A STEP-BY-STEP GUIDE TO CODING ADVANCED GAMES

GODOT FROM ZERO TO PROFICIENCY

(ADVANCED)

PATRICK FELICIA

Godot From Zero to proficiency (Advanced)

FIRST EDITION

USE ADVANCED TECHNIQUES TO BUILD BOTH 2D AND 3D GAMES

Table of Contents

Title Page

Patrick Felicia

Godot From Zero to Proficiency | (Advanced)

Credits

About the Author

Support and Resources for this Book

Table of Contents

Preface

Content Covered by this Book

What you Need to Use this Book

Who this Book is not for

How you will Learn from this Book

Format of each Chapter and Writing Conventions

How Can You Learn Best from this Book

Feedback

Downloading the Solutions for the Book

Improving the Book

Supporting the Author

Chapter 1: Reading Files and Creating Scenes Procedurally

Chapter 2: Accessing and Updating a Database

Chapter 3: Creating a Networked Multiplayer Game

Chapter 4: Creating a Memory Game

Chapter 6: Creating a Platformer

Chapter 6: Frequently Asked Questions

Chapter 7: Thank you

About the Author

Patrick Felicia

Godot From Zero to Proficiency

(Advanced)

First published: September 2021

Credits

Author: Patrick Felicia

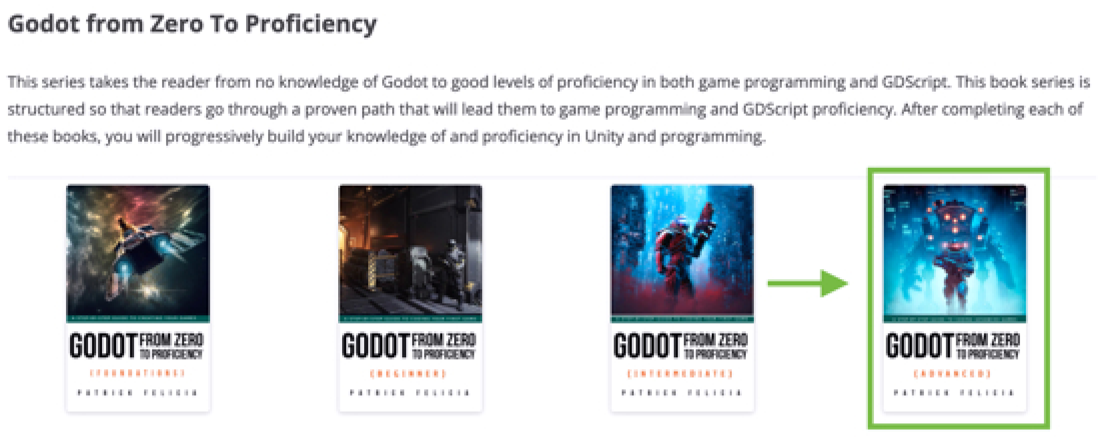
About the Author

Patrick Felicia is a lecturer and researcher at Waterford Institute of Technology, where he teaches and supervises undergraduate and postgraduate students. He obtained his MSc in Multimedia Technology in 2003 and Ph.D. in Computer Science in 2009 from University College Cork, Ireland. He has published several books and articles on the use of video games for educational purposes, including the Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches (published by IGI), and Digital Games in Schools: a Handbook for Teachers, published by European Schoolnet. Patrick is also the Editor-in-chief of the International Journal of Game-Based Learning (IJGBL), and the Conference Director of the Irish Conference on Game-Based Learning, a popular conference on games and learning organized throughout Ireland.

Support and Resources for this Book

To complete the activities presented in this book you need to download the startup pack on the companion website; it consists of free resources that you will need to complete your projects. To download these resources, please do the following:

- Open the page **http://www.learntocreategames.com/books**.
- Click on your book (**Godot From Zero to Proficiency (Advanced)**)

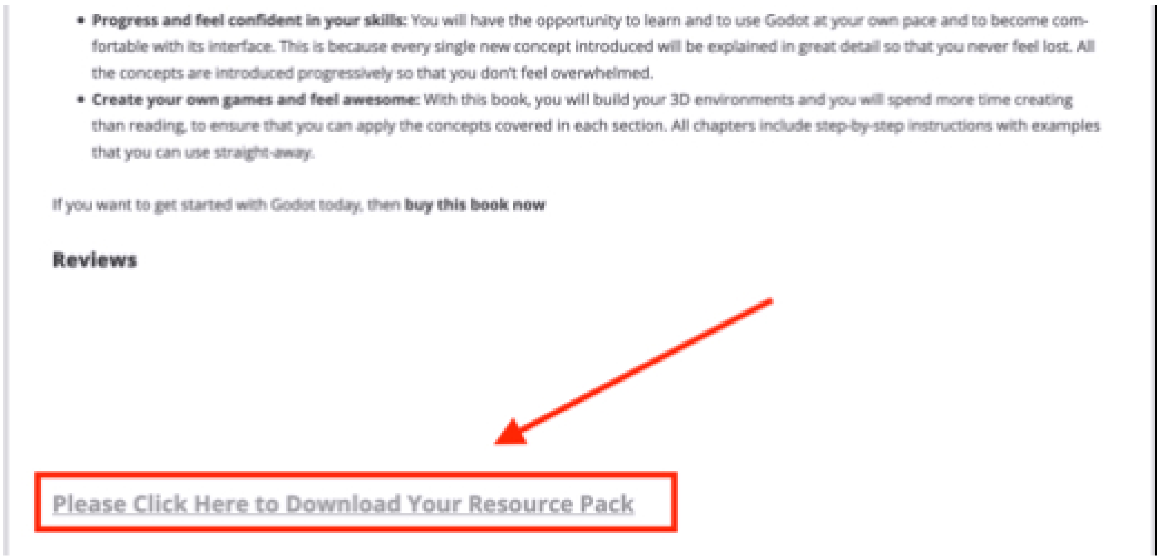


Godot from Zero To Proficiency

This series takes the reader from no knowledge of Godot to good levels of proficiency in both game programming and GDScript. This book series is structured so that readers go through a proven path that will lead them to game programming and GDScript proficiency. After completing each of these books, you will progressively build your knowledge of and proficiency in Unity and programming.

GODOT FROM ZERO TO PROFICIENCY
(FOUNDATIONS)
GODOT FROM ZERO TO PROFICIENCY
(BEGINNER)
GODOT FROM ZERO TO PROFICIENCY
(INTERMEDIATE)
GODOT FROM ZERO TO PROFICIENCY
(ADVANCED)

- On the new page, please click the link that says "**Please Here Click to Download Your Resource Pack**"



- **Progress and feel confident in your skills:** You will have the opportunity to learn and to use Godot at your own pace and to become comfortable with its interface. This is because every single new concept introduced will be explained in great detail so that you never feel lost. All the concepts are introduced progressively so that you don't feel overwhelmed.
- **Create your own games and feel awesome:** With this book, you will build your 3D environments and you will spend more time creating than reading, to ensure that you can apply the concepts covered in each section. All chapters include step-by-step instructions with examples that you can use straight-away.

If you want to get started with Godot today, then **buy this book now**

Reviews

Please Click Here to Download Your Resource Pack

This book is dedicated to Mathis

[ ]

Table of Contents

Chapter 1: Reading Files and Creating Scenes Procedurally

Building your environment from an array

Creating an environment from a text file

Using JSON files for content creation

Creating a virtual solar system based on a JSON file

Level Roundup

Chapter 2: Accessing and Updating a Database

Introduction to online databases

Accessing a database through PHP

Passing data to a PHP script

Accessing PHP from Godot

Setting up your server

Creating new tables

Gathering data from Godot

Level Roundup

Chapter 3: Creating a Networked Multiplayer Game

Introduction

Creating a single-player scene

Implementing a multiplayer-networked mode

Level Roundup

Chapter 4: Creating a Memory Game

Introduction
Creating the interface and the core of the game
Detecting when buttons have been pressed
Creating different states for our game
Playing a sequence of colors
Creating a new sequence of colors
Waiting for the user's input
Processing the user's input
Generating sound effects
Level Roundup

Chapter 6: Creating a Platformer

Introduction
Creating the main character
Moving the character
Making it possible for the player to collect items
Adding sound effects
Adding a dead zone to detect when the player has fallen
Allowing the player character to climb a ladder
Creating magic doors
Creating a scene using tilesets
Creating a parallax effect
Level Roundup

Chapter 6: Frequently Asked Questions

Networking
Accessing databases

Reading files

Chapter 7: Thank you

Preface

This book will show you how you can very quickly use GDScript to implement some advanced features that will improve your games.

Although it may not be as powerful as Unity or Unreal yet, Godot offers a wide range of features for you to create your video games. More importantly, this game engine is both Open Source and lightweight which means that even if you have (or you are teaching with) computers with very low technical specifications, you should still be able to use Godot, and teach or learn how to code while creating video games.

This book series entitled **Godot From Zero to Proficiency** allows you to play around with Godot's core features, and essentially those that will make it possible to create interesting 3D and 2D games rapidly. After reading this book series, you should find it easier to use Godot and its core functionalities, including programming with GDScript.

This book series assumes no prior knowledge on the part of the reader, and it will get you started on Godot so that you quickly master all the wonderful features that this software provides by going through an easy learning curve.

By completing each chapter, and by following step-by-step instructions, you will progressively improve your skills, become more proficient in Godot, and create features that will improve your productivity and gameplay.

Throughout this book series, you will create a game that includes environments where the player needs to find its way out of the levels, escalators, traps, and other challenges, avoid or eliminate enemies using weapons (i.e., guns or grenades), and drive a car or pilot an aircraft.

You will learn how to create customized menus and simple user interfaces using Godot's UI system, and animate and give artificial intelligence to Non-Player Characters (NPCs) that will be able to follow the player character using pathfinding.

Finally, you will also get to export your game at the different stages of the books, so that you can share it with friends and obtain some feedback as well.

Content Covered by this Book

Chapter 1, *Reading Files and Creating Scenes* Procedurally, gets you to create your scenes fast using a wide range of advanced techniques so that you can create your scene using text or image files and read them at run time without the need to add objects manually to your scene. It is also illustrated how these techniques can be used for data visualization by combining advanced GDScript coding and JSON files.

Chapter 2, *Accessing and Updating a Database*, helps you to understand how to transfer data between Godot and a database. It explains how to create your database, and how to access it through the use of a combination of GDScript, PHP and MySQL.

Chapter 3, *Creating a Networked multiplayer game*, explains and illustrates how you can create a simple network game with Godot. You will use Godot's built-in networking features and create a simple game two players have to remotely compete to eat as many pills as possible before their opponent.

Chapter 4, *Chapter 4: Creating* a Memory Game, shows you how to create a game where the player has to memorize and to play an increasing sequence of colors and sounds, in a similar way as the **Simon** game that was popular in the 80s. You will learn how to create and generate audio within Godot and change the sounds' frequency, to detect when a player has pressed a button, to generate colors at random, and also to record the sequence entered by the player and then compare it to the correct sequence

Chapter 5, *Creating a Platformer*, explains how you can create a platform game with most of the options commonly found in this genre. You will learn how to move and animate a 2D character, create a parallax effect, use tilesets to speed up the design process, and many more features that will make your game even more entertaining..

Chapter 6 provides answers to Frequently Asked Questions (FAQs) related to the topics covered in this book (e.g., networking, databases or procedural content generation).

Chapter 7 summarizes the topics covered in the book and provides you with more information on the next steps.

What you Need to Use this Book

To complete the project presented in this book, you only need Godot 3.2.4 or a more recent version and to also ensure that your computer and its operating system comply with Godot's requirements. Godot can be downloaded from the official website **(http://www.godotengine.org/download)**, and before downloading, you can check that your computer fulfills the requirements for Godot on the same page.

At the time of writing this book, the following operating systems are supported by Godot for development: Windows, Linux and Mac OS X.

In terms of computer skills, all knowledge introduced in this book will assume no prior programming experience from you. This book includes programming and all programming concepts will be taught from scratch. So for now, you only need to be able to perform common computer tasks such as downloading files, opening and saving files, be comfortable with dragging and dropping items, and typing.

If you can answer **yes** to all these questions, then this book is for you:

1. Do you want to learn advanced techniques for Godot and GDScript?
2. Would you like to become proficient in the core functionalities offered by Godot?
3. Would you like to start creating great 2D games?
4. Would you like to design and code games faster?
5. Although you may have had some prior exposure to Godot, would you like to delve more into Godot and understand its core functionalities in more detail?

Who this Book is not for

If you can answer yes to all these questions, then this book is **<u>not</u>** for you:

1. Can you already code with GDScript to implement procedural levels?
2. Can you already easily access and update a database using GDScript?
3. Are you looking for a reference book on GDScript?
4. Are you an experienced (or at least advanced) Godot user?

If you can answer yes to all four questions, you may instead look for the next books in the series. To see the content and topics covered by these books, you can check the official website (**www.learntocreategames.com/books**).

How you will Learn from this Book

Because all students learn differently and have different expectations of a course, this book is designed to ensure that all readers find a learning mode that suits them. Therefore, it includes the following:

- A list of the learning objectives at the start of each chapter so that readers have a snapshot of the skills that will be covered.
- Each section includes an overview of the activities covered.
- Many of the activities are step-by-step, and learners are also given the opportunity to engage in deeper learning and problem-solving skills through the challenges offered at the end of each chapter.
- Each chapter ends up with a quiz and challenges through which you can put your skills (and knowledge acquired) into practice, and see how much you know. Challenges consist of coding, debugging, or creating new features based on the knowledge that you have acquired in the chapter.
- The book focuses on the core skills that you need; some sections also go into more detail; however, once concepts have been explained, links are provided to additional resources, where necessary.
- The code is introduced progressively and is explained in detail.
- You also gain access to several videos that help you along the way, especially for the most challenging topics.

Format of each Chapter and Writing Conventions

Throughout this book, and to make reading and learning easier, text formatting and icons will be used to highlight parts of the information provided and to make it more readable.

SPECIAL NOTES

Each chapter includes resource sections, so that you can further your understanding and mastery of Godot; these include:

- A quiz for each chapter: these quizzes usually include 10 questions that test your knowledge of the topics covered throughout the chapter. The solutions are provided on the companion website.
- A checklist: it consists of between 5 and 10 key concepts and skills that you need to be comfortable with before progressing to the next chapter.
- Challenges: each chapter includes a challenge section where you are asked to combine your skills to solve a particular problem.

Author's notes appear as described below:

Author's suggestions appear in this box.

Code appears as described below:

```
var score = 100
var player_name = "Sam"
```

Checklists that include the important points covered in the chapter appear as described below:

- Item1 for checklist
- Item2 for checklist
- Item3 for checklist

How Can You Learn Best from this Book

- **Talk to your friends about what you are doing.**

  We often think that we understand a topic until we have to explain it to friends and answer their questions. By explaining your different projects, what you just learned will become clearer to you.

- **Do the exercises.**

  All chapters include exercises that will help you to learn by doing. In other words, by completing these exercises, you will be able to better understand the topic and gain practical skills (i.e., rather than just reading).

- **Don't be afraid of making mistakes.**

  I usually tell my students that making mistakes is part of the learning process; the more mistakes you make and the more opportunities you have for learning. At the start, you may find the errors disconcerting, or that the engine does not work as expected until you understand what went wrong.

- **Export your games early.**

  It is always great to build and export your first game. Even if it is rather simple, it is always good to see it in a browser and to be able to share it with you friends.

- **Learn in chunks.**

  It may be disconcerting to go through five or six chapters straight, as it may lower your motivation. Instead, give yourself enough time to learn, go at your own pace, and learn in small units (e.g., between 15 and 20 minutes per day). This will do at least two things for you: it will give your brain the time to "digest" the information that you have just learned, so that you can start fresh the following day. It will also make sure that you don't "burn-out" and that you keep your motivation levels high.

Feedback

While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback, you can email me at:

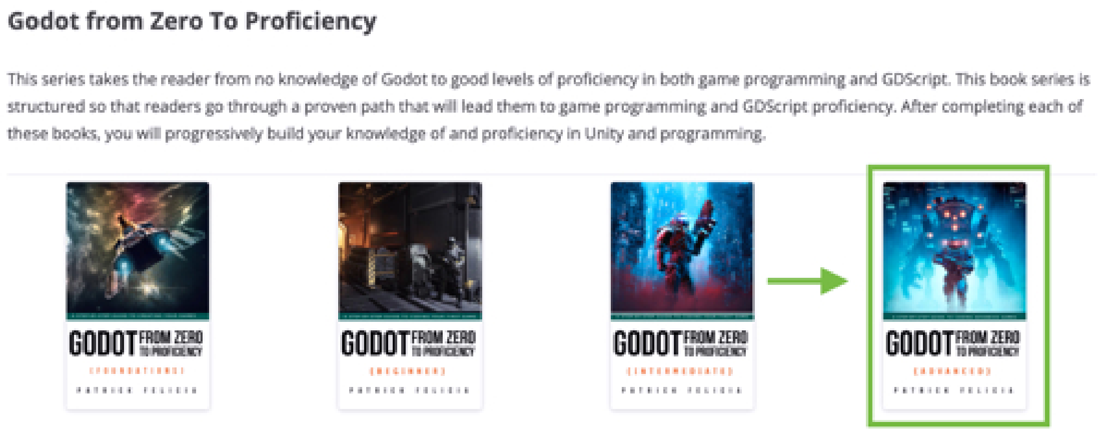
learntocreategames@gmail.com.

Downloading the Solutions for the Book

You can download the solutions for this book after creating a free online account at **http://learntocreategames.com/books/**. Once you have registered, a link to the files will be sent to you automatically.

To download the solutions for this book (e.g., code) you need to download the startup pack on the companion website; it consists of free resources that you will need to complete your projects and code solutions. To download these resources, please do the following:

- Open the page **http://www.learntocreategames.com/books**.
- Click on your book (**Godot From Zero to Proficiency – Advanced -**)

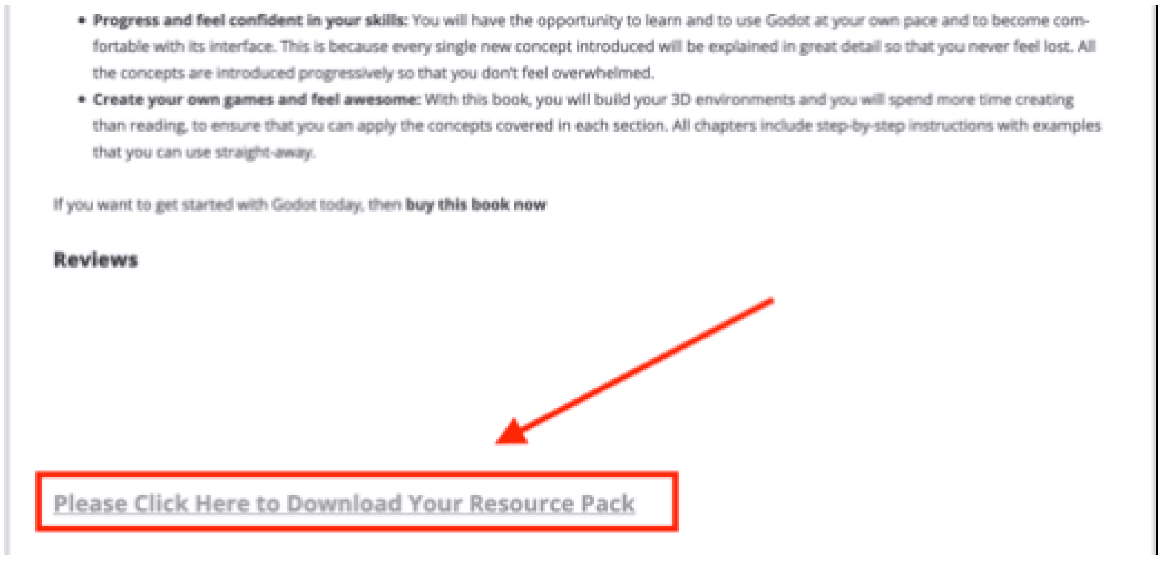


Godot from Zero To Proficiency

This series takes the reader from no knowledge of Godot to good levels of proficiency in both game programming and GDScript. This book series is structured so that readers go through a proven path that will lead them to game programming and GDScript proficiency. After completing each of these books, you will progressively build your knowledge of and proficiency in Unity and programming.

GODOT FROM ZERO TO PROFICIENCY
[FOUNDATIONS]
GODOT FROM ZERO TO PROFICIENCY
[BEGINNER]
GODOT FROM ZERO TO PROFICIENCY
[INTERMEDIATE]
GODOT FROM ZERO TO PROFICIENCY
[ADVANCED]

- On the new page, please click the link that says "**Please Here Click to Download Your Resource Pack**"



- **Progress and feel confident in your skills:** You will have the opportunity to learn and to use Godot at your own pace and to become comfortable with its interface. This is because every single new concept introduced will be explained in great detail so that you never feel lost. All the concepts are introduced progressively so that you don't feel overwhelmed.
- **Create your own games and feel awesome:** With this book, you will build your 3D environments and you will spend more time creating than reading, to ensure that you can apply the concepts covered in each section. All chapters include step-by-step instructions with examples that you can use straight-away.

If you want to get started with Godot today, then **buy this book now**

Reviews

Please Click Here to Download Your Resource Pack

Improving the Book

Although great care was taken in checking the content of this book, I am human, and some errors could remain in the book. As a result, it would be great if you could let me know of any issue or error you may have come across in this book, so that it can be solved and the book updated accordingly. To report an error, you can email me (**learntocreategames@gmail.com**) with the following information:

- Name of the book.
- The page or section where the error was detected.
- Describe the error and also what you think the correction should be.

Once your email is received, the error will be checked, and, in the case of a valid error, it will be corrected and the book page will be updated to reflect the changes accordingly.

Supporting the Author

A lot of work has gone into this book and it is the fruit of long hours of preparation, brainstorming, and finally writing. As a result, I would ask that you do not distribute any illegal copies of this book.

This means that if a friend wants a copy of this book, s/he will have to buy it through the official channels (i.e., your e-store), or the book's official website: **www.learntocreategames.com/books**).

If some of your friends are interested in the book, you can refer them to the book's official website (**http://www.learntocreategames.com/books**) where they can either buy the book, enter a monthly draw to be in for a chance of receiving a free copy of the book, or to be notified of future promotional offers.

[ ]

Chapter 1: Reading Files and Creating Scenes Procedurally

In this section, we will learn how to create your game levels from scripts and external files rather than by adding all objects manually to each scene. This will have the advantage of saving you a lot of time creating your levels; it will also make it easier to modify your levels relatively quickly too. Creating your environment from a script or "procedurally" can be achieved using a wide range of techniques from simple arrays to JSON files. So, after completing this chapter, you will be able to:

- Instantiate objects based on an array or a text file.
- Create a level from an array.
- Create multiple levels using simple text files.
- Create more complex scenes by reading and implementing the content of a JSON file.

Some of the skills you will also learn along the way include:

- Creating and accessing a file in your project where you can store and access resources for your game (e.g., images or text files).
- Reading text files from your game.
- Reading and parsing a JSON document.

Building your environment from an array

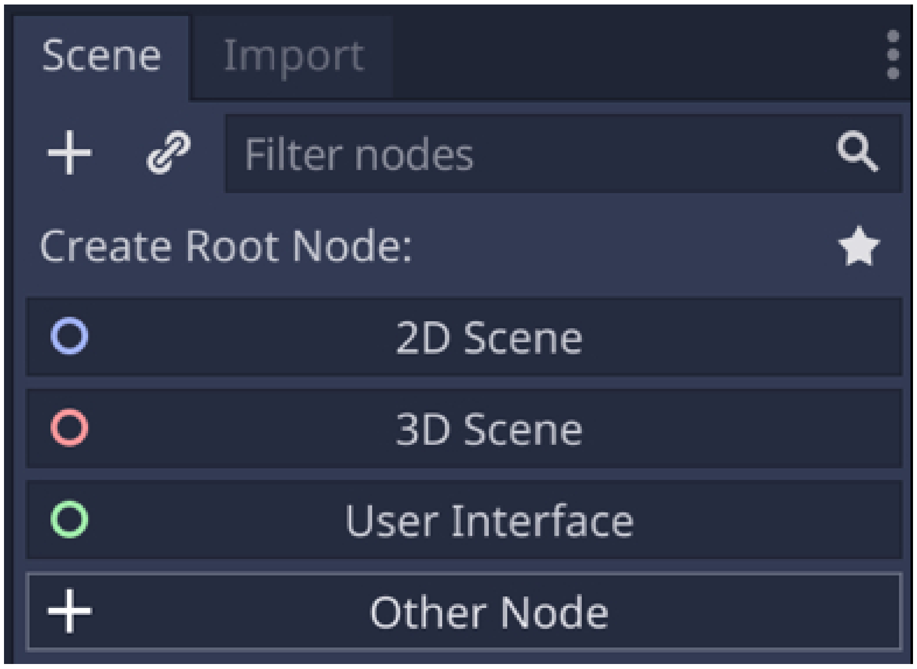
The first and simplest way to create a game environment procedurally is by using a simple array; so to setup our first procedural environment, we will generate an indoor level using a combination of GDScript and arrays.

We will proceed as follows:

- Create an array that represents the environment.
- Read the array.
- Instantiate objects based on the numbers read in the array.

So let's get started:

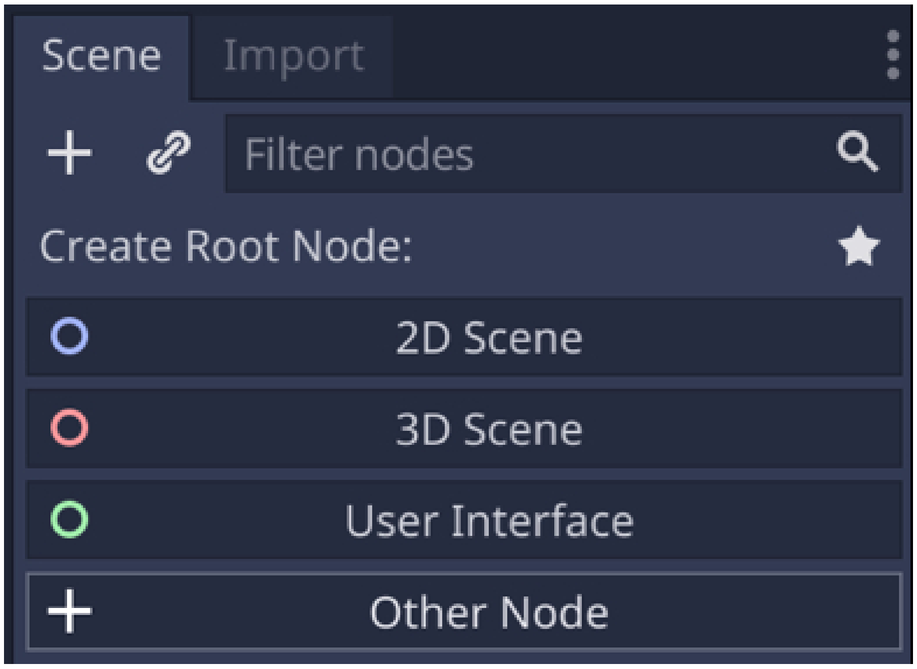
- Please open Godot.
- Open the start-up project located in the resources that you have downloaded.
- Create a new 3D scene: Select **Scene | New Scene**, and then select the option **3D Scene** in the new window.



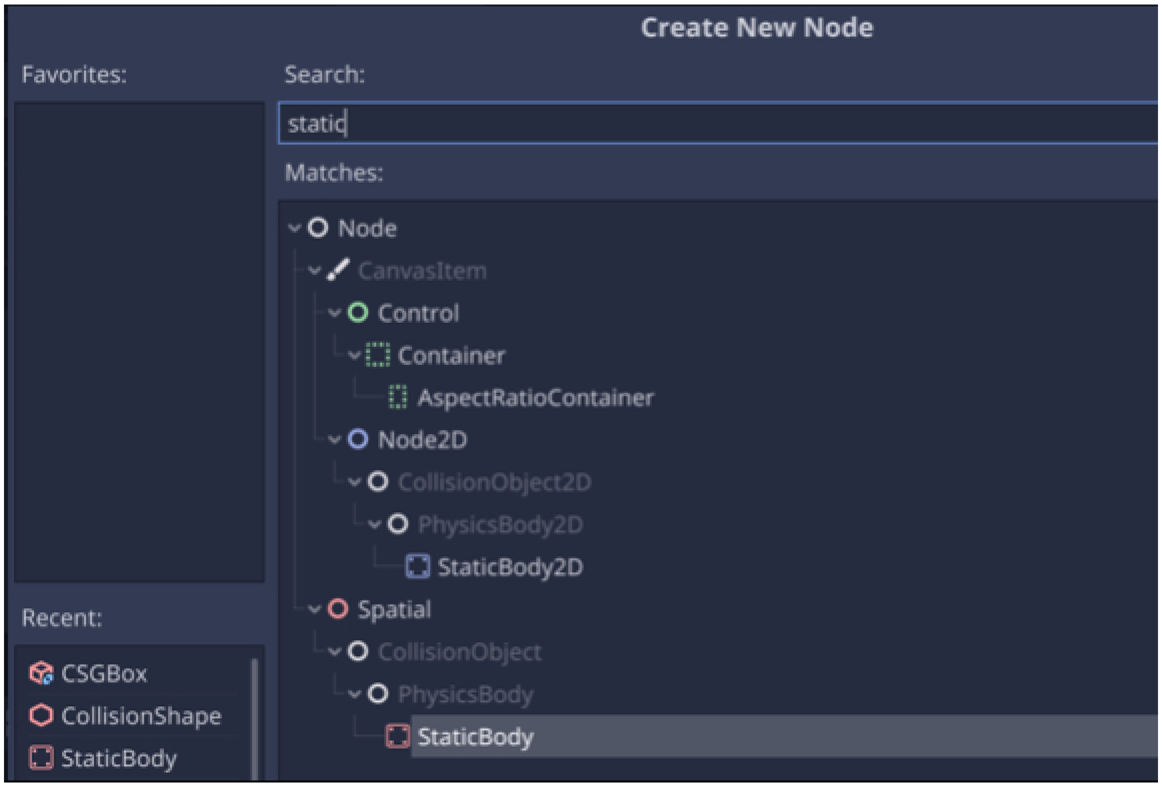
- Save the scene as **game_level_auto,** (**Scene | Save Scene As**), or any other name of your choice.
- Create a new **Spatial** node as a child of the existing **Spatial** node (i.e., right-click on this node, select the option **Add Child Node** and select the node type **Spatial**), and rename this new node **create_environment**.
- Create a **Camera** node as a child of the existing **Spatial** node (i.e., right-click on this node, select the option **Add Child Node** and select the node type **Camera**), change its rotation to **(-90, 0, 0)**, its position to **(0, 100, 0)** and its **FOV**, **Near** and **Far** features to **70**, **0.05**, and **100**, respectively.

Next, we will create an object that will be used to instantiate the walls.

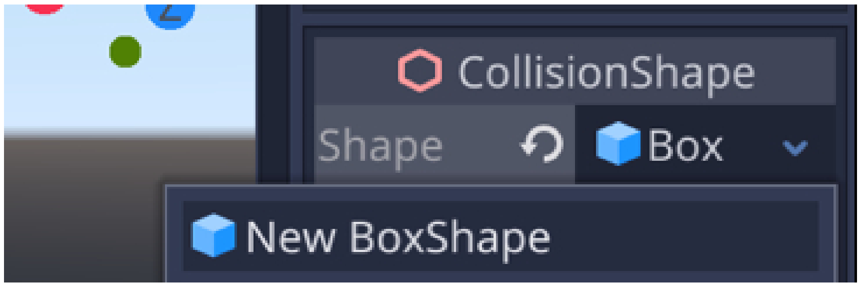
- Please save the current scene (**CTRL + S** or **CMD + S**).
- Create a new scene (**Scene | New Scene**).
- In the new window, select the option "**Other Node**".



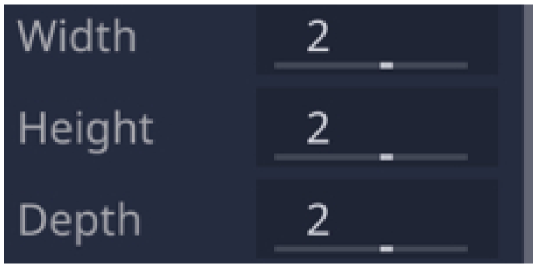
- In the new window, type the text "**static**" in the search field and select the type **StaticBody** from the list of results.



- This will create a new node called **StaticBody** in your 3D scene.
- Please rename this node "**wall**" and set its scale property to **(5, 5, 5)**.
- Add a **CollisionShape** node as a child of the node **wall**.
- Select this node (i.e., **CollisionShape**), and, using the **Inspector**, in the section called **CollisionShape**, click on the downward arrow to the right of the label **Shape** and select the option "**New Box Shape**".



- Finally, add a new node of type **CSGBox** to the node **CollisionShape**.
- Using the **Inspector**, and the section **CSGBox**, set its **Width**, **Height** and **Depth** to **2**.



Width 2
Height 2
Depth 2

- You can now save this scene as **wall.tscn** (**Scene | Save Scene As**).

Next, we will start to create a script that will generate our environment procedurally:

- Please switch to the scene **game_level_auto**.
- Please select the node **create_environment** and attach a new script to it (i.e., right-click on that node and select the option **Attach Script**).
- Rename the new script **generate_maze.gd**.
- Open this script and add the following code at the beginning of the script (new code in bold).

```
extends Spatial
export (PackedScene) var wall
```

```
var array
```

In the previous code, we declare a new variable called **wall** as a **PackedScene** using the keyword **export** which means that it will be a variable that will be available through the **Inspector**. In practical terms, we will be able to drag and drop the scene **wall.tscn** that we have just created to this empty slot, so that this scene can be used as a template to create the walls in our maze.

We also create a variable called **array**; this variable, which is an array, will store a combination of **1**s and **0**s, each defining where we should have a wall in our maze (represented by a **1**) or an empty space (represented by a **0**).

Next, please add the following code to the script:

```
func read_array():
array = [
1,1,1,1,1,1,1,1,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,0,0,1,0,1,0,1,1,
1,1,0,0,1,0,1,0,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,0,0,0,0,0,0,1,1,
1,1,1,1,1,1,1,1,1,1,
]
```

In the previous code:

- We declare a function called **read_array**.
- In this function we initialize the array called **array**.
- This array is mono-dimensional and includes a series of 1s and zeros.

- The values in the array are within opening and closing square brackets and are separated by commas.
- While all the data could have fitted on one line, we have here, for convenience, written 10 numbers per line; the idea behind this layout is that our maze will consist of 100 cells organized in 10 rows and 10 columns (i.e., **10 x 10**); so by writing 10 numbers by line, we can assess visually how the maze will look like. In our case, we can see that there will be walls on all sides of the maze (symbolized by **1**s).

Now that we have defined the structure and content of the maze, it is time to write the code that will translate and transform this array into actual walls, and hence create a 3D maze based on the array.

- Please add the following code after the code that you have previously included, in the same function, just after the declaration of the array, using proper indentation (new code in bold)

```
1,1,1,1,1,1,1,1,1,1,
]
for i in range (0,100,1):
var new_wall = wall.instance()
if (array[i] == 1):
add_child(new_wall)
var x_pos = 50 - (i/10) * 10
var z_pos = -(50 - (i%10) * 10)
new_wall.global_transform.origin = Vector3(x_pos,2,z_pos)
```

In the previous code:

- We create a loop that ranges from **0** to **99** (so we have **100** steps in total; the upper boundary of **100** is excluded, hence the range from 0 to **99** and not **100**).

- Within this loop, we instantiate a new copy of the template called **wall** and save it in the variable called **new_wall**.
- We then read the array and check whether the value of the array at the current index defined by the variable **i** is **1**.
- If that is the case, we then add the new wall (i.e., the node **new_wall**) as a child of the current node.
- We then calculate the **x** and **z** coordinates of this new wall.
- To calculate the position of the wall we need to (1) center it around the center of the maze, (2) know in what row it should be, and (3) know in which column it should be.
- Since the maze is 100 units wide and 100 units deep, its middle will be located at **(50,50)**.
- Because we are using a mono-dimensional array to represent the content of the maze that is **10 x 10** (**10 rows** and **10 columns**), and that every 10 consecutive number describes a specific row, we can deduct that the row for a wall is defined by **i/10** and the corresponding column will be **i%10**.

% is the modulo operator; when applied to two numbers a and b, **a%b** provides the remainder of the division between these two numbers; so while **2%2** is **0**, **3%2** is **1**.

- The **x** and **z** coordinates are based on the size of the actual wall (i.e., **10** x**10** x **10**) and the middle of the actual maze (i.e., **50, 50**). So, in essence, we center the wall based on the middle of the maze.

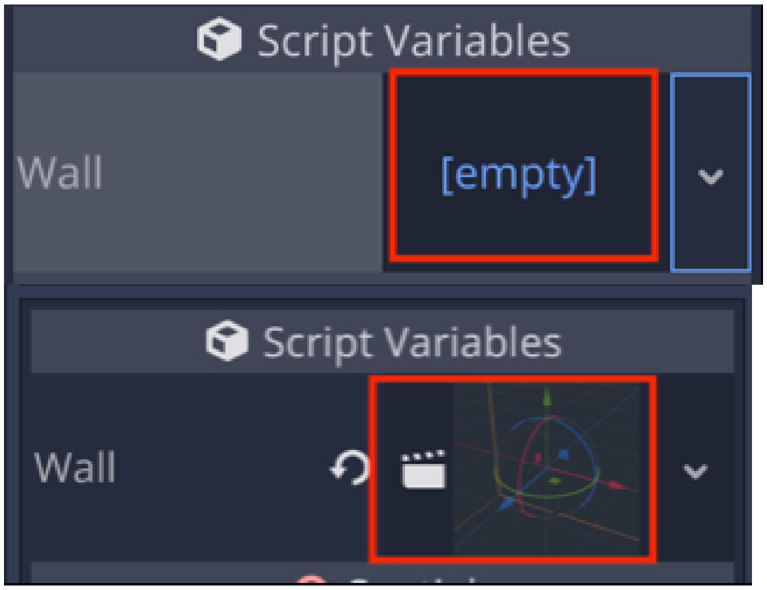
Finally, we just need to call this function from the **_ready** function; so please add the following code to the script (new code in bold).

```
func _ready():
**read_array()**
```

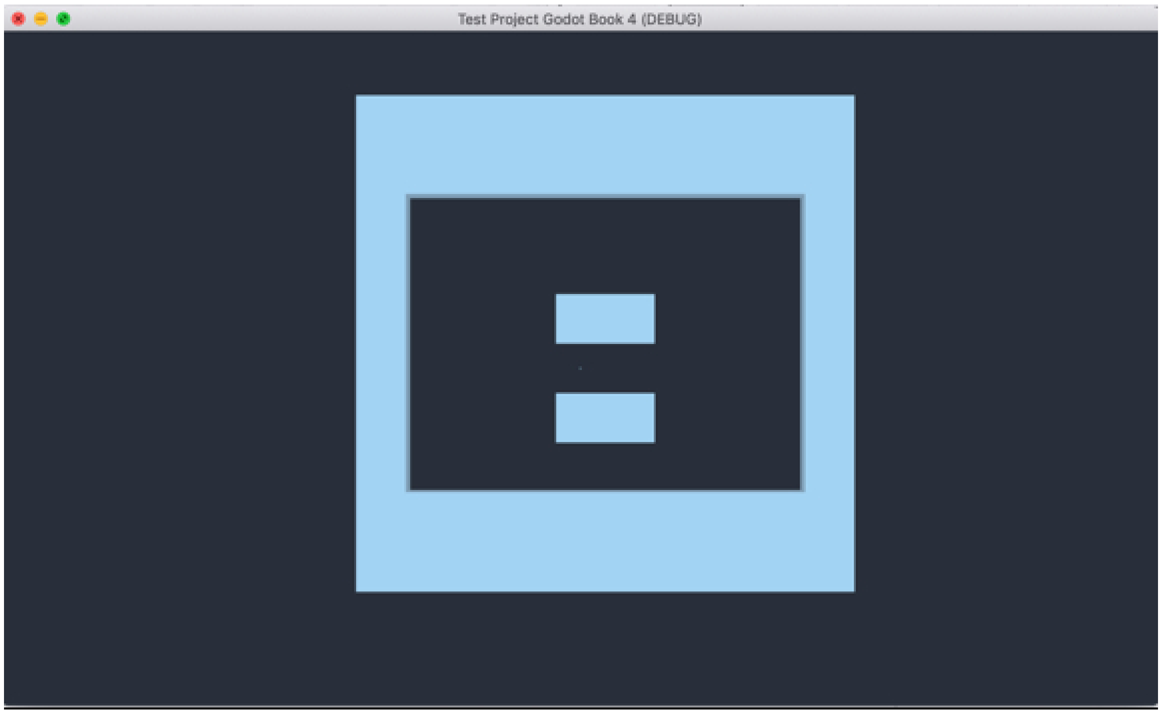
In the previous code, we use the function **_ready** to call the function **read_array** whenever the scene is loaded.

Now, we just need to finish our setup:

- Please save your script (**CTRL + S** or **CMD + S**).
- Return to the main scene (**game_level_auto**) if it is not the case already.
- Select the node **create_environment**.
- Drag and drop the file **wall.tscn** from the **File System** tab to the empty slot labeled **[empty]** in the **Inspector**, to the right of the label **Wall**.



- Once this is done, you can play the scene, and check the layout.


Test Project Godot Book 4 (DEBUG)

Creating an environment from a text file

Now that this works, we could take it a notch further by creating a file that includes all the information about the maze.

You see, using arrays is great; however, it may be more convenient to use a specific file for each level. This will at least do two things for you: (1) it will make it possible to modify the structure of the level without having to modify the code, and (2) it will make it possible to create (and load) individual files for each level.

So you could virtually create a text file for every level and then load it accordingly.

So, for this purpose, we will be using built-in functions that make it possible to read text files from your Godot project.

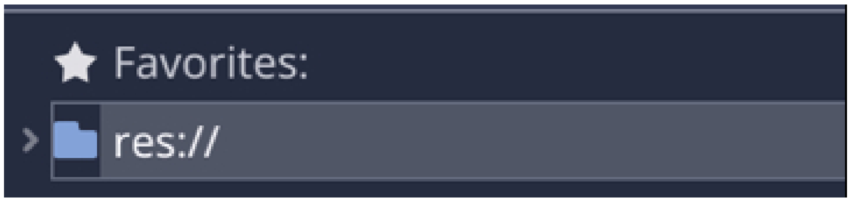
First, we will create such a text file and then access it through our script.

- Please create a new text file using the editor of your choice.
- Add the following text to it.

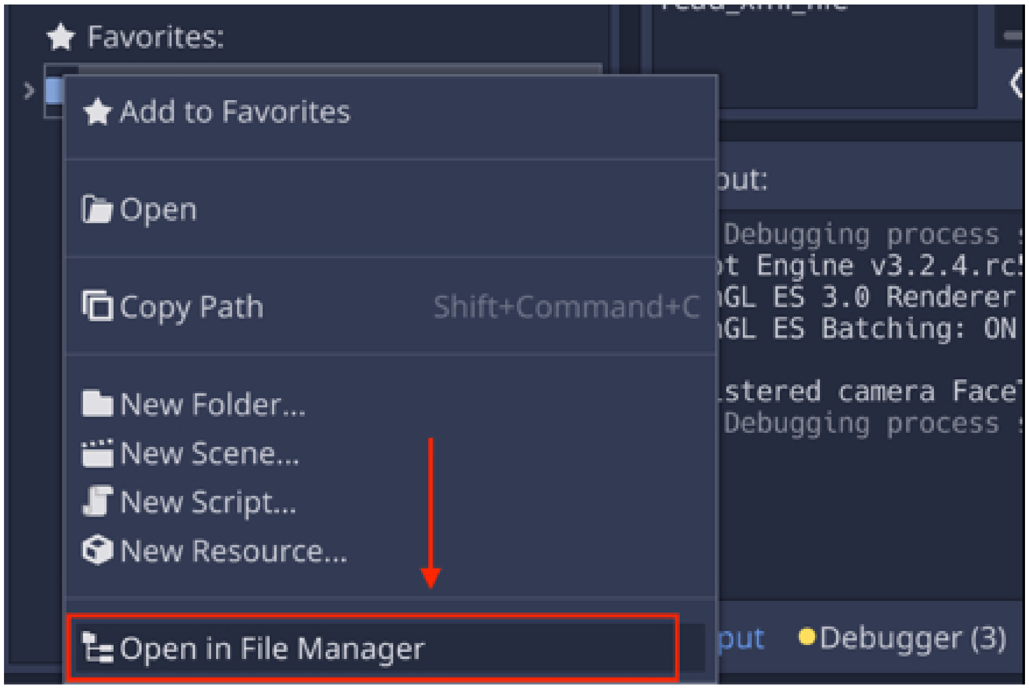
```
1111111111
1010000001
1010101001
1010000001
1011110001
1000000001
1010101111
1001000001
1010000001
1111111111
```

Once this is done, we will need to save this file inside your project; to do so, we need to identify the path to your project (or where your project is saved on your computer).

- Please switch to **Godot**.
- Using the **File System** window, right-click on the folder called **res://**.



- Select the option “**Open in File Manager**” from the contextual menu.



```
Debugging process :
ot Engine v3.2.4.rc!
nGL ES 3.0 Renderer
nGL ES Batching: ON
.stered camera Face
Debugging process :
```

- This will open the folder where all the project’s files are saved.
- Please take note of the location of this folder, return to your text editor and save the text file as **maze.txt** within the folder that you have just located.

Once this is done, we can modify our script so that it reads information from the text file rather than the array and generate the maze accordingly:

- Please open the script **generate_maze** (i.e., click on the script logo to the left of the node **create_environment**)
- Comment (or remove) all the code already present in the **_ready** function.
- Please add the following code at the beginning of the script (new code in bold):

```
extends Spatial
export (PackedScene) var wall
var array
onready var file = 'res://maze.txt'
```

In the previous code, we declare a variable called **file** that refers to the text file that you have created earlier; this code assumes that the file was saved at the root of the project folder.

- Please add the following code to the script (e.g., at the end of the script).

```
func read_file():
var f = File.new()
f.open(file, File.READ)
var full_text = ""
while not f.eof_reached():
full_text += f.get_line()
f.close()
```

In the previous code:

- We create a new function called **read_file** that will be used to read the text file you have just created and to generate a maze accordingly.

- We create a variable called **f** which is a new file instance. This new file will be used to extract the content from the text file that we have created earlier.
- We then use this file instance to open the file defined earlier as **file** (i.e., **maze.txt**) using the keyword **open** and specifying how we want to use this file (i.e., **READ**).
- We create the variable **full_text** and set it to an empty string.
- We then create a **while** loop, a type of structure that checks for a specific condition and then loops based on this condition. In our case, we just check that we have not reached the end of the file yet; and if that is the case, we add every new line read from this file to the variable **full_text**.
- Finally, we close the file as we have finished reading it.

It is now time to add the walls based on the data read in the file:

- Please add the following code to the function **read_file** (new code in bold):

```
f.close()
for i in range (0,100,1):
var new_wall = wall.instance()
if (full_text[i] == "1"):
add_child(new_wall)
var x_pos = 50 - (i/10) * 10
var z_pos = -(50 - (i%10) * 10)
new_wall.global_transform.origin = Vector3(x_pos,2,z_pos)
```

In the previous code, we proceed as we have done previously for reading the array: we check the value read in the file, and if it is one, we instantiate a new wall at a position defined by i.

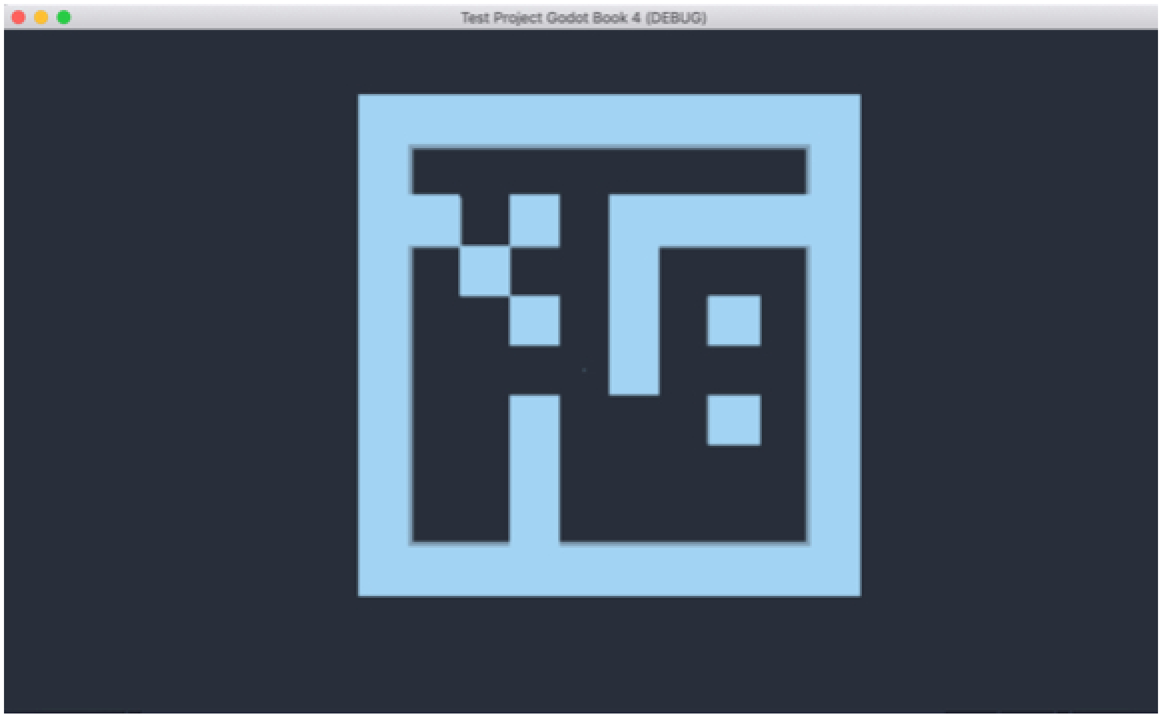
So that we can use this new function, we just need to call it from the **_ready** function, so please modify the **_ready** function as follows (new code in bold):

func _ready():

#read_array()

read_file()

As you play the scene, it should look as per the next figure:



You should see that the scene has been generated as previously, but with the difference that the content is now read from a text file that is saved within your project.

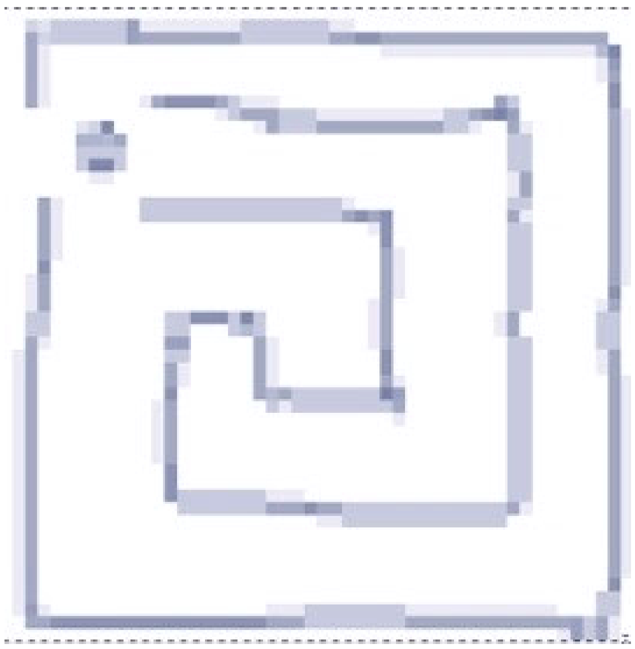
Following this principle, you could have several files that correspond to each game level, each with a different name, and you could then load these depending on the level to be displayed.

Creating an environment from an image file

While the previous techniques are quite interesting and useful, there is another way to create your level, using a more artistic approach, that is: by drawing the outline of your levels as an image, and then by reading this file and instantiating objects based on the color of each pixel present in the image.

Let's see how this can be done:

- Please create a simple jpeg image, using an image editor of your choice, for example Adobe Photoshop, Microsoft Paint or Gimp.
- As you create your image, you can use the brush tool, and ensure that its size is **1** so that you can draw pixel-by-pixel.
- For this particular application, we will leave empty spaces white, and any other pixel painted using the color of your choice.



- The previous figure is my outline; again, it is very simple for the time being; it uses white pixels for empty areas and colored pixels for walls. This image is

500 by 500 pixels, simulating an area that is 500 meters wide and 500 meters long. If you wish, you can find and use this outline from the resource pack (i.e., **outline.png**).

- You can save this image to any format of your choice; in my case, I have saved it as a **png** file (i.e., **outline.png**).

Note that you can use the file **outline.png** that is already present in the **File System** tab.

When this is done, you can open Godot, open your current project, and import the image that you have just created by dragging and dropping it from your Operating System to the **File System** tab in Godot (in the folder called **res://**).

At this stage, we have an image that is ready to be used and read; so all we need to do is to write the code that will create the environment based on this image:

- Please open the script called **create_environment**.
- Please add the following code at the start of the class (new code in bold).

```
extends Spatial
export (PackedScene) var wall
var array
onready var file = 'res://maze.txt'
var image = load("res://outline.png")
```

In the previous code, we create a new variable called **image** that refers to the image that you have previously created and saved in the current Godot project.

- Please add the following function to the script (e.g., at the end of the script):

```
func read_image():
var data = image.get_data()
```

```
data.lock()
for i in range (0,10,1):
for j in range (0,10,1):
var pixel = data.get_pixel(i,j)
if (pixel != Color.white):
```

In the previous code:

- We create the function **read_image** that will be used to read every pixel in the image that you have created previously, and, upon finding a pixel that is not white, create a corresponding wall.
- We gather the data from the image (i.e., its pixels) using the built-in function **get_data**.
- Then, we lock the file, to prevent any file corruption.
- We then read the image, one pixel at a time, using two loops.
- For each pixel present in the image and determined by its **x** and **y** coordinates, we save its color using the method **get_pixel** to the array that we have defined earlier.

Once this is clear, please add the following code just after the previous one (new code in bold):

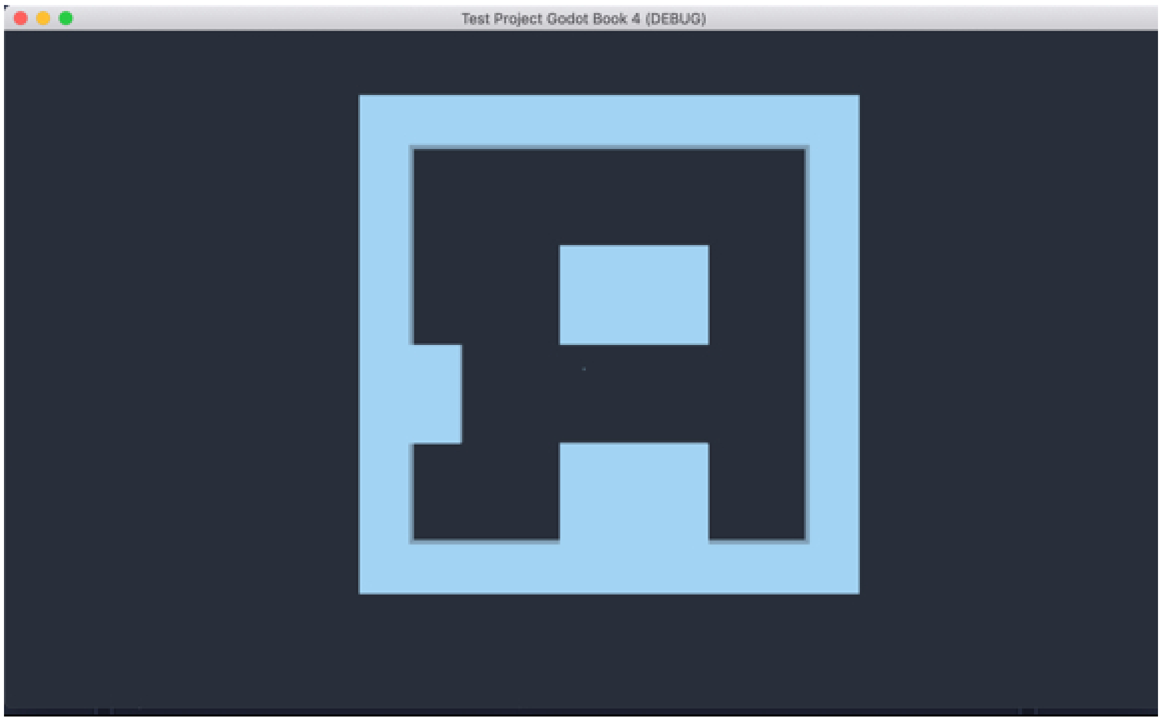
```
if (pixel != Color.white):
var new_wall = wall.instance()
add_child(new_wall)
var x_pos = 50 - (i) * 10
var z_pos = -(50 - (j) * 10)
new_wall.global_transform.origin = Vector3(x_pos,2,z_pos)
```

In the previous code, if the color of the current pixel is not white, then an object is created accordingly, as we have done in the previous sections.

The last thing that we need to do is to call the function **read_image** from the **_ready** function; so please, comment all the code in the **_ready** function, and add this code to it (new code in bold).

```
func _ready():
**#read_array()**
**#read_file()**
**read_image()**
```

- Once the code has been saved; please check that it is error-free; you can then play the scene and it should look like the following figure.



For this technique and all the other procedural generation techniques covered in this chapter, you can of course navigate the scene by adding a **First-Person Controller**. This, for example, can be done when you edit the scene by doing the following:

- Duplicate the scene call **wall.tscn** and rename the duplicate **ground.tscn**.
- Open the scene **ground.tscn**.
- Rename the node **wall** to **ground** and sets its **y** coordinate to **-1**.
- Select the node called **CollisionShape**.
- Using the **Inspector**, modify its **scale** attributes to **(500, 1, 500)**.
- You can also select the node **CSGBox** and add a new **Material** to it so that it appears blue or choose any other color or texture of tour choice.
- Save this scene.
- Switch to the scene **game_level_auto**.
- Add the ground scene: right-click on the node **Spatia**l, select the option **In-stan tiate Child Scene**, and select the scene **ground.tscn**. This will create a new
- Add a first-person controller: right-click on the node **Spatial**, select the option **Instantiate Child Scene**, and select the scene **player.tscn**.
- This will create a new node called **KinematicBody**.
- Change the position of the node **KinematicBody** to, for example, **(0, 1, 0)**, so that it is above the ground.

Now, while this works, there are many ways in which we could optimize this script; we could, for example, include specific prefabs when a color is found in the file. For example, we could use red for doors, and blue for walls with different colors, and so on.

Using JSON files for content creation

Using text files to generate an environment at run time is great; however, this could be limited if we wanted to have more control over the properties of the objects to be instantiated; for example, we could use numbers in our file, and employ 3 for a tree and 2 for a wall; but what if we would like to instantiate different types of walls with each a different texture, size, or rotation. You can see here that using a text file, while useful, may be limited; instead, we could use something a bit more advanced such as JSON files; A JSON file will make it possible to provide more information about the object (or nodes) that we want to create in our game. The nice thing about this feature is that it can also be used for complex visualization techniques to visualize data that is stored in the JSON format, including data on weather, financial transactions, scientific measurements, or news information. Many of these JSON files are freely available and can be accessed and processed. So, for example, you could create a 3D environment that simulates scientific data such as temperatures, or streams in oceans, and so forth; and since these files are usually updated on a regular basis, you could provide a 3D application that simulates real-life phenomena.

Now, before you can move to these complex applications, we will just create a relatively simple application that reads data from a JSON file to configure the game; we will then create another simple file to add objects to be included to the levels, along with settings their properties, all of this from a JSON file.

So first, what is a JSON file?

JSON stands for **J**ava**S**cript **O**bject **N**otation. As you will see in the next code examples, these files use the extension **.json** and have a common structure that makes them easy to read.

Let's look at a JSON file that we could create to describe a scene; it could look like the following.

```
{
"obj1":{"name":"wall1", "color":"red", "location":"10,0,10", "rotation":"0,0,0", "scale":"1,2,1", "level":1},
"obj2":{"name":"wall1", "color":"green", "location":"10,0,10", "rotation":"90,0,0", "scale":"1,2,1", "level":1}
}
```

- The file starts and ends with curly brackets.
- It then includes a succession of objects.
- Each object is defined by an identifier within double quotes; followed by a colon, followed by opening and closing curly brackets.
- Within these curly brackets, we define the attribute of the object using an attribute/value pair separated by a comma. The name of the attribute is usually within double quotes, and the corresponding attribute may (if this is a string) or may not use quotes.
- For example, in this file we have two objects with the id **obj1** and **obj2**.
- The first object has the attributes **name**, **color**, **location**, **rotation**, **scale** and **level** and values associated with each of these attributes.
- The same applies to the second object.

The beauty of this file format is that you can create your own JSON files using the attributes of your choice to best reflect and serve the requirements of your game or application; you could save information about each scene, about the NPCs (e.g., what paths they can use), or the weapons. So yes, you could virtually save any type of information with these files, using an easy-to-read format.

Now that the JSON format is clearer, let's see how we can read a JSON file to create a simple scene.

In the next section we will:

- Load a JSON file.
- Open this file.
- Navigate through each object within the file.
- For each of these objects, create the corresponding game nodes defined for this scene.

The file that we will be using will has the following content (explained in the previous section):

```
{
"obj1":{"name":"wall1", "color":"red", "location":"10,0,10", "rotation":"0,0,0", "scale":"1,2,1", "level":1},
"obj2":{"name":"wall1", "color":"green", "location":"10,0,10", "rotation":"90,0,0", "scale":"1,2,1", "level":1}
}
```

- As you can see from the previous code, this file describes two objects, along with their **name**, **color**, **location**, **rotation**, **scale**, and **level**.

So at this stage, we have a JSON file, and we want to access it, to read its content and to create a scene accordingly;

First, we will copy this file from the resource pack to our project:

- Please copy the file called **scene.json** from the resource pack to the **FileSystem** folder in Godot (e.g., drag and drop).

Next, we will create the code that reads this file.

- Please open the file **generate_maze.gd**.
- Add the following function to it (e.g., at the end of the file).

```
func read_json_file():
var file = File.new()
file.open("res://scene.json", file.READ)
var json_data = parse_json(file.get_as_text())
var json = to_json(json_data)
var dictionary : Dictionary = JSON.parse(json).result
for key in dictionary:
var name = dictionary[key].name
var color = dictionary[key].color
var location = dictionary[key].location
print ("Name:"+ name +", Color:"+ color + ", Location: "+location)
```

In the previous code:

- We create a function called **read_json_file**.
- We instantiate a new file using the code **File.new** and save it in the variable called **file**.
- We then specify that this file will refer to the file called **scene.json**, and that we will be reading it.
- We read the entire file using the built-in function **get_as_text**, this text content is then parsed (i.e., formatted in a way that can be used for JSON in our case) as a dictionary.

A dictionary is a structure that holds a series of key/value pairs. In our file, for example, each object has a key/value pair for its **color**, **location**, etc.

- This dictionary is then saved to the variable **json** using the built-in function **to_json**.
- We then create a new variable of type **dictionary** and save the variable **json**

inside it.

- Once this is done, we loop through our dictionary, and then save and display the value of the attributes **name** and **color** for each object in the file.

The keyword **for in** can be used to go through each attribute for an object defined in the file.

- Last but not least, please modify the code in the **_ready** *function* so that the function **read_json_file** can be called (new code in bold).

```
func _ready():
#read_array()
#read_file()
#read_image()
read_json_file()
```

As you save your code and play the scene, you should be able to see the following message in the **Output** window.

```
Name: wall1, Color:red, Location: 10,0,10
Name: wall1, Color:green, Location: 10,0,10
```

This is just a short introduction to reading JSON files and the next sections will delve more into this topic to create a solar system based on a JSON file.

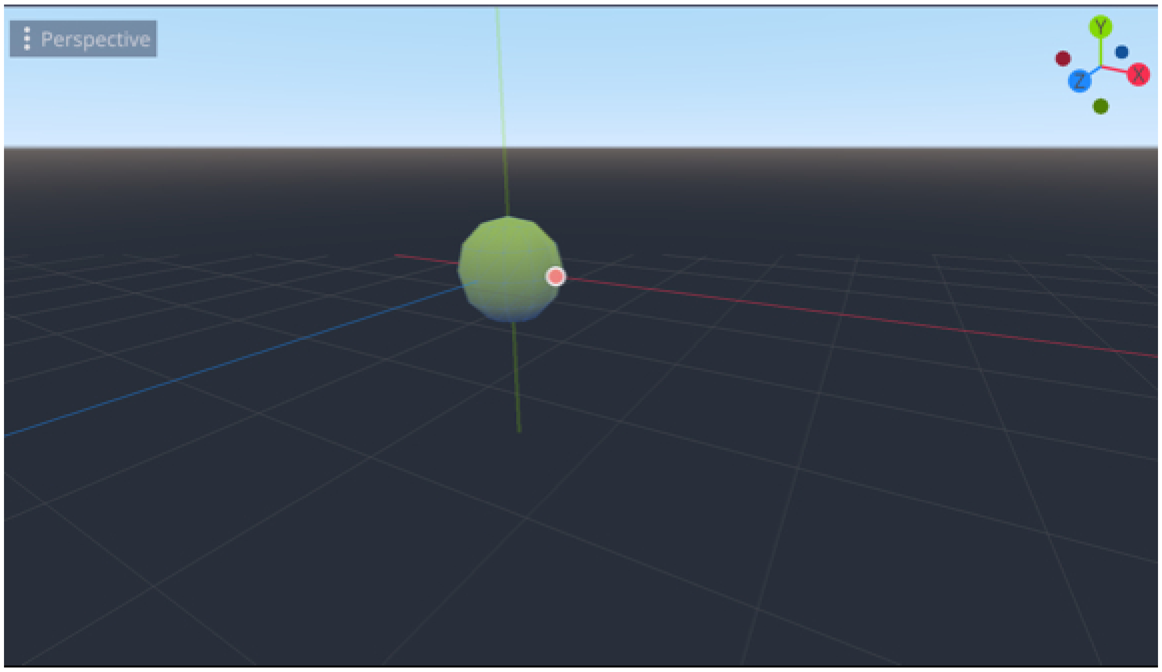
Creating a virtual solar system based on a JSON file

So at this stage, we know how to read a JSON file. Based on this knowledge, we could simulate the generation of a virtual solar system from a JSON file. So the idea here would be to include all information related to planets (e.g., size, orbital velocity, etc.) in a JSON file, and then to load this information to create a simple visualization of this data in Godot. So in the next section, we will do the following:

- Create a scene for a planet that will include variables that can be changed such as size or orbital velocity.
- Read the JSON file that includes information about all these planets.
- Instantiate the planets in Godot and simulate their movement.

So let's get started:

- Please create a new **3D scene**: Select **Scene | New Scene** and then the option **3D Scene** in the new window.
- This will create a new scene with a default node called **Spatial**.
- Save this scene as **solar_system** or another name of your choice (**Scene | Save Scene As**).
- Create a new **CSGSphere** node, add it as a child of the node **Spatial** and re-name it **sun**.
- Set its location to **(0, 0, 0)** and its scale to **(45, 45, 45)**.
- Set its color to **yellow** (i.e., create a new **Spatial Material** and set its **Albedo** at-tribute to **yellow**).



So at this stage, we have created the node that represents the sun and we will now create a scene called **planet** that will be used to instantiate all planets included in the **JSON** file:

- Please save the current scene (**CTRL + S** or **CMD + S**).
- Please create a new scene.
- In the new window, select the option **Other Node**.
- In the next window, select the node type **CSGSphere**.
- This will create a new scene with a default node called **CSGSphere**.
- Save it as **planet.tscn (Scene | Save Scene As)**.
- Rename the node called **CSGSphere** to **planet**.
- Set its color to **red**.
- Save the scene (**CTRL + S** or **CMD + S**)

Once this is done, we just need to set this planet in movement using a simple script.

So let's create this script:

- Using the scene **planet.tscn**, select the node called **planet** and add a script to it.
- Name this script **planet.gd**.
- Add this code at the beginning of the script (just after the first line of code).

```
var rotational_speed = 10
var orbital_speed = 0.2
var orbital_angle = 0.0
var angle = 0.0
var orbital_rotational_speed = 20
var distance_to_sun = 150
onready var sun = get_node("/root/Spatial/sun")
```

In the previous code:

- We set the parameters for this planet including **rotational_speed** (the speed at which the planet revolves around its axis), **orbital_speed** (the speed at which the planet revolves around the sun), **orbital_angle** (the rotation angle around the sun), **angle** (the rotation angle around the planet's axis), **orbital_rotational_speed** (the speed at which the planet revolves around the sun), and **distance_to_sun**.
- We also declare a node called **sun** that will be used as the center of the rotation for the planet.
- Please modify the **_ready** function as described in the next code snippet (new code in bold).

```
func _ready():
global_transform.origin = Vector3(distance_to_sun,0,distance_to_sun)
```

In the previous code, we set a temporary position for the planet.

We can now work on the rotational movement of the planets; this includes a rotation around the planet's axis as well as a rotation around the sun (i.e., orbital rotation).

- Please add the **_process** function as follows:

```
func _process(delta):
rotate_y(rotational_speed * delta)
var tempx
var tempy
var tempz;
orbital_angle += delta * orbital_speed
tempx = sun.global_transform.origin.x + distance_to_sun * cos(orbital_angle);
tempz = sun.global_transform.origin.z + distance_to_sun * sin(orbital_angle);
tempy = sun.global_transform.origin.y;
global_transform.origin  = Vector3 (tempx, global_transform.origin.y, tempz)
```

In the previous code:

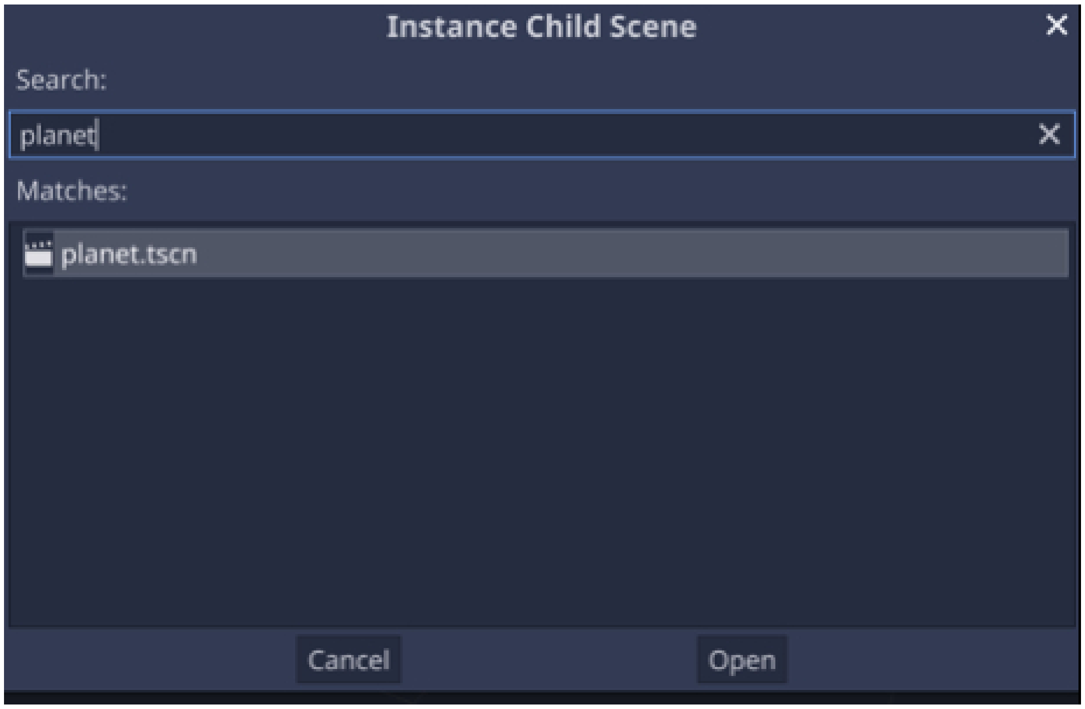
- We rotate the planet around its **y-axis**.
- We then rotate the planet around the sun using its distance to the sun along with the **Sine** and **Cosine** of the angle defined by its position in relation to the sun.

If you are not used to Cosine and Sine: if you consider a circle, its center O, and any point on this circle M. The coordinates of this point M can be determined by the radius of the circle and the corresponding angle defined by the position (angle) of the point M on the circle.

- We use three temporary variables called **tempx**, **tempy** and **tempz** to store the position of the planet and to create a corresponding vector.

We can now save our script and play the scene.

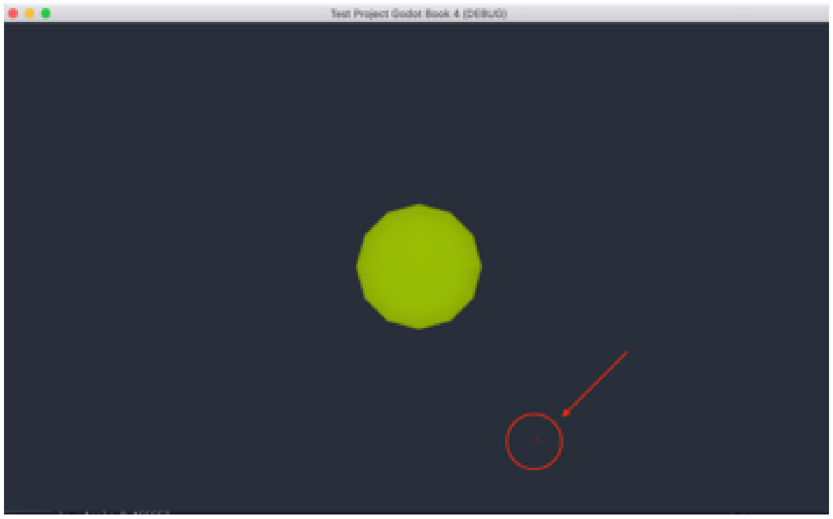
- Please save your script.
- Save the scene called **planet.tscn** (**CTRL** + **S** or **CMD** + **S**).
- Open the scene called **solar_system.tscn**.
- Right-click on the node called **Spatial** and select the option "**Instance Child Scene**".
- In the new window, search for and select the scene **planet.tscn** and click on the button called "**Open**".



- This should add a new child node called **planet** to the node **Spatial**.
- Add a child node of type **Camera** to the node **Spatial**, change its **Far** property

to **9000**, its translation to **(0, 250, 0),** and its rotation to **(-90, 0, 0)**.

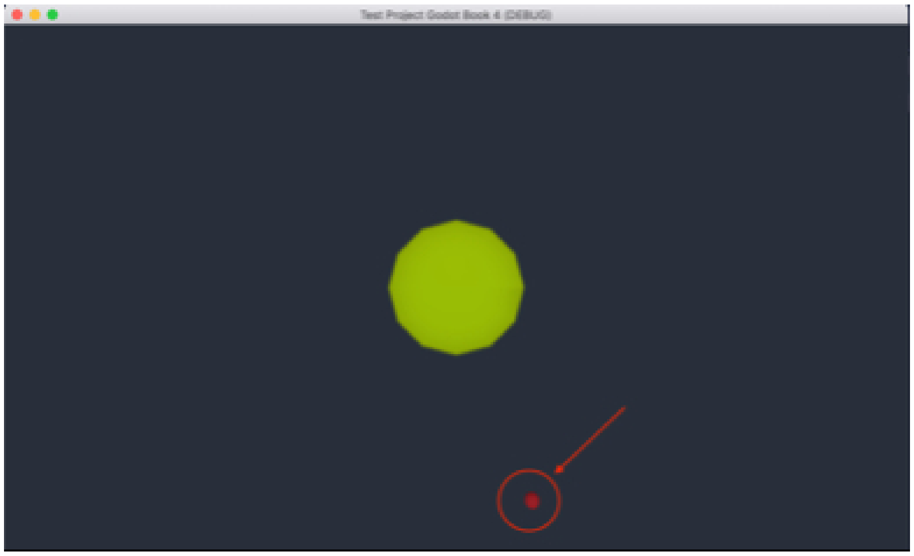
- Once this is done, please play the scene.
- You should see the **planet** object orbiting around the sun as per the next figure.



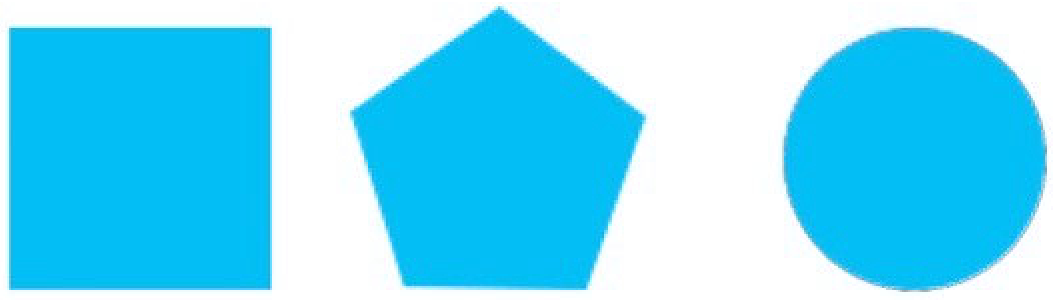
If you can hardly see the planet, you can temporarily modify the function **_ready** in the script **planet.gd** by adding the following line, so that the planets look bigger.

global_scale(Vector3(5, 5, 5))

The next figure shows how the planet should look like after the previous code modification.



While this is working, it may be difficult to locate the planet that is orbiting around the sun and we could create a trail that shows the full orbit of the planet. To do so, we will use **ImmediateGeometry** nodes and vertices; these are often used to draw lines in Godot. In our case, as we want to create an ellipse (or a circle, in our case, to simplify the code), we will create a succession of infinitely small lines, which, seen together, will be perceived as an ellipse (or a circle). For example, the next figure shows that by increasing the number of vertices (i.e., corners) of a polygon, we can progressively approximate its shape from a box to a circle.



- Please add the following function to the beginning of the script **planet.gd** (e.g., just after the first line).

```
var points = Array()
var length_of_line_renderer = 100
var c1:Color = Color.red
```

In the previous code:

- We define an array called **points** that will store all the vertices that will make up the ellipsoidal trajectory of our planet.
- We define the number of lines that will be used to draw the orbital trail; in our case, there will be **100** lines (but we could use more if needed).
- We define the color that will be used to draw our lines.

We can now create the function that will draw a planet's orbit:

- Please add the following function to the script (e.g., at the end of the script).

```
func draw_orbit():
var im1
im1 = get_node("/root/Spatial/Earth")
for i in range(0,length_of_line_renderer,1):
var unit_angle = (2*3.14)/length_of_line_renderer
var current_angle = unit_angle*i;
var pos:Vector3 = Vector3(distance_to_sun *cos(current_angle), 0, distance_-
to_sun*sin(current_angle))
```

In the previous code:

- We declare a function called **draw_orbit**.
- We initialize the variable **im1** so that it is linked to a node called **Earth** that we have yet to create. This node will be an **ImmediateGeometry** node that will make it possible to draw a geometry made of vertices in our 3D environment.

- We create a loop with a counter that starts at **0** and increases up to **100** (i.e., the value of the variable **length_of_line_renderer**).
- We then define, for each point on this path, an angle and a position. To create what will look like a circle, we divide 360 degrees (i.e., 2*3.14 radians) by 100 to find the angle between each of these points.
- Once this angle is found (let's call it **unit_angle**), we multiply it by the rank of the point; for example, the first point (rank **0**) will be at an angle **unit_angle * 0** (0.0628 * 0), the second point will be at an angle **unit_angle* 1** (0.0628*1), and so on.
- We then set the position of each point based on the planet's distance to the sun and the current angle (i.e., **unit_angle**)

We can now start to add these pouts to an array and to create the actual lines:

- Please add the following code after the previous one (within the loop) in the function **draw_orbit**:

```
points.append(pos)
im1.clear()
im1.set_color(c1)
im1.begin(Mesh.PRIMITIVE_LINE_LOOP)
for point in points:
im1.add_vertex(point)
im1.end()
```

In the previous code:

- We add the new point (or vertex) to the array called point.
- We clear everything that was drawn previously.
- We set the color to be used to draw the lines.

- We start to draw using the keyword **begin** and we specify that we will be drawing lines between the vertices defined in the array, ensuring that the line between the last point and the first point is also drawn (hence the PRIMITIVE_LINE_**LOOP** keyword).
- We then loop through all the points in the array of points and add these as vertices of the geometrical shape.
- Finally, we end the drawing.

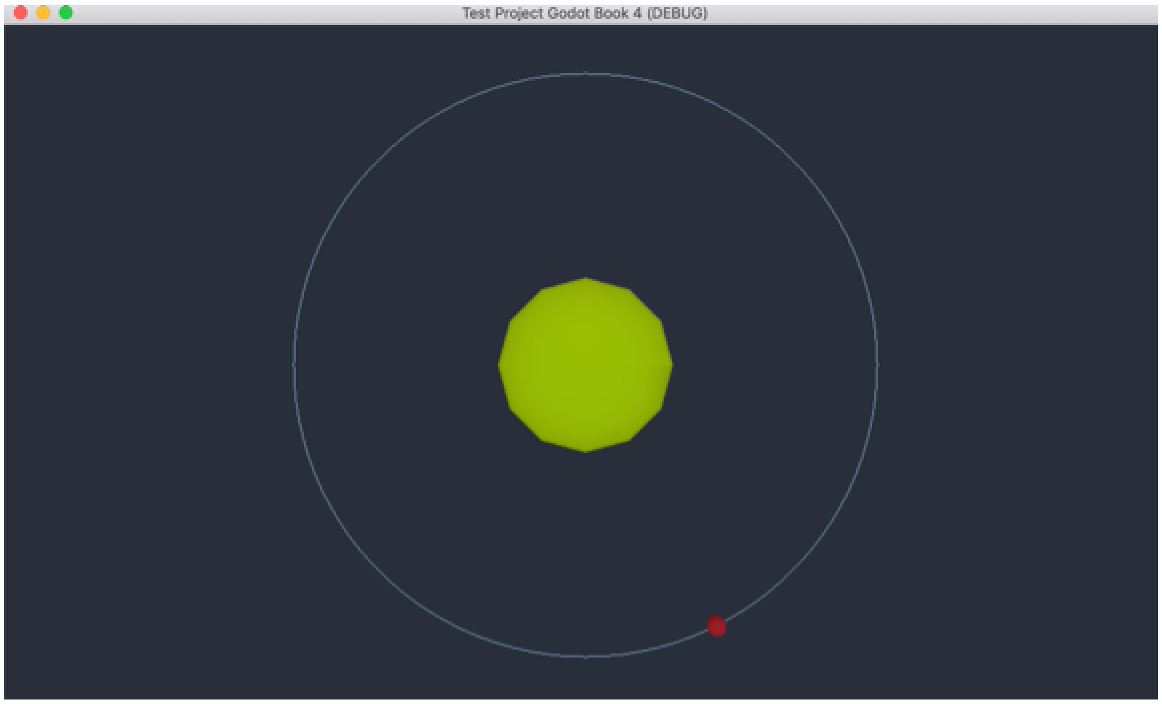
Now that the code for the trail has been created, we just need to call it from the **_ready** function.

- Please add the following code to **_ready** function:

```
draw_orbit();
```

Finally, we just need to create a new **ImmediateGeometry** node to the scene so that our trail can be displayed:

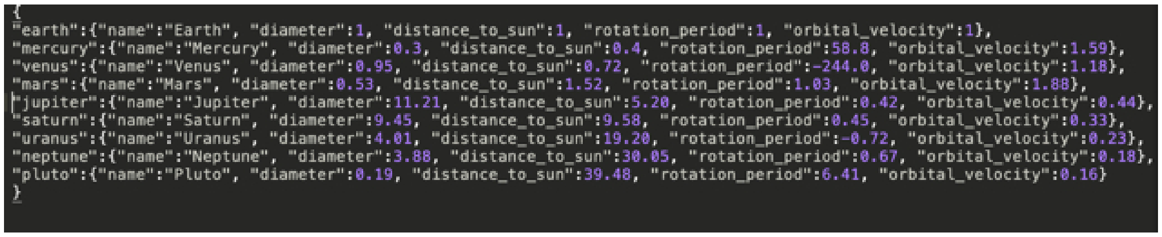
- Please add a child node of type **ImmediateGeometry** to the node **Spatial**, and rename the new node **Earth**.
- Please save your code, play the scene and observe the orbital trail we have just defined.



So, we have a planet that orbits around the sun. The idea now is to create other planets from a JSON file and instantiate their corresponding node in the scene using the template/scene **planet**.

So let's get started.

- The JSON file that we will be using is structured as described in the next snippet.



```
{
"earth":{"name":"Earth", "diameter":1, "distance_to_sun":1, "rotation_period":1, "orbital_velocity":1},
"mercury":{"name":"Mercury", "diameter":0.3, "distance_to_sun":0.4, "rotation_period":58.8, "orbital_velocity":1.59},
"venus":{"name":"Venus", "diameter":0.95, "distance_to_sun":0.72, "rotation_period":-244.0, "orbital_velocity":1.18},
"mars":{"name":"Mars", "diameter":0.53, "distance_to_sun":1.52, "rotation_period":1.03, "orbital_velocity":1.88},
"jupiter":{"name":"Jupiter", "diameter":11.21, "distance_to_sun":5.20, "rotation_period":0.42, "orbital_velocity":0.44},
"saturn":{"name":"Saturn", "diameter":9.45, "distance_to_sun":9.58, "rotation_period":0.45, "orbital_velocity":0.33},
"uranus":{"name":"Uranus", "diameter":4.01, "distance_to_sun":19.20, "rotation_period":-0.72, "orbital_velocity":0.23},
"neptune":{"name":"Neptune", "diameter":3.88, "distance_to_sun":30.05, "rotation_period":0.67, "orbital_velocity":0.18},
"pluto":{"name":"Pluto", "diameter":0.19, "distance_to_sun":39.48, "rotation_period":6.41, "orbital_velocity":0.16}
}
```

As you can see, it consists of:

- Opening and curly brackets at the start and end of the file.
- Several objects are defined by their id (e.g., "**earth**"), and a series of attributes

such as **name**, **diameter**, **distance_to_sun**, etc. All these attributes, except the one called **name**, will be relative to the settings for the planet **Earth** (i.e., the settings for the planet Earth have already been defined earlier in the script **planet.gd**).

So the goal here will be to read this JSON file (this will be file **planets.json** that is already in your project), and, for each planet defined within, to create (or instantiate) a node with the corresponding properties.

- Please hide the node **planet** located in the scene **solar_system**, for the time being.
- Select the node **Spatial**, and add a new child node of type **Spatial** to that node, and rename it **load_planets**. This node will be used to add (and execute) the script that will load all the planets.
- Add a new script to the node **load_planets**, and name it **load_planets.gd**.
- Please add the following code at the beginning of the file (i.e., **load_planets.gd**), just after the first line:

```
export(PackedScene) var planet_template
```

In the previous code, we declare a variable called **planet_template**, which will effectively be a placeholder to which we will be able to drag and drop the scene **planet.tscn** to use it as a template for all the plates to be added to the solar system.

- Please add the following code to the **_ready** function.

```
func _ready():
load_all_planets()
```

We will now create the code for the function **load_all_planets**.

- Please add the following function to the script:

```
func load_all_planets():
var file = File.new()
file.open("res://planets.json", file.READ)
var json_data = parse_json(file.get_as_text())
var json = to_json(json_data)
var dictionary : Dictionary = JSON.parse(json).result
```

In the previous code:

- We declare the function **load_all_planets**.
- We instantiate a new file using the code **File.new** and save it in the variable called **file**.
- We then specify that this file will refer to the file called **scene.json**, and that we will be reading it.
- We read the entire file using the built-in function **get_as_text**; this text content is then parsed (i.e., formatted in a way that can be used for JSON in our case) as a dictionary.
- This dictionary is then saved to the variable **json** using the built-in function **to_json**.
- We then create a new variable of type **dictionary** and save the variable **json** inside it.

Now that we have opened and parsed the JSON file, it is time to extract its content, so please add the following code to the function **load_all_planets**, just after the previous code:

```
for key in dictionary:
var name = dictionary[key].name
var diameter = dictionary[key].diameter
var distance_to_sun = dictionary[key].distance_to_sun
var rotation_period = dictionary[key].rotation_period
var orbital_velocity = dictionary[key].orbital_velocity
```

In the previous code, we loop through our dictionary, and then save the value of the attributes **name**, **diameter**, **distance_to_sun**, **rotation_period**, and **orbital_velocity**.

At this stage, it would be great to know if we can effectively read the content of the JSON file, so please do the following:

- Add this code just after the previous code in the script **load_planets.gd**, to be able to print the name of each planet read from the file, in the function load-_all_planets.

```
print("Name of Planet: "+name)
```

You can now open the scene **solar_system** and play it, and you should see the following message in the **Output** window.

```
Name of Planet: Earth
Name of Planet: Jupiter
Name of Planet: Mars
Name of Planet: Mercury
Name of Planet: Neptune
Name of Planet: Pluto
Name of Planet: Saturn
Name of Planet: Uranus
Name of Planet: Venus
```

So this is working so far and now that the information relative to each planet has been extracted, we just need to create a new planet (based on the template **planet.tscn**) and apply these features (i.e., attributes) to it.

- Please add the following code to the script **load_planets** just after the previous code (new code in bold):

```
print("Name of Planet: "+name)
var g = planet_template.instance()
g.set_distance_to_sun(distance_to_sun)
g.set_orbit_speed(orbital_velocity)
g.set_rotational_speed(1/rotation_period)
g.set_name(name)
g.set_radius(diameter)
add_child(g)
```

In the previous code, we instantiate a new node based on the scene **planet.tscn**, we set the name of this new node, and then call functions that we yet have to define for the script **planet.gd** to be able to set the planet's distance to the sun, its orbital speed, its rotational speed or its radius. Finally, the new planet is added as a child of the current node.

So at this stage, you can manage to load the JSON file and to read its content; so the next phase will be to read the attributes of all the planets and to instantiate the corresponding nodes; however, before this can be done, we will modify our initial script, called **planet.gd**, so that it is possible to modify the properties of each object instantiated from this prefab; so we will create several functions that can be used to set the attributes of the planets that will be instantiated.

So let's create these functions:

- Please add the following code to the script **planet.gd** (e.g., at the end of the script):

```
func set_rotational_speed(s):
```

```
rotational_speed = s * rotational_speed;
func set_orbit_speed(os):
orbital_speed = os * orbital_speed
func set_distance_to_sun(d):
distance_to_sun = distance_to_sun*d;
func set_name(new_name):
name = new_name;
func set_radius (radius):
global_scale(Vector3(radius, radius, radius))
```

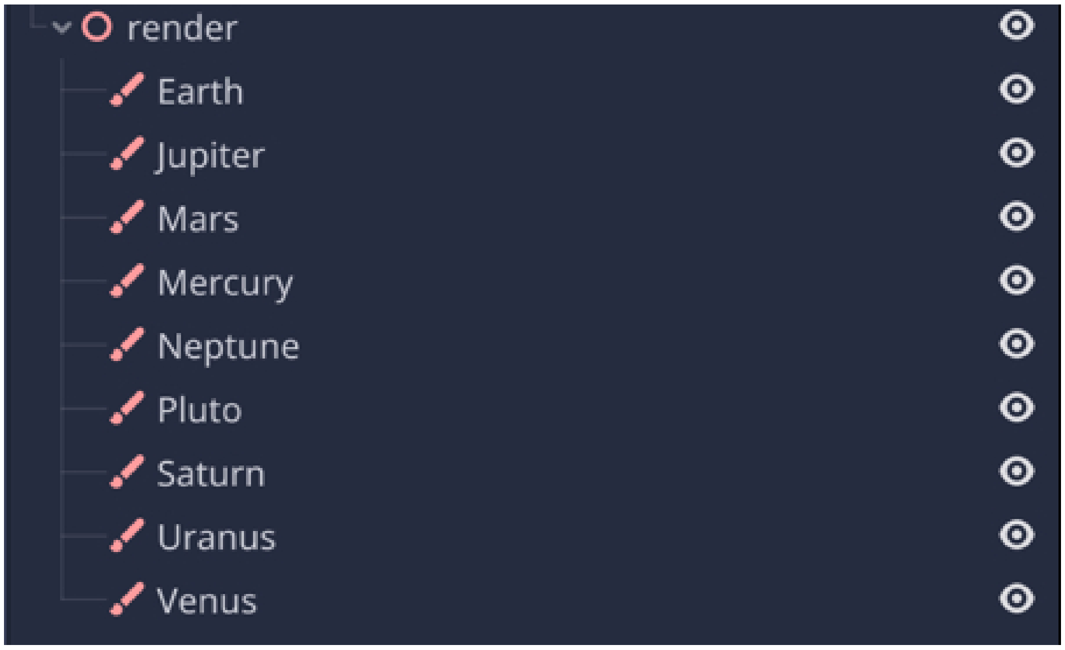
In the previous code:

- We create five methods **set_rotational_speed**, **set_orbit_speed**, **set_distance_to_sun**, **set_name**, and **set_radius**.
- For most of these methods, we use the values passed as parameters to modify the properties of the planet that is instantiated in the scene, bearing in mind that the values included in the **JSON** file are defined in relation to the planet **Earth** (i.e., multipliers).

Next, we just need to create **ImediateGeometry** nodes that will be used for each planet to display their respective trajectories:

- Please open the scene **solar_system**.
- Create a new **Spatial** node as a child of the existing **Spatial** node in the scene, and rename the new node **render**. This node will be used as a folder to store all the **ImmediateGeometry** nodes.
- Please drag and drop the node **Earth** as a child of the node **render**.
- Duplicate this node (i.e., **Earth**) eight times to create the corresponding nodes for the other planets: **Jupiter**, **Mars**, **Mercury**, **Neptune**, **Pluto**, **Saturn**,

Uranus, and **Venus**.



- Once this is done, please modify the script **planet.gd** as follows (new code in bold):

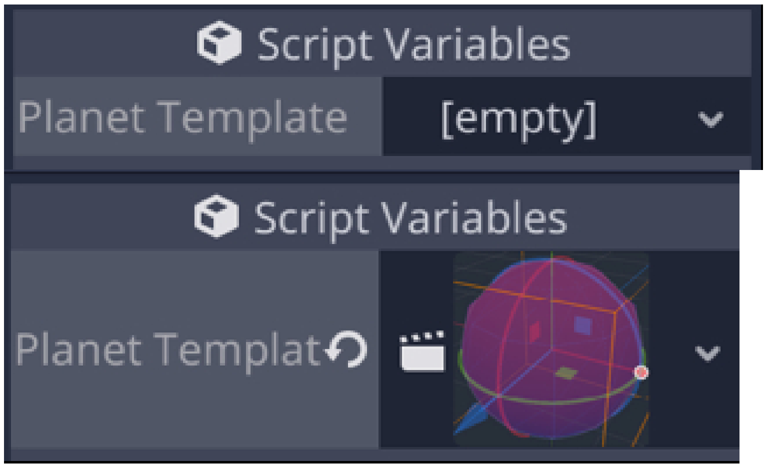
var im1
im1 = get_node("/root/Spatial/render/Earth")
if (name == "planet"): im1 = get_node("/root/Spatial/render/Earth")
else: im1 = get_node("/root/Spatial/render/"+name)

In the previous code, we make sure that the **ImmediateGeometry** node that you have just created is used for the relevant planet based on its name.

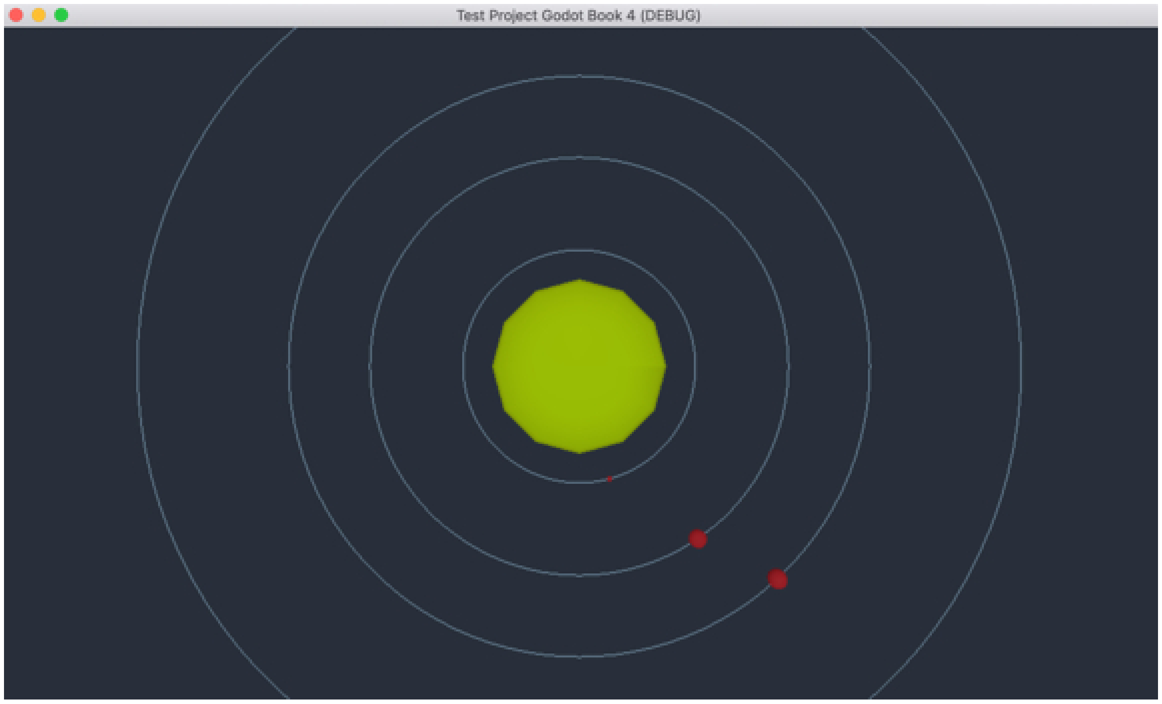
Before we can previoe the scene, please do te hfollowing:

Select the node called load_planets in the Scene Tree.

Drag and drop the file **planet.tscn** to the empty field to the right of the label **Planet Template**, in the **Inspector**.



Once this is done, please save your code, and play the scene. As you play the scene, you should be able to see some of the planets orbiting around the sun with their corresponding trajectories.



While this is working, and because some planets are extremely far away from the sun (compared to planet Earth), it will not be possible to show (and see) them all; so we will just make sure that the user can zoom in and out to be able to see the planets that are both close and far away from the sun:

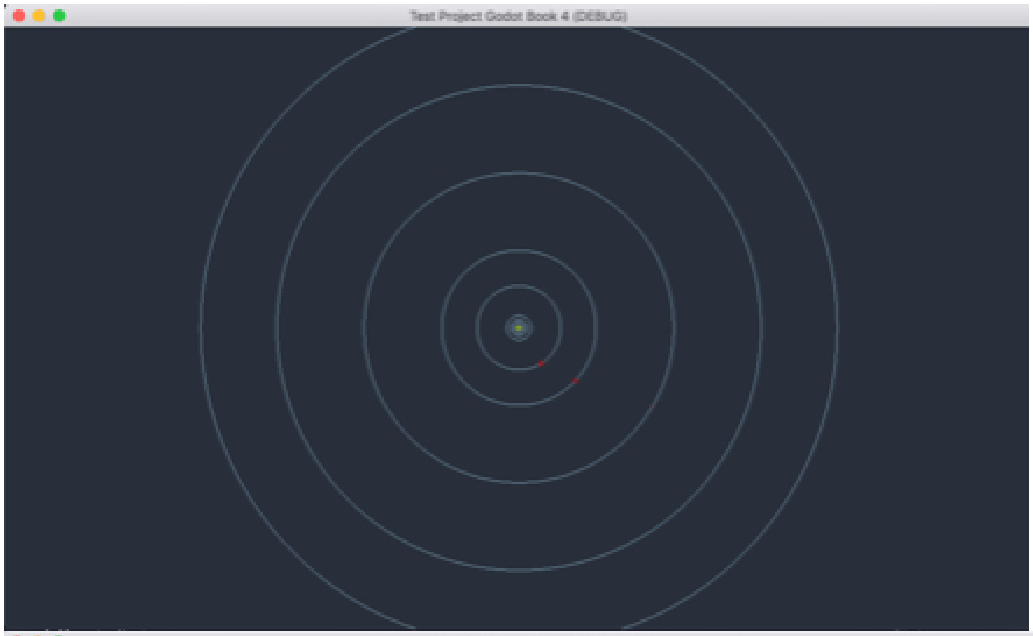
- Please select the **Camera** node already present in the scene.
- Attach a new script to it and rename the script **move_camera**.
- Add the following function to the script:

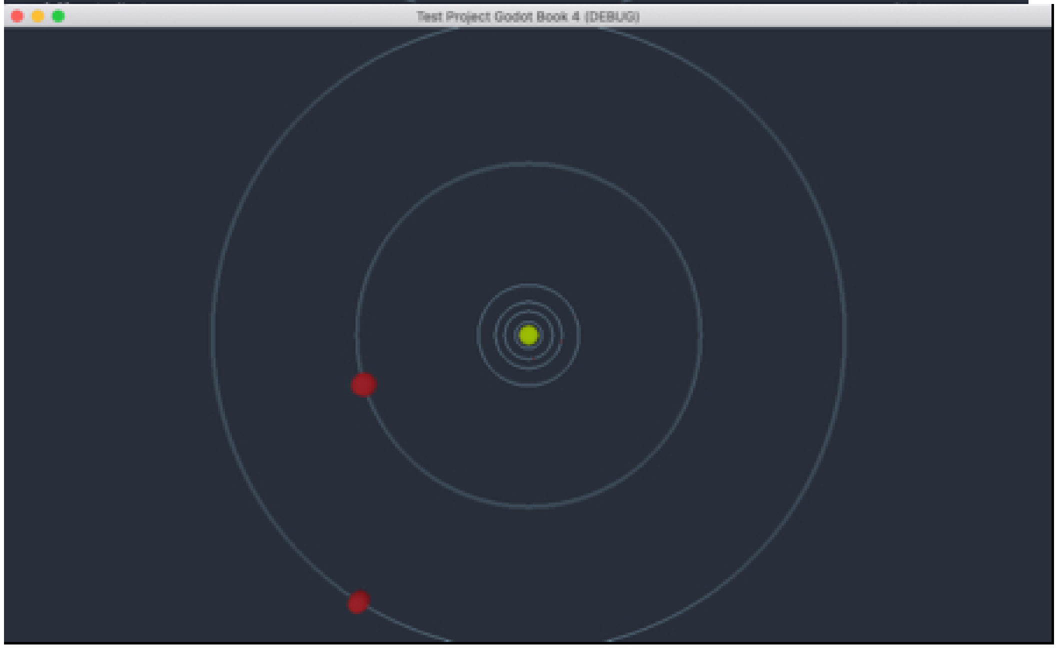
```
func _input(event):
if (Input.is_key_pressed(KEY_Q)):
global_transform.origin = Vector3(0, 8000,0)
if (Input.is_key_pressed(KEY_W)):
global_transform.origin = Vector3(0,2000,0)
if (Input.is_key_pressed(KEY_E)):
global_transform.origin = Vector3(0,250,0)
```

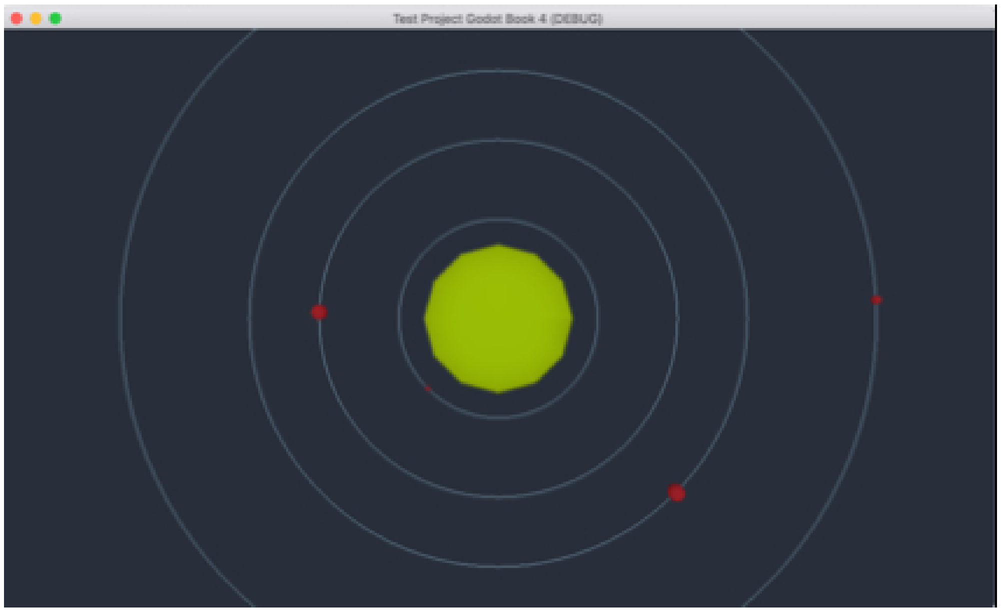
In the previous code:

- We detect the keyboard keys pressed by the user.
- If the user presses the **Q** key, the y coordinate of the camera will be changed to **8000** to zoom out and see the planets that are further away from the sun.
- If the user presses the **W** key, the y coordinate of the camera will be changed to **2000** to zoom in slightly and see the planets that are closer to the sun.
- If the user presses the **E** key, the y coordinate of the camera will be changed to **250** to zoom in and see the planets that are the closest to the sun.

You can now play the scene, and use the keys **Q**, **W**, and **E** to zoom in and out and see the planets orbiting around the sun.


Test Project Godot Book 4 (DEBUG)


Test Project Godot Book 4 (DEBUG)



LEVEL ROUNDUP

In this chapter, we have learned how to create a level procedurally. We became more comfortable with reading text and JSON files. We managed to create levels that can be modified easily either through a simple text file, an image, or even a JSON file. Finally, we have also learned to add and access assets from a JSON file from our script. So, again, we have covered considerable ground to save you considerable time when creating your scenes.

Checklist

You can consider moving to the next stage if you can do the f

- Create a simple scene from an array.
- Read a text file from a script.
- Read the content of a JSON file from a script.
- Understand the concept of keys and attributes when reading a JSON file.

Quiz

Now, let’s check your knowledge! Please answer the following questions (the answers are included on the next page) or specify whether these statements are either TRUE or FALSE.

1. The class called **File** can be used to open files from your script.
2. JSON stands for **J**ava**S**cript **O**bject **N**otation.
3. Please find one error in the following JSON file

```
{
"obj1":{"name":"wall1", "color":"red", "location":"10,0,10" "rotation":"0,0,0",
"scale":"1,2,1", "level":1},
"obj2":{"name":"wall1", "color":"green", "location":"10,0,10" "rota-
tion":"90,0,0", "scale":"1,2,1", "level":1}
}
```

1. It is possible to create JSON files for your own use, provided that they follow proper syntax.
2. If a JSON file is located in your project, you will be able to read it from your code.
3. So that an image can be read pixel by pixel, it just needs to be imported into your project.
4. The attributes stored for all nodes in JSON files are saved as float variables.
5. A JSON file consists of key/value pairs for every object listed within.
6. A dictionary can be used to store information present in a JSON file.
7. To keep data intact, it is sometimes necessary to lock a file while reading it.

Answers to the Quiz

1. TRUE.
2. TRUE.
3. A comma is missing after the attribute location for both objects.

```
{
"obj1":{"name":"wall1", "color":"red", "location":"10,0,10" "rotation":"0,0,0", "scale":"1,2,1", "level":1},
"obj2":{"name":"wall1", "color":"green", "location":"10,0,10" "rotation":"90,0,0", "scale":"1,2,1", "level":1}
}
```

1. TRUE
2. TRUE.
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. TRUE.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, you could use these to make the level creation and loading more efficient. So for this challenge, you will be creating and loading three different levels:

- Create three different text files, each describing the content of a level, using 0s and 1s, as we have done at the start of this chapter.
- Save these files with different names to your project folder.
- Create a function that loads any of these levels based on a parameter passed to this function. For example

```
Fun load_level(level:int)
```

- Create a menu level with three buttons, one for each level.
- When the player presses a button, the corresponding level should be generated (i.e., loaded).

Challenge 2

In this challenge, you will be modifying an existing JSON file and also creating and using your own JSON file:

- Modify the **planets.json** file to add three imaginary planets of your choice and test the scene.
- Create a new color attribute for all planets, that will be used for the trajectory of each planet. You can use any of the following colors: blue, cyan, yellow, gray, green, magenta, red, or white.
- Read this attribute and set the color of the trajectory path accordingly (i.e., the circle that represents the orbit).

Challenge 3

In this challenge, you will also be creating and using your own JSON file:

- Create your own JSON file and create a scene based on this file.
- You can also, if you wish, load the JSON information from an existing JSON file.

Chapter 2: Accessing and Updating a Database

In this section, we will learn how to interact with a database from Godot, using some simple but effective techniques.

After completing this chapter, you will be able to:

- Understand basic database concepts.
- Understand how online databases can be accessed through scripting.
- Understand how to access a database from Godot.
- Access this database to read or update information about the player.

INTRODUCTION TO ONLINE DATABASES

As it is, you may know how to save information from a game so that it is kept between scenes using global variables. This being said, in many situations, there is data that you may like to keep centrally, so that it can be accessed by players (or Godot) regardless of the computer or device used by the player. This can, for example, consist of a list of high scores, players' preferences, or players' details. In this case, as for many web-based applications, these details are stored on a server in an online database.

So what's a database?

To put things simply, a database is a collection of tables; each table contains specific information arranged in columns and rows. Each column corresponds to a specific attribute (e.g., name, score, etc.), while a row refers to a particular data set (each data set has the same type of attributes, but the values may differ between rows). For example, you could have a table that includes details about a player such as: login, password, high-score, last room explored, etc. So, to access this information, you would usually need to access the database and then select this particular table; once the table is selected, you would then look at the row that includes the information for a particular user or player. Each player could be identified by an id, or by their nickname. So, that's it in a nutshell.

Now, in web development, accessing the database is usually performed with what is called a server-side script; a server-side script is a script, or a piece of code, that is saved on the server; when called, it is executed on the server and the results are then sent to the client (for example, your browser). These scripts can be written in several types of languages including PHP (Personal Home Page or Hypertext Preprocessor), a very popular server-side scripting language.

So, in a typical setting, you would do the following:

- Write a PHP script.

- This script would then be saved on a server.
- When executed (e.g., when a url that points to the script is entered in a browser), this script can do several things, including returning text (e.g., displayed on screen), performing calculations, or accessing a database hosted on the same server.
- In the latter case, the database can be a MYSQL database.

So let's look at what a PHP script looks like:

```
<?PHP
echo "Hello";
?>
```

In the previous script:

- The code **<?PHP** indicates that we start PHP commands.
- The second line uses the command **echo** to print (or output) the text **Hello**.
- The third and last line indicates the end of the PHP commands.

Since PHP is a web-based server-side scripting, it is possible to mix both PHP commands and HTML code in a given PHP script; to differentiate between these, PHP commands are usually included between the tag **<?php** (or <**?**) and the tag **?**>.

- We could then usually save this script on a server and give it a name (e.g., **hello.php)**
- We would then typically execute this script by requesting the page **hello.php** and typing its url in a browser, for example: **yourserver.com/hello.php**
- You would then see the text "**Hello**" in your browser

In other words, by entering the url of the script, you are asking the server to execute the script; and in this case, the script will display, through the command

echo, the text "**Hello**"

Now, of course, more complex tasks, including database access, can be done through PHP, but this is the general idea behind executing server-side scripts.

When you want to access an online database with Godot, the process is quite similar in the sense that you do the following:

- Create a script in Godot.
- Within this script, provide the url of the PHP script to be called.
- Call this url from Godot, so that the script is executed.
- Record the result sent from the PHP script in Godot (e.g., by using GDScript).

So now that you have a clearer idea of what server-side scripting is, let’s have a closer look at the nitty-gritty of accessing a database before we can translate this skill to Godot.

ACCESSING A DATABASE THROUGH PHP

To access a database, you usually need the following:

- an IP address.
- a user name.
- a password.

When a database has been set up, it is often associated with a user, its password, along with access permissions. The permissions for this user define what this user can or can't do with the database (e.g., read or write). Once a database and the corresponding user have been created, tables can be created for this database; put simply, tables are similar to a spreadsheet with columns and labels for each column, and a new row for every new record. For example, we could create a table that is labeled **player** and that includes details about each player; details about each player would then be recorded in a corresponding row; these details could be, for example, an id, a password, or a high-score. As for the variables used in GDScript, each of these rows has a type: the id and the score could, for example, be integers, and the login could just be text.

So let's look at a code example that illustrates how a database could be accessed through PHP. You don't need to write this code yet; this is just provided as an example, and we will get to create such scripts later in this chapter.

```
<?php
function connect()
{
$host="localhost";
$database="mydatabase";
$user="user1";
$password = "mypassword";
$error = "Cant connect";
$con = mysqli_connect($host,$user,$password);
mysqli_select_db($con, $database) or die("Unableto connect to database");
}
?>
```

In the previous code snippet:

- We create a function called **connect** that will perform a connection to the database of our choice.
- We declare variables that will be used to establish a connection with the data-base (note that each variable starts with the **$ sign** in PHP).
- **$host**: this variable defines the address of the host or the server that we would like to connect to; in our case, we will use the server where the PHP script is stored; this is usually referred as the **localhost**.
- **$database**: this variable refers to the database that we want to access; we will see, later in this chapter, how to set up this database along with users that can access it.
- **$user**: this refers to the user we have defined for this database.

Note that several users can be granted access to a database, all with different levels of permissions. It is usually a good idea to define at least two levels of per-mis sions; for example, one user with **read-only access**, and another user with **read and write access**.

- $**password**: this refers to the password defined for the user (s) stated above.
- **$error**: this will be the error displayed if PHP can't manage to connect to the database.
- We then create a connection to the server using the variable defined

previously through the command **mysqli_connect**. This is a built-in PHP function.

- We finally select the database defined previously; if this is not possible (i.e., error when trying to access the database), we display an error message.

So at this stage, we can understand some of the code that can be used to connect to a server and a corresponding database; we will now look at how it is possible to read from a database in PHP.

In PHP, it is possible to request information from a database using what is called a query. This query is usually performed using what are usually called **SQL commands**. SQL commands make it possible to select a database and a table to, for example, read information from or write data to a table. These commands can only be processed once a connection has been established with the server and the database.

So let's say that we'd like to read all the records from a specific table; we could then use the following code:

```
$query = "SELECT * FROM players";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
for ($i = 0; $i < $n; $i++)
{
$name = mysqli_fetch_assoc($result)["name"];
$score = mysqli_fetch_assoc($result)["score"];
echo $name."\t";
echo $score."\n";
}
```

In the previous code:

- We create a new string called **$query**; this string includes an SQL query that basically selects all records from the table called **players**. The star character * indicates that all records are selected (we will see, later on, how we can focus on or select a specific record).
- We then execute this query using the command **mysqli_query**; the result of the query (the information returned by the database) is then stored inside the variable called **$result**.
- Because the table may include several records, we could obtain several records in the results and the number of records, in this case, is stored in the variable called **$n**;
- We then loop through all the records sent back from the database (in response to our query), and for each record, we access a particular attribute (or column). So the attribute **name** is saved in the variable **$name**, and the attribute called **score** is saved in the variable called **$score** using the command **mysqli_fetch_assoc**. The **mysqli_fetch_assoc** command includes two parameters: (1) the result (records) returned by the database based on the query, and (2) the attribute that we are interested in.
- Finally, we display the corresponding name and score.

Note that using SQL commands, it is also possible to sort the results based on specific criteria; for example, we could ask the database to return the name and score, and to order these in ascending or descending order of scores.

Let's look at the next code snippet to illustrate this principle.

```
$query = "SELECT * FROM players ORDER by score DESC";
```

In the previous code, the SQL query requests all records from the table players; it also requests that these should be ordered in descending order of scores.

We could also, to make this even neater, limit the result to the top three scores using the following query.

```
$query = "SELECT * FROM players ORDER by score DESC LIMIT 3";
```

In the previous code, we add the command **LIMIT** to limit the number of records returned from the database to **3**.

One of the other interesting things we can do is to write data to a database; for example, you may register a new player in the database and save or update its score; this, again, can be done with SQL queries. So let's see how this can be done.

The following example illustrates an SQL query that inserts a new player as well as its score in the database.

```
$name = "Paul";
$score = 25;
$query = "insert into players values ('$name', '$score');";
```

In the previous code:

- Two variables are declared for both the **name** and the **score** of the player.
- We then create a query that will insert a new record in the table called **players**; for this new record, the two columns for the player's name and score will be set with the values previously defined.

Now, the previous scenario may happen only when the user registers for the first time in the database; what may happen next, is that the player may need to save or update its score after each subsequent game. In this case, because the player's information has already been recorded in the database, we could use another type of SQL command, the command **UPDATE**, as illustrated in the next code.

```
$name = "Paul";
$score = 25;
$query = "UPDATE players SET score = '$score' WHERE name = '$name'";
```

In the previous code:

- We define the variables **$name** and $**score**.
- We then create a query that updates the columns **score** in the table called **players** for the record with the name **$name**. So here, we need to be more specific so that we can access a particular row; this row is identified by its attribute called **name**; so it is assumed that all players have a different name in this case.

So, as you can see, using SQL commands you can do several things including reading a table, writing to a table, or even updating a table. For the latter, you need to specify an attribute that will identify the row that you need to specifically modify; this is often an id, but it can consist of other types of variables if need be.

PASSING DATA TO A PHP SCRIPT

One of the last things that will be of help when transferring data between Godot and a database, is the ability to pass information to a PHP script; in many cases, you will want, not only to receive information and data from the script, but also to be able to pass data to the script, so that this data can be processed by the script and used to, for example, to update the database or to check for some information (e.g., login details). For example, you may want to check the login or password for a particular user, or specify what record should be updated and what data should be used in this case.

In PHP you can pass variables along with their values by adding them to the url of the PHP page as follows:

http://www.mysite.com/index.php?playerName=john&score=100

In the previous code:

- We first use the url of the PHP page and then add variables after the url of the page.

http://www.mysite.com/index.php

- A question mark is added to the url.

http://www.mysite.com/index.php?

- Then the name of the variable and its value are passed using the syntax **variable = value**.

http://www.mysite.com/index.php?**playerName=john&score=100**

- The **&** character is used between each pair of variables and values.
- So here, we have two variables (**playerName** and **score**) along with their

corresponding values (**john** and **100**).

So once this information is passed to the PHP script, it can be captured (and used) by the PHP script as follows:

```
<?PHP
$name = $_GET['playerName'];
$score = $_GET['score'];
?>
```

In the previous code:

- We define two variables **$name** and **$score**.
- The first variable is initialized with the value of the variables **playerName** passed to the PHP script through the url (as we have seen earlier).
- The same is done for the second variable, except that this time we use the variable **score**.

So that's pretty much it as far as PHP and MYSQL are concerned; there is, of course, more to SQL and PHP programming; however, to be able to access the database, the previous sections should be sufficient to understand how this works.

ACCESSING PHP FROM GODOT

So when your PHP scripts have been created and your database set up, the last thing you will need to do will be to connect to the PHP script through Godot. This can be done using the **WWW** class, a class that makes it possible to retrieve content from a url, as illustrated in the next code snippet.

```
onready var http_request = get_node("../HTTPRequest")
...
http_request.connect("request_completed",self, "post_request_completed")
...
http_request.request("http://localhost:8888/updateScore_b.php")
```

In the previous code:

- We create a new variable called **http_request** that refers to an existing **HTTPRequest** node; this type of node is used to make http requests, that is, opening pages from a specific url.
- We then connect the event called **request_completed** to a specific function, in our case the name of the function is **post_request_completed**.
- Finally, we perform a request to open the url **http://localhost:8888/updateScore_b.php**.
- Once the server has processed this request, the function **post_request_-completed** will be called and we can use it to process the information sent to us by the server.

SETTING UP YOUR SERVER

Great. So I hope that the whole principle behind accessing a database from Godot is clearer now (and if it's not, the next section will help you to put this knowledge into practice). So in the next sections, we will create a simple system whereby:

- A player will be asked for his/her id.
- If the id is already in the database, then the database will be updated with the new score.
- If the id is not in the database, then a new record will be created for this player with an id of his/her choice, along with the new score.
- Once the score has been saved, the top 5 scores (i.e., the high scores) will be displayed on screen with their corresponding names.

So the workflow will be as follows:

- Set up the database and associated users.
- Create a new table with some players and corresponding scores.
- Create PHP scripts that will be able to either insert, update, or read data in the database we have created.
- Save these scripts on the server.
- Design a simple user interface in Godot whereby players can enter their id or register.
- Access the PHP scripts that we have created previously in order to perform the previous tasks.

So first, let's set up your server. In order to execute the PHP scripts, these scripts need to be saved on a dedicated server. This server will also make it possible to create and set up your database. If you already have your own website, the chances are that you also have different packages installed that make it possible to set up a database, as described in the next figure.

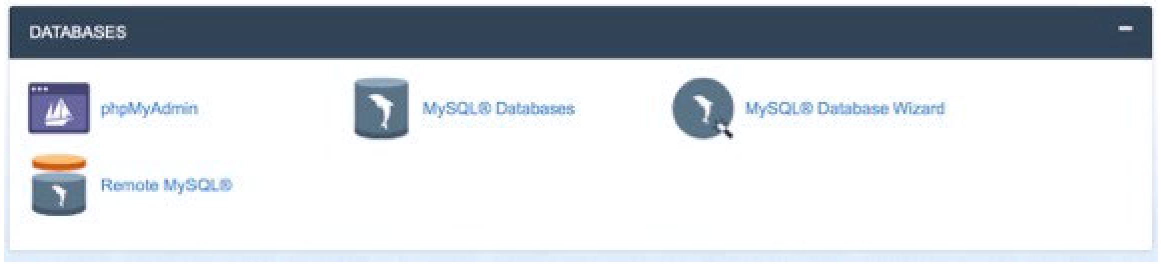


Figure 1: Database tools included in web packages

For example, in the previous figure, you can see some of the standard tools offered as part of your control panel (if you have a control panel included in your website package), to manage databases, including **PHPMyAdmin**, and **MySQL Databases**. The former will make it possible to create new databases, while the latter will make it possible to create and update new tables.

Now, you may not have a website with these tools, and that's perfectly fine too. In fact, if you have never done any server-side scripting before, I would suggest that you use a local server for the time being. You could, in this case, use a tool called **WAMP** (if you are using a windows machine) or **MAMP** (if you are using a Mac). This software includes an Apache server, one of the world's most used webservers, and PHP (so that you can run PHP scripts), along with MySQL. MAMP is available from the official website: https://www.mamp.info/en/.

AMP stands for **A**pache (the webserver), **M**ySQL, and **P**HP. It is a package that includes all the elements needed to run PHP scripts and access a MYSQL database. I would suggest that you use this package (instead of a webserver) for the next sections of the chapter, as it should make the creation of your database much easier.

When you are using this software (i.e., M-AMP or W-AMP), the only difference is that your server is hosted on your own computer (rather than online), which makes testing easier, because you do not need to be connected to the Internet in order to access the server or connect to the database. Once your code has been tested locally (i.e., using WAMP or MAMP), you can then create the database on the remote server and upload your PHP code so that the application runs "live", and so that it is accessible from anyone through the Internet.

Whether you are using a local server (i.e., MAMP or WAMP) or a remote server, you will connect to the server using the address **localhost** or **127.0.0.1** because the pages will be hosted where the server is (either on the remote or the local server).

So, if you are using a remote server (but again, I suggest that you use W/MAMP instead, as explained in the next sections), you will need to do the following:

- Create a new database using the MySQL Databases tool
- Create a user with sufficient rights to access the database and complete some commands such as **READ, WRITE** or **UPDATE**.
- Add tables and their content to the database using **PHPMyAdmin**.

Creating your database using a remote server

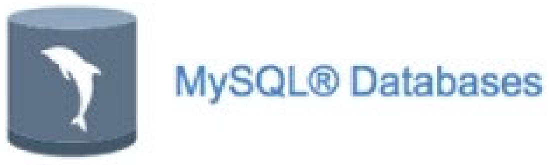
You can skip this section if you are using (or prefer to use) a local server (e.g., W/MAMP). In fact, I would strongly suggest that you skip to the next section, called **Creating a database using a local server**, if you don't already have some experience with PHP or PHPMySQL or setting up a remote server.

So, let's get started!

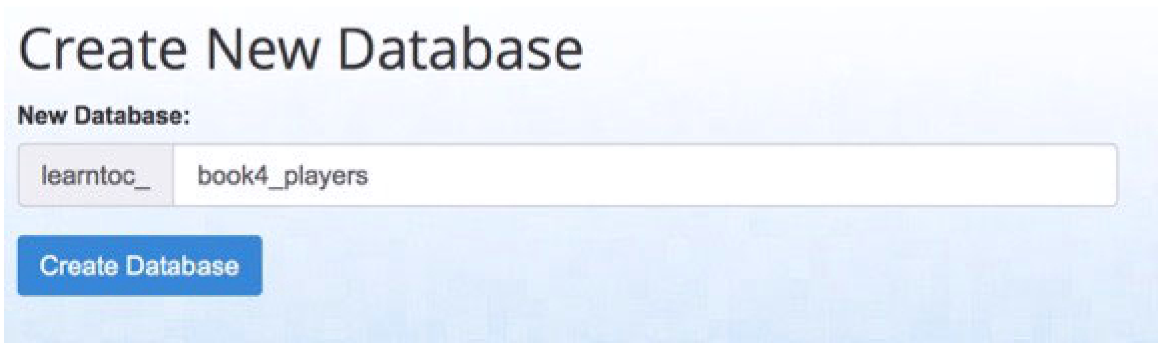
Please note that the look and feel of the control panels may differ across hosting companies; however, the principles and the steps explained in this section should remain relevant.

To create your database:

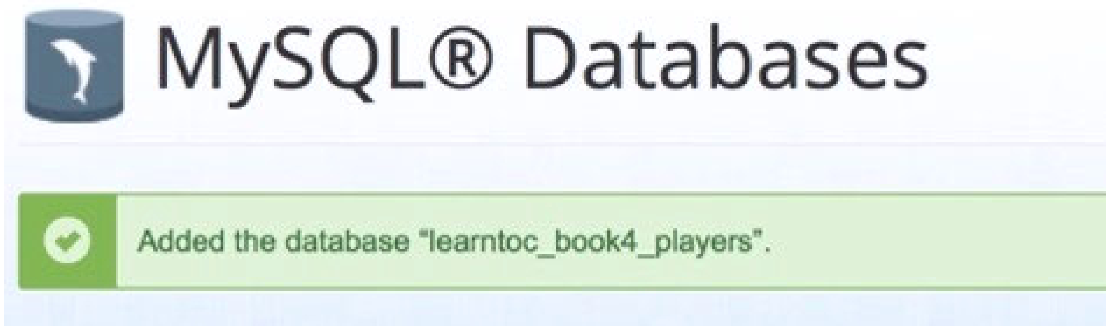
- Connect to your website.
- Open your control panel.
- Click on **MySQLDataBase** tool.



- Once the new page opens, enter the name of your database and click on **Create Database**.

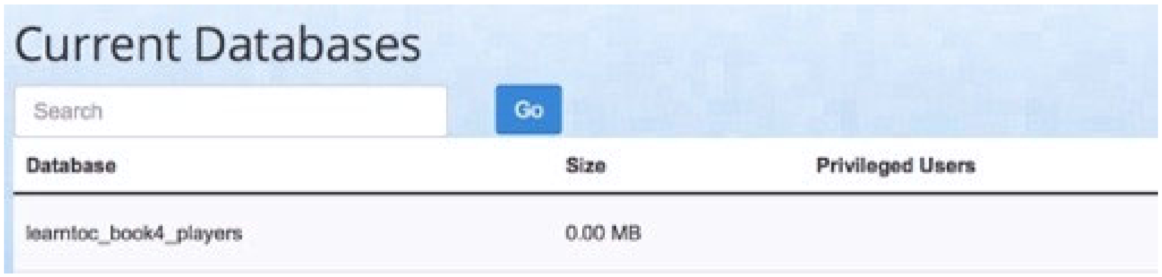


- The following page will then be displayed to acknowledge that the new database has been created.



Added the database "learntoc_book4_players".

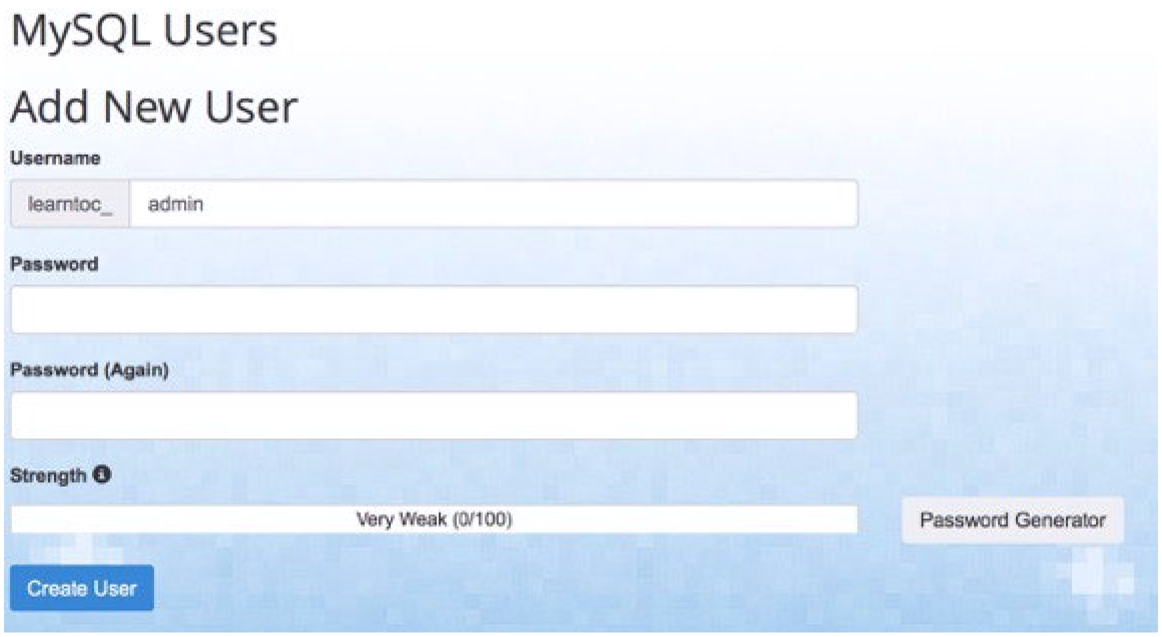
- You can then click on the button labeled **Go Back**, to return to the previous page.
- At this stage, you should see that your new database has been created, as described on the next figure.


Current Databases

Database	Size	Privileged Users
learntoc_book4_players	0.00 MB	

However, you will see that it doesn't have any associated users who could be using it, or corresponding access rights. So we will need to add such users:

- Please scroll down to the section called **MySQL Users**.
- Create a new user by setting a **name** and a **password**.


MySQL Users

Add New User

Username: learntoc_ admin

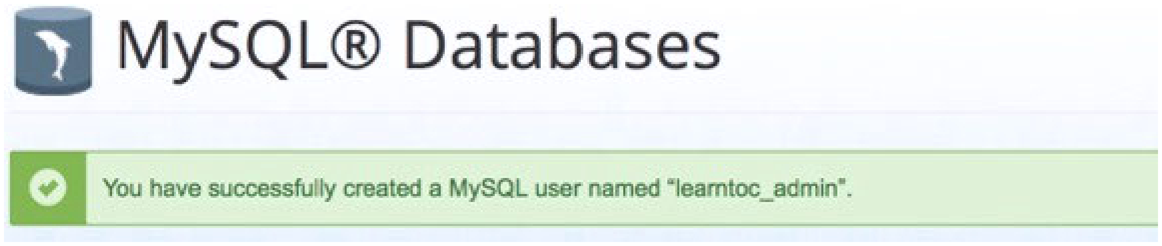
Password:

Password (Again):

Strength: Very Weak (0/100)

Please take note of the name of the user, along with its password, as we will need this information later on when accessing the database through PHP.

- Once this is created, a confirmation page should be displayed as follows.

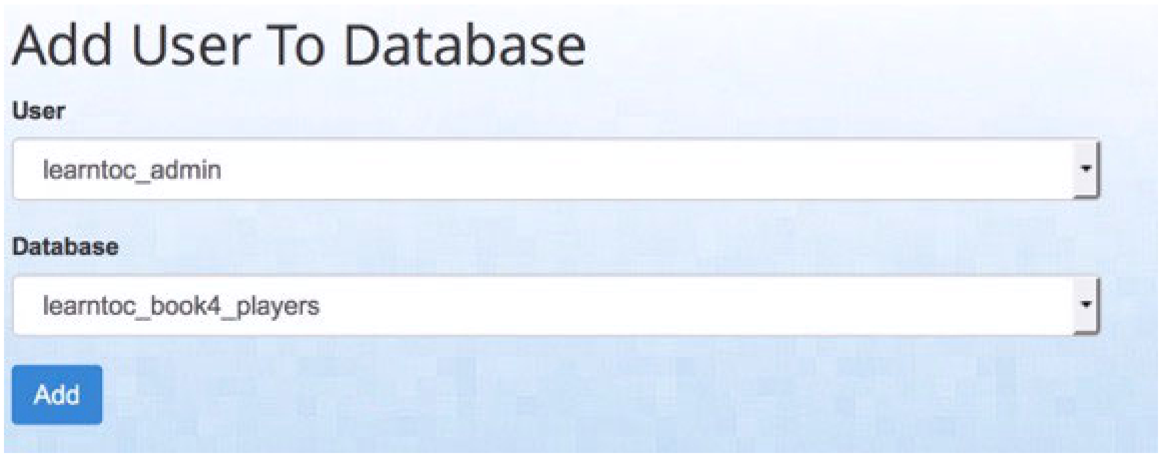

MySQL® Databases

You have successfully created a MySQL user named "learntoc_admin".

- You can then click on the button labeled **Go Back**, to return to the previous page.

The last thing we need to do now is to associate this new user to the database that we have just created:

- Please scroll down to the section called **Add User to Database**.


Add User To Database

User

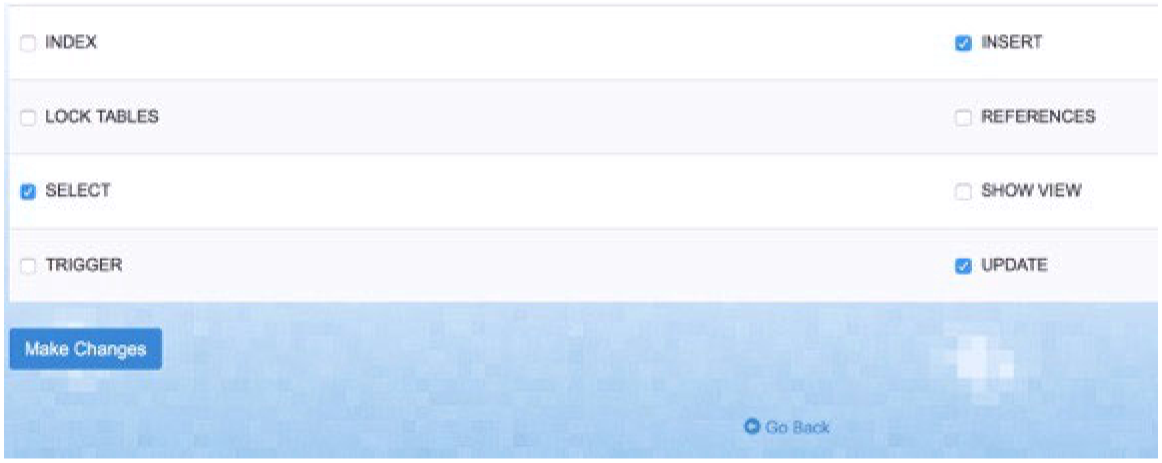
learntoc_admin

Database

learntoc_book4_players

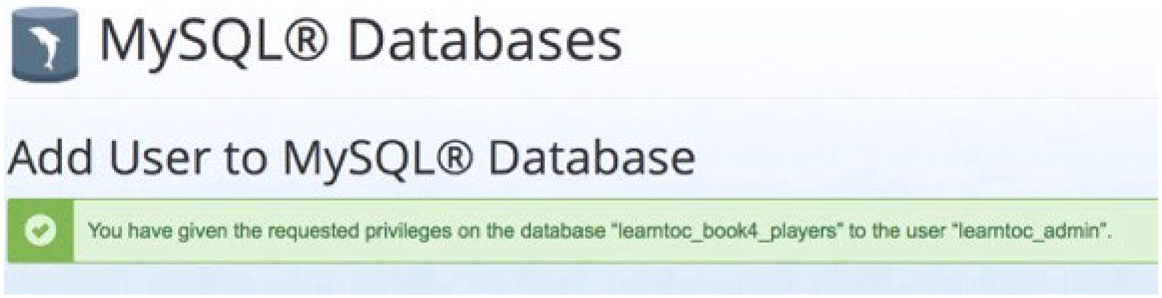
- Make sure that the correct user and database are selected (using the drop-down menus). For example, in the previous figure you can see that I am setting privileges for the user **learntoc_admin** (created previously) to be linked to the database **learntoc_book4_players**.

- Then click on the button labeled "**Add**".
- This will open a new window labeled **Manage User Privileges**.



☐ INDEX
[✓] INSERT
☐ LOCK TABLES
☐ REFERENCES
[✓] SELECT
☐ SHOW VIEW
☐ TRIGGER
[✓] UPDATE

- In this window, select the privileges **SELECT**, **INSERT** and **UPDATE,** as described on the previous figure. These are the key commands that we will need to perform.
- Once this is done, click on the button labeled "**Make Changes**".
- The following window should then appear:



MySQL® Databases

Add User to MySQL® Database

You have given the requested privileges on the database "learntoc_book4_players" to the user "learntoc_admin".

- Please click on the button called "**Go Back**"

So at this stage, if you are using a remote server (i.e., your web host), you have managed to create a database, an the associated user, along with access rights for this user: so we are almost there :-).

Creating a database using a local server

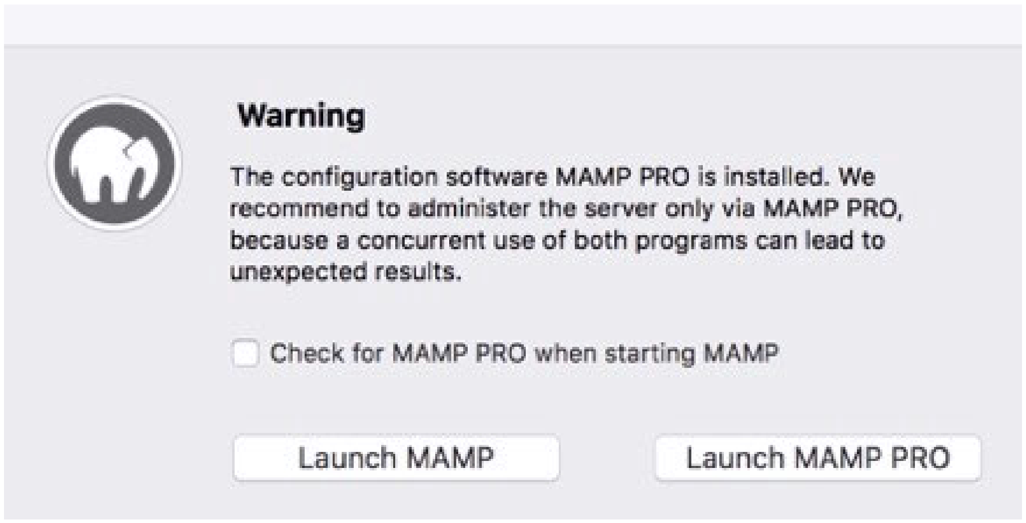
So if you are using a local server (i.e., W/MAMP), the steps to create your database and associated user will be slightly different; however, the idea and principles will remain similar.

- Please download the software **MAMP** (if you are using a Mac) or **WAMP** (if you are using a Windows-based computer) from the following site:

https://www.mamp.info/en/downloads/.

- Install it on your computer following the instructions.
- Once it is installed you can launch it, and the following (or similar window) will appear.

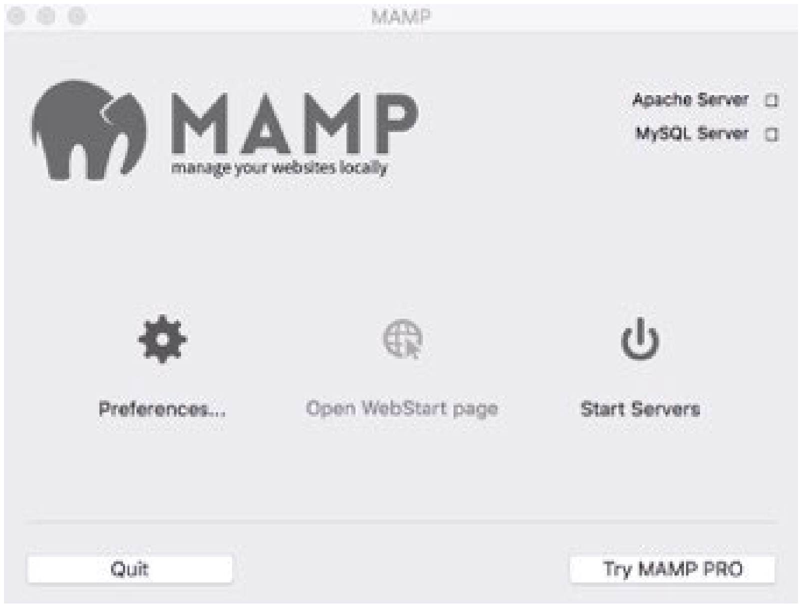
Bear in mind that the examples provided in the next section are from a Mac Operating System (i.e., using MAMP), so the screen on Windows computers may differ slightly; however, the layout should be similar.


Warning

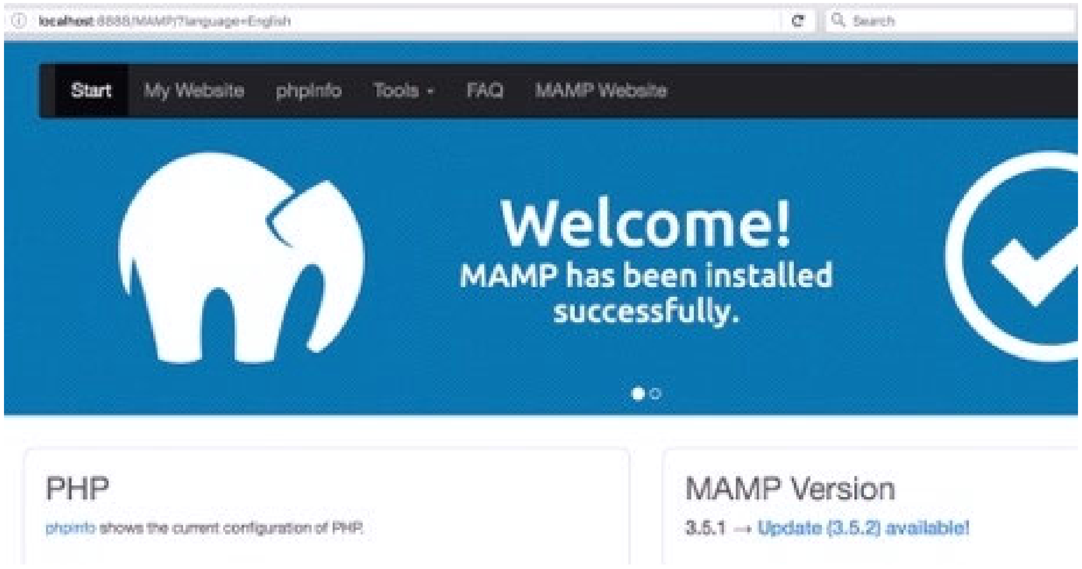
The configuration software MAMP PRO is installed. We recommend to administer the server only via MAMP PRO, because a concurrent use of both programs can lead to unexpected results.

☐ Check for MAMP PRO when starting MAMP

- Launch the application by clicking on "**Launch MAMP**".



- In the next window, click on **Start Servers** (as described in the previous figure).
- After a few seconds, you should see that the icons located in the top-right corner of the window are ticked (green) indicating that the Apache Server and MySQL servers have started.
- A new page will also open in your browser, as described in the next figure.


localhost:8888/MAMP/?language=English
Search
Start
My Website
phpinfo
Tools
FAQ
MAMP Website
Welcome!
MAMP has been installed successfully.

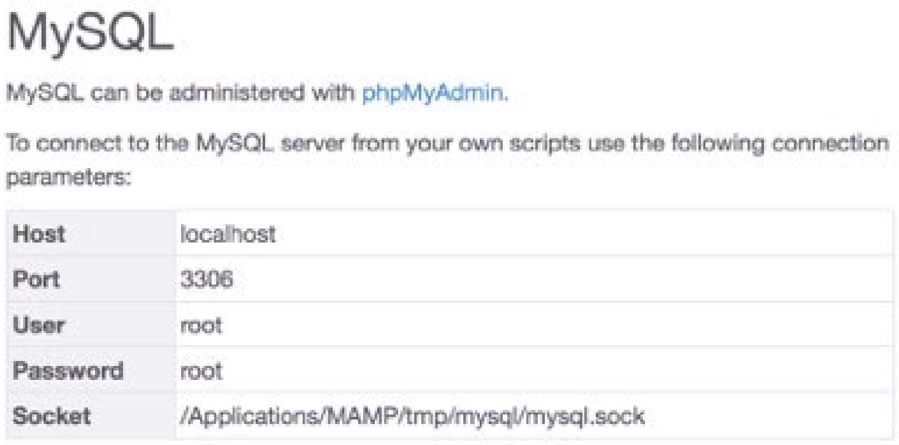
PHP

phpinfo shows the current configuration of PHP.

MAMP Version

3.5.1 → Update (3.5.2) available!

This page includes some important information, including a default host name, a user name, and a password for this user to access the server through PHP, along with a PHP example.

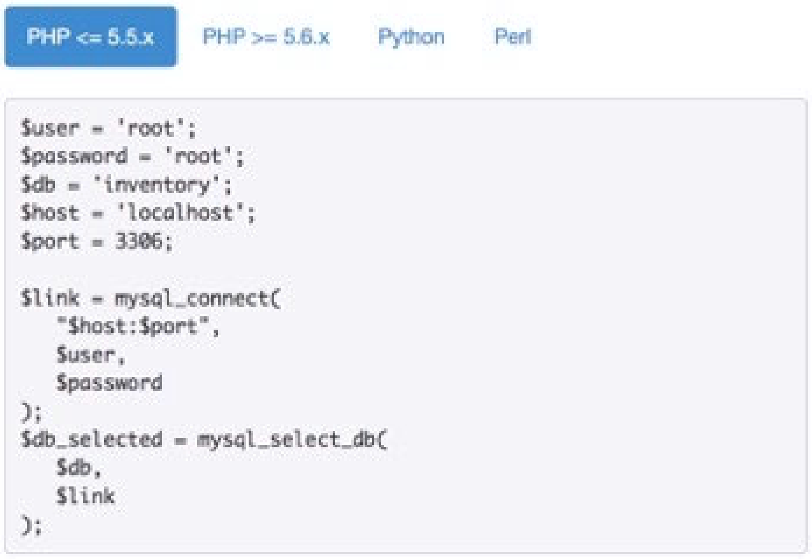


MySQL

MySQL can be administered with phpMyAdmin.

To connect to the MySQL server from your own scripts use the following connection parameters:

Host	localhost
Port	3306
User	root
Password	root
Socket	/Applications/MAMP/tmp/mysql/mysql.sock



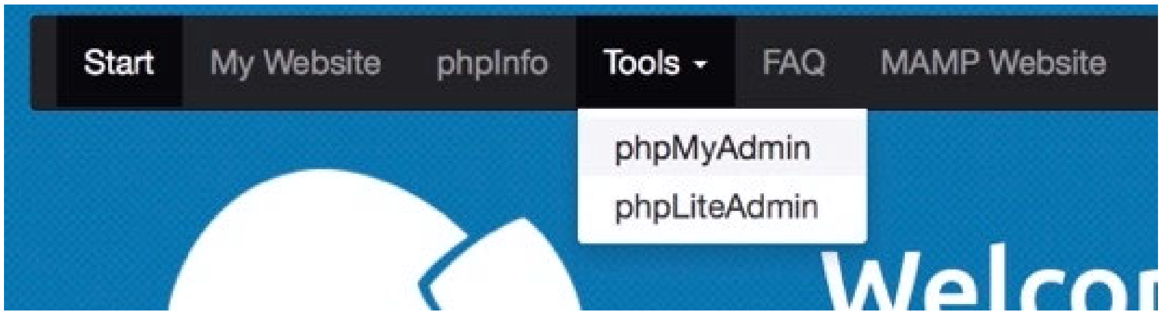
```
$user = 'root';
$password = 'root';
$db = 'inventory';
$host = 'localhost';
$port = 3306;

$link = mysql_connect(
   "$host:$port",
   $user,
   $password
);
$db_selected = mysql_select_db(
   $db,
   $link
);
```

So, at this stage, our servers are running; the only thing that we need to do is to create a database; note that because this set up will be used locally, we can use the default user **root** which has, by default, all access privileges to the database (e.g., READ, WRITE, UPDATE, and SELECT, etc.). So let's create a new database:

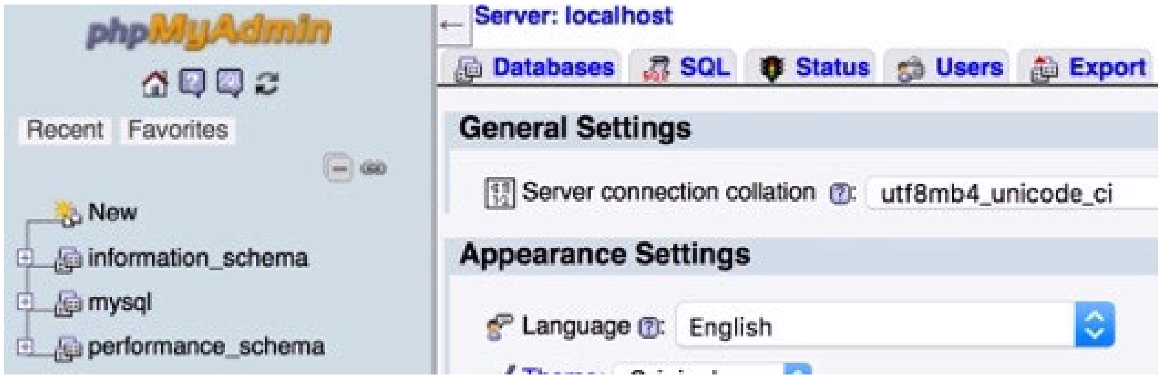
- Select the tool **PHPMyAdmin** from the drop-down menu located at the top of the browser in the page that is already open (as described in the next figure).

If the MAMP window is not opened yet, you can open it using the url http://localhost:8888/MAMP/?language=English



Note that you can also use the address http://localhost:8888/PHPMyAdmin/ to access PHPMyAdmin on your computer. This will open a new window with information on your MYSQL database.

- Click on the tab labeled **Databases**.



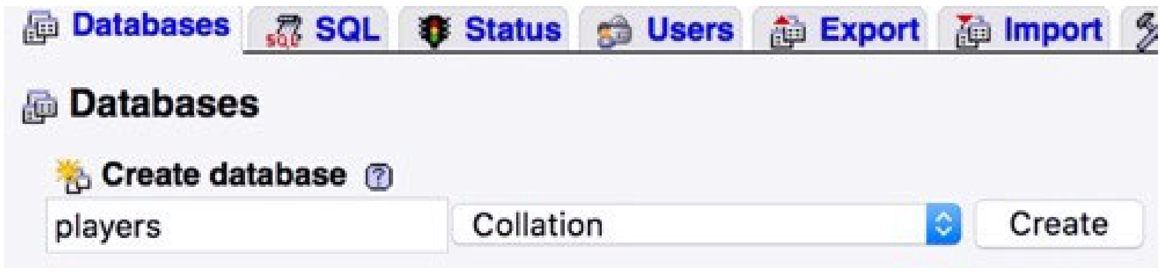
General Settings

Server connection collation: utf8mb4_unicode_ci

Appearance Settings

Language: English

- Give a name to your database, for example, **players**, and click on the button labeled **Create** (leave the **Collation** default option).

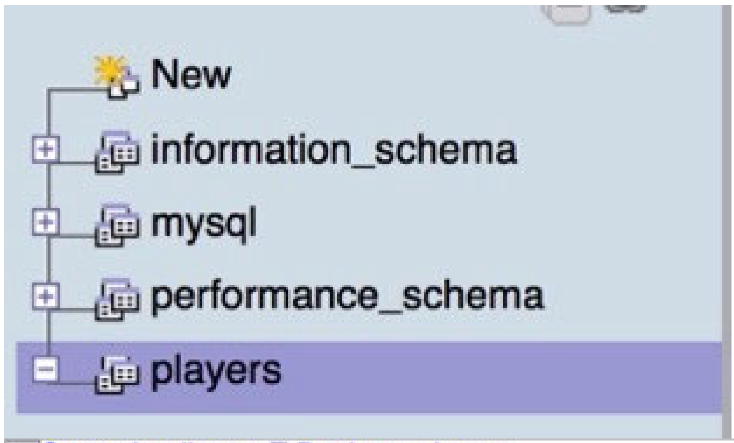


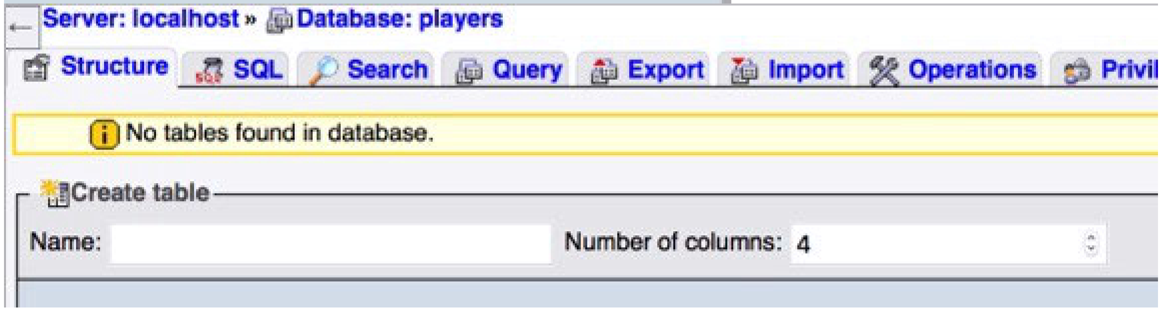
Databases

Create database

players | Collation | Create

- At this stage, as the database has been created, you will see it listed in the left frame of the window, as well as at the top of the window.





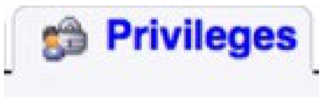
No tables found in database.

Create table

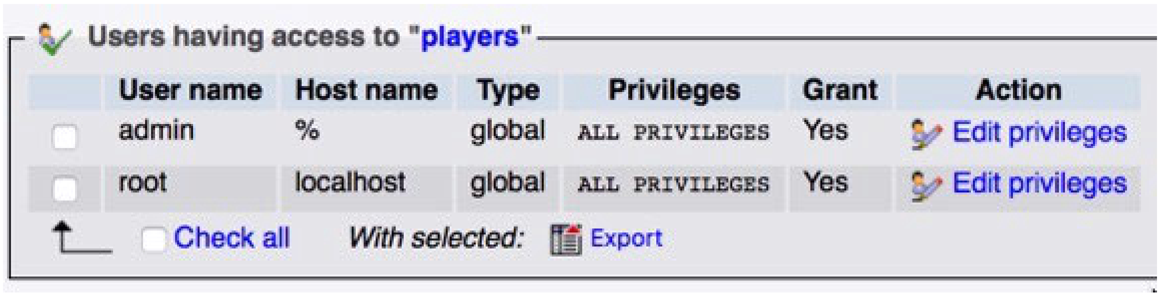
Name:

Number of columns: 4

- Please click on the tab called **Privileges**.



You should see that, by default, a user **root** has already been created; the default password for this user is **root**.



Users having access to "players"

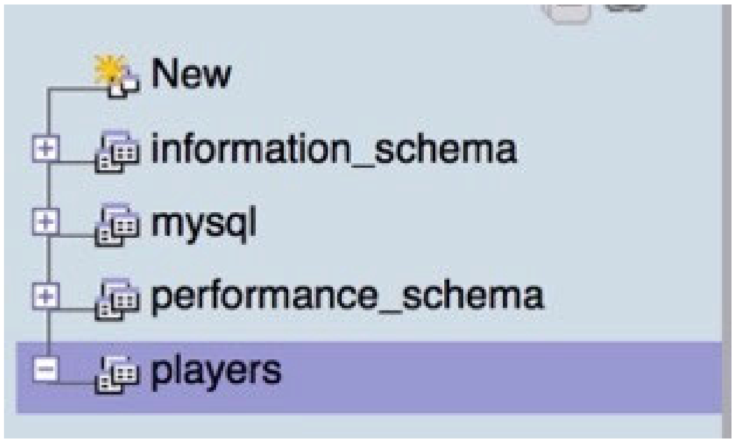
	User name	Host name	Type	Privileges	Grant	Action
☐	admin	%	global	ALL PRIVILEGES	Yes	Edit privileges
☐	root	localhost	global	ALL PRIVILEGES	Yes	Edit privileges

So at this stage, we have managed to create a database on the local server (or the remote server if you have followed the previous section), and it is time to create a table that will hold information about the players' scores.

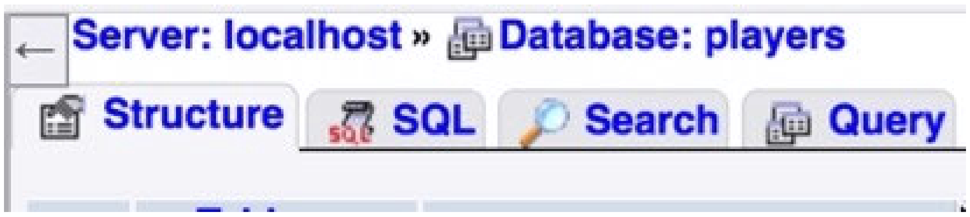
CREATING NEW TABLES

The following can be performed whether you use a local or a remote server. The only difference is the way you access **PHPMyAdmin;** you can use the address http://localhost:8888/PHPMyAdmin/ if you are using a local server.

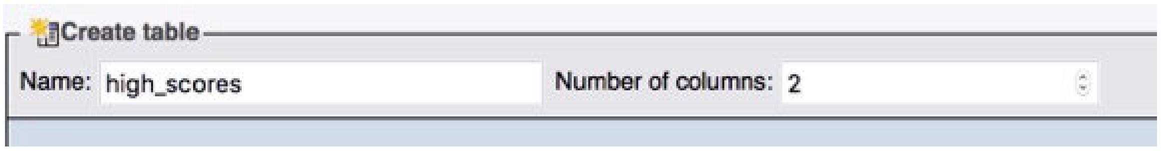
- Please open **PHPMyAdmin**.
- Select your database (i.e., click on **players** in the left-hand menu).



- Select the tab called **Structure**. This section makes it possible to define the structure of a table including the name and the type of the columns within.



- Within this tab, specify a new name for the new table, for example, **high_scores**, and a number of columns (**2**: one for the name and one for the score), as illustrated in the next figure.


Create table

Name:	high_scores	Number of columns:	2

- Click on the button labeled **Go**, located at the bottom of the page, to complete the creation of the table.
- In the new window, that will be used to specify the name and type of the variables stored in the table, please enter the following information:

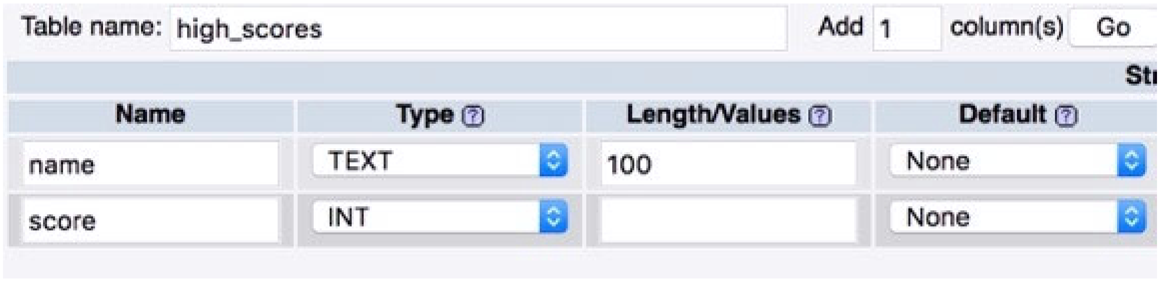


Table name: high_scores Add 1 column(s) Go

Str

Name	Type	Length/Values	Default
name	TEXT	100	None
score	INT		None

First row:

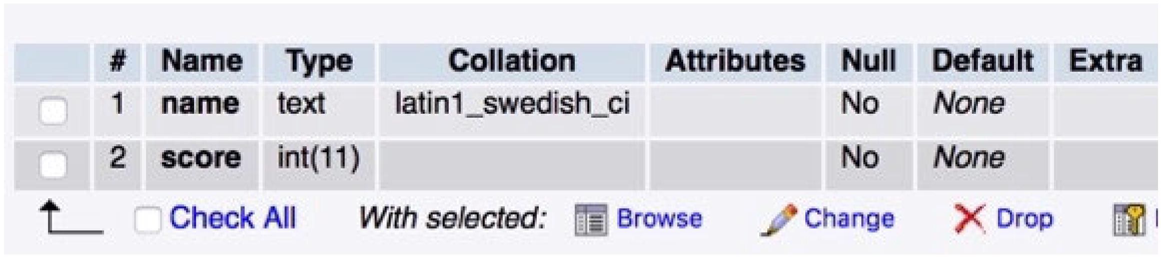
- **Name = name**
- **Type = TEXT**
- **Length/Values = 100**
- Leave all other options as default.

Second row:

- **Name = score**
- **Type = INT**
- Leave all other options as default.

Once this is done, please click on the button labeled **Save**, located in the bottom-right corner of the window, as illustrated in the next figure.

- Then the following window will appear, indicating that the table was successfully created; it should include the columns that we have defined previously (e.g., name and score).

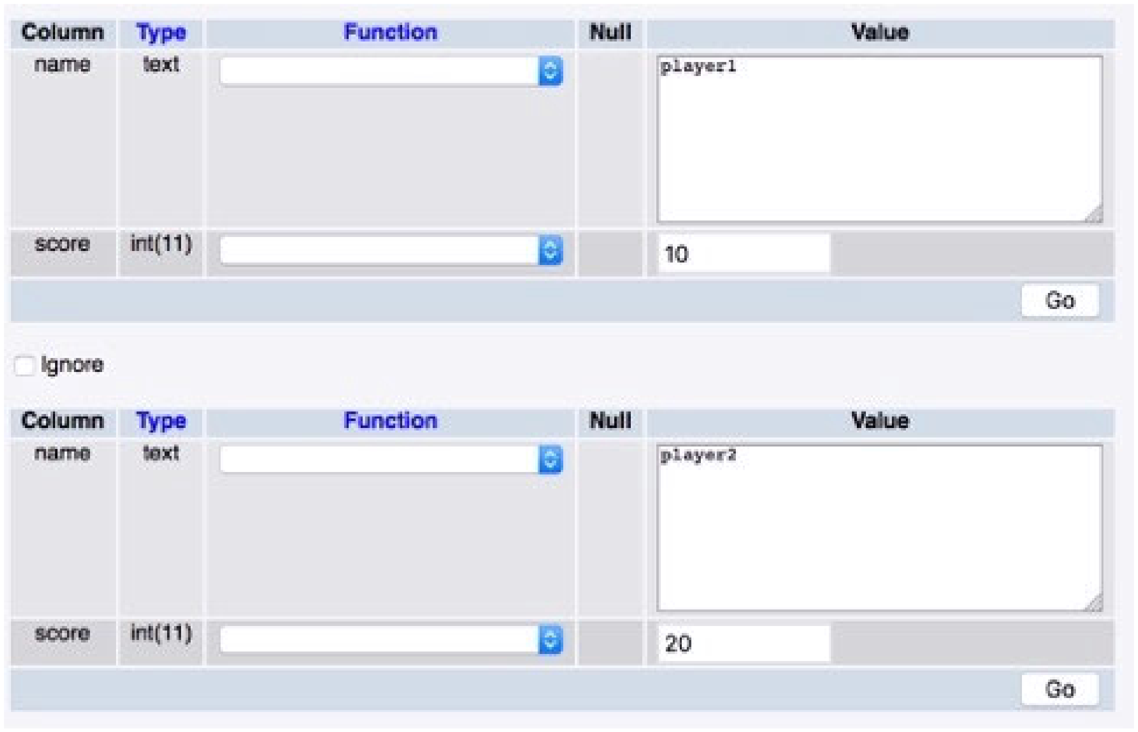


	#	Name	Type	Collation	Attributes	Null	Default	Extra
	1	name	text	latin1_swedish_ci		No	*None*	
	2	score	int(11)			No	*None*	

So at this stage, we just need to add information to this table so that we can access it and test it accordingly from PHP.

- Please select the tab called **Insert** from the top of the window; this section is used to insert rows (i.e., data) into our table.

- Insert two new records as follows:



Column	Type	Function	Null	Value
name	text			player1
score	int(11)			10

Go

☐ Ignore

Column	Type	Function	Null	Value
name	text			player2
score	int(11)			20

Go

First Record:

- **Name = player1**
- **Score = 10**

Second Record:

- **Name = player2**
- **Score = 20**

You can then click the button labeled **Go** located after the second record.

Note that you can add one or several records (or rows) at a time. This can speed up the process of setting up your tables.

Although we don't need it now, for those used to SQL commands, the following could have been used within the SQL window (i.e., the SQL tab). This will have the same effect as creating the previous record manually.

INSERT INTO `players`.`high_scores` (`name`, `score`) VALUES ('player1', '10'), ('player2', '20')

So now that this has been done, it is time to use some PHP code to access the database.

Creating and running your PHP script

In this section, we will locate where to create and execute your PHP script on your computer, and then write the necessary code to access our database from your script.

If you are using a remote server, your script can be added anywhere within the **public_html** or **www** folder; this folder is usually the folder where all webpages to be accessed by Internet users can be seen and/or executed. So, if your script is called **accessDB.php** and is stored in the **public_html** folder, then it will be accessible using: **http://yoursiteaddress/accessDB.php**. In order to add this file to the **public_html** folder, you can either use file managers included in your web package or an ftp client.

An FTP client is a software that makes it possible to connect to your website and to transfer files from or to it. If you are using an ftp client, you will need to connect using the details provided by your host (the name of the server, along with an ftp user and password).

If you are using WAMP or MAMP, the location for your PHP files (also referred as the www folder) will be as follows;

- For MAMP (Mac OS) the folder is: **Applications/Mamp/htdocs**
- For Wamp (Windows) the folder is: **C:\wamp\www\**
- From now, this folder will be referred as the **www** folder, whether you are using a local or remote server.

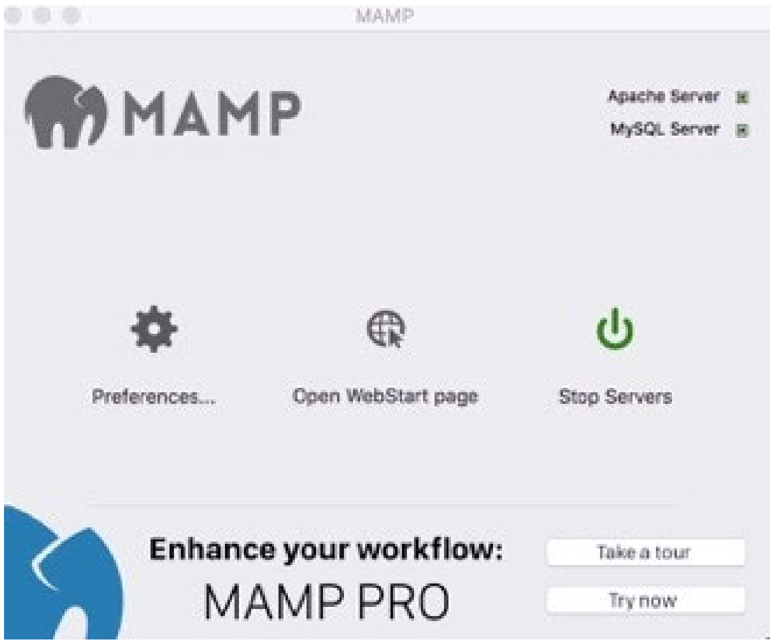
So, if your script is called **accessDB.php** and is stored in the **www** folder, it will be accessible using: **http://yoursiteaddress/accessDB.php**, if you are using a remote server, and **http://localhost:8888/accessDB.php** if you are using a WAMP or MAMP server.

When using a local server through MAMP or WAMP, the url used to access your PHP script includes the address of the server (i.e., **localhost**), as well as a default port for AMP (**8888**). The port can be compared to a phone line through which the server listens to and replies to queries.

Next, we will be specifying the PHP version that we will be working with; for the code introduced in this chapter, version 5 of PHP will be used, although if you read this book a few months or years after it has been released, this version may or may not be available. There are slight differences across PHP versions; this being said the concepts introduced in this chapter shall remain similar across versions (e.g., +/- minor necessary changes).

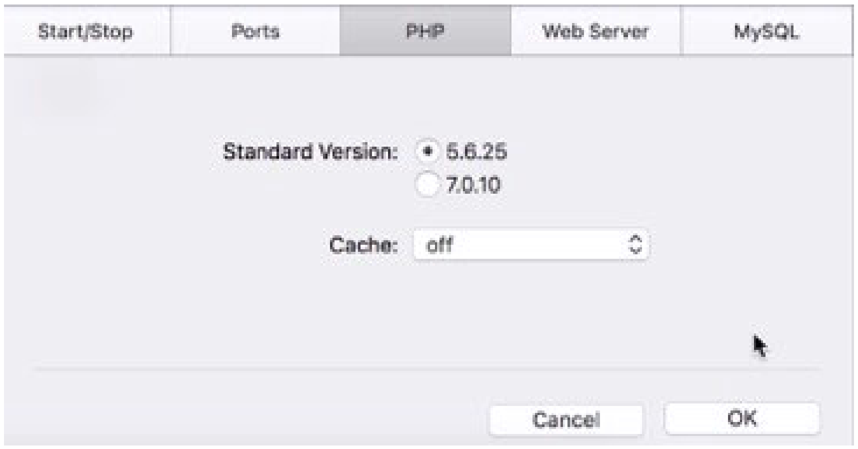
If you are using a local server:

- Open the MAMP admin.
- Click on the icon called **Preferences**.



- In the new window, select the tab called **PHP**.
- Select the Standard **Version 5.6.25**.

Please note that these PHP versions may differ over time; however, the principle explained in the next pages should remain similar.

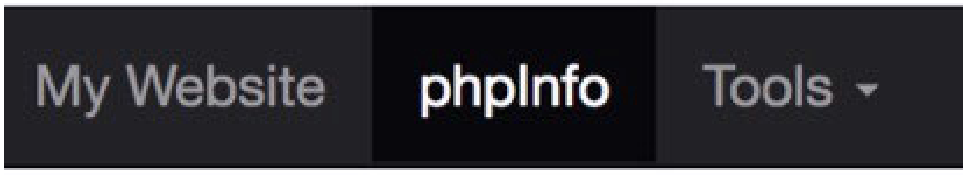


Standard Version: [✓ 5.6.25] 7.0.10

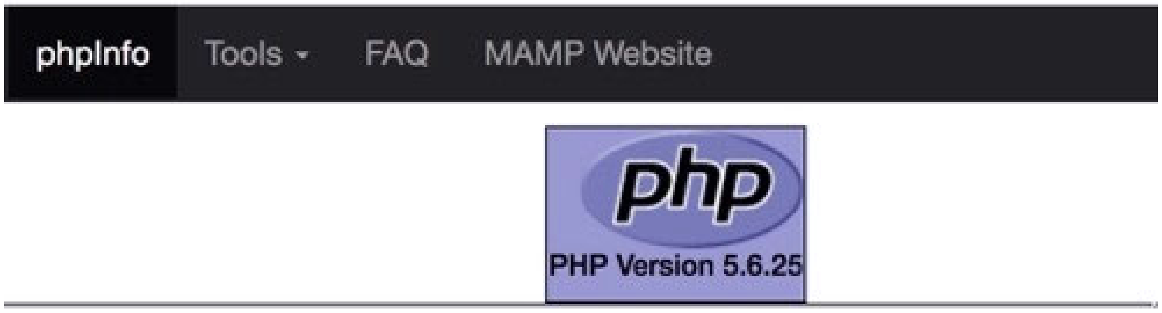
Cache: off

- Click **OK**.
- The server will then restart; you can then check that the correct version is

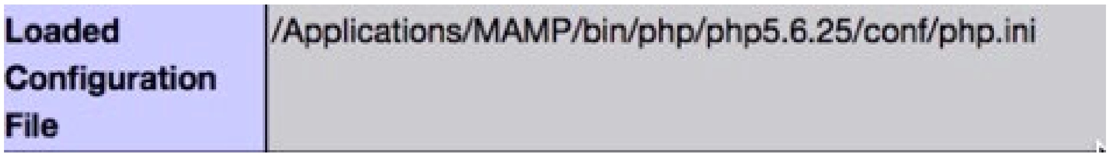
running by opening the url: **localhost:8888/MAMP/** in your browser, and then by clicking on the tab called **phpinfo**.



This should display your PHP version at the top of the page.



Next, we will also make sure that any PHP error is displayed on screen, so that we can see if and where these occur; to do so, we will need to modify a file called **PHP.ini**; the location of this file is provided in the **phpinfo** window in the section labeled **Configuration File**. So if you open the url **localhost:8888/MAMP/**, and then click on the tab called **phpinfo**, you should see the corresponding section, as described in the next figure.



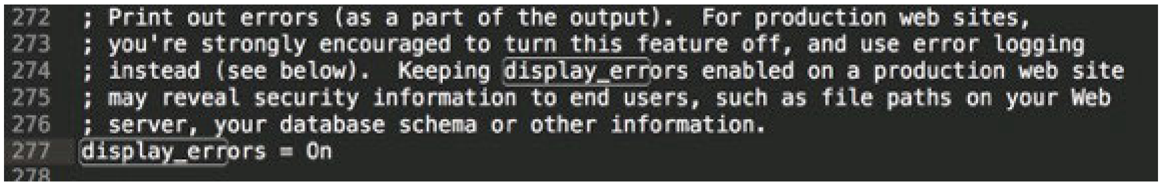
Loaded Configuration File	/Applications/MAMP/bin/php/php5.6.25/conf/php.ini

Please note that the previous figure shows the path to the file **PHP.ini** on a Mac OS computer; the path for a windows machine should be different.

So we can now access and modify this file using the path provided by **phpinfo**.

- Please open this file (i.e., **php.ini**) using a text editor of your choice.
- Search for the text **display_errors**.

- Once you have found it, please replace the text **display_errors = Off** with **display_er rors = On,** as described in the next figure.

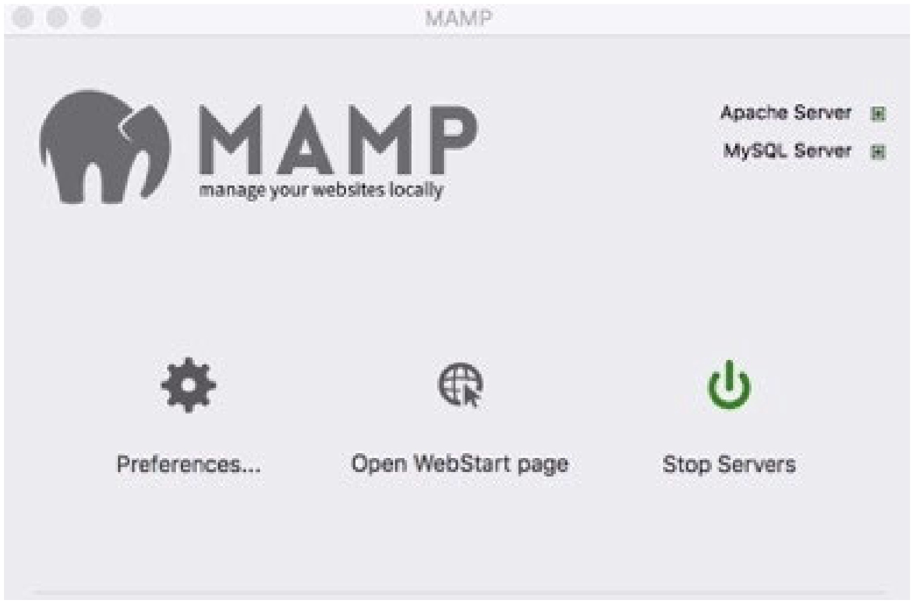


```
; Print out errors (as a part of the output).  For production web sites,
; you're strongly encouraged to turn this feature off, and use error logging
; instead (see below).  Keeping display_errors enabled on a production web site
; may reveal security information to end users, such as file paths on your Web
; server, your database schema or other information.
display_errors = On
```

- Please save the file.

So that this change can be applied, we need to restart the server:

- Please stop the server using the **AMP** control panel.
- And restart it again.



So let's test our PHP settings:

- Please create a new file with the text editor of your choice, and save it as **updateScore.php** inside the **www** folder.

For Mac OS computers, the **www** folder is located in **Applications/ Mamp/htdocs**

For Windows computers the **www** folder is located in **C:\wamp\www\**

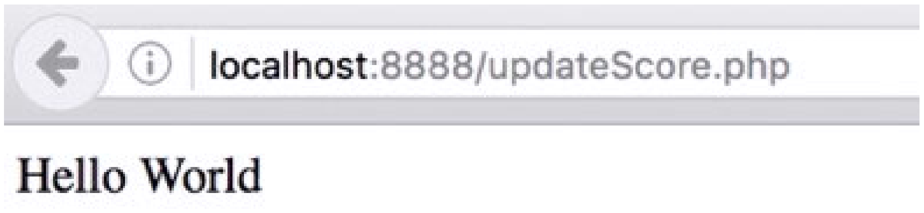
- Add the following code to it.

```
<?php
echo "Hello World";
?>
```

- Once this is done, save the file, and open this page in your browser, for example, by typing the following address.

http://localhost:8888/updateScore.php

- If you are using a remote server, the url may look like: **mywebsever.com/updateScore.php**.
- This should display the text "**Hello World**".



Hello World

Once this is done, we can try to connect to our database. Note that if you are using a remote server, you will need to use the information that you created when initially you set up the database.

- Please add the following code (in bold) to the file **updateScore.php**.

```
<?php
echo "Hello World";
$host="localhost" ;
$database="players";
$user="root";
$password = "root";
$error = "Cant connect";
$con = mysqli_connect($host,$user,$password);
mysqli_select_db($con, $database) or die("Unableto connect to database");
?>
```

In the previous code:

- As we have seen in the previous sections, we connect to our database using the default user and password for our localhost.
- If no error message is displayed, then the connection has been successful.

Next, we will try to read some records from the database.

- Please add the following code to the PHP script:

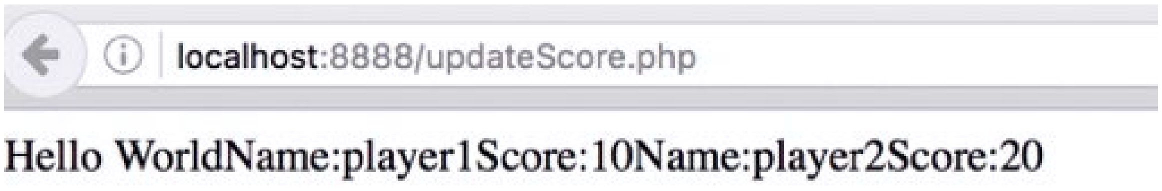
```
$query = "SELECT * FROM high_scores";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
while ($row = mysqli_fetch_assoc($result))
{
$name = $row["name"];
$score = $row["score"];
echo "Name:".$name;
echo "Score:".$score;
```

}

- Save your PHP file.
- Switch to your browser and refresh the following page.

http://localhost:8888/updateScore.php

- The following should be displayed:



```
Hello WorldName:player1Score:10Name:player2Score:20
```

If you see this output, you have successfully managed to access and read data from the database. Congratulations!

GATHERING DATA FROM GODOT

Now that we know that our PHP script works and that we can access the database, we will modify it and create a new script in Godot so that we can read and display this information in Godot.

- Please comment the following line in your PHP script, as follows:

```
//echo "Hello World";
```

In PHP, you can comment your code using **//** to comment only one line or **/* and */** at the start and the end of the section that you would like to comment.

Now that it is done, let's work on the Godot side of things and try to access this PHP script and gather its output (i.e., the list of players and their score) from Godot.

- Please launch Godot.
- Create a new Scene (**Scene | New Scene**).
- In the new window, select the option **2D Scene**.
- This will create a new scene with a node called **Node2D** by default.
- Please add a new node of type **HTTPRequest** as a child of the node **Node2D**.
- Add a new script to this node (i.e., **HTTPRequest**) and name this script **HTTPRequest** (this is the default name for the script).
- Open this script.
- Add the following code to it (new code in bold).

```
extends HTTPRequest
func _ready():
connect("request_completed", self, "_on_request_completed")
request("http://localhost:8888/updateScore.php")
func _on_request_completed(result, response_code, headers, body):
```

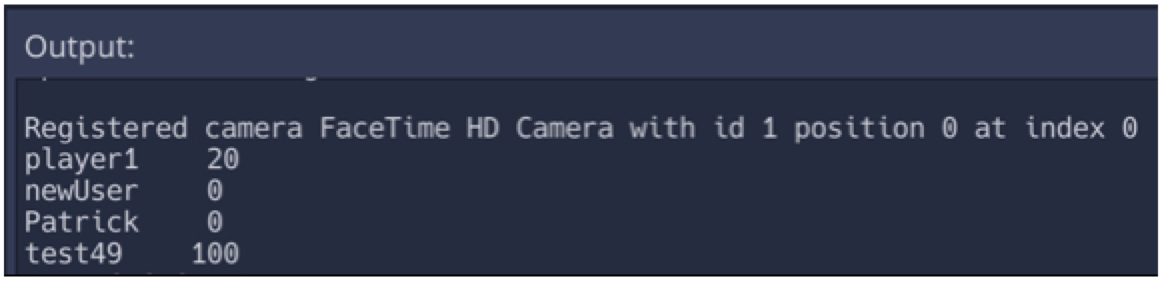
var message = body.get_string_from_utf8()

print(message)

In the previous code:

- We declare two functions.
- In the first function called **_ready**, we connect the event **request_completed** to the function **_on_request_completed**, and we then open the url **http://localhost:8888/updateScore.php**.
- We then define the function **_on_request_completed**; this function is called when the data has been received from the server following the previous request.
- In this function, the data received is saved in the variable called **message**, and then displayed accordingly in the **Output** window.

Please, save your scene as **db.tscn** (**Scene | Save Scene As**). When this is done, you can make sure that the MAMP server is running, and play the scene, and check the **Output** window which should look like the following figure.



```
Output:

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
player1    20
newUser    0
Patrick    0
test49    100
```

So if you see this (or a similar) message, you have successfully managed to connect to the database and to read its content from Godot. Well done!

Now, while we have managed to collect this information, we could try to display it in a text field; this will mean that we need to create a field, write this confirmation to it, but also format our text, as for the time being, the text gathered from the PHP script may not be easily readable.

- Please open the PHP script that you have created earlier.
- Modify it as follows.
- Change the code...

```
echo "Name:".$name;
echo "Score:".$score;
```

- to...

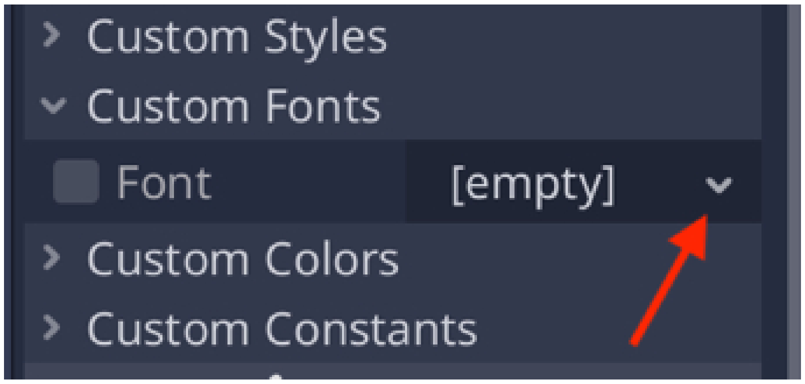
```
echo $name."\t";
echo $score."\n";
```

In the previous code:

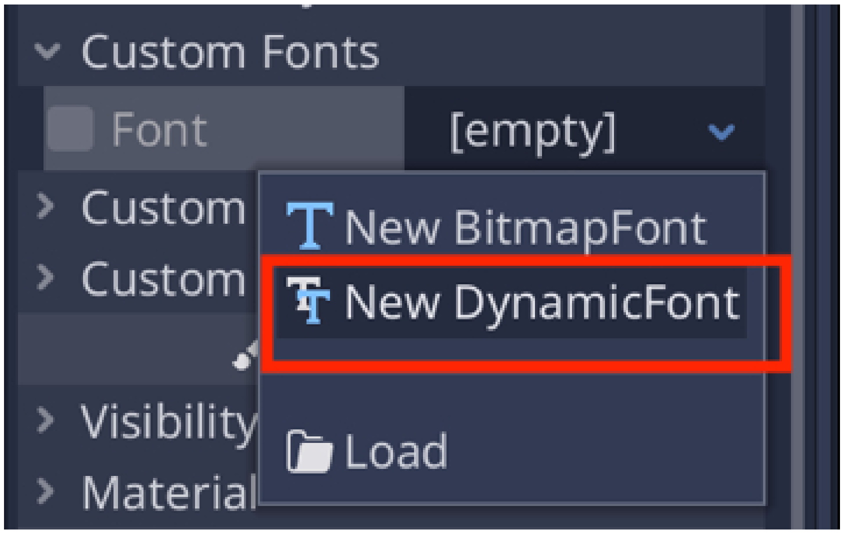
- We have modified the line of code that displays the name by adding the character **"\t"** which is a tabulation.
- We have modified the line of code that displays the score by adding the character **"\n"** which includes a line break.

Now that this change has been made, we can save the PHP script and focus on our script.

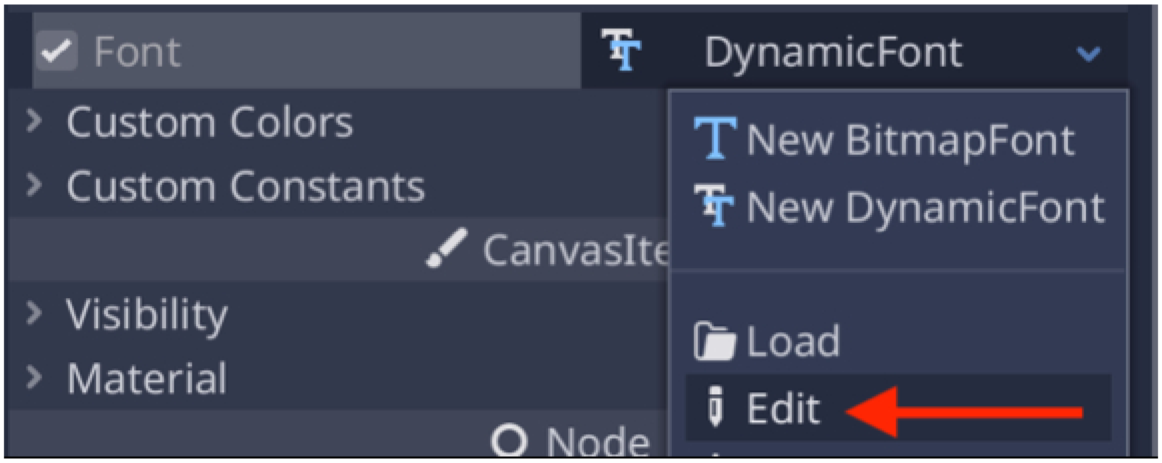
- Please save the PHP script (**CTRL + S** or **CMD + S**).
- Switch back to Godot.
- Add a new node of type **TextEdit** as a child of the node **Node2D**, and rename this new node **txt_field_name**.
- Using the **Inspector**, change its **Rect | Size** property to **(200, 70)**.
- Using the **Inspector**, scroll down to the section entitled **Custom Fonts** and click on the downward-facing arrow to the right of the label **Font**, as per the next figure.



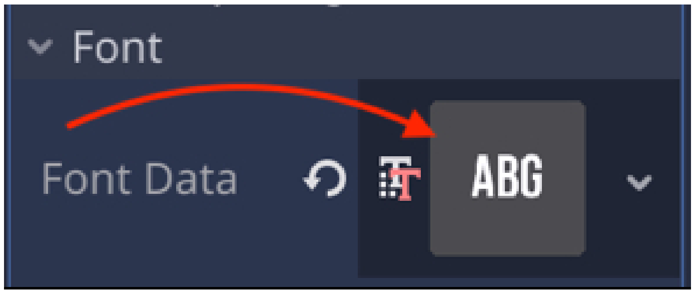
- From the contextual menu, select the option **New DynamicFont**.



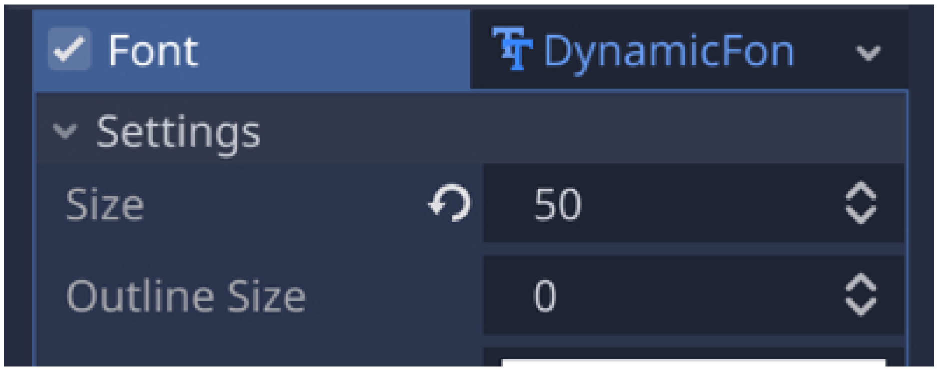
- Then click on the downwards-facing arrow to the right of the **DynamicFont** label and select **Edit** from the contextual menu.



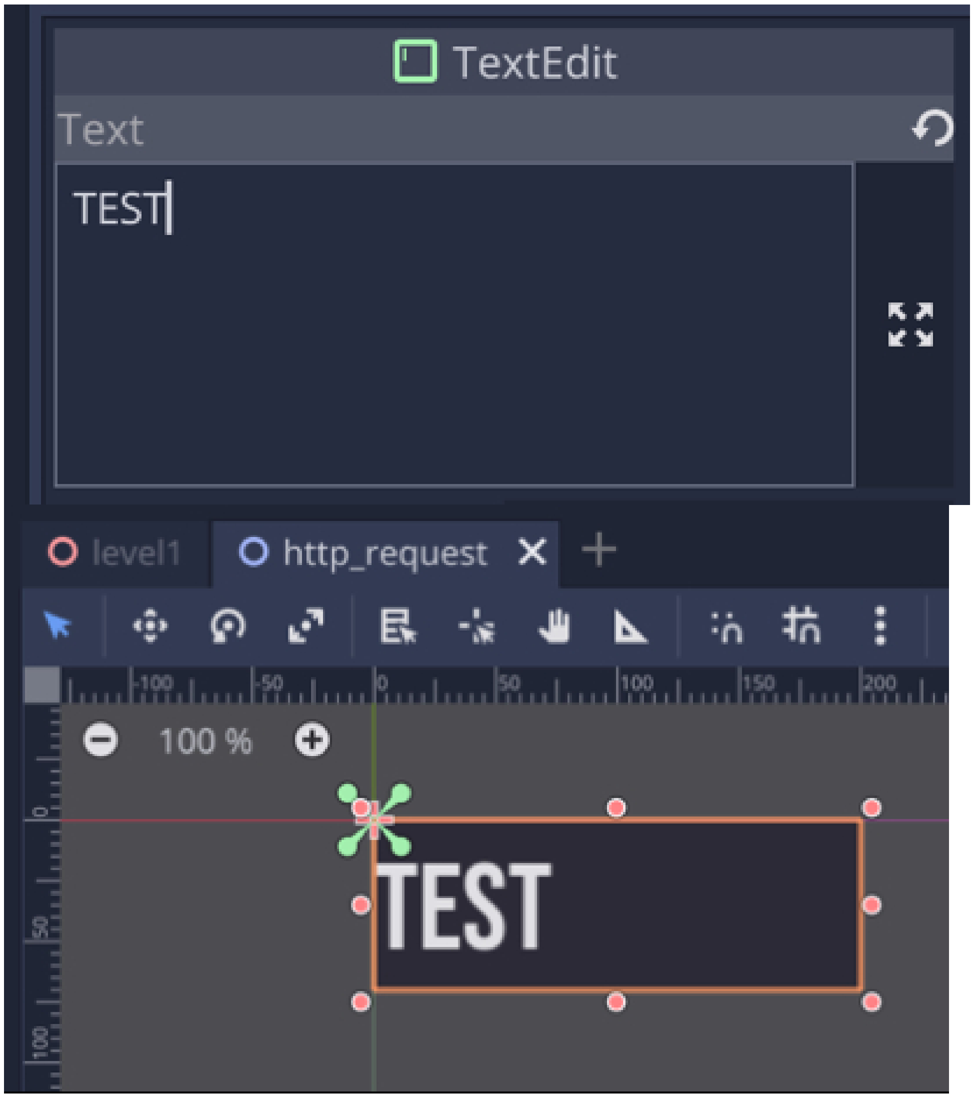
- In the new window, expand the section called **Font** and drag and drop the file called **BebasNeue-Regular.ttf** from the **File System** tab to the section called **Font Data**.



- You can now expand the section called **Settings**, and set the font size (e.g., **50**).

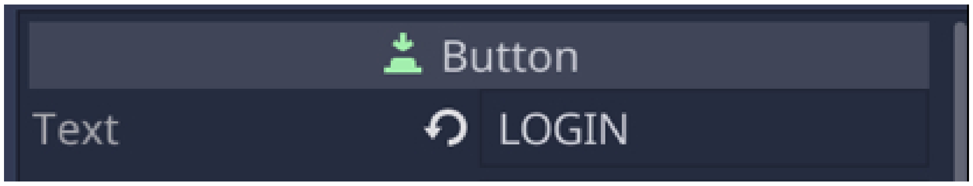


- Move the node to the center of the screen using the **Move** tool.
- You can type some text in the **Text** attribute of this node, to check how the text will look when the game is running.



Finally, we will create a button that will be used to validate the data entered in the text field; this is because the user will enter its name in the text field and then press a button so that its score is saved in the database accordingly.

- Please, add a node of type **Button** as a child of the node **Node2D**.
- Place this button just below the text field that you have just created, using the **Move** tool.
- Change the label of this button to "**LOGIN**" using the **Inspector**, by modifying the attribute called **Text** in the section called **Button**.



- Change the font for this button as we have done for the previous text field.

Now that the user interface has been defined, it is time to create a script that will process the entry from the player, and update the database accordingly, when the button is pressed.

- Please select the node **Button**, add a script to it, and rename the script **save_score.gd**.
- Add the following code at the start of the script (new code in bold).

```
extends Button
onready var http_request = get_node("../HTTPRequest")
onready var txt_field_name = get_node("../txt_field_name")
var player_name
var player_score
```

In the previous code:

- We create a new variable called **http_request** that refers to the node called **HTTPRequest**.

- We create a variable called **txt_field_name** that refers to the text field that we have created earlier.
- We then create two additional variables called **player_name** and **player_score** that will be used to save and/or update the player's name and score.

We will now add the code that processes the entry from the player:

- Please add the following code to the script (new code in bold):

func _ready():

connect("pressed",self,"save_score")

http_request.connect("request_completed",self, "post_request_completed")

In the previous code:

- We define the function **_ready**.
- We then connect the event **pressed** to the function **save_score**. This means that whenever the button is pressed, the function **save_score** is called.
- We also link the event **request_completed** related to the node **http_request** to the function **post_request_completed**.

Please add the following code (e.g., at the end of the script):

func save_score():

player_name = txt_field_name.text

player_score = 787878

print("Starting to Save Score")

http_request.request("http://localhost:8888/
up-dateScore_b.phname="+player_name+"&score="+ str(player_score))

In the previous code:

- We create the function **save_score**.
- We save the text entered in the text field to the variable **player_name**.
- We save the value **787878** in the variable **player_score**.
- We print the message "**Starting to Save Score**".
- We then launch an http request, opening a url that includes two parts.
- The first part is the address: **http://localhost:8888/updateScore_b.php**; it opens a php page hosted on the server.
- The second part is **?name="+player_name+"&score="+ str(player_score)**; this passes the value for the parameters **player_name** and **score** as we have done beforehand with PHP files. We use the keyword **str** to convert the value of the variable **player_score**.

Once this is done, we just need to define the function **post_request_completed**; so please add the following code to the script:

```
func post_request_completed():
print("DB updated")
```

In the previous code, we just print the message "**DB updated**" once the function **post_request** has been called.

You can now test your scene: please play the scene, enter the name **player1** and press the button; as you check the database, you should notice that the score stored for this player is now **787878** instead of **100**. You should also see the following message in the **Output** window.

```
Starting to Save Score
name: player1Found 1 record
```

Similarly, you can enter the name **NEWPLAYER1000** and press the button; as you check the database, you should notice that a new user called **NEWPLAYER1000**, along with its score, have been created. You should also see the following message in the **Output** window.

```
Starting to Save Score
name: NEWPLAYER1000Sorry not registered
```

LEVEL ROUNDUP

Well, this is it!

In this chapter, we have learned about communicating with a database, setting up a server, creating a database and tables, and finally reading from, writing to or updating a database through Godot. In the process we were also introduced to PHP and MYSQL commands, and we finally managed to create a system whereby the player's score is saved in this database. So yes, we have made some considerable progress, and we have by now looked at some simple ways to store and access information about the game and the players on a server.

Checklist

You can consider moving to the next chapter if you can do th

- Create a database.
- Create tables.
- Access a database through PHP and Godot.
- Understand the links between Godot, PHP, and a MYSQL database.

Quiz

It's now time to check your knowledge with a quiz. So please try to answer the following questions (or specify whether the statements are correct or incorrect). The solutions are included on the next page. Good luck!

1. The event **request_completed** is called whenever an http request was processed successfully.
2. To be able to access a database a user with proper privileges needs to be set up.
3. Crossdomain restrictions apply if you are trying to access a database on your local server.
4. Write the missing PHP code to select all records from the table players.

```
$query = "SELECT MISSING CODE
```

1. Find two errors in this PHP code.

```
$result= mysqli_queries($con, $query);
$n = mysqli_numrow($result);
```

1. The event **pressed** is called whenever a button has been pressed.
2. The keyword **IEnumerator** when added before the name of a function indicates that it is a co-routine.
3. Find one error in the following code, assuming that we want to resume the function once data was sent back from the PHP script.

```
string url = "http://localhost:8888/updateScore_b.php";
url += "?name=" + playerNname + "&score=" + score;
WWW www = new WWW(url);
return;
```

1. Data can be passed to a PHP script using its url.
2. The PHP function **$_GET** can be used to receive (or get) data from a MYSQL database.

Solutions to the Quiz

1. TRUE.
2. TRUE.
3. FALSE.
4. Write the missing PHP code to select all records from the table players.

```
$query = "SELECT * from players";
```

1. Find two errors in this PHP code.

```
$result= mysqli_queries($con, $query);
$n = mysqli_numrow($result);
```

1. FALSE.
2. TRUE.
3. Find one error in the following code, assuming that we want to resume the function once data was sent back from the PHP script.

```
string url = "http://localhost:8888/updateScore_b.php";
url += "?name=" + playerNname + "&score=" + score;
WWW www = new WWW(url);
yield return;
```

1. TRUE.
2. FALSE (it is used to gather information passed in the url).

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let's modify the scripts to add more interaction.

- Add a new column to the database that you have already created to save a number (i.e., int) that will represent the last level achieved by the player.
- In Godot, create a splash-screen, along with three other scenes.
- In the splash screen, ask the user for its name, and either load the first scene if s/he is not registered in the database, or load the corresponding last level achieved stored in the database for this user.

Chapter 3: Creating a Networked Multiplayer Game

In this section we will discover how to create a multiplayer game that will be played over the network.

After completing this chapter, you will be able to:

- Understand key and necessary concepts to be able to network your game.
- Understand how these concepts can be used in Godot for a multiplayer game.
- Understand the principle of client and server.
- Create a simple multiplayer game.

INTRODUCTION

So far, in the previous three books in the series, we have learned to create a First-Person Shooter (FPS) whereby the player could collect and use weapons against a Non-Player Character (NPC). While this format is quite challenging, and interesting, many of you would have already played multiplayer games where they can compete directly against remote friends. This is usually a lot of fun, and this is the reason why it would be interesting to create such a game. The main difference between the games that we have created to date and this type of game is that the game is played over a network. The players are in different locations (on the same or different sub-networks) and can play (and share) the same virtual environment.

So, the game is usually hosted on a server, clients (i.e., players) connect to this server, and can play the game. Such games also pose several challenges, in comparison to single-player games, because all the resources provided (and shared) in the game need to be synchronized across the network and between players so that all players obtain a seamless experience. In addition, network speed may also add to the challenges posed by the design of this type of games.

As we will progress through this chapter, we will get to understand the key similarities between single- and multiplayer games; and we will also discover network concepts that will help you to better understand how the game needs to be structured. By creating a network game, you will understand key concepts that you will be able to use in other environments and games; and while you may not become a network specialist, it will always be an advantage to understand and develop skills related to networked games, as many companies usually add a multiplayer option to their games; so it may be a good asset if you would like to pursue a career in the gaming industry or just create games casually.

The high-level scripting api

When you are creating networked games in Godot, you are creating a system whereby several users will communicate and share resources over a network. This could be over the Internet or even over a Local Area Network (LAN). For example, the players could be located in the same office, in the same house, or in the same university. When developing networked applications, programmers often use what are called **low-levels API** to be able to network their game. An API is, put simply, a set of tools that provide you access to resources or to special features. This being said, in Godot, to make this process easier, you have the possibility to use what is called a **High-Level API**. Using this API, you can access networking features without having to know too much about what is going on under the hood. Using this API, you will, for example, be able to create a networked multiplayer game, make it possible for players to send messages to each-other, send different types of data between the players, synchronize the view for each player, and to create clients as well as servers.

Adding networking capabilities to your game, may also open your (and any of the computers joining this game) to be compromised, as it may happen when creating a networked application; so in any case, when you release your multiplayer game, you will need to consider and address any security concerns.

The way we will create our game is quite similar to what we have done in the last three books in the series, and it will involve the following steps:

- Creating a simple 3D world with game objects.
- Adding networked features so that it possible to "share" and "synchronize" the game between players.
- Managing connections between the players and the server.
- Managing the game from the point of view of the server or the client, and ensuring that the data shared between players is always up-to-date.

Throughout this chapter, we will be using some of the built-in components provided by Godot's High Level multiplayer API to implement our game. This API will be referred to as **HLMAPI** from now on in this book.

About clients and servers

When creating a networked game, you will often hear about the concepts of clients and servers; these concepts are, of course, present in most network applications. So what are they?

In computer science terms, a server provides services that may be requested by clients. So usually clients would connect to the server to avail of a service. In terms of web development, a server will deliver a page or execute a programme and provide the output to a client (e.g., through a client browser). In a networked game, and in the particular case of the game that we will be creating in this chapter, the server will host the entire game, and the clients will connect to the server to access the game; the server will also handle some of the updates in the game (e.g., when new objects are created), so that all players play the same version of the game; in other words, we want all clients to see the same version of game. So the clients and the server, once connected, will continuously exchange information. Because of the amount of information involved, as we will see later, we will also have to think about the frequency at which this information is exchanged, and there will be a balance to be found between updating the server regularly and keeping the game running smoothly with no or little lag or delays.

So in Godot, using the HLMAPI, you can see the game (or virtual world) from the perspectives of both the clients and the server.

Generally, a player will usually be seen as a client; it will connect to the server to update the server on its own features (e.g., position, health, or ammos), and the server in turn will send each player information about other players (e.g., position, health, ammos) or objects that are part of the game (e.g., whether an ammo pack is still present in the environment). Again here the idea is that, because the game is shared between players who are in different locations, all the data in the game is synchronized (i.e., the same) across players. So for example, if player1 picks-up an ammo in the game, the game should be updated so that player2 (and other players) can't see this ammo anymore. The same applies to health, if player1 is hit by a projectile, its health should decrease, and the next time this happens, its health should be known by at least the server, so that the projectile either inflicts further damage or eliminates the player.

As for web development, a server can be a dedicated computer that acts solely as a server, or it can also be a computer that both acts a server and a client (as we have done on the second chapter).

Connection between clients and servers

To be able to exchange data, a client and a server need to be connected. So, typically a server would have an address and be contacted at this address through a port (i.e., similar to a channel) by a client. It's a bit like having to contact a friend. You can contact your friends by dialing their mobile number, by dialing their landline number, or by emailing them. In all cases, your friends will be waiting for your call or email by using (or listening to) a specific channel (e.g., mobile phone, landline, or email).

So a server would usually be contactable using an IP address. An IP address is a succession of numbers, separated by dots that help to identify (and connect to) a computer on a network (over the Internet or a local network). For example, the IP address 17.142.160.59 is linked to a server that is used to store the web pages for the website apple.com, or the IP address 75.126.59.147 is linked to a server that is used to store the web pages for the website Godot3d.com. If you enter these addresses in a browser, both sites should come-up. In fact, you could try to guess the IP addresses for a particular site by opening a command prompt in Windows or Terminal on a mac computer and typing ping followed by the name of their website; the system will then return the IP address linked to the server where the pages are hosted. The addresses mentioned above are public, in the sense that the servers linked to these addresses can be accessed directly because there is no firewall between the client and the server. A firewall would typically restrict access to the devices on its network (e.g., clients or servers) and hide the actual IP address of these also

However, a server can also use a private address; this is usually the case when the server is installed behind a firewall. By default, in Godot, and we will see this later, if the server is hosted on your own computer, the address of the server will be **127.0.0.1**, which, in computer terms, usually defines your own computer. So by typing **127.0.0.1** in a browser, and if you have an active server listening to the web port (i.e., http), a page should be displayed. In Godot, you can specify the network port for your game.

When you access a computer on your own Local Area Network (LAN), the address of the computer acting as a server may be determined by the firewall. So for example, if you have a router/firewall at home, the router is the point of contact between the outside world (i.e., the Internet) and your local network. So the router will have a public address; however, all the computers on the network (including yours) will have a private address that will not be seen by the outside world (i.e., a private address), but that will be allocated by the router. For example, if you are on a network at home, your IP address may look like **168.0.0.2** for example (**168.0.0.1** would typically be allocated to the router). This is usually referred as NAT (Network Address Translation), whereby your router/firewall will allocate private addresses to computers on the same network.

So generally, the simplest case is when the server has a public address; in this case, a client may access this server, provided that the client uses the correct address and port, and that the port is actually open on the server (i.e., that the server is listening for a client's request through this port.) In case we deal with a server with a private address, more complex methods, including NAT punch-through may be employed.

DNS and IP addresses

As we have seen in the previous section, a public server can be contacted using a public IP address, and to make things more simple, it is possible to use an actual name (using alphanumeric characters) to express this IP address. To take the example provided earlier, instead of having to remember the IP address 17.142.160.59, we can, instead type apple.com in a browser window, and this will directly open the Apple site. So there is a correspondence between the IP address and the actual corresponding url, and this is done using DNS which stands for Domain Name System. In short, using DNS, a table that operates the translation between an IP address and the corresponding domain name is kept on a server so that a site (or server) can be contacted either using its IP address or the corresponding domain name. So as you will be using Godot and creating a server for your game, this server will be contactable using either its IP address or a domain name.

Managing clients and servers using HLMAPI

When no dedicated servers have been set up to host your game, you can use a client (e.g., your own computer) to also act as a server.

In Godot when your networked multiplayer game has been created, it can run as a host or a remote client. A host will include both a local server and a local client; the remote client usually connects to and communicate with the host remotely (via the network), whereas the local client and server connect directly.

In this case, your computer becomes a host. This host is therefore both a client and a server. In other words, it will be able to host the game, but it will also have a local client that will be able to access the local server. So, on the host computer, you would have, in this case, a server and a local client. We could compare it to owning your own shop or a family shop; your shop would be comparable to the host (as you will be selling and providing goods and services); one of your family members can act as a cashier while another member of the family could be a client (i.e., buy goods from your shop). So in this case, both members of the family are hosts (they live in the same building and are part of the family who owns the shop), and they can be either customers (at times) or sellers.

In addition to the host, remote clients can connect to the host to request services. So, in this case you have two types of clients: local clients (from the same computer) and remote clients (from a different computer). So, typically, a player hosting the game will be a host; this player will have a server to which other players can connect to and access the game, as well as a client for the player (who hosts the game) to also play the game.

This feature is interesting as it will make it possible for us to simulate a networked multiplayer game using only one computer. As you will see later, we will create a host, with a local server and local client on your computer.

CREATING A SINGLE-PLAYER SCENE

So, I hope that the network concepts are clear at this stage, if not, not to worry, these will be explained in context in the next sections. For now, we will get started on creating our networked scene and using some of the built-in networking components available in Godot.

The plan for the next sections is to do the following:

- Create a simple scene where the player is initially represented by a capsule and is controlled using the arrow keys: the player will be able to collect pills and increase its score.
- Add networking capabilities so that this scene can be played by (and shared amongst) several players over the same network.
- Make sure that every time a new player joins the scene, s/he is represented by a capsule in the virtual world and that s/he can see the other players accordingly.

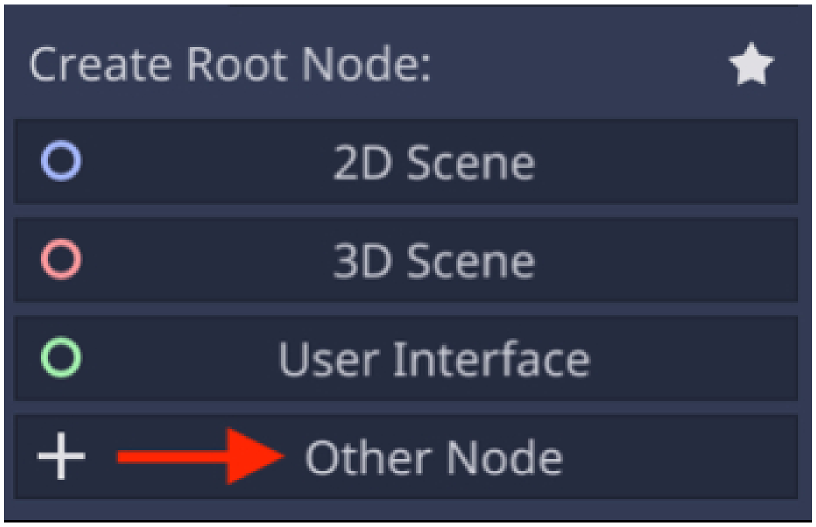
This will be done step-by-step and all the networking aspects will be covered in detail, so that you get a solid grasp of implementing a simple networked game; once you understand how to create such a game, you will be able to transfer these skills to a game of your choice.

Creating a the player character

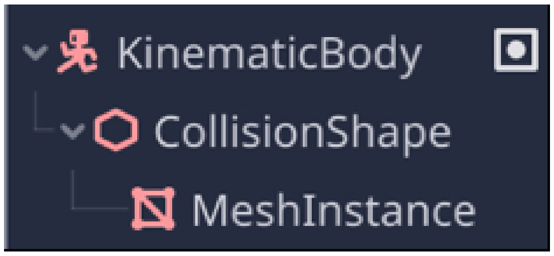
To start with our game, we will create a simple environment with a ground symbolized by a scaled box and a capsule. We will also add movement to the capsule.

First, let's create the player's capsule.

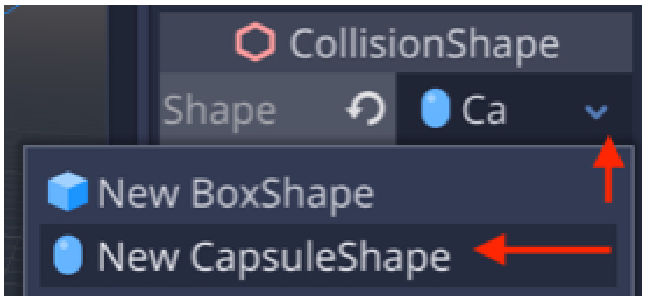
- Please create a new scene (**Scene | New Scene**).
- In the new window, please select the option "**Other Node**".



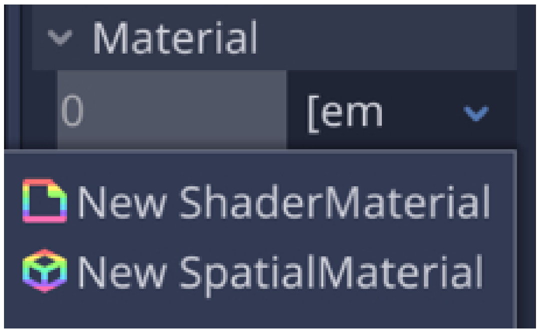
- In the new window, select the node type "**Kinematic Body**" and click on "**Create**".
- This will create a scene with a default node called **KinematicBody**.
- Add a child node of type **Collision Shape** to the node **KinematicBody**.
- Add a child node of type **MeshInstance** to the node **CollisionShape**.



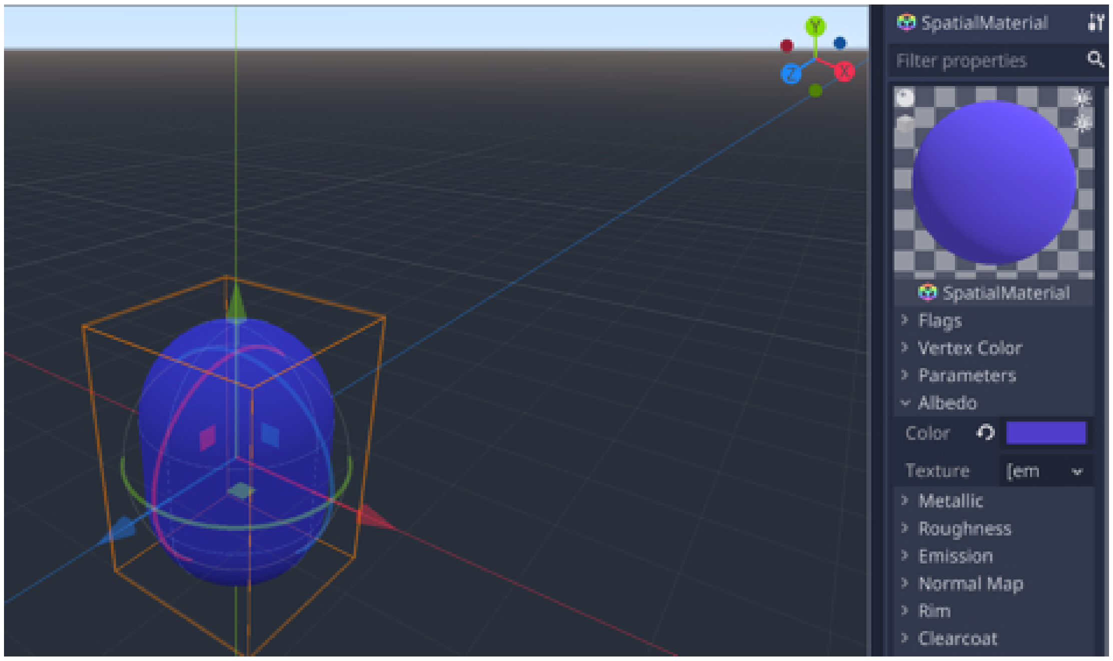
- Select the node **CollisionShape**, and using the **Inspector**, scroll to the section called **CollisionShape**, click on the downward-facing arrow to the right of the label **Shape**, and select the option **New CapsuleShape** as illustrated in the next figure.



- In the section called **Translation**, change the **Rotation** property to **(90, 0, 0)**.
- Select the node **MeshInstance**, and using the **Inspector**, scroll to the section called **MeshInstance**, click on the downward arrow to the right of the label **Mesh**, and select the option **New CapsuleMesh**.
- In the section called **Material**, click on the downward arrow to the right of the number **0**, and select the option **New SpatialMaterial** as illustrated in the next figure.



- Once this is done, click on the white disc in the same section, scroll down to the **Albedo** section, and select a color of your choice (e.g., **blue**).



Now that we have created the player character, let's create a script that will be used to move the player character around the level:

- Please save the current scene as **net_player.tscn** (**Scene | Save Scene As**).
- Please select the node **KinematicBody**.
- Add a script to this node by right-clicking on that node and selecting the option "**Attach Script**" from the contextual menu, name it as **move_net_player.gd**.
- Once the script is open, please add the following code to it (new code in bold)

```
extends KinematicBody
var speed = 3
var score = 0
var direction = Vector3()
```

In the previous code:

- We create three variables: **speed**, **score** and **direction**.
- The variable called **speed** will be used to set the speed of the player character.
- The variable called **score** will be used to save the score of the player character as it collects pills.
- The variable **direction** is a vector that will store the current direction of the player character.

Once these variables have been declared, we will start to handle the key pressed by the player to move the player character.

- Please add the following function to the script:

```
func _physics_process(delta):
if (Input.is_action_pressed("move_left")):
direction -= transform.basis.x
elif (Input.is_action_pressed("move_right")):
direction += transform.basis.x
elif (Input.is_action_pressed("move_up")):
direction -= transform.basis.z
elif (Input.is_action_pressed("move_down")):
direction += transform.basis.z
else: direction = Vector3(0,0,0)
direction = direction.normalized()
move_and_slide(direction*speed,Vector3.UP)
```

In the previous code:

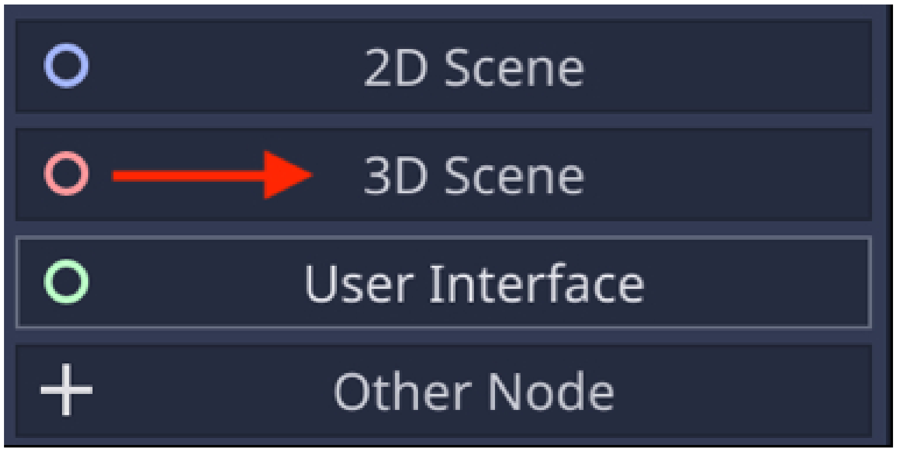
- We declare the function **_physics_process** which will be called every frame.
- In this function we test whether the player has pressed the keys mapped as "**move_left**", "**move_right**", "**move_up**", and "**move_down**".

- This will correspond to the left, right, up and down arrow keys and this mapping will be completed through the project's settings in the next sections.
- When the player presses one of these keys, the vector called **direction** will be changed to left (negative direction along the x axis), right (positive direction along the x axis), up (negative direction along the z axis), and down (positive direction along the x axis).
- If no keys are pressed, then the direction is **(0, 0, 0)**.
- The vector called **direction** is normalized, meaning that speed will remain constant even if the player keeps the key(s) pressed.
- Finally, we move the player character using the built-in function **move_and_slide**; when using this function, we specify that the moving direction is determined by the vector called **direction** and the variable **speed**; we also specify what is the **up** direction in this game, and in our case it is the **Vector3.UP**.

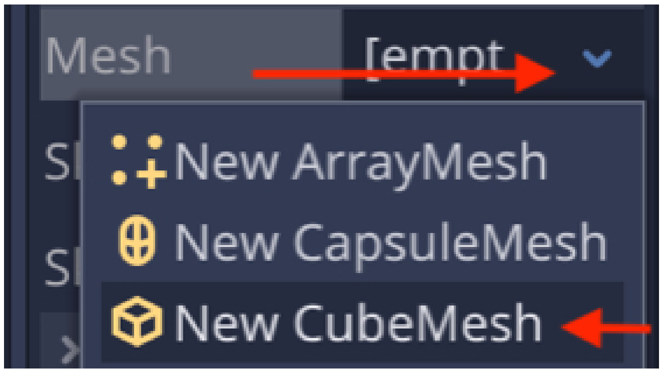
Creating the game scene

Now that we have created our player, we will create a simple scene for the game, and also test whether the player character can be moved around this environment.

- Please save the current scene (**Scene | Save**).
- Create a new scene (**Scene | New Scene**).
- In the new window, please select the option "**3D Scene**".



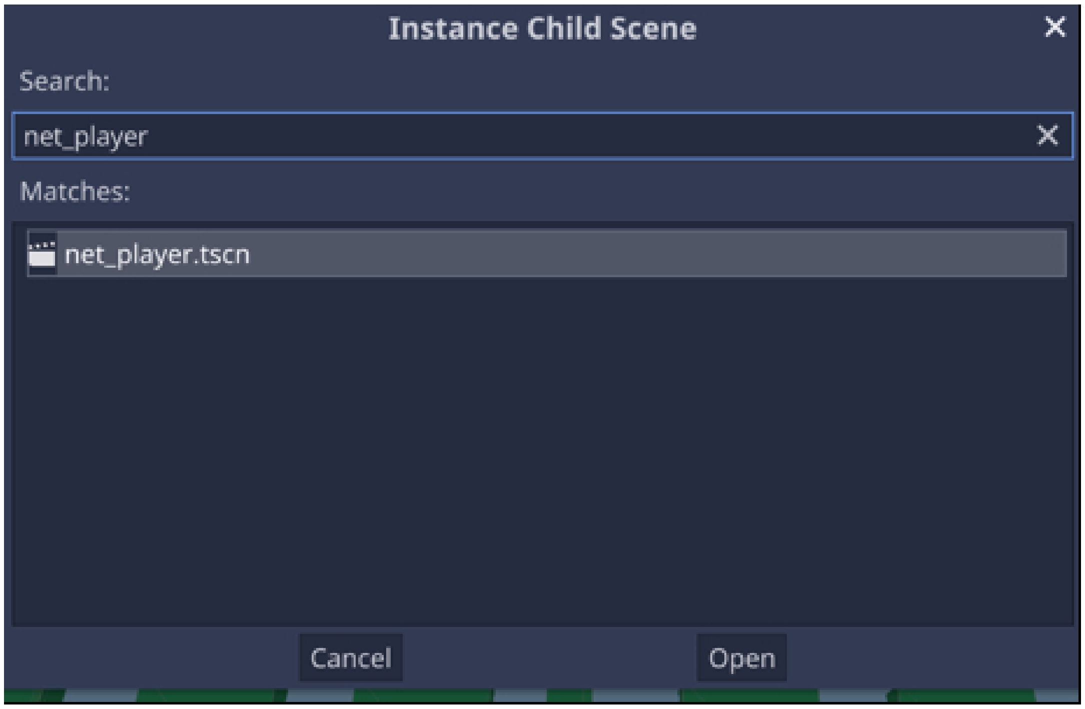
- This will create a new scene with a default node called **Spatial**.
- Please rename this node "**main**".
- Save this scene as **networked_scene.tscn** (**Scene | Save As**).
- Add a new node of type **Spatial** to the node **main** and rename this new node **level1**.
- Add a **MeshInstance** node as a child of the node **level1**.
- Select this new node, and, using the **Inspector**, scroll down to the section **MeshInstance**, click on the downward-facing arrow to the right of the label called **Mesh** and select the option **New CubeMesh**.



- Click on the white cube to the left of the label **Mesh**, and change the **size** of this mesh to **(50, 2, 50)**.

Now that we have created a basic ground for the game, we can add the player and test if it is possible to move it:

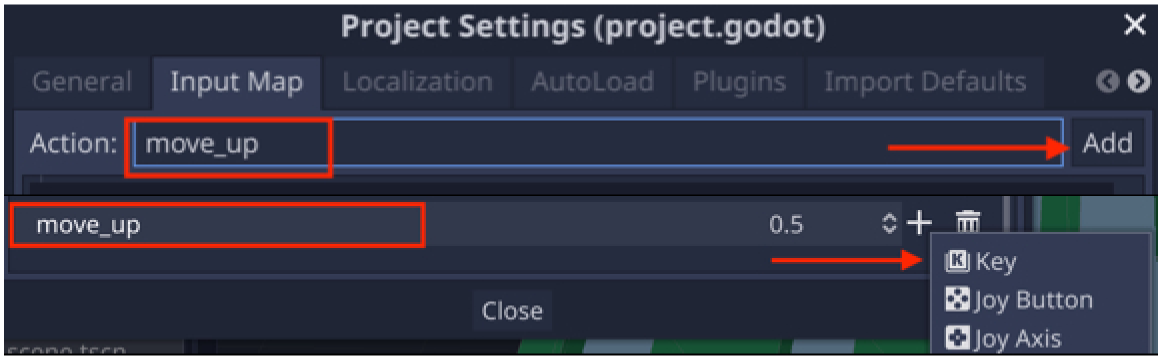
- Please right-click on the node **level1** and select the option **Instance Child Scene**.
- In the new window, select the scene called **net_player.tscn** and press the button labeled **Open**.



- This will create a new child node called **KinematicBody** to the node **level1**.
- Please set the position of this new node to **(-11, 0.3, -10)** or any other location as long as it is slightly above the ground.
- Please add a new node of type **Camera** as a child of the node **level1**; set the **position** of this new node to **(0, 36, 1)** and its **rotation** to **(-90, 0, 0)**.

Finally, now that the environment and the player have been set up, we just need to map the keys that will be used to move the player character, so please do the following:

- Open the Project's Input preferences (**Project | Project Settings | Input Map**).
- In the new window, add new actions called "**move_up**", "**move_down**", "**move_left**" and "**move_right**" if they don't already exist, and map them to the arrow keys "**up**", "**down**", "**left**", and "**right**" respectively, as per the next figures.



Once this is done, you can now play the scene **networked_scene** and check that you can move the player character by using the arrow on the keyboard.

Completing the design of the level

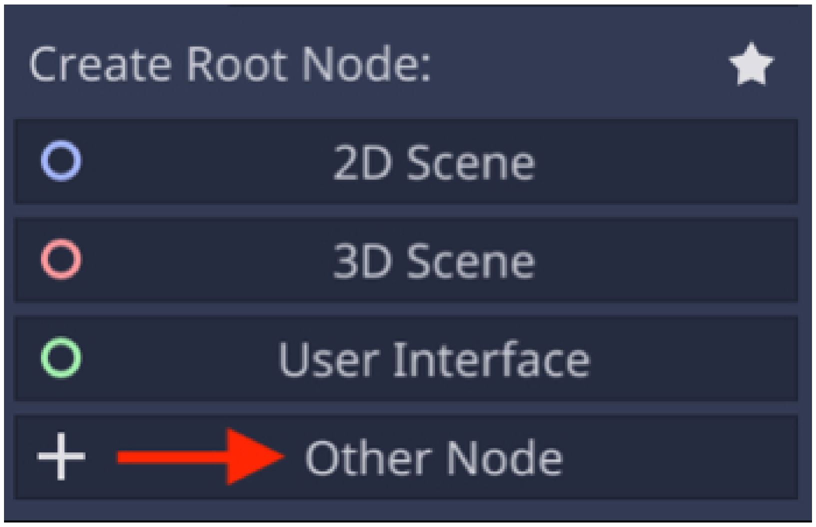
Now that we can move the character player around the environment, we will complete the design of the level by adding the following features:

- External and internal walls.
- Pills that the player can collect to increase the score accordingly.
- Speed pills that when collected by the player will increase the speed of the player character.

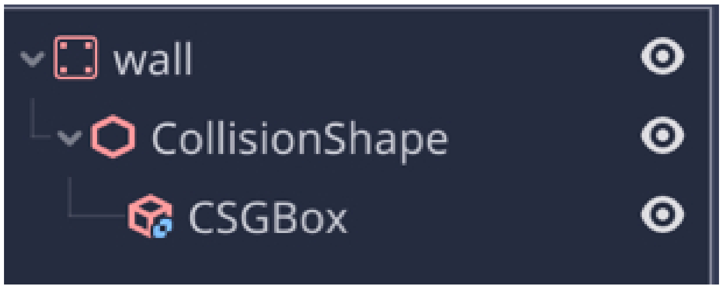
First, let's create a template for the walls that will be used in the level:

- Please create a new scene (**Scene | New Scene**).

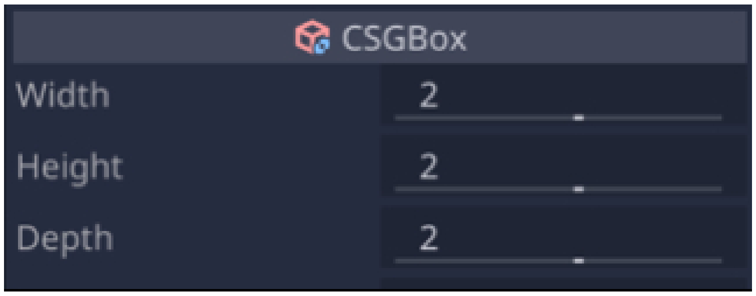
- In the new window, please select the option “**Other Node**”.



- In the new window, select the node type “**Static Body**” and click on “**Create**”.
- This will create a scene with a default node called **StaticBody**
- Please rename this node **wall**.
- Add a child node of type **Collision Shape** to the node **wall**.
- Add a child node of type **CSGBox** to the node **CollisionShape**.



- Select the node **CollisionShape**, and using the **Inspector**, scroll to the section called **CollisionShape**, click on the downward arrow to the right of the label **Shape**, and select the option **New BoxShape**.
- Select the node **CSGBox**, and using the **Inspector**, scroll to the section called **CSGBox**, and set the **width, height and depth** attributes to **2**.

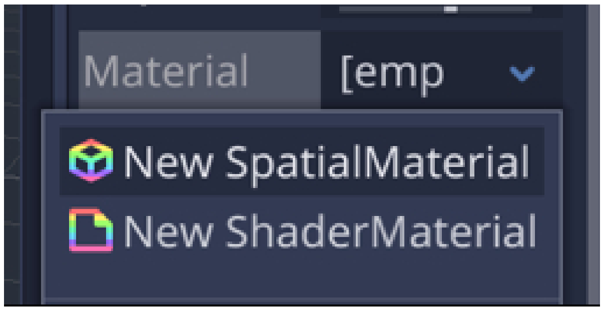

CSGBox

Width 2

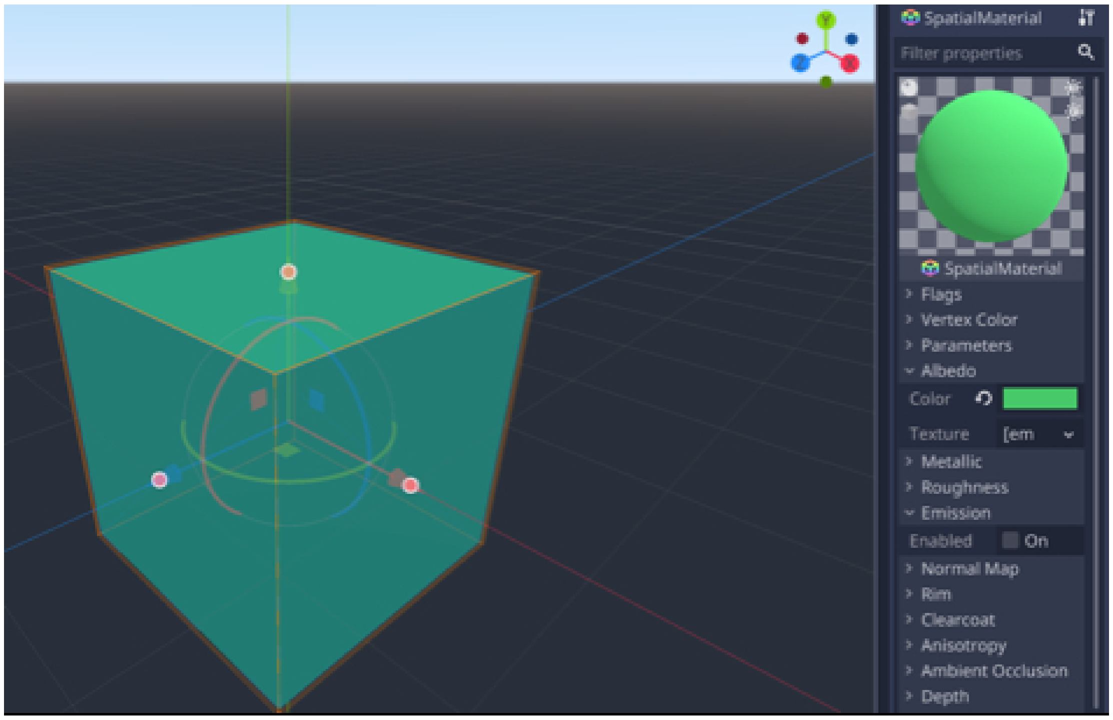
Height 2

Depth 2

- In the section called **Material**, click on the downward arrow to the right of the label **Material**, and select the option **New SpatialMaterial** as illustrated in the next figure.



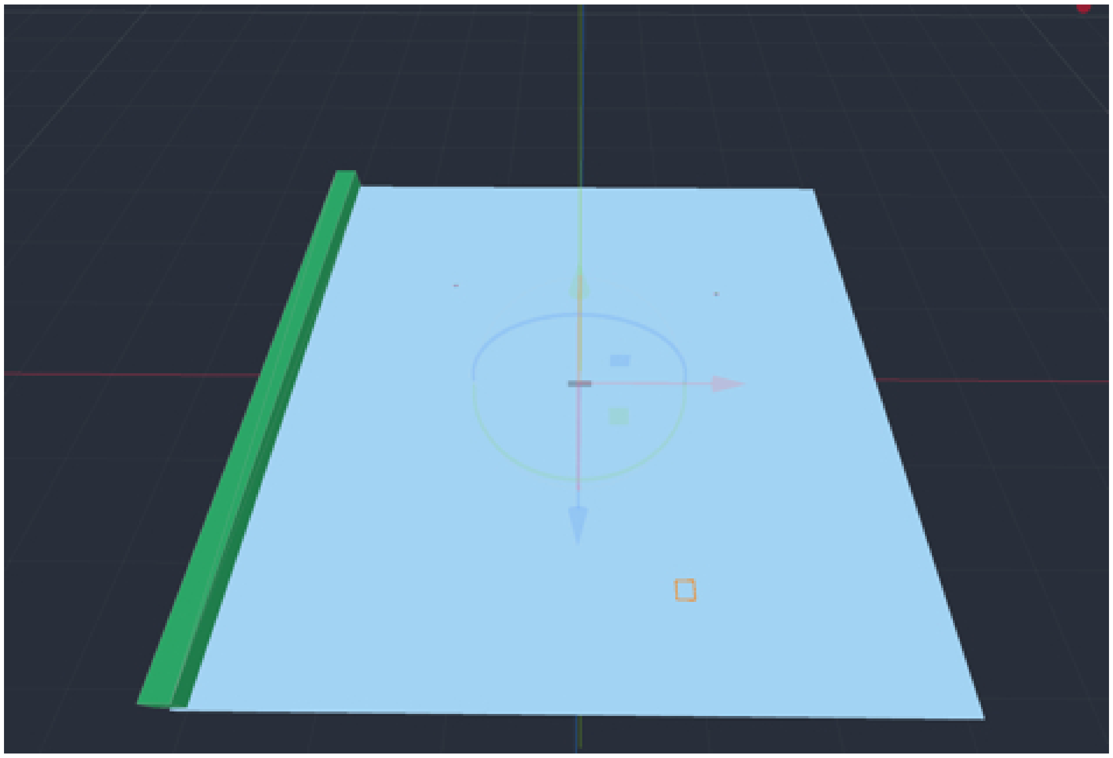
- Once this is done, click on the white sphere in the same section, scroll down to the **Albedo** section, and select a color of your choice (e.g., green).



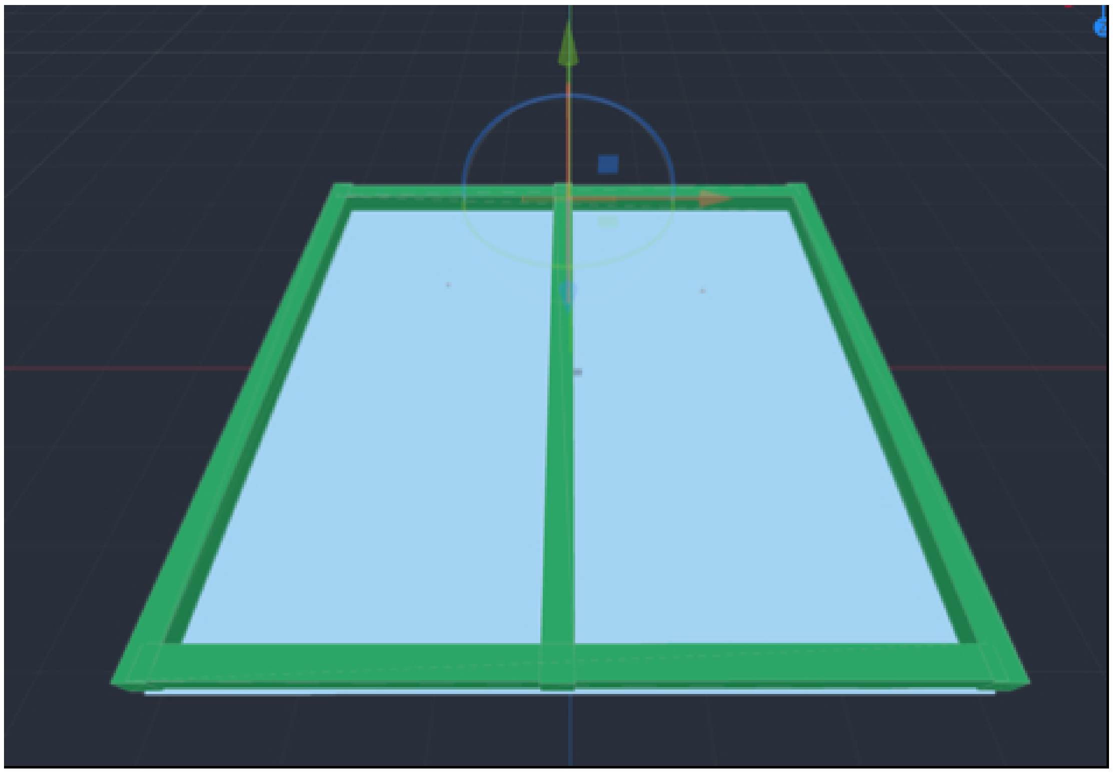
- Please save the scene as **net_wall.tscn** (**Scene | Save Scene As**).

Once this is done, please do the following:

- Open (or switch to) the scene **networked_scene.tscn** so that we can add walls to it.
- Using the **Scene Tree** window, right-click on the node **level1**, select the option **Instance Child Scene** from the contextual menu, select the scene **net_wall.tscn**, and click on "**Create**"; that will add a child called **wall** to the node called **level1**.
- Select this node (i.e., **wall**)
- Using the **Inspector**, set its position to **(-25, 1, 0)** and its scale to **(1, 1, 25)**.

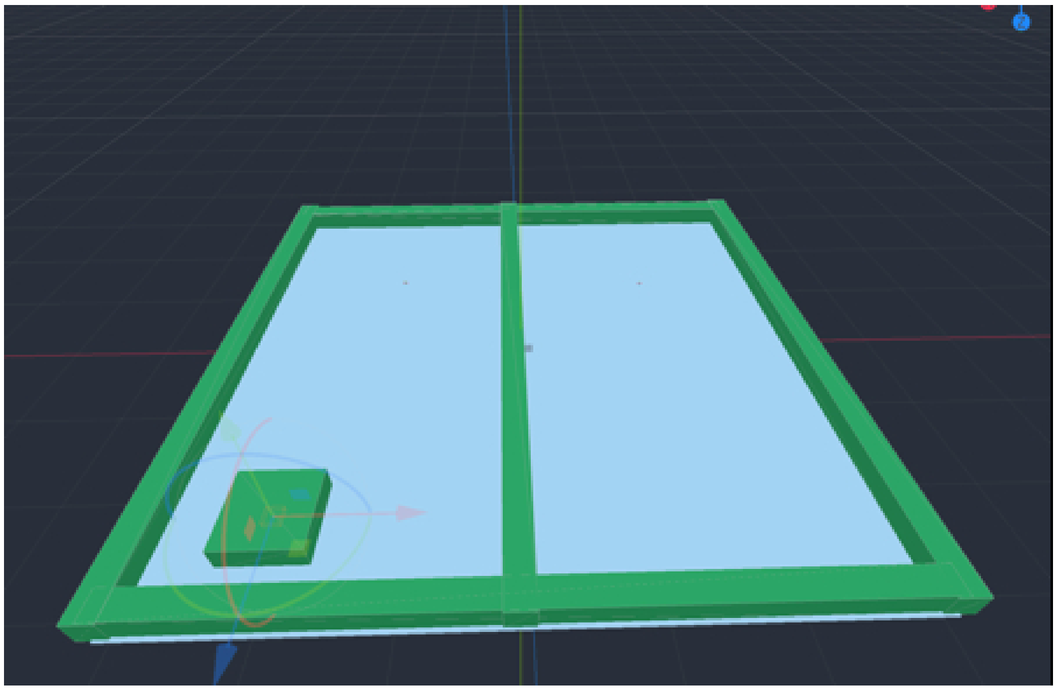


- Duplicate and rotate this node four times to obtain the following layout.

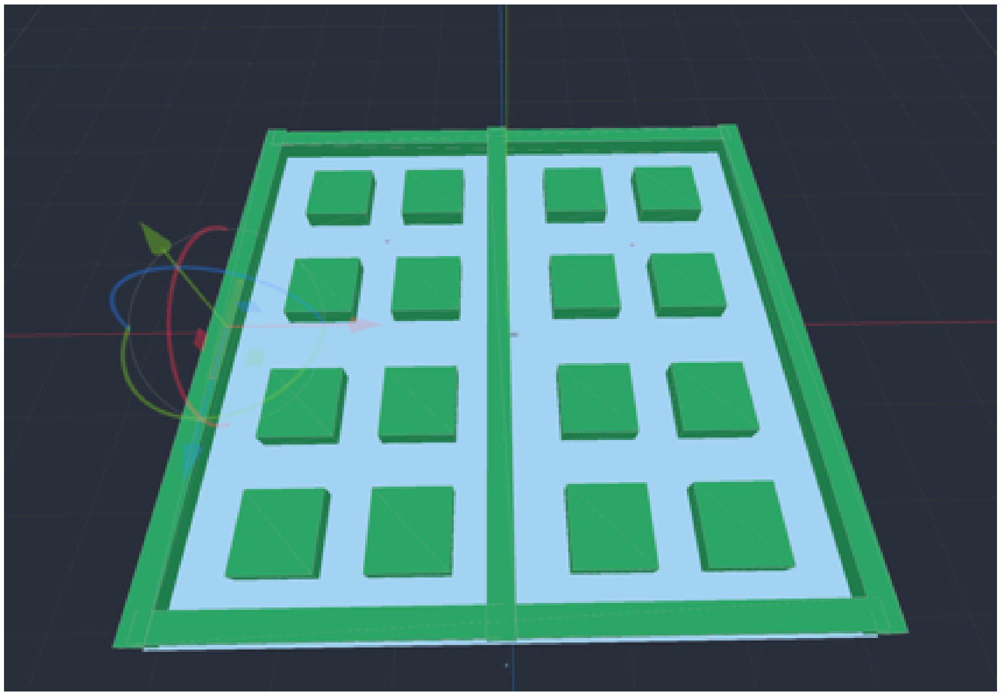


Next, we will create the internal walls:

- Please duplicate the node called **wall**, and rename the duplicate **square_wall**.
- Select the node **square_wall**, change its scale to **(3, 1, 3)** and add it near the bottom left corner, for example, at the position **(-1, 1, 18)**, as per the next figure.



- Duplicate this node several times to obtain the layout described in the next figure, ensuring that the **y** coordinate of the duplicate is **1**.

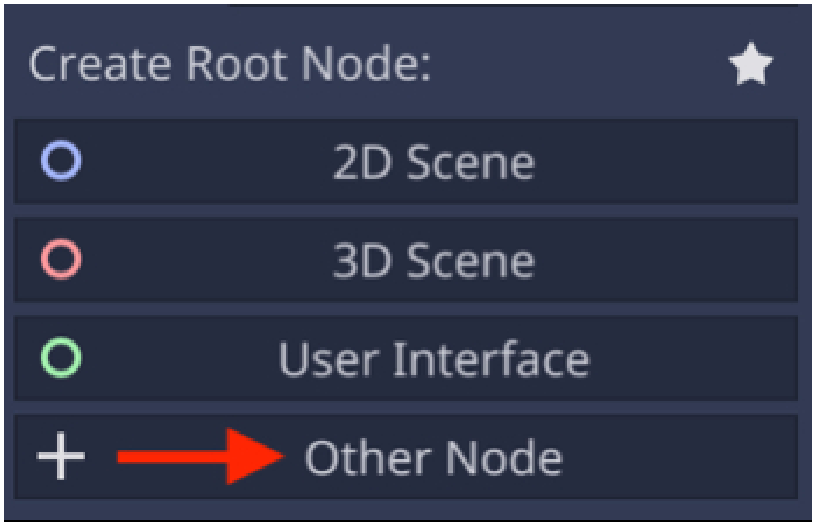


Once this is done, you can test the scene and ensure that the player can navigate through it and not go through the walls.

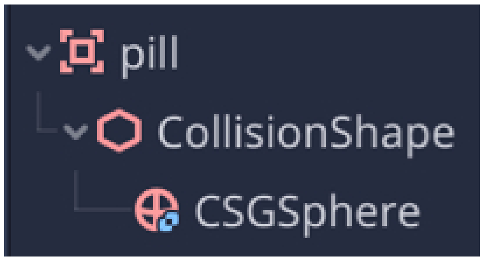
Adding pills to be collected

At this stage, you can move the player character, and we will add a new feature whereby the player will be able to collect pills and increase its score; in case a speed pill is collected, the player character's speed will be increased.

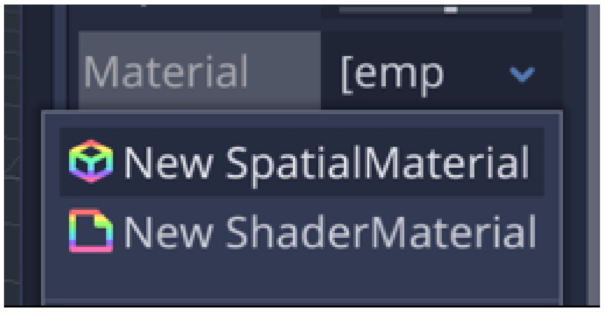
- Please save the current scene (**Scene | Save Scene**).
- Please create a new scene (**Scene | New Scene**).
- In the new window, please select the option "**Other Node**".



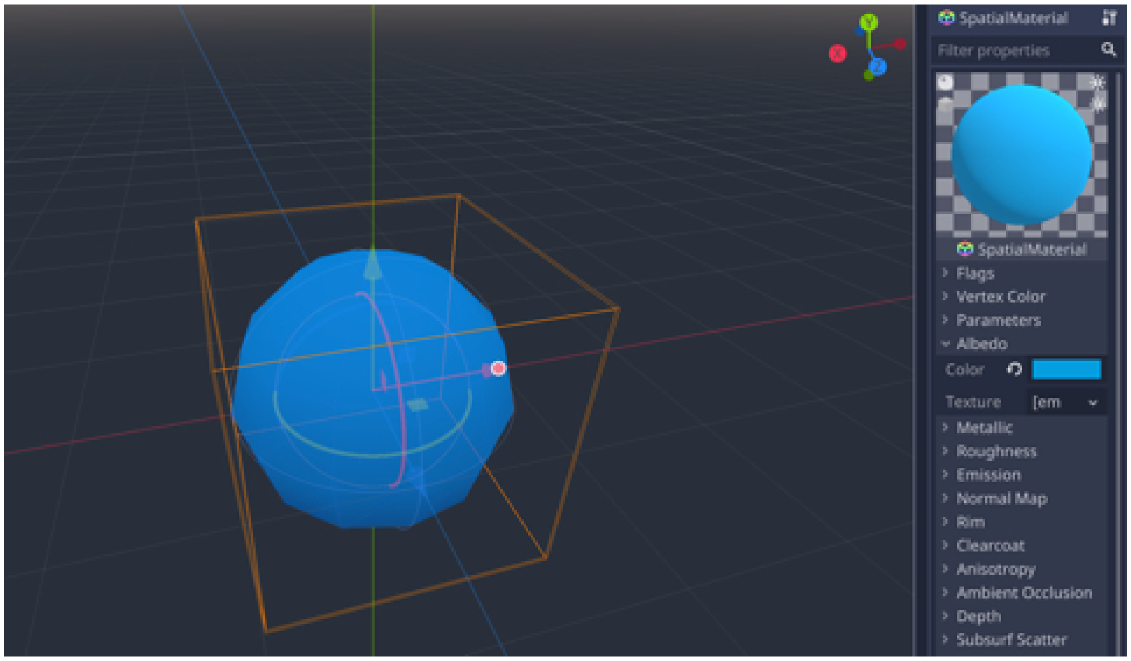
- In the new window, select the node type "**Area**" and click on "**Create**".
- This will create a scene with a default node called **Area**.
- Please rename this node **pill**.
- Add a child node of type **Collision Shape** to the node **pill**.
- Add a child node of type **CSGSphere** to the node **CollisionShape**.



- Select the node **CollisionShape**, and using the **Inspector**, scroll to the section called **CollisionShape**, click on the downward arrow to the right of the label **Shape**, and select the option **New SphereShape**.
- This will create a new node called **CSGSphere**.
- Select the node **CSGSphere**, and using the **Inspector**, scroll to the section called **CSGSphere** and check that the **radius** is set to **1**.
- In the section called **Material**, click on the downward arrow to the right of the label **Material**, and select the option **New SpatialMaterial** as illustrated in the next figure.



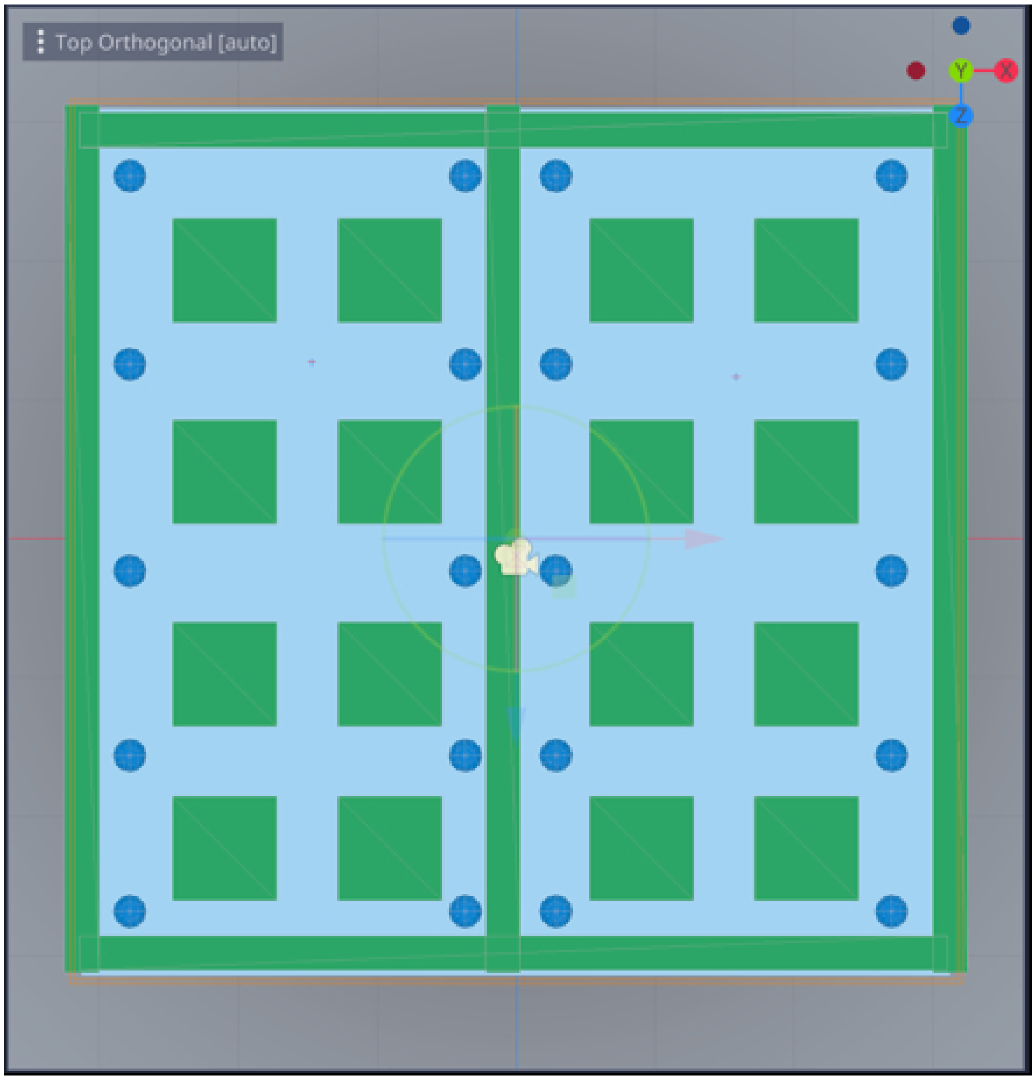
- Once this is done, click on the white sphere in the same section, scroll down to the **Albedo** section, and select a color of your choice (e.g., **blue**).



- Save your scene as **pill.tscn** (**Scene | Save Scene As**).

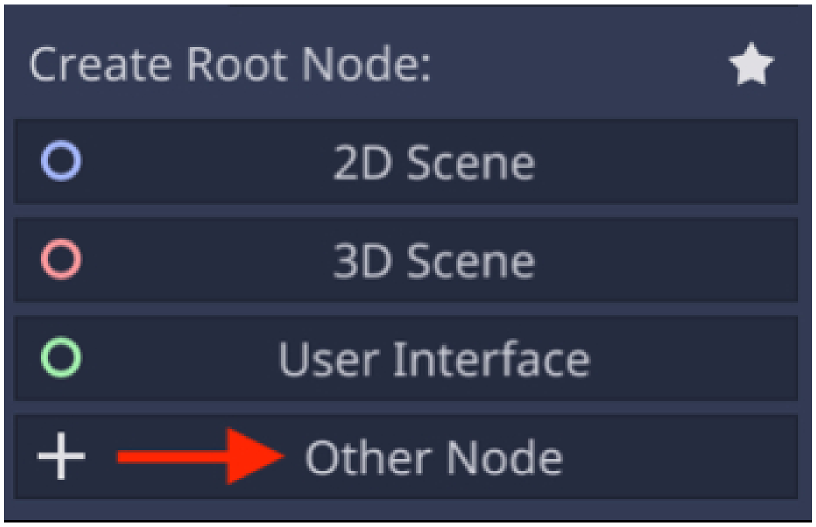
Now that we have created the scene **pill.tscn**, we can use this template to add pills around our level.

- Please open (or switch to) the scene **networked_scene**.
- Right-click on the node called **level1**.
- From the contextual menu select the option **Instance Child Scene**.
- In the new window, select the scene **pill.tscn** and then click on the button **Open**.
- This will add a node called **pill** as a child of the node **level1**.
- When this is done, please select this new node and check that its **y** coordinate is **1** so that it is just above the ground.
- Once you have checked this information, please duplicate the node **pill** several time to obtain a layout similar to the one illustrated in the next figure, with **20** pills in total.

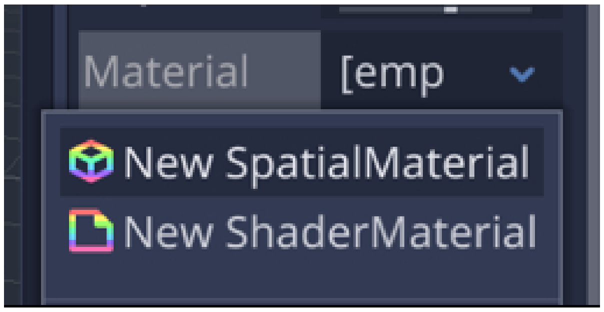


Once this is done, it is time to add other types of pills: speed pills; these pills, when collected by the player, will increase its speed. For this purpose, we will proceed as we have for the scene called pill:

- Please save the current scene (**Scene | Save Scene**).
- Please create a new scene (**Scene | New Scene**).
- In the new window, please select the option "**Other Node**".



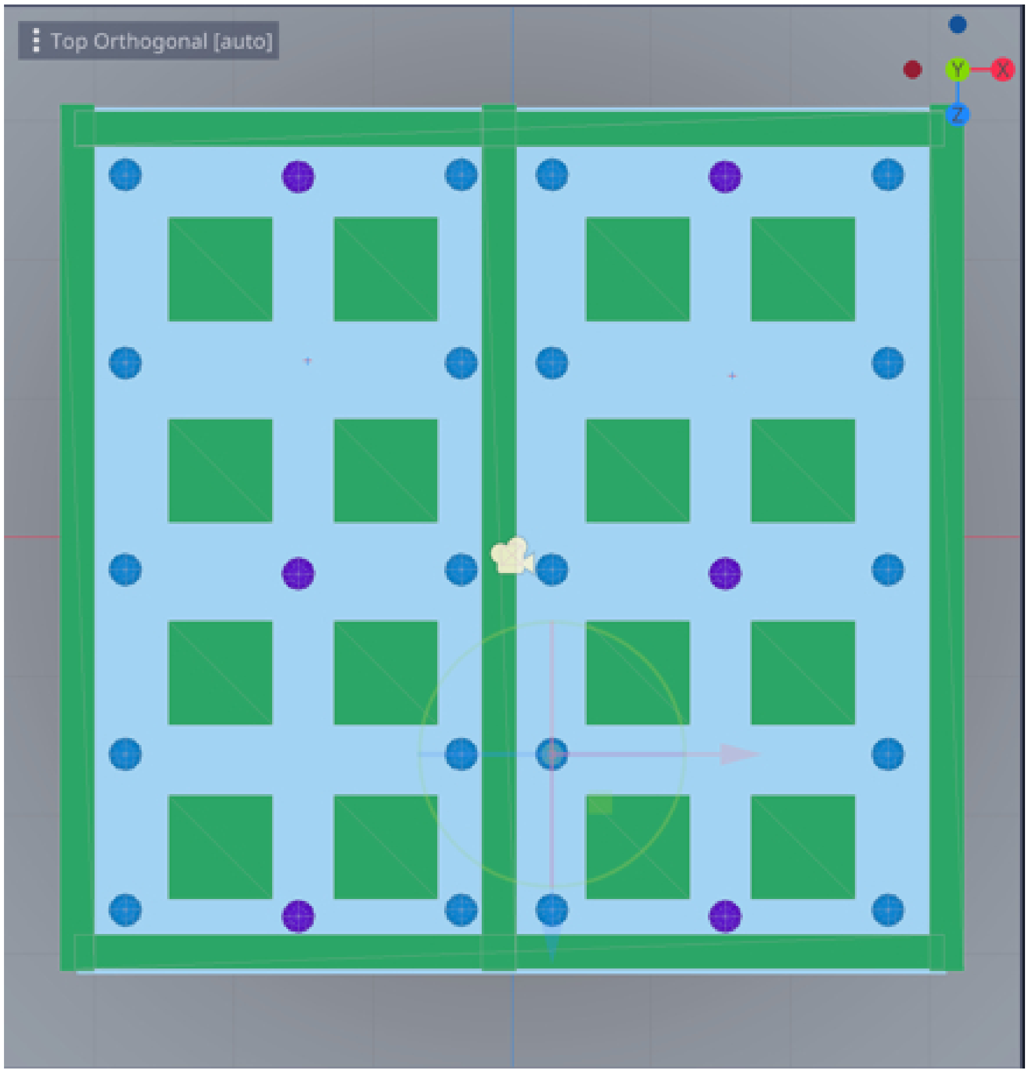
- In the new window, select the node type "**Area**" and click on "**Create**".
- This will create a scene with a default node called **Area**.
- Please rename this node **speed_pill**.
- Add a child node of type **Collision Shape** to the node **pill**.
- Add a child node of type **CSGSphere** to the node **CollisionShape**.
- Select the node **CollisionShape**, and using the **Inspector**, scroll to the section called **CollisionShape**, click on the downward arrow to the right of the label **Shape**, and select the option **New SphereShape**.
- Select the node **CSGSphere**, and using the **Inspector**, scroll to the section called **CSGSphere and check that the radius** is set to **1**.
- In the section called **Material**, click on the downward arrow to the right of the label **Material**, and select the option **New SpatialMaterial** as illustrated in the next figure.



- Once this is done, click on the white sphere in the same section, scroll down to the **Albedo** section, and select a color of your choice that is different to the one you have already chosen for the regular pills (e.g., purple).
- Please save your scene as **speed_pill.tscn**.

Now that we have created the scene **speed_pill**, we can use this template to add speed pills around our level.

- Please open (or switch to) the scene **networked_scene**.
- Right-click on the node called **level1**.
- From the contextual menu select the option **Instance Child Scene**.
- In the new window, select the scene **speed_pill.tscn** and then click on the button **Open**.
- This will add a node called **pill** as a child of the node **level1**.
- When this is done, please select this new node and check that its **y** coordinate is **1** so that it is just above the ground.
- Once you have checked this information, please duplicate the node **speed_pill** several time to obtain a layout similar to the one illustrated in the next figure, with **six speed pills** in total.

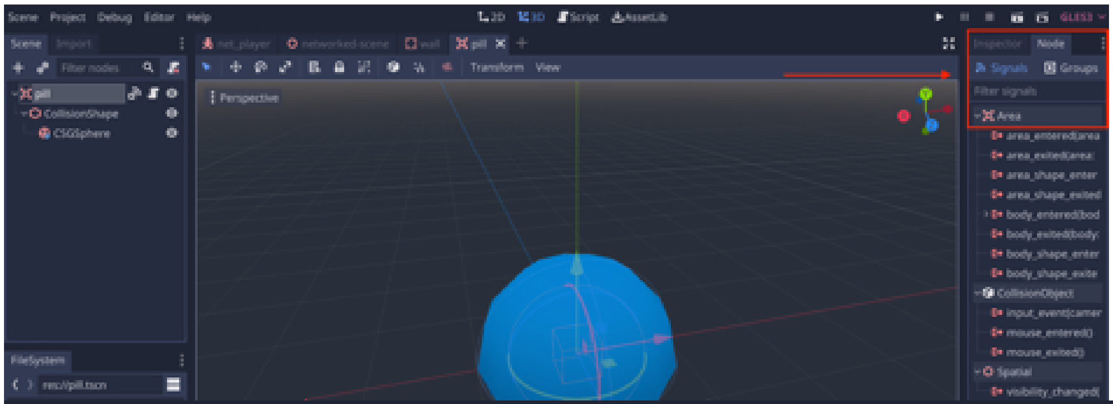


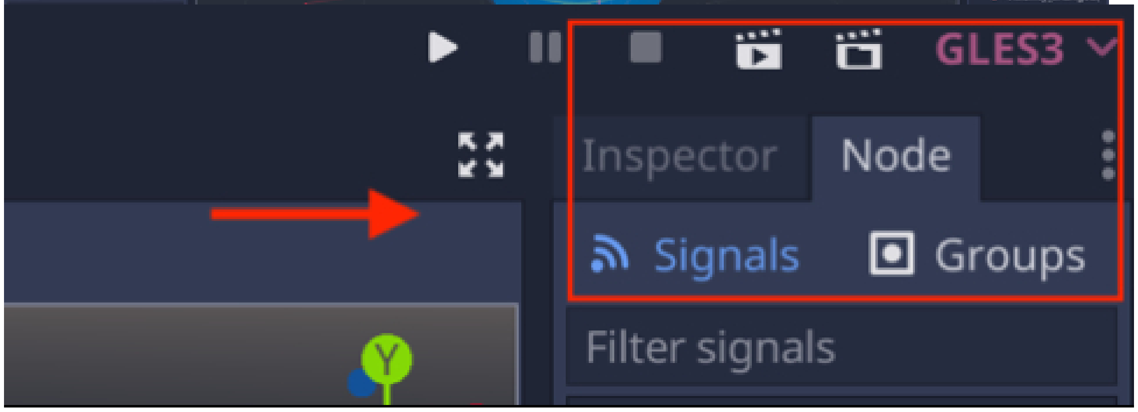
Managing the collection of pill s and the score

At this stage, we have managed to create a player character that can navigate in the virtual environment, along with objects that need to be collected; so in this section, we will add the necessary code for the player character to collect the pills, and increase the score and the player character's speed.

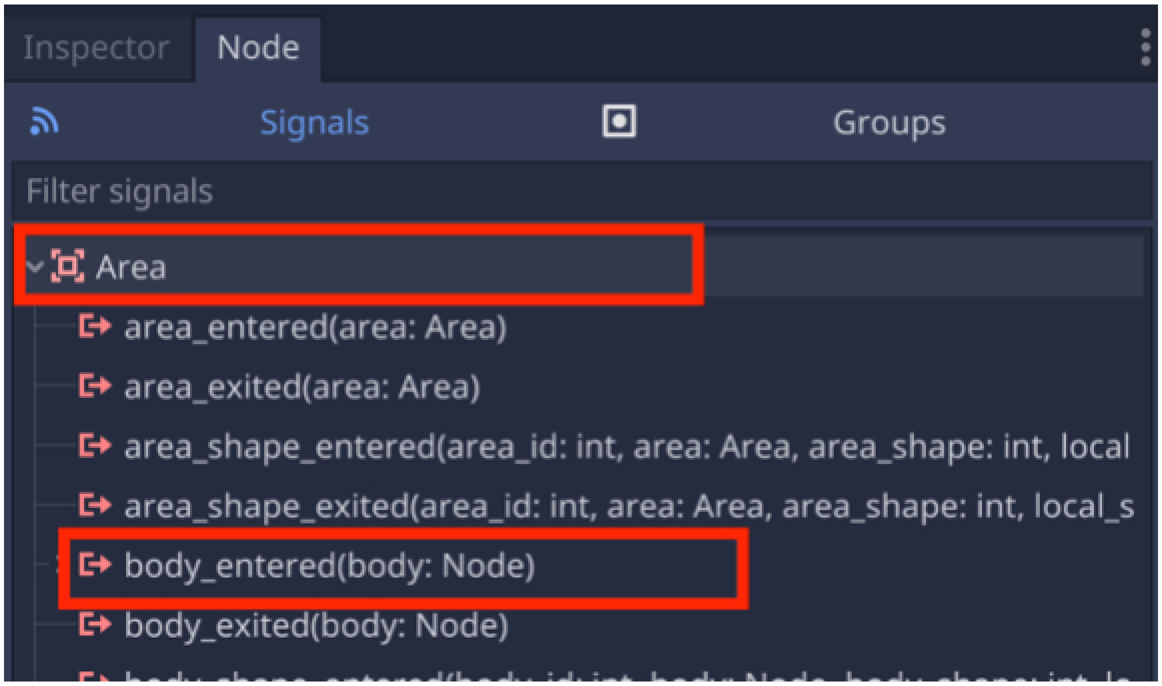
First, let's work on the collection of the regular pills:

- Please save the current scene (**Scene | Save Scene**).
- Open (or switch to) the scene **pill.tscn**.
- Please select the node called **pill**.
- Add a script to this node and save this script as **pill.gd**: right-click on the node **pill**, select the option **Attach Script**, type "**res://pill.gd**" in the new window, and then click on the button labeled "**Create**".
- Select the tab called **Node | Signal** located in the top right corner of Godot (if it doesn't appear, click on a different node, and then click again on the node **pill**), as described in the next figures.



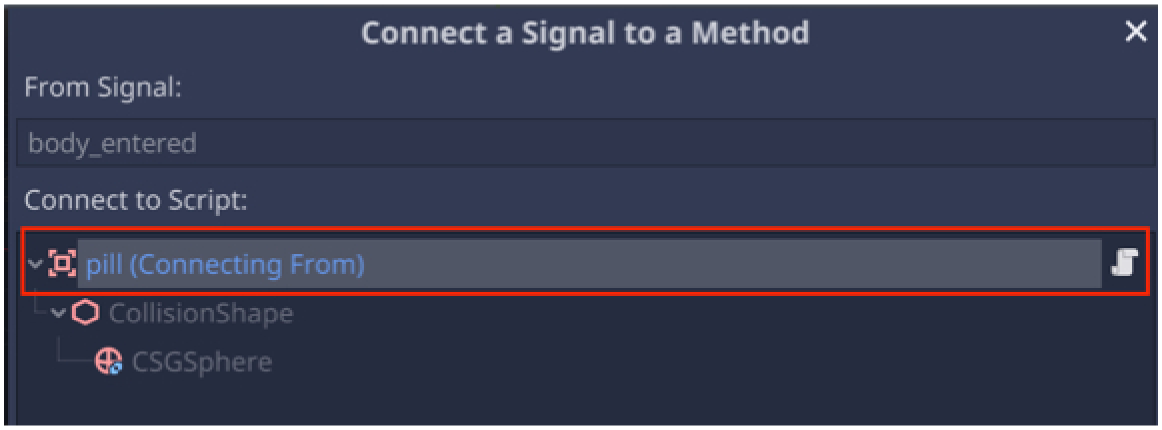


- In that tab, expand the section called "**Area**" (if it is not already expanded), and then double-click on the section called "**body_entered (body:Node)**".



area_entered(area: Area)
area_exited(area: Area)
area_shape_entered(area_id: int, area: Area, area_shape: int, local
area_shape_exited(area_id: int, area: Area, area_shape: int, local_s
body_entered(body: Node)
body_exited(body: Node)

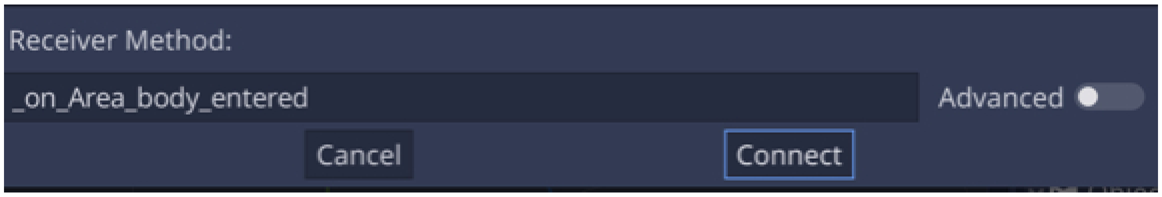
- In the new window, select the node **pill**.



From Signal: body_entered

Connect to Script: pill (Connecting From)

- Leave the default text in the field "**Receiver Method**" and press on "**Connect**".



Receiver Method: _on_Area_body_entered

- That should open the script **pill.gd** and show that a new function named **_on_pill_body_entered** has been created.
- This function will be called every time a node of type **KinematicBody**, including the player, will enter the area defined by its collision shape.
- Please modify this function as follows (new code in bold):

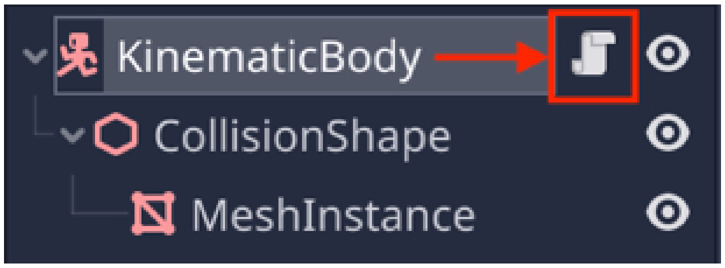
```
func _on_pill_body_entered(body):
if (body.is_in_group("player")):
body.update_score(5)
queue_free()
```

In the previous code:

- We check whether the object entering this area is a node that belongs to the group called **player**.
- If this is the case, then we call the function **update_score** (that we have yet to define) passing **5** as a parameter.
- We then destroy the pill.

Now that we have defined the script that will handle the collision, we will need to create the function **update_score** so that the score can be updated accordingly.

- Please save the current script (**CTRL + S** or **CMD + S**).
- Open (or switch to) the scene **net_player.tscn**.
- Click on the script logo to the right of the node **KinematicBody**, so that the script **move_net_player.gd**, that is linked to this node, opens.



- Once the script is open, please add the following function to it:

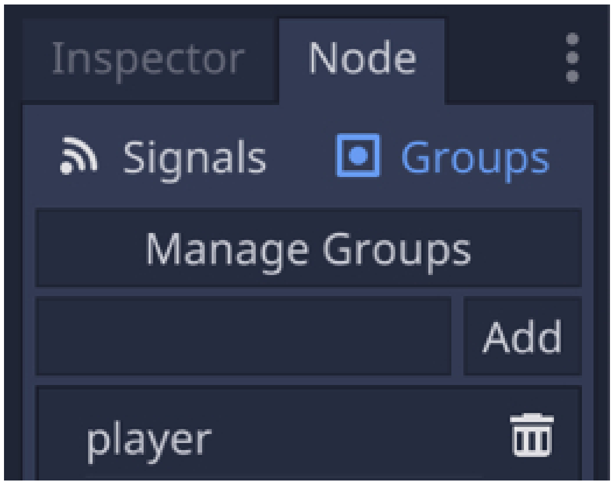
```
func update_score(new_increment):
score += new_increment
print("Score: "+str(score))
```

In the previous code:

- We create a function called **update_score** that takes one parameter referred to as **new_increment** within this function.
- We add the value of the variable **new_increment** to the variable **score**.
- We then print the value of the variable **score** in the Output window.

Before we can test the scene, we just need to make sure that the node for the player character belongs to the group called **player**, so that the collision between the pills and the player can be detected.

- Please open (or switch to) the scene **net_player**.
- Select the node **KinematicBody**.
- Select the tab **Node | Groups** located in the top-right corner of Godot (if the **Group** tab doesn't appear, please click on a different node, and then click on the node **KinematicBody** again).
- Add the group **player** to this node (i.e., type the word **player** in the text field below the label **Manage Groups**, and then click **Add**).

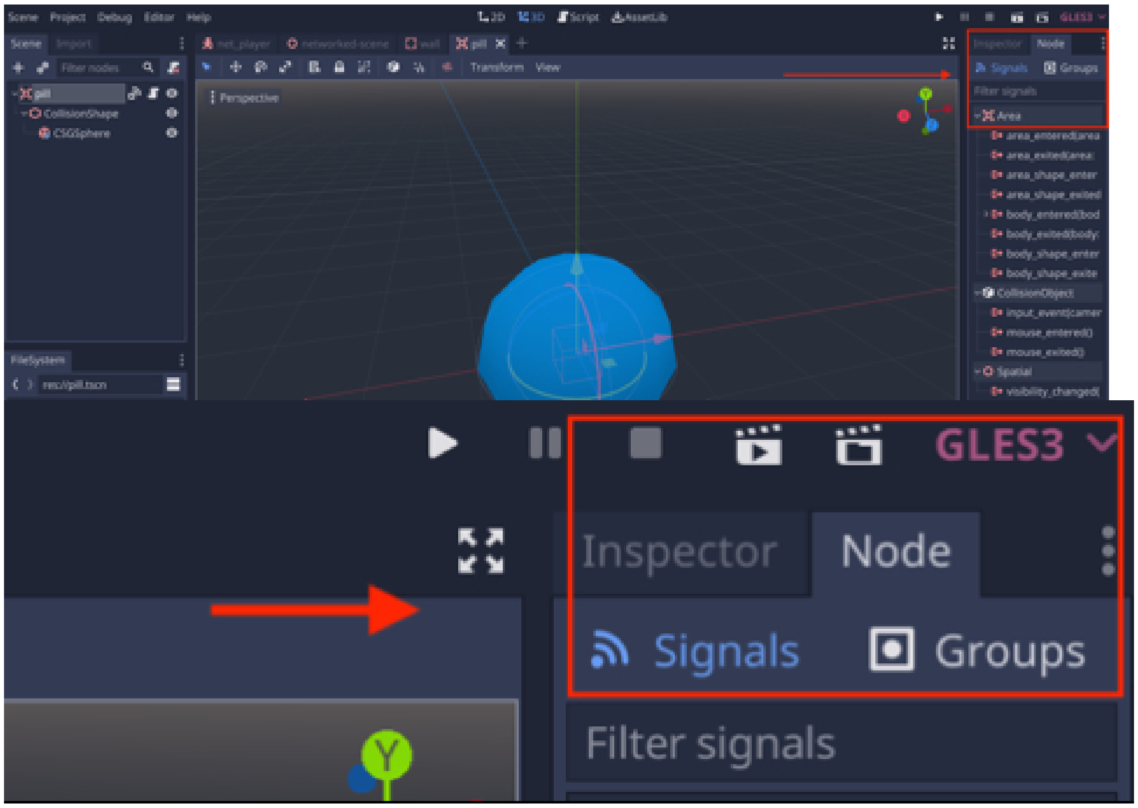


Once this is done, you can open and run the scene **networked_scene** and check that after collecting a pill that the score is increased and displayed in the **Output** window.

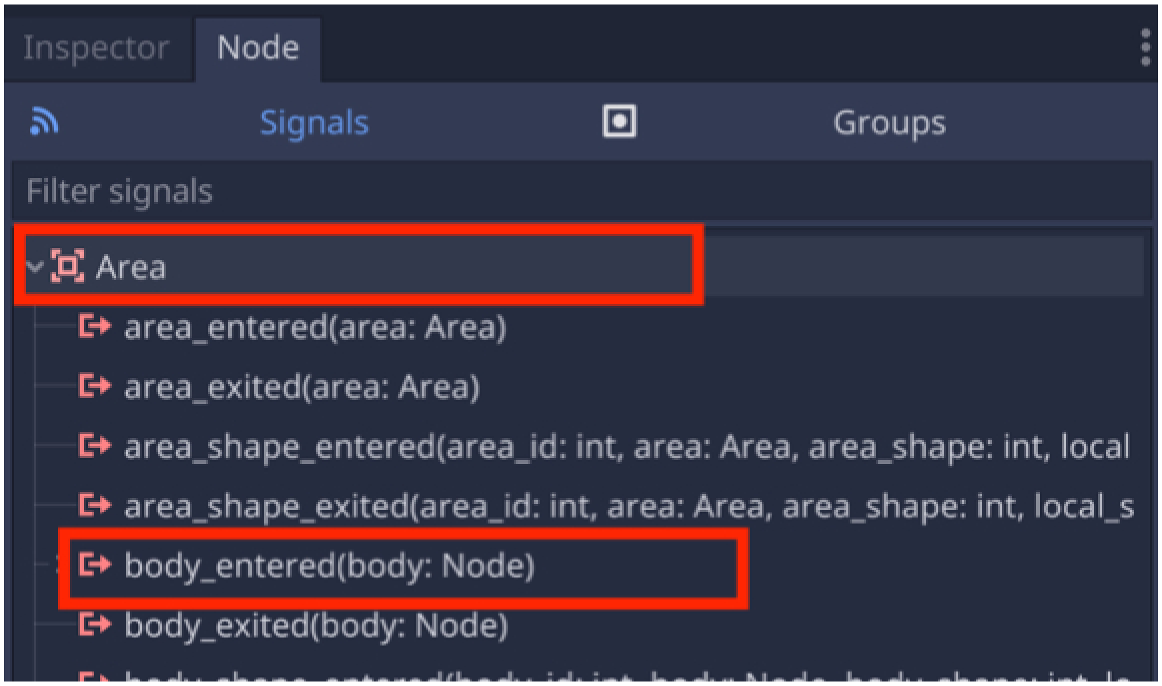
Managing the collection of speed pill s and the score

In this section, we will write the necessary code to collect the speed pills and to increase the speed of the character player accordingly.

- Please open the scene **speed_pill.tscn**.
- Select the node called **speed_pill**.
- Add a script to this node and save this script as **speed_pill.gd**: right-click on the node **pill**, select the option **Attach Script**, type "**res://speed_pill.gd**" in the new window, and then click on **Create**.
- Select the tab called **Node | Signals** located in the top right corner of Godot, as described in the next figure.

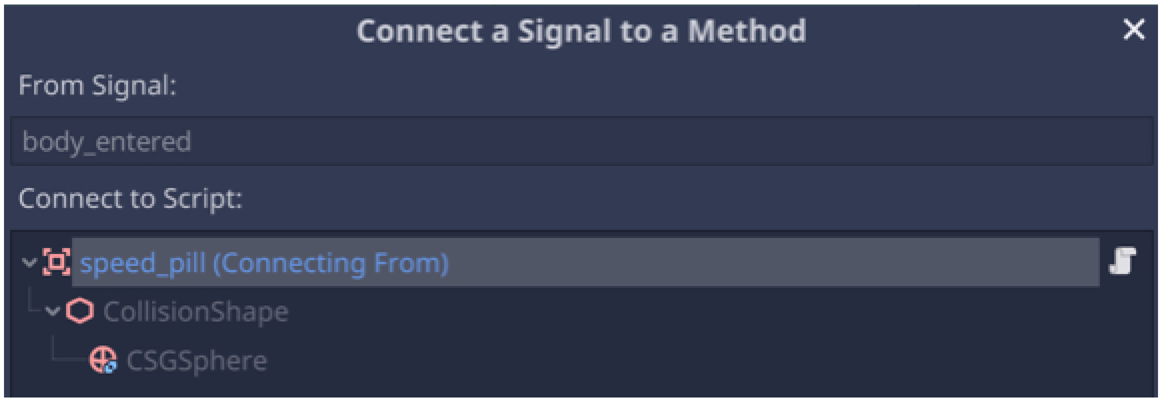


- In that tab, expend the section called “**Area**”, and then double-click on the section called “**body_entered (body:Node)**”.



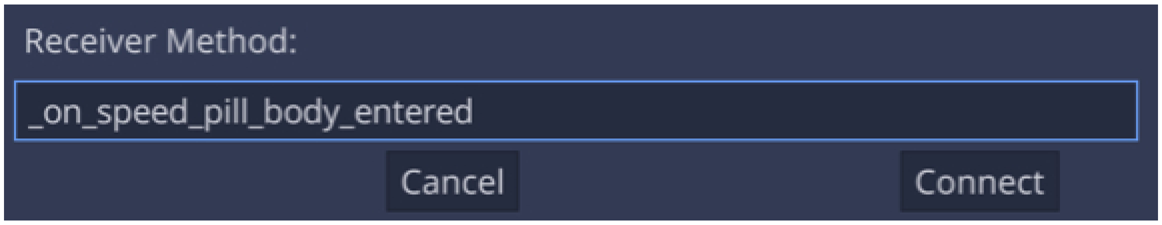
area_entered(area: Area)
area_exited(area: Area)
area_shape_entered(area_id: int, area: Area, area_shape: int, local
area_shape_exited(area_id: int, area: Area, area_shape: int, local_s
body_entered(body: Node)
body_exited(body: Node)

- In the new window, select the node **speed_pill**.



From Signal:
body_entered
Connect to Script:
speed_pill (Connecting From)
CollisionShape
CSGSphere

- Leave the default text in the field "**Receiver Method**" and press on "**Connect**".



Receiver Method:
_on_speed_pill_body_entered

- That should open the script **speed_pill.gd** and show that a new function named **_on_speed_pill_body_entered** has been created.
- This function will be called every time a node of type **KinematicBody**, including the player, will enter the area defined by its collision shape.
- Please modify this function as follows (new code in bold):

```
func _on_speed_pill_body_entered(body):
if (body.is_in_group("player")):
body.speed += 5
queue_free()
```

In the previous code:

- We check whether the object entering this area is a node that belongs to the group called **player**.
- If this is the case, then we increase the value of the variable **speed** that is part of the script attached to the player node (i.e., the script **move_net_player.gd**).
- We then destroy the speed pill.

You can now save your script and scene, open and run the scene **networked_scene** and check that after collecting a speed pill that the player character's speed has been increased.

IMPLEMENTING A MULTIPLAYER-NETWORKED MODE

So at this stage, we have a single-player game whereby the player character can navigate the level, collect pills, increase its score and also collect speed pills that will increase its speed.

Our goal is now to network this scene so that it can be played simultaneously by two remote players.

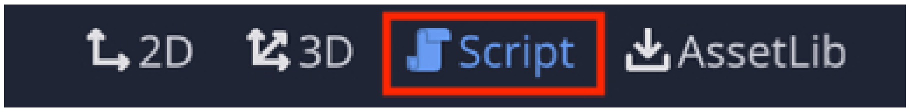
First, we will do as follows:

- Create a lobby: the lobby will be the first scene where players can join the game by clicking on a button.
- There will be two buttons available: a button to host the game, and a button to join the game.
- One player will create a server that will host the game by clicking on the corresponding button.
- The second player will join the game hosted by the first player by clicking on the corresponding button.
- Once the second player has joined the game, the scene **networked_scene** will be loaded for both players.
- Both scores will be displayed on screen.
- The first player who has managed to collect 10 pills will win the contest and each player will then see a message indicating whether they have won or lost.

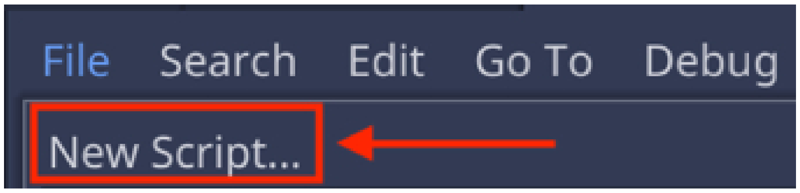
Defining default network settings and functions

Before we create the scene for the lobby (the place from where players can host or join the game), we will need to create a script where information can be saved persistently across scenes and with functions that can be accessed throughout our project. For this purpose, we will create a singleton script, a script only instantiated once that will be used to initialize the networked game, by creating the server, and connecting new players to the server.

- Please switch to the **Script** workspace.



- Create a new Script (**File | New Script**).



- Name this script **Global.gd**.
- Please add this code to the script (new code in bold).

```
extends Node
const DEFAULT_IP = "127.0.0.1"
const DEFAULT_PORT = 31400
const MAX_CLIENTS = 2
var player1_id = 0
var player2_id = -1
var player1_score = 0
var player2_score = 0
```

In the previous code:

- We specify that the default address for the server is **127.0.0.1**.
- The default port will be **31400**.
- The maximum number of clients connected to the server will be **2**.

Now that the defaults settings have been specified for our networked game, we will define two functions, each to be used whenever a player either creates a new game server, or joins a game server.

- Please add the following function to the script:

```
func create_new_server():
var peer = NetworkedMultiplayerENet.new()
peer.create_server(DEFAULT_PORT, MAX_CLIENTS)
get_tree().set_network_peer(peer)
print("Just created Server")
```

In the previous code:

- We create the function **create_new_server**.
- This function will be called whenever the player will decide to create a new server (and a game).
- We create a variable called **peer** and it is created using the function **NeworkedMultiplayerENet.new**. This variable (i.e., **peer**), is a **NetworkMultiPlayerPeer** node, a type of node that is required when creating a networked scene.
- The **NetworkMultiPlayerPeer** node is then initialized as a server, with its corresponding listening port and its maximum number of clients.
- Finally, the **NetworkMultiPlayerPeer** node is associated with the current scene.
- We also display a message confirming that the server has been created in the **Output** window.

Now that the code that manages the creation of a game server has been created, we can create the code that handles the addition of a new player to a game server.

- Please add the following function:

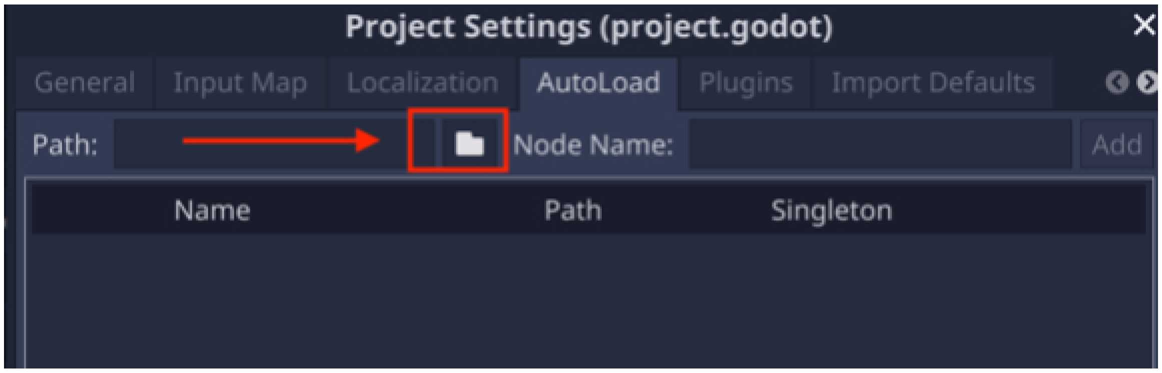
```
func join_new_server():
var peer = NetworkedMultiplayerENet.new()
peer.create_client(DEFAULT_IP, DEFAULT_PORT)
get_tree().set_network_peer(peer)
print("Just joined server")
```

In the previous code:

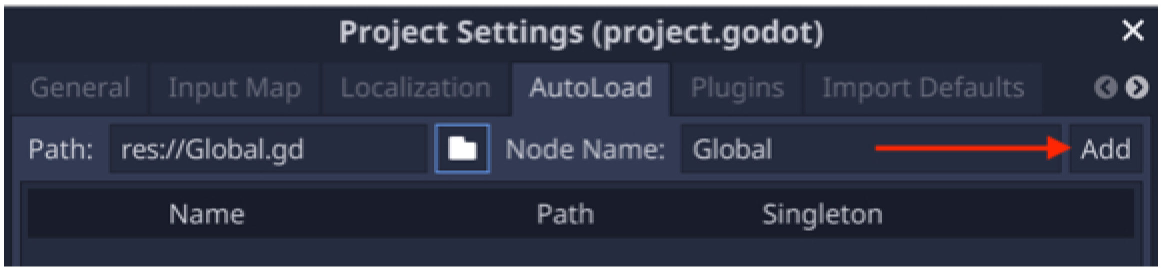
- We create the function **join_new_server**.
- This function will be called whenever a player will decide to join a new server (and a game).
- We create a variable called **peer** and it is created using the function **NeworkedMultiplayerENet.new**. This variable (i.e., **peer**), is a **NetworkMultiPlayerPeer** node, a type of node that is required when creating a networked scene.
- The **NetworkMultiPlayerPeer** node is then initialized as a client, with the address of the server and its corresponding listening port.
- Finally, the **NetworkMultiPlayerPeer** node is associated with the current scene.
- We also display a message confirming that the client has joined the server in the **Output** window.

So at this stage, the script called **Global.gd** has been created, and we aim to use it as a singleton; to do so, we need to modify our project's settings so that the singleton script is loaded whenever the main scene is open.

- Please save the script (**CTRL + S** or **CMD + S**).
- Please open the **AutoLoad** project settings: Select **Project | Project Settings | AutoLoad**.
- In the new window, click on the white folder to the right of the label "**path**", as per the next figure.

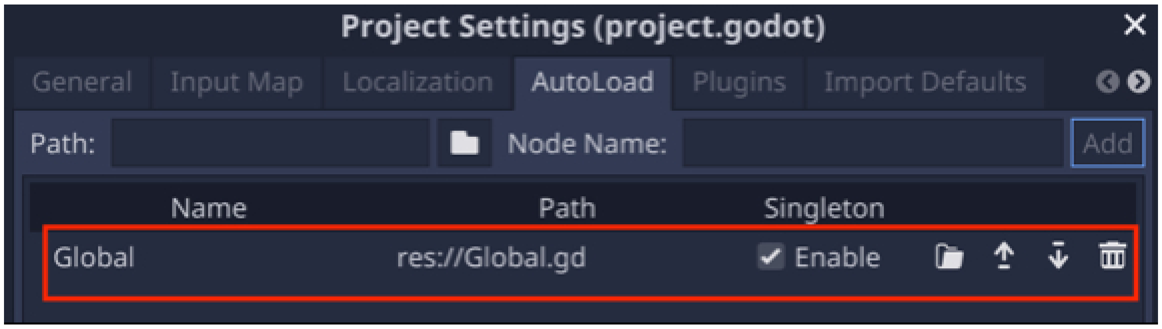


- In the new window, select the file **Global.gd** that you have created earlier, and press the button labeled "**Open**".
- In the next window, press the button labeled "**Add**".



Path: res://Global.gd
Node Name: Global

- This will create a new line with a path to the script that we have just added as a **singleton**.


Project Settings (project.godot)

General | Input Map | Localization | AutoLoad | Plugins | Import Defaults

Path: | Node Name: | Add

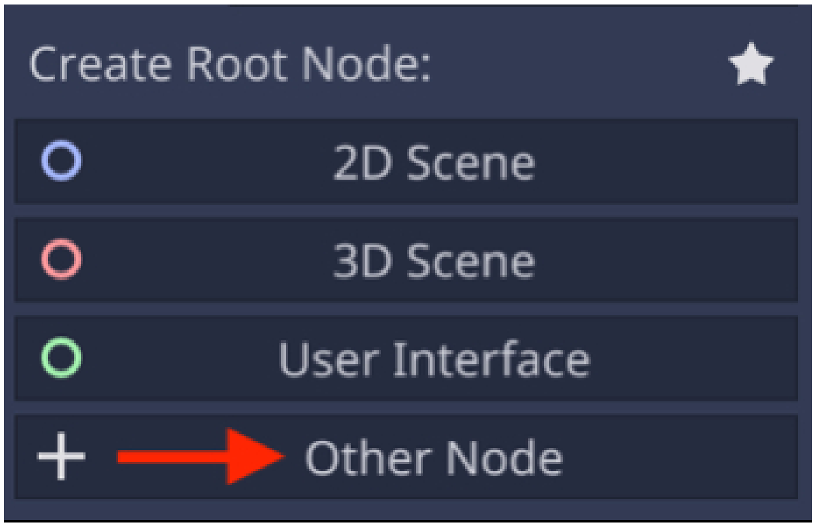
Name	Path	Singleton
Global	res://Global.gd	✓ Enable

- Please make sure that the option “**Enable**” is selected.
- You can now press the button “**Close**” to close this window.

Creating the lobby

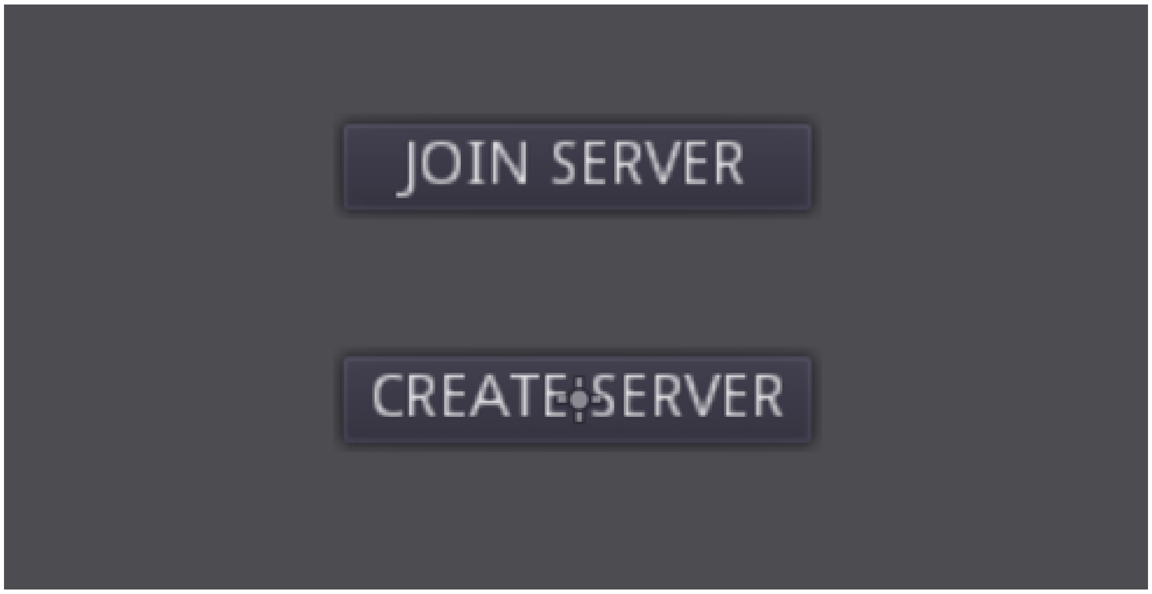
Once the default settings and functions have been created, it is time to create the lobby where the players will be able to create a game server or join an existing one.

- Please create a new scene (**Scene | New Scene**).
- In the new window, please select the option “**Other Node**”.



- In the new window, select the node type “**Node2D**” and click on “**Create**”.

- This will create a scene with a default node called **Node2D**.
- Please rename this node **main**.
- Add a child node of type **Button** to the node **main** and rename the new node **bt_create_server**.
- Change the label of this button to "**CREATE SERVER**" and use the font of your choice for this button.
- Duplicate this button, rename the duplicate **bt_join_server**, change its label to "**JOIN SERVER**", and move this button just below the previous one.
- Save this scene as **network_lobby.tscn (Scene | Save Scene As)**.



Once these buttons have been created, we will need to handle clicks on these buttons and to execute a specific function when this happens. To keep everything tidy, we will create just one script that will include both functions to be called when either of the buttons are pressed, and we will store this script on the node called Main.

The general sequencing of the events is as follows:

- If one player clicks on the **CREATE** button, the system will create a new server.

- If another player clicks on the **JOIN** button, the system will try to connect him/her to the server, and then notify the game using the event "**connected to server**".
- Once we receive this event, we can load the new scene (i.e., the scene **networked_scene** that we have created earlier) and hide the lobby.

Let's create this code:

- Please open (or switch to) the scene **network_lobby.tscn**.
- Attach a new script called main.gd to the node **main** and call this script **main.gd**.
- Modify the function **_ready** as follows (new code in bold).

```
func _ready():
get_tree().connect("network_peer_connected",self,"player_connected")
```

In the previous code, we connect the event "**network_peer_connected**" to the function **player_connected**; this signal/event is triggered once a client has joined the server and is emitted to both clients and the server.

- Please add the following function to the script:

```
func player_connected(id):
Global.player2_id = id
var game = preload("res://networked_scene.tscn").instance()
get_tree().get_root().add_child(game)
hide()
```

In the previous code:

- We create the function **player_connected**

- This function is called whenever a client connects to the server, and its id is also passed as a parameter, so that it can be used later on to identify the player.
- We set the id of the second player.

Finally, we just need to create the two functions to be called whenever we press the button **CREATE** or **JOIN**.

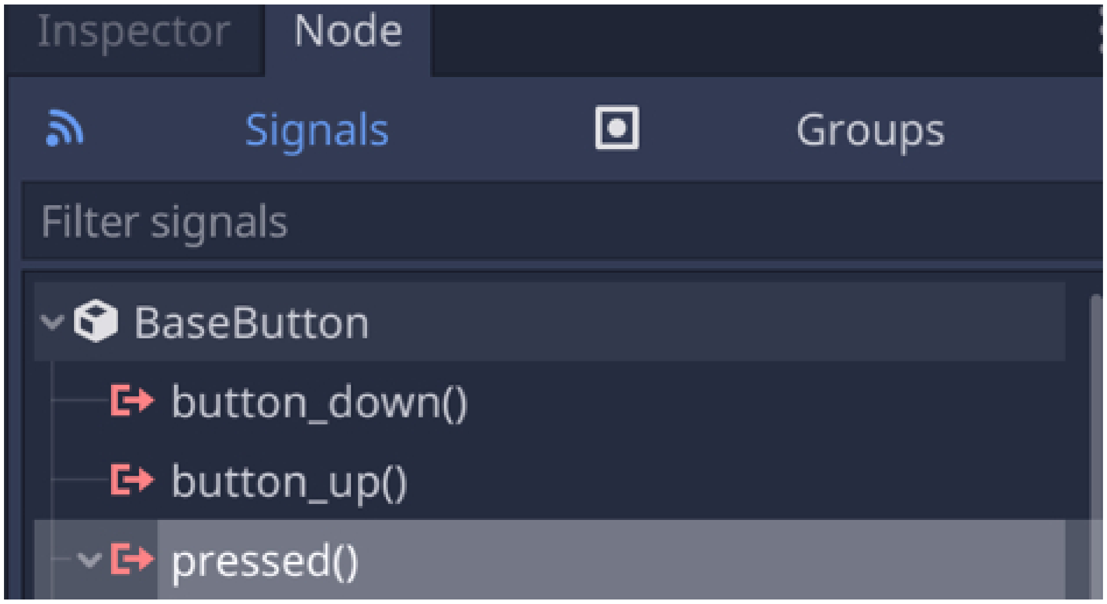
- Please add the following two functions to the script.

```
func on_create_bt_pressed():
Global.create_new_server()
func on_join_bt_pressed():
Global.join_new_server()
```

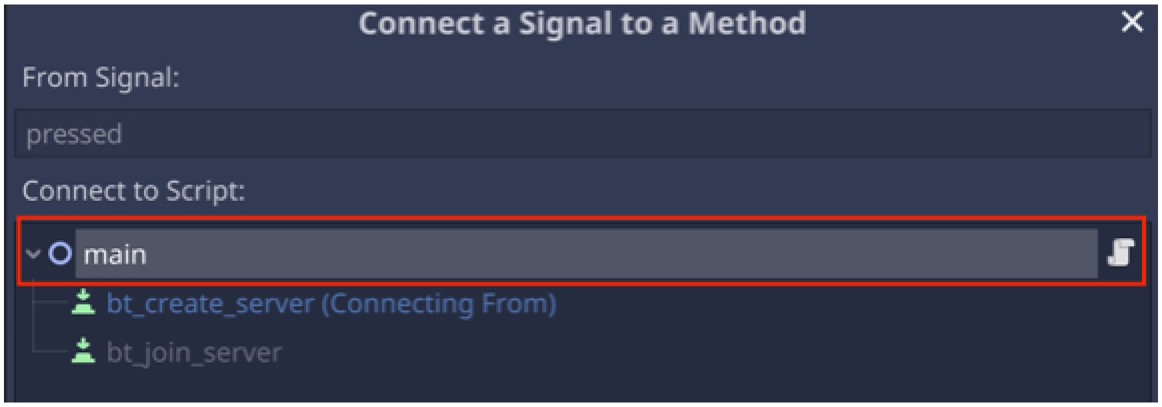
In the previous code, we create two functions **on_create_bt_pressed** and **on_join_bt_pressed**; these will be used when a player presses the **CREATE** and **JOIN** buttons, respectively; this being said, we still need to link the press event on any of these buttons to one of these function, and that will be done in the next sections.

Finally, we need to link the two buttons to these functions:

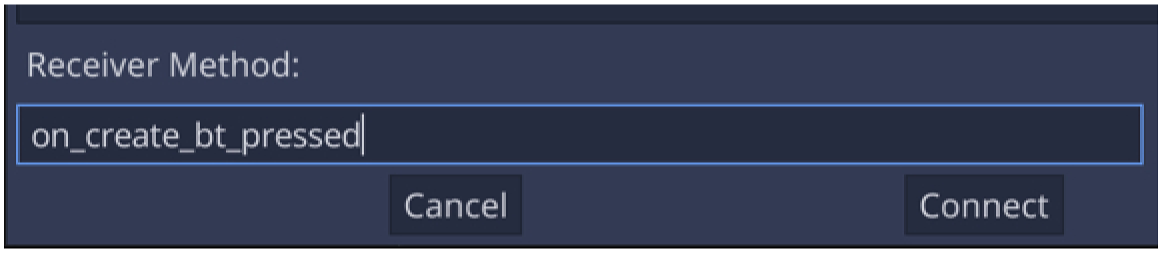
- Please select the node **bt_create_server**, select the tab **Node | Signals**, and double-click on the event called **pressed**, as per the next figure.



- In the new window, select the node **main**.


Connect a Signal to a Method
From Signal:
pressed
Connect to Script:
main
bt_create_server (Connecting From)
bt_join_server

- Type **on_create_bt_pressed** in the text field "**Receiver Method**" and press **Connect**.



- Repeat the previous steps with the node **bt_join_server**, except that the function for the **Receiver Method** field should be **on_join_bt_pressed**.

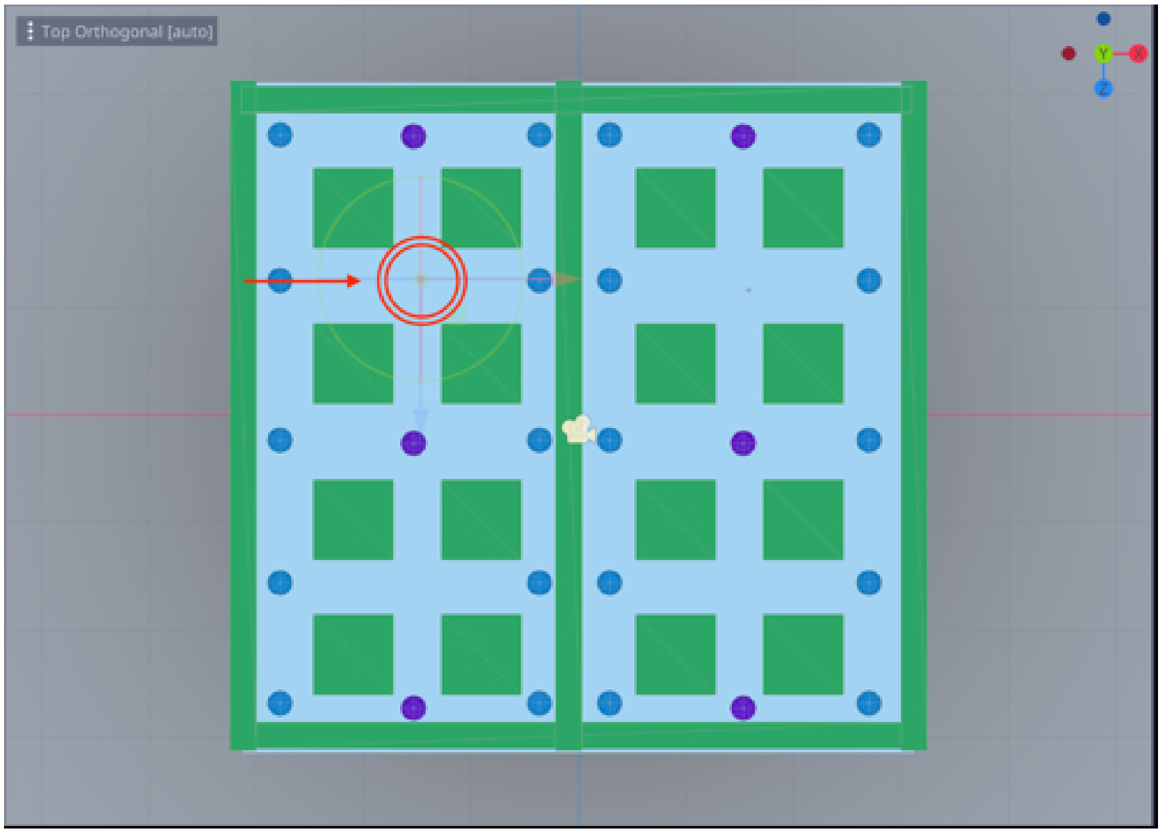
Adding and controlling players remotely

So at this stage, we have created a user interface for the lobby, where a player can create the server, and the other one join the server; when this is done, each player should be able to see the networked scene; however, we still need to add each player to the scene and ensure that their movement is synchronized across the network; so the idea in this section is to:

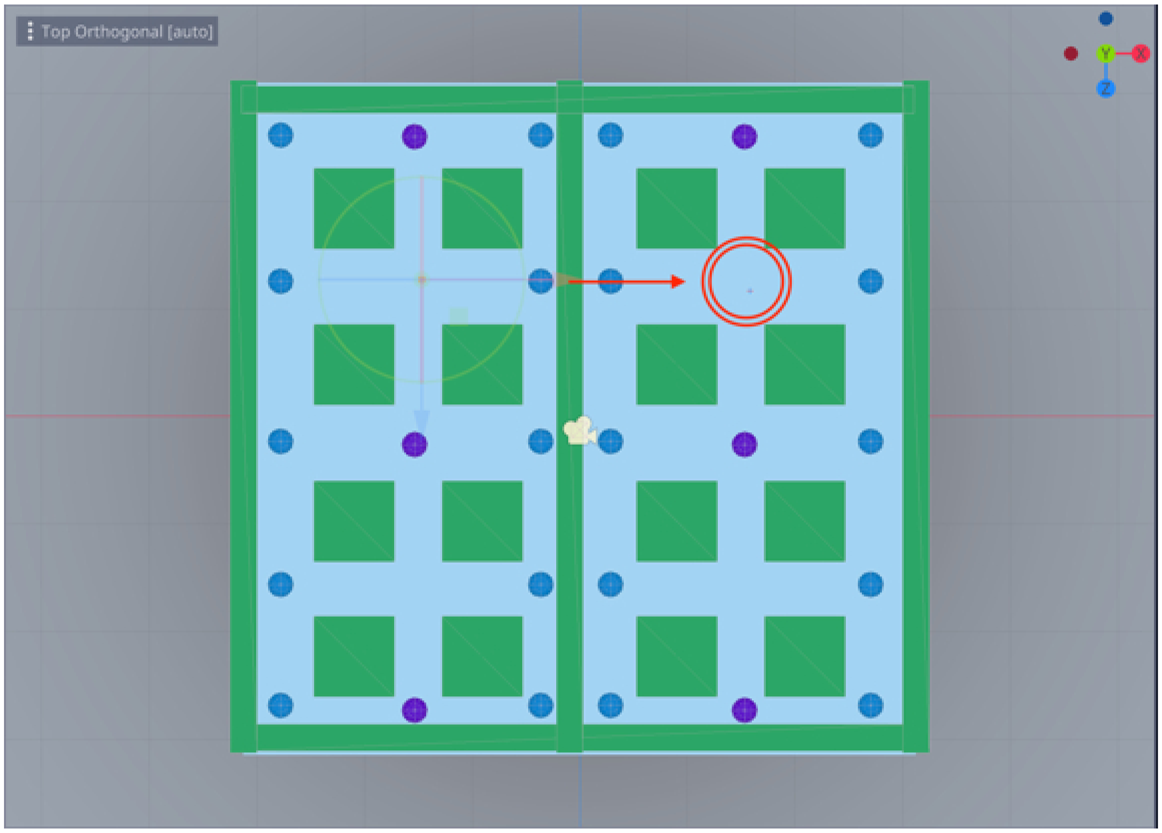
- Define default positions for each player when the game starts.
- Add these players at their corresponding starting positions.
- Make sure that these player characters are only controlled by their corresponding players.
- Synchronize the movement of these player characters over the network.

So first let's define and allocate the position for each player:

- Please save the current scene (**Scene | Save Scene**).
- Please open the scene **networked_scene**.
- Remove the node for the player (i.e., **KinematicBody**): right-click on the node **KinematicBody** and select "**Delete Node**" from the contextual menu.
- Add a **Spatial** node as a child of the node called **level1**.
- Rename this new node **player1_pos**, and move it to a location on the left side of the maze where there are no pills, for example the position **(-11, 0.3, -10)**, as illustrated in the next figure.



- Duplicate this node and call it **player2_pos**.
- Move the node **player2_pos** to a location on the right side of the maze where there are no pills, for example the position **(11, 0.3, -10)**, as illustrated in the next figure.



Now that we have defined the locations for each player, we will modify our script so that the player characters are effectively moved to these locations at the start of the game.

- Please open the scene **networked_scene.tscn**.
- Select the node **level1**. Add a new script to it an call the script **main_net_game.gd**.
- Please open the script **main_net_game.gd**.
- Add the following code at the start of the script (new code in bold).

extends Spatial

onready var player1_pos = get_node("player1_pos")

onready var player2_pos = get_node("player2_pos")

In the previous code, we declare two variables **player1_pos** and **player2_pos** which refer to the two nodes that we have created previously.

- Please modify the function **_ready** as follows:

```
func _ready():
var player1 = preload ("res://net_player.tscn").instance()
player1.set_name(str(get_tree().get_network_unique_id()))
player1.set_network_master(get_tree().get_network_unique_id())
player1.global_transform = player1_pos.global_transform
add_child(player1)
var player2 = preload ("res://net_player.tscn").instance()
player2.set_name(str(Global.player2_id))
player2.set_network_master(Global.player2_id)
player2.global_transform = player2_pos.global_transform
add_child(player2)
```

In the previous code:

- We declare the variable **player1** that is an instance of the scene **player.tscn**.
- We set the name of the node **player1**.
- We ensure that the master of this node is the server.
- We change the position of this node so that it matches the position of the node **player1_pos**.
- We then add the **player1** node to the scene.
- For the second player (i.e., the client) the procedure is similar, except that the name of the node will be set with the id that has been saved previously (in the script **Global.gd**), and we ensure that the second player, who is the client, is the master of its corresponding player character.

Next, we will need to synchronize the position of the players over the network; for this purpose, we will modify the script called **move_net_player.gd**:

- Please open the script **move_net_player.gd**.
- Modify the script as follows (new code in bold):

```
else: direction = Vector3(0,0,0)
direction = direction.normalized()
if (direction != Vector3()):
if (is_network_master()):
move_and_slide(direction*speed,Vector3.UP)
rpc_unreliable("set_remote_pos", global_transform.origin)
```

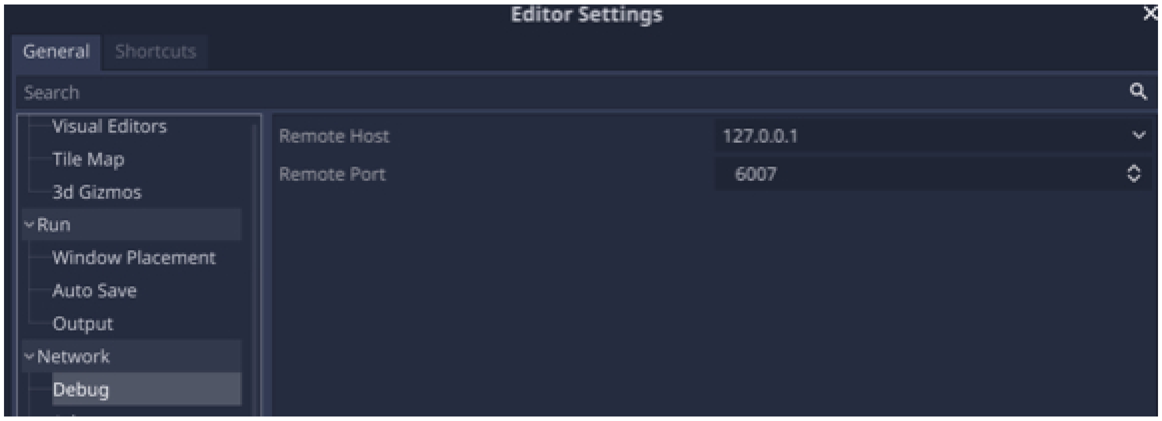
In the previous code:

- We make sure that the player character only moves when a direction has been set (this is to avoid any jittery movement)
- We check whether the player can move the current player character.
- If this is the case, then we move the player character.
- We also call the function **set_remote_position** passing the position of the node as a parameter; to call this function, we use the keyword **rpc_unreliable**, so that this information is sent to the other peer (i.e., the other player); the **rpc_unreliable** method is usually employed to synchronize positions across the network; because this instruction is sent almost every frame, even if the network is unreliable, the data will eventually arrive, and fast.
- Please add the following code to the script **move_net_player.tscn**.

```
remote func set_remote_pos(pos):
global_transform.origin = pos
```

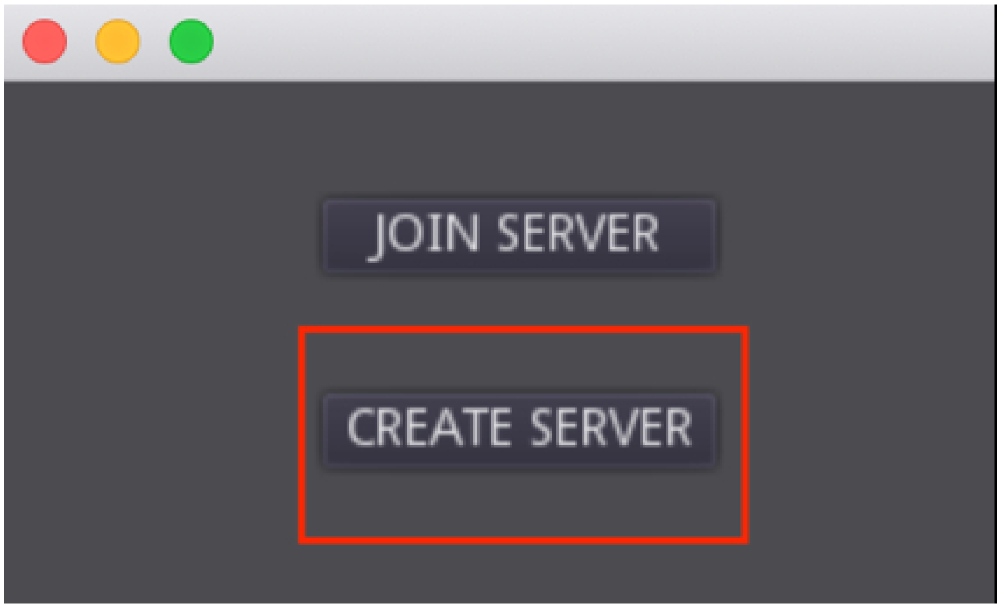
We can now start to test the networked scene to see if both players can see each other:

- Please launch another instance of Godot, and open your project again, so that you have the project open twice: you can go navigate to your file system where the project is saved and double click on the file called **project.godot**.
- In the first project, please select: **Editor | Editor Settings | Network| Debug**.
- Set the **Remote Port** attribute to **6007**.

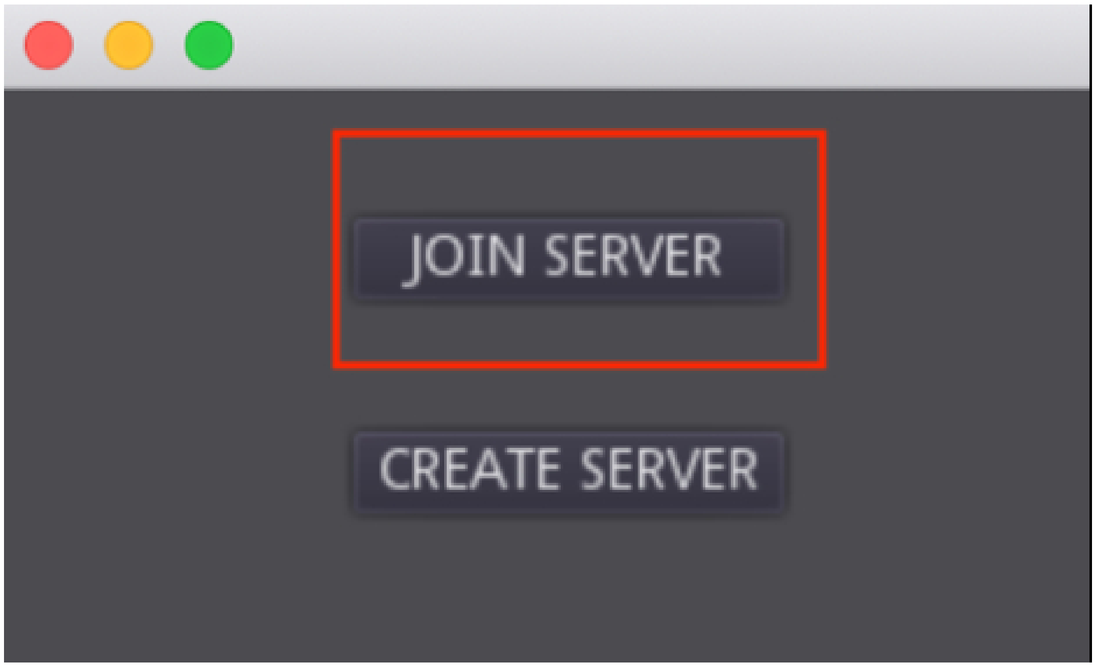


Remote Host 127.0.0.1
Remote Port 6007

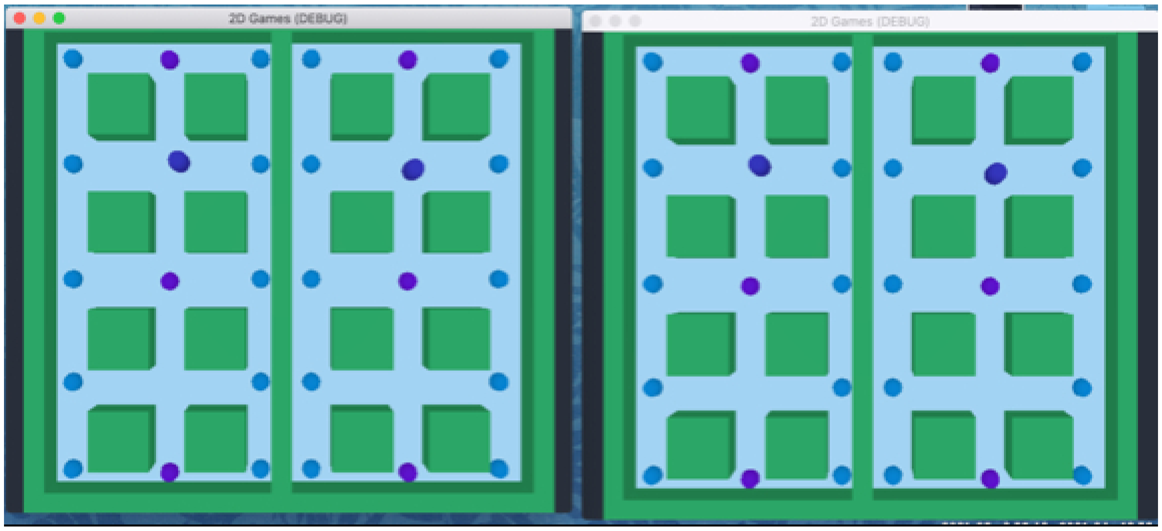
- Using the second instance of Godot, please set the remote port to **6008**: select: **Editor | Editor Settings | Network| Debug** and set the set the **Remote Port** attribute to **6008.**
- You can now open the scene **network_lobby.tscn** and play it in the first instance of the scene and select to "**Create**" a server.



- You can now open the scene **network_lobby.tscn**, play it in the second instance of the Godot, and select to "**Join**" the server.



- This should open the main scene twice, once for the client and once for the server, as per the next figure.



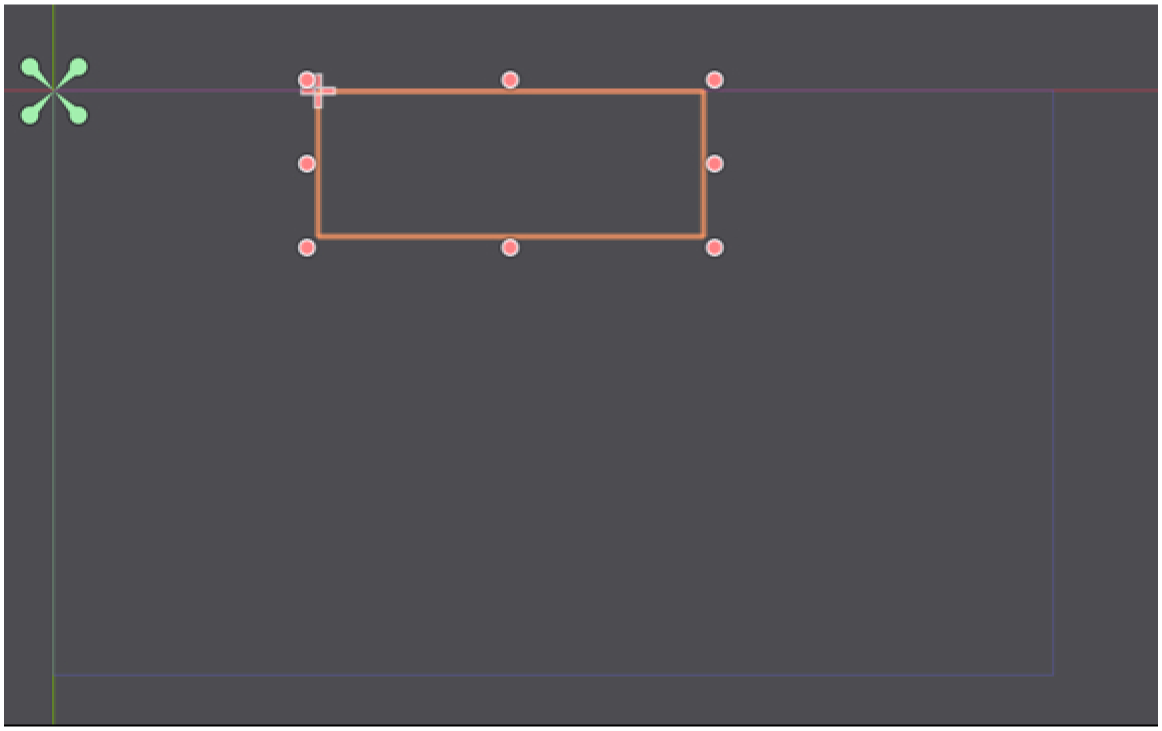
Once this is done, you can click on one of the windows and use the arrow keys, you should see that your movement is replicated in the other window.

Updating the score

So at this stage, we have managed to create a networked game with two players who can collect pills and move around the level, while their position is synchronized over the network.

In this section, we will keep track of the score for each of the players, and also display this score on screen.

- Please open the scene **networked_scene**.
- Add a child of type **Label** to the node **level1**.
- Rename this new node **label_game_info**.
- Using the **2D** workspace, move this node at the top of the screen and resize it, with a font of your choice, as per the next figure.



Next, we will control how the score is displayed in this node.

- Please open the script **move_net_player.gd**.
- Add this code at the beginning of the script (after the line that starts with **extends**).

```
onready var game_info = get_node("../label_game_info")
```

In the previous code, we create a new variable that is linked to the node that we have just created.

- Please modify the function **update_score** as follows:

```
func update_score(new_increment):
if (is_network_master()):
Global.player1_score += new_increment
score = Global.player1_score
```

```
else:
Global.player2_score += new_increment
score = Global.player2_score
rpc("update_score_remote")
#score += new_increment
#print("Score: "+str(score))
```

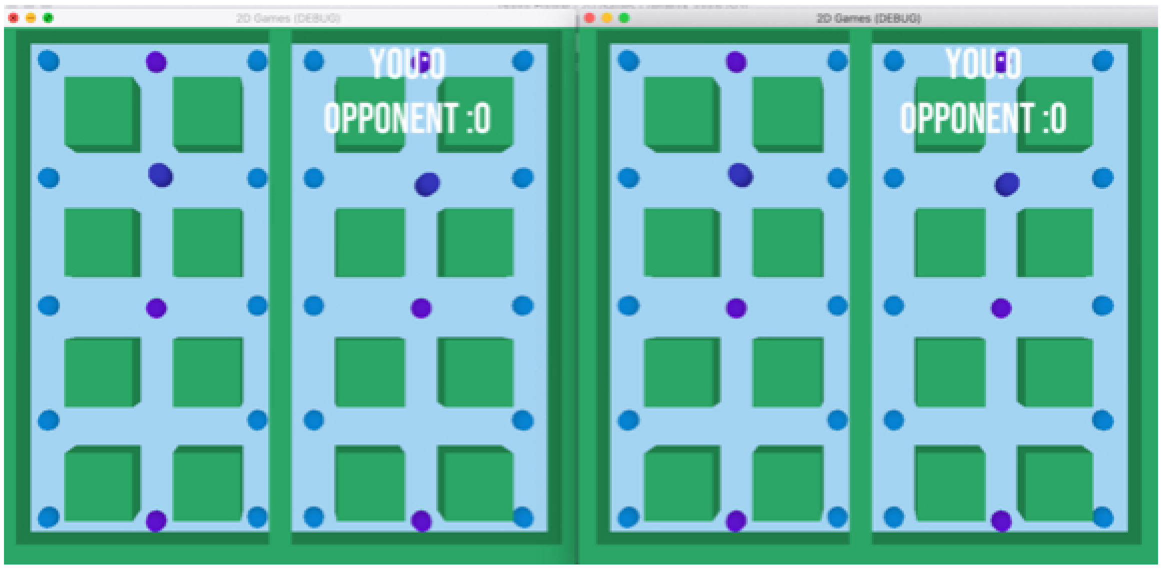
In the previous code, we save the score in the variables **Global.player1_score** or **Global_player2_score**, depending on whether the current player is the server (i.e., **player1**) or the client (i.e., **player2**). We also make an **rpc** call (to the other peer) to the function **update_score_remote** (that yet has to be created) so that the scores are updated for both players.

- Please add the following function to the script **move_net_player.tscn**:

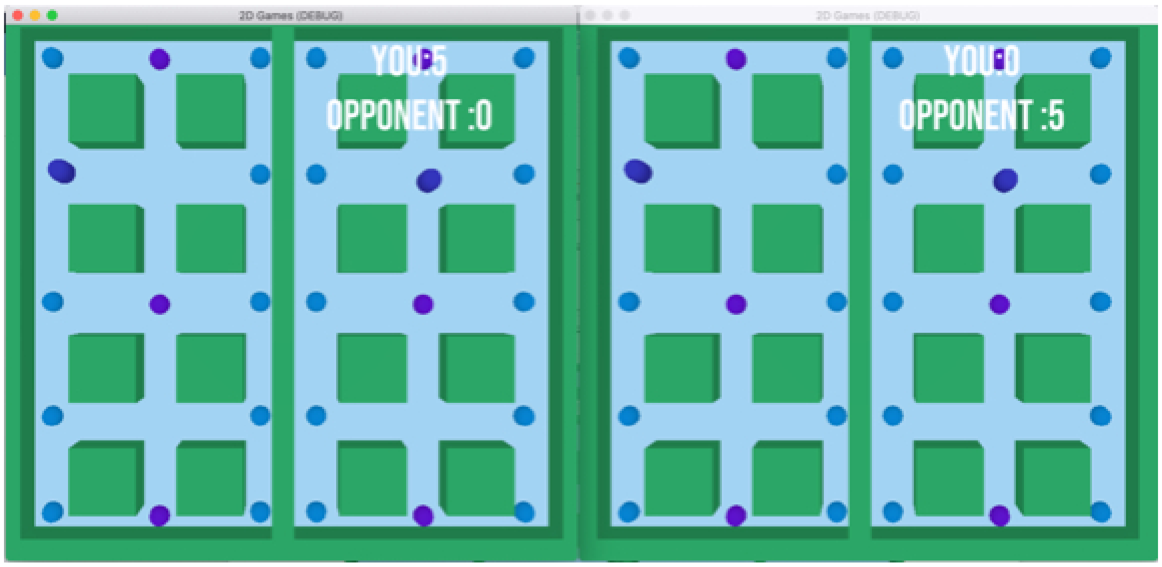
```
remotesync func update_score_remote():
game_info.text = "You:"+ str(Global.player1_score) + "\n"+ "Opponent :" + str(-
Global.player2_score)
```

In the previous text we display both scores on screen.

- You can now test the game by opening the lobby in two different instances of Godot, and by collecting items; you should see that the score is updated for both players as per the next figures.



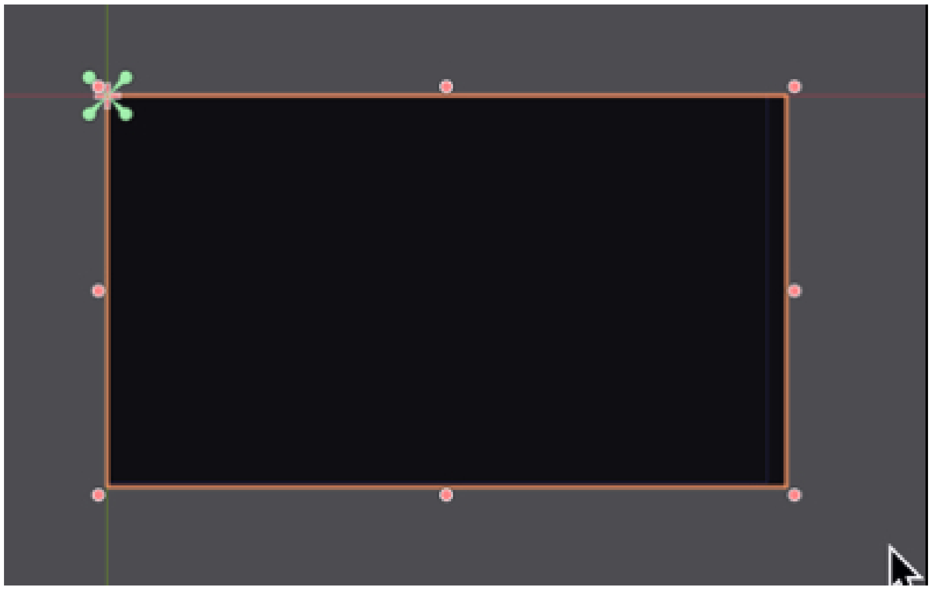
- After collecting a pill, both screens should be updated, as per the next figure:



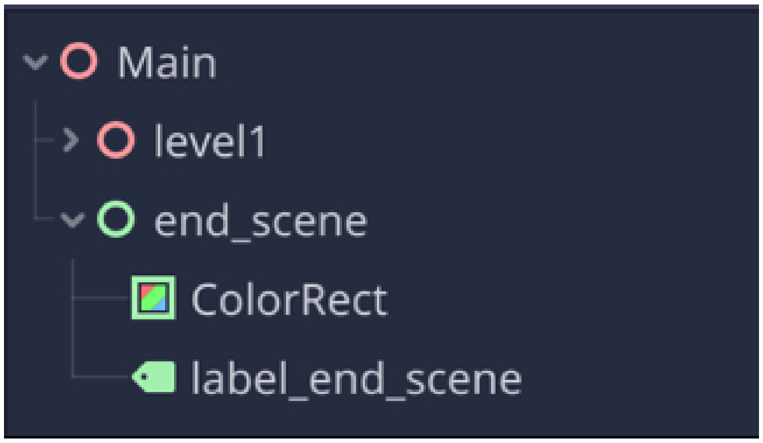
The last step is to detect when one of the players has reached a given score, for example **15**, and to then display a screen that specifies who won.

- Please open the scene **networked_scene**, if it is not already open.
- Create a new **Control** node as a child of the node **Main**.

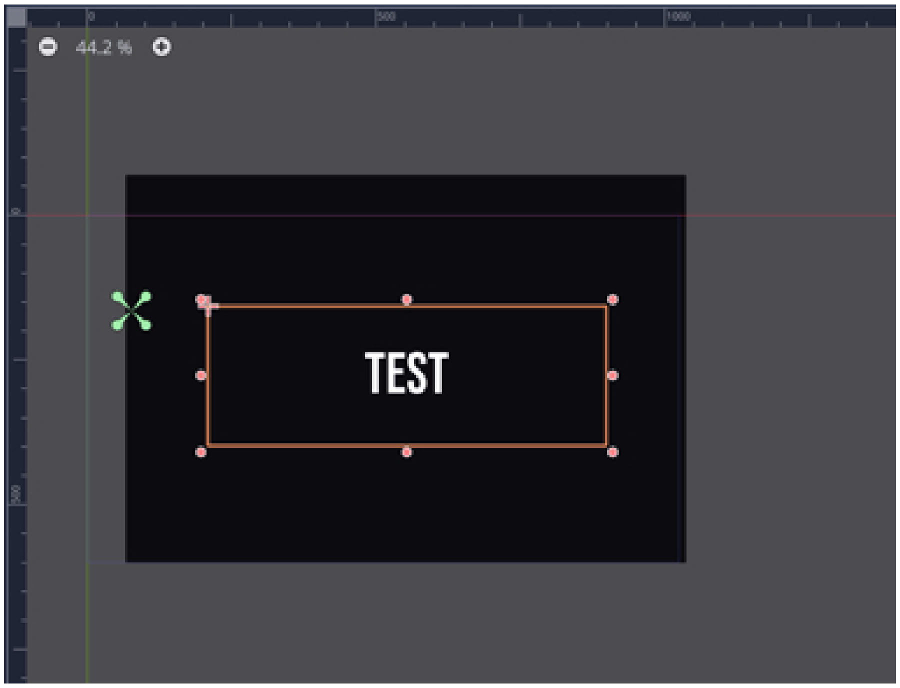
- Call the new node **end_scene**.
- Add a **ColorRect** node as a child of the node **end_scene**.
- Using the **Scale** tool, scale-up this node so that it covers the entire visible screen area and set its color to black, or any other color of your choice.



- Add a label node as a child of the node **end_scene**, and call the new node **label_end_scene**.



- Modify the size and position of the note **label_end_scene**, so that it occupies the center of the screen.



- Add a new script to the node **label_end_scene** and call this script **label_end_scene.gd**.
- Open the script, and modify it as follows (new code in bold).

```
extends Label
func _ready():
text = ""
remotesync func update_ui():
if (Global.player1_score > Global.player2_score):
text = "YOU WON"#+str(Global.player1_name)
elif(Global.player1_score < Global.player2_score):
text = "YOU LOST"# +str(Global.player2_name)
```

In the previous code we display a message depending on whether the player has lost or won.

Next, because we want this message to appear only when the game is over, we need to hide the **end_scene** node at the beginning of the game, so please, open the script **main_net_game.gd** and modify the code as follows (new code in bold):

```
onready var label_end_scene = get_node ("../end_scene")
func _ready():
label_end_scene.hide()
```

In the previous code, we create a new variable called **label_end_scene** that is linked to the node **end_scene**, we then hide this node when the game starts.

Next, we need to display the **end_scene** node whenever the game is over:

- Please save the script **main_net_game.gd**.
- Open the script **move_net_player.gd**.
- Add the following code at the beginning of the script (after the first line).

```
onready var label_end_scene = get_node("../../end_scene/label_end_scene")
onready var end_scene = get_node("../../end_scene")
```

In the previous code, we create two variables that will be used to refer to the end scene, or to the label present in the end scene.

- Please modify the function **update_score** as follows (new code in bold)

```
else:
Global.player2_score += new_increment
score = Global.player2_score
if (score >= 15) : rpc("end_game")
rpc("update_score_remote")
```

In the previous code, we call the function **end_game** if the score of the current player is equal to or greater than **15**.

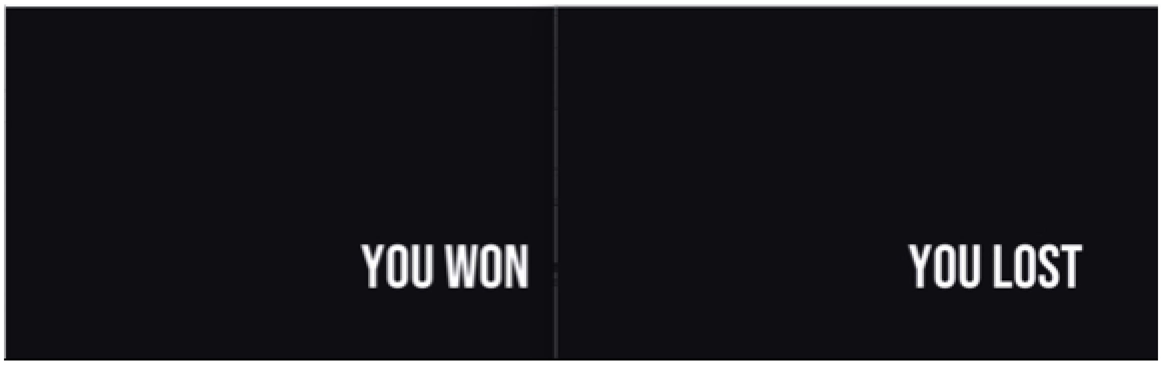
- Please add the following function to the script **move_net_player.gd**:

```
remote func end_game():
end_scene.show()
label_end_scene.rpc ("update_ui")
hide()
```

In the previous code:

- We create a function called **end_game**.
- It displays the end scene.
- It also calls the function **update_ui** from the label present and the end scene.

You can now play the game and test hat after one of the player has reached 15 points that an end screen is displayed for each player with a message that displays either "**You Won**" or "**You Lost**".



LEVEL ROUNDUP

Summary

In this chapter, we have become familiar with networking concepts and we have created a simple multiplayer game. Along the way, we have used several network components as well as some interesting ways to call functions (e.g., rpc calls).

Checklist

You can consider moving to the next stage if you can do the f

- Identify the difference (s) between a client and a server.
- Understand the role of the network master.
- Use singletons.

Quiz

Now, let's check your knowledge! Please answer the following questions (the answers are included on the next page) or specify whether they are TRUE or FALSE.

1. A server usually hosts the game, while a client connects to the game.
2. A server is usually contactable only through a port number.
3. The event "**body_entered**" can be used with **Area** nodes to detect objects entering this area.
4. A lobby can be used to manage the creation and setup of a networked game.
5. When creating a network game, a peer can be created using the following code:

```
var peer = NetworkedMultiplayerENet.new()
```

1. Once a network peer has been created it can only be set as a server.
2. One a network peer has been created and set up it is usually associated with

the current scene using the following code:

```
get_tree().set_network_peer(peer)
```

1. A singleton is created and instantiated only once in the lifecycle of the game.
2. The **rpc_unreliable** method is usually employed to synchronize positions across the network
3. Rpc requests can used for communication between network peers.

Solutions to the Quiz

1. TRUE.
2. FALSE (an IP address may be required).
3. TRUE.
4. TRUE.
5. TRUE
6. FALSE (It can also be a client).
7. TRUE
8. TRUE.
9. TRUE
10. TRUE.

Challenge:

For this chapter, your challenge will be to modify the game as follows:

- Modify the speed pills so that their effect only lasts for five seconds.
- Add pills that will randomly either increase or decrease the speed of the player.

Chapter 4: Creating a Memory Game

In this section, we will learn how to create a memory game similar to the Simon Game, whereby the player needs to remember a sequence of colors associated with a sound. As the game progresses, the sequence will become longer, hence, increasing the challenge for the player.

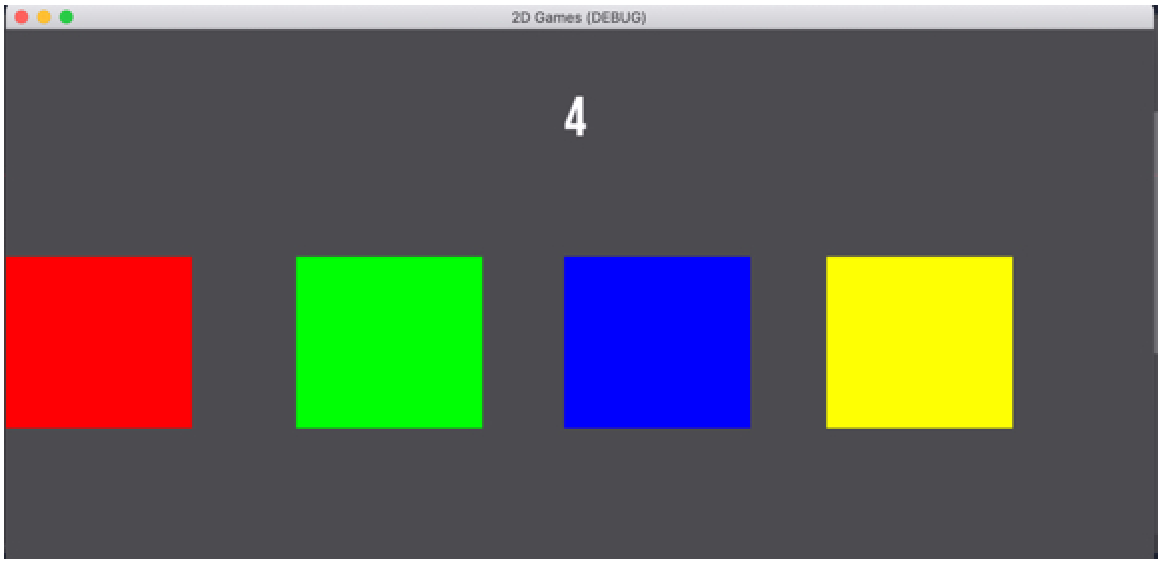
After completing this chapter, you will be able to:

- Generate sound effects from Godot and to modify their frequency.
- Detect when a player has pressed a button.
- Generate colors at random.
- Generate a sequence of colors and sound.
- Record the sequence entered by the player and compare it to the correct sequence.

INTRODUCTION

In this chapter, we will learn how to create visual and audio effects. We will also create a process by which a sequence of colors is created, and then compared to the sequence entered by the player.

The game will consist of four colored boxes that the player will need to press to reproduce a sequence created randomly by the game.

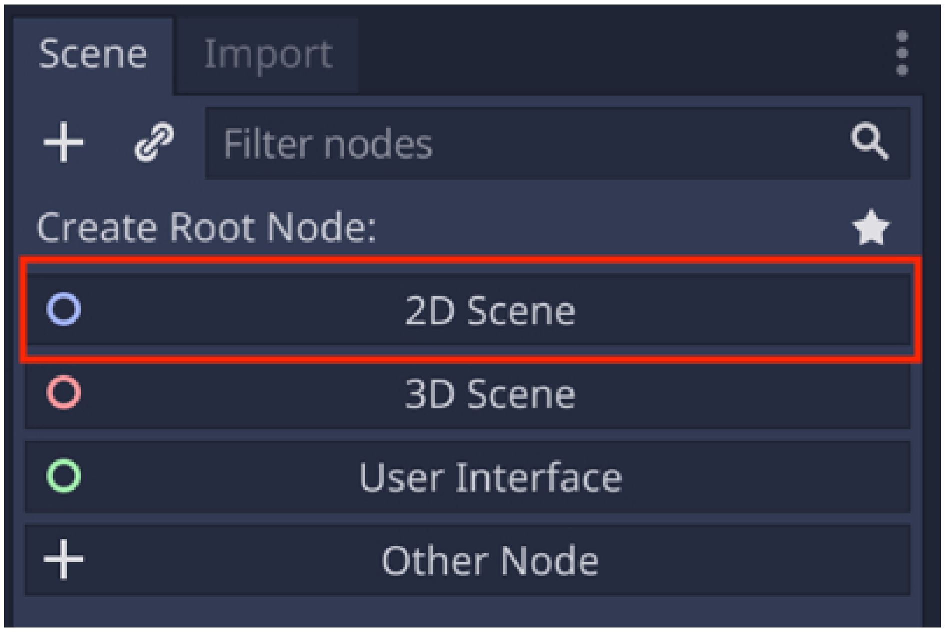


CREATING THE INTERFACE AND THE CORE OF THE GAME

In this section, we will create the interface for the game; it will consist of four buttons of different colors that the player can click on.

So let's proceed:

- Please save your current scene (**File | Save Scene**).
- Create a new scene for this new game (**File | New Scene**).
- In the new window, select the option "**Node2D**".

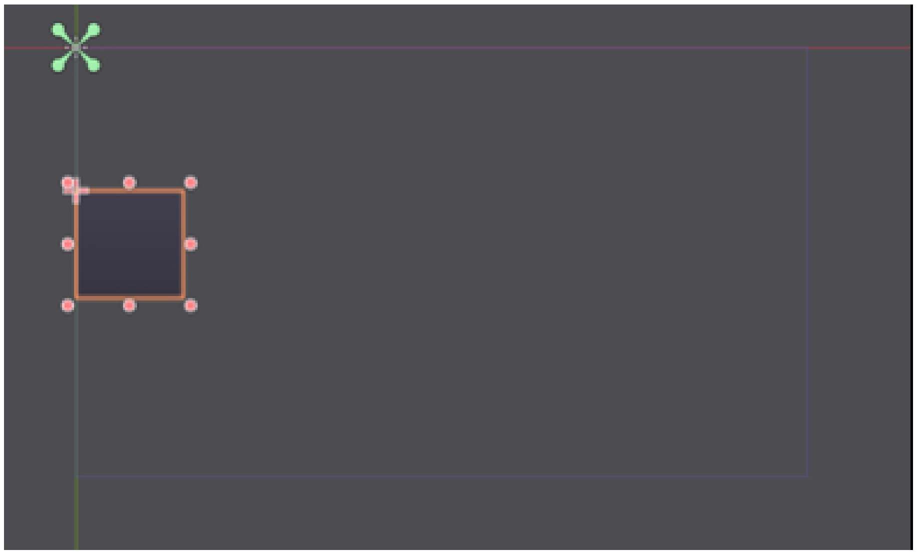


- This will create a new scene with a default node called **Node2D**.
- Please rename this node **main**.
- Save this new scene as **simon_game** (**Scene | Save Scene As**).

Once this is done, we can start to create the buttons that will be used for our game.

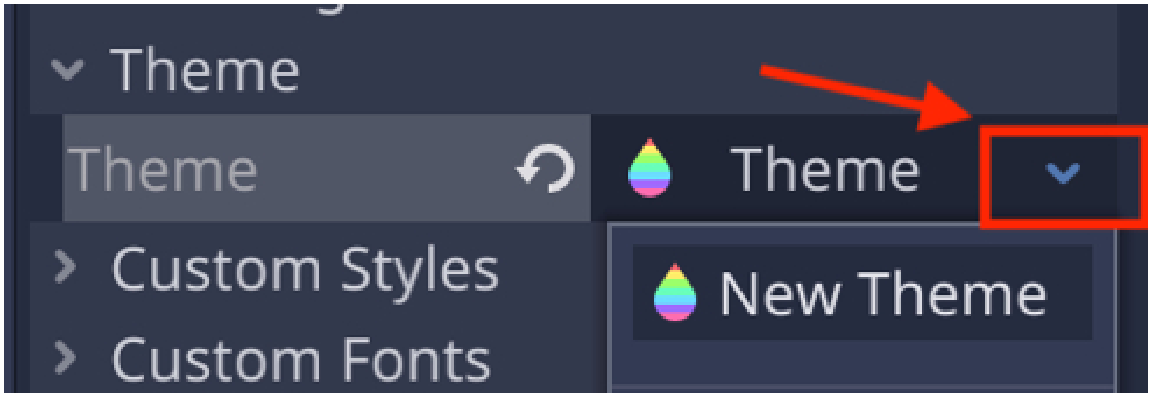
- Please create a new **Button** node as a child of the node **main**.

- Rename this button **red_bt**.
- Using the **Inspector**'s **Control | Rect** section, change its position to **(0, 200)** and its size to **(150, 150)**.

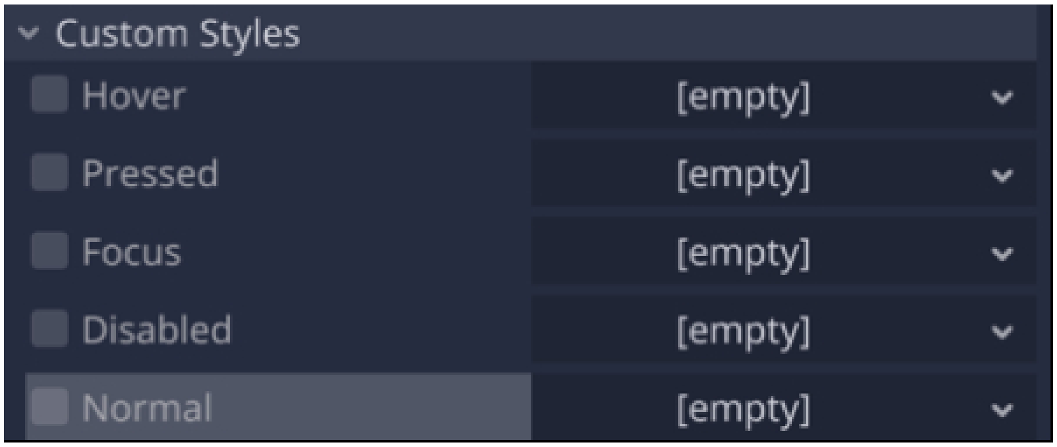


We will now add a color to this button. In Godot, whenever you want to add style (e.g., color) to items that belong to the user interface, you often need to create and apply a new theme and this is what we are about to do:

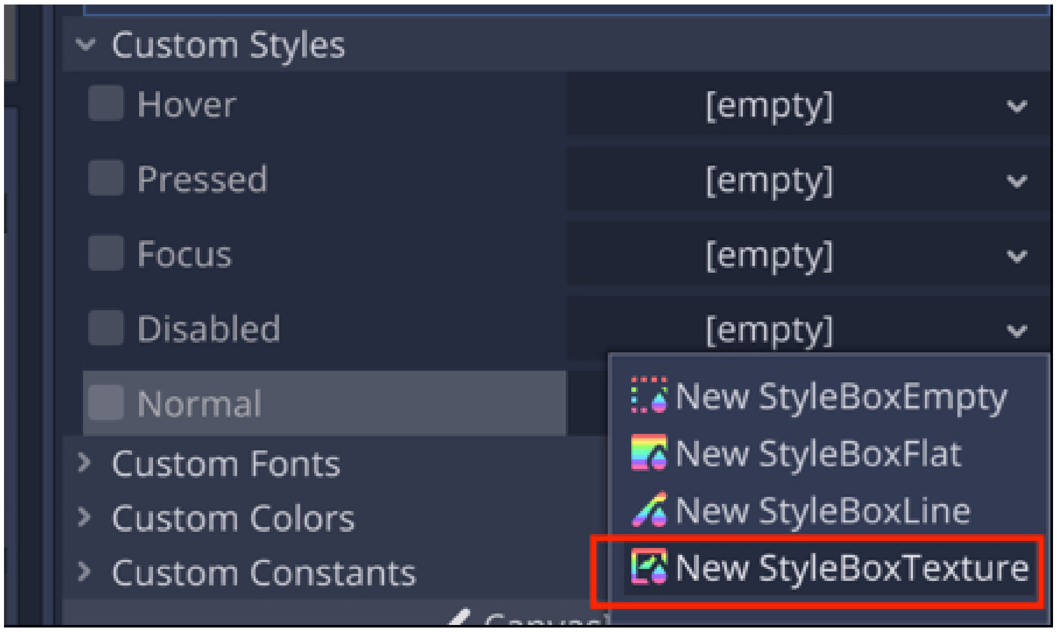
- Please select the node **red_bt**.
- Using the **Inspector**, locate the section **Control | Theme**.
- Expand the section called **Theme** (by clicking on the downwards arrow to the left of the label **Theme**)
- Click on the downward-facing arrow, to the right of the second label **Theme**, and select the option "**New Theme**" from the contextual menu.



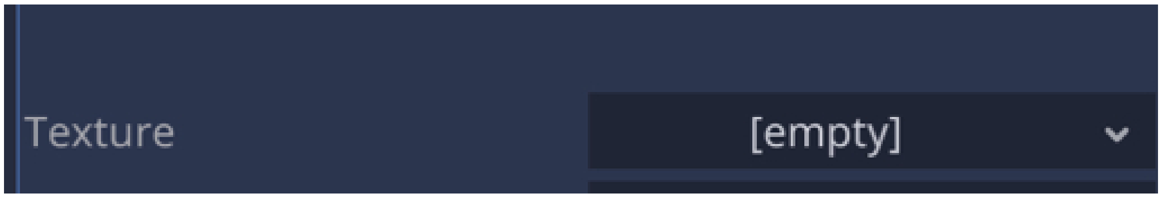
- Expand the section **Custom Styles**, using the arrow to the left of the label **Custom Styles**.



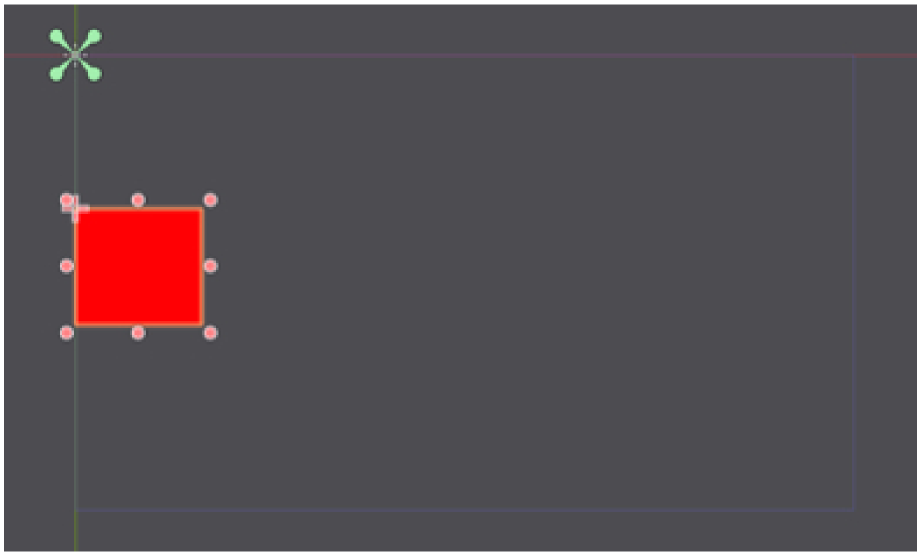
- Click on the downward arrow to the right of the label **Normal**, and select the option **New StyleBoxTexture** from the contextual menu.



- Click on the label **New StyleBoxTexture** to the right of the label **Normal**. This will display additional attributes, including **Texture** (you may need to scroll down).

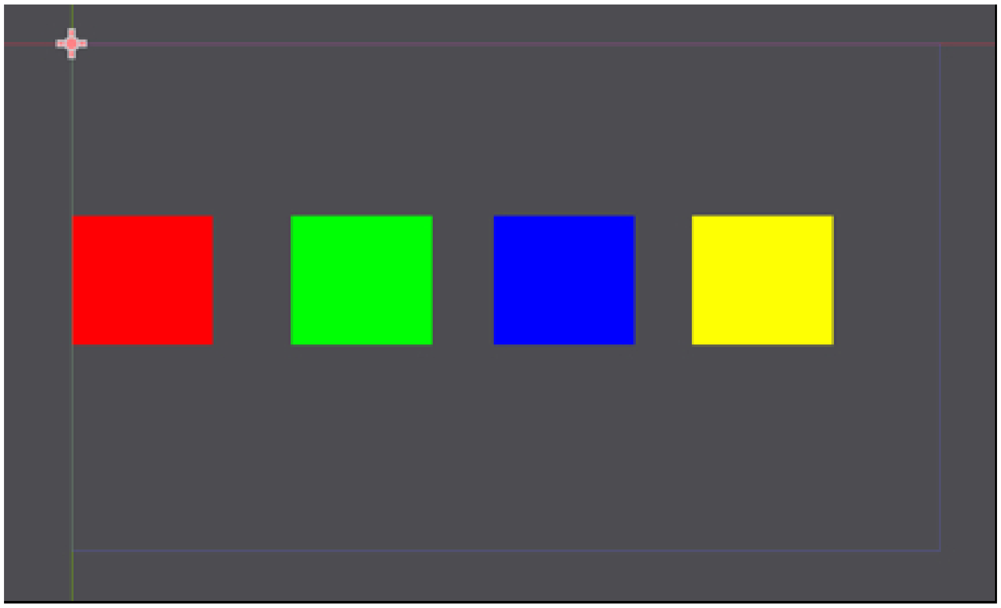


- Please drag and drop the texture called **red.jpg** from the **File System** tab, to the empty placeholder to the right of the label **Texture**.
- Your button should now turn to red.



At this stage, we have managed to create a red button and we now need to create three additional buttons: a green, a blue, and a yellow button.

To do so, please repeat the previous steps (i.e., create new buttons and corresponding themes) to create new buttons (**bt_green**, **bt_blue**, and **bt_yellow**) that will look like and be laid out as the next figure:



If you decide to duplicate the red button, please make sure that you define a new theme for each duplicate; otherwise, using the main theme will mean that the buttons will also have the same color.

DETECTING WHEN BUTTONS HAVE BEEN PRESSED

So at this stage we have four buttons, and we need to detect when the player has pressed one of them; for this purpose, we will do the following:

- Detect the event "**press**" for each button, which is triggered when a button has been pressed.
- Connect the "**press**" event to a function that will be part of a script that is linked to the node main.
- Specify which button was pressed, by passing a unique parameter to that function when the button is pressed.
- Write a message that confirms which button has been pressed.

So let's get started:

- Please attach a new script to the node **main**, and name this script **simon_main.gd**
- Add this new function to the script.

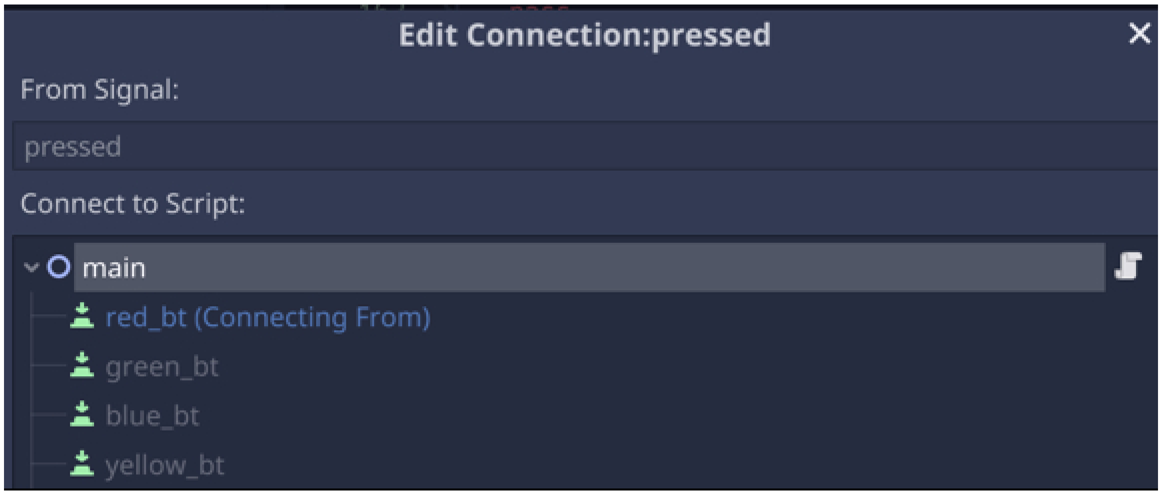
```
func press_bt(name_of_bt):
print("Button Pressed is: "+str(name_of_bt))
```

In the previous code:

- We define a function called **press_bt**.
- This function takes one parameter referred to as **name_of_bt** within the function.
- We then print a message that includes the value of the parameter passed to that function.
- The message uses the keyword **str** to convert the parameter from **int** to **String**.

Next, we need to detect and connect the **press** event on the buttons:

- Please select the node **red_bt**.
- Open the tab: **Node | Signal**.
- Double click on the event **pressed**.
- In the new window, select the node called **main**.

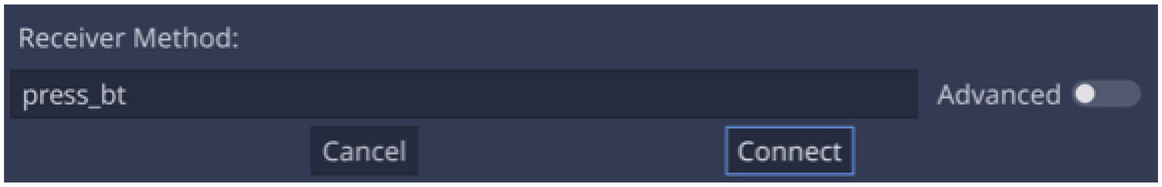

Edit Connection:pressed

From Signal: pressed

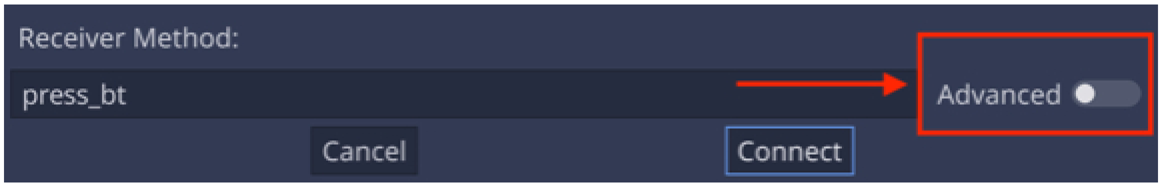
Connect to Script:

- main
  - red_bt (Connecting From)
  - green_bt
  - blue_bt
  - yellow_bt

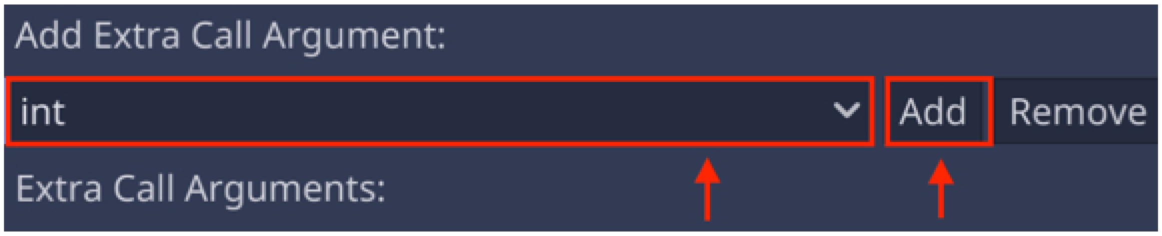
- In the field called **Received Method**, please type **press_bt** (this is the name of the function in the script linked to the node **main**, that we want to call, in case the player presses this button).


Receiver Method: press_bt

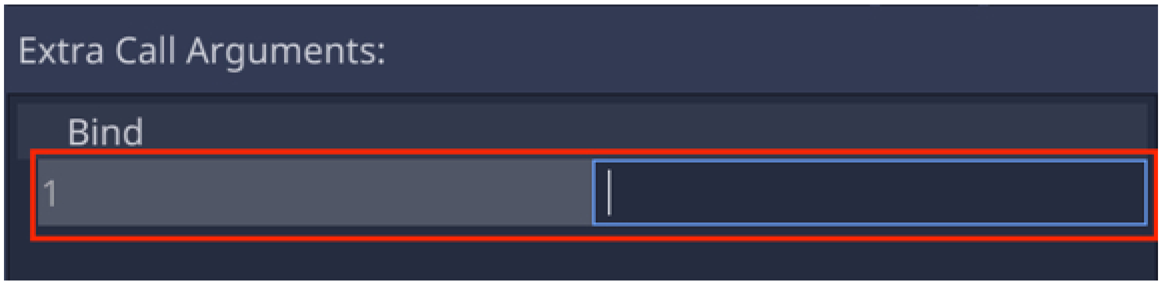
- Once this is done, please click on the button labelled **Advanced** (to the right of the text field)



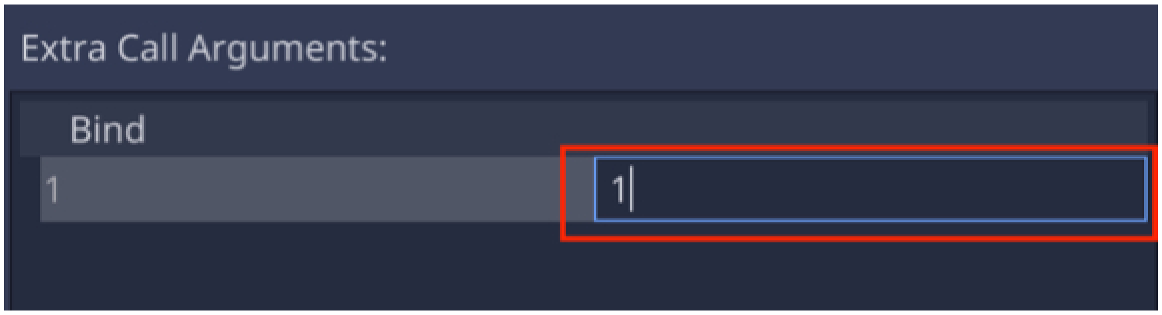
- In the new window, please select the **int** type from the drop-down menu, and press **Add**, as per the next figure.



- Once this is done, a new line will appear in the section called **Extra Call Arguments**.



- Please enter **1** in the right field; this means that the parameter passed to the function (i.e., **press_bt**) is **1**. This will be a way to keep track of the button that was pressed; for example, for the second button, we will use the same mechanism, and include **2** as a the parameter passed to the function.



- Once you have entered this value, please press the button labelled **Connect**.
- Please repeat the previous steps to connect the buttons **green_bt**, **blue_bt**, and **yellow_bt**, ensuring that the parameter passed to the function is **2**, **3**, and **4**, respectively.

Now that all buttons are connected, please save and play your scene. As you press the buttons, you should see the corresponding messages in the Output window; for example "**Button Pressed is 1**".

CREATING DIFFERENT STATES FOR OUR GAME

At this stage, we have four buttons that the player can press, and we can also detect which button was pressed. The next phase is to start to manage the different phases of the game; generally for this game:

- A new sequence of color is displayed.
- The player is asked to reproduce this sequence by pressing on the corresponding buttons.
- The player's entries are assessed to check whether they correspond to the initial sequence.

To start, let's declare the different states that will apply to our game.

- Please open the script called **simon_main.gd**.
- Add the following code at the beginning of the script (e.g., just after the first line).

```
const STATE_PLAY_SEQUENCE = 1
const STATE_WAIT_FOR_USER_INPUT = 2
const STATE_PROCESS_USER_INPUT = 3
```

In the previous code:

- We declare three constants; these are variables that will not change throughout our game. Although they are integers, their name will be easier to remember than numbers.
- All these variables are there to define a state in which the game will be at any given time.

So the game will be in one of these states:

- **Playing a sequence of colors**: once a new color has been created, the new sequence that includes all the colors generated is played. Every turn, one new color is added to the sequence.
- **Waiting for the user's input**: After a new sequence has been played/displayed by the game, the player needs to click on the boxes in the correct order to reproduce this sequence.
- **Processing the user's input**: Once the user has pressed the different buttons, the game will need to process this information to determine whether the player has remembered and played the previous sequence correctly.
- Once this is done, the game will either add a new color to the sequence, or restart from the beginning.

The core of our game will be based on all these states; so we will now modify the **_process** function to mirror these different states.

- Please add the following code at the beginning of the script:

```
var current_state
```

- Please add the following code in the **_ready** function:

```
current_state = STATE_PLAY_SEQUENCE
```

In the previous code, we just specify that at the start of the game, the game will play the new sequence.

- Please add the **_process** function as follows:

```
func _process(delta):
match (current_state):
STATE_PLAY_SEQUENCE:
```

```
pass
STATE_WAIT_FOR_USER_INPUT:
pass
STATE_PROCESS_USER_INPUT:
pass
_:
pass
```

In the previous code:

- We create a match statement; this is comparable to branching instructions based on a specific variable; in our case, we will execute code based on the value of the variable called **current_state**.
- So if the value of the variable **current_state** is **STATE_PLAY_SEQUENCE**, we will go to the corresponding section called **STATE_PLAY_SEQUENCE**.
- The idea of this structure is that all branches are mutually exclusive which means that you can be in only one state at any given time. So the game will either be playing a new sequence of colors, waiting for the user's input, or assessing the user's inputs.

PLAYING A SEQUENCE OF COLORS

Next, we will be dealing with the state called **STATE_PLAY_SEQUENCE**; in this state, the game should do the following:

- Pick a color at random.
- Add it to the existing sequence of colors.
- Play the new sequence of colors (including the last color) and light up the corresponding boxes sequentially.
- Note that if we are at the start of the game, only the new color that has been generated will be displayed (i.e., the corresponding boxed will be lit up).

For this purpose, we will need to do the following:

- Generate a new color at random.
- Store the sequence of colors in an array.
- Hide all the boxes currently displayed on screen.
- Display (and subsequently hide) each individual box that is part of this sequence.

First, let's create the code that will generate a random number.

- Please add the following code at the beginning of the script (i.e., after the first line):

```
var sequence_of_color = []
var index_of_color = 0
```

In the previous code, we define the array that will be used to store the sequence created by the game.

- Please add the following code to the **_ready** function:

```
current_state = STATE_PLAY_SEQUENCE
for i in range(0,100):
sequence_of_color.append(0)
```

In the previous code, in addition to specifying the current state, we initialize the array **sequence_of_color**; as it is, it should handle up to 100 items.

- Please add the following code just before the end of the script:

```
func generate_new_color():
var rnd = RandomNumberGenerator.new()
rnd.randomize()
var r = rnd.randi_range(1, 4);
sequence_of_color [int(index_of_color)] = r
index_of_color += 1
```

In the previous code:

- We declare a function called **generate_color**.
- We create a new random number with a value that can range between **1** and **4**.
- We then add this number to the array that stores the sequence of colors generated by the game.
- We then increment the index of the array **index_of_color**. This is so that the next time a color is created, it will be added at the index (or position) **index** in the array.

Next, we will create a function that will hide all the boxes; this is because we want to clear the screen before displaying each of the boxes that are part of the sequence.

- Please add the following code at the beginning of the script.

```
onready var red_bt = get_node("red_bt")
onready var green_bt = get_node("green_bt")
onready var blue_bt = get_node("blue_bt")
onready var yellow_bt = get_node("yellow_bt")
var array_of_bts = []
```

In the previous code, we declare four nodes **red_bt**, **green_bt**, **blue_bt**, and **yellow_bt** that refer to the buttons that we have created earlier. We also declare an array called **array_of_bts**.

- Add the following code to the **_ready** function (new code in bold).

```
for i in range(0,100):
sequence_of_color.append(0)
array_of_bts = [red_bt, green_bt, blue_bt, yellow_bt]
```

In the previous code, we initialize the array with the name of each of the buttons on the interface.

- Please add the following function at the end of the script.

```
func hide_boxes():
for a_button in array_of_bts:
a_button.hide()
```

In the previous code, we loop through the four boxes that we have created earlier (i.e., the boxes that are on screen) and hide them.

Next, we will need another function that displays a specific box; this is so that we can display individually each of the colors included in the sequence generated by the game.

- Please add the following function at the end of the script.

```
func display_box(index_of_box):
for a_button in array_of_bts:
if (a_button.is_in_group(str(index_of_box))):
a_button.show()
```

In the previous code:

- We declare a new function called **display_box**, that will be used to display a particular box.
- We loop through the four boxes and detect the box that belongs to the group defined by the value of the variable **index_of_box**.
- In that case, we just display that box.

Before we can test our script, we just need to set a group for each of the buttons on the interface.

- Please select the node **red_bt**, select the tab **Node | Groups**, create a group called **1** and add it to that node.
- Please repeat the previous step so that the buttons **green_bt**, **blue_bt** and **yellow_bt** are associated with the groups **2**, **3**, and **4** respectively.
- Please save your script. At this stage, you can test the functions **hide_boxes** and **display_box** by adding the following code to the **_ready** function.

```
hide_boxes ();
display_box (1);
```

As you play the scene, you should see that only the red box is displayed.

CREATING A NEW SEQUENCE OF COLORS

Now that we can hide and display boxes, we will get to do the following:

- Pick a color/box at random.
- Add it to a new sequence.
- Display the sequence.
- Make sure that there is a delay between the display of each box, so that the sequence can be identified clearly by the player; otherwise, the sequence would be displayed too fast to be remembered.
- Wait for the player to reproduce this sequence by clicking on the corresponding boxes.

First, we will create a timer to display each of the colors that are part of the sequence generated by the game.

- Please add the following code to the beginning of the script:

```
var new_color_has_been_generated
var all_boxes_displayed
var start_timer;var waiting_timer_activated
var waiting_time_elapsed; var timer
var waiting_time; var time_delay_between_displays
var animation_index; var nb_colors_submitted
```

In the previous code, we declare a set of variables that will be used for the state called **STATE_PLAY_SEQUENCE**; each of these variables will be explained in the next sections.

- Please add the following code to the **_ready** function:

```
new_color_has_been_generated = false;
```

```
start_timer = false;
waiting_timer_activated = false;
waiting_time_elapsed = false;
all_boxes_displayed = false;
timer = 0;
waiting_time = 0;
animation_index = 0;
time_delay_between_displays  = 2;
nb_colors_submitted = 0;
```

In the previous code:

- We set the variable **new_color_has_been_generated** to **false**.
- This variable is used to know whether the game has already selected a new color at random.
- We also declare a series of variables that will be explained in detail later.

Once this is done, we can look at what the game should do in the state called **STATE_PLAY_SEQUENCE**.

- Please add the following code to the **match** section that corresponds to the state **STATE_PLAY_SEQUENCE** (new cod in bold).

```
STATE_PLAY_SEQUENCE:
if (!new_color_has_been_generated):
init_timer ();
generate_new_color ();
new_color_has_been_generated = true;
all_boxes_displayed = false;
hide_boxes ();
```

In the previous code:

- We are in the state called **STATE_PLAY_SEQUENCE**, a state where the game is supposed to pick a new colored box at random, add this color to the existing sequence, and then play the full sequence (including the last color picked).
- If a new random color has not yet been picked, we proceed as follows: we call a function called **init_timer** that we yet need to define; this function will basically initialize the timer that will be used to make sure that there is enough time or delay between the display of each box in the sequence.
- We call the function **generate_new_color**; if you remember well, this function picks a color at random and adds it to a sequence.
- Next, we specify that we have picked a new color by setting the variable **new_-col or_has_been_ generated** to **true**.
- Finally, we hide all the boxes and set the corresponding value of the variable **all_boxes_displayed** to **false**.

So we can now add the function **init_timer** at the end of the script as follows:

```
func init_timer():
start_timer = true
timer = 0;
animation_index = 0
```

In the previous code:

- We set the variable **start_timer** to true, so that the timer can start ticking.
- Its initial value is **0**.
- The animation index is set to **0**; when we successively display a new sequence to the player, we effectively create an animation that includes several

stages depending on the box in the sequence that is currently displayed as part of the animation. So we will start with the first element of the animation (i.e., the first box in the sequence).

We can now resume the structure of the state **STATE_PLAY_SEQUENCE**.

- Please add the following to the code that you have already included in the **match** portion that corresponds to the state **STATE_PLAY_SEQUENCE,** but not within the same if statement (new code in bold).

```
timer += delta;
if (start_timer && timer >= time_delay_between_displays && !waiting_timer_activated) :
timer = 0
hide_boxes ()
display_box (sequence_of_color [animation_index])
animation_index += 1
if (animation_index >= index_of_color) :
start_timer = false
animation_index = 0
waiting_timer_activated = true
```

In the previous code:

- We increase the value of the variable **timer** by one every second.
- This timer is used to determine when we should display the next part of the animation.
- This is the case when the timer has started (i.e., when **start_timer** is **true**), and when we have reached the threshold called **time_delay_between_displays** (i.e., **timer >= time_delay_between_displays**).

- We also use a variable called **waiting_timer_activated**; this is employed to add a slight delay at the end of the animation; without this delay, the last color would be displayed too briefly; so we create a waiting timer that will be used once we have reached the end of the sequence; this is why, in this par-tic ular case, we also require **waiting_timer_activated** to be false.
- When these three conditions are fulfilled, we do perform the next steps.
- The timer is reset to 0.
- We hide all the boxes.
- We display the box at the current index (i.e., **animation_index**) in the se-quence.
- We increase the value of the variable **animation_index** by one.
- If we have reached the end of the animation (i.e., **animation_index >= index_of_color**), we reset the timer, as well as the **animation_index**, and we also specify that we will add a slight delay at the end of the sequence by acti-vating the waiting timer (i.e., **waiting_timer_activated = true**).
- Please add the following to the code that you have already included in the match portion that corresponds to the state **STATE_PLAY_SEQUENCE** (if en-sure about the indenting, please check the solution code). The start of this if statement should be indented once from the code **STATE_PLAY_SEQUENCE:**.

```
if (waiting_timer_activated):
waiting_time += delta
if (waiting_time >= time_delay_between_displays):
waiting_time = 0;
waiting_timer_activated = false;
waiting_time_elapsed = true;
```

In the previous code:

- This code is used when the waiting timer is activated at the end of the sequence.
- Once it is activated, we just increase its time every second.
- Whenever it has reached a specific threshold (i.e., **waiting_time >= time_delay_between_displays**) we then reset this timer.
- Then, the variable **waiting_time_activated** is set to false (since the delay has already been applied).

Please add the following code just after the code that you have typed (i.e., within the same if statement), at the same level of indentation as the code that starts with **if (waiting_timer_activated):**.

```
if (waiting_time_elapsed):
current_state = STATE_WAIT_FOR_USER_INPUT;
nb_colors_submitted = 0
```

In the previous code: if we have reached the end of the animation and applied the slight delay at the end (i.e., **waiting_time_elapsed = true**), then we move on to the next state where the user needs to repeat this sequence.

We can also add the code to re-initialize all the variables used in the animation in the **_ready** function, as follows.

```
index_of_color = 0;
start_timer = false;
waiting_timer_activated = false;
waiting_time_elapsed = false;
all_boxes_displayed = false;
timer = 0;
waiting_time = 0;
animation_index = 0;
time_delay_between_displays  = .5;
```

```
new_color_has_been_generated = false;
for i in range(0,100):
sequence_of_color.append(0)
sequence_of_color [0] = 1
sequence_of_color [1] = 2
sequence_of_color [2] = 3
sequence_of_color [3] = 4
sequence_of_color [4] = 2
index_of_color = 4;
```

The last part of the code creates a virtual sequence of colors and sets the variable **index_of_color** to **4** (remember: for an array, the first element starts at the index 0, so here the index of the last element in this array is not 5 but **4**), indicating that the last element of the sequence (or array) is at the index **4**.

You can now save your script and play the scene, you should see that all the boxes disappear and that a sequence of **5** random boxes is displayed on screen.

WAITING FOR THE USER'S INPUT

So, at this stage we are able to generate a random color and to play a sequence; so the next step will be to wait for the user's input; so, in this section, we will write code that will do the following:

- Wait for the user's input.
- Let the player select (i.e., click on) a sequence of boxes.
- Record the boxes that were selected by the player.
- Record the number of boxes selected.
- When the expected number of boxes has been reached (e.g., if the sequence includes two colors, we expect the player to choose two boxes), we will then check if the correct sequence was entered by the player by comparing it to the sequence previously generated by the game.
- In case the sequence entered by the player is correct, the game will restart the process by generating a new color, and add it to the previous sequence.
- In case the sequence entered by the player is incorrect, the game will restart with just one color initially.

First, we will record the sequence entered by the player:

- Please add the following code at the beginning of the script **simon_main.gd**

```
var sequence_of_colors_submitted = []
var color_submitted
```

- Please add the following code to the **_ready** function.

```
for i in range(0,100):
sequence_of_colors_submitted.append(0)
```

In the previous code, we declare and initialize an array that will be used to store the sequence of colors chosen by the player.

- Please add the function **submit_color** as follows.

```
func submit_color(name_of_color):
print("You have pressed color:" + str(name_of_color));
if (current_state == STATE_WAIT_FOR_USER_INPUT):
color_submitted = int(name_of_color);
sequence_of_colors_submitted [nb_colors_submitted] = name_of_color;
nb_colors_submitted += 1;
if (nb_colors_submitted == (index_of_color)) :
current_state = STATE_PROCESS_USER_INPUT;
```

In the previous code.

- We record the color (or the corresponding group) of the box that was just selected by the player.
- We increase the index of the variable called **nb_colors_submitted**, so that we know how many boxes the player has selected.
- Then, if the player has selected the required number of boxes, we then proceed to the state called **STATE_PROCESS_USER_INPUT**.

We can then make a few more changes.

- Please, add the following code to the **_process** function (new code in bold):

```
STATE_WAIT_FOR_USER_INPUT:
if (!all_boxes_displayed):
set_box_colors ()
all_boxes_displayed = true
```

In the previous code:

- We check whether all the boxes are displayed.
- This is because, in the previous state, the colors are displayed one by one; however, now the player needs to be able to choose the correct sequence; therefore, all boxes should now be displayed, so that the player can click on them.
- We call a function called **set_box_colors** which we yet have to define, that will display all the boxes.
- Please add the following code to the class:

```
func set_box_colors():
for a_button in array_of_bts:
a_button.show()
```

In the previous code, we display all the boxes.

PROCESSING THE USER'S INPUT

In this section, we will process the user's input.

- Please modify the **_process** function as follows (new code in bold) in the script **simon_main**:

```
STATE_PROCESS_USER_INPUT:
var ok_result = assess_user_move ()
if (ok_result):
animation_index = 0;
new_color_has_been_generated = false
timer = 0
waiting_timer_activated = false
waiting_time_elapsed = false
current_state = STATE_PLAY_SEQUENCE
else:
load_lose_level ()
```

In the previous code:

- We are in the state where the game is processing the user's inputs.
- We create a Boolean variable that will store the result returned by the function **assess_user_move**; this function, that we yet have to define, will return **true** if the user has entered the correct sequence, and **false** otherwise.
- So if the player has entered the correct sequence (i.e., if **ok_result** is **true**), then the variable **animation_index** is set to **0** (so that the animation can start from the first element of the next sequence); we also re-initialize several of the variables linked to the state **STATE_PLAY_SEQUENCE**, including **new_color_has_been_generated**, **timer**, **waiting_timer_activated**,

waiting_time_elapsed, and **current_state**.

- If the sequence is incorrect, we call the function **load_lose_level**, that we yet have to define, which loads a scene with a message that indicates that the player has lost.

So next, we can start to create the function **assess_user_move**, so that we can check if the player has entered the correct sequence.

- Please add the following function at the end of the script:

```
func assess_user_move():
var all_perfect = true;
for i in range (0, index_of_color):
var a = sequence_of_color [i]
var b = sequence_of_colors_submitted [i]
print("Comparing "+str(a)+" and "+str(b))
if (int(a) != int(b)):
all_perfect = false
print("Not a match")
if (all_perfect):
print ("WELL DONE!")
return true
else:
print ("NOT RIGHT!")
return false
```

In the previous code:

- We define a Boolean variable called **all_perfect** and set it to **true**. This variable will be used to determine if all the colors selected by the player were entered in the right sequence; if any error was made, this variable (i.e., **all_perfect**) will be set to **false**.

- We then loop through the colors entered by the player; the variable **in-dex_of_ color** indicates the number of colors entered by the player; so we go from the first to the last color chosen by the player. The sequence of colors entered by the player is stored in the variable **sequence_of_colors_submitted** and the correct sequence of colors is submitted in the array **sequence_of_color**; so we compare these two arrays at the current index (e.g.., for the first or the second color chosen).
- If the colors are different at the current index, then the variable **all_perfect** is set to false; so at the end of the loop, if one of the colors amongst the sequence chosen by the player is incorrect, the variable **all_perfect** is set to **false** and the value **false** is returned by the function.
- Otherwise, the value **true** is returned.

Last but not least, we will create the function **load_lose_level** that will be called when the player has entered the incorrect sequence along with the corresponding scene.

- Please add the following code at the end of the script.

```
func load_lose_level():
SimonSingleton.player_score = index_of_color - 1
get_tree().change_scene("res://end_simon_game.tscn")
print("You just lost")
```

In the previous code:

- We define the function **load_lose_level**.
- We refer to a script called **SimonSingleton**; this script, that we have not

created yet, will be created and set up so that it is a singleton, and as such, it will be accessible throughout the game, and its variables will be persistent; in other words, we will be saving the score in this script, and the score will be kept regardless of whether we change scene.

- In this function, we save the score which is basically the length of the sequence that the player has managed to reproduce. So, for example, if the player has managed to remember a sequence of two colors but has failed to remember (and to reproduce) the next sequence, then the score will be **2**. Because **index_of_colors** is incremented just before we start a new sequence, we need to subtract one from this number to calculate the correct score if the current sequence was not reproduced correctly.
- We then load the scene called **end_simon_game** (that we yet have to create).

So, before we go any further, let's create and set up the script **SimonSingleton**.

- Please open the **Script** workspace, by clicking on the **Script** icon located at the top of the window.
- Then create a new script: Select **File | New Script**.
- Name this script **simon_singleton.gd**.
- Open this script.
- Add this code at the beginning of the script (new code in bold).

```
extends Node
var player_score
```

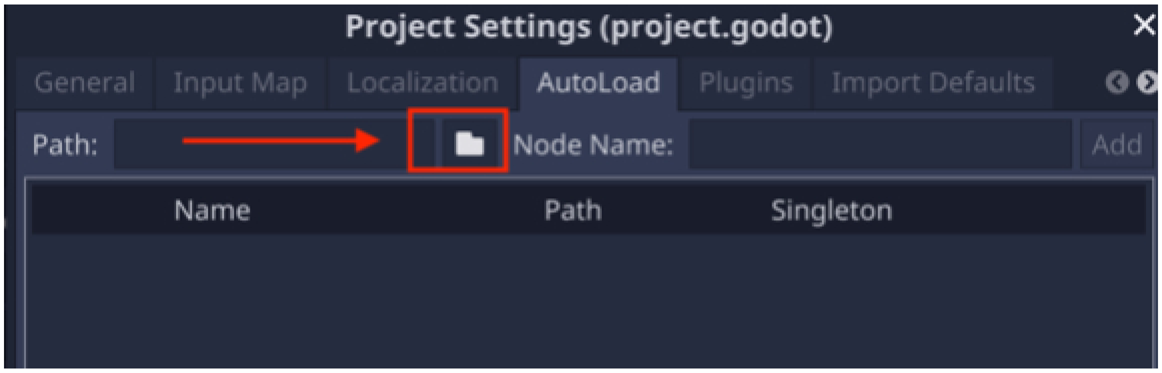
In the previous code, we declare the variable **player_score** that will be used to save the player's score across scenes.

Now, we just need to declare this script as a singleton.

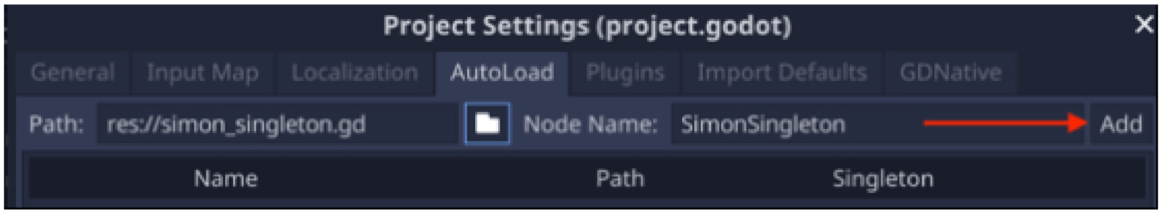
- Please open the **AutoLoad** project settings: Select **Project | Project Settings |**

AutoLoad.

- In the new window, click on the white folder to the right of the label "**path**".



- In the new window, select the file **simon_singleton.gd** that you have created earlier, and press the button labeled "**Open**".
- In the next window, press the button labeled "**Add**".
- This will create a new line with a path to the script that we have just added as a **singleton**.



Path: res://simon_singleton.gd
Node Name: SimonSingleton

- Please make sure that the option "**Enabled**" is selected.
- You can now press the button "**Close**" to close this window.

Now that this is done, and before we check for incorrect answers from the player, we can just check if the game works for correct inputs.

- Please comment the following code in the script **simon_main.gd** :

```
# sequence_of_color [0] = 1
```

# sequence_of_color [1] = 2

# sequence_of_color [2] = 3

# sequence_of_color [3] = 4

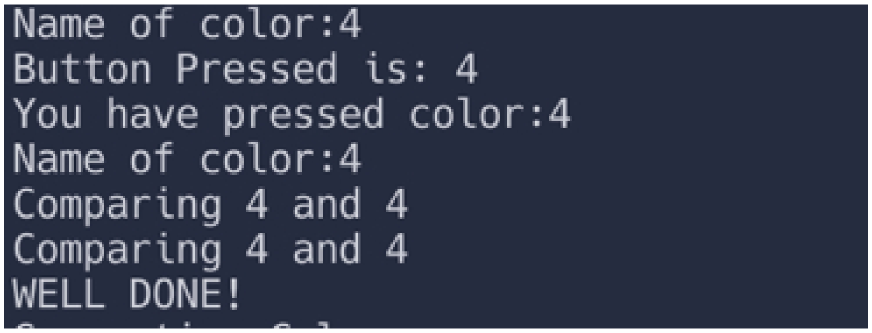
# sequence_of_color [4] = 2

# index_of_color = 4

- Add the following code to the function **press_bt**:

```
func press_bt(name_of_bt):
print("Button Pressed is: "+str(name_of_bt))
submit_color(name_of_bt)
```

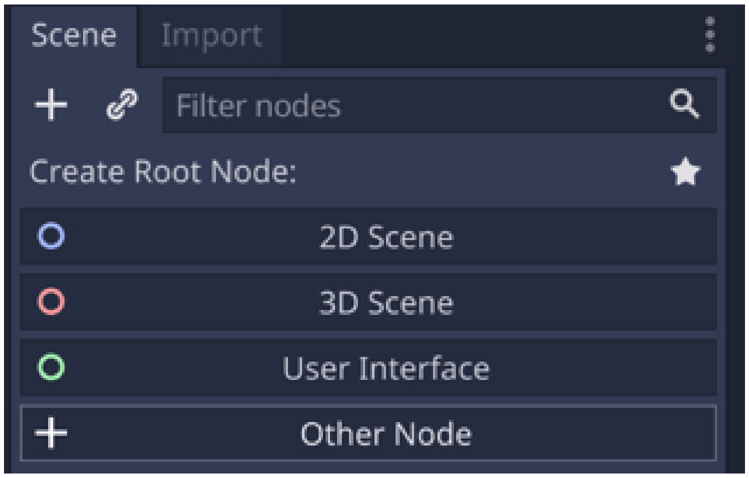
- Please save your script and check that it is error-free
- You can play the scene.
- The game will display one color, then wait for your input. As you click on the correct color, it should now display two colors, and wait for your input again. You should also see the following (or similar) messages in the **Output** window.



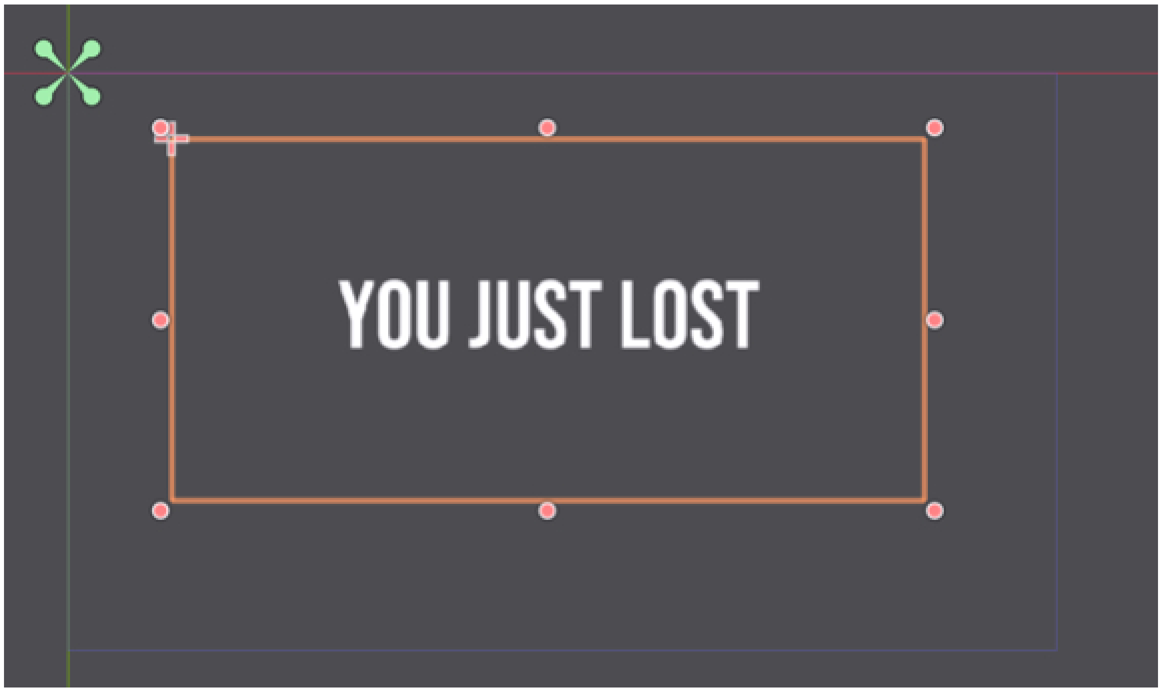
```
Name of color:4
Button Pressed is: 4
You have pressed color:4
Name of color:4
Comparing 4 and 4
Comparing 4 and 4
WELL DONE!
```

Now that you have checked that the game plays correctly, we can now create the scene called **end_simon_game.tscn**, so that the game also accounts for incorrect inputs; the new scene will include a **Label** node that will be used to display the score as well as a button that the player will be able to click to restart the game.

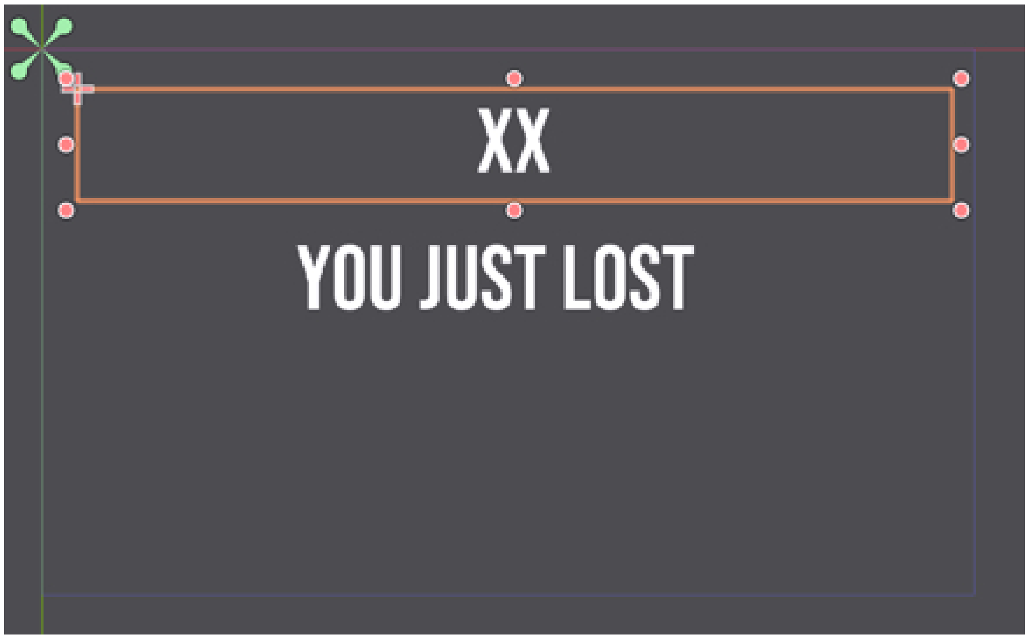
- Please save the current scene.
- Create a new scene (**Scene | New Scene**).
- In the new window select the option **Other Node**.



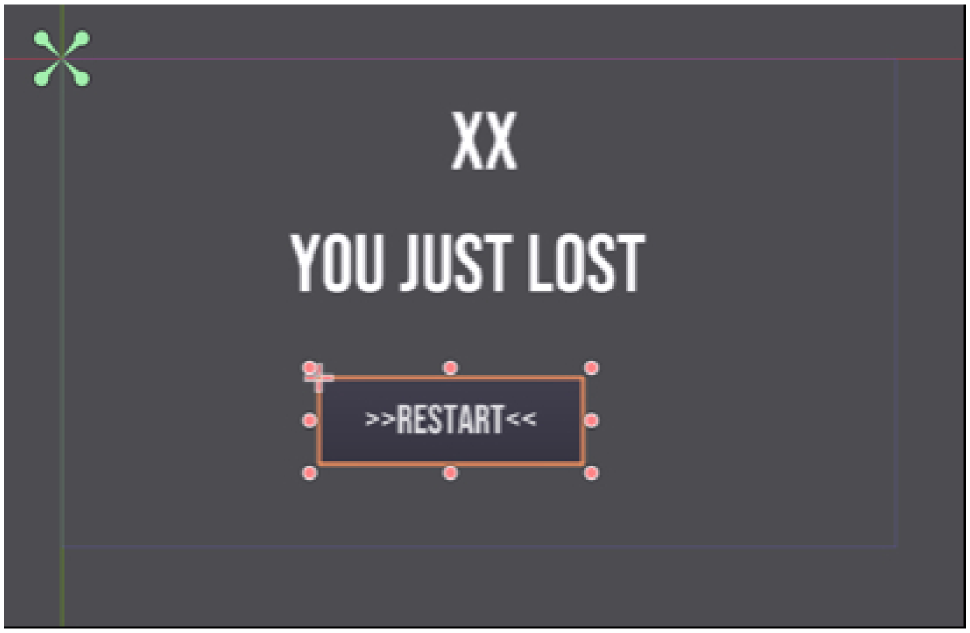
- In the new window, select the node type **Control**.
- This will create a new scene with a default node called **Control**.
- Add a child of type **Label** to this node, and set it up so that it displays the message “**YOU JUST LOST**” in the middle of the screen.



- Add a new child of type **Label** to the node **Control b**or duplicate the current Label node), and rename the new node **score_ui**.
- Set it up so that it features text that is displayed at the top of the screen, as per the next figure.



- Add a new child node of type **Button** to the node **Control**, and set up this button so that it appears at the bottom of the screen and displays the label ">> **Restart** <<", as per the next figure



When this is done, we just need to create a script that we will link to this object and that will display the score through the node called **score_ui**.

- Please add new script to the node **score_ui** and call it **score_ui.gd**.
- Modify the **_ready** function as follows (new code in bold).

```
func _ready():
text = "Score: " + str(SimonSingleton.player_score)
```

In the previous code, we display the score using the object **score_ui**

Please save your scene as **end_simon_game.tscn**, as well as your script. We just need to detect when the player presses the **Restart** button and to reload the main scene accordingly when this happens.

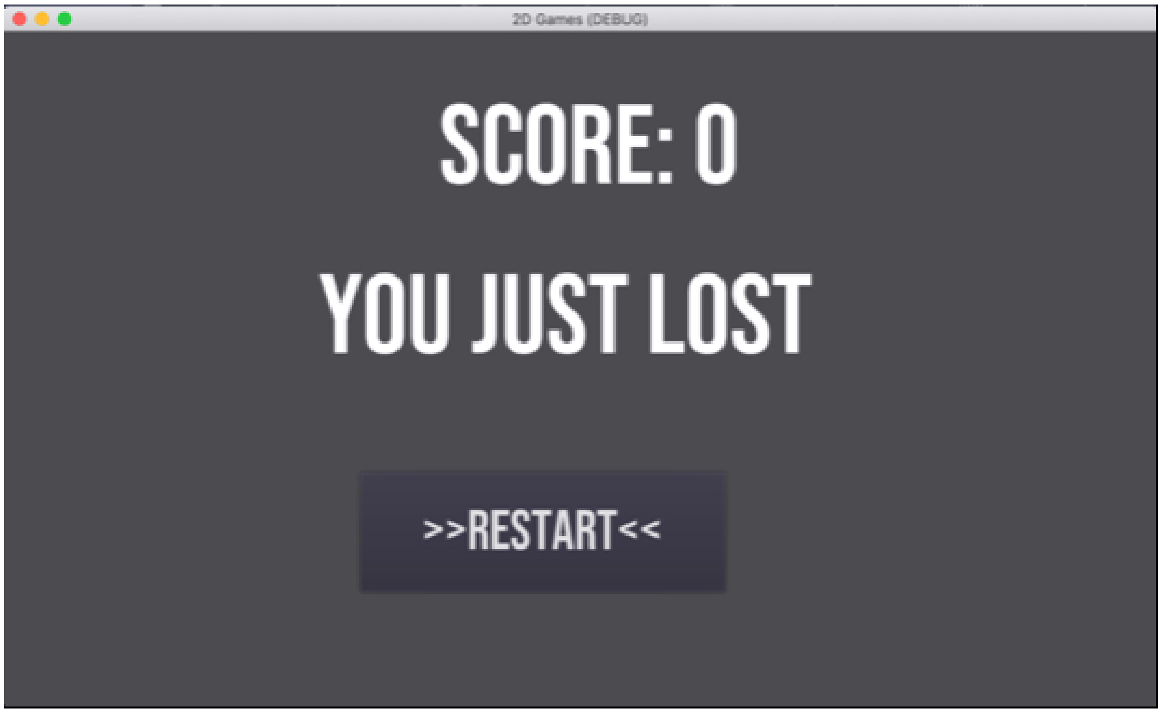
- Please attach a new script to the button that you have created previously and call this script **restart_bt.gd**.
- Add the following function to it.

```
func press_bt_restart():
get_tree().change_scene("res://simon_game.tscn")
```

- Using the **Scene Tree** view, click on the button.
- Select the tab **Node | Signals** and double click on the "**pressed**" event.
- In the new window, select the node **Button** from the list, and enter the text **press_bt_restart** in the field **Receiver Method**.

You can now save the current scene, and re-open and play the main scene.

As you play the main scene, and enter an incorrect sequence, a new scene should appear with your score, and a button to restart the game, as illustrated in the next figure.



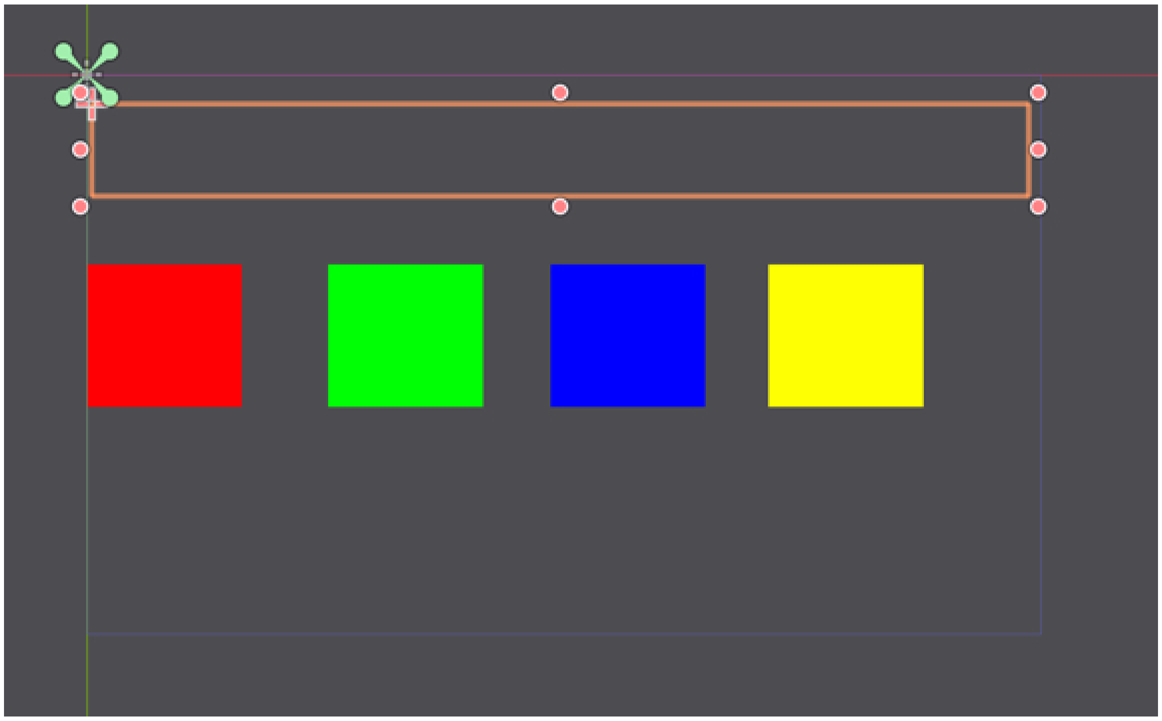
GENERATING SOUND EFFECTS

Ok, so the game works well at present; however, so that the players can obtain additional information on their move, we could add a few more features to our game, including:

- A text field that displays the current score (or the number of colors that the player has managed to memorize so far) during the game.
- A specific sound for each of the colors, so that the player associates (and remembers) the colors associated with a sound.

So, first, let's create the on screen text for the score:

- Please open the main scene.
- Add a new child of type **Label** to the node **main**, and rename the new node **nb_memorized**.
- Place this node at the top of the screen and set up the font type and size (e.g., **50**).



Once this is done, we will create a function that displays the corresponding information in this text field.

- Please open the script **simon_main.gd**.
- Add this code at the beginning of the script (i.e., before the **_ready** function).

```
onready var nb_memorized = get_node("nb_memorized")
```

- Add the following function to this script.

```
func update_ui():
nb_memorized.text = str(index_of_color)
```

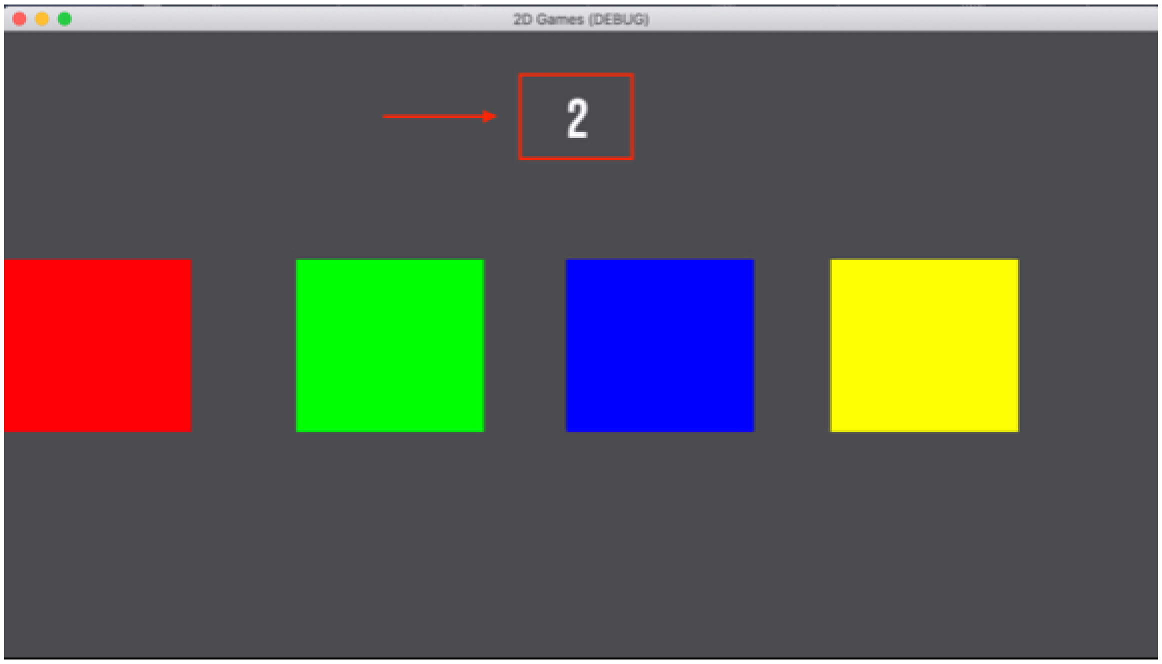
- Please add the following code at the end of the **_ready** function.

```
update_ui();
```

- Add the following code in the function **assess_user_move** (new code in bold).

```
if (all_perfect):
update_ui()
print ("WELL DONE!")
return true;
else:
```

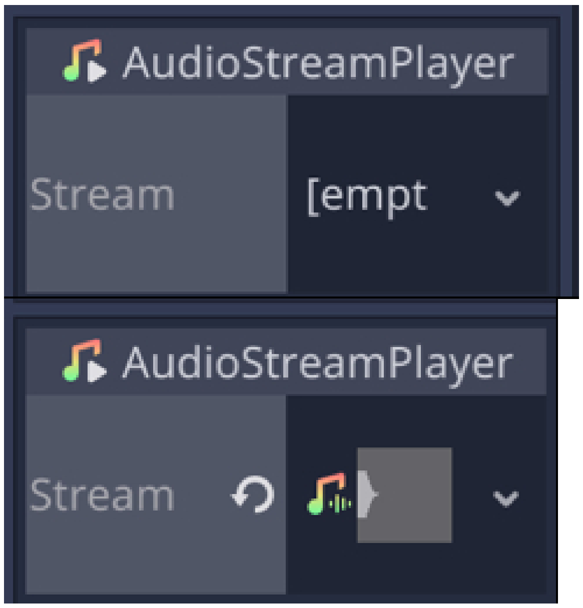
- In the previous code, we update the user interface when the user has guessed the sequence correctly.
- You can now save your script, play the scene, and check that your score is displayed at the top of the screen.



Once this is done, we can start to add some audio. For this, we will create a note for which the pitch will be linked to the label of the color being displayed, or the color chosen by the player. For this purpose, we will first create an

AudioStreamPlayer node, and then use it to generate a sound effect at run-time. So, let's get started:

- Please open (switch to) the scene **simon_game.tscn**.
- Add a child node of type **AudioStreamPlayer** to the node **main**, and change the name of the new node to **audio_player**.
- Drag and drop the file **beep.wav** from the **File System** window to the empty slot in the section **AudioStreamPlayer | Stream** in the **Inspector**, as per the next figure.



Once this is done, we can start to modify our script to be able to generate different notes.

- Please open (or switch to) the script **simon_main.gd**.
- Add the following code at the beginning of the script.

```
onready var audio_stream_player = get_node("audio_player")
```

- Add the following function to it.

```
func play_note(note_to_play):
audio_stream_player.pitch_scale = int(note_to_play)
audio_stream_player.play()
```

In the previous code:

- We define a new note based on the button pressed.
- The pitch of the sound is based on this button.
- We then play the sound at the pitch defined earlier.

Now that we have defined the function **play_note**, we can then call it whenever a box is displayed or when a box has been selected by the player.

- Please add the following code at the beginning of the function **submit_color**.

```
play_note (name_of_color)
```

- Please add this code at the end of the function **display_box**.

```
play_note(index_of_box)
```

- You can now save your script and test the scene; you will notice that a different sound is played every time a color is displayed or selected by the player.

The last thing we need to do now is to be able to check whether the player is selecting the right color: at present we have to wait for the full sequence to be entered before completing the test. So, what we could do instead is to perform this check every time the player selects a box (instead of waiting for the player to enter a full sequence).

For this purpose, we will create a new function called **assess_user_current_move** and call it whenever the user has selected a box.

- Please open (or switch to) the script **simon_main.gd**.
- Add the following function before the end of the script.

```
func assess_user_current_move():
if (int(sequence_of_colors_submitted [nb_colors_submitted-1]) == int(sequence_of_color[nb_colors_submitted-1])):
return true
else:
return false
```

In the previous code:

- We define the function **assess_user_current_move**.
- We check if the color chosen by the player matches the corresponding color in the sequence created by the game.

Next, we will just need to modify the function **submit_color** as follows (new code in bold):

```
if (current_state == STATE_WAIT_FOR_USER_INPUT):
color_submitted = int(name_of_color);
sequence_of_colors_submitted [nb_colors_submitted] = name_of_color;
nb_colors_submitted += 1;
var  right_move = assess_user_current_move()
if (!right_move): load_lose_level()
if (nb_colors_submitted == (index_of_color)) :
current_state = STATE_PROCESS_USER_INPUT;
```

In the previous code, as we have done earlier, we check if the player’s move is correct (i.e., right color selected) and then call the function **load_lose_level** otherwise.

You can test your scene and check that after an incorrect move, the game will display the scene called **lose**, without waiting for you to enter the complete sequence.

That's it!

LEVEL ROUNDUP

In this chapter, we have learned about creating a relatively simple memory game with interesting features such as: displaying or hiding objects, playing sounds and generating frequencies for this sound, creating a finite-state machine, arrays to store a sequence, and a timer to play this sequence. So, we have covered some significant ground compared to the last chapter.

Checklist

You can consider moving to the next chapter if you can do th

- Create a singleton.
- Use match statements to create different states for your game.
- Display or hide objects from your script.

Quiz

It's now time to check your knowledge with a quiz. So please try to answer the following questions (or specify whether the statements are TRUE or FALSE). The solutions are on the next page.

1. In GDScript, when using **match** statements, you need to add a **break** statements for each case.
2. The following code will create a constant variable.

```
const STATE_PLAY_SEQUENCE = 1
```

1. The following code will hide a node.

```
my_node.hide()
```

1. Provided that **audio_stream** is an **AudioStreamPlayer** node for which an audio file has been allocated, the following code will play this audio file.

```
audio_stream_player.play()
```

1. The following code, if added within the function **_process** will add one to the variable **timer** every seconds.

```
timer += delta
```

1. The following code will create a new array.

```
var sequence_of_color = []
```

1. When a variable is declared and saved in a singleton script, it will be available throughout the game.
2. The following variable will be accessible from the **Inspector**.

```
export var my_variable:int
```

1. When connecting a button to a function based on an event through Godot's user interface, it is possible to specify what value is passed to this function in that case.
2. The following code will initialize the array called **sequence_of_color**:

```
for i in range(0,100):
sequence_of_color.append(0)
```

Answers to the quiz.

1. FALSE (this is not necessary when using GDScript).
2. TRUE.
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. TRUE.
8. TRUE.
9. TRUE.
10. TRUE:

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let's do the following.

- Modify the texture of each box so that, instead of displaying colored boxes, the game displays an image; this image needs to be different for each box.
- Add a splash-screen where the player can press on a start button to start the game.

Chapter 6: Creating a Platformer

In this chapter, we will be creating a platformer game that includes most of the features found in common platformers including:

- An animated 2D character that will be able to jump and walk.
- Simple platforms.
- A camera that follows the player.
- Objects that you can collect.
- Ladders that the player character can climb.
- Magic doors that will teleport the player character to specific areas.
- A 2D environment that gives the illusion of depth using parallax.
- A polished 2D environment that uses tiles.
- Dead zone areas that detect when the player has fallen off a platform.

So, after completing this chapter, you will be able to:

- Create an animated 2D character from a sprite sheet.
- Control this character, and include gravity.
- Use areas to detect collision with objects.
- Configure and use a tilemap.
- Use **Sprite** and **ParallaxLayer** nodes.
- Adjust the sped of each layer comprised in the parallax effect.
- Destroy objects upon collision.

Introduction

In this chapter we will create a simple level with a 2D character following this:

- Create an animated character.
- Control the animated character.
- Add simple platforms on which the character can move and jump.
- Add objects to collect.
- Add sound effects.
- Define a dead zone that detects when the player has fallen.
- Add ladders that the player character can climb.
- Add magic doors.
- Add a tile map.
- Add a parallax effect.

Creating the main character

The very first thing that we will do is to create an animated 2D character that the player will be able to control. This character will be able to walk left, and right, and to jump also. The process will be as follows:

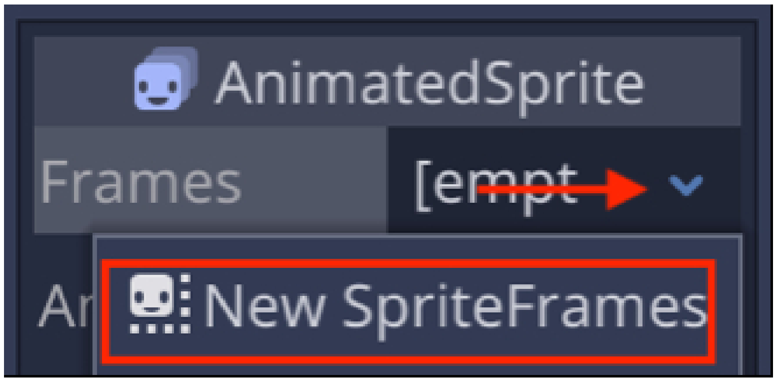
- Import the images that will make up the animations for the character.
- Create animations from these files (e.g., walking left, walking right, being idle, or jumping).
- Detect the player's keyboard inputs, move the character accordingly (e.g., walk, or jump) and apply the corresponding animation to the player character.

So let's get started:

- Please create a new scene, if the previous project is already open (**Scene | New Scene**).
- In the new window, please select the option **Other**.
- In the next window, select the node type **KinematicBody2D**.
- This will create a new scene with a default node called **KinematicBody2D**.
- Please rename this node **player**.
- Add a child node of type **AnimatedSprite** to the node **player**.
- Select the node labelled **AnimatedSprite**.
- Using the **Inspector**, scroll down to the section called **Transform** and change the scale to **(0.5, 0.5)**.
- Add a child node of type **CollisionShape2D** to the node **player**. We will set-up this node later.
- Add a child node of type **Camera2D** to the node **player**.

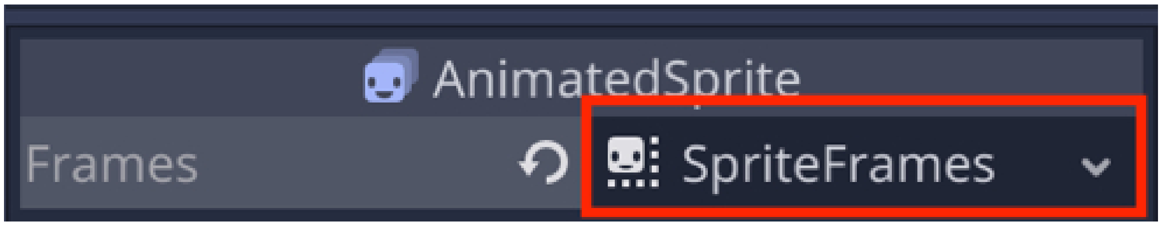
At this stage we have added all the elements that will be necessary to design our character; our next step will be to create the animations that will be used for this character.

- Please select the node **AnimatedSprite**.
- Using the **Inspector**, scroll down to the section **AnimatedSprite | Frames**, click on the downward-facing arrow to the right of the label **Frames** and select the option **New SpriteFrames**, as per the next figure.

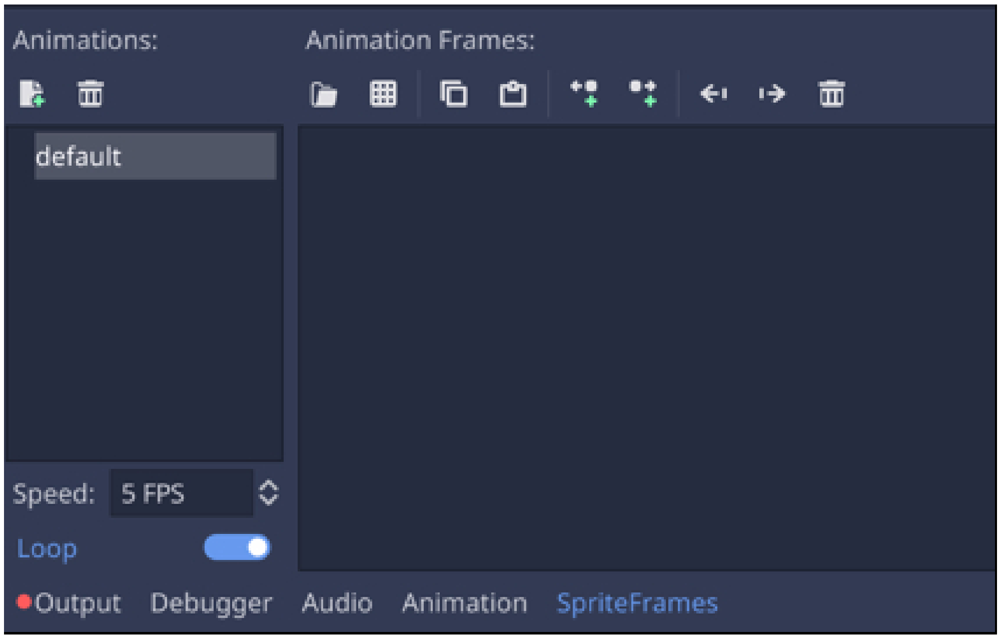


.

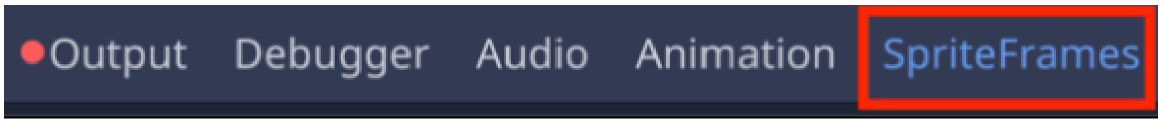
- Once this is done, please click to the left of the downward-facing arrow, on the item labeled **SpriteFrames**,



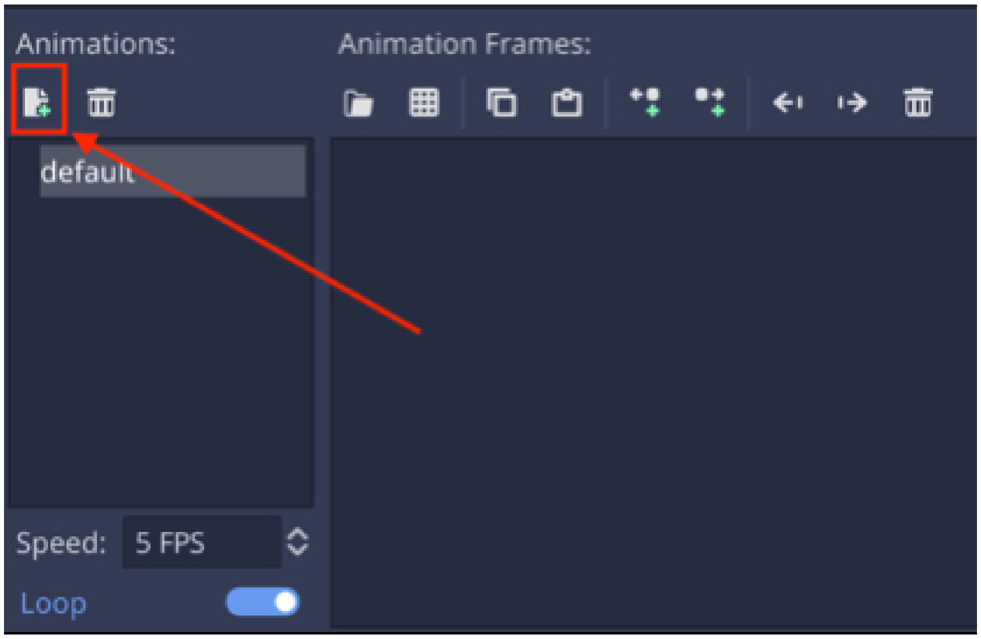
- The **SpriteFrames** window should open (at the bottom of the screen), as per the next figure.



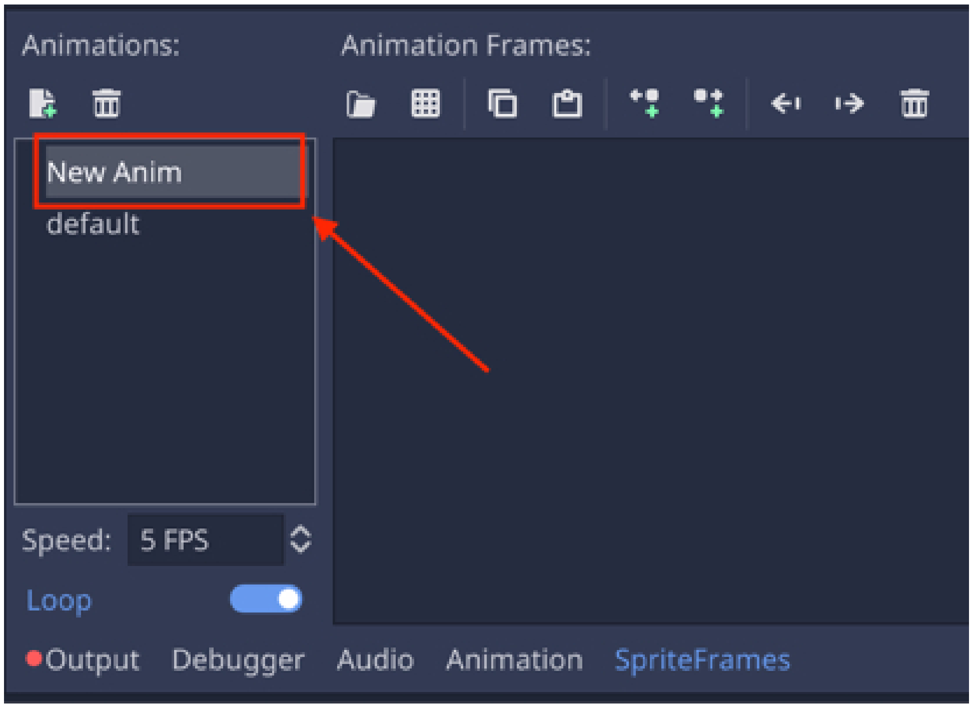
- If it doesn't open, please press the tab **SpriteFrames** located at the bottom of the window.



- Click on the button to create a "**New Animation**", as per the next figure.

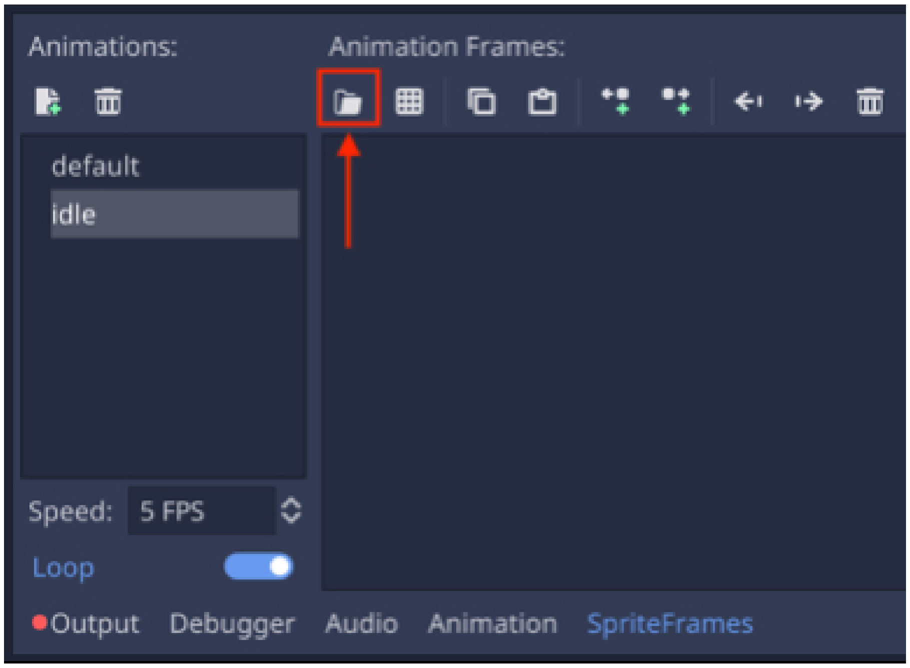


- This will create a new animation label **NewAnim**.

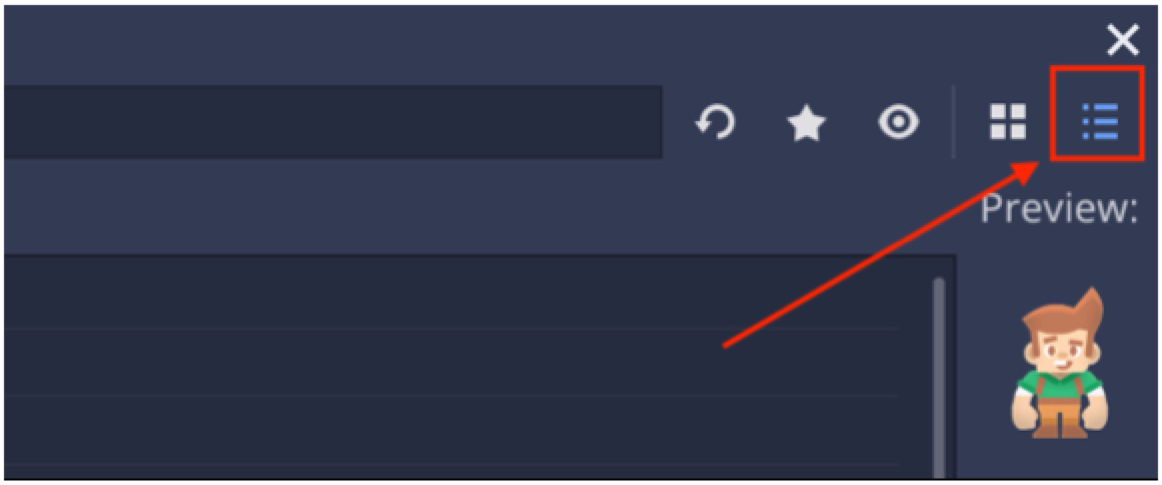


- Please double click on this label and change its name to **idle**.

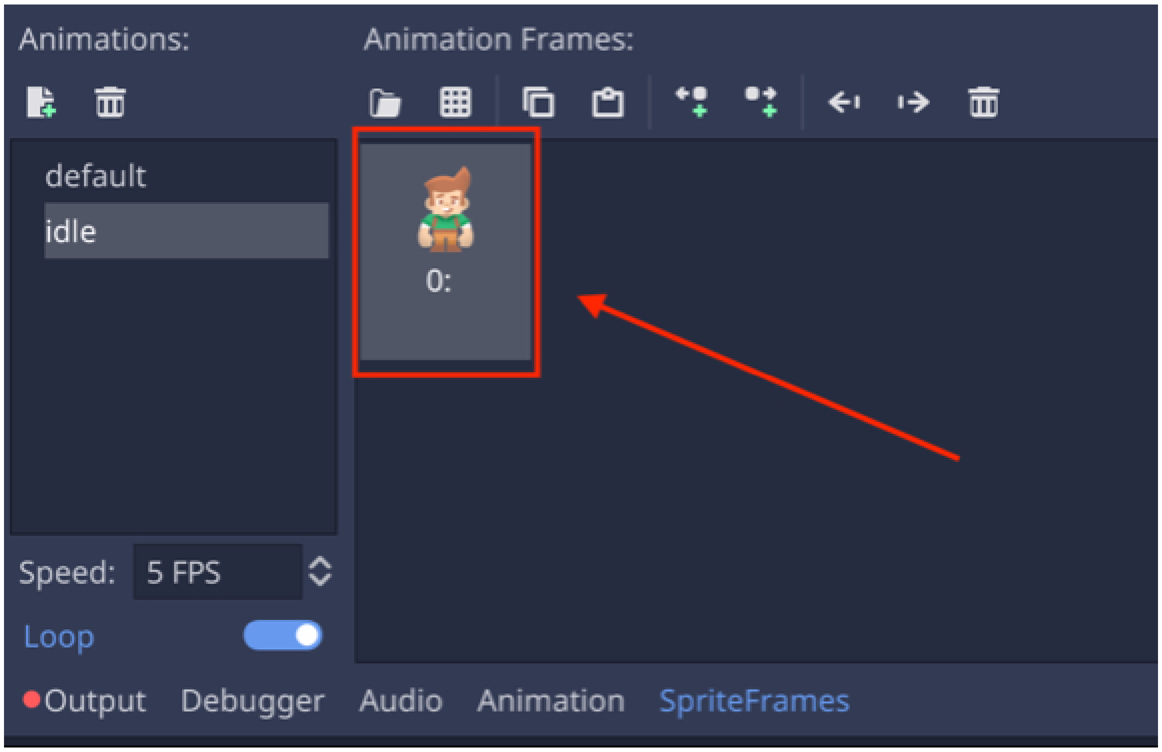
- Select the button labeled "**Add a texture from File**", as per the next figure.



- In the new window, navigate to the folder: **res"//animations/player_character**, and select the file called **character_maleAdventurer_idle.png.**
- To be able to see the files more clearly, you may use the option "**View Items as List**", as illustrated in the next figure

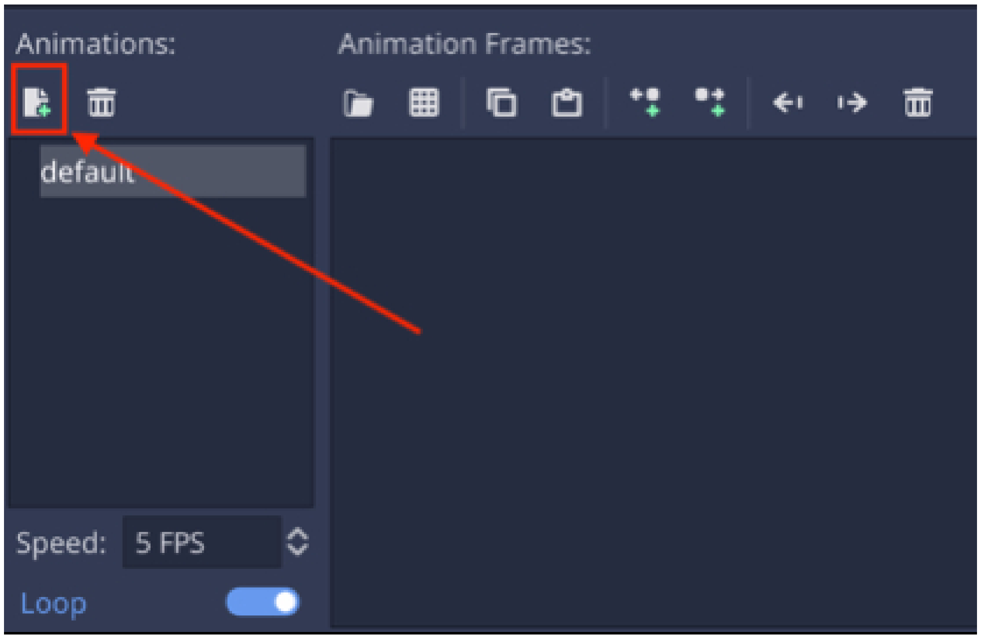


- Once you have selected the file, please press the button labeled "**Open**". This should add one image to the animation **idle**.



So far we have managed to create the animation for the **idle** state, when the player character is not moving; to be completely accurate this is not animation, because there is no change in the image over time; this being said, for the next state (i.e., **walking**), we will, indeed, create a new animation (i.e., that includes more than one image).

- Please create a new animation by clicking on the corresponding button.



- Rename the new animation **walking**.
- Click on the button labeled “**Add a texture from File**”, as we have done for the previous animation.
- In the new window, navigate to the folder: **res”//animations/player_character**, and select the file called **character_maleAdventurer_idle.png**, as we have done earlier.

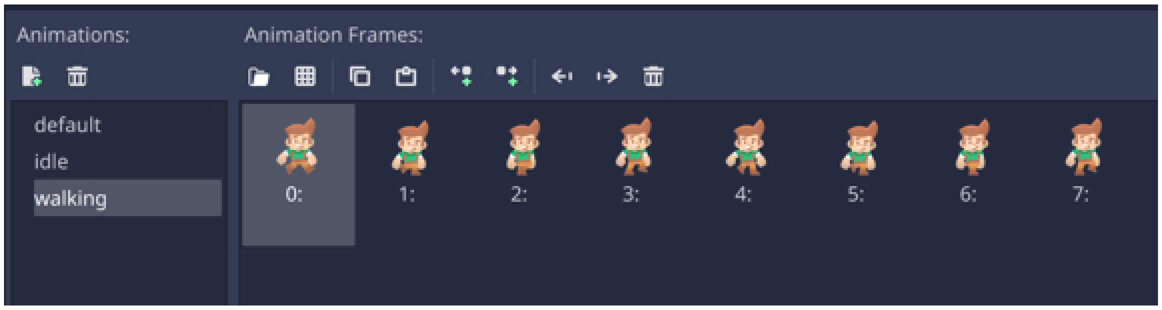
This time, select the following eight files (**SHIFT + CLICK** or **CTRL + CLICK**):

- character_maleAdventurer_walk0.png,
- character_maleAdventurer_walk1.png,
- character_maleAdventurer_walk2.png
- character_maleAdventurer_walk3.png,
- character_maleAdventurer_walk4.png,
- character_maleAdventurer_walk5.png,
- character_maleAdventurer_walk6.png,

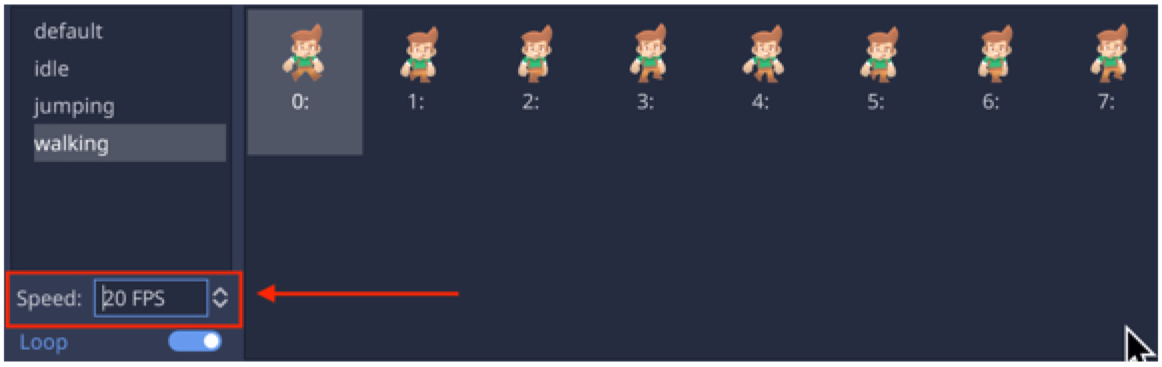
- character_maleAdventurer_walk7.png.

Once this is done, please press the button labeled "**Open**".

- This should add these eight images to the animation walking, as illustrated in the next figure.

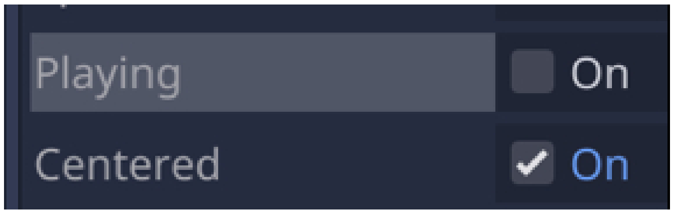


- Please change the speed of the animation to 20 FPS, using the corresponding text field located in the bottom-left corner of the window, as per the next figure.



Speed: 20 FPS

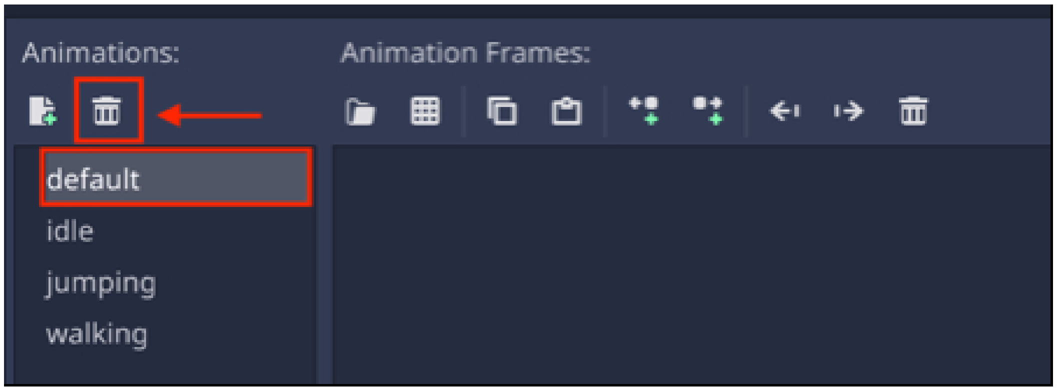
- You can preview the animation by zooming-in on the character in the view port and by then setting the option **Playing | On** to **true** in the **Inspector**.



Please reset the option **Playing | On** to **false** after checking that the animation works well.

Now that we know that the animation is working, we just need to create an additional animation for the jumping movement; it will consist in, as for the **idle** state, just one image.

- Please create a new animation, rename it **jumping**, and associate it with the image **character_maleAdventurer_jump.png**.
- You can also delete the animation called **default**, by selecting it and then pressing the button symbolized by a bin, as illustrated in the next figure.

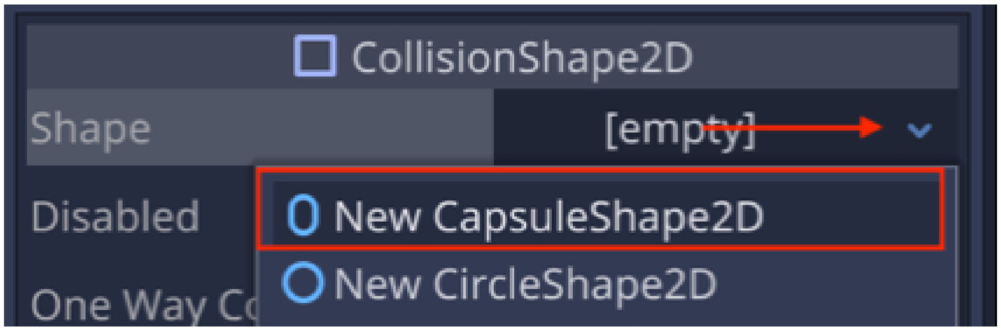


So at this stage, you have created three animations for the **idle**, **walking** and **jumping** states.

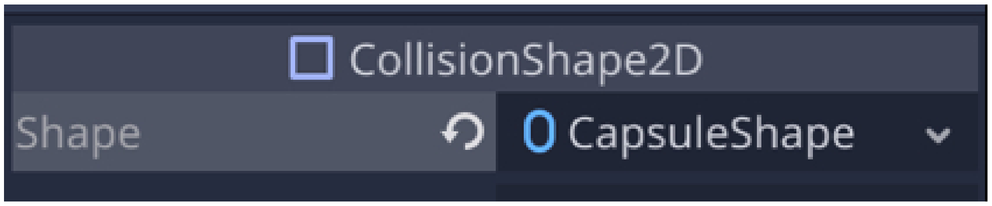
The next thing we will need to do is to configure the **CollisionShape2D** node attached to the player character; this is so that collisions can be detected between the player character and other objects such as platforms or items to collect.

- Please select the node **CollsionShape2D**.

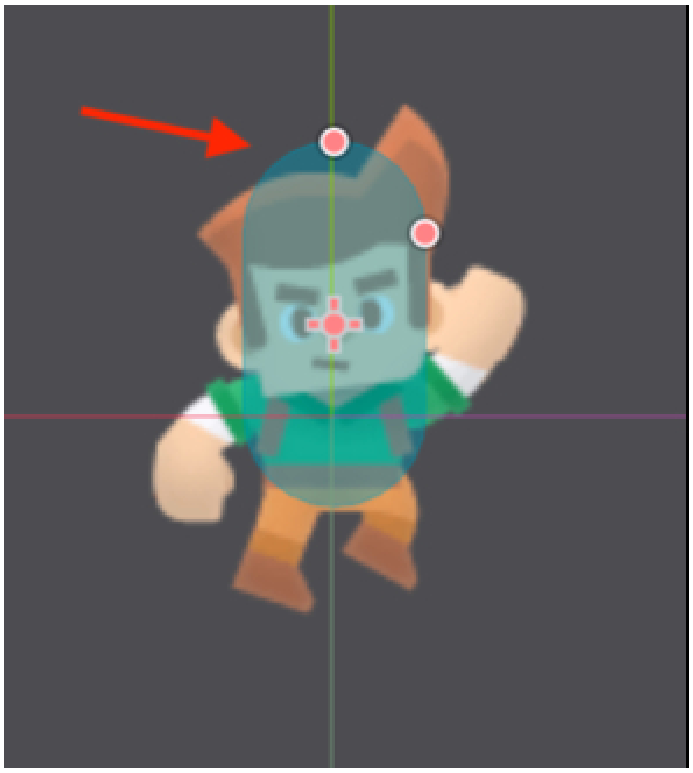
- Using the **Inspector**, scroll down to the section **CollisionShape2D**, click on the downward-facing arrow to the right of the label **Shape**, and select the option **New CapsuleShape2D**.



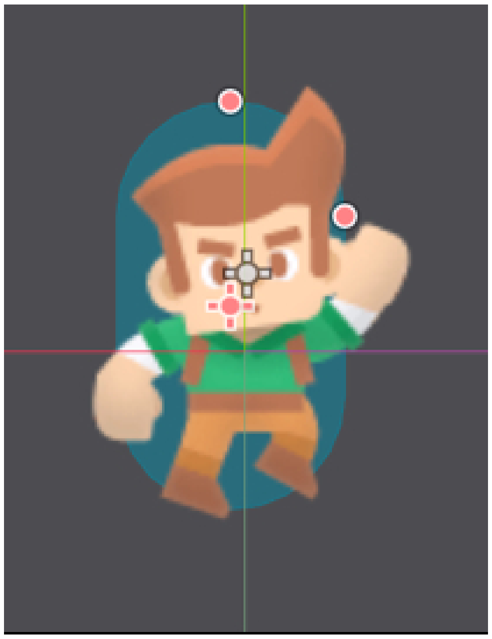
- Click on the new item labeled **CapsuleShape2D** that has just been created.



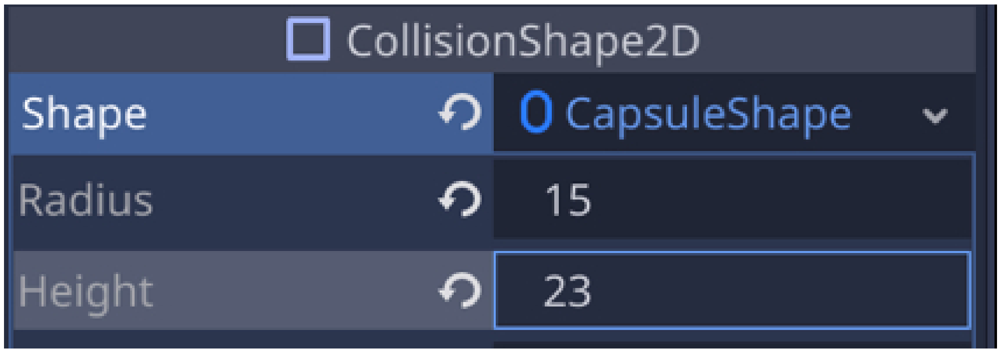
- You should now see, in the **viewport**, a green capsule that is superimposed atop the character player.



- This capsule defines the boundary of the collision shape; please drag the red handles (i.e., the red dots) to modify its width and height so that the capsule covers most of the torso and legs of the player character, as illustrated in the new figure.



- If in doubt, you can switch to the **Inspector**, scroll down to the section **Collision Shape2D** and change the radius and height of the **CollisionShape2D** to **15** and **23** respectively.



CollisionShape2D

Shape	CapsuleShape
Radius	15
Height	23

So we now have a character with animations for its different states, along with the ability to collide with other items in the game. The next step will be to move this character.

- Please save your scene as **player_character.tscn (Scene | Save Scene As)**.

Moving the character

In this section, we will detect the player's keyboard inputs so that the player character can be moved accordingly; in our game we will be using the following scheme:

- The left and right arrow keys will move the character to the right and to the left.
- The space bar will make the character jump.
- When the player is not pressing any of these keys, then the character will be idle.

To perform all these actions, we will be creating a script attached to the player character

So let's get started:

- Please open (or switch to) the scene **player_character.tscn** (i.e., the one that you have just created).
- Select the node called **player**.
- Attach a new script to this node and name the script **character.gd**.
- Please add the following code to the script (new code in bold):

extends KinematicBody2D

const SPEED = 100

const GRAVITY = 4

const JUMP_FORCE = 180

var velocity = Vector2(0,0)

In the previous code we define the constants **SPEED**, **GRAVITY**, and **JUMP_-FORCE** which will be used to define the speed of the character when it moves to the left and to the right, the gravity, and the speed of the jumps, respectively. We also define a **Vector2** variable called **velocity** that will define the overall velocity of the character.

- Please add the following function to the script:

```
func _physics_process(delta):
if (Input.is_action_pressed("move_right")):
velocity.x = SPEED
if (is_on_floor()):
$AnimatedSprite.play("walking")
$AnimatedSprite.flip_h = false
elif (Input.is_action_pressed("move_left")):
velocity.x = -SPEED
if (is_on_floor()):
$AnimatedSprite.play("walking")
$AnimatedSprite.flip_h = true
else:
$AnimatedSprite.play("idle")
```

In the previous code:

- We create the function **physics_process** that is called every frame
- In this function, we detect the player's keyboard inputs.
- If the player presses the action mapped to the key "**move_right**" the horizontal velocity is set.
- We then check whether the player is on the ground; this is because the player can press the left and right keys even when the character is in the air; when the player character is on the ground, the animation should be the **walking** animation. Note that we use the keyword **$Sprite**, which means that we select

a child node of type **Sprite**.

- We then flip the animated image; this is because the walking animation as we have defined it earlier, shows the character going to the right; so to be able to re-use this animation for the left movement, we flip it horizontally when the character is walking to the left, and leave it unchanged (i.e., no flip) when the character is moving to the right. Note that if the player character is in the air and that the jumping animation is played (as we will see later), this animation will still be flipped.
- We then do the exact same but this time for the action "**move_left**".
- Finally, we set the animation to **idle** if none of these actions are processed (i.e., **move_right** or **move_left**).

We can now process the jumping movement:

- Please add the following code, after the one you just created (new code in bold):

else:
$AnimatedSprite.play("idle")
if(Input.is_action_just_pressed("jump")):
velocity.y = -JUMP_FORCE
$AnimatedSprite.play("jumping")

In the previous code:

- We check whether the player has pressed the key linked to the action "**jump**".
- We set the vertical velocity using the constant **JUMP_FORCE**.
- We set the corresponding animation (or the image in that case).

Finally, we just need to determine the overall velocity for the player character and move it accordingly.

- Please add the following code, just after the previous code (new code in bold):

$AnimatedSprite.play("jumping")

velocity.y += GRAVITY

velocity = move_and_slide(velocity, Vector2.UP)

if (is_on_floor()): move_and_slide_with_snap(velocity, Vector2.DOWN*50, Vector2.UP)

velocity.x = lerp(velocity.x,0,0.2)

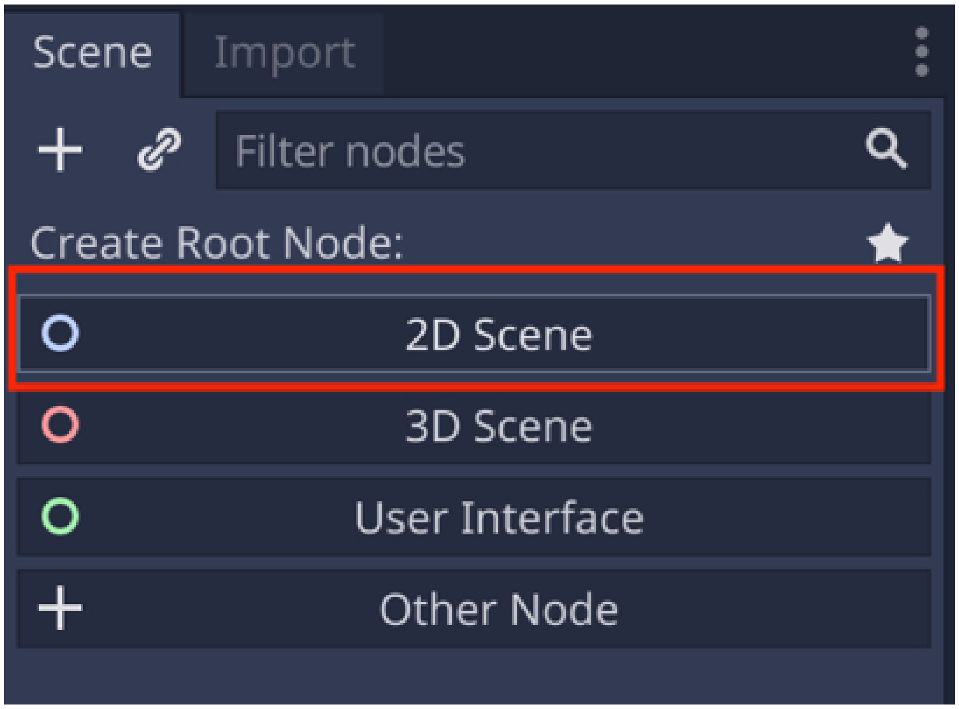
In the previous code:

- We account for the gravity and modify the vertical velocity accordingly.
- We set the velocity.
- We move the character left or right if it is on the ground
- Finally, we make sure that the character eventually stops (horizontal velocity) over time using the keyword **lerp**; this will interpolate between the current horizontal velocity and a velocity of **0** over time; **lerp (velocity.x, 0, 0.2)** means that the horizontal speed will be decreased by **20%** every frame until it reaches **0**.

Please save your code and the scene.

It is now time to test the movement of our character. For this purpose, we will create a new scene and add a simple platform to it.

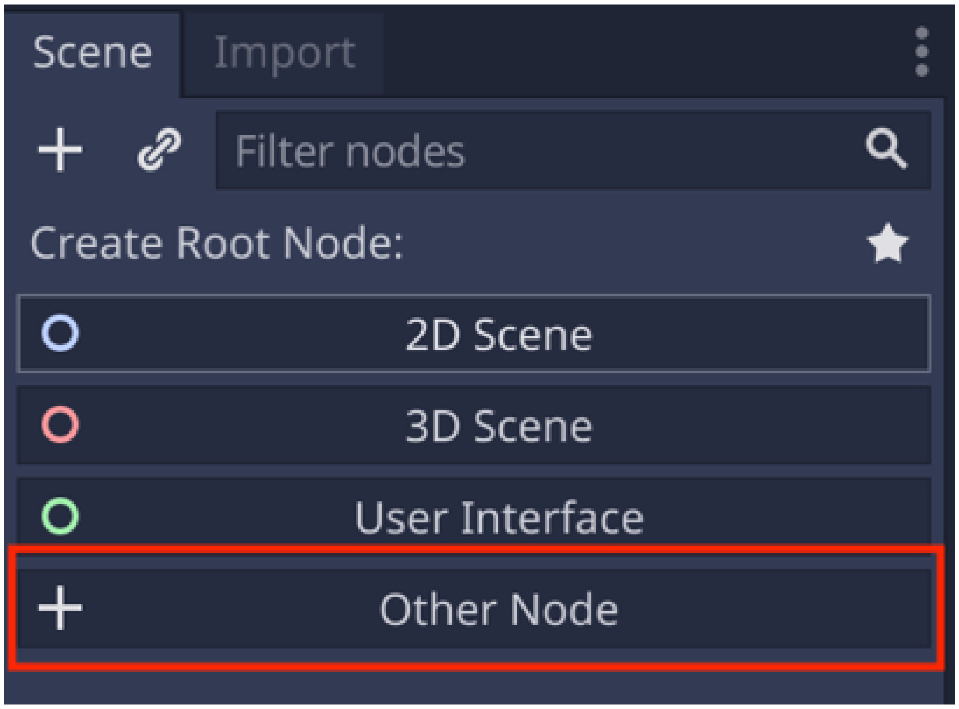
- Please save the current scene (**Scene | Save Scene**).
- Create a new scene (**Scene | New Scene**).
- In the new window, select the option "**2D Scene**".



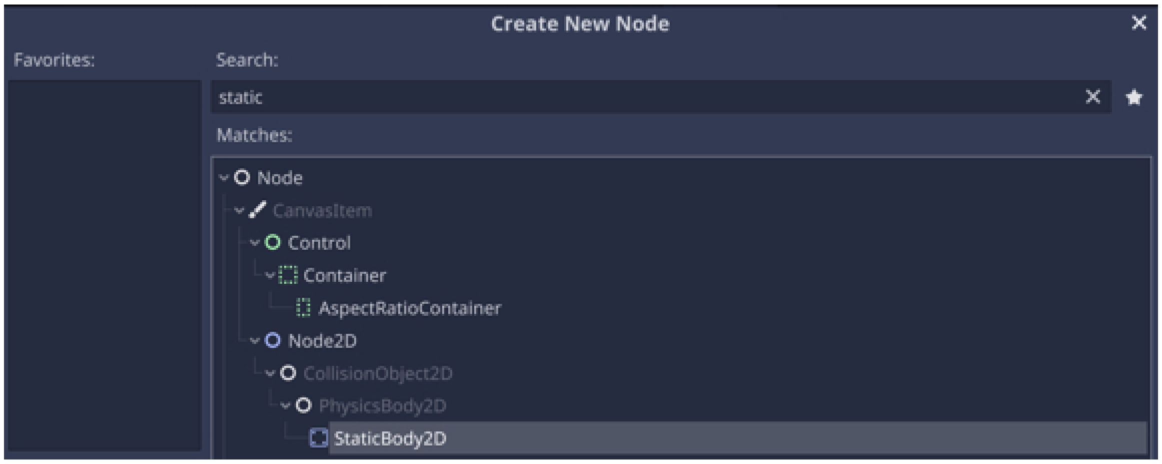
- This will create a new scene with a default node of type **Node2D**.
- This will be our main scene for the platformer.
- Please save this scene as **platformer.tscn** (**Scene | Save As**).

At this stage, we just need to create a template for a platform as a separate scene and to then add it to the main scene.

- Please create a new scene (**Scene** | **New Scene**).
- In the new window, select the option **Other Node**.



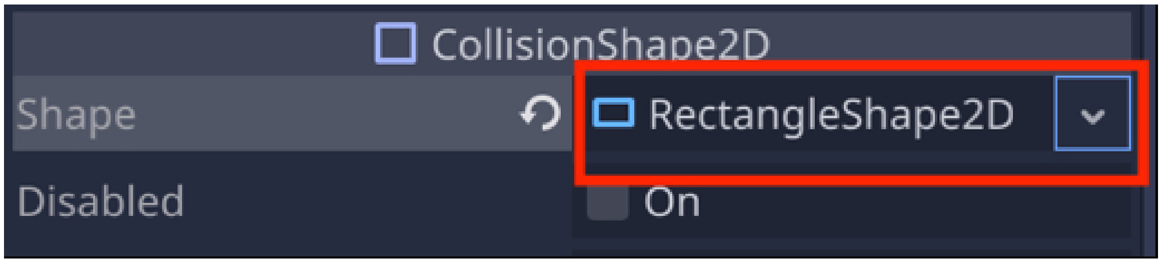
- In the new window, please select the type **StaticBody2D** and click on the button labeled "**Create**".



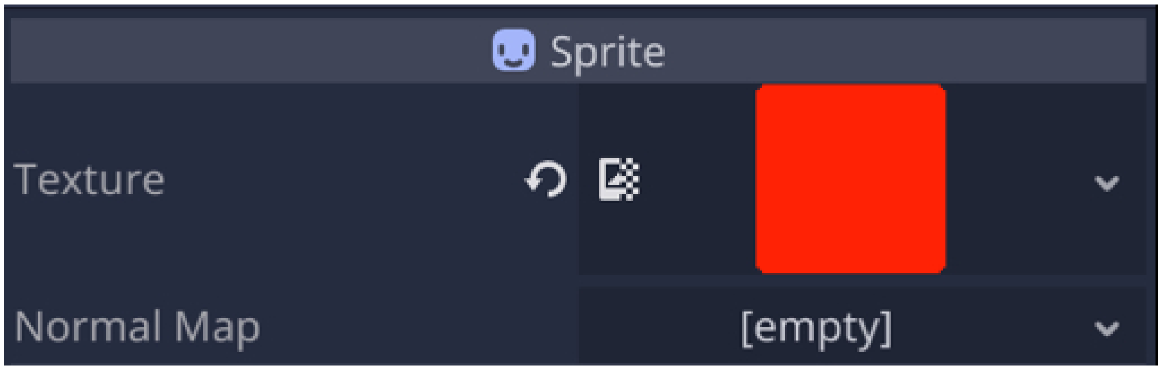
- This will create a new scene with a default node of type **StaticBody2D**.
- Please rename this node **platform**.
- Add a child of type **CollisionShape2D** to the platform node.

- Add a child of type **Sprite** to the platform node.
- Please select the node **ColisionShape2D**.
- Using the **Inspector**, scroll down to the section called **CollisionShape2D**.

  Click on the downward-facing arrow to the right of the label **Shape**, and select the option **New RectangleShape2D**.
- Once this is done, click on the label **RectangleShape2D**.



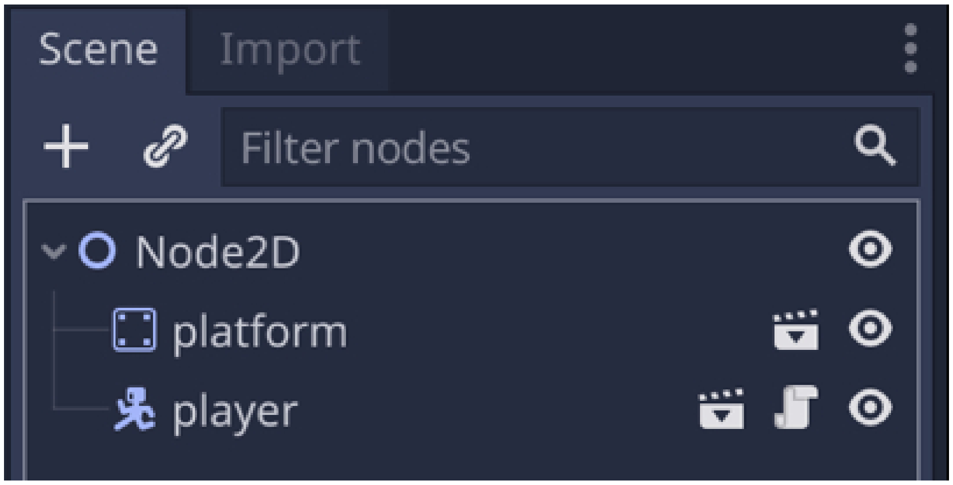
- Please select the node called **Sprite**.
- Using the **Inspector**, scroll down to the section called **Texture** and drag and drop the texture **red.jpg** (or any other texture of your choice) from the **File System** tab to the empty field to the right of the label **Texture**.



- Change this node's scale properties to **(2, 2)**.
- Select the node platform an change its **scale** to **(10, 1)**.
- Please save the current scene as **platform.tscn** (**Scene | Save Scene**).

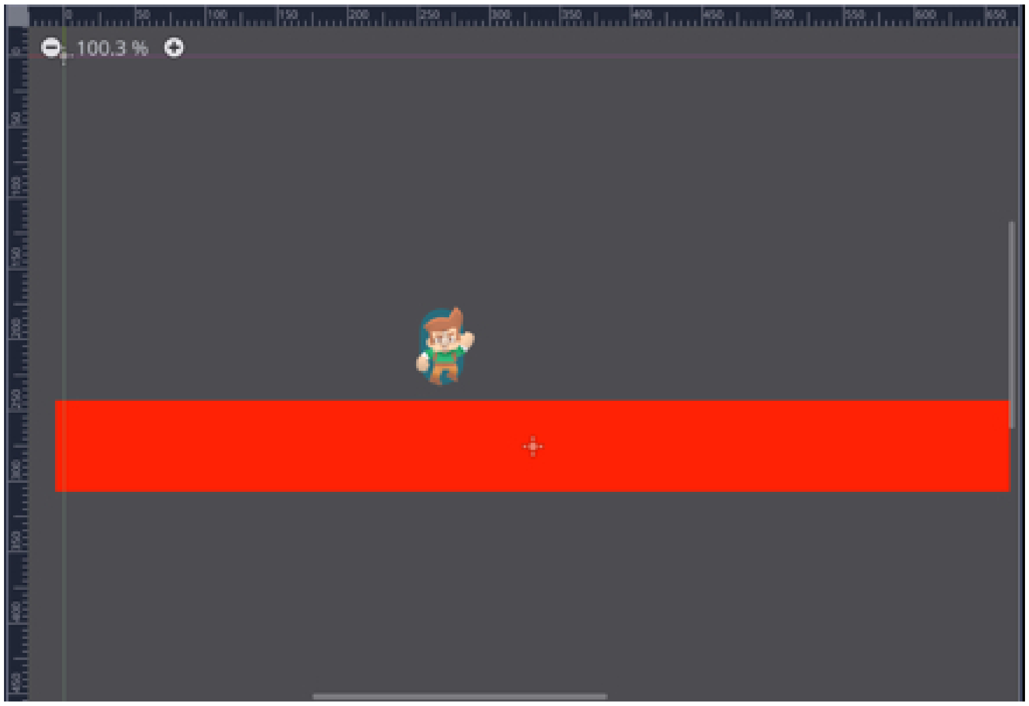
So at this stage, we have all the components to create our first scene, and it is time to combine them:

- Please re-open (or switch to) the main scene **platformer.tscn**.
- Right-click on the node **Node2D**.
- Select the option **Instance ChildScene**.
- In the new window, select the scene **platform.tscn**, and press the button labeled “**Open**.”
- This will add a new node called “**platform**” to the current scene.
- Please right-click on the node **Node2D**.
- Select the option **Instance ChildScene**.
- In the new window, select the scene **player_character.tscn**, and press the button labeled “**Open**.”
- At this stage, you should have three nodes in your scene: **Node2D**, **platform**, and **player**.



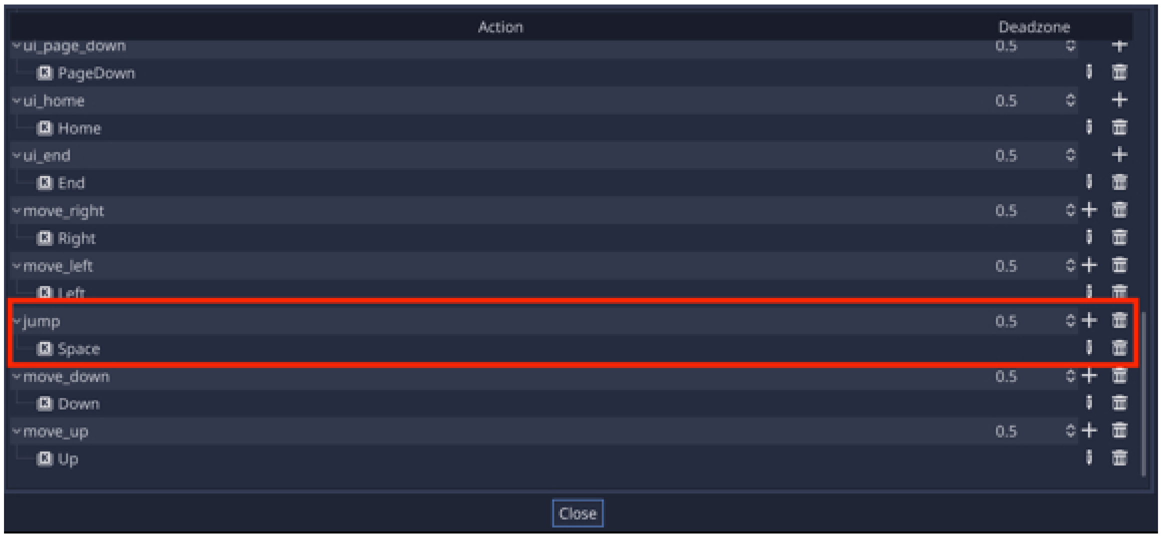
Node2D
platform
player

- Using the viewport, please move the player character just above the platform, as per the next figure.



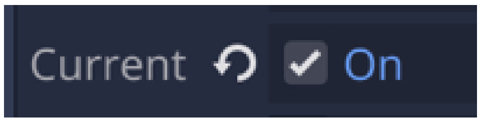
Before we can play our scene, we just need to map the space key to the action "**jump**",

- Please open the Project's input preferences (**Project | Project Settings | Input Map**).
- In the new window, add the action called "**jump**".
- Map this action to the space bar.



Action	Deadzone
ui_page_down	0.5
PageDown	
ui_home	0.5
Home	
ui_end	0.5
End	
move_right	0.5
Right	
move_left	0.5
Left	
jump	0.5
Space	
move_down	0.5
Down	
move_up	0.5
Up	

Last but not least, switch to the scene **player_character.tscn**, select the **Camera** node and, using the Inspector, set its attribute **Current** to **On** so that this camera is active.



Current ✓ On

Once this is done, it is time to test our game. Please play the scene and check that the character can move right and left, and that it can also jump.

Making it possible for the player to collect items

In this section, we will write code that will make it possible for the player to collect items.

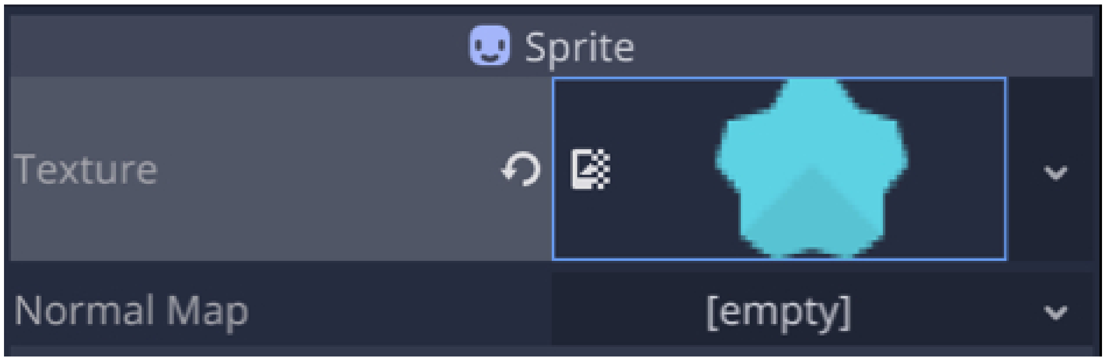
So we will do the following:

- Create a template scene for **diamonds** to be collected. These will consist of an **Area2D** node coupled to a **CollisionShape2D** node and a **Sprite** node.
- An event will be generated whenever the player enters the area defined for this node.
- When this event is triggered (i.e., when the player gets close to the diamond), a specific function will be called on a script attached to the diamond, that will destroy the diamond, and also, later on, increase the player's score, and play a sound effect

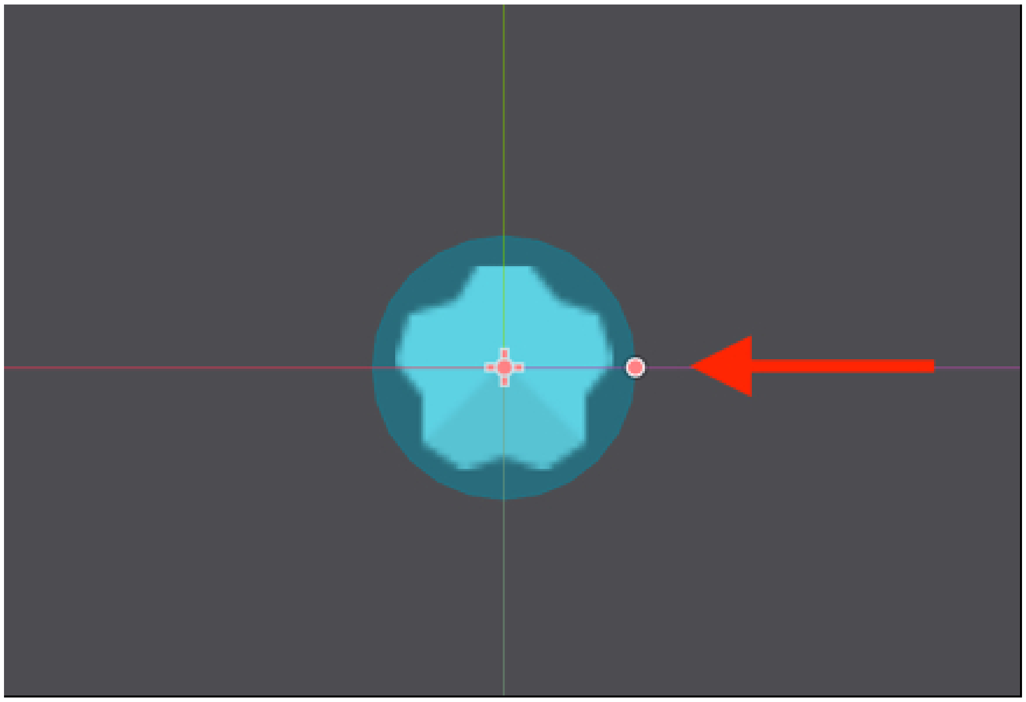
So let's start:

- Please create a new scene (**Scene | New Scene**).
- In the new window, select the option "**Other**".
- In the new window, select the node type **Area2D** and press the button "**Create**".
- This will create a new scene with a default node of type **Area2D**.
- Please rename this node **diamond**.
- Add a child of type **CollsionShape2D** to the node **diamond**.
- Add a child of type **Sprite** to the node **diamond**.
- Select the node **Sprite**.
- Using the **Inspector**, scroll down to the section called **Texture**.
- Drag and drop the texture called **blue_crystal** from the **File System** window in the project to the empty field to the right of the label **Texture** (in the

Inspector).



- Select the node called **CollisionShape2D**.
- In the **Inspector**, scroll down to the section called **CollisionShape2D**.
- Click on the downwards-facing arrow to the right of the label **Shape** and select the option **New CircleShape2D**.
- This should create a blue disc around the diamond image in the viewport.
- Please resize this shape by dragging the red handle, so that the disc fully covers the diamond, as illustrated in the next figure.



At this stage the collision shape is all set up; we just need to detect when the disc is entered by the player and to create a function that will be called in that case.

- Please select the node **diamond**, add a new script to it and rename it **diamond**.
- Open the script.
- Add the following code to it (new code in bold).

extends Area2D

onready var player = get_node("../player")

In the previous code, we create a variable called **player** that will be linked to the player node.

- Please add the following function to this script:

func entered_diamond(body):

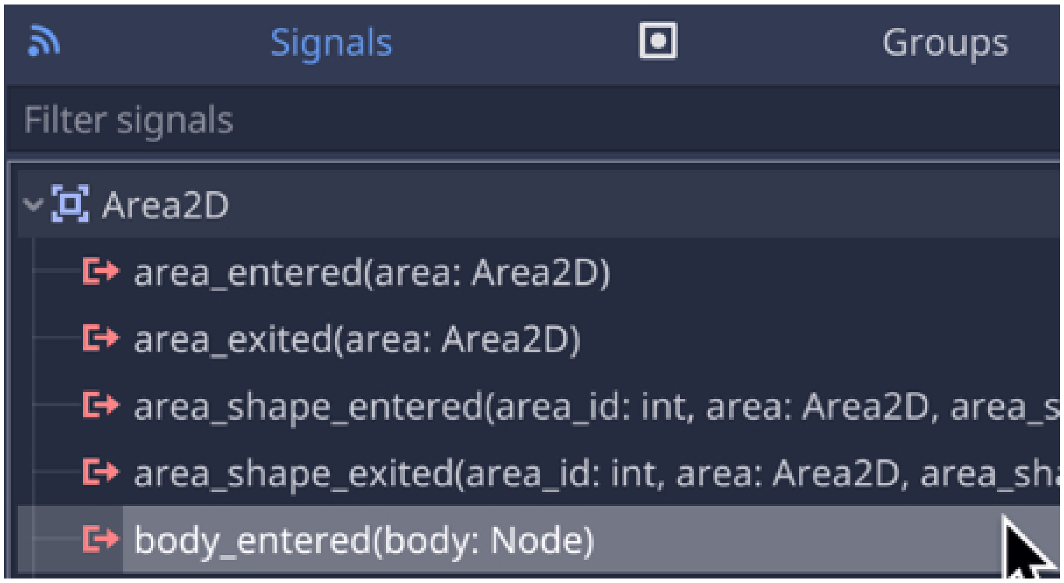
```
player.increase_score(5)
queue_free()
```

In the previous code:

- We create a new function called **entered_diamond**.
- This function calls the function called **increase_score** (that we yet have to create), and passes **5** as a parameter so that the score is increased by **5**.
- We then destroy the current node (i.e., the diamond).

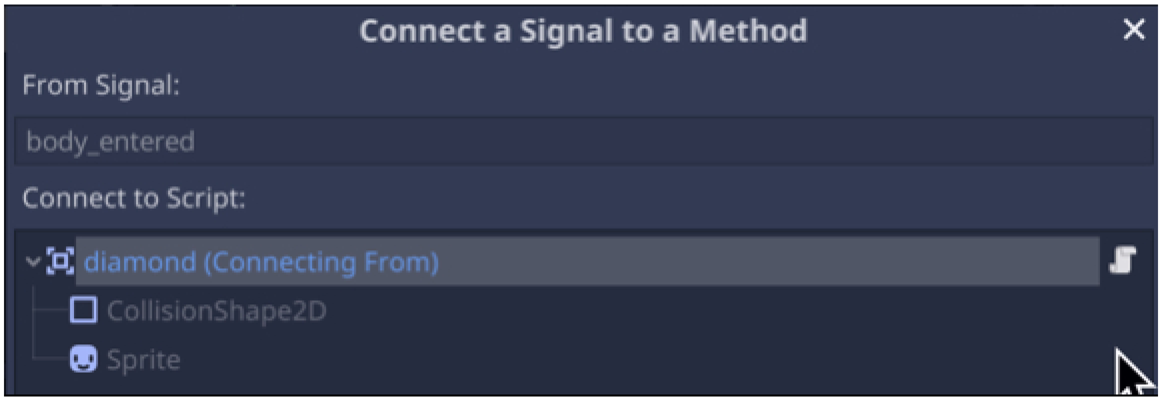
We now need to connect the event called enter to the function that we have just created:

- Please select the node **diamond**.
- Open the tab **Node | Signals**.
- Double click on the event **body_entered**.



```
area_entered(area: Area2D)
area_exited(area: Area2D)
area_shape_entered(area_id: int, area: Area2D, area_s
area_shape_exited(area_id: int, area: Area2D, area_sh
body_entered(body: Node)
```

- Select the node **diamond** from the list.

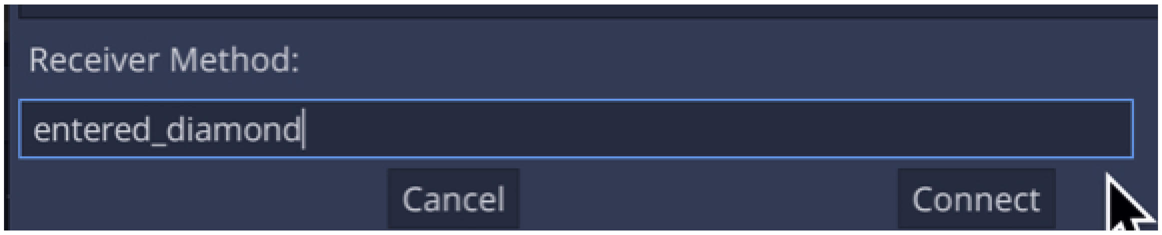

Connect a Signal to a Method

From Signal:

body_entered

Connect to Script:

- Enter the text **entered_diamond** in the field **Receiver Method**, and press **Connect**.


Receiver Method:

entered_diamond

You can now save the current scene as **diamond.tscn** (**CTRL + S**).

It is now time to modify the script linked to the player to create the function **increase_score**:

- Please open the script called **character.gd**.
- Add the following code before the **_ready** function.

```
var score = 0
```

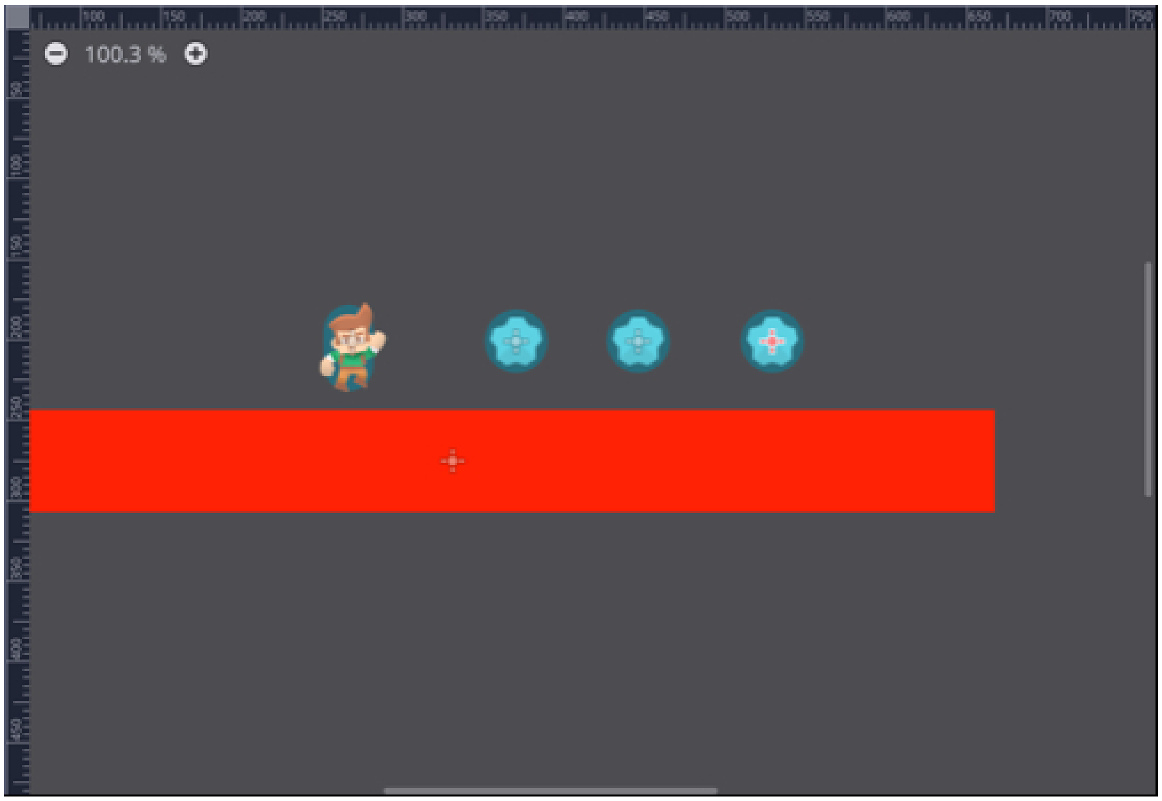
- Please add the following function at the end of the script:

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

In the previous code, we create a new function called **increase_score** that increases the current score by the amount passed as a parameter.

We can now add a few diamonds to the scene and check that the player can collect them.

- Please open (or switch to) the scene called **platformer.tscn**.
- Right-click on the node **Node2D**.
- Select the option **Instance Child Scene** and select the scene **diamond.tscn**.
- This will create a new node called **diamond**.
- You can duplicate this node several times and move the duplicates to different locations in the scene, as illustrated in the next figure.



- You can now play the scene, move the player character and try to collect diamonds.

Adding sound effects

At this stage, the player character is able to collect items in the level, and, to provide additional feedback to the player, we could add a sound effect whenever a diamond has been collected or when the player character is jumping. For this purpose, we will do the following:

- Create a node of type **AudioStreamPlayer**.
- Detect when the player has collected a diamond or jumped.
- In that case, we will load and play the corresponding sound effect.

So let's proceed:

- Please open (or switch to) the scene called **player_character.tscn**.
- Add a child node of type **AudioStreamPlayer** to the node **player**.
- Rename this node **audio_stream_player**.
- Open the script **character.gd**.
- Add the following code to it (new code in bold):

extends KinematicBody2D

export (AudioStream) var collect_sound

export (AudioStream) var jump_sound

onready var audio_stream_player = get_node("audio_stream_player")

In the new previous code:

- We create three variables.
- The first variable is called **collect_sound** and its type is **AudioStream**; the keyword **export** means that this variable will be accessible through the **Inspector**, allowing us to drag and drop a file (in our case an audio file) to the corresponding empty slot in the **Inspector**. This sound will be used when the

player collects an item (i.e., a diamond).

- The second variable is called **jump_sound** and its type is **AudioStream**; the keyword **export** means that this variable will be accessible through the **Inspector**, allowing us to drag and drop a file (in our case an audio file) to the corresponding empty slot in the **Inspector**. This sound will be used when the player jumps (i.e., a diamond).
- The last variable called **audio_stream_player** is linked to the node **audio_stream_ player** that we have just added to the **player** node.

Next, we just need to play these sounds when the player jumps or collects items:

- Please add the following function to the script **character.gd**.

```
func play_collect_sound():
audio_stream_player.stream = collect_sound
audio_stream_player.play()
```

- Please add the following code to the script **diamond.gd** (new code in bold).

func entered_diamond(body):

player.increase_score(5)

player.play_collect_sound()

queue_free()

- Please add the following code to the script **character.gd**.

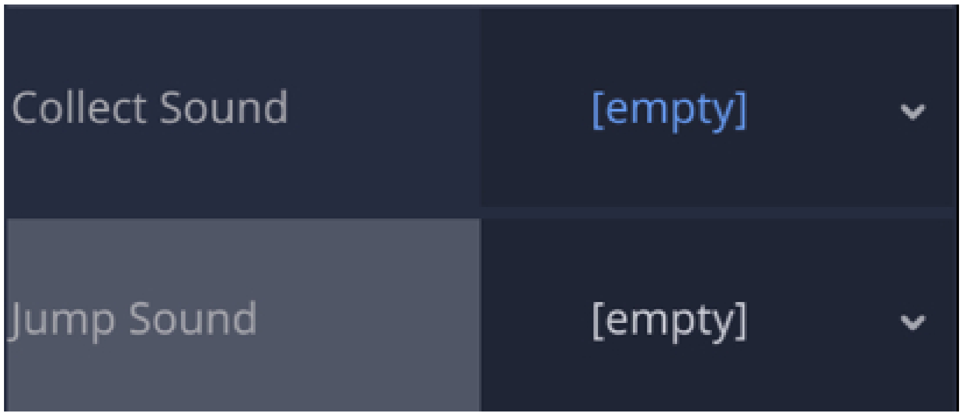
```
if(Input.is_action_just_pressed("jump") ):
velocity.y = -JUMP_FORCE
$AnimatedSprite.play("jumping")
```

audio_stream_player.stream = jump_sound

audio_stream_player.play()

Last but not least, we need to initialize the variables **jump_sound** and **collect_sound**:

- Please switch to the scene **player_character.tscn**.
- Select the node **player**, and you should now see two empty slots named **Jump Sound** and **Collect Sound**.



- Drag and drop the file **collect_sound.wav** from the **File System** tab (from the folder **audio**) to the empty slot to the right of the label **Collect Sound**.
- Drag and drop the file **jump_sound.wav** from the **File System** tab (from the folder **audio**) to the empty slot to the right of the label **Collect Sound**.
- Once this is done, you can switch back to the main scene (i.e., **platformer.tscn**), play the scene and check that a sound effect is played whenever the player character collects an item or jumps.

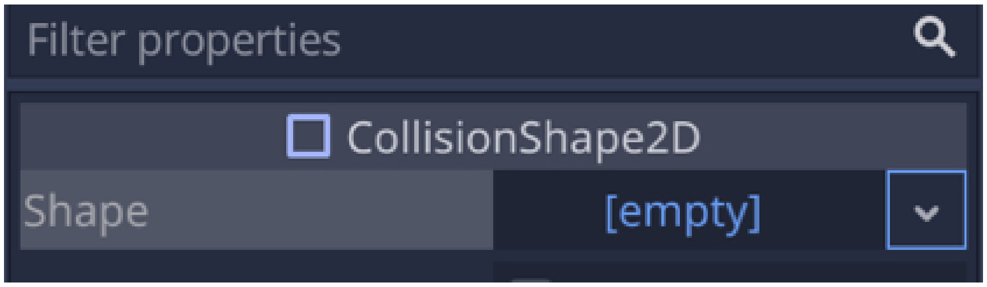
Adding a dead zone to detect when the player has fallen

So at this stage, our character can move on the platform and collect items with corresponding sound effects. However, from time to time, your player character may fall indefinitely off the edges of the current platform. So it would be great to be able to detect when the player character has fallen, and to put it back to its original position accordingly when that happens. For this purpose we will proceed as follows:

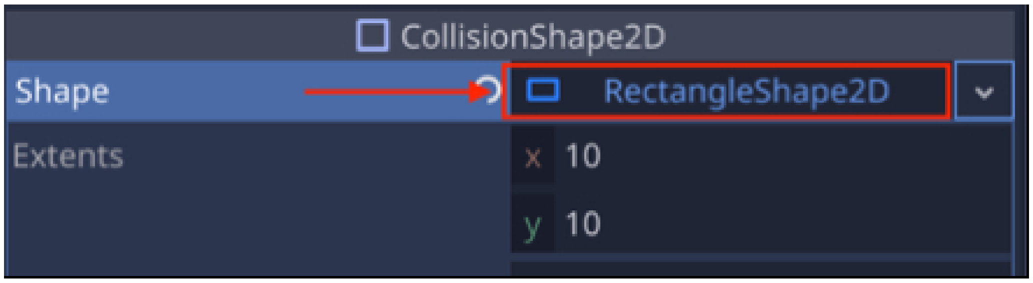
- Save the starting position of the player character at the beginning of the game.
- Create an invisible rectangular located below the platform.
- Detect when the player collides with this rectangle
- Move the player back to its original position.

So let's proceed:

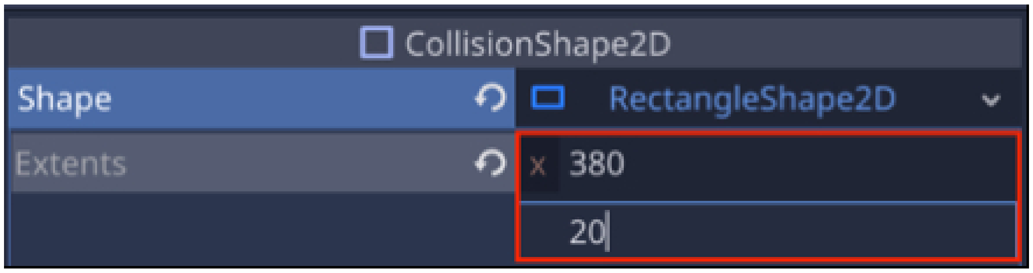
- Please create a new scene (**Scene | New Scene**).
- In the new window, select the option "**Other**".
- In the next window select the node type **StaticBody2D** and press the button labeled "**Create**".
- This will create a new scene with a default node named **StaticBody2D**; please rename this node **dead_zone**.
- Add a child node of type **CollisionShape2D** to the node **dead_zone**.
- Select the node **dead_zone**.
- Using the **Inspector**, click on the downward-facing arrow to the right of the label **Shape** in the section **CollisionShape2D**, and select the option **New RectangleShape2D**.



- You can now click on the label **RectangleShape2D** to the left of the arrow.

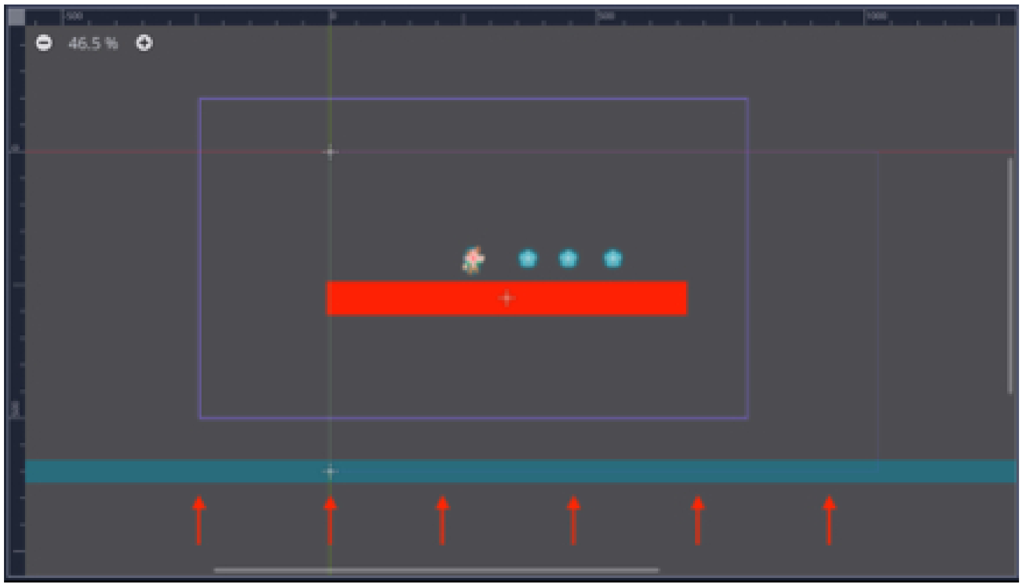


- This will show the attribute **Extents**; please change its value to **(380, 20)**.



- You can now save this scene as **dead_zone.tscn** (**Scene | Save Scene As**).
- Please switch to the main scene (i.e., **platformer.tscn**).
- Add the dead zone: right-click on the node **Node2D** and select the option **Instance Child Scene**.
- In the new window, select the scene **dead-zone.tscn** that you have just created, and press the button labeled "**Open**".
- This will add a new node called **dead_zone** to the scene.
- Using the **Inspector**, rescale this node along the x-axis by changing the scale

attribute to **(10, 1)** and change its position so that it is below the player and the platform, for example, move it to **(0, 600)**, as illustrated in the next figure.



At this stage the dead zone has been created and added to the main scene; so we just need to save the original position of the player, and detect whenever it has collided with this zone.

- Please open the script **character.gd**.
- Add the following code to the script (new code in bold):

const CLIMBING_SPEED = 100

const GRAVITY = 4

const JUMP_FORCE = 180

var initial_position:Vector2

In the previous code, we just declare a vector named **initial_position** that will be used to store the player character's initial position.

- Add the following code to the function **_ready** (new code in bold)

```
func _ready():
initial_position = global_transform.origin
```

In the previous code, we set the value of the vector **initial_position** to be the current position for the player character which is stored in **global_transform.origin**.

- Add the following code to the function **_physics_process** (new code in bold):

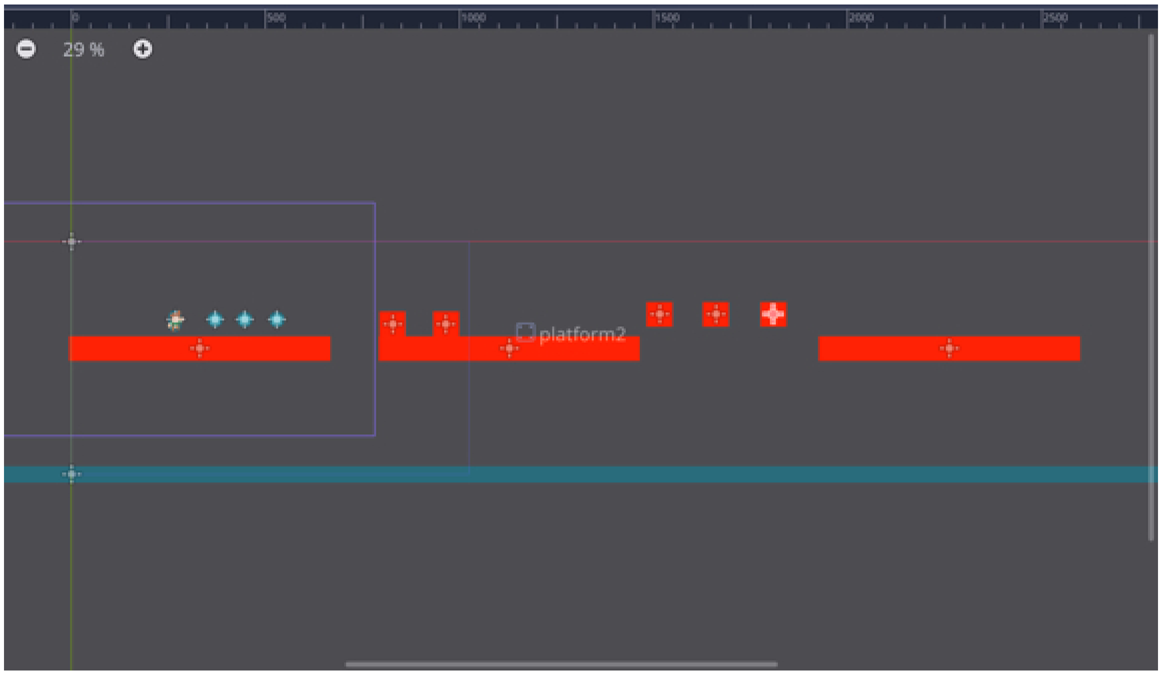
```
velocity.x = lerp(velocity.x,0,0.2)
for i in get_slide_count():
var collision = get_slide_collision(i)
if(collision.collider.name == "dead_zone"):
global_transform.origin = initial_position
```

In the previous code:

- We create a loop that will be used to go through all the nodes currently colliding with the player.
- We create a variable called **collision** that will store one of the nodes currently colliding with the player.
- If this node's name is **dead_zone**, we then move the player to its original position by changing the value of the variable **global_transform.origin** to the value of the variable **initial_position** stored earlier.

You can now save your script and play the scene; move the player character so that it jumps from the platform, you see should, after a few seconds, that it has been repositioned to its original location.

Now that the dead zone is working, you can extend the platform game by duplicating and scaling the platform to achieve the layout of your choice or a layout similar to the one illustrated in the next figure.


0
500
1000
1500
2000
2500
29 %
platform2

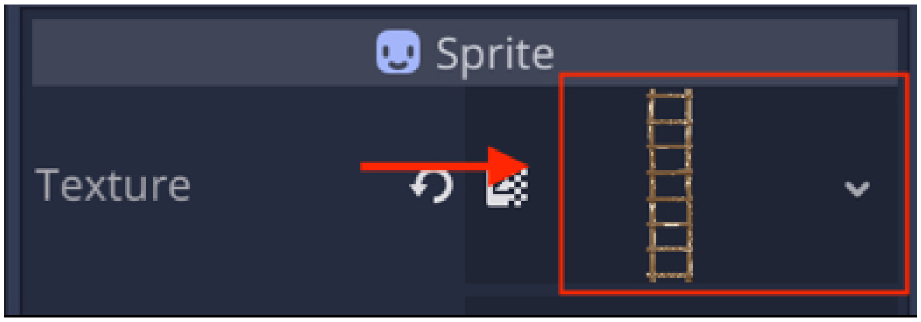
Allowing the player character to climb a ladder

One of the platformers' key features is the ability for the character player to climb ladders; so in this section, we will add this feature and do the following:

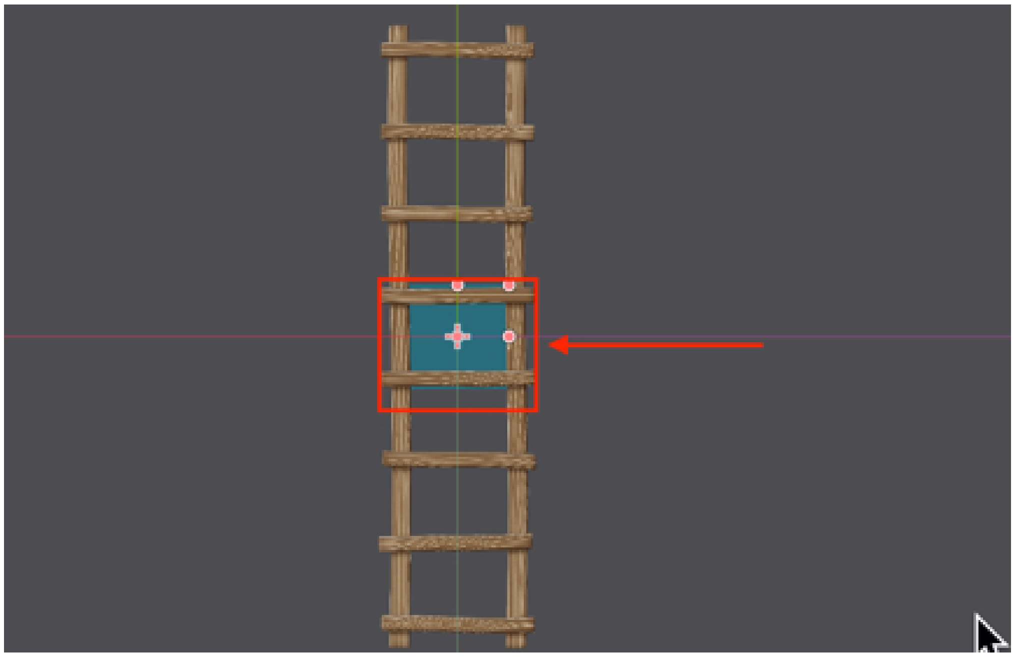
- Create a ladder from a sprite.
- Create an area around this ladder.
- Detect when the player has entered or exited this area.
- Ensure that when the player enters the ladder area, that it can only move up or down, and that the corresponding animation is played.
- Ensure that when the player character exits the ladder, that it will be able to use any of the previous controls to either move left and right or jump.

So first, let's create and configure a ladder:

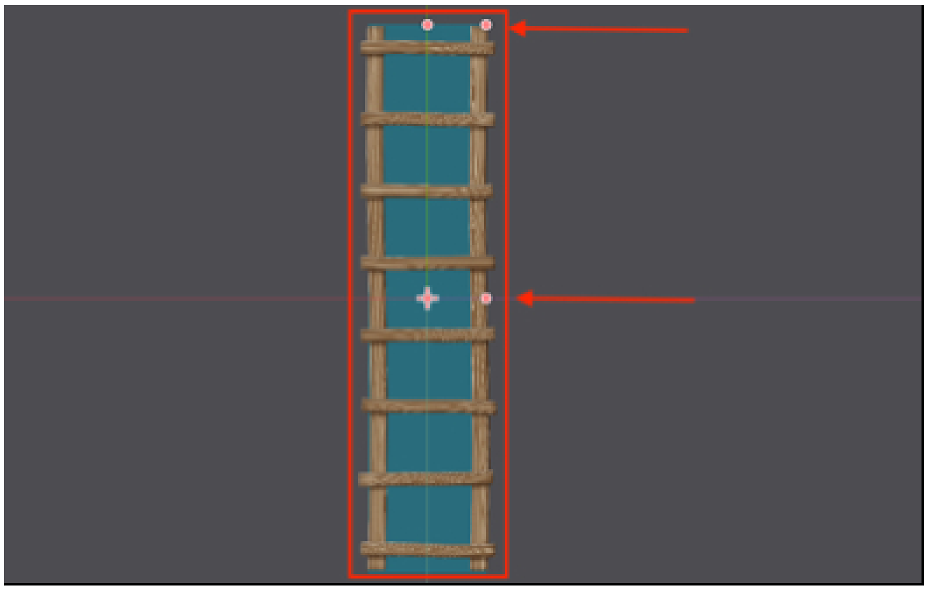
- Please create a new scene (**Scene | New Scene**).
- In the new window, select the option "**Other**".
- In the next window, select the node type **Area2D**.
- This will create a new scene with a default node named **Area2D**.
- Rename that node **ladder**.
- Add a child node of type **CollisionShape2D** to the node **ladder**.
- Add a child node of type **Sprite** to the node **CollisionShape2D**.
- Select the node **Sprite**.
- Using the **Inspector**, locate the section called **Sprite | Texture**.
- Drag and drop the texture **ladder.png** from the **File System** tab to the empty slot in the section **Texture** from the **Inspector**.



- Select the node **ladder** and change its scale to **(0.04, 0.04)**.
- Select the node **CollisionShape2D**.
- Using the **Inspector**, click on the downward-facing arrow to the right of the label **Shape**, in the section **CollisionShape2D**, and select the option **New RectangleShape2D**.
- This will create a new blue area that delimitates the collision area.



- Please drag the red handles so that the blue area covers the ladder, as per the next figure.



Next, we will detect and process the event that is triggered when the player is entering or exiting the area defined for the ladder.

- Please save the current scene as **ladder.tscn**, add a new script to the node **ladder**, and name this script **ladder.gd**.
- Add the following code to the script (new code in bold).

```
extends Area2D
onready var player_node = get_node("../player")
```

- Please add this function to the script:

```
func entered_ladder(body):
player_node.is_near_ladder = true
player_node.snap_ladder_x_pos = global_transform.origin.x
```

In the previous code:

- We create a function called **entered_ladder**; we will, in the next section link this function to the event that is triggered when the player enters the area defined for the ladder.

- We then set the variable **player_is_near_ladder**, which is declared in the script linked to the player, to **true**.
- Finally, we set the variable **player_node_snap_ladder_x_pos**, which is declared in the script linked to the player, to the x coordinate of the ladder. This is so that the layer stays aligned with the ladder as it's going up and down the ladder.

We can now define a function that will handle the **exit** event.

- Please add this function to the script:

```
func exited_ladder(body):
player_node.is_near_ladder = false
player_node.player_is_on_ladder = false
```

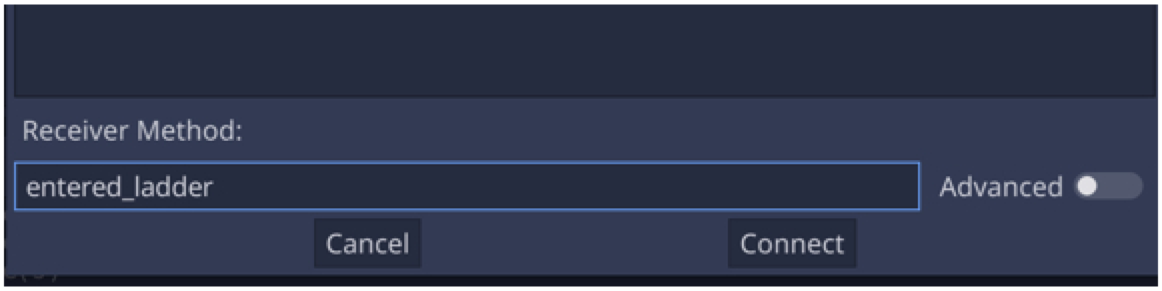
In the previous code:

- We create a function called **exited_ladder**; we will, in the next section link this function to the event that is triggered when the player exits the area defined for the ladder.
- We set the variable **player_is_near_ladder**, that is declared in the script linked to the player, to **false**.
- We set the variable **player_is_on_ladder**, that is declared in the script linked to the player, to **false**.

We just need to link the **exit** and enter **events** for the ladder to these functions:

- Please select the node **ladder**.
- Select the tab **Node | Signals**.
- Double click on the event **body_entered**.

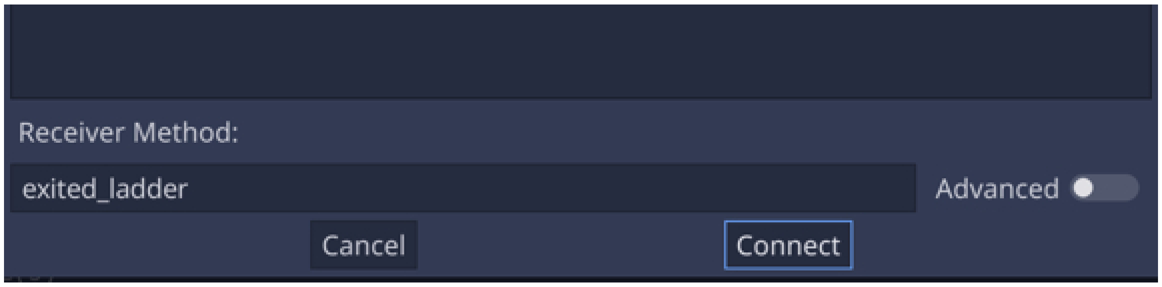
- In the new window, select the node **ladder** and enter the text **entered_ladder** in the text field labeled "**Receiver Method**", and press the button labeled "**Connect**".



Receiver Method:

entered_ladder

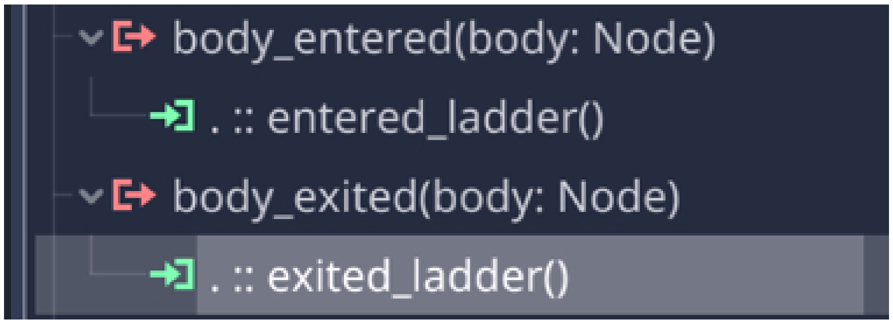
- Select the tab **Node | Signals**.
- Double click on the event **body_exited**.
- In the new window, select the node **ladder** and enter the text **exited_ladder** in the text field labeled "**Receiver Method**", and press the button labeled "**Connect**".



Receiver Method:

exited_ladder

- You should now be able to see in the tab **Node | Signals** that both the signals **body_entered** and **body_exited** have been connected to the functions **en-tered_lad der** and **exited_ladder** respectively, as illustrated in the next figure.



```
body_entered(body: Node)
    . :: entered_ladder()
body_exited(body: Node)
    . :: exited_ladder()
```

Once this is done, we just need to add some code to the script linked to the player node, so that it can move up and down the ladder.

- Please open the script **character.gd**.
- Add the following code before the function **_ready**.

```
var player_is_on_ladder = false
var is_near_ladder = false
var snap_ladder_x_pos
```

- Please indent all the code present in the function **_physics_process** to the right: select that code, right-click on it and select the option **Indent Right** from the contextual menu.
- Add the following code at the start of the function **_physics_process** (new code in bold).

func _physics_process(delta):

if (!player_is_on_ladder):

if (Input.is_action_pressed("move_right")):

velocity.x = SPEED

- Select all the code below the one you just entered (that belongs to the same function) and indent it to the right: select the code, right click on the code selected, and select the option **Indent Right**.

- Change the following code (new code in bold).

```
$AnimatedSprite.play("idle")
if(Input.is_action_just_pressed("jump") && is_on_floor()):
if (!is_near_ladder):
velocity.y = -JUMP_FORCE
$AnimatedSprite.play("jumping")
audio_stream_player.stream = jump_sound
audio_stream_player.play()
else:
player_is_on_ladder = true
global_transform.origin.x = snap_ladder_x_pos
```

In the previous code we check that by pressing the jump button, and when on the floor but away from a ladder, the non-player character will jump.

- Add the following code at the end of the function **_physics_process** (new code in bold)

```
for i in get_slide_count():
var collision = get_slide_collision(i)
if(collision.collider.name == "dead_zone"):
global_transform.origin = initial_position
nb_lives -= 1
if (nb_lives <0): print("Game Over")
else:
velocity.x = 0
velocity.y = 0
global_transform.origin.x = snap_ladder_x_pos
```

In the previous code:

- We add an else statement that corresponds to when the variable **player_is_on_ladder** is **true**.
- In that case, we initialize the vector **velocity**.
- We ensure that the player is always aligned with the ladder.

It is now time to make it possible for the player character to climb up and down the ladder:

- Please add the following code after the one you have just typed (new code in bold):

```
global_transform.origin.x = snap_ladder_x_pos
if (Input.is_action_pressed("move_up")):
velocity.y = 1
global_transform.origin.y -= 1
$AnimatedSprite.play("climbing_ladder")
elif (Input.is_action_pressed("move_down")):
if (!is_on_floor()):
global_transform.origin.y += 1
$AnimatedSprite.play("climbing_ladder")
else:
player_is_on_ladder = false
else:
$AnimatedSprite.play("idle_ladder")
```

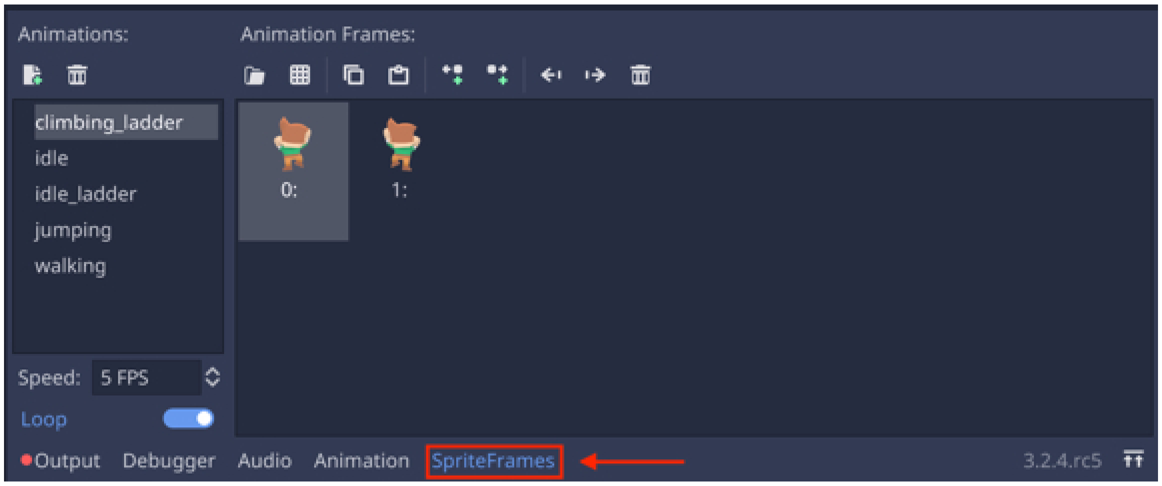
In the previous code:

- If the key mapped to the action "**move_up**" is pressed, we move the player up and we also make sure that the corresponding animation is played (note that we yet have to create this animation).

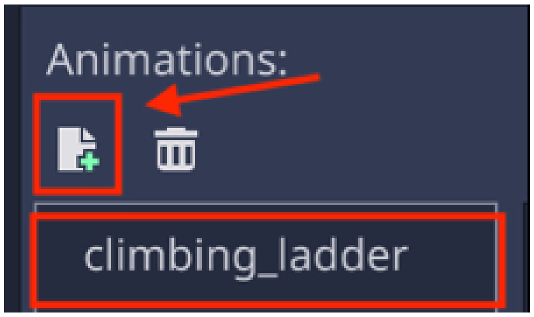
- If the key mapped to the action "**move_down**" is pressed and the player has not reached the floor yet, we move the player down and we also make sure that the corresponding animation is played. Otherwise, if the player has reached the floor, we then set the variable **player_is_on_ladder** to false.
- We set the vertical velocity to -1 to indicate that the player character is going up.
- Otherwise, we play the animation corresponding to the player character being **idle** on the ladder.

The last thing we need to do is to create the animations that will be used for when the player character is either **idle** on the ladder or **climbing** the ladder.

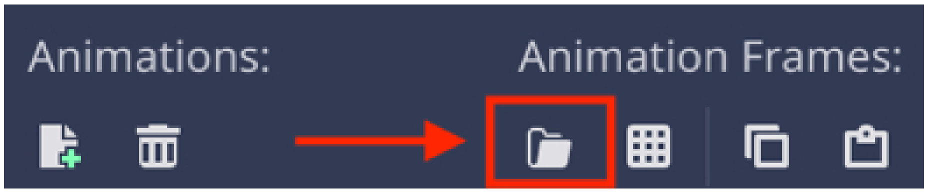
- Please open the scene **player_character.tscn**.
- Click on the node called **AnimatedSprite**.
- Open the **SpriteFrames** window, if it is not already open, by clicking on the corresponding tab as per the next figure.



- Create a new animation by clicking on the corresponding button and rename the new animation **climbing_ladder,** as illustrated in the next figure.

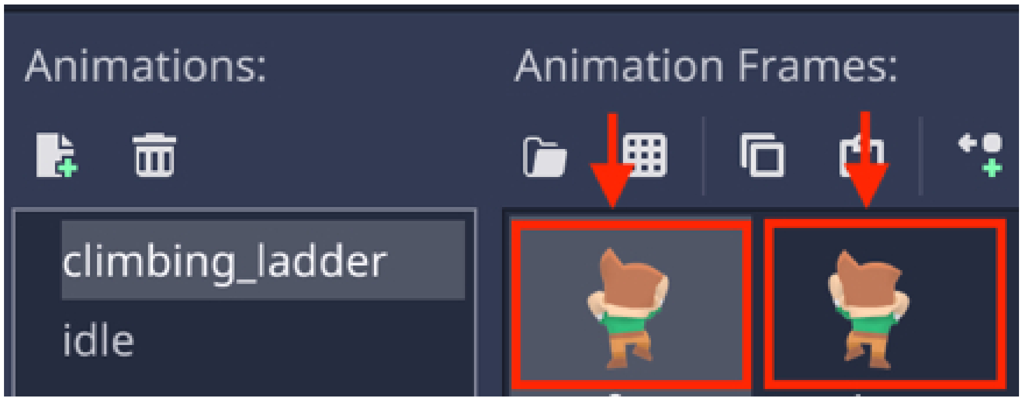


- Click on the button labeled "**Add a Texture from File**", as illustrated in the next figure.



In the new window, select both the files **character_maleAdventurer_climb0.png** and **character_maleAdventurer_climb1.png**, and press the button labeled "**Open**".

- This should create two images that are now part of the animation **climbing_ladder**,

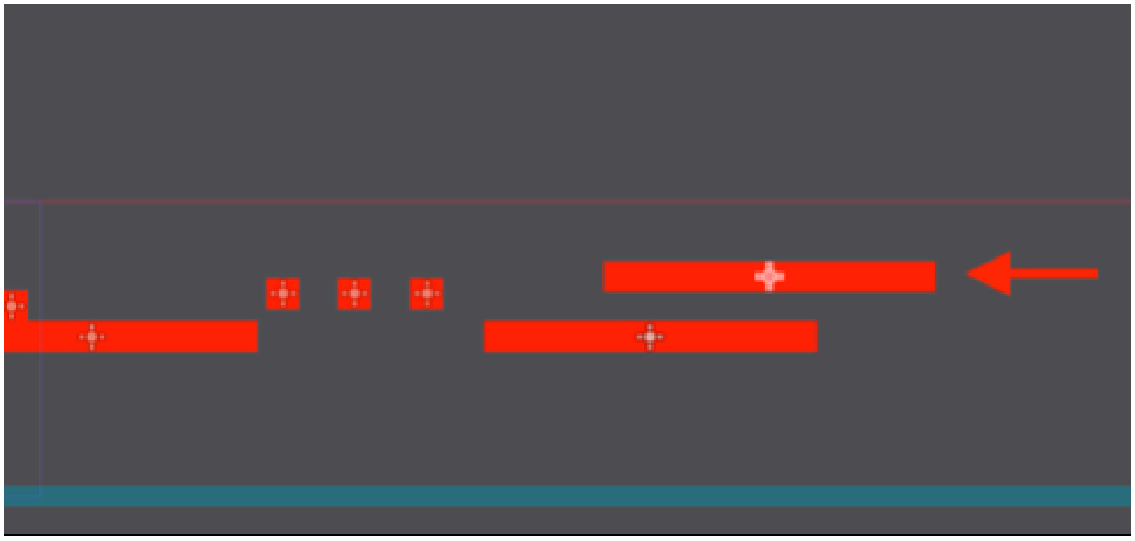


We will now create the animation for when the player character is **idle** on the ladder.

- Please create a new animation by clicking on the corresponding button and rename the new animation **idle_ladder**.
- Click on the button labeled "**Add a Texture from File**".
- In the new window, select the file **character_maleAdventurer_climbo.png**, and press the button labeled "**Open**".
- This should create one image that is now part of the animation **idle_ladder**,

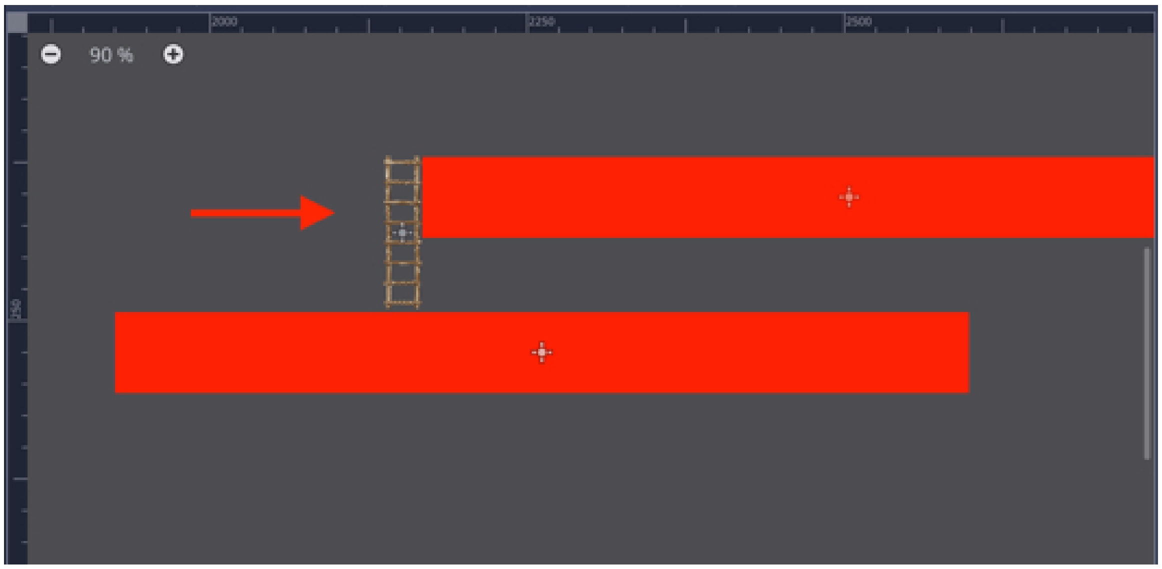
That's it, we have now all the elements set up for the player character to be able to climb a ladder; we just need to add a ladder to the main scene.

- Please switch to the main scene (**platformer.tscn**).
- Duplicate one of the platforms and place the duplicate slightly above, as illustrated in the next figure.

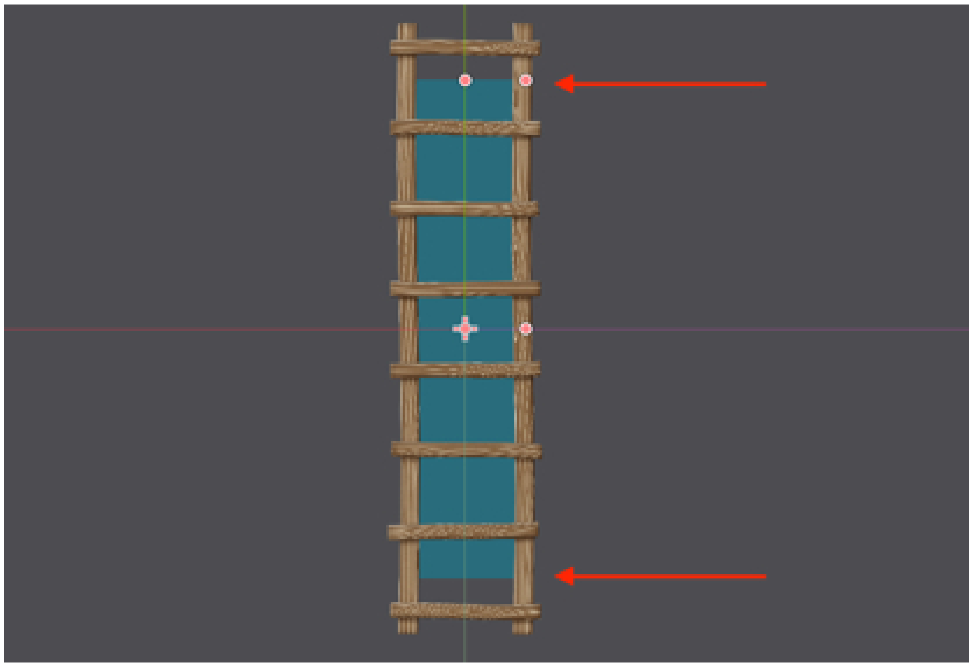


- Right-click on the node **Node2D**.
- Select the option **Instance Child Scene**.

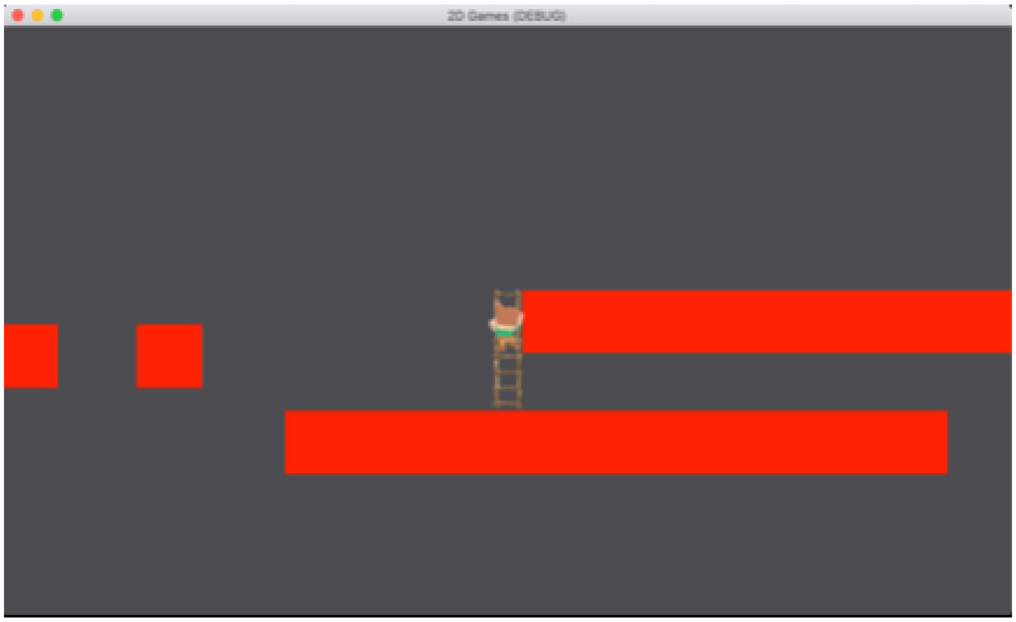
- In the new window, select the scene **ladder.tscn** and then press the button labeled "**Open**".
- This will add a new node named **ladder** to the scene.
- Please move this ladder above the platform that you have just duplicated so that it is effectively between the two platforms.



- Before we can play the scene and to ensure that the player steps off the ladder properly, you could open the scene **ladder.tscn**, select the node **Collision-Shape2D** and move the handles so that the blue area does not cover the first and last steps, as per the next figure. Please make sure that you save this scene after adding this change.



- Once this is done, you can play the scene and move the player close to the ladder, jump so that it gets on the ladder, and then climb up or down using the **up** and **down** arrow keys.



So hopefully that worked for you and this is great. In the next section, we will be adding more features that will make this game more challenging and also more aesthetically appealing.

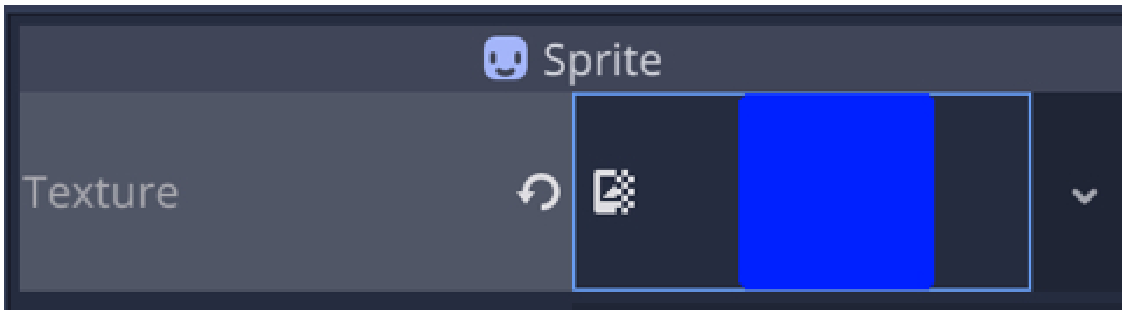
Creating magic doors

Magic doors are found in many platform games; the idea is that the player character will be able to literally teleport itself to a different part of the maze by entering what is often referred as a magic door. So, in this section we will implement this feature; to do so we will:

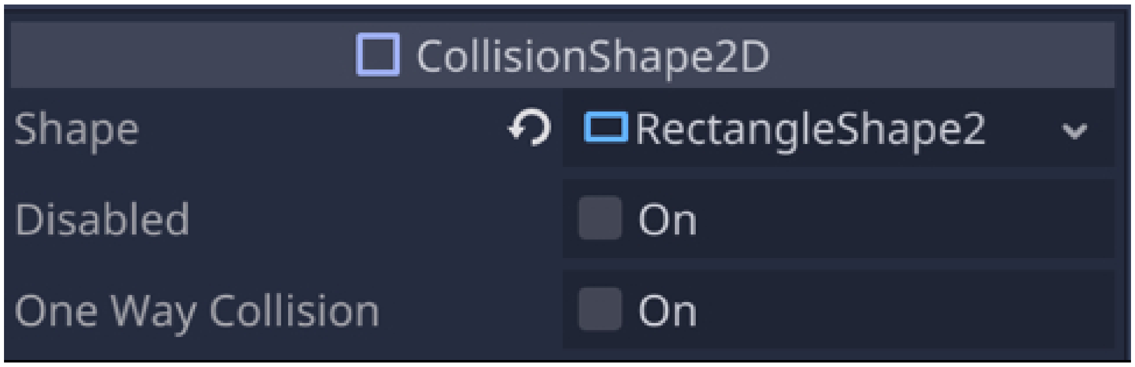
- Create a sprite that represents the magic door and associate it to an **Area2D** node so that it is possible to detect when the player enters the are defined by this door.
- Create a node that represents the location (i.e., the exit) where the player will be teleported to after entering the magic door.
- Detect when the player enters the magic door.
- Teleport the player character to the second location (i.e., the exit).

So first let's create the magic door:

- Please create a new scene (**Scene | New Scene**).
- In the new window, select the option labeled **Other**.
- In the next window, select the node type **Area2D**.
- This will create a new scene with a default node of type **Area2D** named **Area2D**.
- Rename this node **magic_door**.
- Add a child of type **CollisionShape2D** to the node **magic_door**.
- Add a child of type **Sprite** to the node **CollisionShape2D**.
- Select the node **Sprite**, and drag and drop the file **blue.jpg** from the **File System** window to the empty field to the right of the label **Texture**, in the **Inspector**.



- Select the node **CollisionShape2D**, and, using the **Inspector**, click on the downward-facing arrow to the right of the label **Shape**, in the section **CollisionShape2D**, and select the option **New RectangleShape2D** from the contextual menu.



- Change the scale properties of this node to **(0.5, 0.5)**.
- Save your scene as **magic_door.tscn (Scene | Save Scene As)**.
- Using the **File System** window, please duplicate the scene **magic_door.tscn** (i.e., **CTRL + D** or **CMD + D**) and rename the duplicate **magic_exit.tscn**.
- Open this new scene (i.e., **magic_exit.tscn**).
- Change the name of the node **magic_door** to **magic_exit**.
- Save this scene (**Scene | Save Scene**).

Now that we have created both the magic doors and the exit, it is time to detect when the player enters the magic door and to teleport it accordingly to the exit.

- Please switch to the scene **magic_door.tscn**.
- Add a new script to the node **magic_door** and rename this script **magic_door.gd**.
- Open that script.
- Add the following code at the start of the script (new code in bold).

```
extends Area2D
onready var player = get_node("../player")
onready var magic_exit = get_node("../magic_exit")
```

In the previous code, we declare two variables **player** and **magic_exit**, which are linked to the nodes **player** and **magic_exit**, respectively.

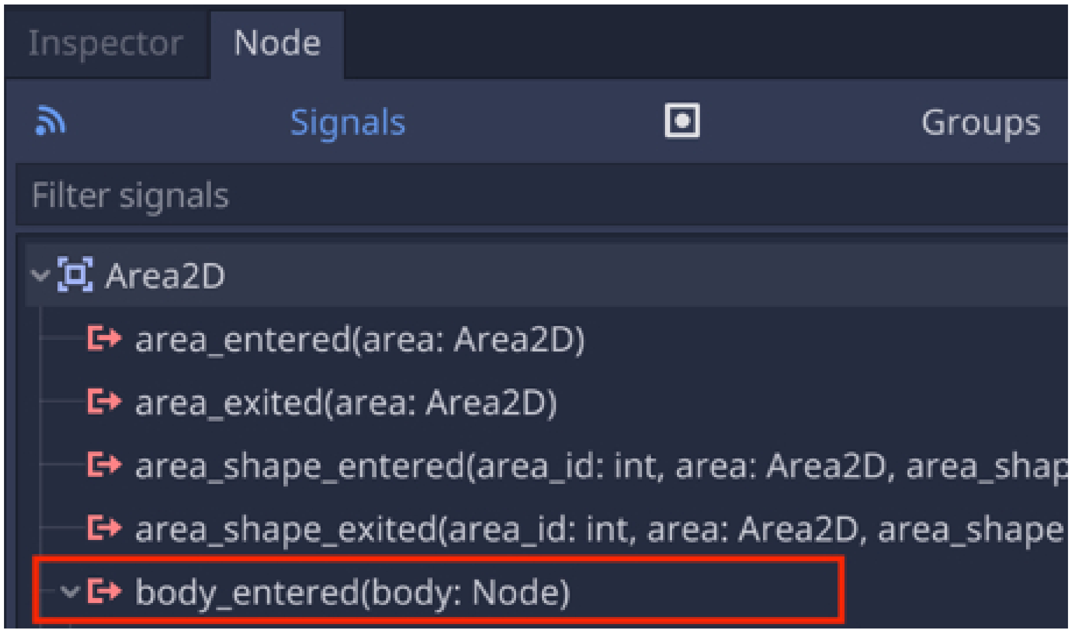
We will now create the code that will teleport the player to the exit:

- Please add the following function to the script.

```
func teleport_to_other_magic_door(body):
player.global_transform.origin = magic_exit. global_transform.origin
```

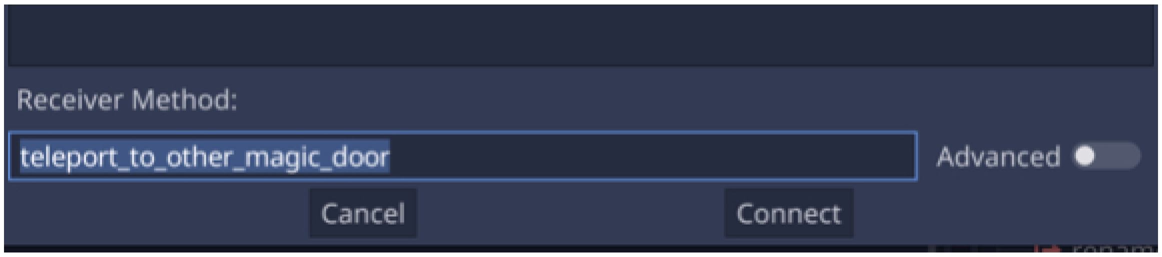
In the previous code we create a function called **teleport_to_other_magic_door**, where we change the position of the player so that it matches the position of the node **magic_exit**.

- Please save your script.
- Click on the node **magic_door**.
- Select the tab **Node | Signals**.
- Double-click on the event **body_entered**.



area_entered(area: Area2D)
area_exited(area: Area2D)
area_shape_entered(area_id: int, area: Area2D, area_shap
area_shape_exited(area_id: int, area: Area2D, area_shape
body_entered(body: Node)

- In the new window, select the node **magic_door**.
- Type the text **teleport_to_other_magic_door** in the **Receiver Method** field.



Receiver Method: teleport_to_other_magic_door

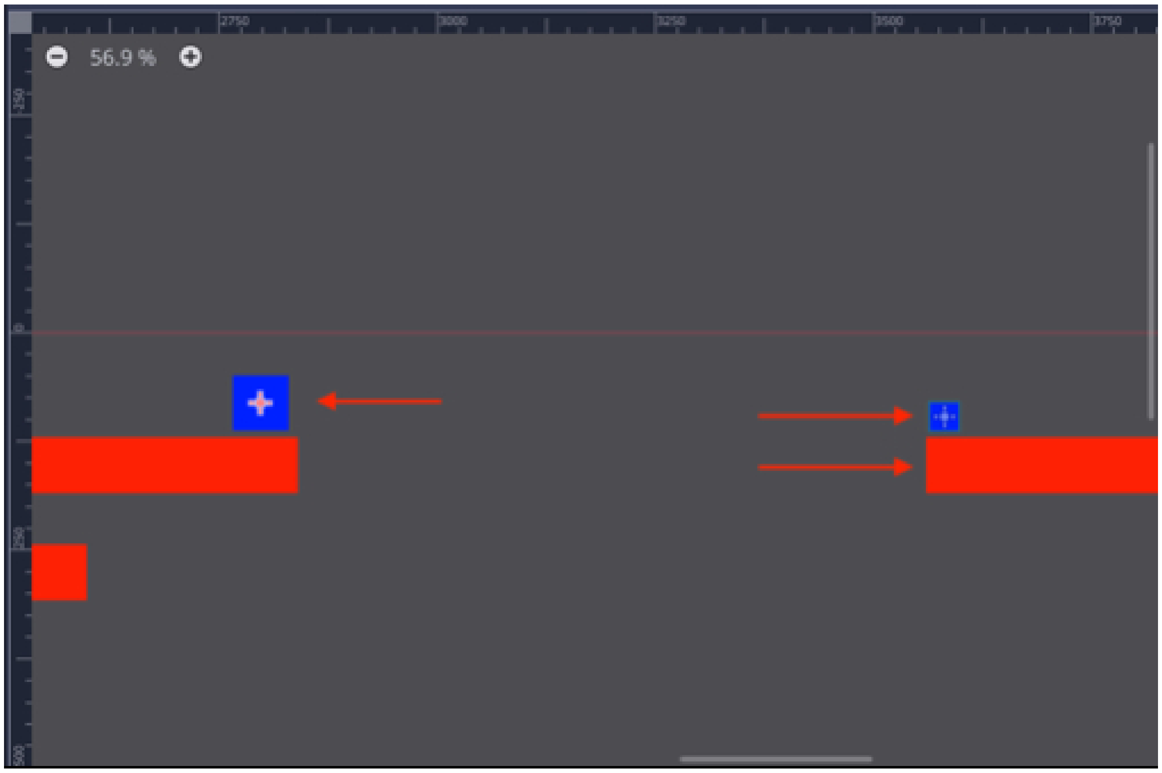
- Press the button labeled "**Connect**".
- Switch to the scene **magic_exit**.
- Add a new script to the node **magic_exit** and rename the script **magic_exit**.
- Add the following code to the script (new code in bold).

```
func _ready():
hide()
```

In the previous code, we just ensure that the **magic_exit** node is hidden as soon as the game starts.

That's it: we have configured the magic door and the exit; we just need to add them to the main scene.

- Please switch to the main scene (i.e., **platformer.tscn**).
- Right-click on the node **Node2D**.
- Select the option **Instance Child Scene**.
- Select the scene **magic_door** from the list and click on the button labeled **Create**.
- Repeat the last steps to add a node based on the scene **magic_exit**.
- At this stage, you should have two additional nodes to your scene: **magic_-door** and **magic_exit**.
- Create a new platform to the right of the scene, and move the magic door in locations similar to the ones described in the next figure, so that the right platform is only accessible by using the magic door.
- You may need to rescale both doors slightly.



- You can now save and play the scene, and check that after entering the magic door that the player character is teleported to the exit (i.e., the other platform).

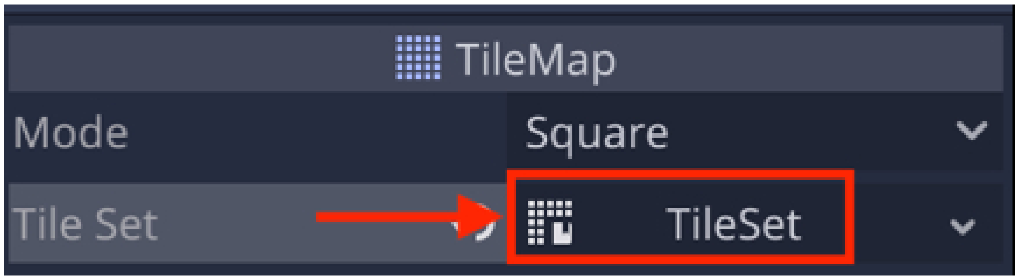
Creating a scene using tilesets

While we have been using basic shapes to create the platforms in our game, there is a feature available in Godot that makes it easier to create platforms and other objects that make up a 2D game: tilesets. Using tilesets, you can create a level just as you would paint an image in a graphics editor, and this is what we will do in this section. So the steps involved in using a tileset will involve:

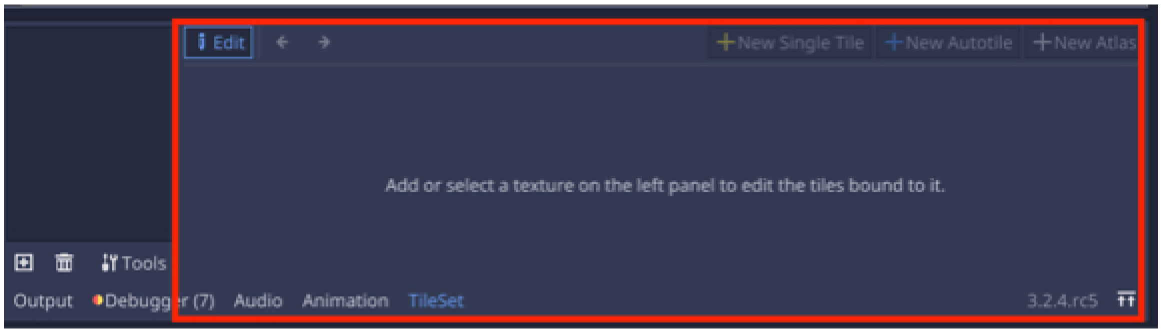
- Identifying and selecting an image that includes the tiles that we want to use for our game.
- Selecting part of this image and saving it as a tileset to be used later.
- When applicable, generating colliders for this tileset.
- Using the tileset that you have created earlier to generate platforms in the game.

So, first, let's create tiles and tilesets:

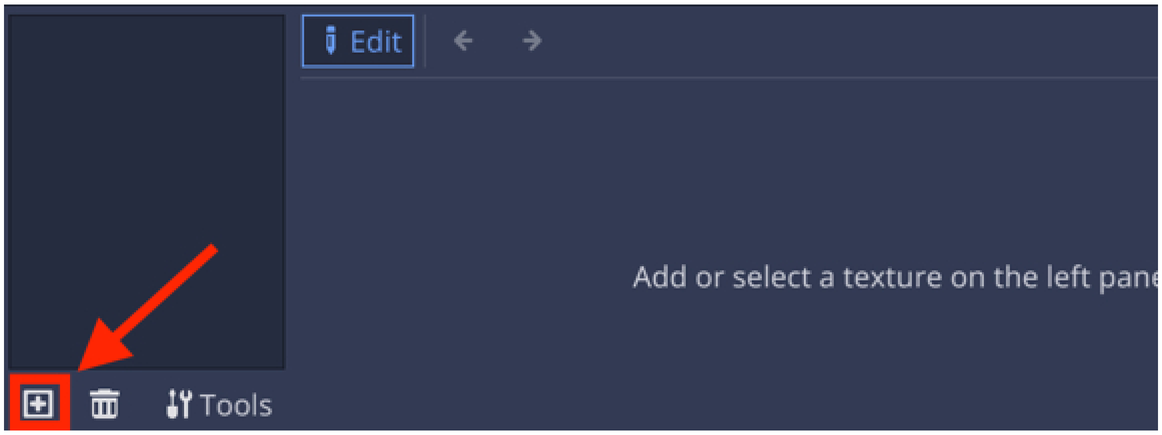
- Using the **File System** window, duplicate the scene **platformer.tscn** and re-name the duplicate **plateformer_with_tiles.tscn**.
- Open the scene **platformer_with_tiles.tscn**.
- Remove all the nodes except the root node (i.e., keep the node **Node2D**).
- Add a child of type **TileMap** to the node **Node2D**.
- Select the new node **TileMap**.
- Using the **Inspector**, scroll down to the section **TileMap**, click on the down-ward- fac- ing arrow to the right of the label **TileSet**, and select the option **New TileSet** from the contextual menu.
- Once this is done, please click on the item labeled **TileSet** as per the next fig-ure.



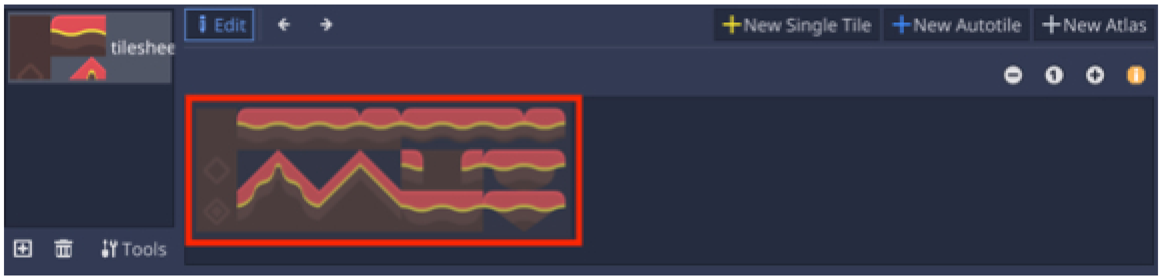
- This will open a new **TileSet** window at the bottom of the screen.



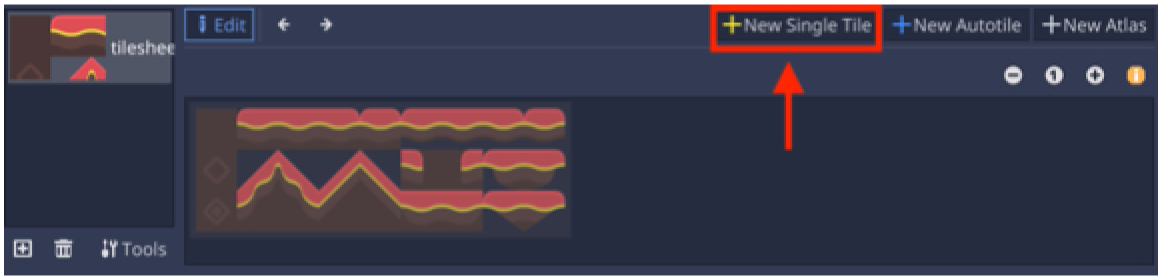
- Click on the button labeled **Add Texture to TileSet**, as per the next figure.



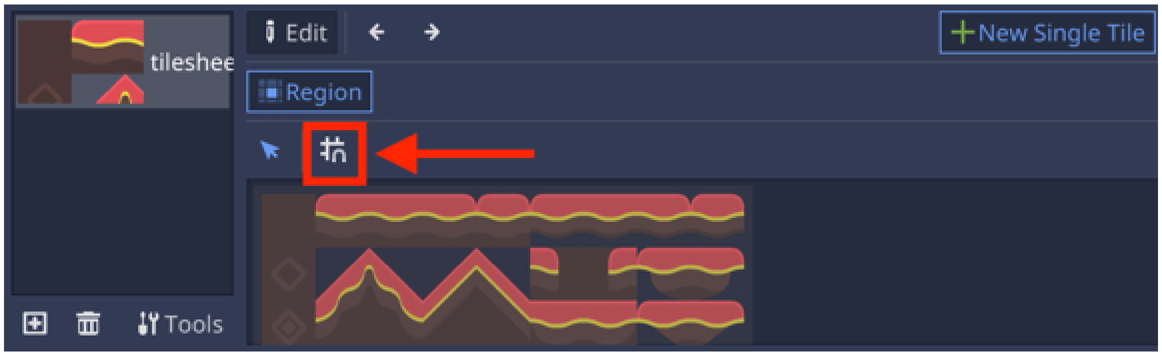
- In the new window, please select the file **tilsesheet.png** and then press **Open**.
- This should add a new tile sheet to your scene.



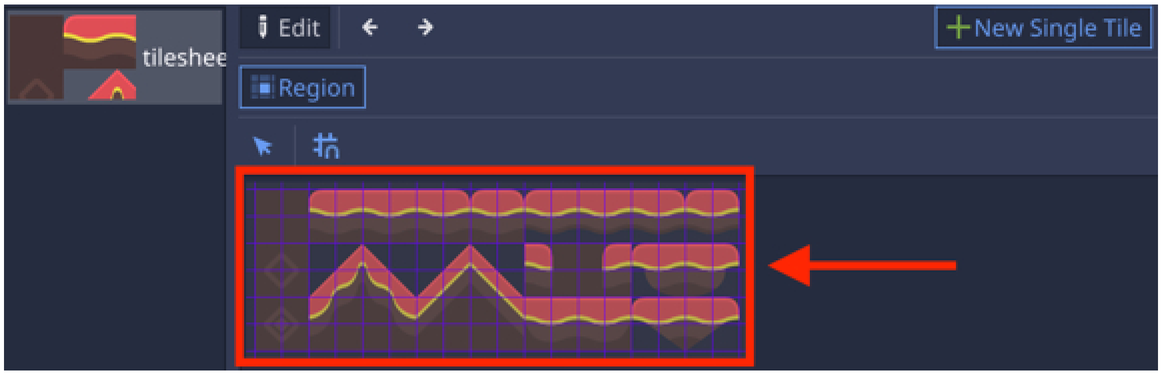
- Click on the button labeled “**New Single Tile**”, as per the next figure.



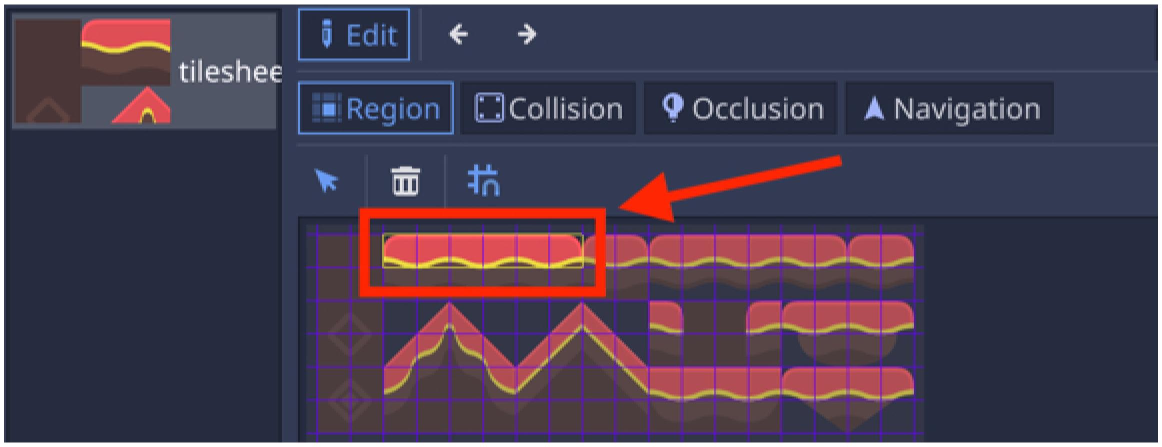
- Click on the button labeled “**Enable Snap and Show Grid**”, as per the next figure.



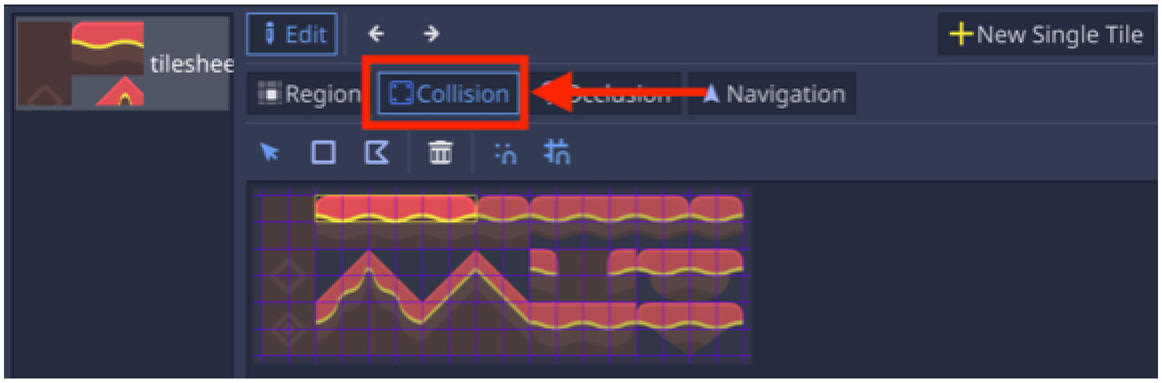
- You should now see a grid over the existing tile sheet.



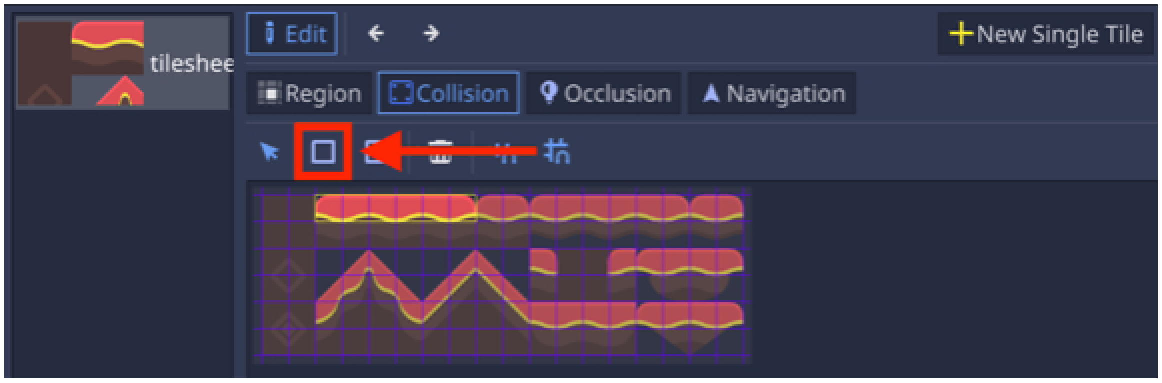
- Please drag and drop your mouse over the tile sheet so as to select the third, fourth, fifth, and seventh tiles from the top row, as illustrated in the next figure.



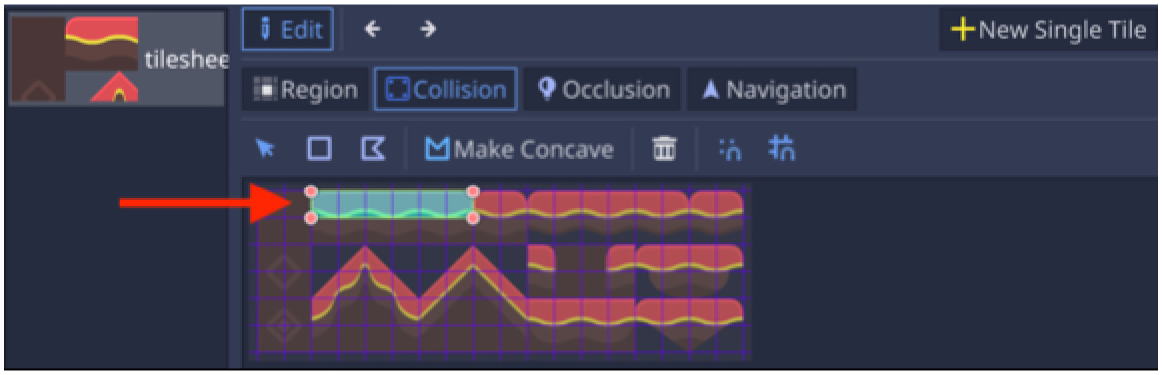
- Clicking on the **Collision** tab so that you can apply colliders to this tileset, as illustrated in the next figure.



- Click on the button labeled “**New Rectangle**” so that we can define a collision area for this tileset, as illustrated in the next figure.

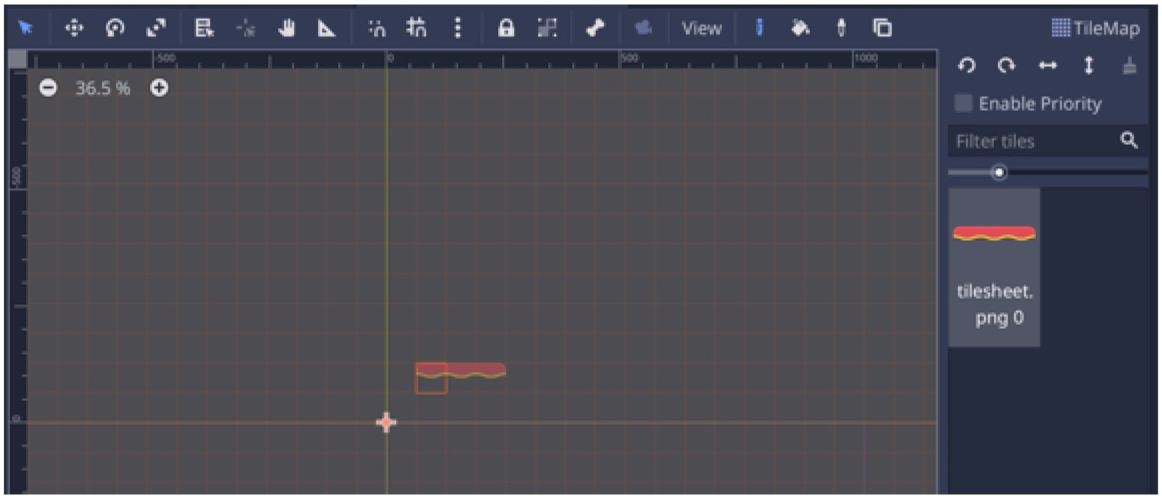


- Click in the top left corner of the area that you have defined earlier and drag and drop your mouse so as to create a rectangle that covers the corresponding area. As per the next figure.



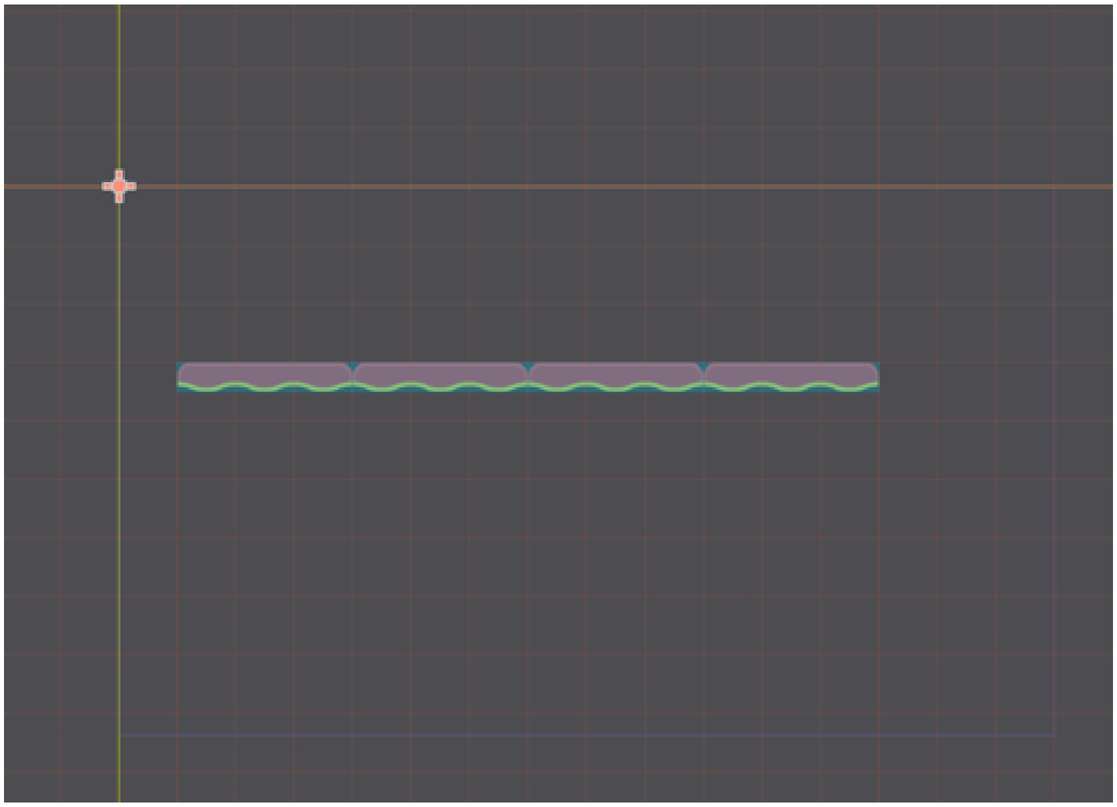
At this stage we are ready to add our first tile.

- Using the **Inspector**, click on the node **Node2D** and then click on the node **TileMap** in the **Scene Tree**, you should see that a grid is now displayed in the viewport and also that as you move your mouse over it, a tileset (the one you have defined earlier) appears to the right of your mouse.



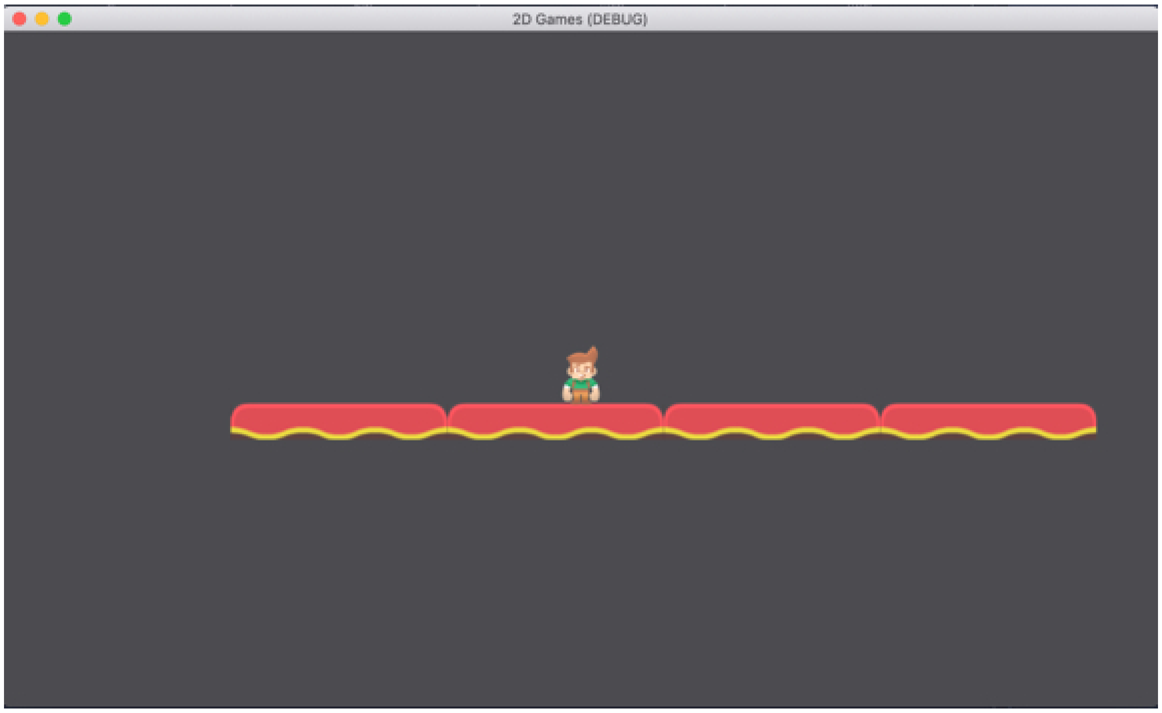
- You can now apply this tileset by left-clicking within the viewport several times.

You can also remove a tile by pressing CTRL and the left mouse button on that same tile.



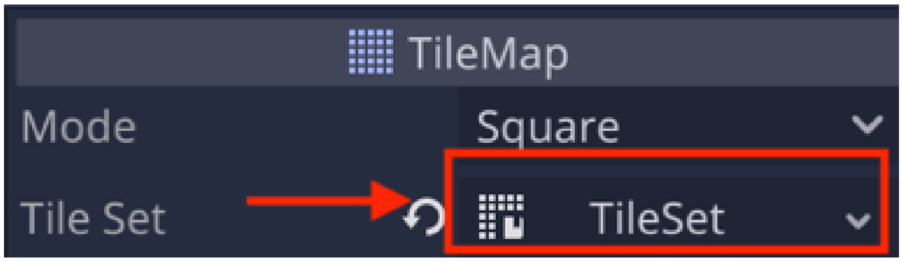
- Right-click on the node **Node2D**, select the option "**Instance Child Scene**".
- In the new window, select the scene **player_character.tscn** and press the button labeled "**Open**".
- This will add a new node called **player**.
- Please move this node just above the tiles that you have just created and test the scene to check that the player can walk and jump on these tiles.

When you add tilesets (i.e., made of one or more tiles), for tilesets with one or two tiles you can press the **SHIFT** key as you press the left mouse button; for longer tilesets (i.e., with three or more tiles), it may be more useful to click once at a time.



Once this is done we could add more tiles:

- In the **Scene Tree** tab, please select the node **TileMap**.
- In the **Inspector**, click on the item labeled **Tileset** to the right of the label **Tile-Set** in the section **TileMap**.

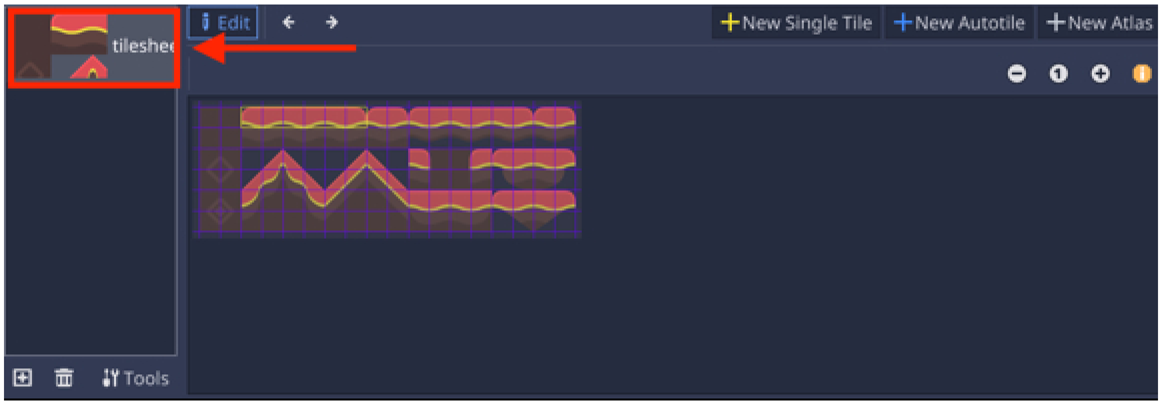


TileMap

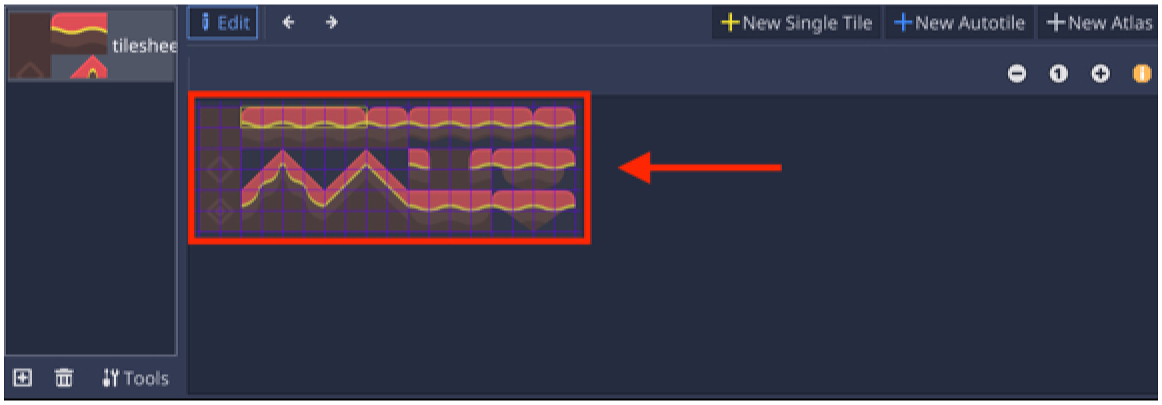
Mode: Square

Tile Set: TileSet

- This will open the **TileSet** window.
- Please click on the tile sheet that we have already opened, as per the next figure.

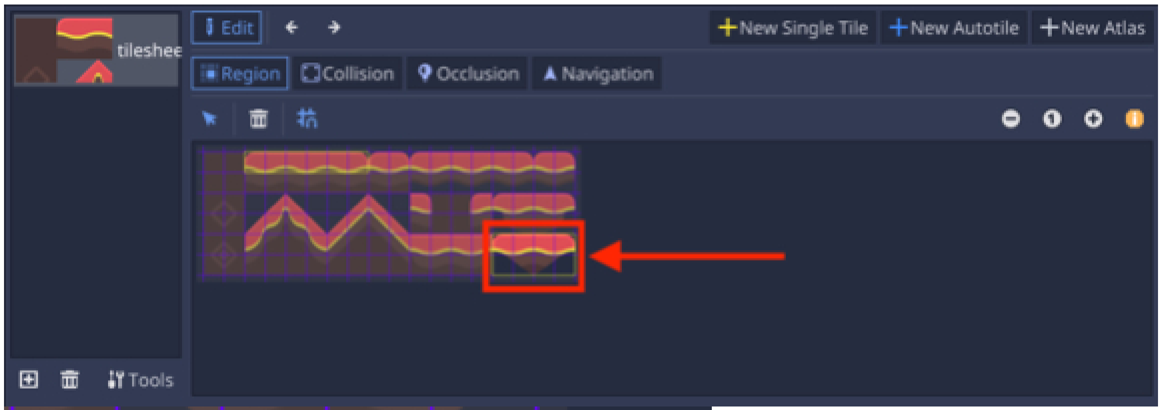


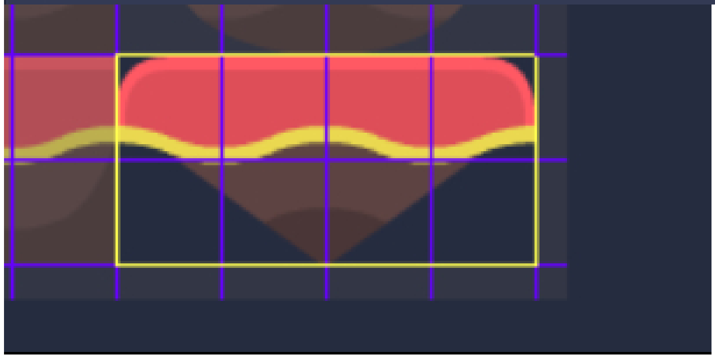
- This should show your tile sheet along with a grid.



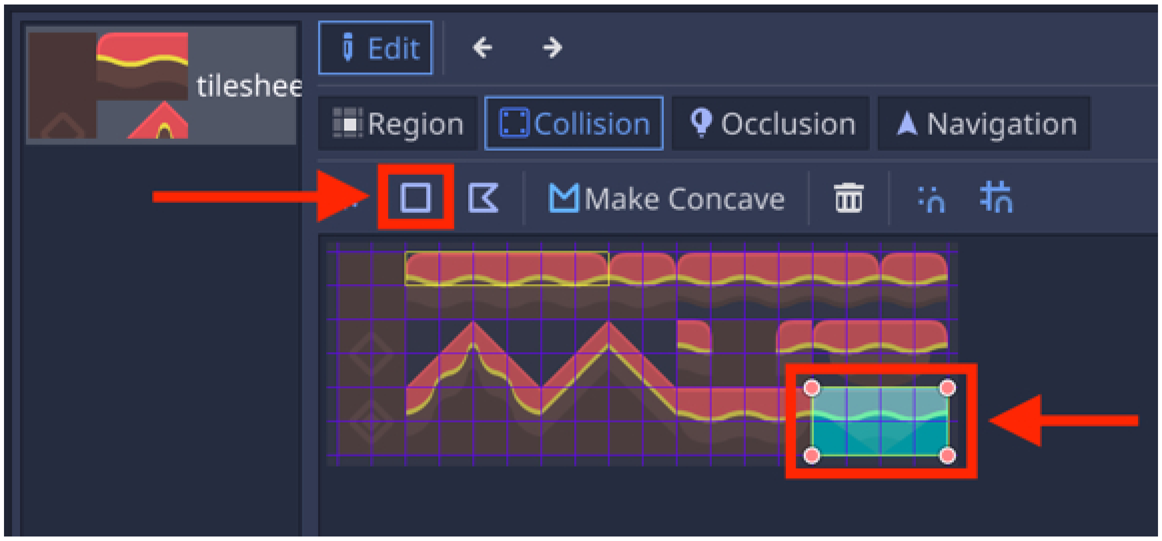
As we have done before, we will select part of this tile sheet to create new items (i.e., tilesets) to be added to the scene.

- Please click on the button labeled "**New Single Tile**".
- Drag and drop your mouse over the area highlighted in the next figures.

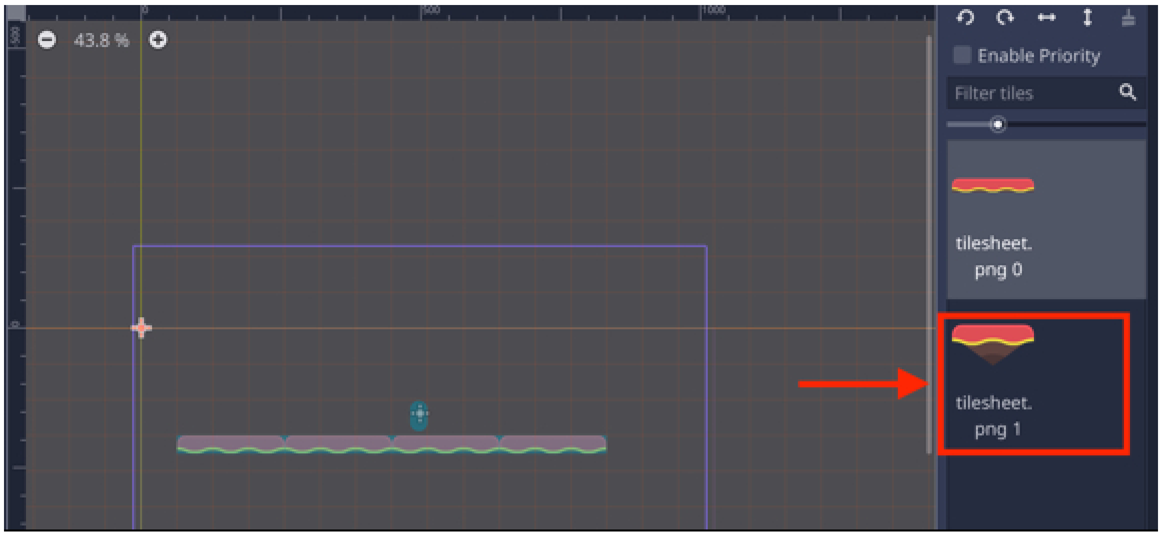




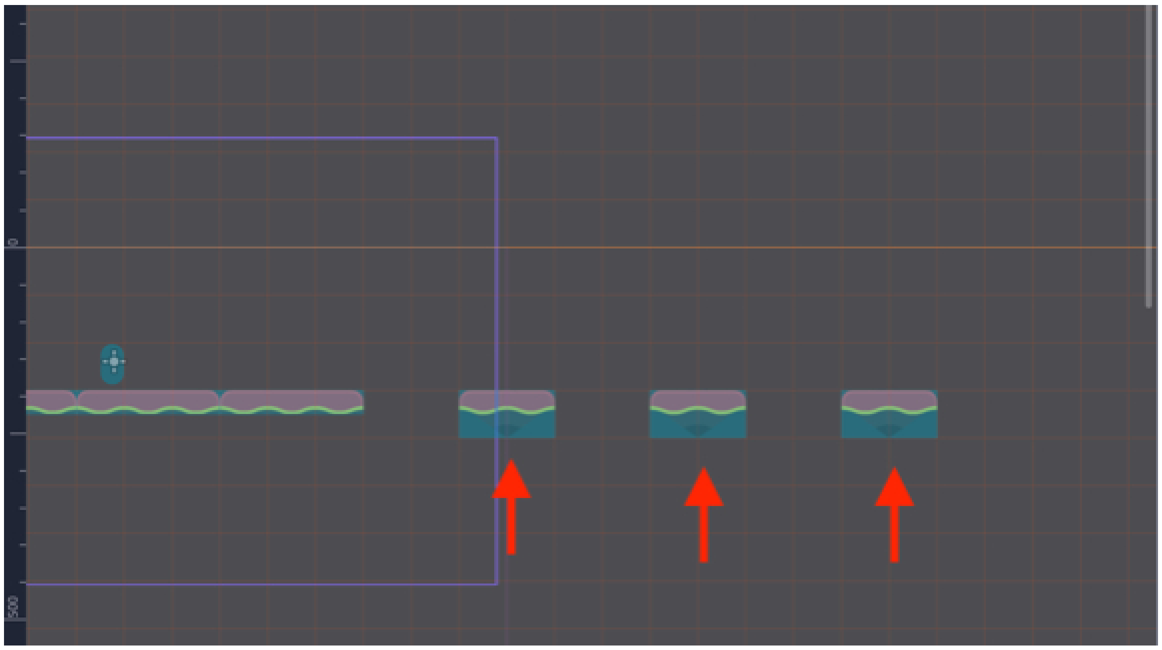
- Once this is done, switch to the **Collision** tab. Select the "**New Rectangle**" button, and drag your mouse over the same area so as to create a rectangle that covers this area.



- Using the **Inspector**, click on the node **Node2D** and then click on the node **TileMap**, and you should see that a grid is now displayed in the viewport and also that as you move your mouse over it, a tileset (the one you have defined earlier) appears to the right of your mouse.
- You should now see, to the right of the viewport, two different types of tiles that can be added to the scene; these represent the two tilesets that you have created earlier. Please, select (i.e., click on) the second tileset (the last one that you created).

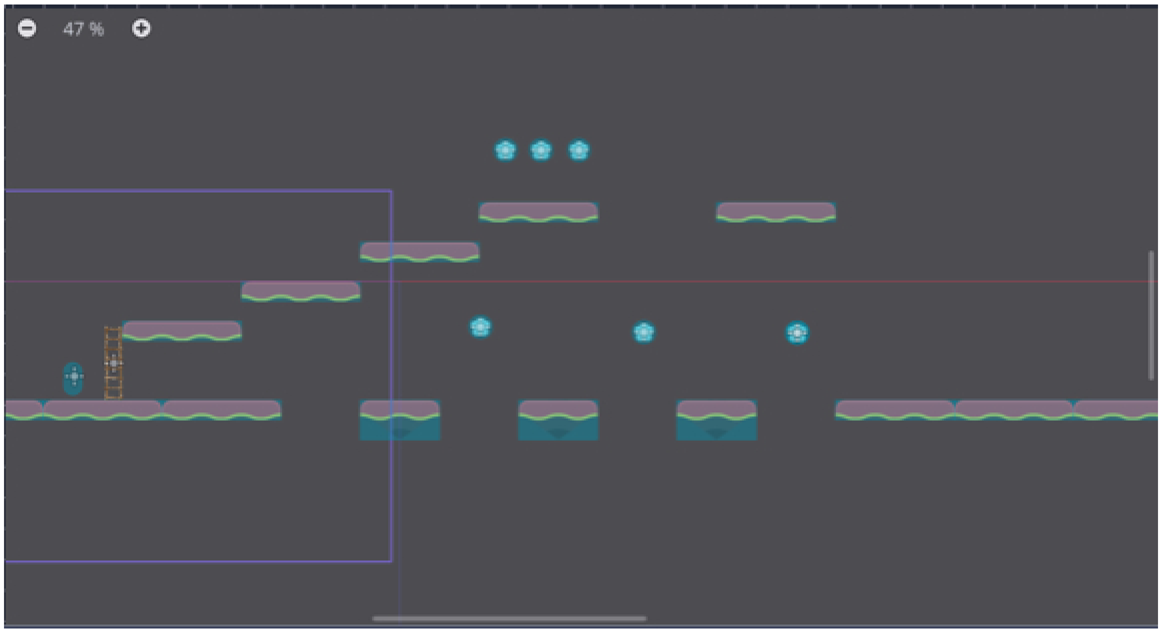


- You can now click to the right of the platforms that you have already created to create three (or more) standalone platforms based on your new tileset.

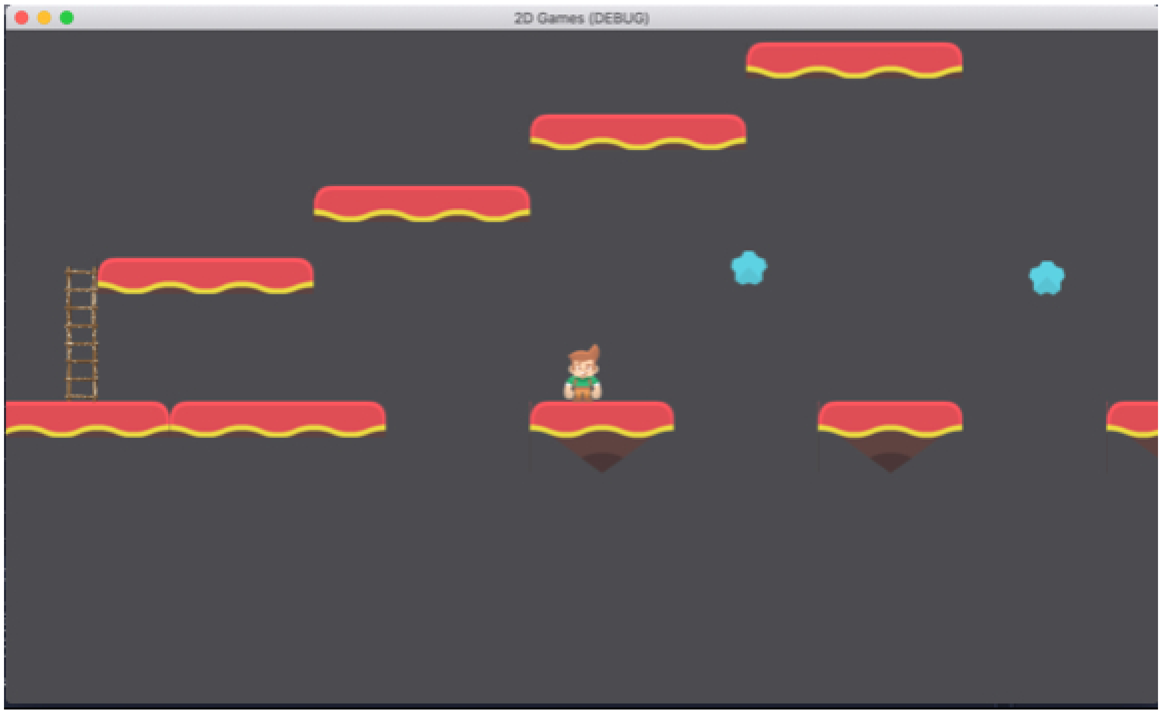


- Once this is done, you can save your scene and test it, moving the player character so that it jumps over the three new platforms that you have just created.

You can finish the level, by adding platforms to the right, additional platforms that can be accessed with a ladder, a dead zone, secret doors, or diamonds to be collected using a layout similar to the one illustrated in the next figure.



- You can now test your scene.



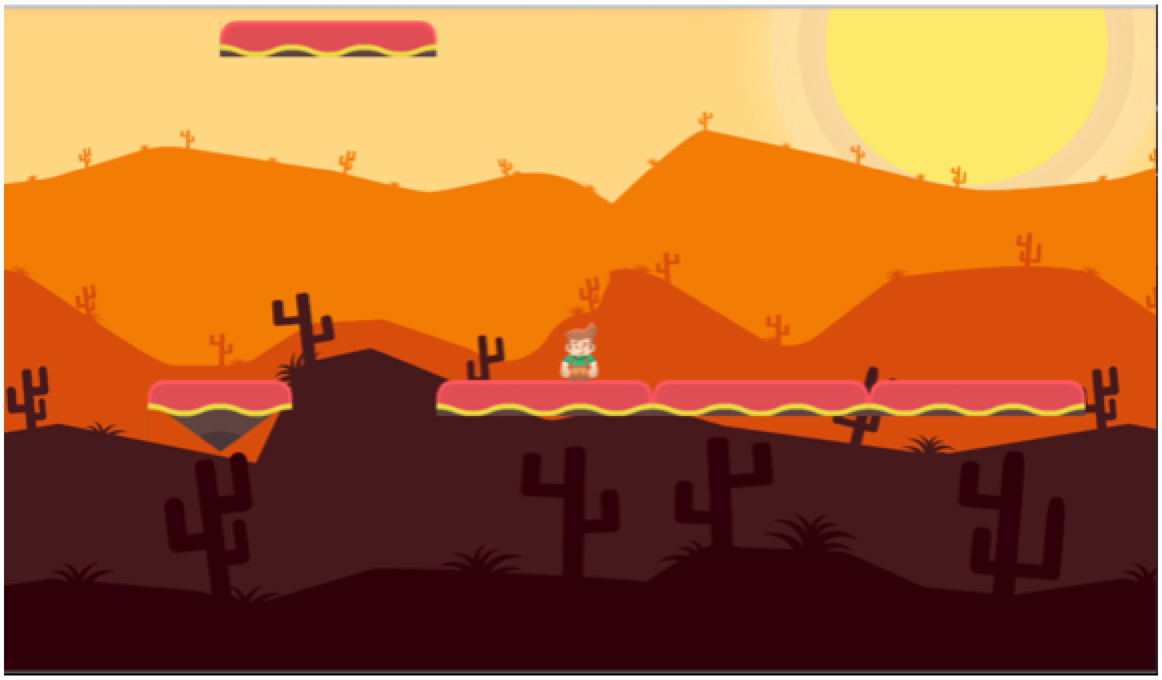
You can of course create other tilesets for your game, and the general approach

to creating tilesets will be similar to what we have done in this section.

Creating a parallax effect

In this section we will create a parallax effect to add some visual effects to our game; a parallax effect is based on the idea that objects further away from the camera will look like they move slower than the ones closer to the camera. So for example, in a platformer game, if the environment is made of trees and mountains further away, as your player character progresses, the trees will look like they move faster than the mountains.

After using a parallax effect for our platformer, the final layout for our game will look like the following.

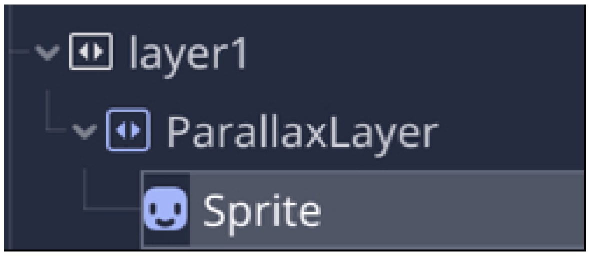


This will consist of several layers that will move at different speeds, hence giving the illusion of depth through a parallax effect.

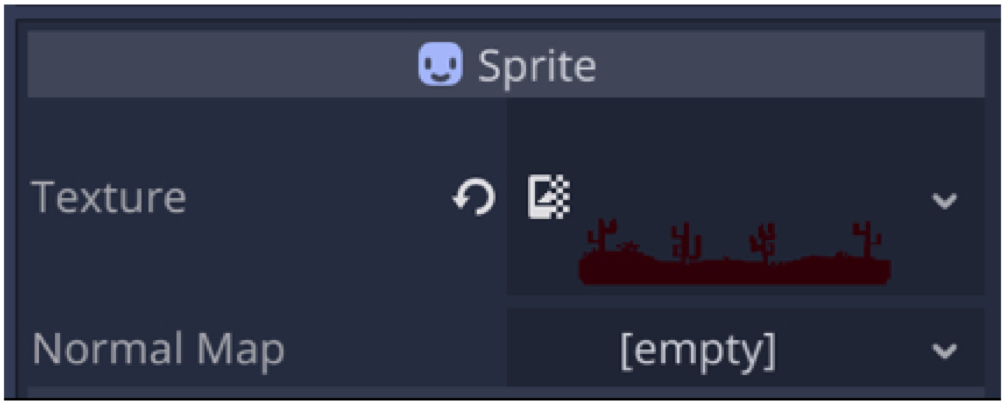
So let's start creating this parallax effect:

- Please add a node of type **ParallaxBackground** to the node **Node2D** and rename this new node **layer1**.
- Add a child node of type **ParallaxLayer** to the node **layer1**.

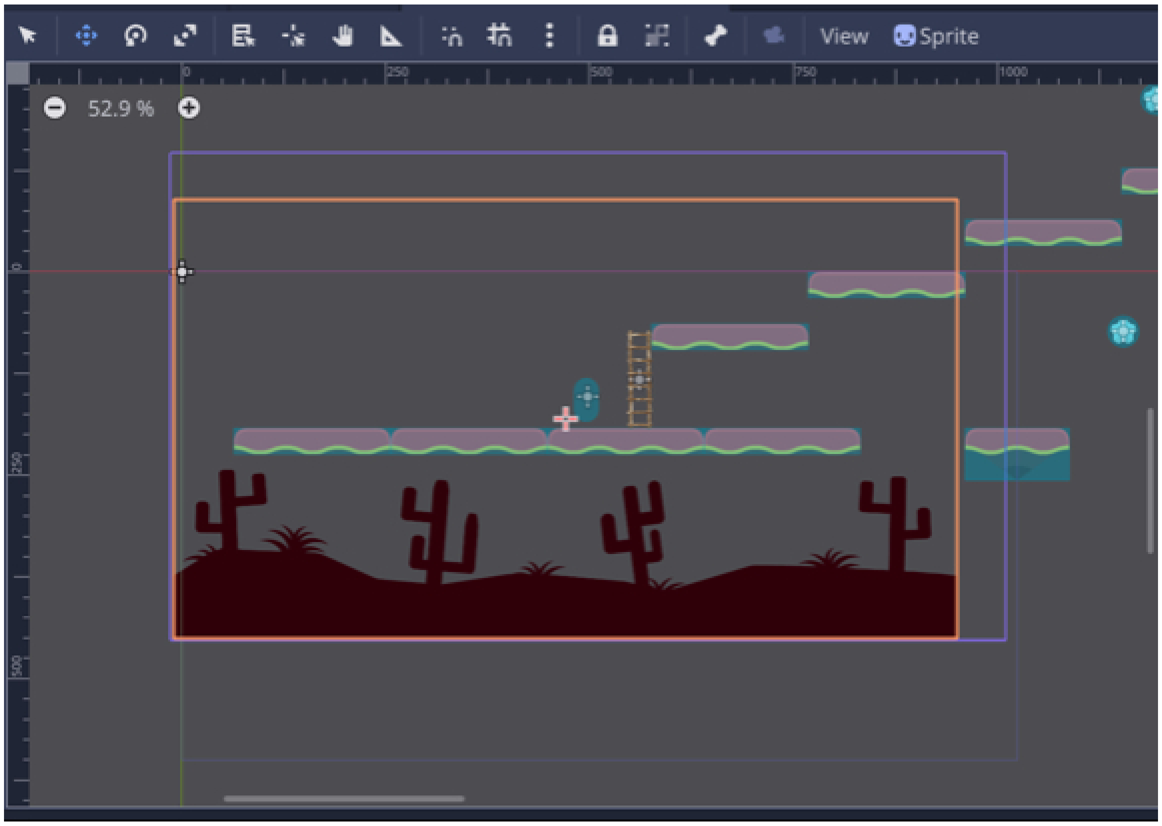
- Add a child node of type **Sprite** to the node **ParallaxLayer**.



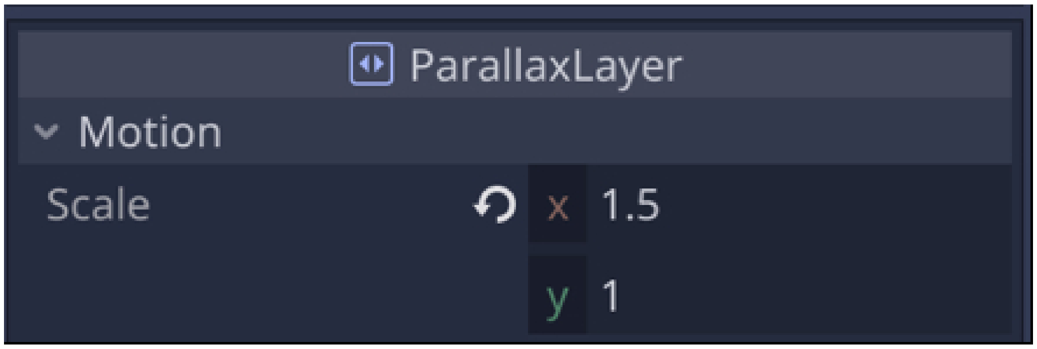
- Click on the **Sprite** node that you have just created.
- Drag and drop the picture **layer1.png** from the **File System** (within the folder **parallax**) to the **Inspector** in the empty field to the right of the label **Texture** in the section **Sprite**.



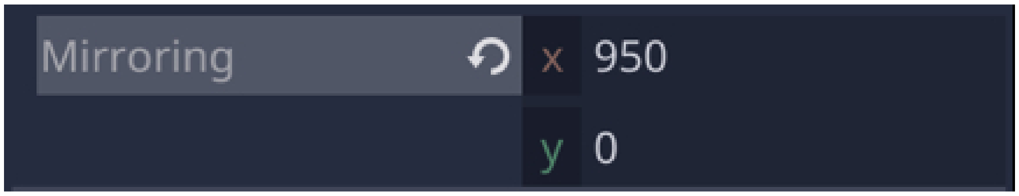
- Using the **Inspector**, change the **Scale** property of this node to **(0.5, 0.5)**.
- Using the viewport, move this **Sprite** node so that it occupies about three quarters of the visible area, or just so that the cactuses appear just below the player and the first set of platforms, as per the next figure.



- Click on the node **ParallaxLayer**, and, using the **Inspector**, scroll down to the section called **Motion**, and change the **Motion | Scale** property (in the section **ParalllaxLayer | Motion**) to **(1.5, 1)**.

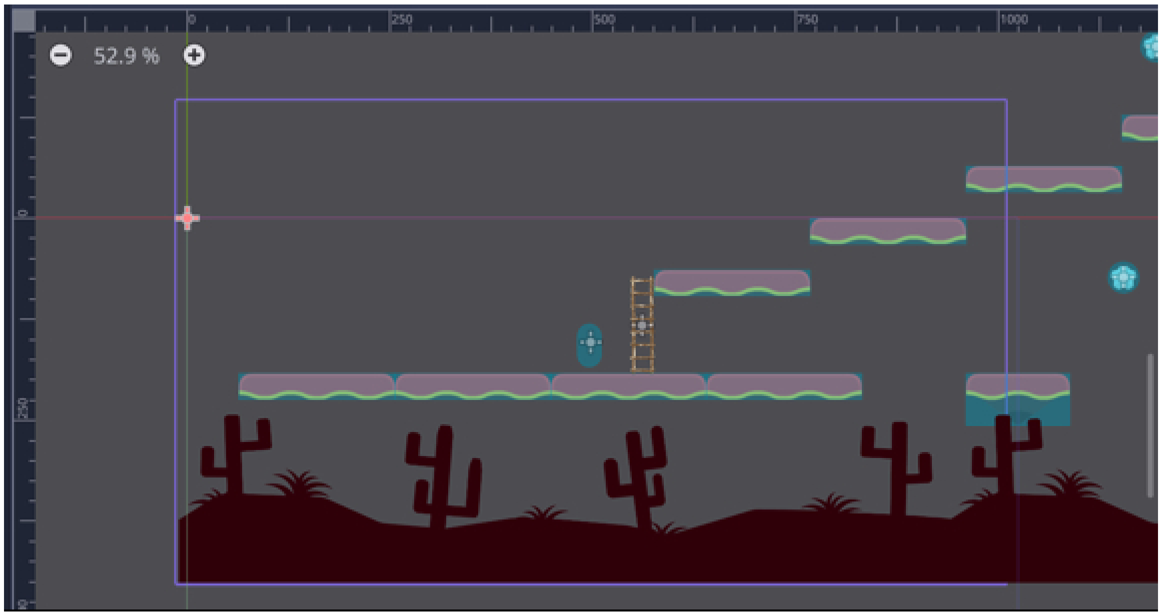

ParallaxLayer
Motion
Scale x 1.5 y 1

- Finally, set the attribute **mirroring** to **(950, 0)**.



Mirroring x 950 y 0

- You should then see that the sprite is repeated horizontally.

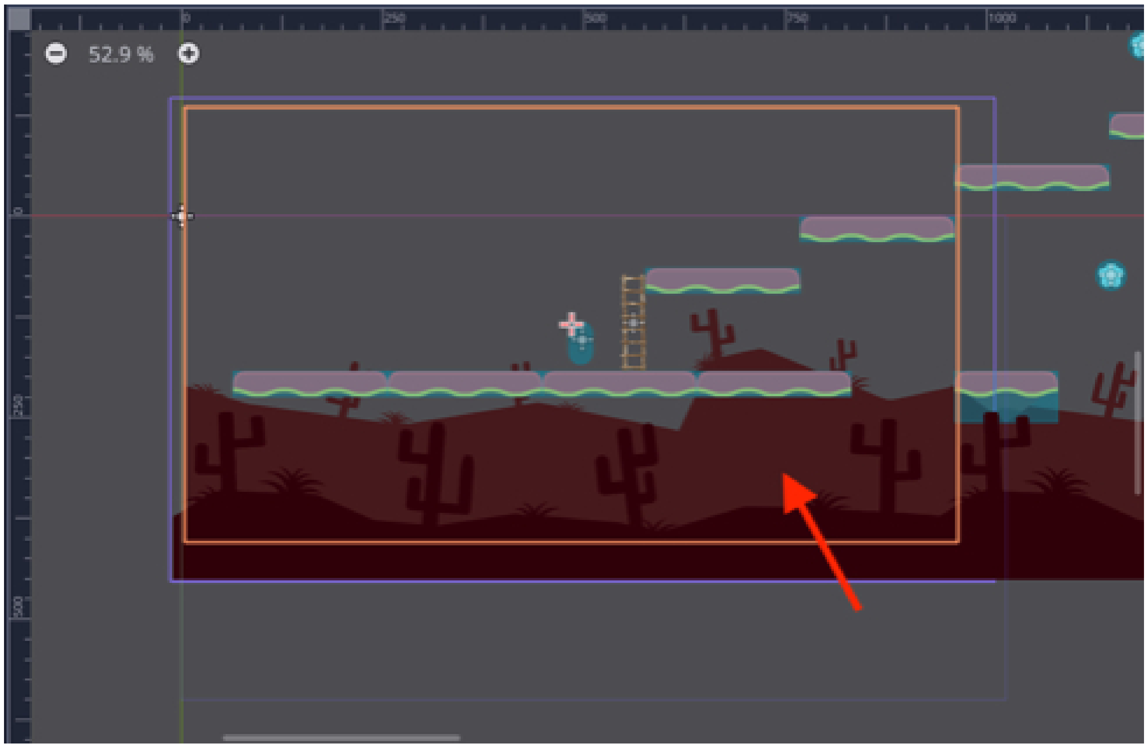


Last but not least, select the node **layer1**, and using the Inspector set the attribute **Layer** to **0**.

We have just set up our first layer, the one that is the closest to the player; as you play and test the scene, you should see that the sprite for this layer moves as the player progresses to the right or left of the scene. For this parallax effect to be complete, we just need to set up the other six layers.

- Please duplicate the node **layer1** (**CTRL + D**), this will create a new layer named **layer2**.
- Click on the node **Sprite** that is a child of the node **layer2**, set its sprite with the image called **layer2**, and move this **Sprite** up so that it is above the

previous image, as per the next figure.



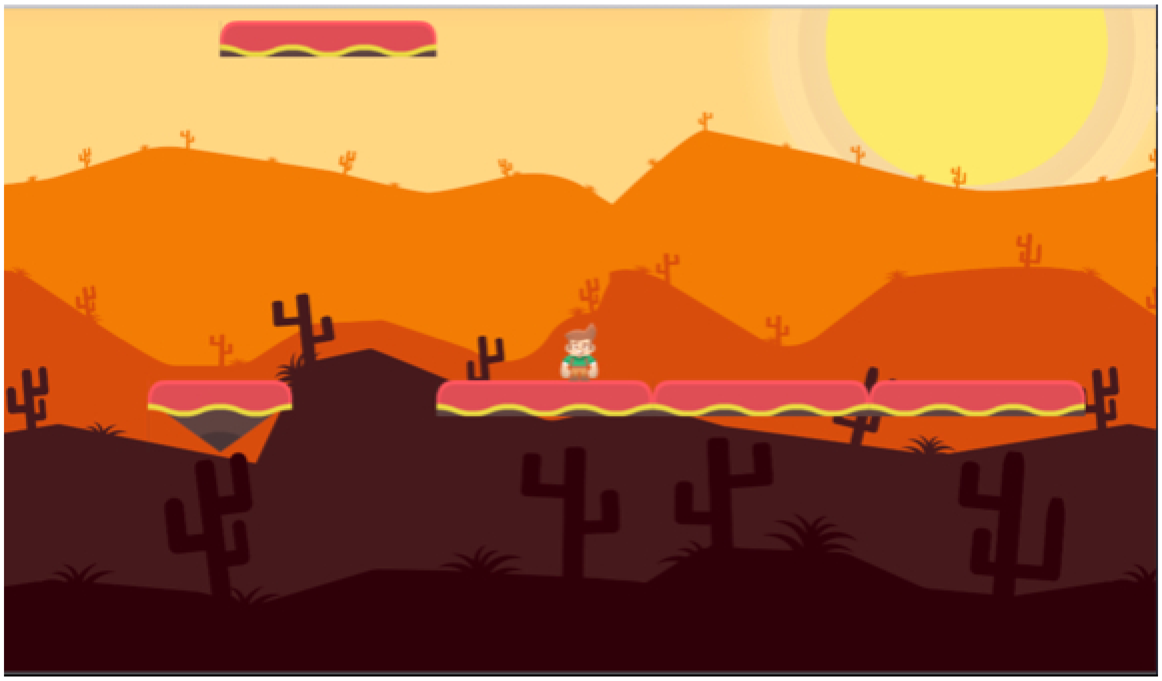
- Once this is done, click on the node **ParallaxLayer** that is a child of the node **layer2**, scroll down to the section called **Motion**, and change its **Scale** attribute to **(1, 1)**.
- Finally, select the node **layer2** and, using the **Inspector**, change its **Layer** property to **-10**.

You can repeat this process to create four additional layers with the following attributes:

- **layer3**: **layer** attribute = **-20**, **ParallaxLayer** node**: Motion | Scale** attribute = **(0.5, 1)**, **Sprite** node: **Sprite** = **Layer3.png**.
- **layer4**: **layer** attribute = **-30**, **ParallaxLayer** node**: Motion | Scale** attribute = **(0.4, 1)**, **Sprite** node: **Sprite** = **Layer4.png**.

- **layer5**: **layer** attribute = **-40**, **ParallaxLayer** node**: Motion | Scale** attribute = **(0.3, 1)**, Sprite node: **Sprite** = **Mountains.png**.
- **layer6**: **layer** attribute = **-50**, **ParallaxLayer** node**: Motion | Scale** attribute = **(0.1, 1)** and **Mirroring** = **(0, 0)**, **Sprite** node: **Sprite** = **Sun.png**.

So at this stage, your scene should look like the following figure.



You can now test your scene and check the parallax effect.

LEVEL ROUNDUP

In this chapter, we have learned to create a complete platformer with many of the features commonly found in this genre, including an animated character, platforms, items to collect, ladders to climb, magic doors. Along the way, you have also added sound effects to your game, and improved its visual appeal by using tilesets and a parallax effect.

Quiz

Now, let's check your knowledge! Please answer the following questions (the answers are on the next page) or specify whether they are TRUE or FALSE.

1. An animated sprite can include one or more images.
2. An **AudioStreamPlayer** can be used to plays sound effects in your game.
3. When an **Area2D** node is defined, it can be associated with events triggered when another node enters or exits this area.
4. It is possible to duplicate a scene using **CTRL + D**.
5. It is possible to duplicate a node using **CTRL + D**.
6. Using a parallax effect, objects closer to the player character will appear to move slower than those further away.
7. To create tilesets and tiles in a scene, it is necessary to create a node of type **TileMap**.
8. A **ParralaxLayer** node can be used to set the relative speed of the layer in order to create the visual parallax effect.
9. A **ParralaxBackground** node can be used to define the depth of the image used for the parallax.
10. The following code will play the clip associated with an **AudioStreamPlayer** node

```
Audio_stream_player.play()
```

Answers to the Quiz

Now, let's check your knowledge! Please answer the following questions (the answers are on the next page) or specify whether they are TRUE or FALSE.

1. TRUE.
2. TRUE.
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. TRUE.
8. TRUE.
9. TRUE.
10. TRUE.

Challenge 1

For this chapter, you can improve and extend your existing scenes by adding the following features:

- A score displayed on screen.
- Checkpoints that add 50 points to the score when the player character passes them.
- Dangerous items that will move the player back to its original position upon collision.

Chapter 6: Frequently Asked Questions

This chapter provides answers to the most frequently asked questions about the features that we have covered in this book.

NETWORKING

What is the difference between a client and a server?

In computer science terms, a server provides services that may be requested by clients. So usually clients would connect to the server to avail of a service. In terms of web development, a server will deliver a page or execute a programme (e.g., script) and provide the output to a client (e.g., through a client browser). In a networked game, and for the particular case of a FPS, the server will host the entire game, and the client will connect to the server to access the game, and to also avail of any update in the game, so that all players have a synchronized version of the game world; in other words, we want all clients to see the same version of the world in which they play. So the client and the server, once connected, will exchange information. Because of the amount of information involved and possible bandwidth challenges, there is a balance to be reached between updating the server regularly and also keeping running the game smoothly with no or little delay.

What is the difference between a public and a private address?

Servers that use public address can be accessed directly from the Internet because there is no firewall between the client and the server. A firewall would typically restrict access to the devices on its network (e.g., clients or servers) and hide the actual IP address of these also. However, a server can also use a private address; this is usually the case when the server is installed behind a firewall. By default, in Godot, if the server is hosted on your own computer, the address of the server will be, in most cases, 127.0.0.1, which, in computer terms, usually defines your own computer (i.e., localhost).

How do I run a PHP script?

A PHP scrip needs to be executed by a server, so it needs to be hosted on a server and then accessed through its address (i.e., url). These scripts are usually saved in the public folder of the webserver.

How do I access a database using PHP?

To access a database in PHP you need a server name (or address), a user name, a password, the name of a database, along with the name of the table that you need to access. In the next code snippet, you should be able to see how a connec tion is initiated in PHP through a function called **connect**; the function **mysql_-connect** is employed to establish a connection to a server, based on settings such as: the name of the server, the name of the database to be accessed, a user name, and its password.

```
function connect()
{
$host="localhost" ;
$database="mydatabase";
$user="user1";
$password = "mypassword";
$error = "Cant connect";
$con = mysqli_connect($host,$user,$password);
mysqli_select_db($con, $database) or die("Unableto connect to database");
}
```

Do I need to host my pages on a website to run PHP scripts or to create a database?

While an online server is a common solution, you can also download and install AMP and run the server on your own computer for testing purposes; AMP packages (MAMP or WAMP for example), include a webserver, as well as MySQL, so that you can also create and access a database hosted on your own computer, if need be.

In terms of accessing a database, what does the term localhost mean?

Localhost, when used within a PHP file, refers to the server where the PHP script is stored; so if the script is stored on your computer, and you have a local server, then the localhost will be the webserver on your computer; however, if these files are hosted on a remote server, then this remote server will be the localhost in this case.

What is the link between Godot, PHP, and MYSQL?

When accessing a MYSQL database through Godot, the process is usually as follows: (1) Godot accesses a PHP page, this page contacts the database and performs actions (e.g., read or write); (2) in case data is read, this data is stored in the PHP script, and then sent back to Godot from this script.

What are SQL statements?

SQL statements can be used to perform a query on a database; they are set of instructions used to perform actions such as: selecting, ranking, reading or writing to records.

Why is there a need to set privileges for a database?

When you create an online (or local) database, you usually want to restrict access to it so that only admin users can modify it. This is so that the database is not changed by mistakes, or to avoid possible malicious and unauthorized changes to the database.

READING FILES

What is JSON?

JSON stands for **J**ava**S**cript **O**bject **N**otation. As you will see in the next code examples, these files use the extension **.json** and have a common syntax that makes them easy to read.

What advantage is there to using JSON for my games?

You can create your own JSON files using the attributes of your choice to best reflect and serve the requirements of your game or application; you could save information about each scene, about the NPCs (e.g., what paths they can use), or the weapons. So yes, you could virtually save any type of information with these files, using an easy-to-read format.

How can I read the content of a text file from Godot?

You can store this file in your project and read its content as follows:

```
onready var file = 'res://maze.txt'
func read_file():
var f = File.new()
f.open(file, File.READ)
var full_text = ""
while not f.eof_reached():
full_text += f.get_line()
f.close()
```

How can I read the content of an image (the color of each pixel)?

To read the pixels from an image at the position (x, y), you can use the method **get_pixel**, as demonstrated in the next code.

```
var image = load("res://outline.png")
var data = image.get_data()
data.lock()


for i in range (0,10,1):
for j in range (0,10,1):
var pixel = data.get_pixel(i,j)
```

Chapter 7: Thank you

I would like to thank you for completing this book; I trust that you are now comfortable with database access, procedural generation, networked games, and 2D games.

So that the book can be constantly improved, I would really appreciate your feedback. So, please leave me a helpful review on your e-store letting me know what you thought of the book and also send me an email (**learntocreategames@gmail.com**) with any suggestions you may have. I read and reply to every email.

Thanks so much!

[ ]