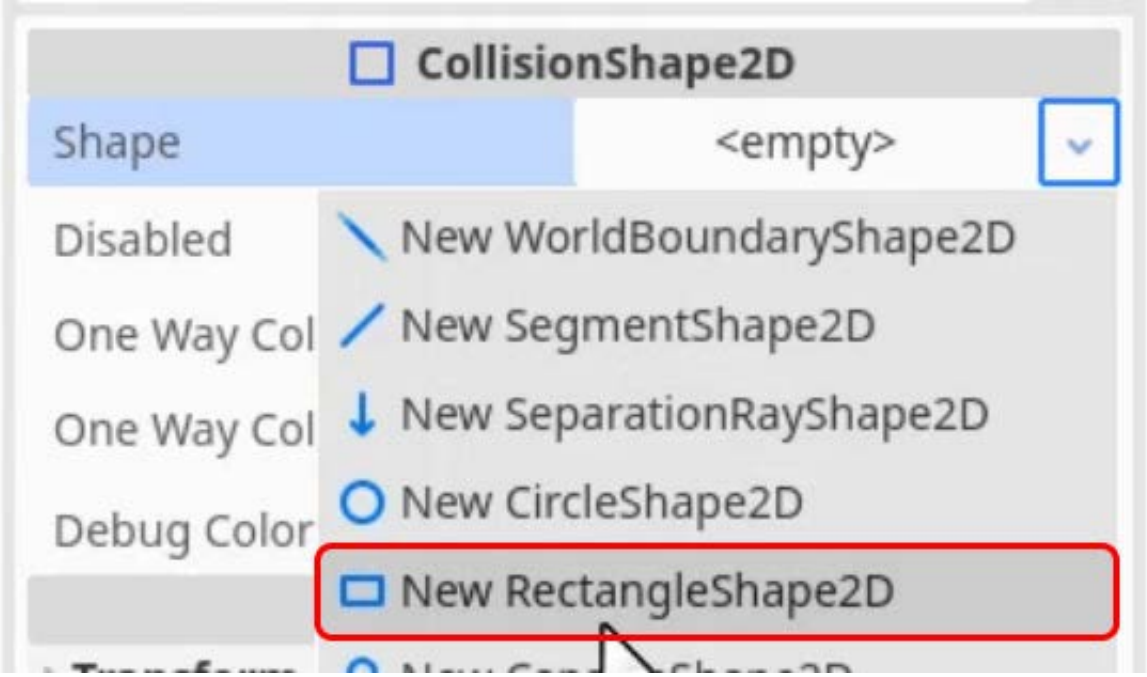
Remember to set the **Position** property of the new *Sprite* to **(0,0)** to center the node.



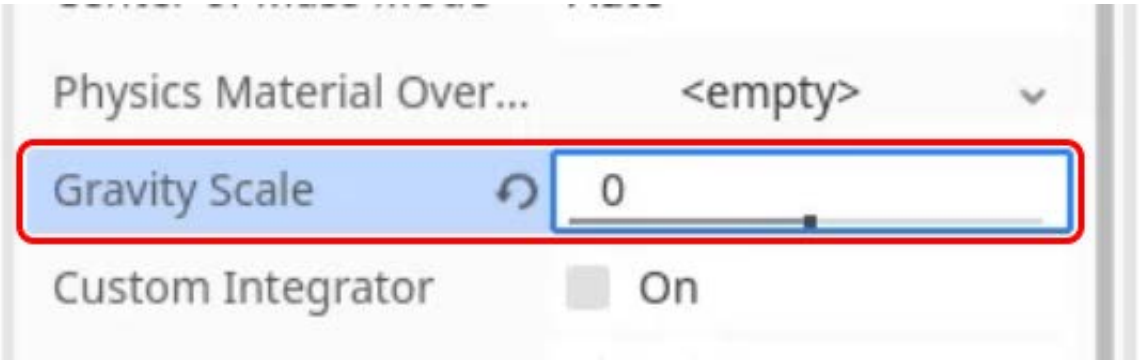
Next, we need to add a **CollisionShape2D** as a child node.



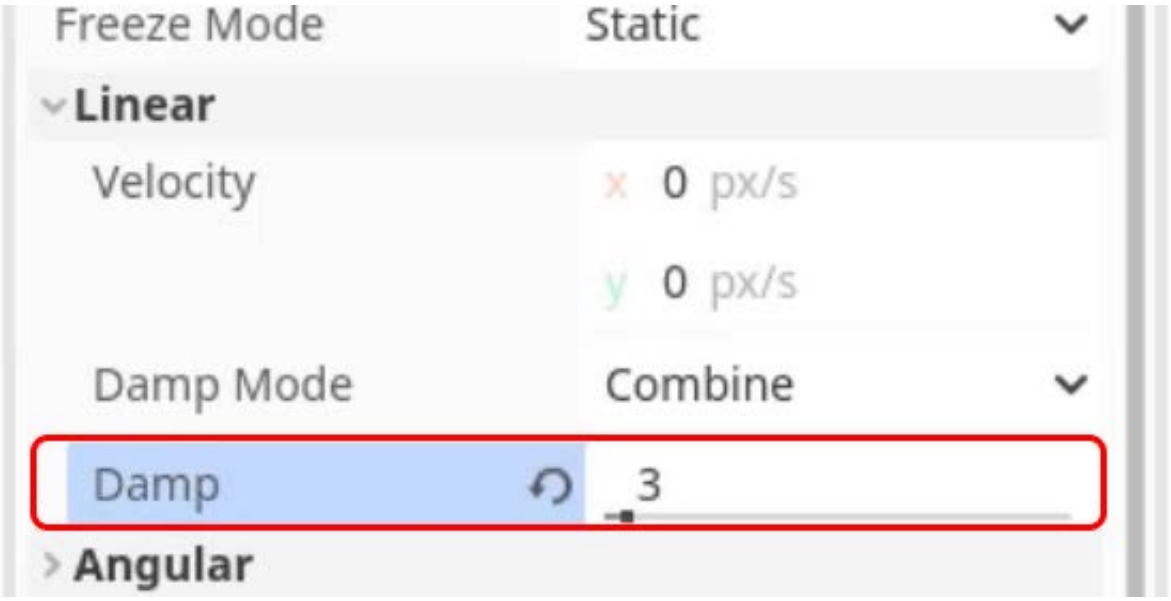
We can set the **Shape** property to a **New RectangleShape2D** and adjust the bounds to fit the crate.



Finally, we want to set the **Gravity Scale** to **0** to stop it from falling to the ground.

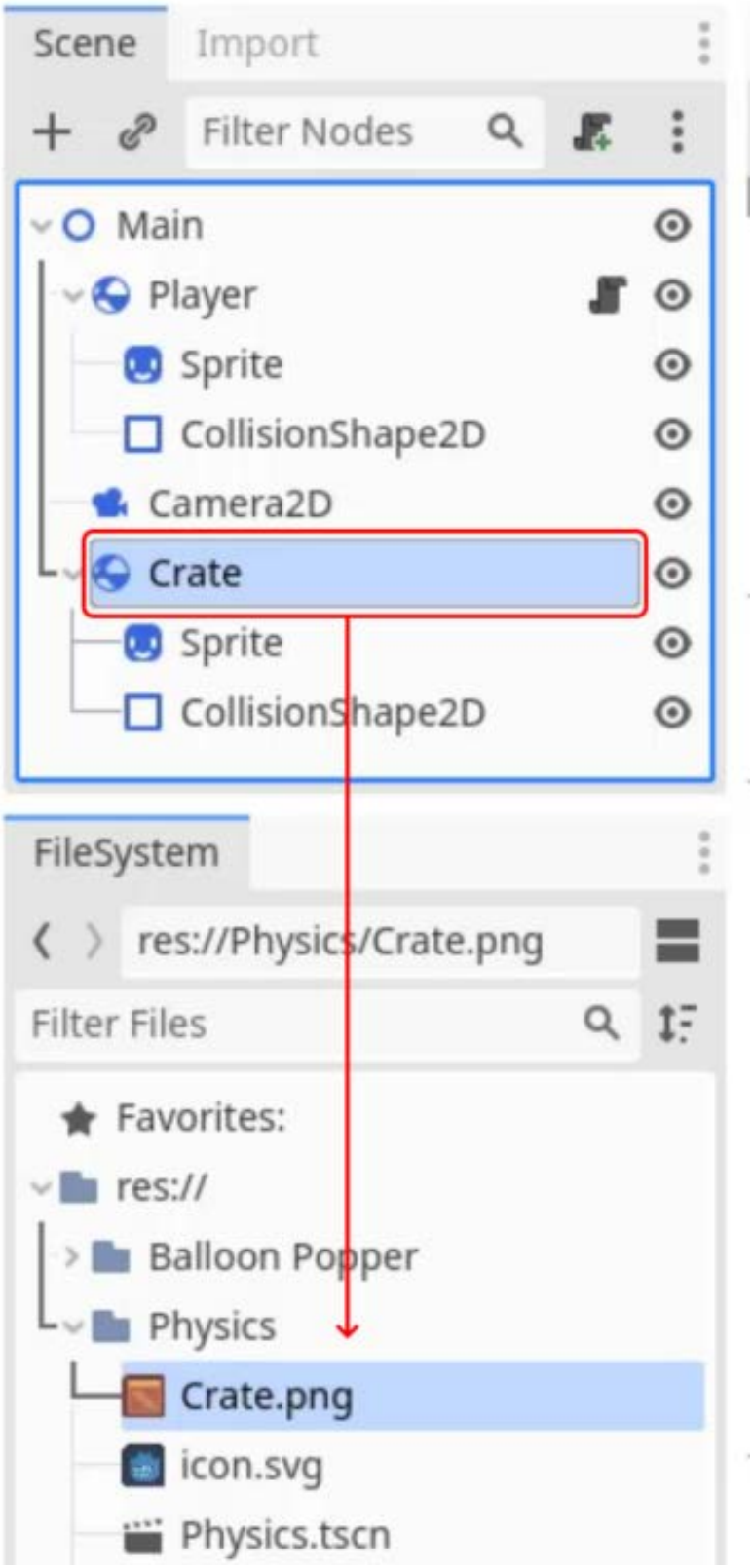


If you now press **Play** you will see you can move towards the crate and hit it to move it around. We will want to add some drag to this so that the crate doesn't move around forever, we will use a value of 3 here, but feel free to change this to your liking.

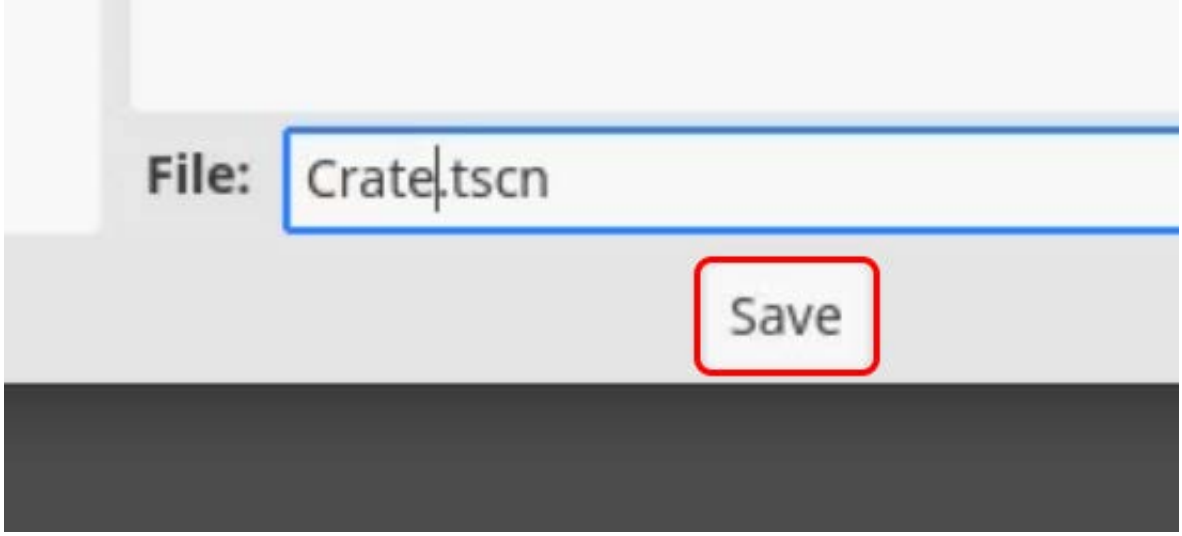


Creating Multiple Crates

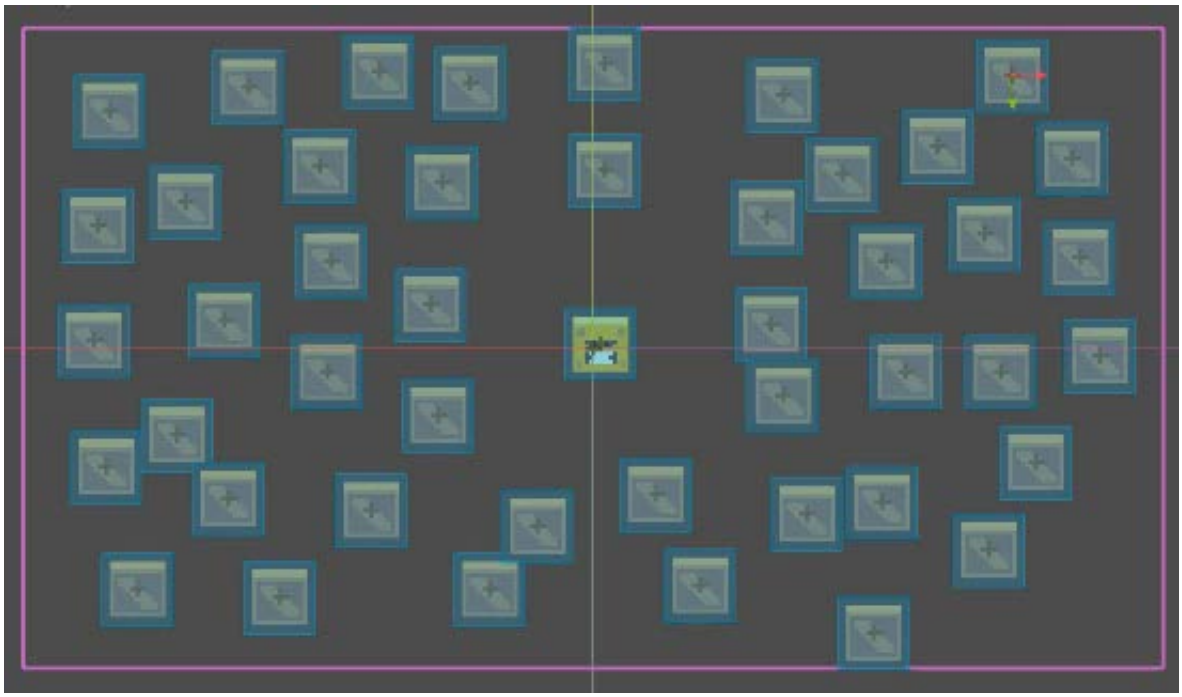
Finally, we can save the crate as a scene and duplicate it to create multiple crates. We will do this by dragging it from the *Scene* to the *FileSystem*.



We will then save the scene with the name *Crate.tscn*.



We can then **duplicate** (**CTRL+D**) to create lots of instances of the crate around the scene.



Now we can press **Play** and move the player around to interact with all of the crates in the scene.

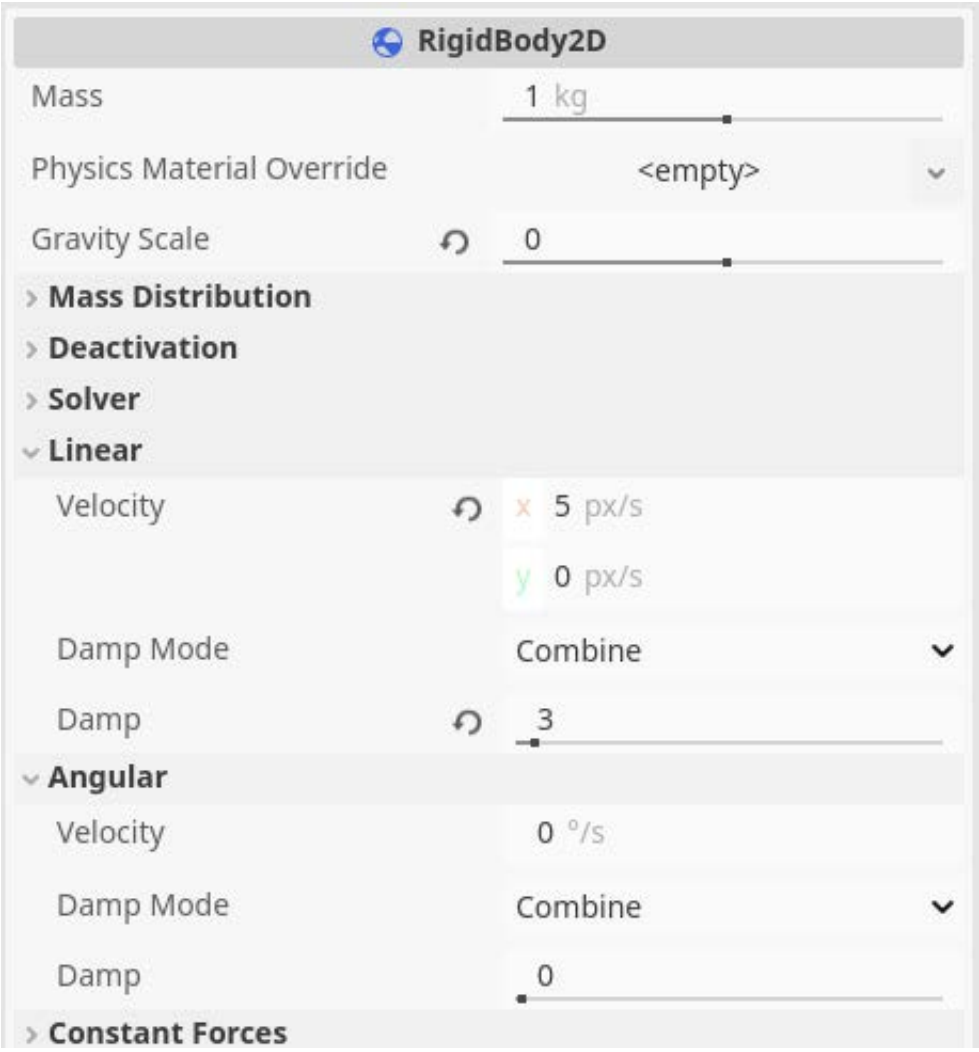
This was our first look at Godot's physics system. So let's have a more in-depth look at what's actually going on, as well as getting a better understanding of our script.

RigidBody2D

The **RigidBody2D** node is one that requires a CollisionShape2D. This is because the rigid body is affected by physics – gravity, collisions, drag, etc.

You'll notice that there are many different properties in the Inspector for the rigid body, so let's go over a few of the most important:

- **Mass** – How heavy the object is in kilograms. The heavier it is, the more force required to move it.
- **Physics Material Override** – This is a resource where you can define levels of bounciness, friction, etc for the object.
- **Gravity Scale** – How strongly gravity will pull down the object.
- **Linear Velocity** – The velocity that will be applied each frame.
- **Linear Damp** – This acts as drag applied to the velocity, think air resistance.
- **Angular Velocity** – The velocity but for rotation.
- **Angular Damp** – Drag applied to the angular velocity.



PhysicsPlayer Script

Inside of this script, we are checking each frame for the left mouse button. When pressed, we are applying a force to the player in the direction of our mouse button. Let's have a look at these different elements and see exactly what they are doing:

- **Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT)**
  - This returns true or false for if the left mouse was pressed down this frame.
- **global_position**
  - This accesses our node's global position vector2.
- **get_global_mouse_position()**
  - This returns the vector2 position of our mouse.
- **a.direction_to(b)**
  - This calculates the vector2 direction between a and b. Think drawing a line from one point on a grid to another.
- **apply_impulse(vector)**
  - This applies an impulse force to the RigidBody2D node. Think hitting a golf ball with a club – one instantaneous flash of velocity.

```
func _process(_delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
        var dir = global_position.direction_to(get_global_mouse_position())
        apply_impulse(dir * hit_force)
```

Additional Resources

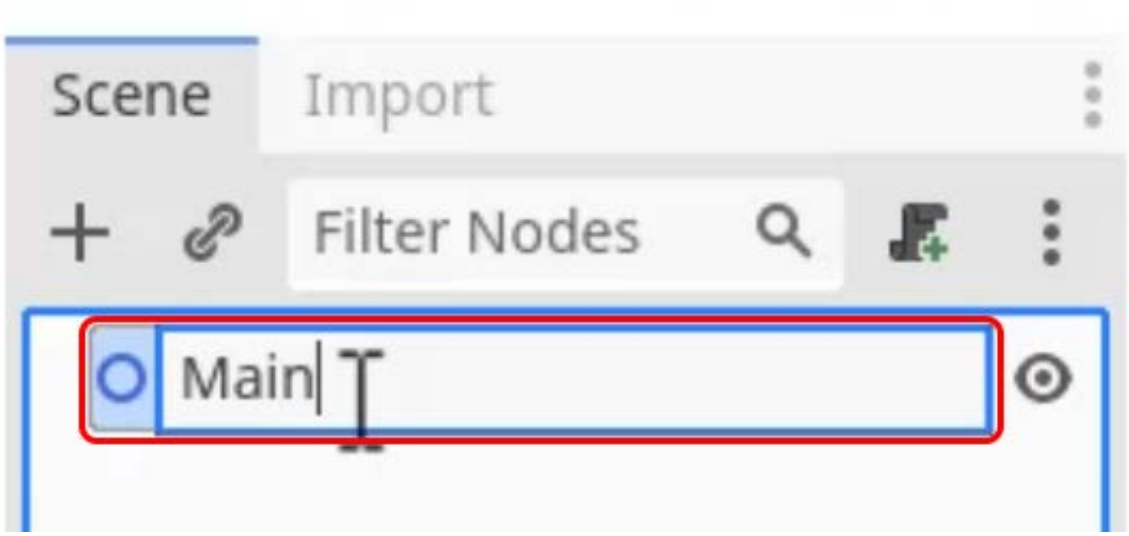
If you wish to learn more, then you can refer to the Godot documentation.

- RigidBody2D
- Physics in Godot
- Vector2.direction_to
- global_position
- is_mouse_button_pressed

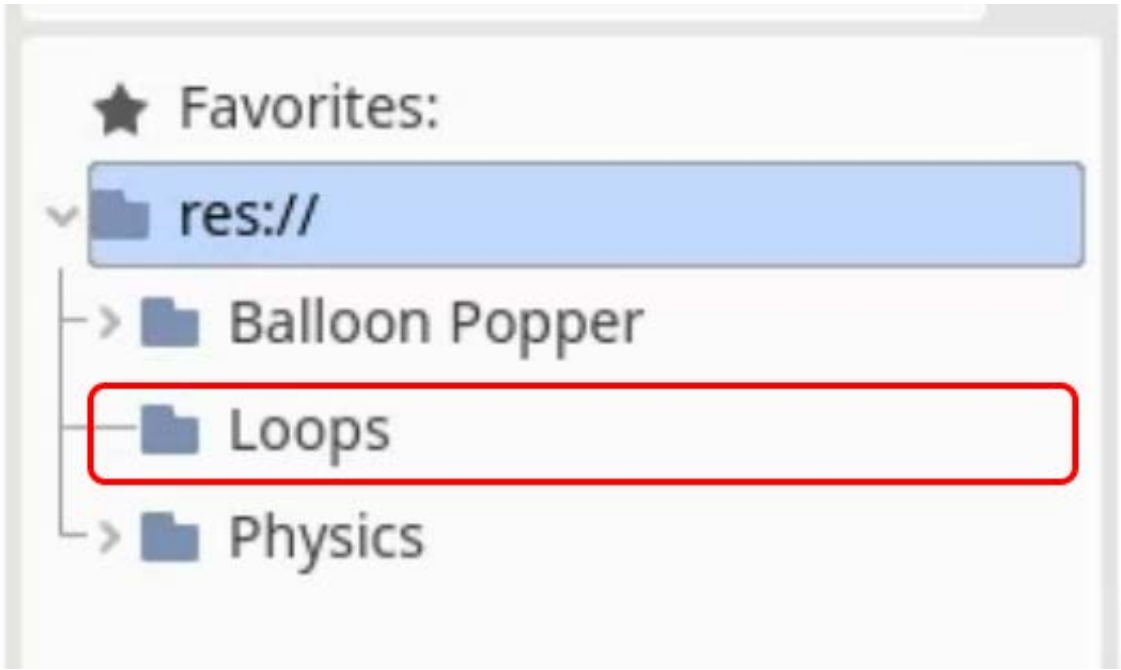
In this lesson, we are going to be creating our third mini-project, which is based on learning to use loops to create a randomly generated star system.

Creating the Scene

As with our previous mini-projects, we will need to create a **new scene** for this project. For this project, we will use a **2D** root node, named *Main*.



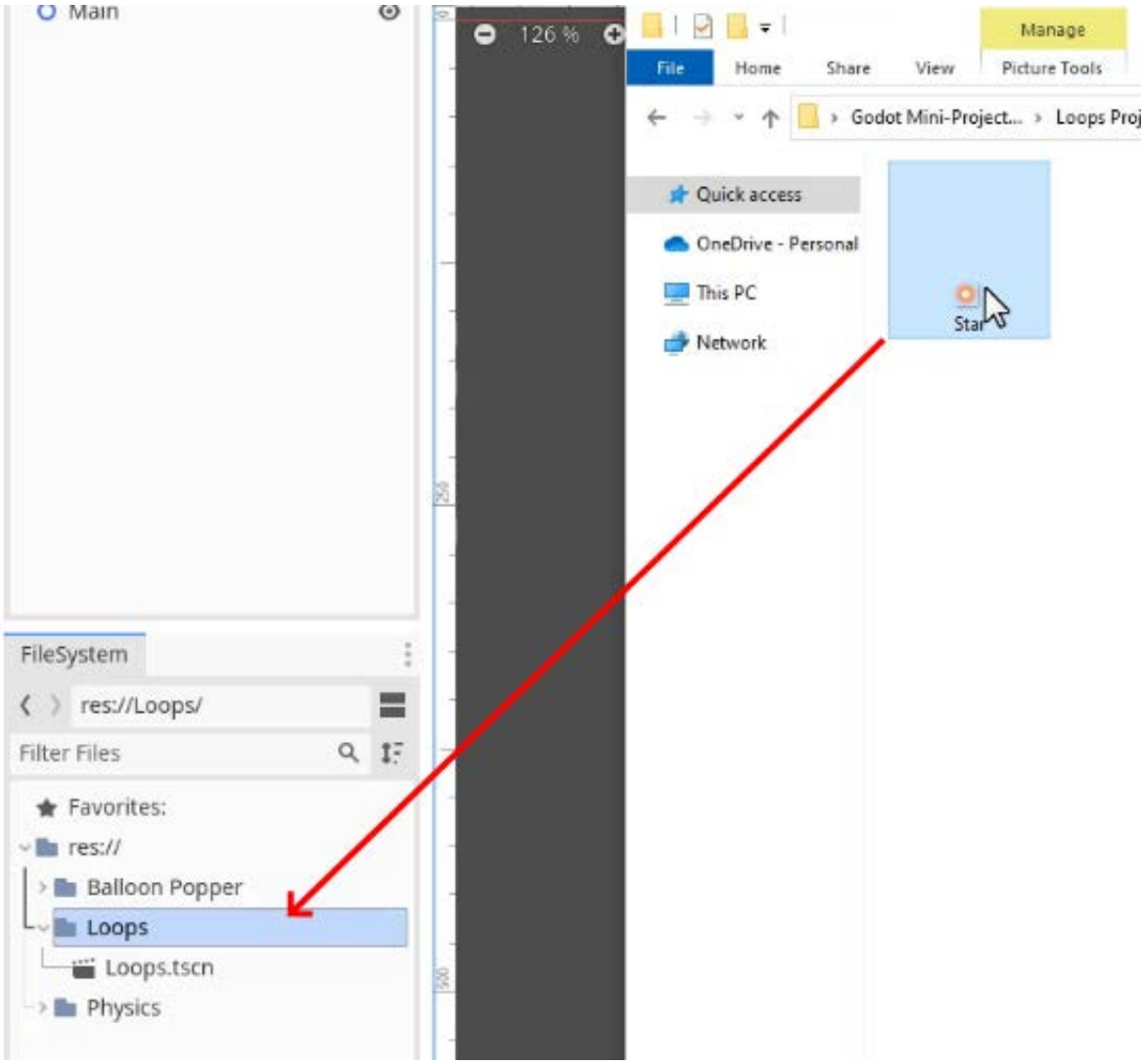
We will also create a new folder in the *FileSystem* named *Loops* for storing the files related to this project.



We can then save this as *Loops.tscn* inside the new folder.

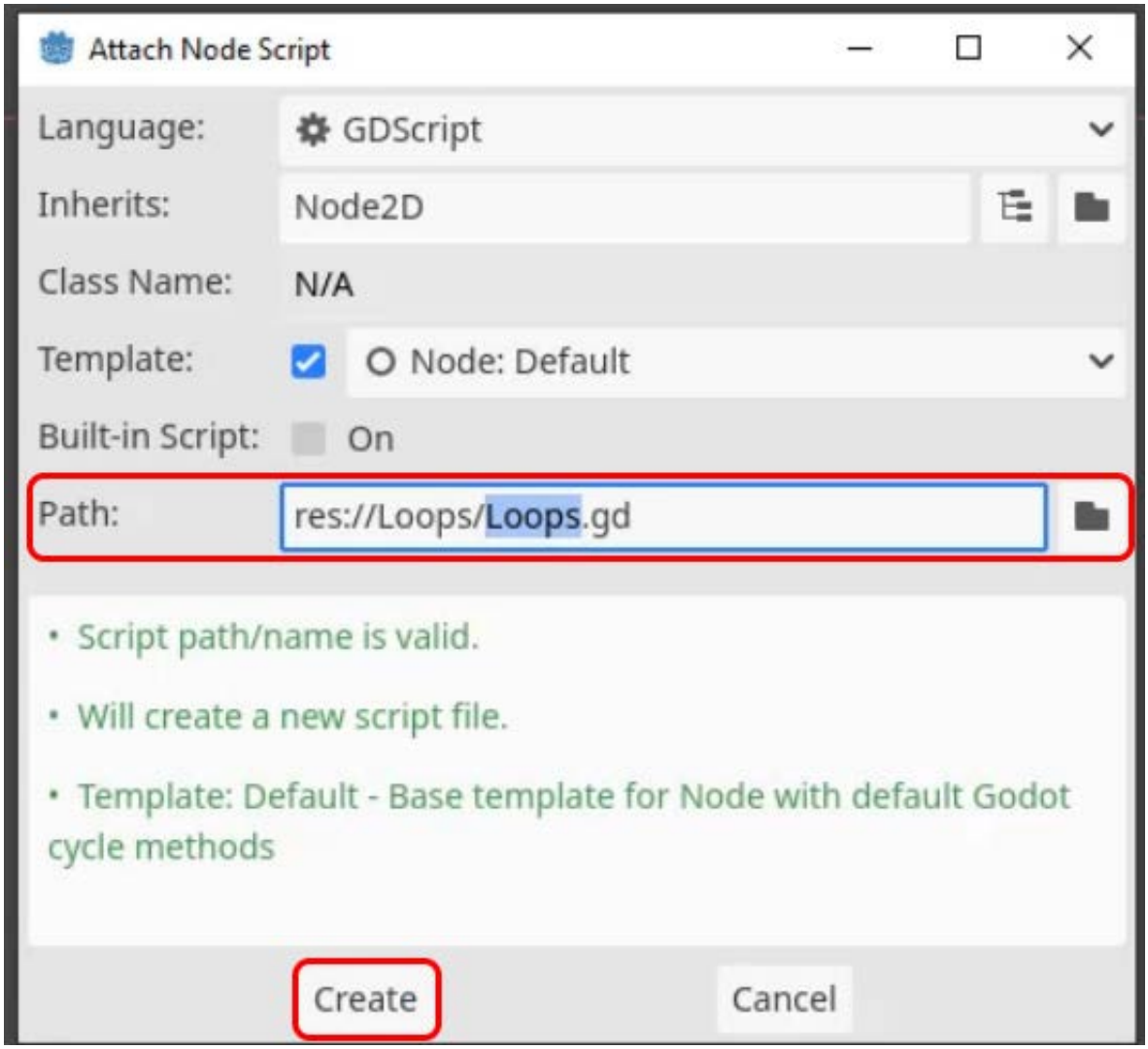
Adding our Assets

In the extracted asset files that we gathered in the previous project, there is also a folder named *Loops Project*. This contains the *Star.png* sprite that we will be using for our project. As with the previous project, feel free to use a different asset than the ones provided in the course files. You can then drag the *Star.png* file into the *Loops* folder that we created in Godot.



What are Loops?

To begin working with loops, we will create a **New Script** on our *Main* root node. We will call this *Loops.gd* and save this in the *Loops* folder.



We won't be using the *_process* function in this lesson, so that can be deleted, but make sure to keep the *_ready* function for making our loops in.

So, what is a loop? A loop is a way of running a block of code multiple times. For example, if we want to print the word *Hello!* to the *Output* ten times, we could write the *print* line ten times but this isn't very effective. Instead, we can use a *for* loop as shown in the code below.

```
func _ready():
    for i in 10:
        print("Hello!")
```

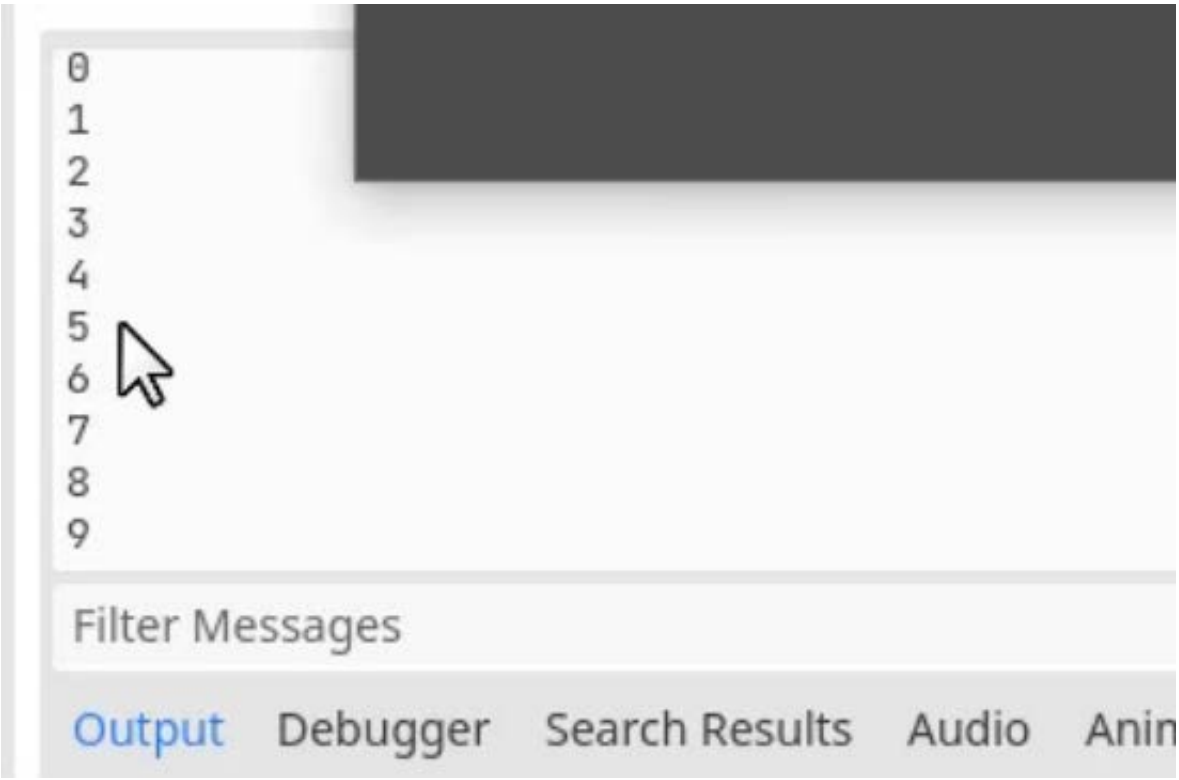
If you press **Play** you will see this writes the *Hello!* output to the console ten times, as intended. What we are doing here is creating a temporary variable named *i* (for iteration) and then the *for* loop will run the block of code inside, incrementing *i* by 1 each time, until it reaches the maximum of 10 that we set. We can see this by replacing *"Hello!"* with our *i* variable in the print statement.

```
func _ready():
```



```
for i in 10:
    print(i)
```

Now if you press **Play** you will see 0-9 printed in the *Output* window, as expected



You could change 10 to any number you like, such as 1000. If you did this, you will see the numbers 0-999 printed in the *Output* window. It is important to note that our *for loop* will run in one frame and will block any other code from running until it is complete, much like an *if statement* when it runs.

Challenge

As a challenge before the next lesson, create a variable named **score** with a value of **0** and create a for loop to increase the score by 5 every iteration. And then print the final value of *score* after the for loop. To get you started, here is the code to create the *score* variable.

```
func _ready():
    var score = 0
```


In this lesson, we are going to be creating the actual star-system scene.

Challenge Solution

To begin with, however, we set a challenge in the last lesson, to create a variable called score, which increases by five 10 times, and then print out the results. To do this, we will create a variable called **score** and set it to **0** in the *_ready* function.

```
func _ready():
    var score = 0
```

We will then create a *for loop* which will loop 10 times and increase the score by 5 each time.

```
func _ready():
    ...

    for i in 10:
        score += 5
```

Finally, we will print out the final value of *score*.

```
func _ready():
    ...

    print(score)
```

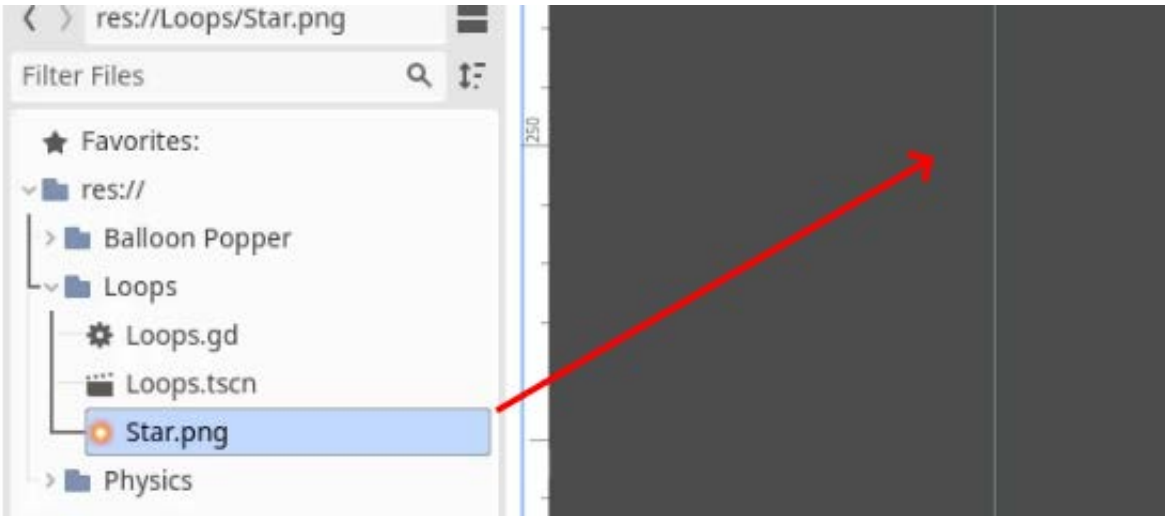
If you now press **Play** to run the script, you will see a result of 50 in the *Output* window.

It is now time to begin creating the star system, to do this we will remove the code we added in our *Loops.gd* script, leaving us with only the following code.

```
extends Node2D

func _ready():
    pass
```

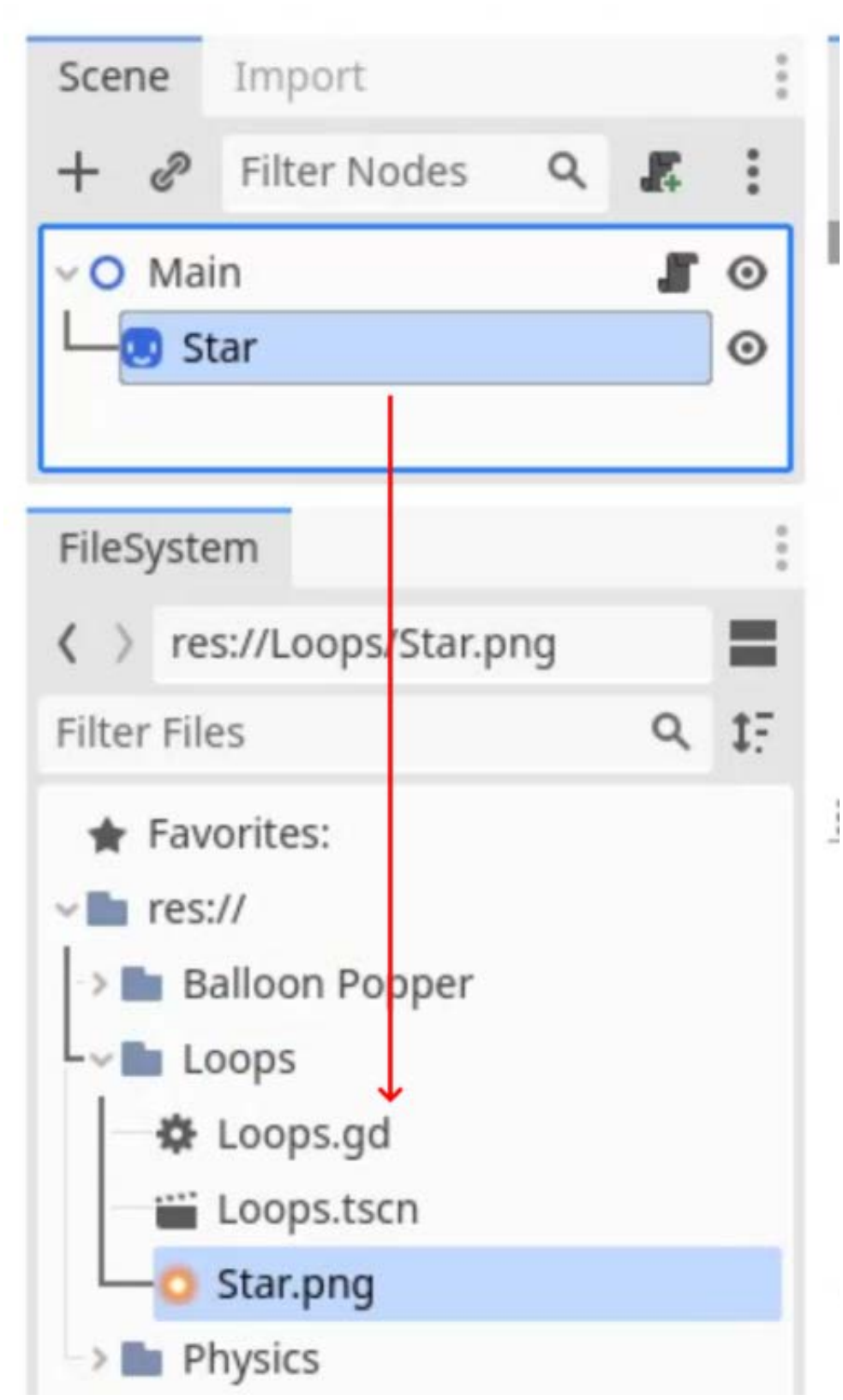
Scene Setup

Before writing the code we first need a scene to create from our script. To do this we will drag the **Star.png** asset into the viewport to create a new sprite.

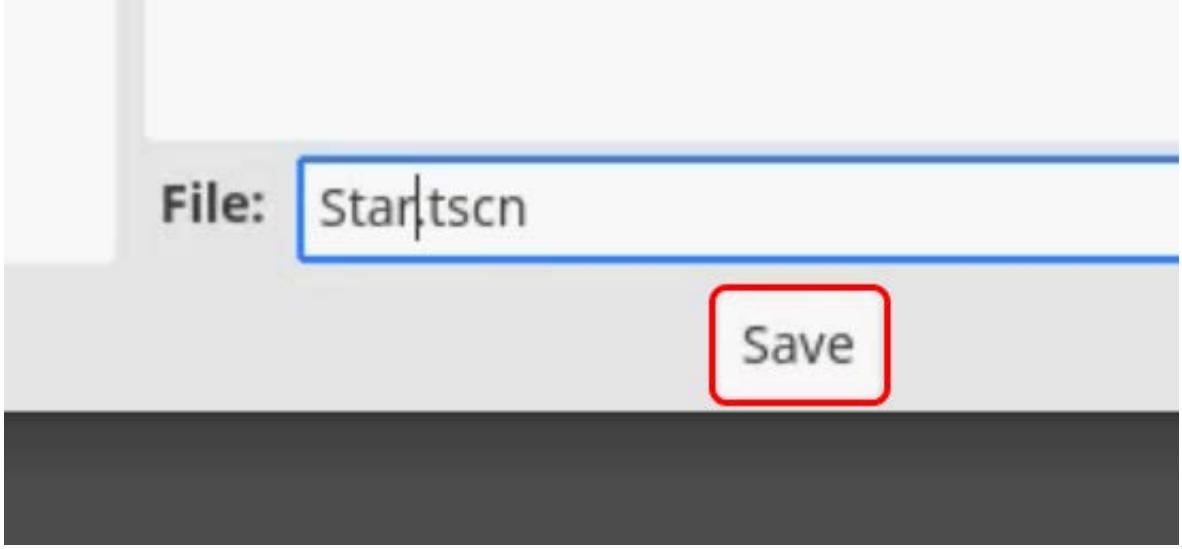


Make sure that the sprite is named *Star* and that the **Position** property is set to **(0,0)**.

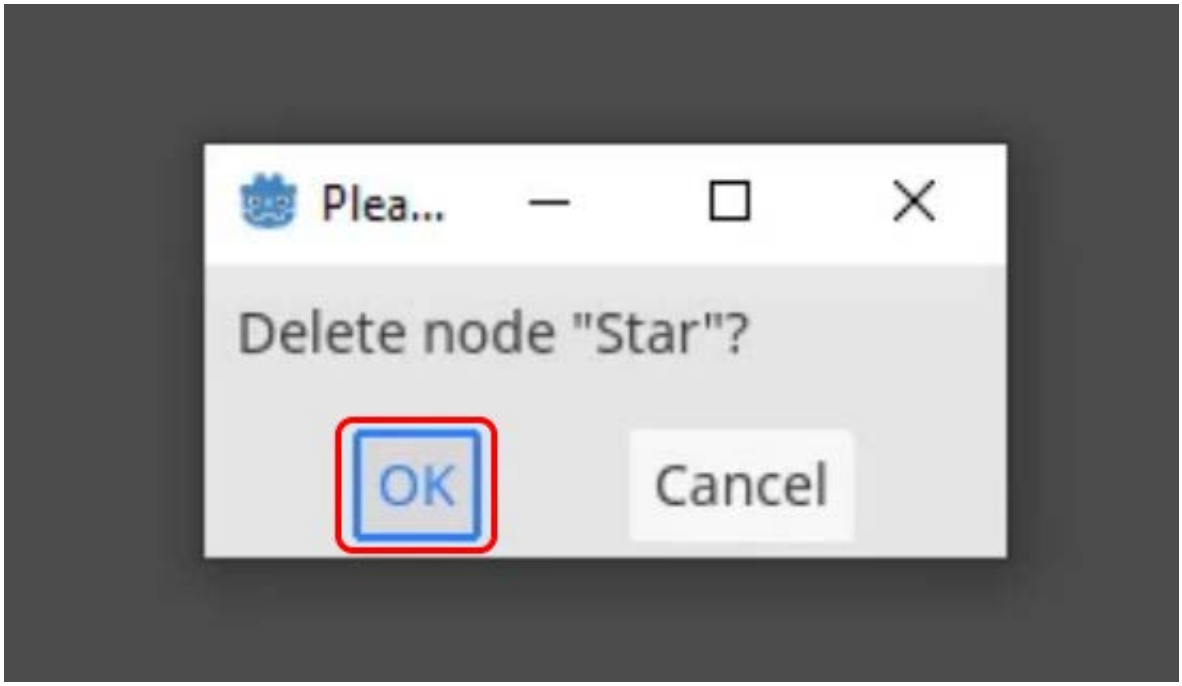
Finally, drag the node into the *Loops* folder to turn it into a scene.



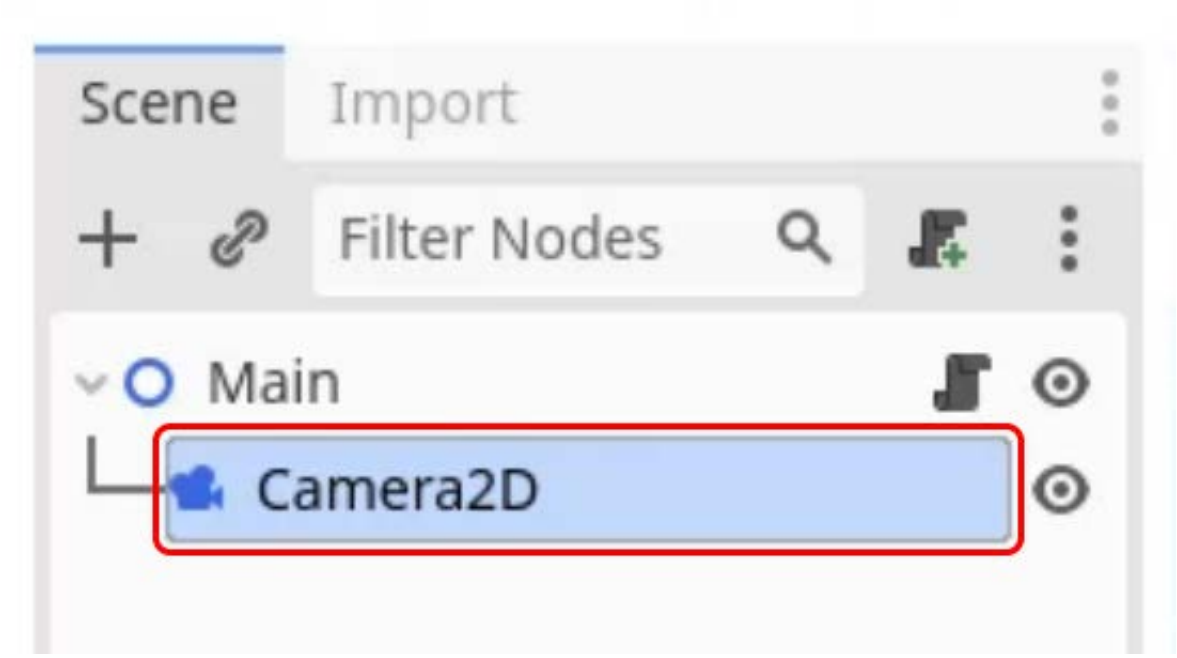
We will name the scene *Star.tscn* and press *Save*.



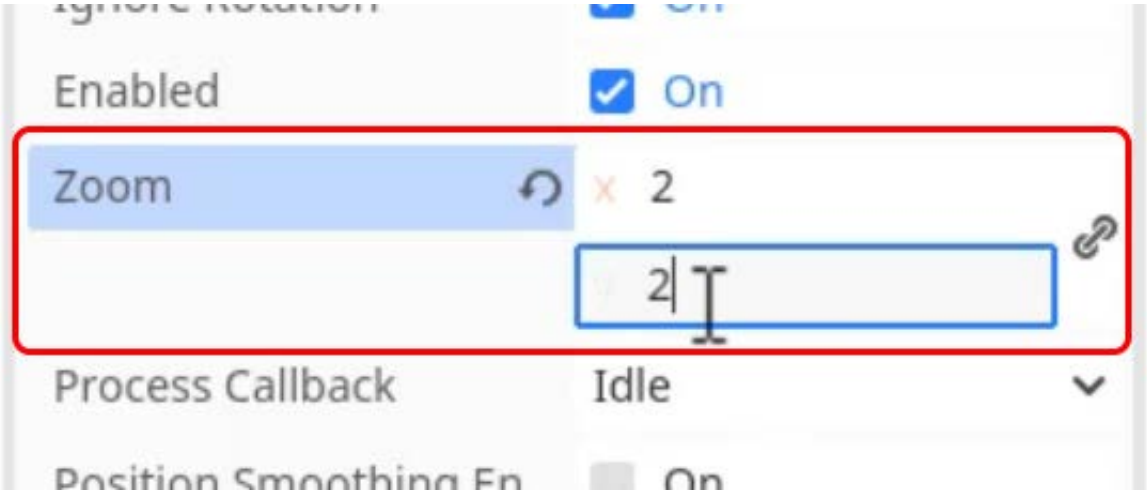
Now that we have created the *Star* scene we can **delete** the node in the viewport.



We will also add a **Camera2D** node to the scene so that we can see the starfield that is generated.



We will also set the **Zoom** value to **(2,2)** inside the *Inspector*.



Generating our Stars

We will now return to our *Loops.gd* script and create two variables. The first will be called **spawn_count**, it will be an **int** and be set to **200** by default, we will also add a *@export* tag in front so that we can edit the value from the inspector. Our second variable will store our *Star* scene, we will use the *preload* function with the path to our *Star.tscn* file to get this.

```
@export var spawn_count : int = 200
var star_scene = preload("res://Loops/Star.tscn")
```

Next, inside the *_ready* function we will create a *for loop* that will use the *spawn_count* variable as the number of iterations. Inside this *for loop*, we will *instantiate* the *Star* scene. *Instantiate* in game engines is another word for spawning an object in the scene, such as a *scene* in this case.

```
func _ready():
```



```
    for i in spawn_count:
        var star = star_scene.instantiate()
```

We then need to add it to our scene tree, as currently it isn't attached as a child of the *root* node.

```
func _ready():
    for i in spawn_count:
        ...

        add_child(star)
```

For this lesson, we will be using the following values as the boundaries of our starfield:

- **X:** -280 to 280
- **Y:** -150 to 150

You can find these values by taking an instance of the *Star* scene and taking it to the furthest points visible to the *Camera2D* node (shown by the pink square in the viewport). This will then tell you the furthest possible positions on each axis that we want to spawn our *Star* scenes at.

To set a random position we will use the *randi_range* function on each axis of the *star.position* property. We will set the range in the *randi_range* function to the values we have found above.

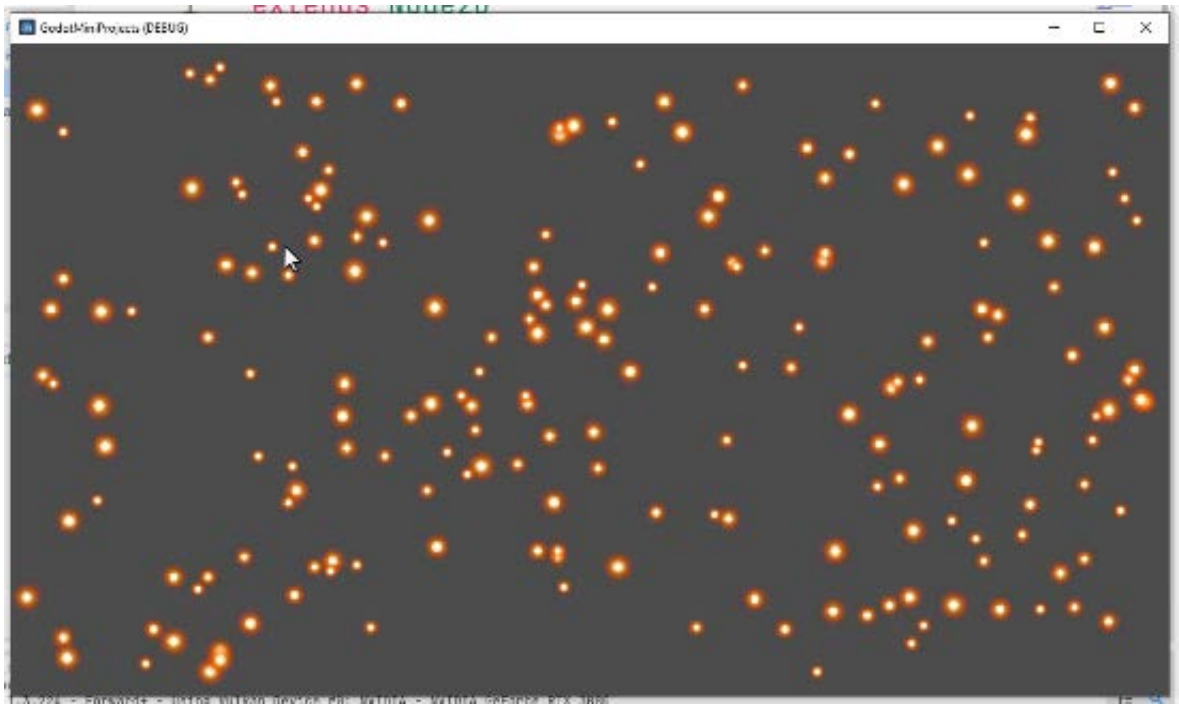
```
func _ready():
    for i in spawn_count:
        ...

        star.position.x = randi_range(-280, 280)
        star.position.y = randi_range(-150, 150)
```

Now if you **save** the script and press the **Play** button you will see the *Star* scene instantiated across the game scene in random positions.

To make this look more natural we will add a random size to the stars to give some depth. We will do this by generating a new *star_size* variable using the *randf_range* function. We use *randf* instead of *randi* here because we want a float instead of an integer. We will then assign the *star_size* variable to each axis on the *star.scale* property.

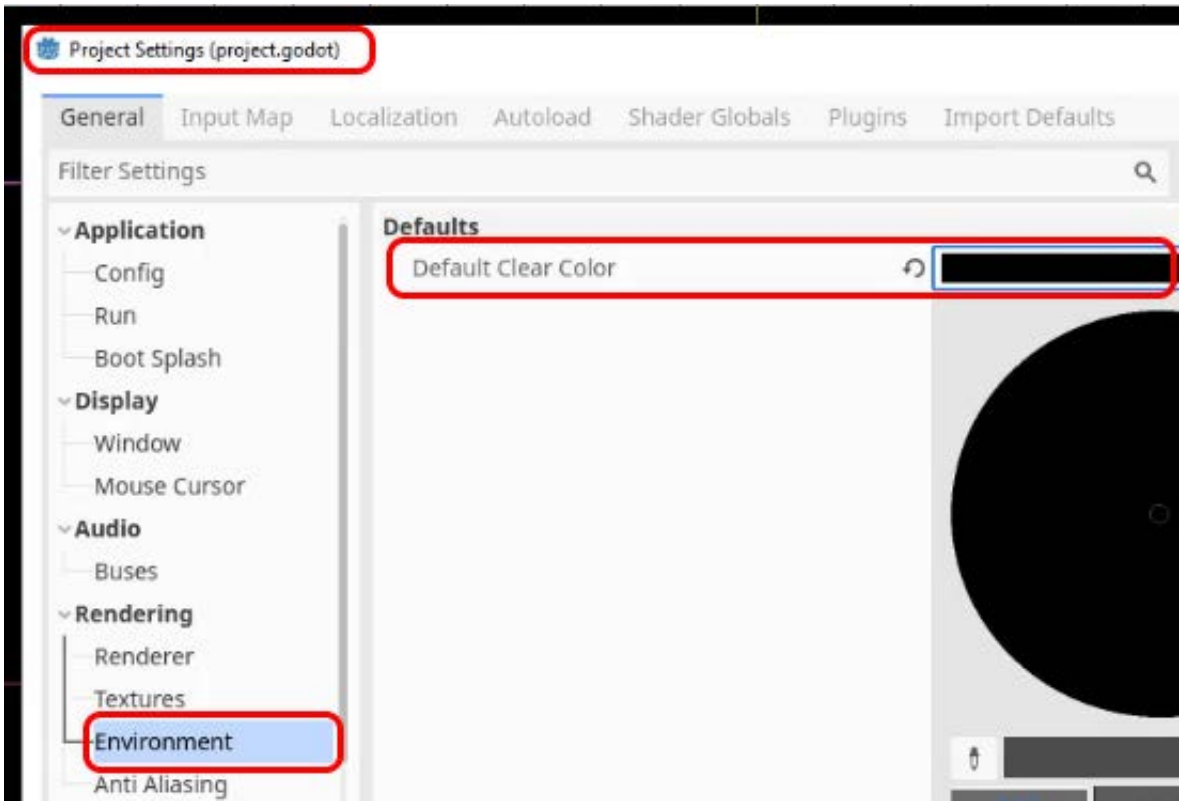
```
func _ready():
    for i in spawn_count:
        ...

        var star_size = randf_range(0.5, 1.0)
        star.scale.x = star_size
        star.scale.y = star_size
```

This will improve the look of our starfield by changing the size of some stars randomly.



You can reload the scene as many times as you like and you will get a random starfield every time. You can also change the **Spawn Count** property on the *Main* root node from the *Index* window to change the number of stars that appear.

The final thing we can do is change the **Default Clear Color** property in the **Project Settings** to black, which will make our *Stars* look much more natural.



In the next lesson, we will begin work on the *Skiing mini-game*, which will be our final mini-game,

and focus on handling collisions inside Godot.

This project introduced the concept of loops. Let's now have a deeper dive into exactly what they are and how they work.

What is a Loop?

Loops allow us to iterate multiple times upon a single piece of code. If you wanted to run a function 1000 times, you could go ahead and copy/paste that function call 1000 times, but it would take too long, it would look messy, and it would not be easy to modify in the future. So with a loop, all we need to do is create a for loop that runs 1000 times and call the function inside of that just once.

Structure of a Loop

What we created is known as a **for loop**. You define it like so:

- for [loop variable] in [iterate over]

So if I were to write this, how many times would the print message print?

```
for i in range(15):
    print("Looped")
```

15, that's right.

Now what about this *i* variable? We can actually call that whatever we want, as we are essentially defining a new temporary variable – "i" is just the naming standard.

This is an integer number which increases by 1 each iteration of the for loop. This can be used as an index for an array, a way to offset something relative in the loop, etc. This variable always starts at 0 for the first itertation of the loop, and continues on like this:

- 1st iteration, i = 0
- 2nd iteration, i = 1
- 3rd iteration, i = 2
- etc…

So running this code, what do you think the print output would look like?

```
for i in range(5):
    print(i * 2)
```

- 0
- 2
- 4
- 6
- 8

Since "i" increases by 1 each iteration, the output is going to be different each time.

Learn more about for loops here.

Preload

Inside of our **Loops** script, we are preloading the **star_scene** variable. What does this mean? It's essentially loading in the star scene resource to memory for us to use.

```
var star_scene = preload("res://Loops/Star.tscn")
```

There are two ways to load in a resource through script:

- **load** – This will load the resource when it is required. So for our scene example, the star scene would not be loaded until we require to instantiate it.
- **preload** – This will load the resource as early as it can when the node containing the script is initialized. This means that by the time we want to instantiate the star, the scene has already been loaded into memory.

These two methods have their pro's and con's, and it really depends upon what you are doing to decide upon which to implement.

Instancing Scenes

The **instantiate** function is how we create new scenes at runtime. It's important to see scenes as templates. In order for us to use them in our games, we must instance them. There is a process behind this:

1. We must first load the scene into memory (which we discussed above with preloading).
2. We then instantiate the scene, creating a new instance of it.
3. We must then add that node to the tree in order for it to exist.

```
var star = star_scene.instantiate()
add_child(star)
```

Additional Resources

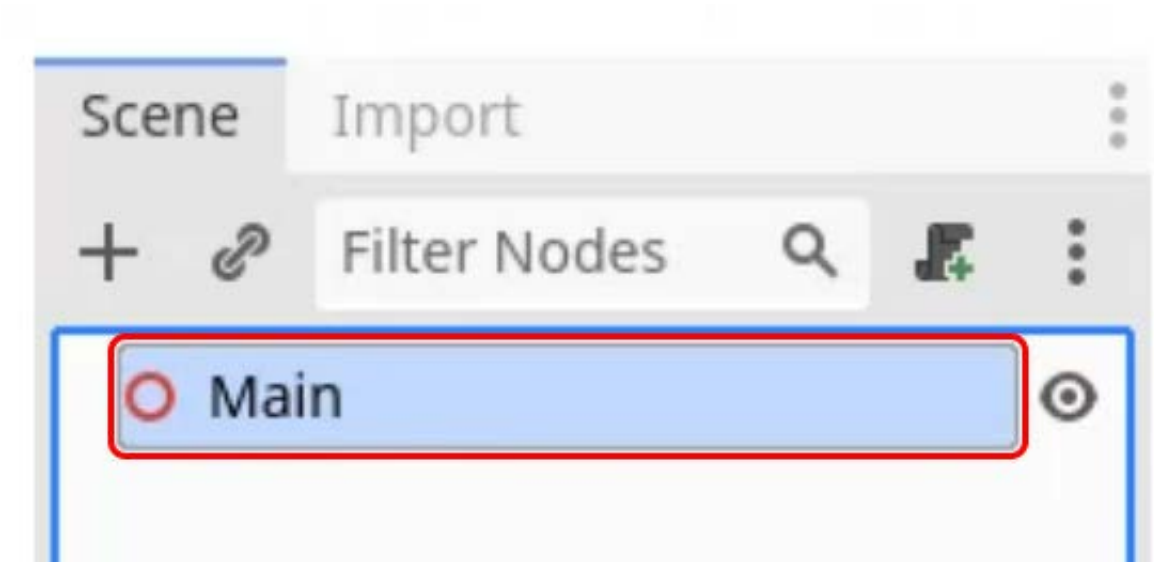
If you wish to learn more, you can refer here to the Godot documentation.

- For Loop
- Loading vs Preloading
- Instancing Scenes

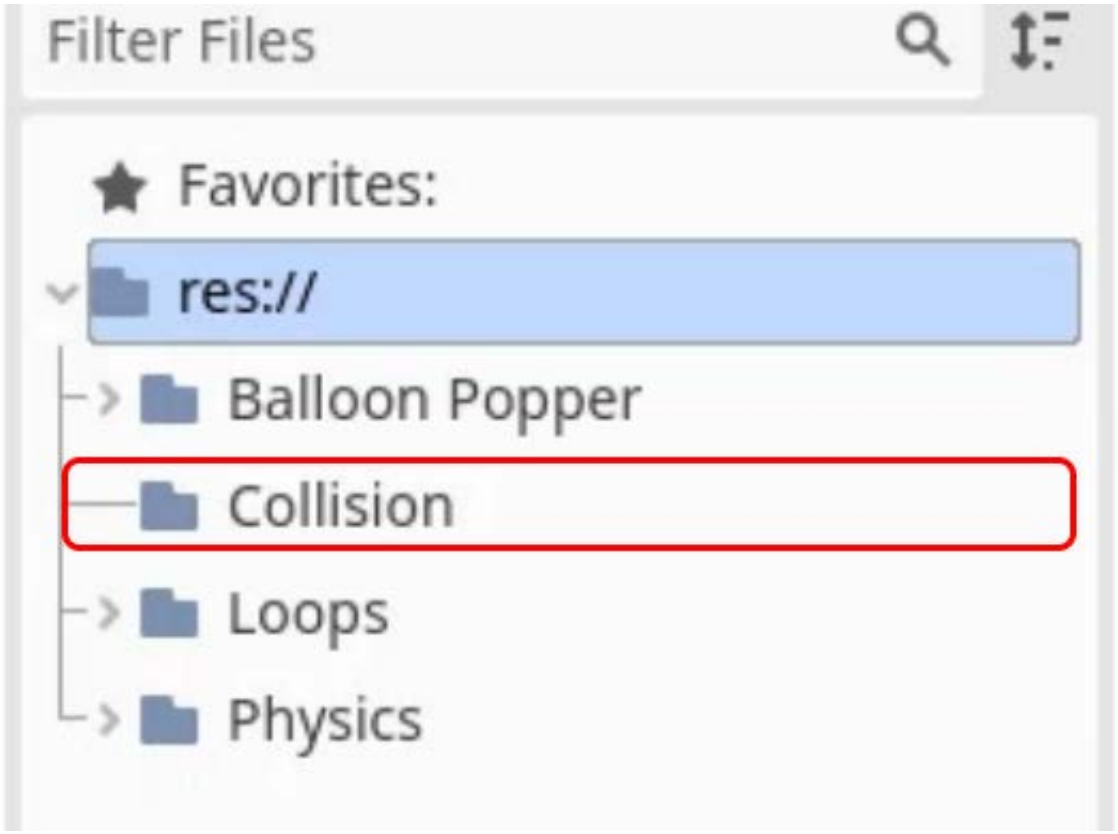
In this lesson, we are going to start the fourth mini-game project which will focus on Godot's collision interactions using a skiing game.

Setting up the Scene

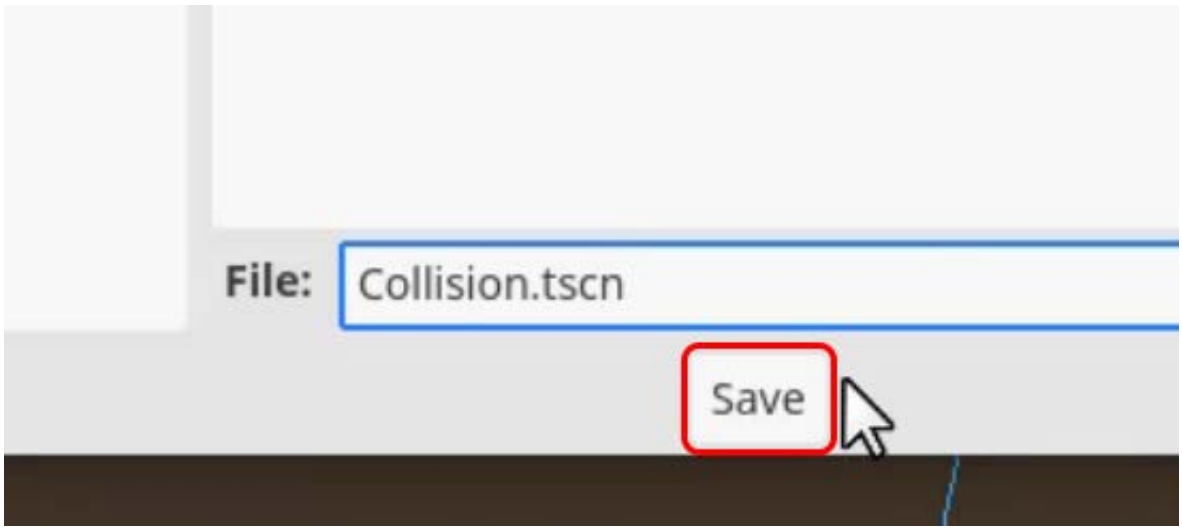
As with the other mini-projects, we will start by creating a new scene. This game will be a **3D** project, so we will use a 3D root node, which we will name *Main*.



We will create a new folder called *Collision* for this project.



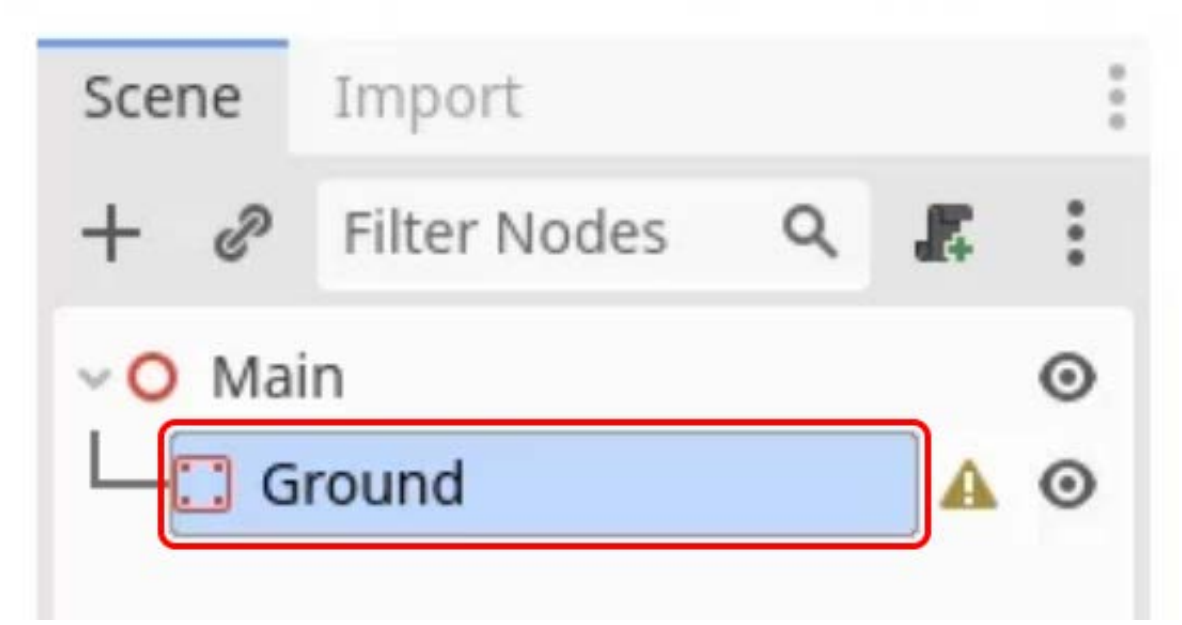
Inside the new folder, we will save the scene as *Collision.tscn*.



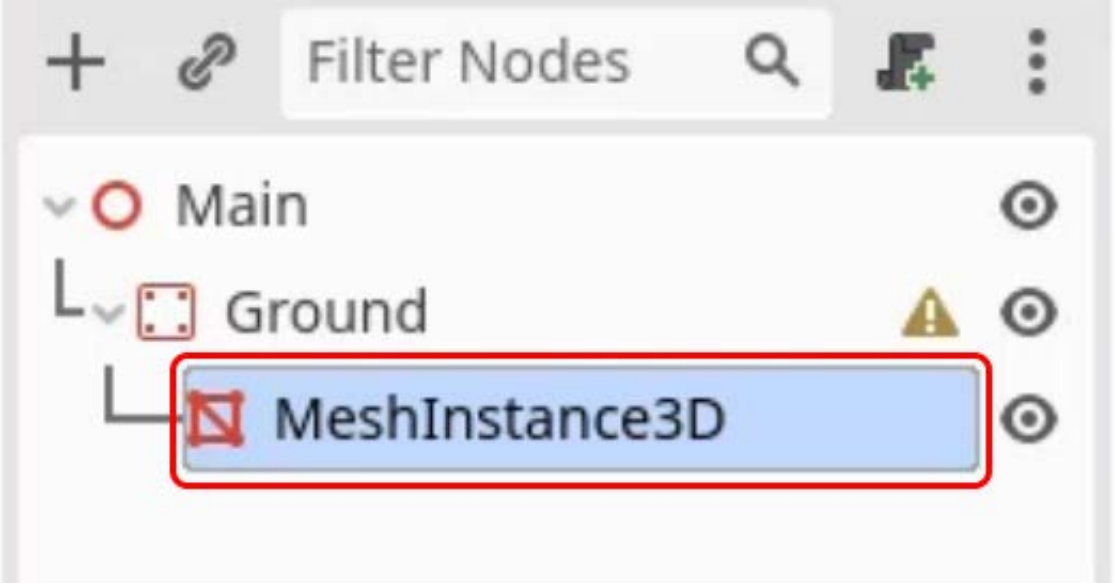
Creating the Ski Slope

We are going to begin by creating our *Ski Slope*. This will be a **StaticBody3D** node, which is used for having physical, static objects in a 3D game world.

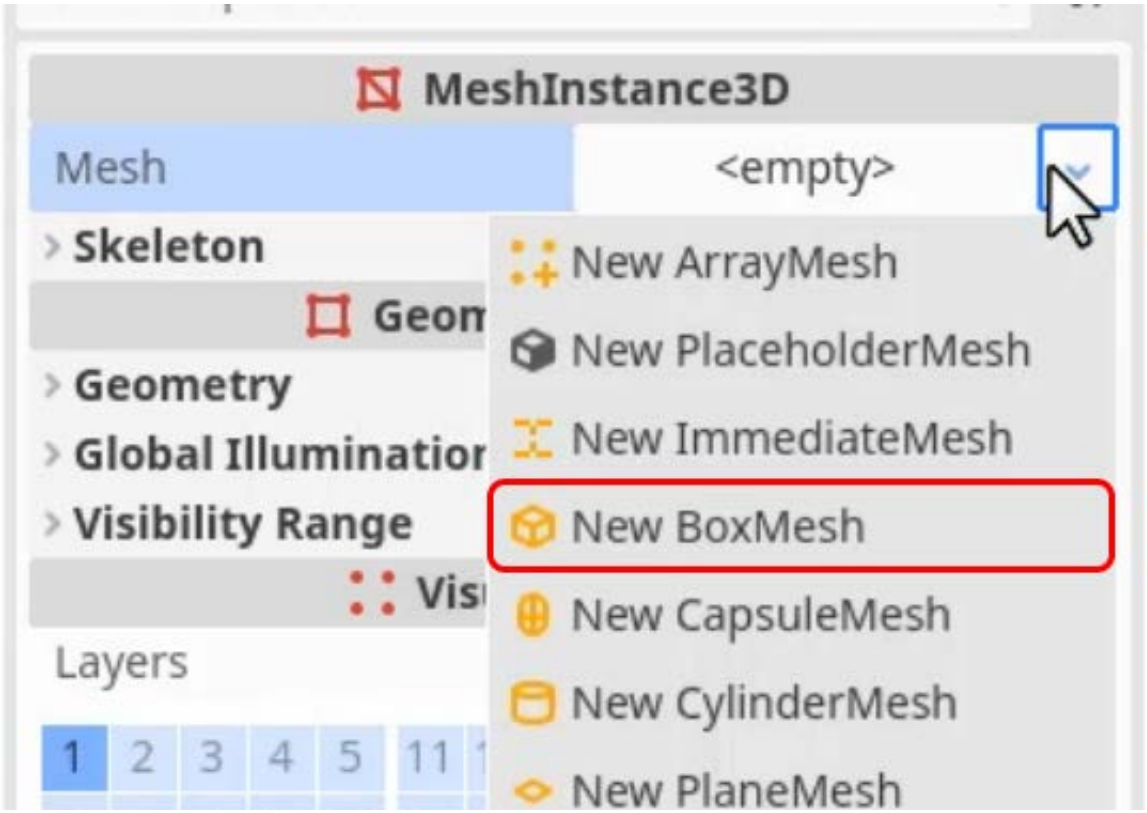
We can then rename this to *Ground* to represent its use in the scene.



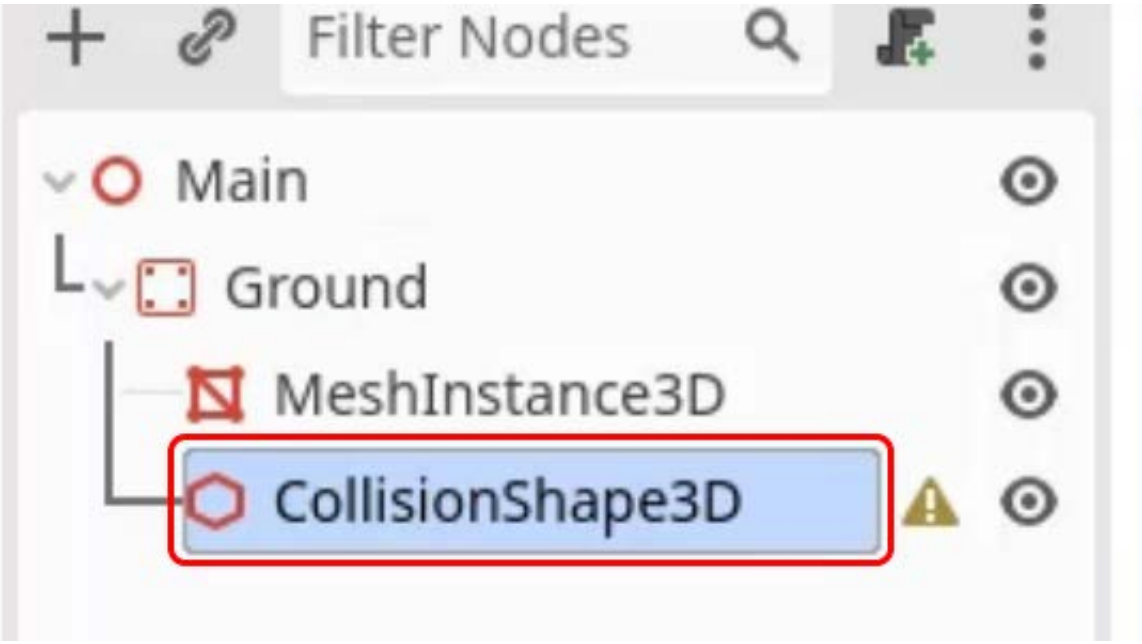
Next, we will add a **MeshInstance3D** node as a child of our *Ground* node.



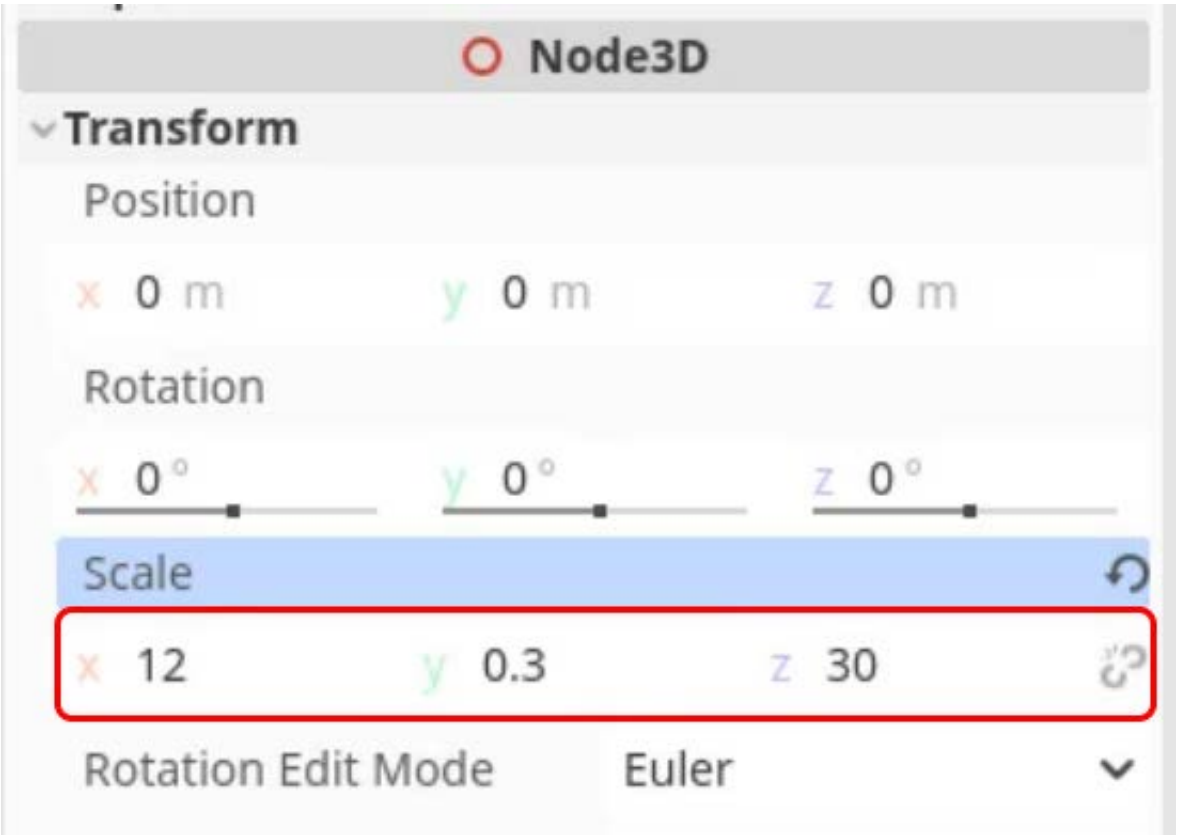
This will be used to render the mesh to the screen, and we will use a **New BoxMesh** as the value for the **Mesh** property.



We will then add a **CollisionShape3D** to match this.



For the **Shape** property, we will use a **New BoxShape3D** value to match the *MeshInstance3D* node. We then need to change the size of the cube. Select the static body parent, and select the **Scale Tool** (**R**). Scale the cube on the X-axis to 12, the Y-axis to 0.3, and the Z-axis to 30.

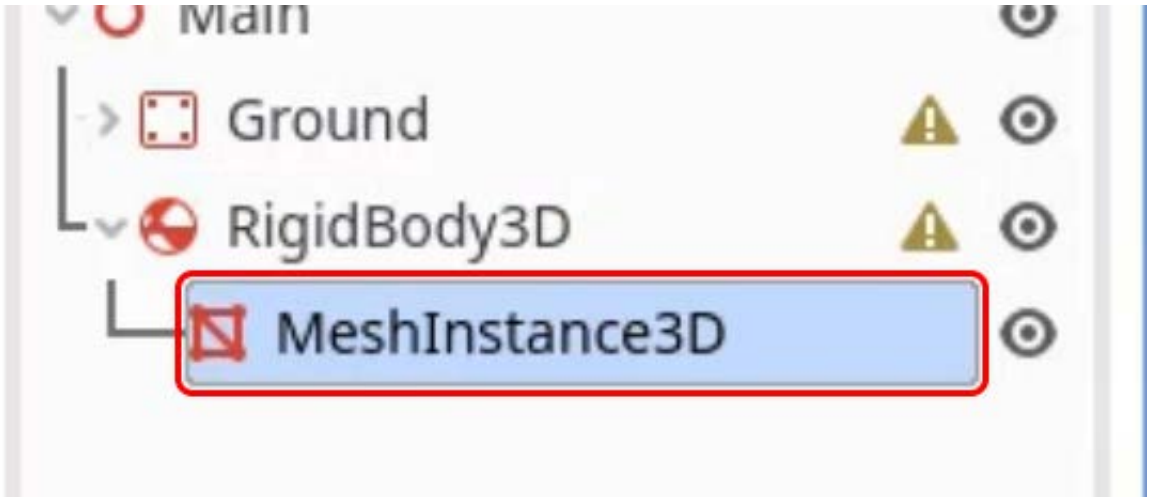


Then select the **Rotate Tool** (**E**), and rotate the *Ground* node around -20 on the X-axis.

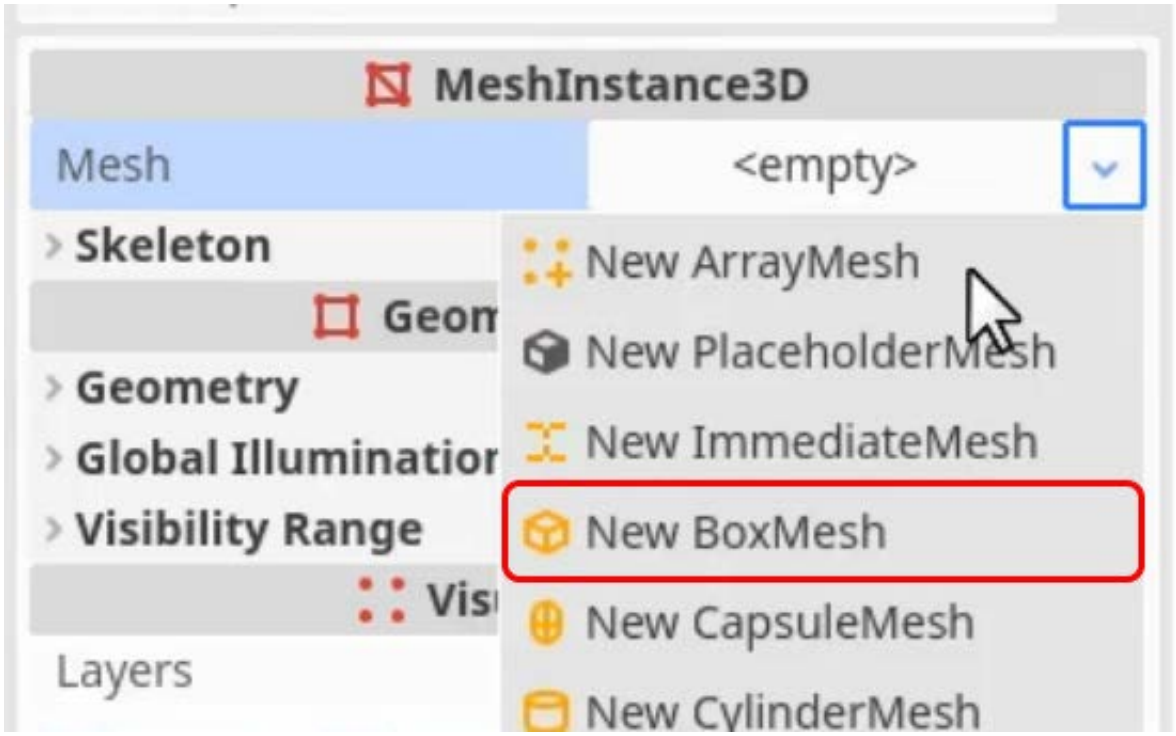
Creating the Player

We now need to create our player. For the parent of our *Player,* we will use a **RigidBody3D** node.

We will then add a **MeshInstance3D** node as a child so that we can see it.

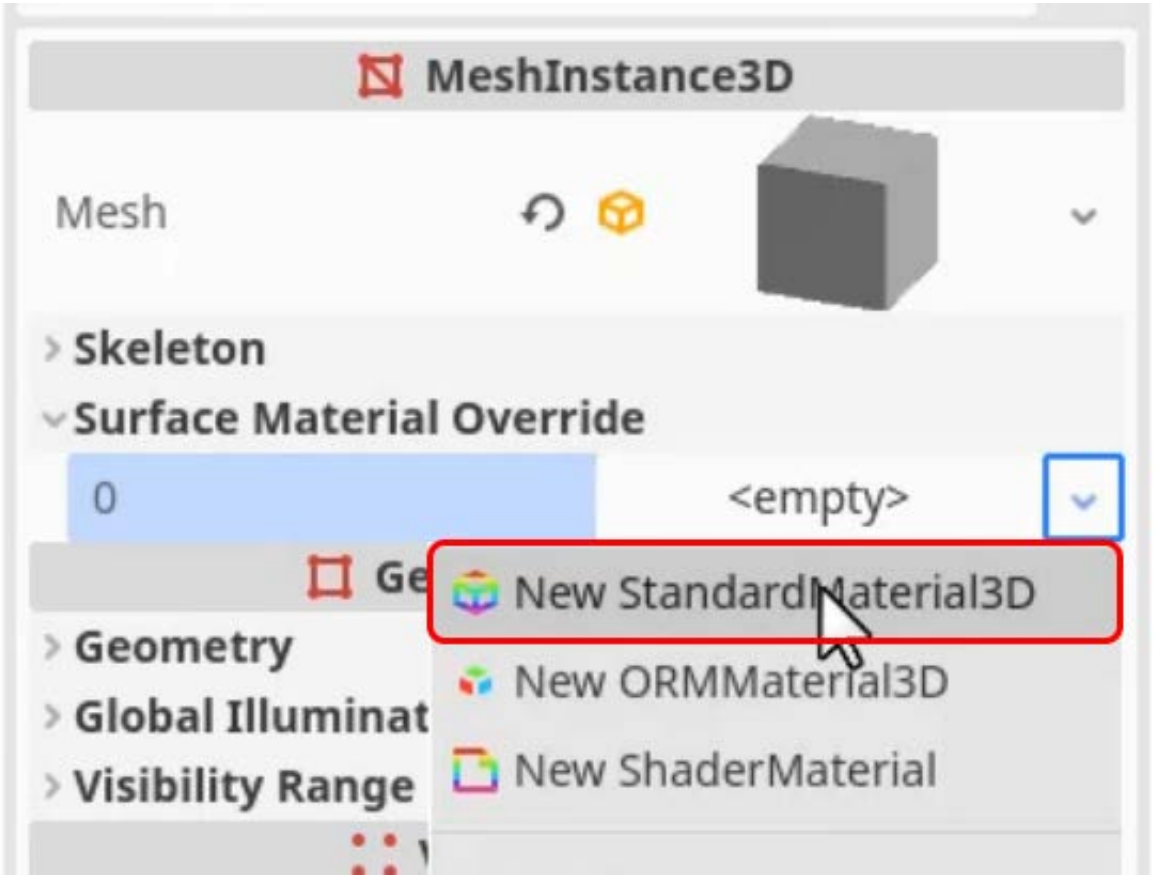


The **Mesh** property will be a **New BoxMesh**.

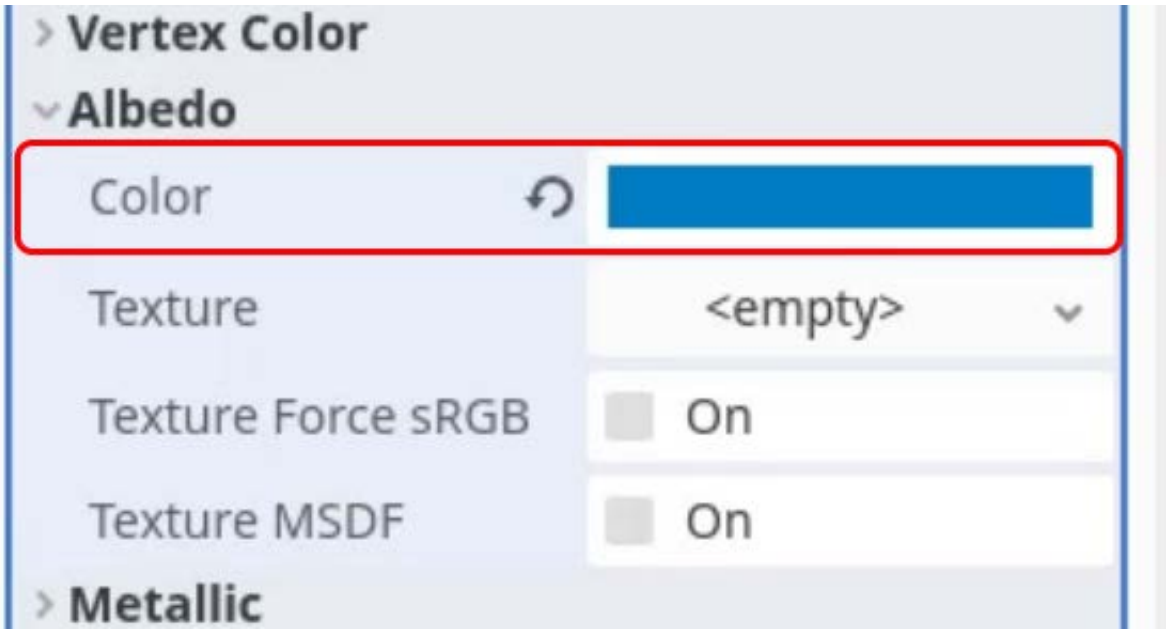


We will be using multiple *MeshInstance3D* nodes to create a model of our player. For this cube, we want to change the color by adding a **New StandardMaterial3D** in the **Surface Material**

Override of the mesh instance.

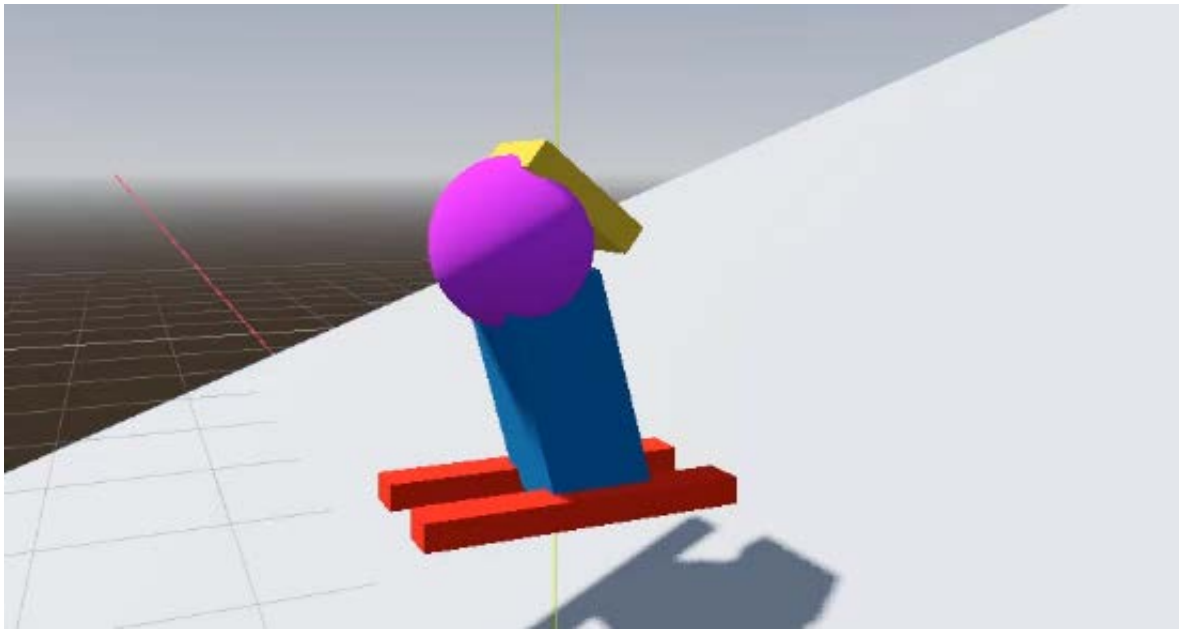


From here we will change the **Albedo Color** to be a color you like, we will use blue.



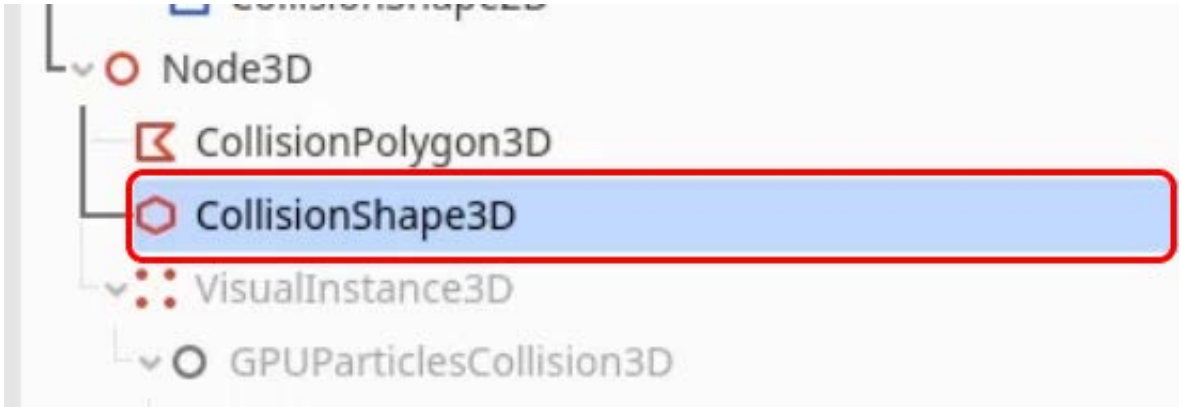
From here you can duplicate, resize, rotate, and move the cubes to design your own player character model.

In this lesson, we will be setting up our character's movement and environment in the game engine Godot. Before doing this, however, in the last lesson, we set the challenge to create a character model using different mesh instances with their own materials. Here is the example we made using five different *MeshInstance3D* nodes to create our skiing character.

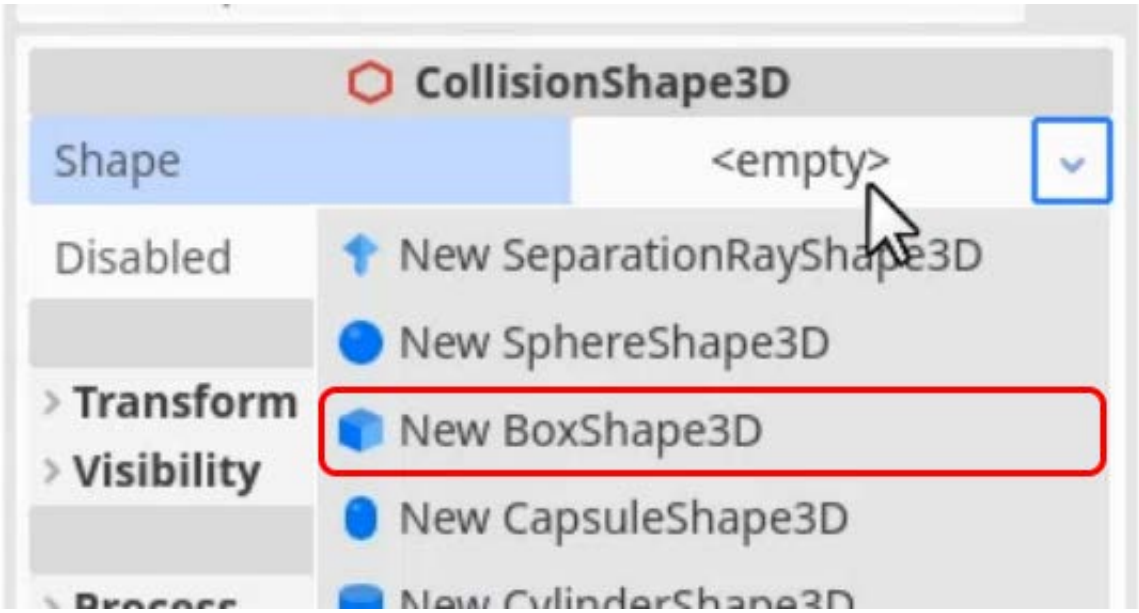


Creating the Player

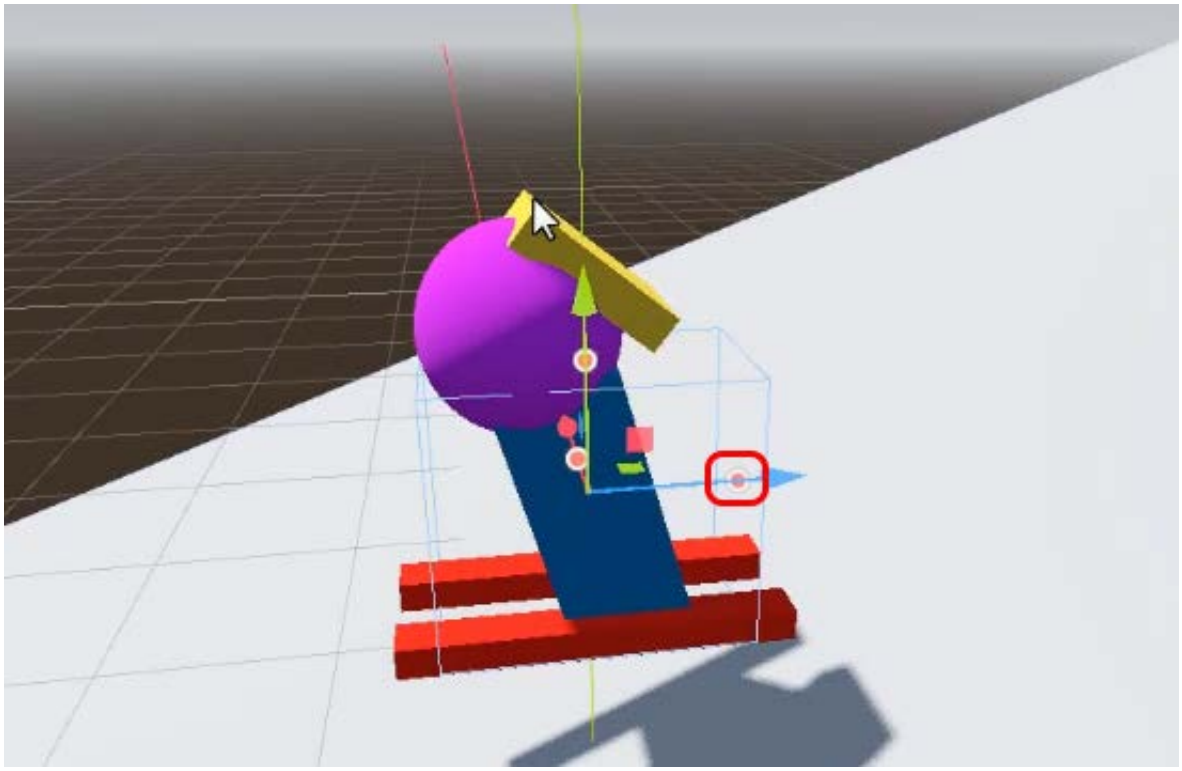
First, we will create a **CollisionShape3D** as a child of the *RigidBody3D* node.



We will make the **Shape** property a **New BoxShape3D** to match our character.

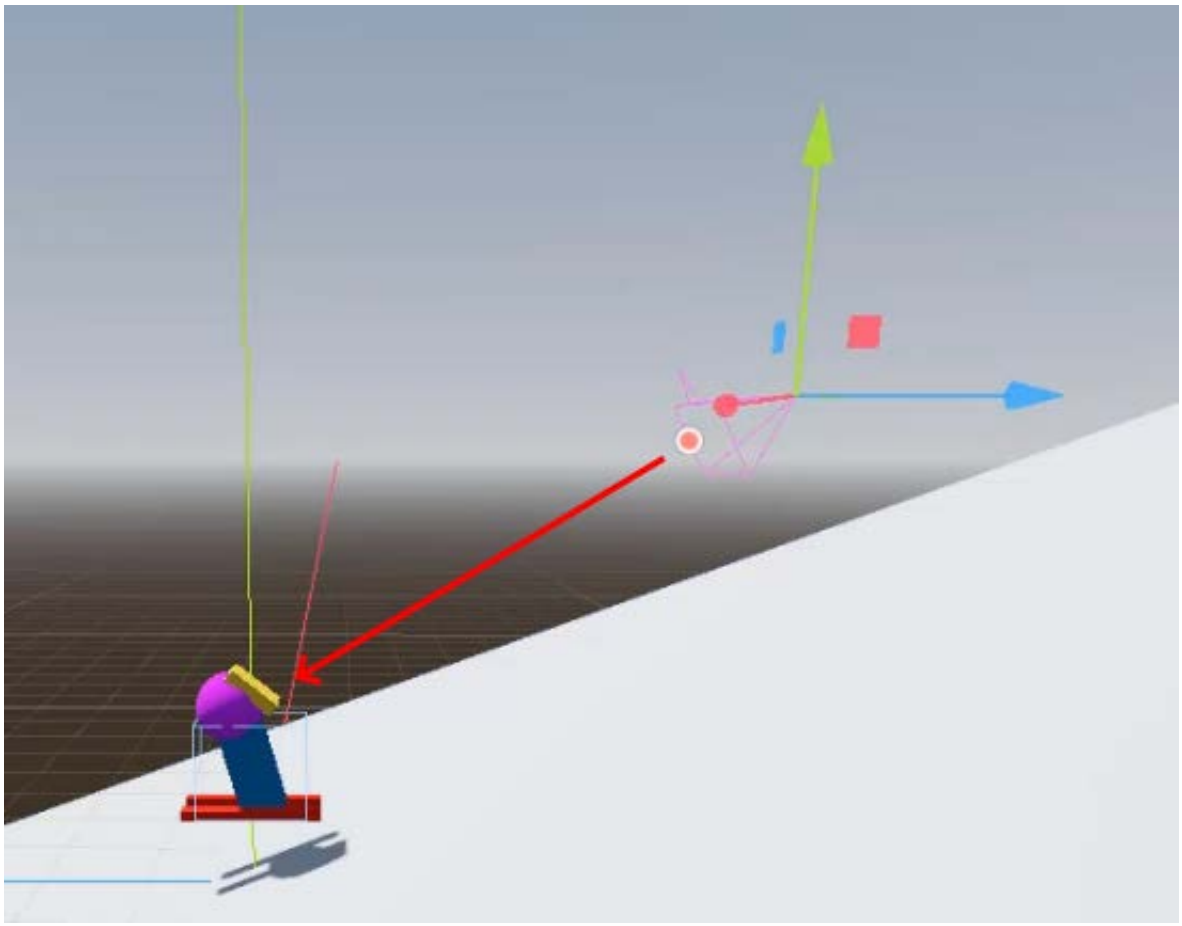


We then want to fit this to the bounds of our player model, using the orange circles to scale the box.

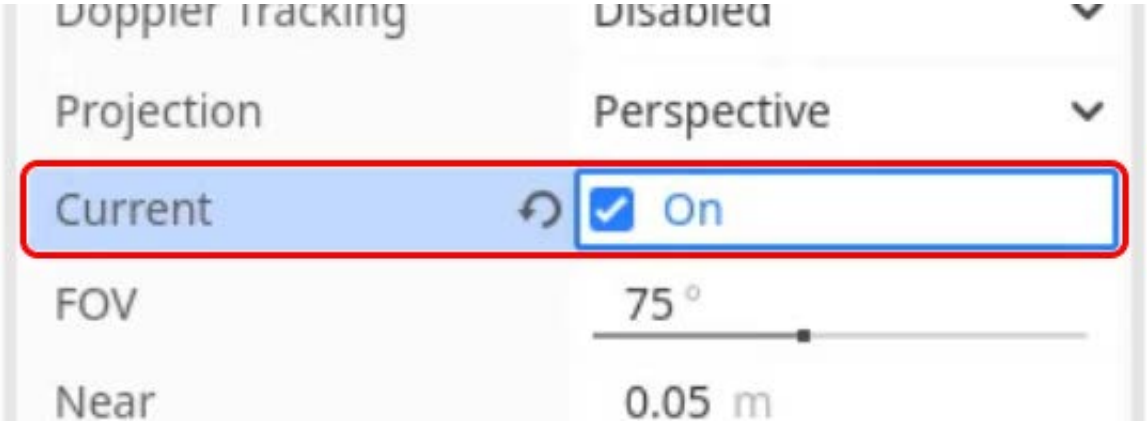


We then want to add a **Camera3D** as another child node of the *RigidBody3D* node, so that we can track the player down the slope.

Then we want to **move** and **rotate** our camera so that it is looking down on our player.

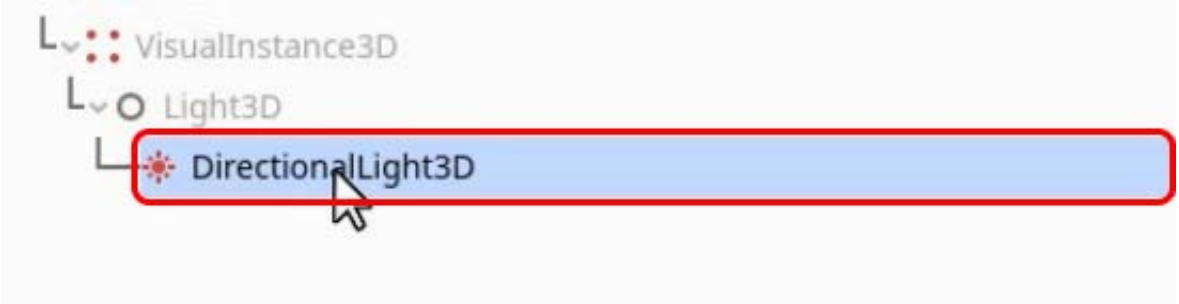


We then want to enable the **Current** property in the inspector, to tell Godot to use this as the camera for the game window.

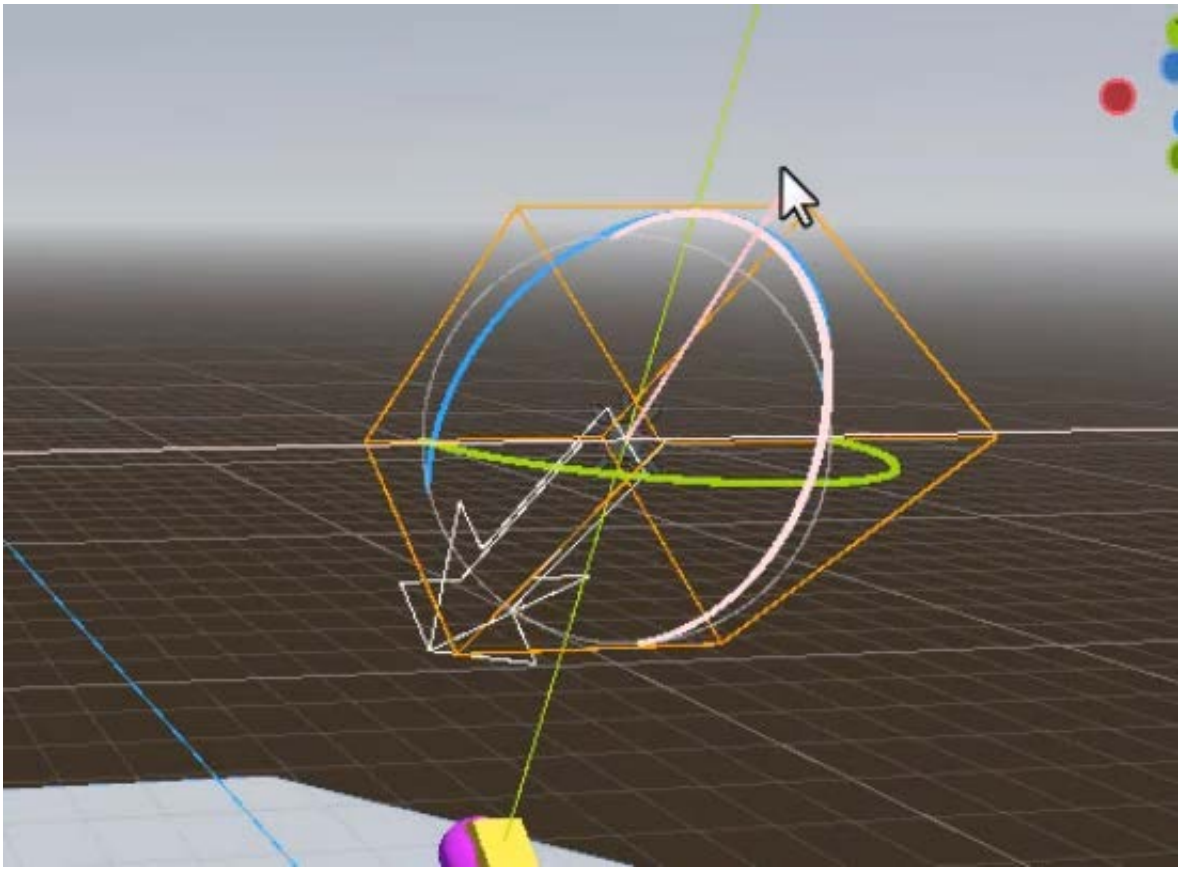


Creating the Environment

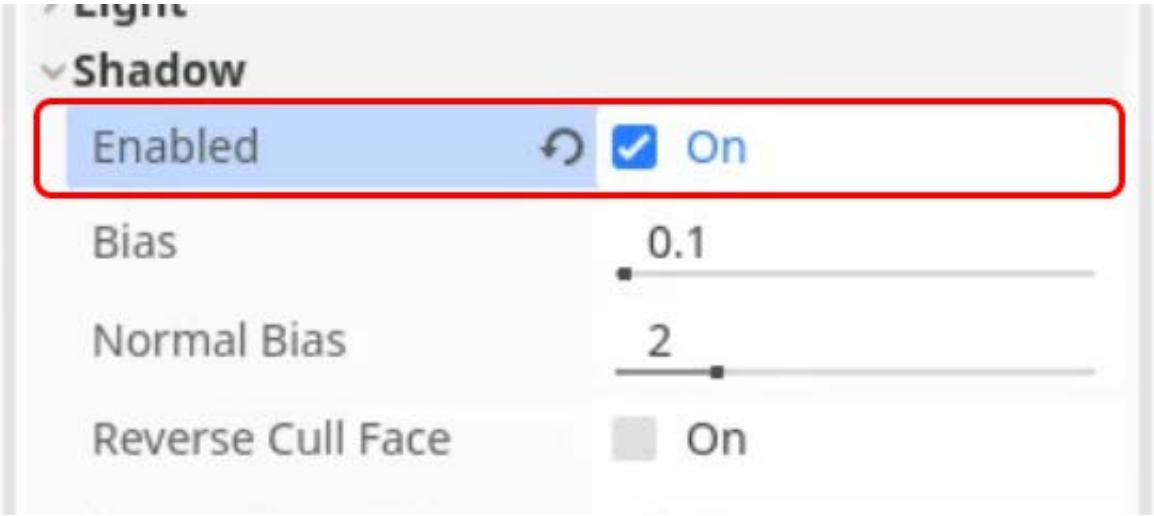
We will then create a **DirectionalLight3D** node to provide lighting in our scene.



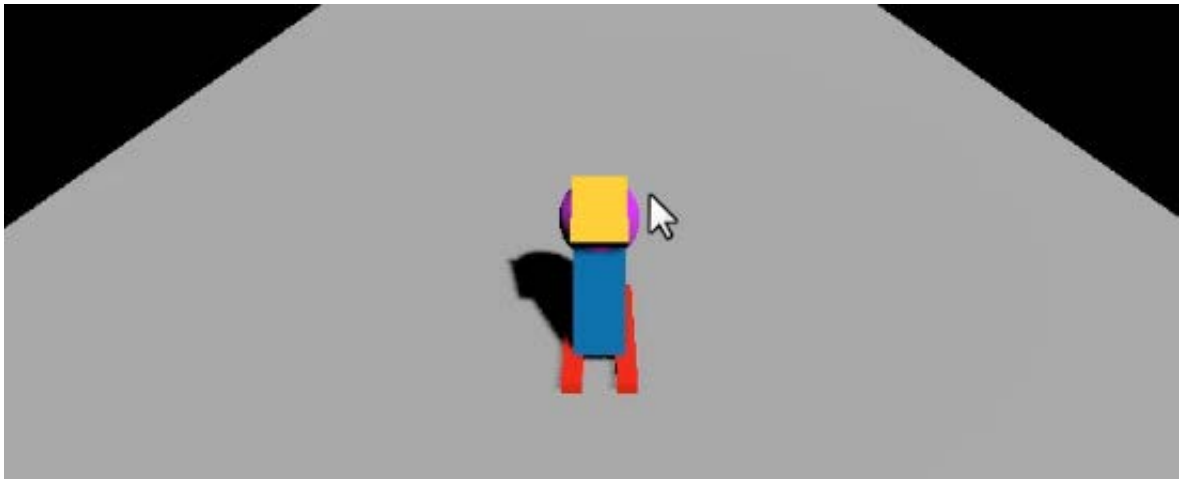
From here, position the light to shine down on an angle at our player.



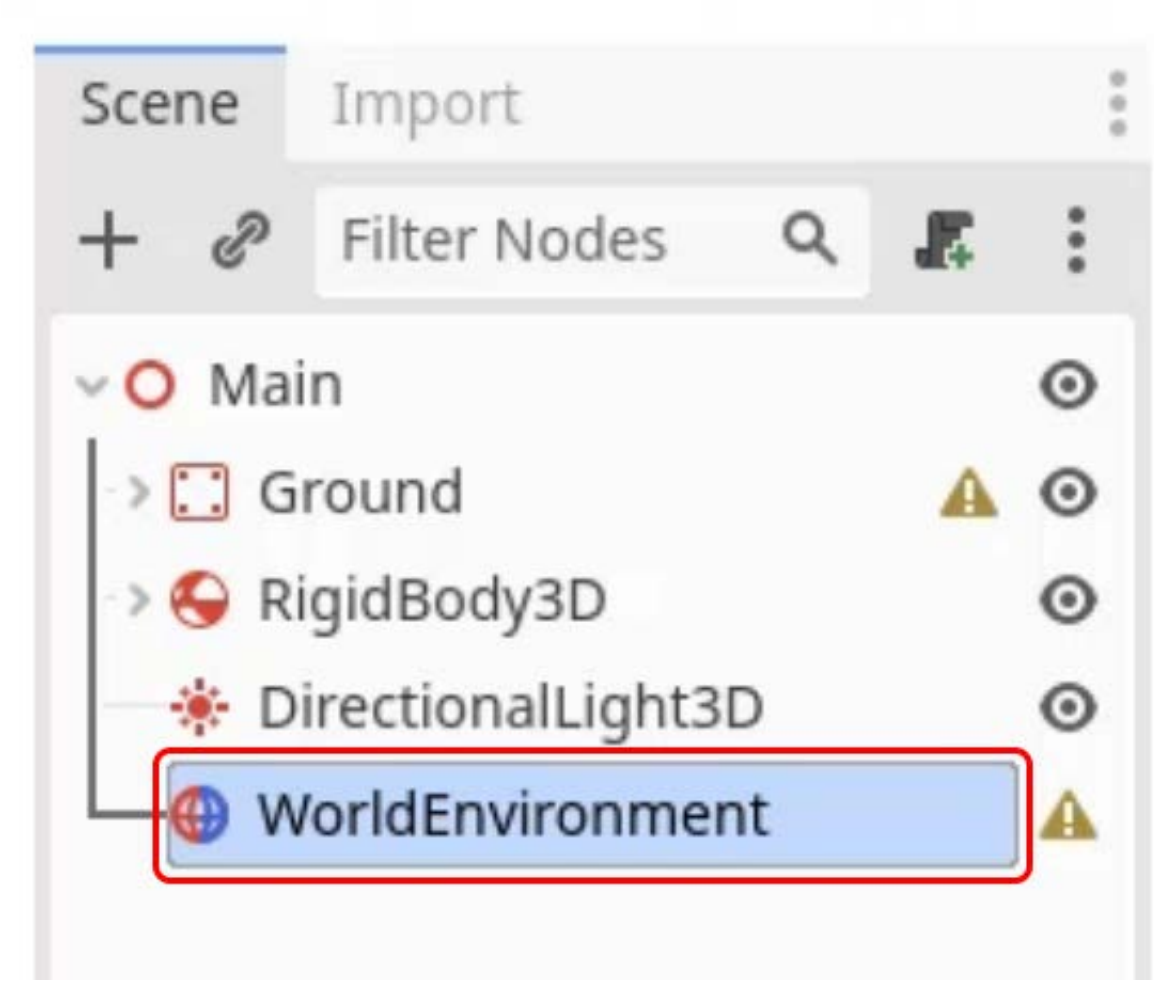
Finally, enable the **Shadow** property of the *DirectionalLight3D*.



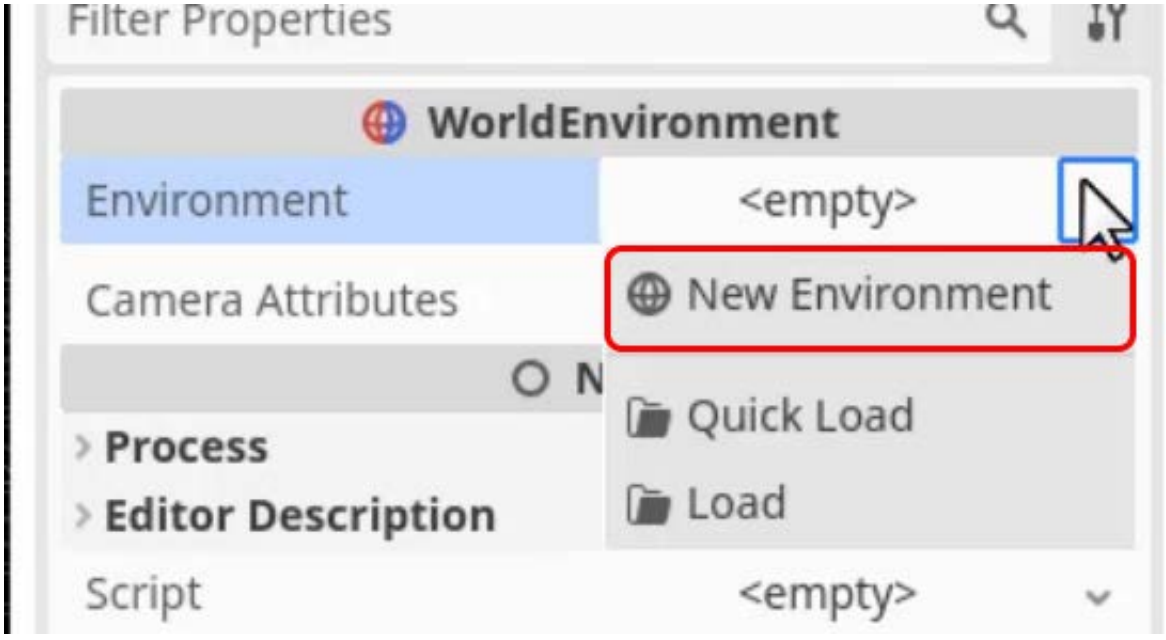
Now, when you press **Play** you will see the scene is lit and we can see our character.



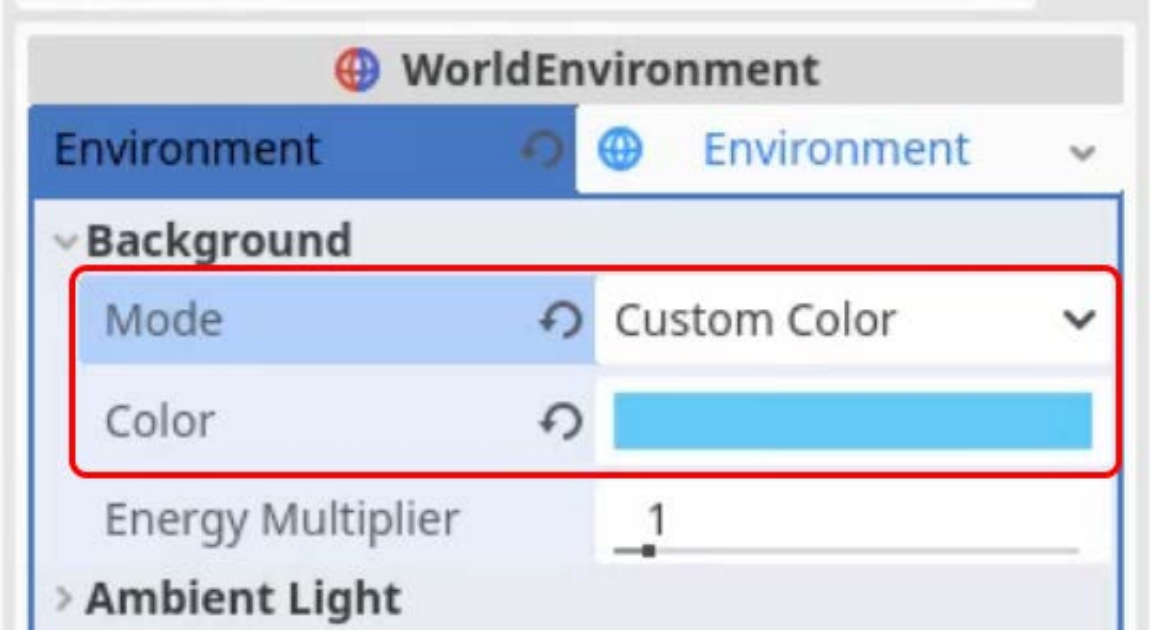
Currently, however, it doesn't really look like a skiing environment. To fix this we will use a **WorldEnvironment** node, which will allow us to change things such as background color, fog, and lots more.



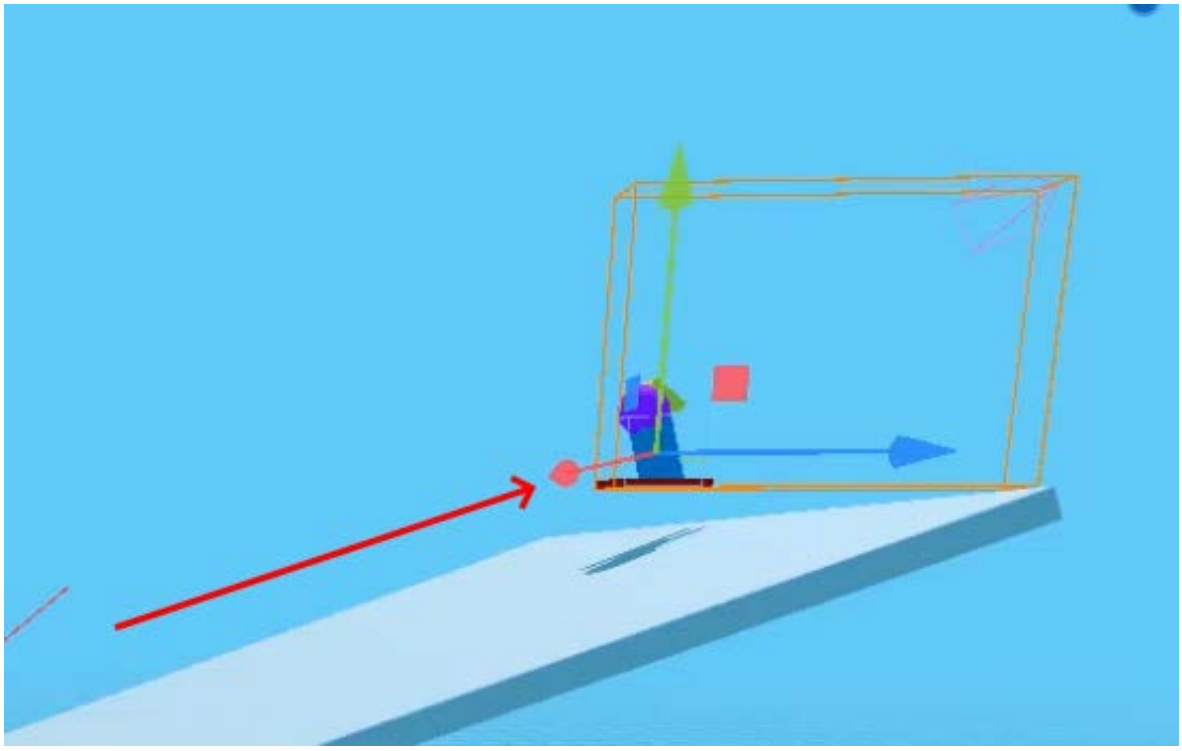
We will create a **New Environment** for the **Environment** property.



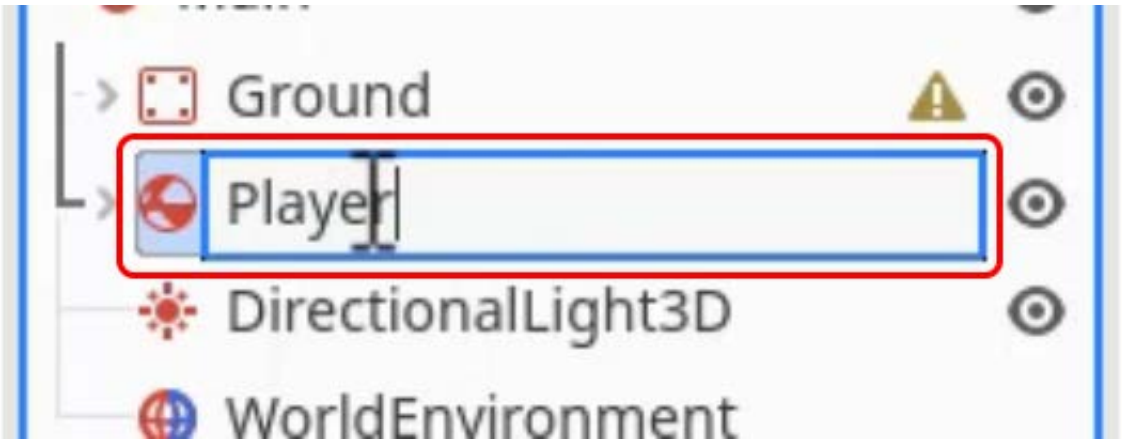
To begin with, we will change the *Mode* to **Custom Color** in the **Background** tab. We will change this to a light blue hue.



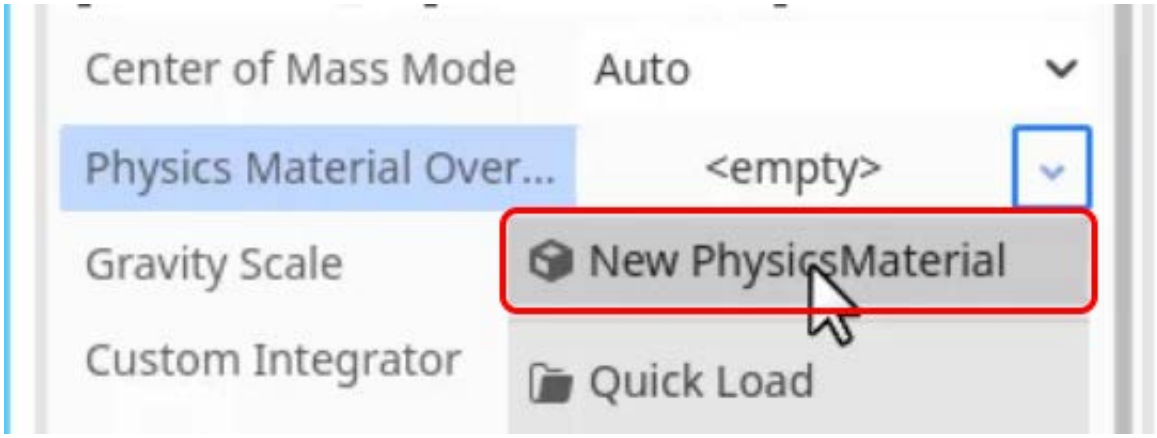
Finally, we will move the *RigidBody3D* player node to the top of the slope, so that when we press *Play* it starts at the top.



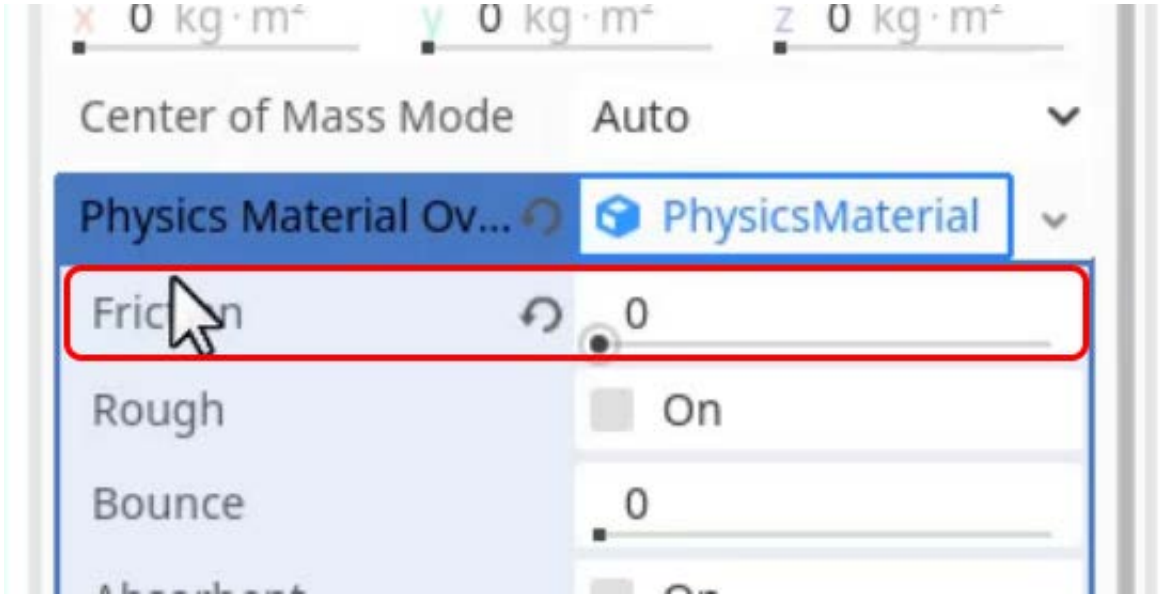
We will also rename the *RigidBody3D* node to *Player* so that we can easily identify it.



To make the *Player* slide down the slope, we will also add a **Physics Material Override** value.



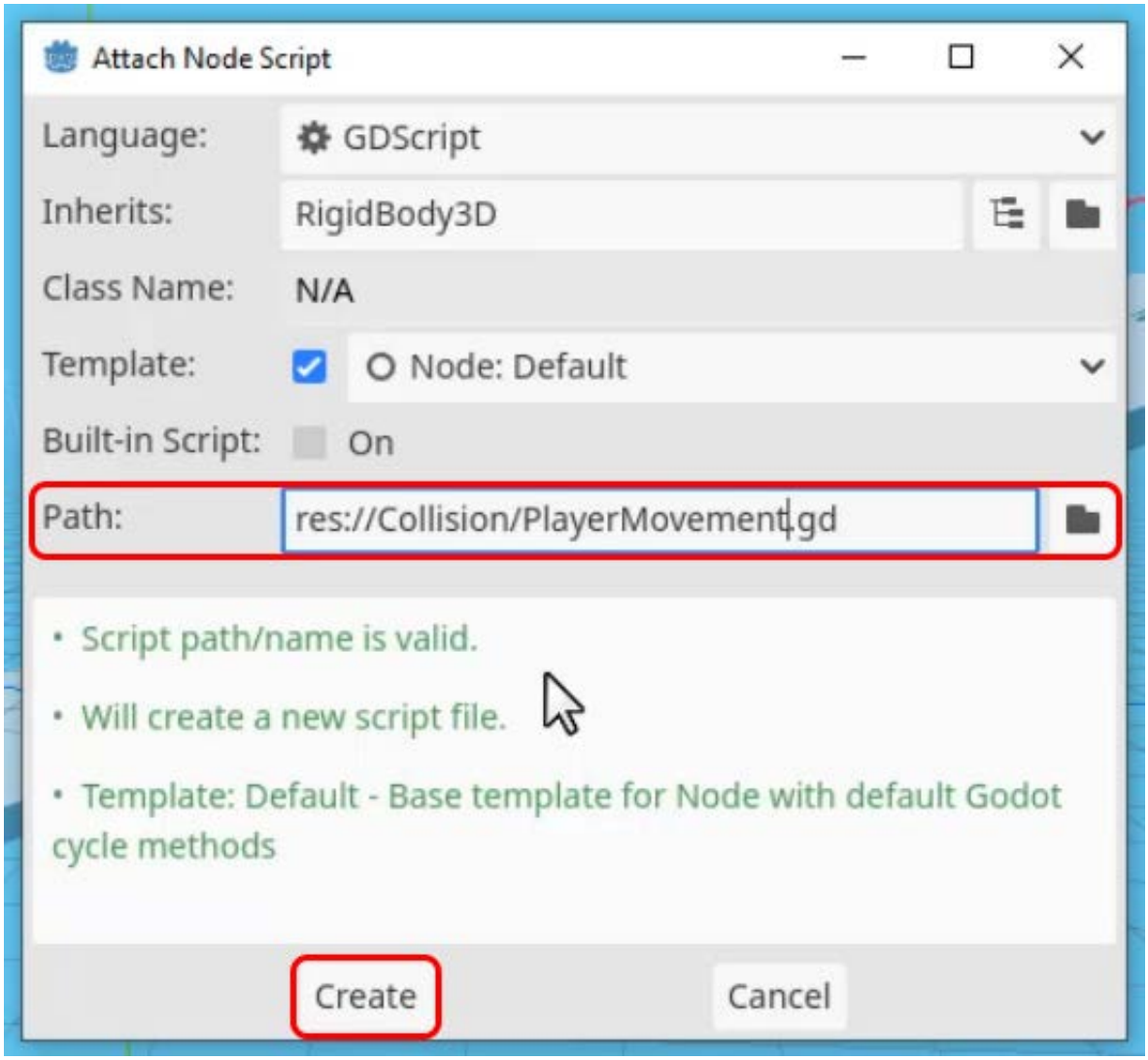
We will then set the **Friction** property to **0** so that our *Player* slides down the slope.



Now when you press **Play** you will see the *Player* slides down the hill as intended.

Creating the Movement

We will then select our **Player** node and create a script called *PlayerMovement.gd,* and make sure it inherits from *RigidBody3D*.



We will delete the *_ready* and *_process* functions and create an exported variable called **moveSpeed** of type **float** that has a default value of **2.0**.

```
@export var move_speed : float = 2.0
```

We will then create a function called *_physics_process* which will detect our key presses. The *_physics_process* function is useful for physics because it runs at a fixed rate, unlike *_process* which can change depending on the FPS of the game.

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

We will then set our **linear velocity** on the **x-axis** to be negative or positive *move_speed* depending on whether the left or right arrow key is pressed.

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



```
    if Input.is_key_pressed(KEY_LEFT):
        linear_velocity.x = -move_speed
    if Input.is_key_pressed(KEY_RIGHT):
        linear_velocity.x = move_speed
```

Now when you press **Play** you will be able to move left and right as we ski down the hill.

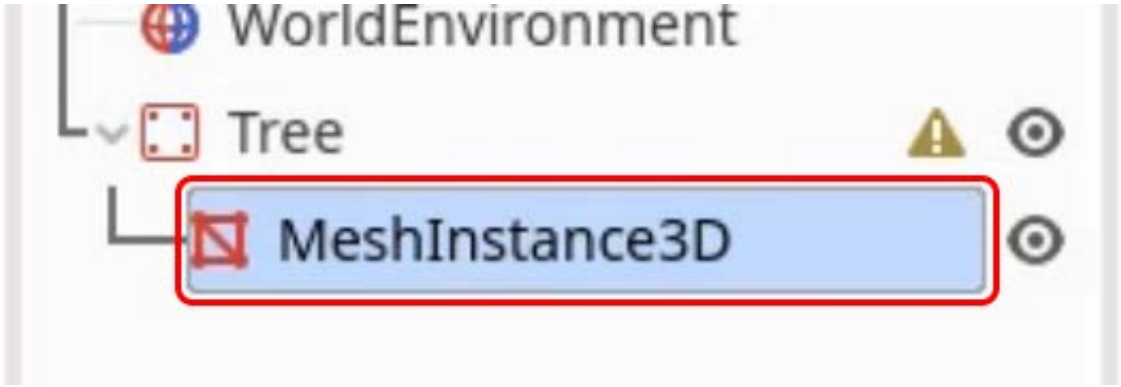
In the next lesson, we will begin implementing our trees for the *Player* to collide with.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

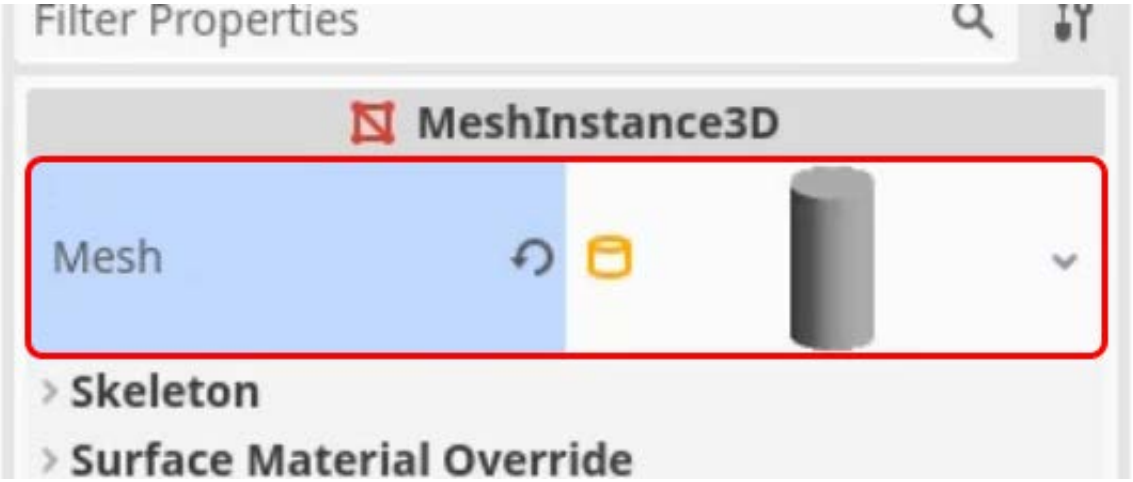
In this lesson, we will be creating our tree obstacle.

The first step is to create a new **StaticBody3D** node, which we will call *Tree*.

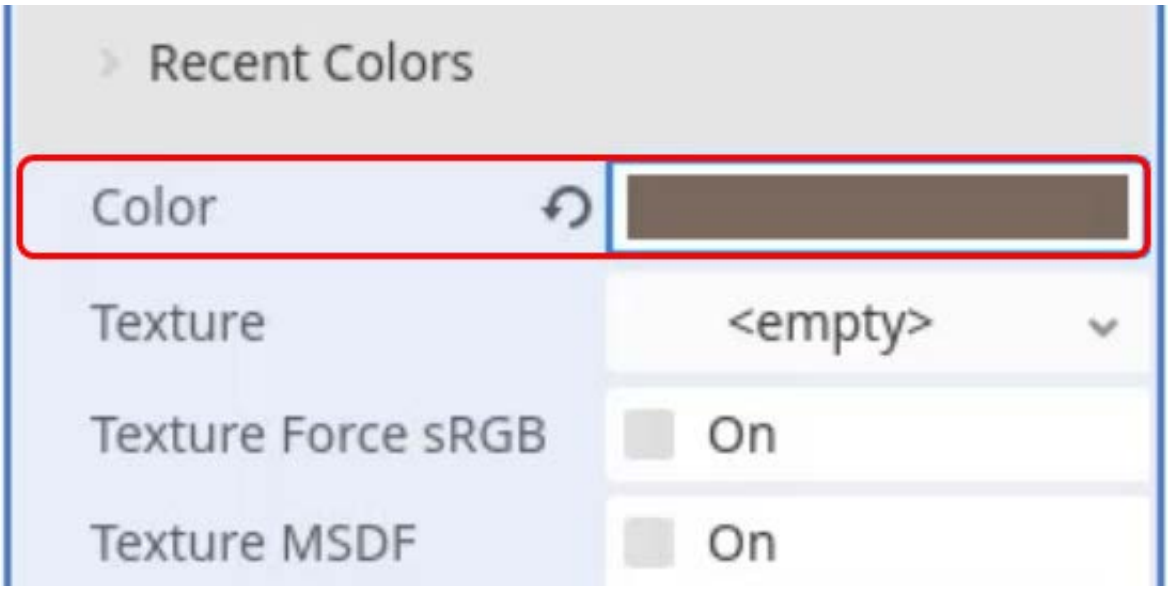
We will then add a **MeshInstance3D** node as a child of *Tree*.



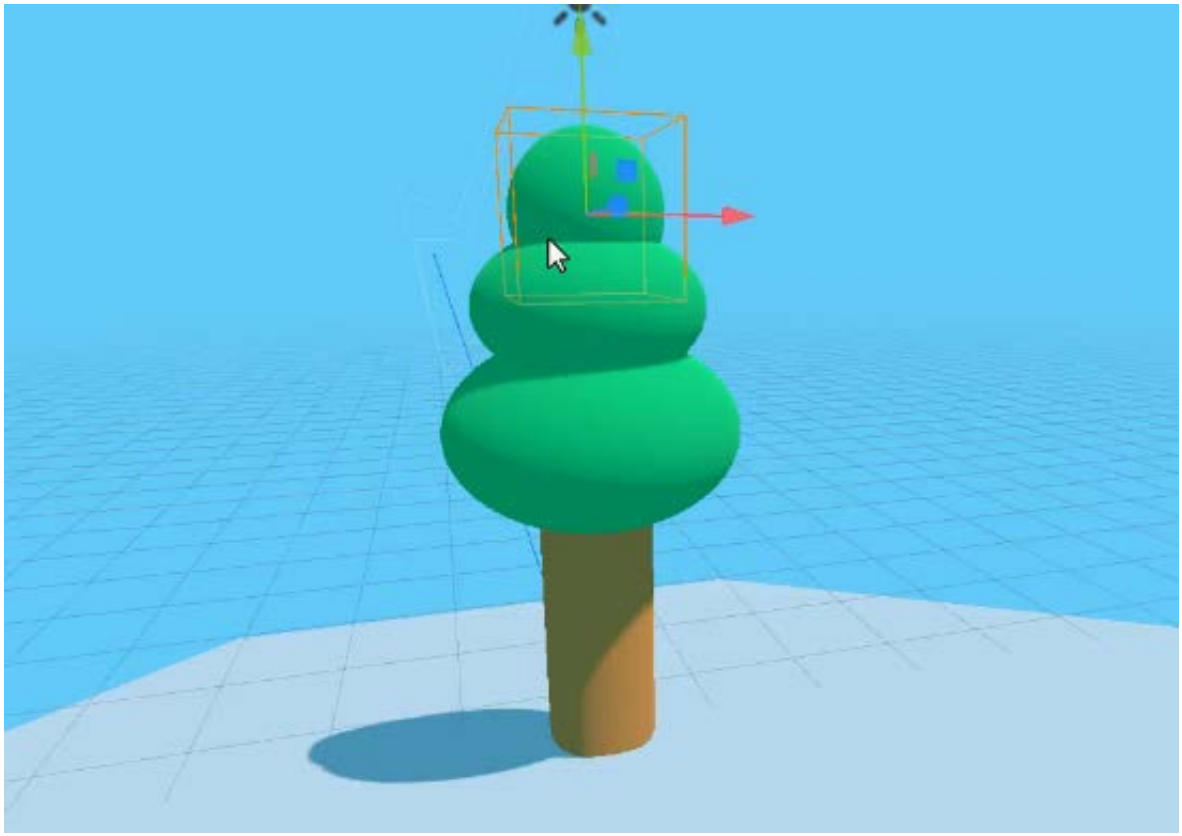
This will be a **Cylinder** for the trunk of the tree.



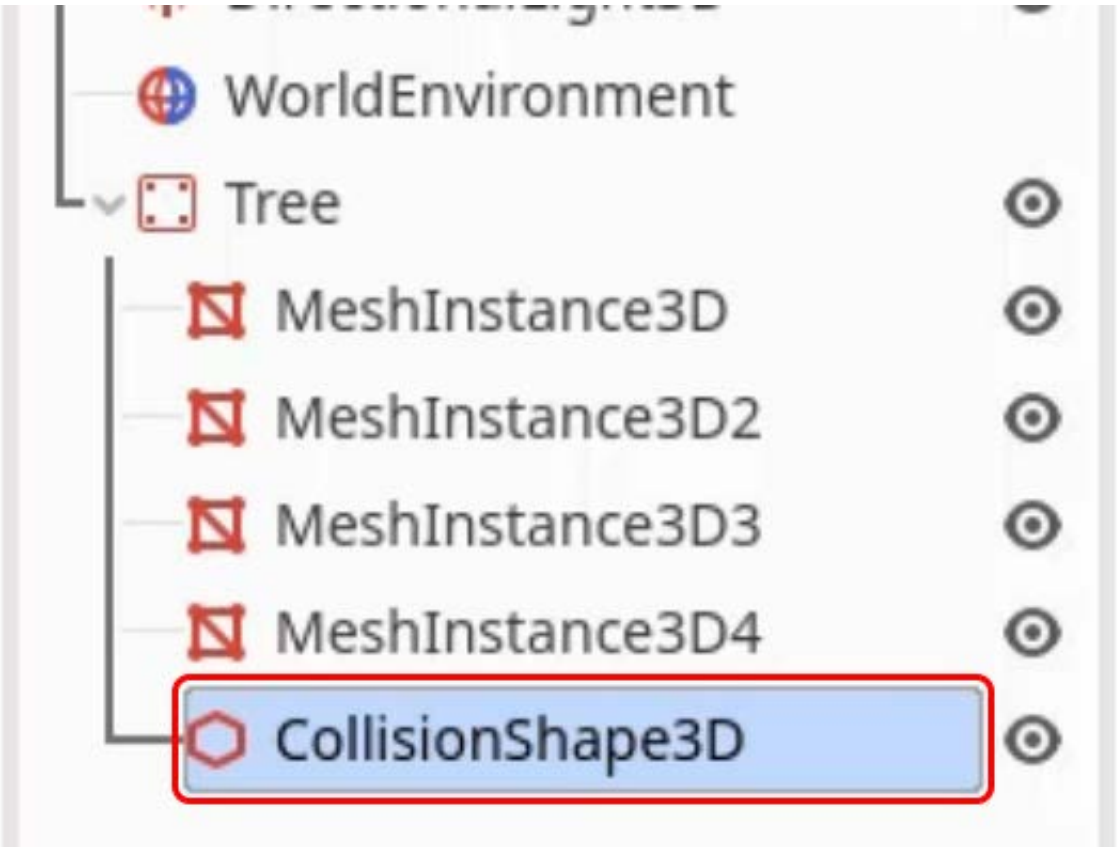
We will then give this **brown material** to be the color of bark.



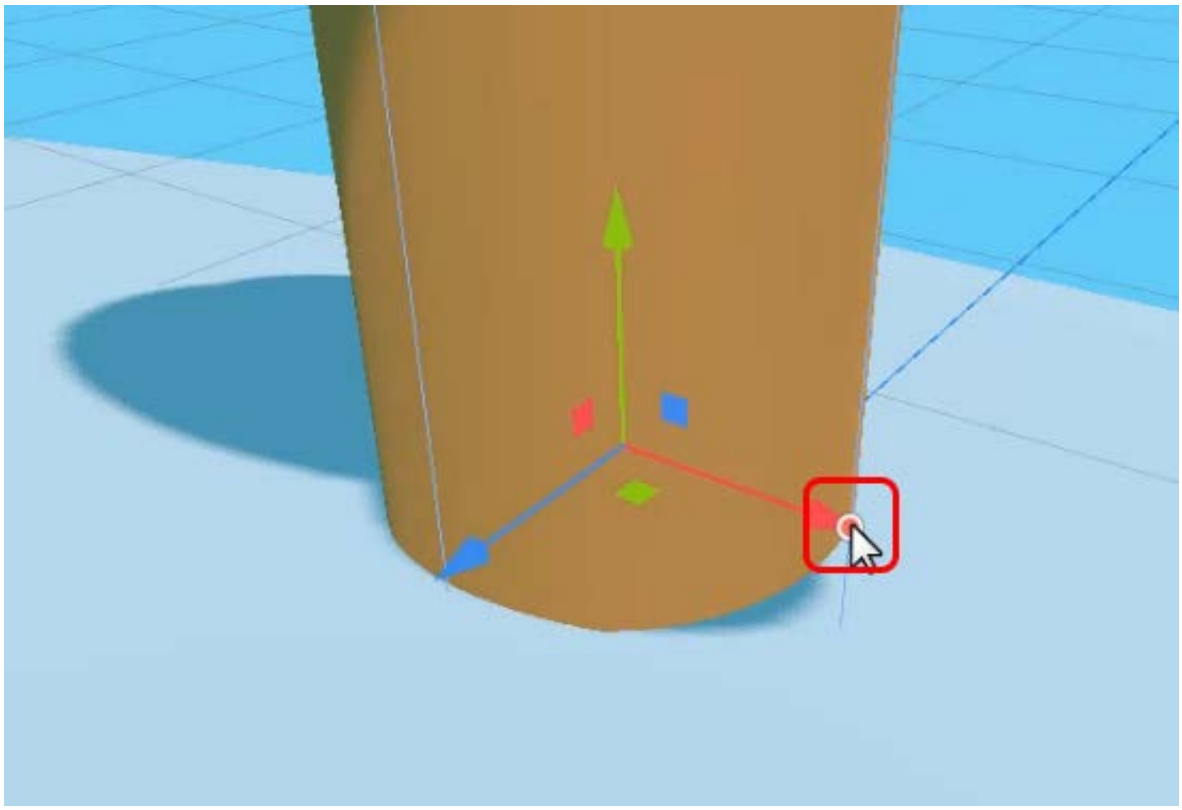
From here you can add more *MeshInstance3Ds* to create leaves on the tree, or alternatively create an entirely different model to be an obstacle.



We also want to add a **CollisionShape3D** as a child of our tree, so that our *Player* has something to collide with.

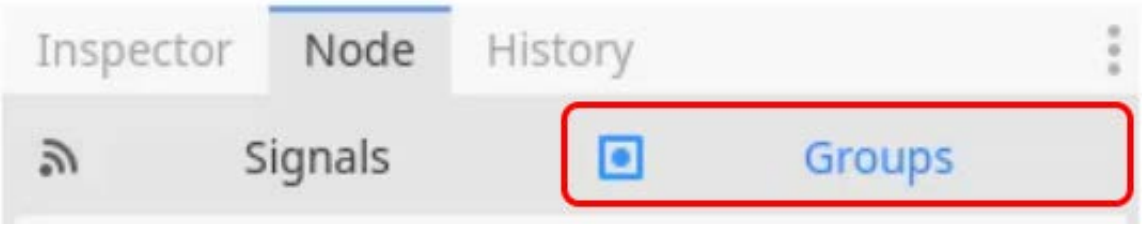


This will be a **CylinderShape3D**, which we can fit to the bounds of our tree trunk.

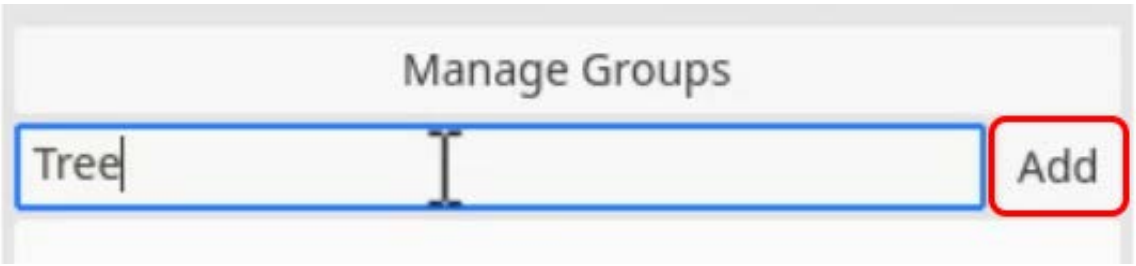


We then add a group to our tree node, we will do this in the *Node* tab with the *Tree* node selected. We will use a group to add a label to our tree so that we can detect if it is a tree our *Player* has hit.

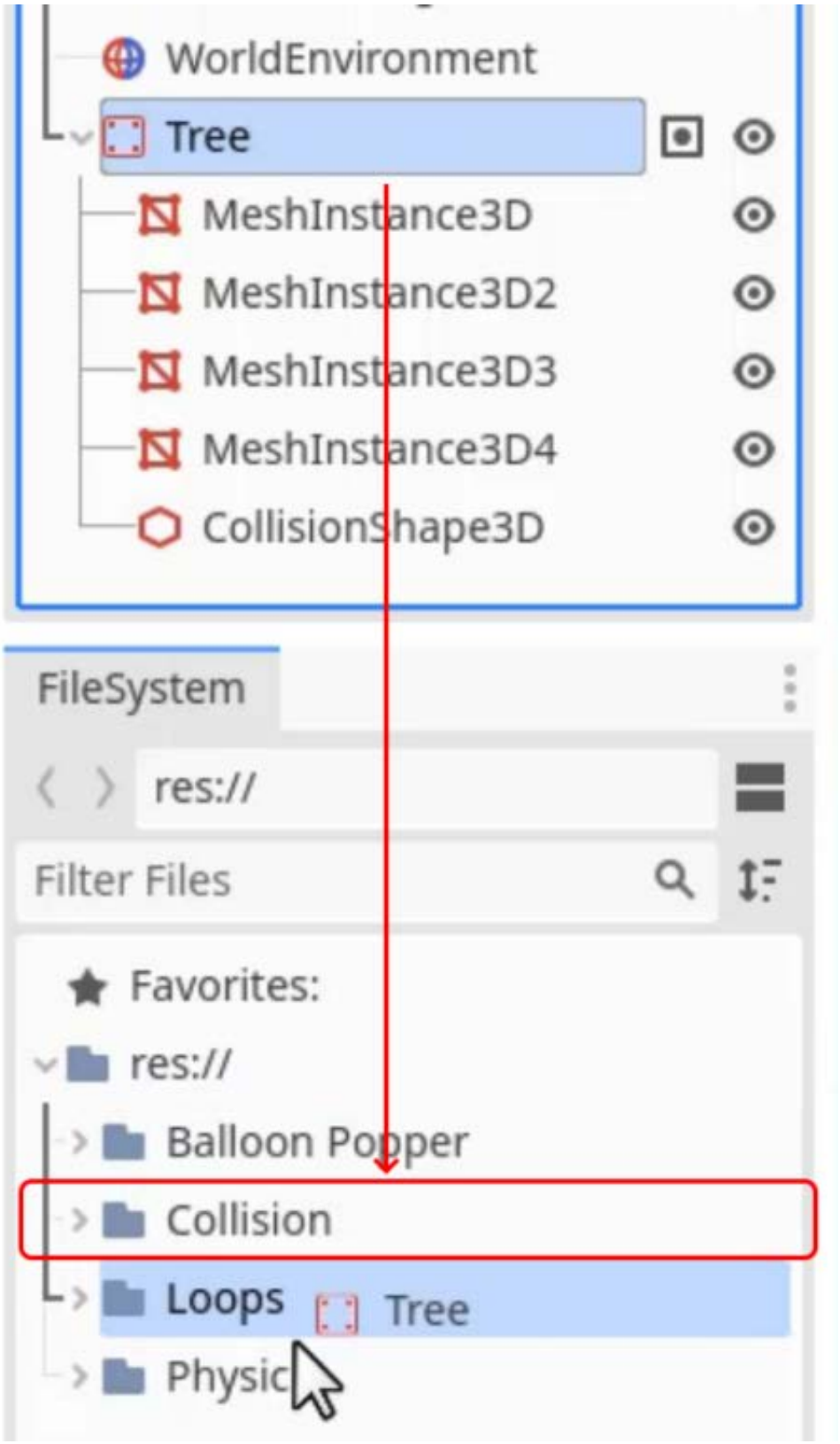
This will allow us to detect the difference between the *Tree's* and the *Ground's StaticBody3D* nodes.



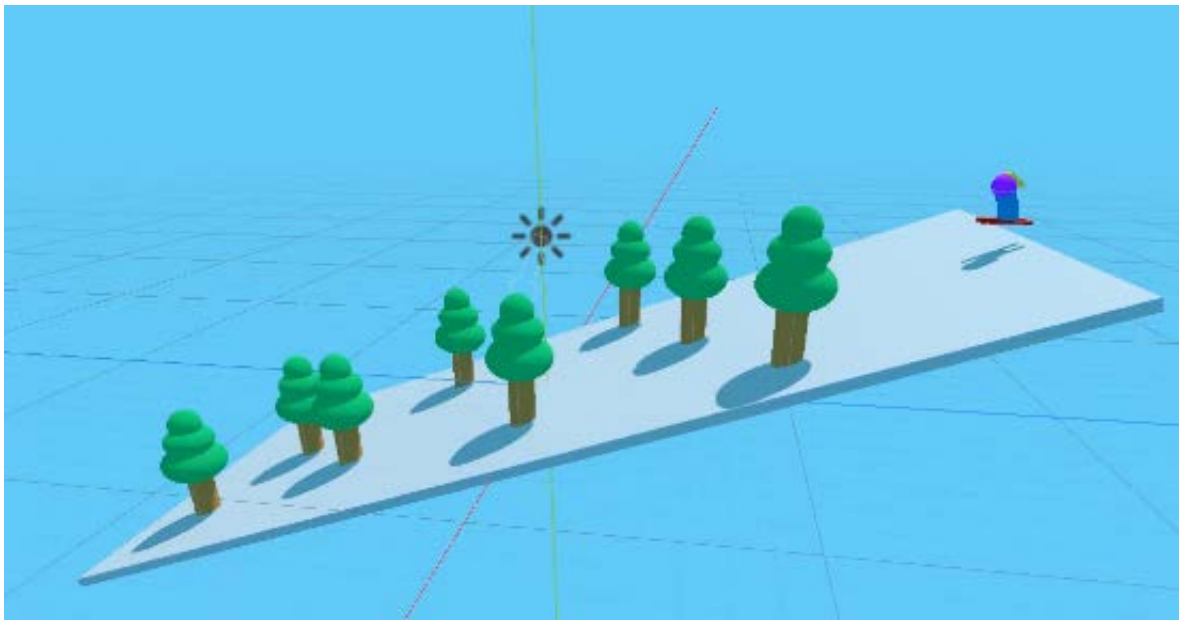
We will add a new group to our node with the name *Tree*.



Finally, we will drag the *Tree* node into the *FileSystem* to turn it into a scene, allowing us to edit multiple instances at once. Remember to save the scene in the *Collision* folder. We will be using the name *Tree.tscn.*

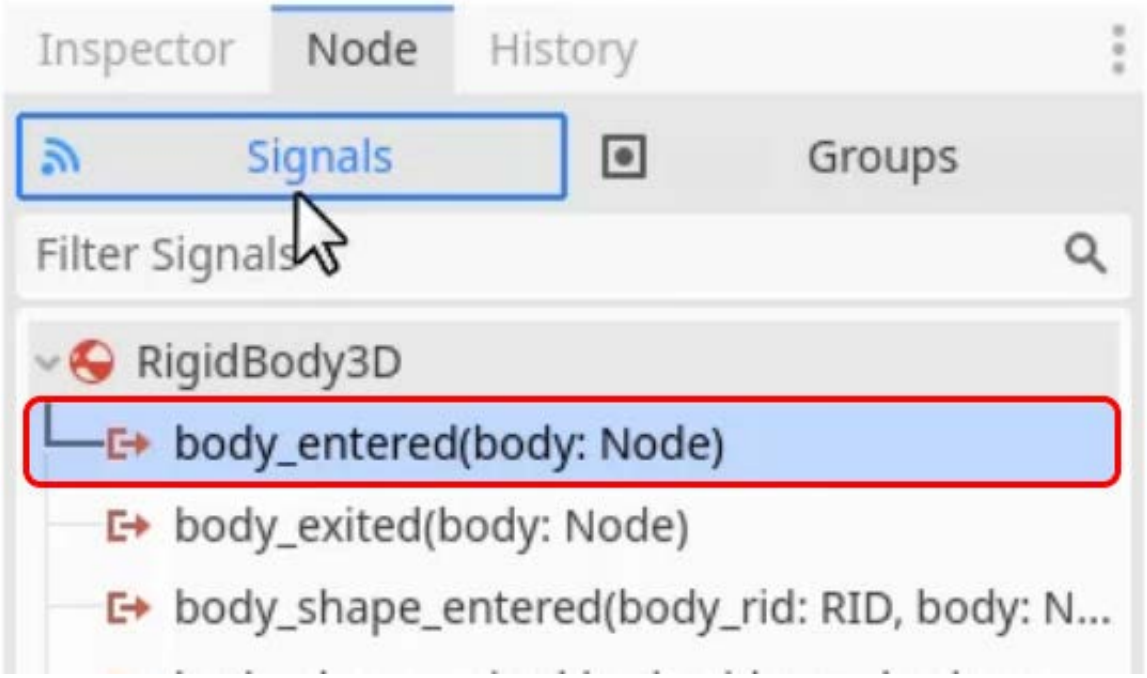


You can now **duplicate** (**CTRL+D**) the obstacles around the level, to give the *Player* something to dodge as they slide down the slope.

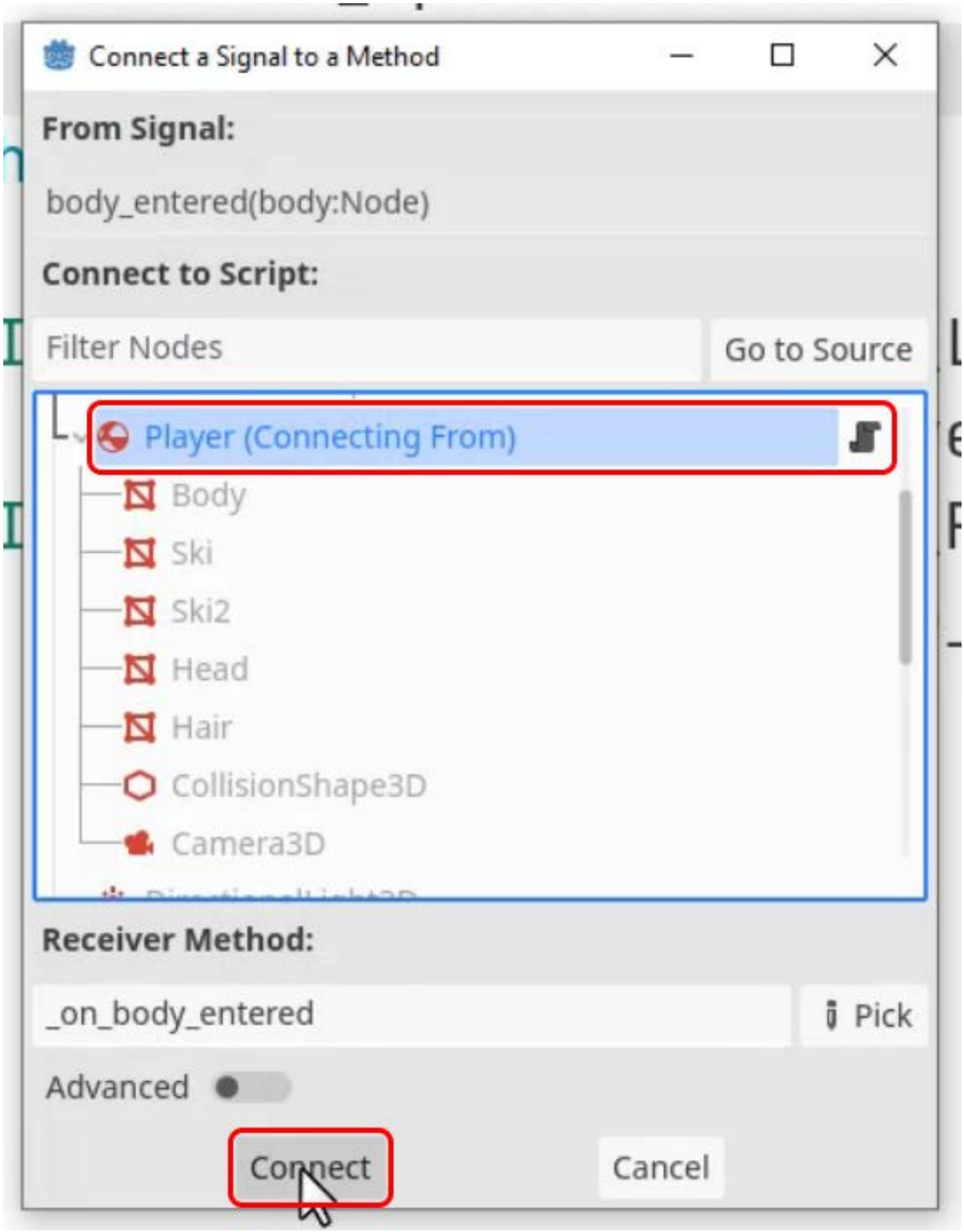


Detecting Collisions

Currently, if you press *Play* you will notice when you touch a tree nothing happens. To fix this, we will select our *Player* rigid body node and go to the **Signals** tab. We can then connect the **body_entered** signal to our *Player* node. A *signal* is a way of calling a function when an event happens, in this case when the *Player* makes a collision with another object.



In the pop-up window, make sure the *Player* node is selected and press **Connect**.



Inside the new *_on_body_entered* function, we can now check to see if the body that we have collided with is within the *Tree* group. If so, we then reload the scene.

```
func _on_body_entered(body):
    if body.is_in_group("Tree"):
        get_tree().reload_current_scene()
```

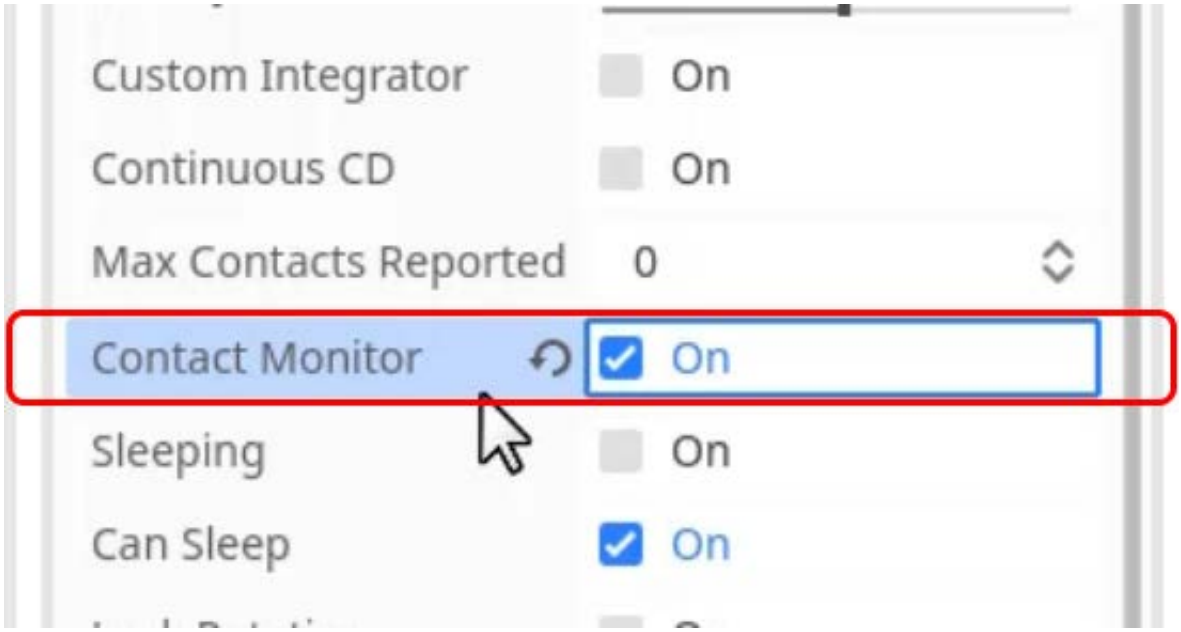

The following code has been updated, and differs from the video:

Calling the **reload_current_scene()** method directly causes the following error: “Removing a CollisionObject node during a physics callback is not allowed and will cause undesired behavior. Remove with call_deferred() instead”. To fix this, we would need to use Callable.call_deferred().

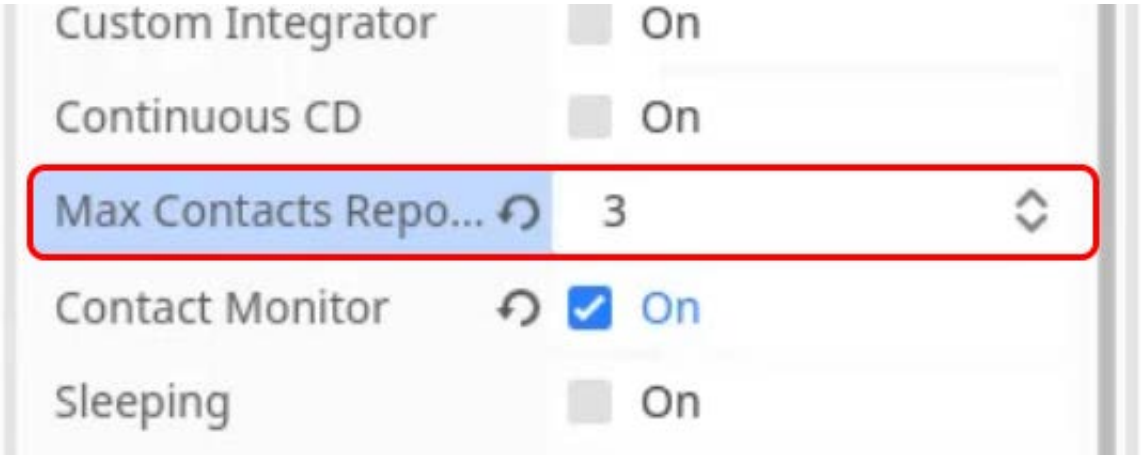
```
func _on_body_entered(body):
  if body.is_in_group("Tree"):
    get_tree().reload_current_scene.call_deferred()
```

Enabling Contact Monitoring

If you press *Play* now, you will notice nothing happens still. This is because we need to enable the **Contact Monitoring** property on our *Player* node.



This will allow us to detect contacts with other colliders. We will also change **Max Contacts Reported** to **3**. This will mean we can track multiple collisions at once, which is necessary as we will have one collision with the ground so we need more than one collision report at a time to be able to hit the trees.



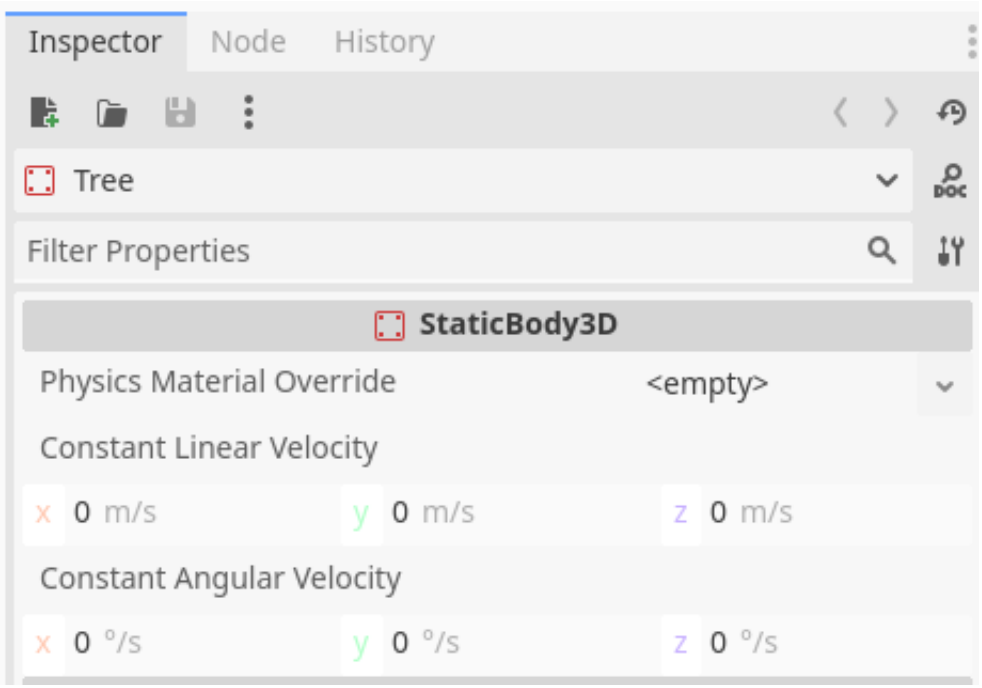
Now if you press **Play** you will see you can ski down the hill and collide with the *Trees*, which will reset the *Player* to the top of the slope. As a challenge, feel free to add a collider at the end to detect when the *Player* has reached the end of the slope.

That is everything for this mini-game, which has taught us to add colliders to our objects, detect collisions between the objects, and check the type of an object by the groups applied to it.

We just looked at setting up collisions in Godot. Now let's explore further.

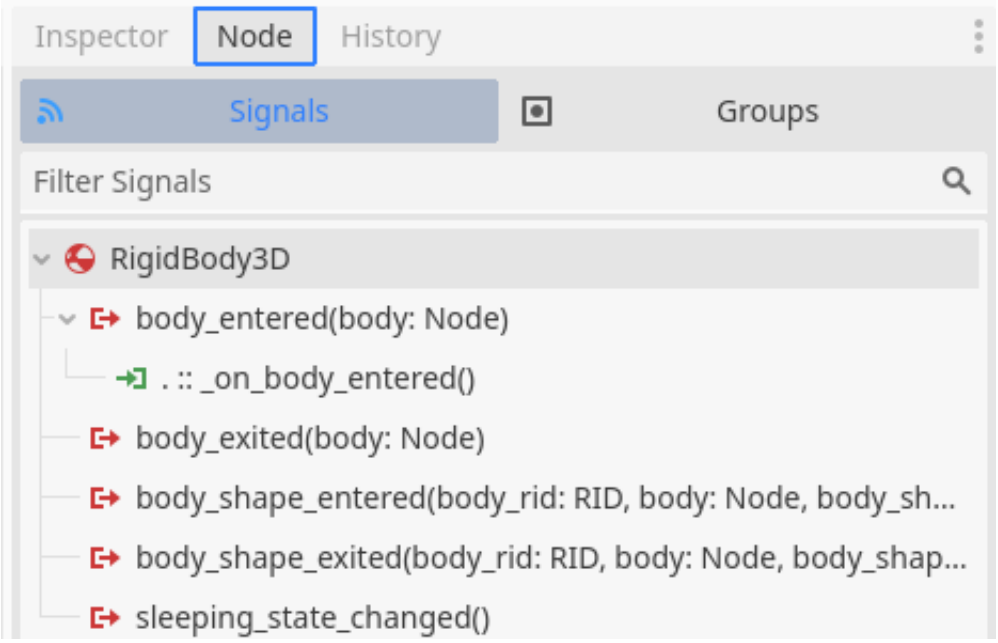
StaticBody3D

This node is a 3D physics body that cannot be moved by external forces. Already in this course, we have explored the RigidBody node, which is a physics body that *can* be affected by external forces. In order for us to have solid surfaces that we can bounce and move along, they must be defined as static bodies.

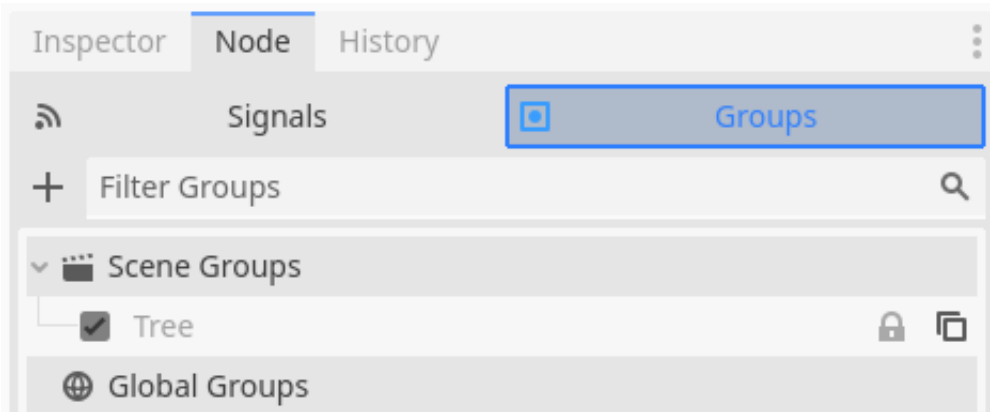


Collision Detection

To detect collisions, you need to connect to the appropriate signal. On our rigid body player, we are connecting to its **body_entered** signal. This gets called when another physics body connects or enters out collider. Likewise, there is the body_exited signal for when the opposite happens.



Now we don't want to reload the scene when the player collides with *any* collider – only the trees. So for our tree nodes, we assign it a **group** called "Tree". Groups are essentially a way to organize nodes for easy access and identification in scripts. For example, you might want all your enemy nodes to be in the "Enemy" group, so the player knows what it can attack.



We then can fill in our **_on_body_entered** function which is connected to the appropriate signal. Here, we are first checking if the body we entered is of the group "Tree". If so, we then reload the scene.

```
func _on_body_entered(body):
    if body.is_in_group("Tree"):
        get_tree().reload_current_scene()
```

Additional Resources

If you wish the learn more, you can refer to the Godot documentation.

- StaticBody3D
- Collision Detection
- body_entered Signal

Congratulations on completing the course. We have just finished creating four mini-projects inside the Godot Engine.

Balloon Popper Mini-Game

The first project we created was the *Balloon Popper mini-game* where we detected clicks using the input function. We then check to see if a *Balloon* was being clicked, and if it was, increase the size. If a *Balloon* got too big we would then pop it and increase our score. The score value would then be displayed on a *UI Label*.

Physics Game

The next project we worked on was a *Physics Game* that explored Godot's physics system. We looked at the *Rigidbody* node and the properties that came with that node. We added an impulse to the player in the direction of the player, when the user clicks, and this would allow us to crash into the boxes. This covered many of the physics principles inside Godot, and they can all translate very well to 3D as well.

Loops

Next, we looked at *loops*. Loops are a fundamental part of programming that will be used time and time again when you are creating games. We used loops to generate a random starfield giving stars a random position and size with an effectively infinite number of stars we could generate.

Skiing Game

The final project we created was a *Skiing Game* that covered *collision detection* inside Godot. This was a game where you skied down a small hill, and if you collided with a tree your position would be reset. We did this using the *Body Entered signal* on our player to check for collisions to see if they are in the group of *trees*.

Thank you very much for following along with the course, and we wish you the best of luck with your future Godot games!

About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

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

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

Balloon.gd

Found in Project/Balloon Popper

This script controls a 3D area in Godot, a “balloon”, allowing it to be clicked a set number of times before disappearing. With each click, the area increases in size. Once the area has been clicked the defined number of ‘clicks_to_pop’ times, it adds a defined score to the main game and then removes itself from the game.

```
extends Area3D

@export var clicks_to_pop : int = 5
@export var size_increase : float = 0.2
@export var score_to_give : int = 1

func _on_input_event(camera, event, position, normal, shape_idx):
  if event is InputEventMouseButton and event.button_index == MOUSE_BUTTON_LEFT and e
vent.pressed:
    scale += Vector3.ONE * size_increase
    clicks_to_pop -= 1

    if clicks_to_pop == 0:
      get_node("/root/Main").increase_score(score_to_give)
      queue_free()
```

BalloonManager.gd

Found in Project/Balloon Popper

This script acts as a score manager for a 3D Game (balloon) in Godot. It holds a score value which can be increased through the ‘increase_score’ function, and updates the displayed score on a user interface label each time the score changes.

```
extends Node3D

var score : int = 0
@export var score_text : Label

func increase_score (amount):
  score += amount
  score_text.text = str("Score: ", score)
```

Loops.gd

Found in Project/Loops

This code adds, or “spawns,” a certain number of star nodes (defined by `spawn_count`) to a 2D game scene in Godot. Each star’s appearance and position is randomly generated: their x and y coordinates, and their size all vary.

```
extends Node2D

@export var spawn_count : int = 200
var star_scene = preload("res://Loops/Star.tscn")

# Called when the node enters the scene tree for the first time.
func _ready():
  for i in spawn_count:
    var star = star_scene.instantiate()
    add_child(star)

    star.position.x = randi_range(-280, 280)
    star.position.y = randi_range(-150, 150)

    var star_size = randf_range(0.5, 1.0)
    star.scale.x = star_size
    star.scale.y = star_size
```

PhysicsPlayer.gd

Found in Project/Physics

This code, for a 2D physics-enabled node in Godot, applies an impulse towards the mouse’s position whenever the left mouse button is pressed. The strength of the impulse is determined by `hit_force`.

```
extends RigidBody2D

var hit_force : float = 50.0

func _process(_delta):
  if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
    var dir = global_position.direction_to(get_global_mouse_position())
    apply_impulse(dir * hit_force)
```

PlayerMovement.gd

Found in Project/Collision

This script controls the movement of a 3D rigid body in Godot. It responds to left and right arrow key presses to move at a defined speed. If the body collides with a body in the “Tree” group, the current scene is reloaded.

```
extends RigidBody3D

@export var move_speed : float = 2.0
```



```
func _physics_process(_delta):

  if Input.is_key_pressed(KEY_LEFT):
    linear_velocity.x = -move_speed
  if Input.is_key_pressed(KEY_RIGHT):
    linear_velocity.x = move_speed

func _on_body_entered(body):
  if body.is_in_group("Tree"):
    get_tree().reload_current_scene.call_deferred()
```

