A STEP-BY-STEP GUIDE TO CODING YOUR FIRST GAME

GODOT FROM ZERO TO PROFICIENCY

(BEGINNER)

PATRICK FELICIA

Copyrighted material

Godot From Zero to proficiency (Beginner)

FIRST EDITION

A STEP-BY-STEP GUIDE TO CODING YOUR FIRST GAME.

Table of Contents

Patrick Felicia

Godot From Zero to Proficiency

(Beginner)

Copyright © 2021 Patrick Felicia

All rights reserved. No part of this book may be reproduced, stored in retrieval systems, or transmitted in any form or by any means, without the prior written permission of the publisher (Patrick Felicia), except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either expressed or implied. Neither the author and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

First published: April 2021

Published by Patrick Felicia

Credits

Author: Patrick Felicia

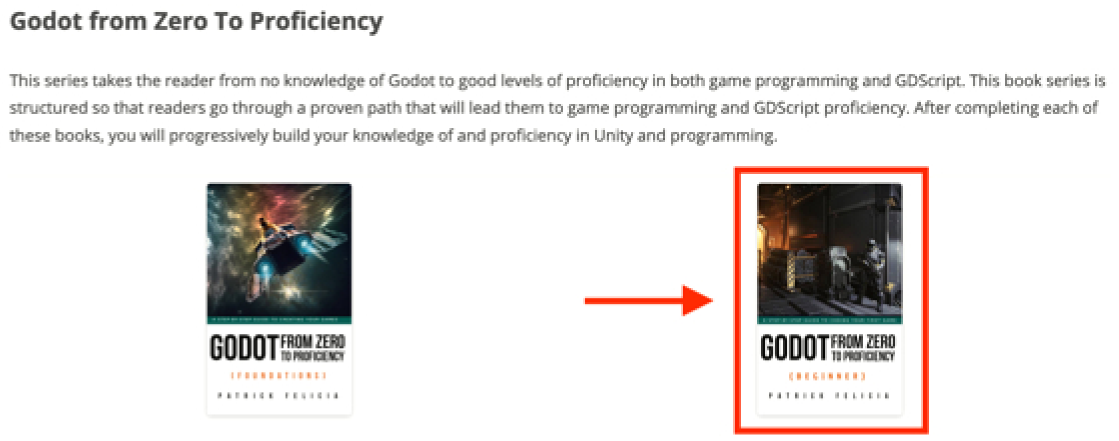
About the Author

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 PhD 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 Symposium 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 (Beginner)**)

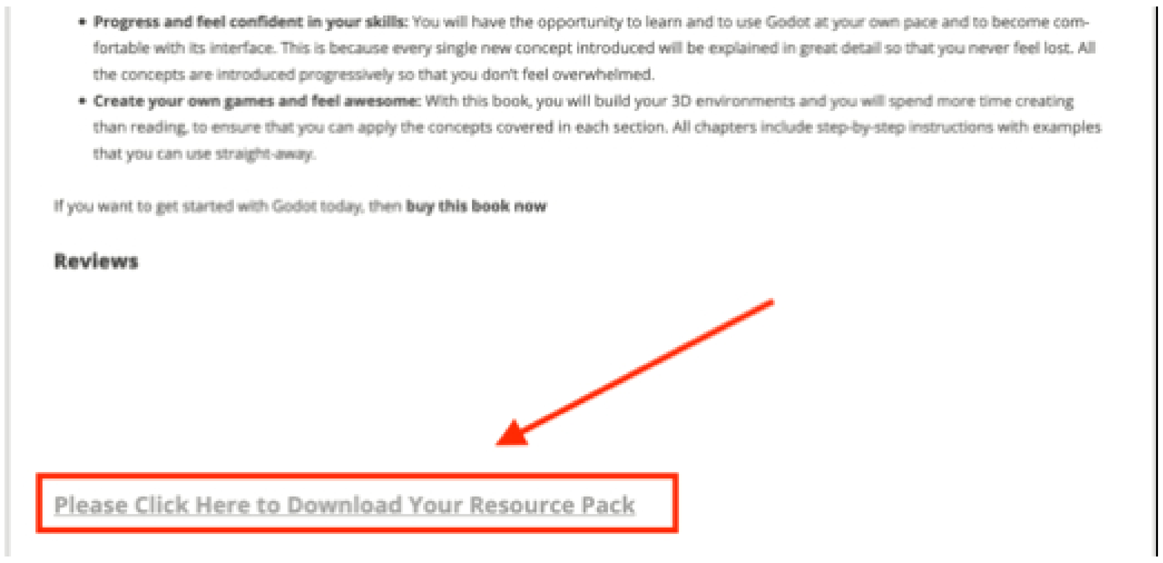


Godot from Zero To Proficiency

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]
PATRICE FELICIA
GODOT FROM ZERO TO PROFICIENCY
[BEGINNER]
PATRICE FELICIA

- 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 Helena & Mathis

Table of Contents

Preface

This book will show you how you can very quickly start using Godot and code in GDScript, a scripting language similar to Python.

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 specification, 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 a survival game using Godot's core features in terms of programming (i.e., GDScript), game design, and drag and drop features.

In addition to understanding and being able to master Godot's core features, you will also create a game that includes many of the common techniques found in video games, including: level design, object creation, textures, collision detection, lights, weapon creation, character animations, particles, artificial intelligence, and menus.

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, Introduction to Programming in GDScript, provides an introduction to GDScript and to core principles that will help you to get started. It explains key programming concepts such as variables, variable types, or functions.

Chapter 2, Creating your First Script, helps you to code your first script. It explains common coding mistakes and errors in GDScript, and how to avoid them easily. It also goes through some common error messages for beginners, and explains what they mean and how they can be avoided easily.

Chapter 3, Adding Interaction with GDScript, gets you to improve your scripting skills and your game and add more interaction. You will learn to create a scoring system, detect collisions, and load new levels.

Chapter 4, Creating and Updating a User Interface, explains how you can create a user interface using Godot's UI system. You will add onscreen elements such as images, or text, and update them with your scripts to display the score and other messages to the user.

Chapter 5, Polishing Our Game, explains how you can improve your game, by adding a splash-screen, displaying items collected onscreen, or adding sound effects and a mini-map.

Chapter 6 provides answers to Frequently Asked Questions (FAQs) related to the topics covered in this book (e.g., scripting, audio, interaction, AI, or user interface). It also provides links to additional exclusive video tutorials that can help you with some of your questions.

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** (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 the reader. This book does not include any programming, as this will be introduced in the second book in the series. 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.

Who this book is for

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

1. Are you a total beginner in GDScript?
2. Would you like to become proficient in the core functionalities offered by Godot?
3. Would you like to teach students or help your child to understand how to create games, using programming?
4. Would you like to start creating great 2D or 3D games?
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 **not** for you:

1. Can you already code with GDScript to implement simple behaviors such as score, collision detection, or to update the user interface?
2. Can you already easily code a 3D game with Godot with built-in objects, controllers, cameras, lights, and terrains?
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 structure 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 allowed 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 in 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.

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.

The full solution for the project presented in this book is available for download on the official website **(http://learntocreategames.com/books)**. So if you need to skip a section, you can do so; you can also download the solution for the previous chapter that you have skipped.

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.

The author's notes appear as described below:

The 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 the checklist
- Item2 for the checklist
- Item3 for the 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 your 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

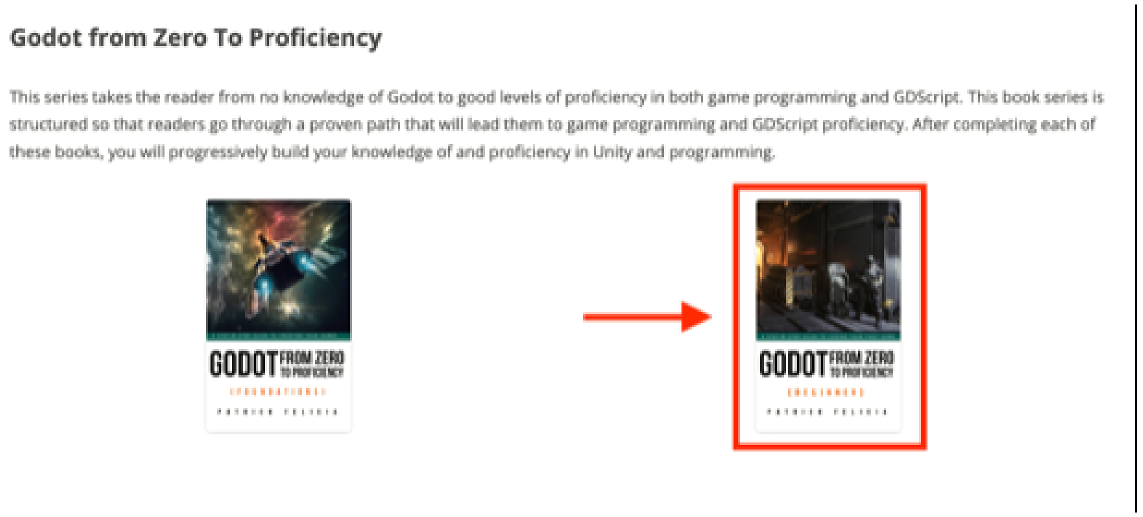
While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback, you can email me at **learntocreategames@gmail.com**.

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:

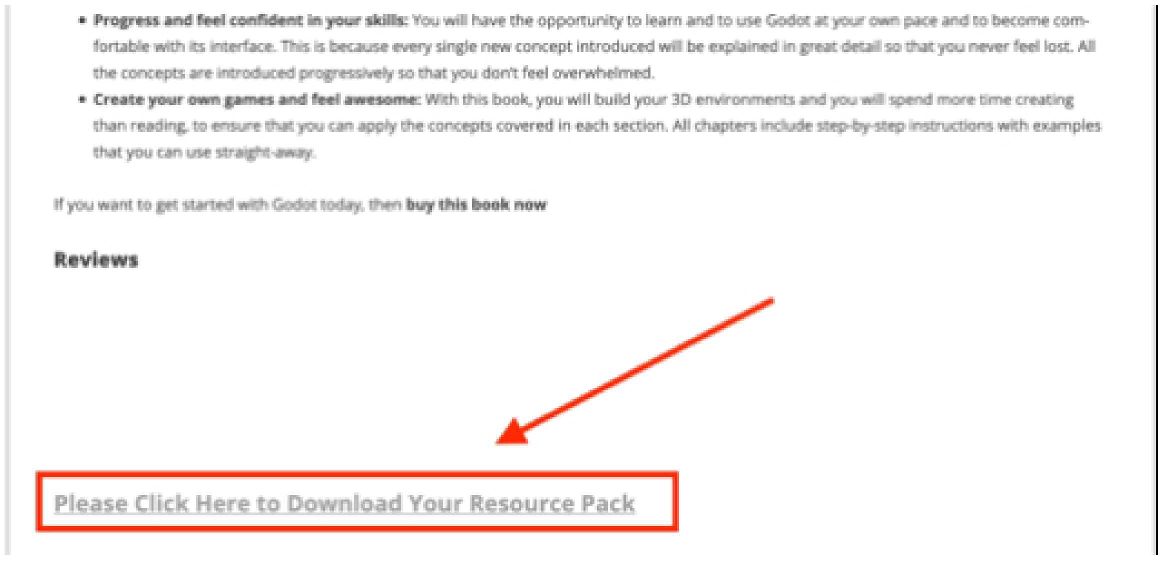
- Open the page **http://www.learntocreategames.com/books**.
- Click on your book (**Godot From Zero to Proficiency (Beginner)**)



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.

- 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 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., through Amazon, lulu.com, or the book's official website: **http://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: Introduction to Programming in GDScript

In this section we will discover GDScript programming principles and concepts, so that you can start programming in the next chapter. If you have already coded using GDScript (or a similar language), you can skip this chapter.

After completing this chapter, you will be able to:

- Understand Object-Oriented Programming (OOP) concepts when coding in GDScript.
- Become familiar with and understand the concepts of variables, methods, and scope.
- Understand key best practices for coding, especially in GDScript.
- Understand how to use conditional statements and decision-making structures.
- Understand the concept of loops.

INTRODUCTION

When coding in Godot, you are communicating with the Game Engine and asking it to perform actions. To communicate with the system, you are using a language or a set of words bound by a syntax that the computer and you know. This language consists of keywords, key phrases, and a syntax that ensures that your instructions are understood properly. In computer science, this sentence needs to be accurate, precise, unambiguous, and with correct syntax. In other words, it needs to be **exact**. The syntax is a set of rules that are followed when writing code in GDScript. In addition to its syntax, GDScript programming also uses classes; so your scripts will be saved as classes.

In the next section, we will learn how to use this syntax. If you have already coded in Python or other object-oriented programming languages, some of the information provided in the rest of this chapter may look familiar and this prior exposure to programming will definitely help you.

When scripting in GDScript, you will be using a combination of the following:

- Classes.
- Objects.
- Statements.
- Comments.
- Variables.
- Constants.
- Operators.
- Assignments.
- Data types.
- Methods.
- Decision-making structures.
- Loops.
- Inheritance (more advanced).
- Events.
- Comparisons.

- Type conversions.
- Reserved words.
- Messages to the console windows.
- Declarations.
- Calls to methods.

The list may look a bit intimidating but, not to worry, we will explore these in the next sections, and you will get to know and use them smoothly using hands-on examples.

STATEMENTS

When you write a piece of GDScript code, you need to ask the system to execute your instructions (e.g., to print information onscreen) using statements. A statement is literally an order or something you ask the system to do. For example, in the next line of code, the statement will tell Godot to print a message in the **Output window**:

```
print (“Hello World”)
```

When writing statements, there are a few rules that you need to know:

- Order of statements: each statement is executed in the order it appears in the script. For example, in the next example, the code will print **hello**, then **world**; this is because the associated statements are in that sequence.

```
print (“hello”)
print (“world”)
```

- Use one statement per line, as much as possible. Otherwise, use a semicolon to separate these statements.
- For example, the next line of code has incorrect syntax.

```
print(“hello”) print (“world”)
```

- Multiple spaces are ignored for statements; however, it is good practice to add spaces around the operators such as **+**, **-**, **/**, or % for clarity. For example, in the next example, we say that **a** is equal to **b**. There is a space both before and after the operator =.

```
a = b;
```

- Statements to be executed together (e.g., based on the same condition) can be grouped using what is usually referred to as **code blocks**. In GDScript code blocks are implemented using indentation. So, in other words, if you needed to group several statements, we would indent all of them at the same level, as follows:

```
if (x>100):
print (“hello stranger!”)
print (“today, we will learn about scripting”)
```

In the previous statements, if x is greater than 100 then two print statements will be executed.

As we have seen earlier, a statement usually employs or starts with a keyword (i.e., a word that the computer understands). All these keywords have a specific purpose and the most common ones (at this stage) are used for:

- Printing a message in the **Output window**: the keyword is **print**.
- Declaring a variable: the keyword **var**.
- Declaring a function: the keyword is **func**.

In GDScript the keywords **method** and **function** are used interchangeably.

- Marking a block of instructions to be executed based on a condition: the keywords are **if**...**else**.
- Exiting a function: the keyword is **return**.

Note that in GDScript you can end your statements using a semi-colon; this is however optional in GDScript; this being said, it can be useful if you would like to write multiple statements on one line, each separated by a semicolon, as in the next code:

```
print ("Hello"); print ("World")
```

COMMENTS

In GDScript, you can use comments to explain the code and to make it more readable. This becomes important as the size of your code increases; and it is also important if you work in a team, so that the other team members can understand your code and make amendments in the right places, if and when it is needed.

When some code is commented it is not executed. For comments, a **#** is added at the start of a line or after a statement, so that this line (or part thereof) is commented, as illustrated in the next code snippet.

```
#the next line prints Hello in the console window
print ("Hello")
#the next line declares the variable name
var name;
name = "Hello";#sets the value of the variable name
```

If you want to comment multiple lines in Godot you can select the code that you want to comment and press **CTRL + K**; to uncomment the same lines you can select the same code and press **CTRL + K** again.

In addition to providing explanations about your code, you can also use comments to prevent part of your code from being executed. This is very useful when you would like to debug your code and find where the errors or bugs might be, using a very simple method. By commenting part of your code, and by using a process of elimination, you can usually find the issue quickly. For example, you can comment all the code and run the script, then comment half the code, and run the script. If it works, it means that the error is within the code that has been commented, and if it does not work, it means that the error is in the code that has not been commented. In the first case (if the code works), we could then just comment half of the portion of the code that has already been commented. So, by successively commenting more specific areas of our code, we can get to discover what part of the code includes the bug. This process is often called **dichotomy** as we successively divide a code section into two. It is usually effective to debug your code because the number of iterations (dividing part of the code in two) is more predictable and also potentially less time-consuming. For example, for 100 lines of codes, we can successively narrow down the issue to 50, 25, 12, 6, and 3 lines and therefore use 5 to 6 iterations in this case, rather than going through the whole 100 lines of code.

VARIABLES

A variable is a container. It includes a value that may change over time. When using variables, we usually need to: (1) declare the variable by specifying its type (but this is not always needed), (2) assign a value to this variable, and (3) possibly combine this variable with other variables using operators. This is illustrated in the next code snippet.

```
var my_age:int #we declare the variable
my_age = 20 #we set the variable to 20
my_age = my_age + 1 #we add 1 to the variable my_age
```

In the previous example, we have declared a variable **my_age**, its type is **int** (integer), we set it to **20** and we then add **1** to it.

In GDScript a variable is declared using its name followed by its type. This being said, the type is optional.

Note that in the previous code we have assigned the value **my_age + 1** to **my_age**; the = operator is an assignment operator; in other words, it is there to assign a value to a variable and is not to be understood in a strict algebraic sense (i.e., that the values of the variables on both sides of the = sign are equal).

When using variables, there are a few things that we need to determine including their name, their type, and their scope:

- **Name of a variable:** A variable is usually given a unique name so that it can be identified uniquely. The name of a variable is usually referred to as an identifier; it can contain letters, digits, a minus, or an underscore, and it usually begins with a letter. Identifiers cannot be keywords. For example, the keyword **if** cannot be a variable name.
- **Type of variable:** variables can hold several types of data including numbers (e.g., integers, doubles, or floats), text (i.e., strings or characters), Boolean values (e.g., true or false), arrays, or nodes (i.e., we will see this concept later in this chapter) as illustrated in the next code snippet.

```
var my_name:String = "Patrick"#the text is declared using double quotes
var current_year:int  = 2015 #the year needs no decimals and is declared as an integer
var width:float = 100.45 #width is declared as a float (i.e., with decimals)
```

- **Variable declaration:** a variable needs to be declared so that the system knows what you are referring to if you use it in your code. A variable needs to be declared before it can be used. At the declaration stage, the variable does not have to be assigned a value, as this can be done later, as illustrated in the next code snippet.

```
var my_name:String
my_name = “My Name”;
```

In the previous example, we declare a variable called **my_name** and then assign the value **“My Name”** to it.

- **Scope of a variable:** a variable can be accessed (i.e., referred to) in specific contexts that depend on where in the script the variable was initially declared. We will look at this concept later.
- **Accessibility level:** as we will see later, a GDScript programme consists of classes; for each of these classes, the methods and variables within can be accessed depending on **accessibility** levels. We will look at this principle later on (there is no need for any confusion at this stage :-)).

Common variable types include:

- **String**: same as text.
- **Int**: integer (1, 2, 3, etc.).
- **Boolean**: true or false.
- **Float**: with a decimal value (e.g., 1.2, 3.4, etc.).
- **Arrays**: a group of variables of the same type. If this is unclear, not to worry, this concept will be explained further in this chapter.
- **Nodes**: a game node (any node in your game).

ARRAYS

Sometimes arrays can be used to make your code leaner, by applying features and similar behaviors to a wide range of data.

As we will see in this section, arrays can help to declare fewer variables (for variables storing the same type of information) and to also access them more easily.

When creating arrays, you can create single-dimensional arrays and multidimensional arrays.

Let's look at the simplest form of arrays: single-dimensional arrays. For this concept, we can take the analogy of a group of 10 people who all have a name. If we wanted to store this information using a String variable, we would need to declare (and set) ten different variables.

```
var name1:String
var name2:String
var name3:String
```

While this code is perfectly fine, it would be great to store these in only one variable. For this purpose, we could use an array. An array is comparable to a list of elements that we access using an index. This index usually starts at 0 for the first element in the list.

So let's see how we could do this with an array; first, we could declare the array as follows:

```
var names = []
```

You will probably notice the syntax **var name_of_the_array = []**. The syntax **string []** means that we declare an **array**.

We can then store information in this array as described in the next code snippet.

```
names [0] = "Paul"
names [1] = "Mary"
...
names [9] = "Pat"
```

In the previous code, we store the name **Paul** as the first element in the array (remember the index starts at 0); we store the second element (with the index 1) as **Mary**, as well as the last element (index is 9), **Pat**.

Note that for an array of size **n**, **the index of the first element is 0** and **the index of the last element is n-1**. So for an array of size 10, the index for the first element is 0, and the index of the last element is 9.

If you were to use arrays of integers or floats, or any other type of data, the process would be similar.

Now, one of the cool things you can do with arrays is that you can initialize your array in one line, saving you from the headaches of writing 10 lines of code if you have 10 variables, as illustrated in the next example.

```
names = ["Paul","Mary","John","Mark",  "Eva","Pat","Sinead","Elma","Flaithri",
"Eleanor"]
```

This is very handy, as you will see in the next chapters, and this should definitely save you a lot of time.

Now that we have looked into single-dimensional arrays, let's look at multidimensional arrays, which can also be very handy when storing information. This type of array (i.e., multidimensional array) can be compared to a building with several floors, and on each floor, several apartments. So let's say that we would like to store the number of tenants for each apartment; we would, in this case, create variables that would store this number for each of these apartments.

The first solution would be to create variables that store the number of tenants for each of these apartments with a variable that refers to the floor, and the number of the apartment. For example, **ap0_1** could be for the first apartment on the ground floor, **ap0_2**, would then be for the second apartment on the ground floor, **ap1_1**, would then be for the first apartment on the first floor, and **ap1_2**, would then be for the second apartment on the first floor. So in term of coding, we could have the following:

```
int ap0_1 = 0
int ap0_2 = 0
...
```

Using arrays instead we could do the following:

```
var app_array = []
app_array [0][1] = 0
app_array [0][2] = 0
print (app_array[0][1])
```

In the previous code:

- We declare our array.
- We add values to our array.
- The last line of code prints (in the **Output window**) the value of the first element of the array.

One of the other interesting things with arrays is that, using a loop, you can write a single line of code to access all the elements of this array, and hence, write more efficient code.

CONSTANTS

So far we have looked at variables and how you can store and access them seamlessly. The assumption then was that a value may change over time, and that this value would be stored in a variable. However, there may be times when you know that a value will remain constant. For example, you may want to define labels that refer to values that should not change over time, and in this case, you could use constants. Let's look at the following example: let's say that the player may have three choices in the game (e.g., referred to as 0, 1, and 2) and that you don't really want to remember these values, or that you would like to find a way that makes it easier to refer to them. Let's look at the following code:

```
if (user_choice == 0): print ("you have decided to restart")
if (user_choice == 1): print ("you have decided to stop the game")
if (user_choice == 2): print ("you have decided to pause the game")
```

In the previous code:

- The variable **user_choice** is an integer and is set to **2**.
- We then check its value and print a message accordingly.

Now, you may or may not remember that 0 corresponds to restarting the game; the same applies to the other two values. So instead, we could use constants to make it easier to remember (and use) these values. Let's look at the equivalent code with the use of constants.

```
const CHOICE_RESTART: int = 0
const CHOICE_STOP:int = 1
const CHOICE_PAUSE:int = 2
...
...
...
if (user_choice == CHOICE_RESTART): print ("you have decided to restart")
if (user_choice == CHOICE_STOP): print ("you have decided to stop the game")
if (user_choice == CHOICE_PAUSE): print ("you have decided to pause the game")
```

In the previous code:

- We declare three **constants**.
- These constants are then used to check the choice made by the user.

In the next example, we use a constant to calculate a tax rate; this is a good practice as the same value will be used across the programme with no or little room for errors when it comes to using the exact same tax rate across the code.

```
const VAT_RATE:float = 0.21;
...
var price_before_vat:float = 23.0
var price_after_vat:float = price_before_vat * VAT_RATE;
```

In the previous code:

- We declare a **constant** float for the VAT rate.
- We declare a **float** variable for the item’s price before the VAT.
- We calculate the amount of tax.

Note that constants usually need to be declared at the beginning of your script in GDScript.

It is a very good coding practice to use constants for values that don’t change across your programme. Using constants makes your code more readable; it saves work when you need to change a value in your code, and it also decreases possible occurrences of errors (e.g., for calculations). Also, note that it is common practice to uppercase constants.

OPERATORS

Once we have declared and assigned values to a variable, we can use operators to modify or combine variables. There are different types of operators including: arithmetic operators, assignment operators, comparison operators, and logical operators.

Arithmetic operators are used to perform arithmetic operations including additions, subtractions, multiplications, or divisions. Common arithmetic operators include **+**, **-**, *, **/**, or **%** (modulo).

```
var number1:int = 1 #the variable number1 is declared
var number2:int = 1 # the variable number2 is declared
var sum = number1 + number2 # adding two numbers and store them in sum
var sub = number1 - number2 # subtracting two numbers and store them in sub
```

Assignment operators can be used to assign a value to a variable and include =, **+=**, **-=**, ***=**, **/=** or %=.

```
var number1:int = 1;
var number2:int = 1;
number1 += 1; #same as number1 = number1 + 1;
number1 -= 1; #same as number1 = number1 - 1;
number1 *= 1; #same as number1 = number1 * 1;
number1 /= 1; #same as number1 = number1 / 1;
number1 %= 1; #same as number1 = number1 % 1;
```

Note that the **+** operator, when used with strings, will concatenate strings (i.e., add them one after the other to create a new string).

When you need to concatenate a number and a string, you usually need to convert the number to a string first; for example:

```
var name:String = "Cars"
var my_number:int = 3
var message:String = "I have " + str(my_number) + " " + name
print(message)
```

Comparison operators are often used for conditions to compare two values;

comparison operators include **==, !=**, >, <, **>=** and **>=**.

```
var number1:int = 1; var number2:int = 3;
if (number1 == number2):print("number1 equals number2")
if (number1 != number2):print("number1 and number2 have different values")
if (number1 > number2): print("number1 is greater than number2")
if (number1 >= number2): print("number1 is greater than or equal to number2")
if (number1 < number2): print("number1 is less than number2")
if (number1 <= number2): print("number1 is less than or equal to number2")
```

CONDITIONAL STATEMENTS

Statements can be performed based on a condition, and in this case, they are called **conditional statements**. The syntax is usually as follows:

if (condition): statement

This means **if the condition is verified (or true) then (and only then) the statement is executed**. When we assess a condition, we test whether a declaration is true. For example, by typing **if (a == b)**, we mean **"if it is true that a equals b"**. Similarly, if we type **if (a>=b)** we mean **"if it is true that a is greater than or equal to b"**

As we will see later on, we can also combine conditions. For example, we can decide to perform a statement if two (or more) conditions are true. For example, by typing **if (a == b && c == 2)** we mean **"if a equals b and c equals 2"**. In this case, using the operator **&&** means **AND**, and that both conditions will need to be true. We could compare this to deciding on whether we will go sailing tomorrow. For example, "**if the weather is sunny and the wind speed is less than 5km/h then I will go sailing**". We could translate this statement as follows.

if (weather_is_sunny == true && wind_speed < 5): I_go_sailing = true

When creating conditions, as for most natural languages, we can use the operator **OR** noted **||.** Taking the previous example, we could translate the following sentence "**if the weather is too hot or if the wind is faster than 5km/h then I will not go sailing** " as follows.

if (weather_is_too_hot == true || wind_speed > 5): I_go_sailing = false;

Another example could be as follows.

```
if (my_name == "Patrick"): print("Hello Patrick")
else: print ("Hello Stranger")
```

When we deal with combining true or false statements, we are effectively applying what is called **Boolean logic**. Boolean logic deals with Boolean variables that have two values 1 and 0 (or true and false). By evaluating conditions, we are effectively processing Boolean numbers and applying Boolean logic. While you don't need to know about Boolean logic in-depth, some operators for Boolean logic are important, including the **!** operator. It means **NOT** or the opposite. This means that if a variable is true, its opposite will be false, and vice versa. For example, if we consider the variable **weather_is_good = true**, the value of **! weather_is_good** will be **false** (its opposite). So the condition **if (weather_is_good == false)** could be also written **if (!weather_is_good)** which would literally translate as “if the weather is **NOT** good”.

MATCH STATEMENTS

If you have understood the concept of conditional statements, then this section should be pretty much straightforward. Match statements are a variation on the if/else statements that we have seen earlier. The idea behind match statements is that, depending on the value of a specific variable, we will switch to a particular portion of the code and perform one or several actions. The variable considered for the match structure can be of different types including **integer** or **String**. Let's look at a simple example:

```
var choice:int = 1;
match choice:
1:
print ("you chose 1")
2:
print ("you chose 2")
3:
print ("you chose 3")
_:
print ("Default option")
print ("We have exited the match structure")
```

In the previous code:

- We declare the variable **choice**, as an **integer** and initialize it to **1**.
- We then create a **match** structure whereby, depending on the value of the variable **choice**, the programme will switch to the relevant section (i.e., the portion of code starting with **1:**, **2:**, etc.). Note that in our code, we look for the values 1, 2, or 3. However, if the variable **choice** does not equal 1, 2, or 3, the program will branch to the section starting with **_**. This is because this section is executed if any of the other possible choices (i.e., 1, 2, or 3) have not been fulfilled (or selected).

Note that in GDScript, contrary to other languages, the system will exit the match structure once one of the options (or the default one) has been executed. So the **break** statement that is usually found in other languages to specify to leave the switch structure after executing the commands included in the branch (or the current choice), is no longer necessary.

So let's consider the previous example and see how this would work. In our case, the variable **choice** is set to **1**, so we will enter the **match** structure, and then look for the section that deals with a value of **1** for the variable **choice**. This will be the section that starts with **case 1:**; then the command **print ("you chose 1");** will be executed, we then exit the match structure (implicit break); finally the command **print ("We have exited the match structure")** will be executed.

Match statements are very useful to structure your code and when dealing with mutually exclusive choices (i.e., when only one of the choices can be processed) based on an integer value, especially in the case of menus. besides, match structures make for cleaner and easily understandable code.

LOOPS

There are times when you have to perform repetitive tasks as a programmer; many times, these can be fast-forwarded using loops. Loops are structures that will perform the same actions repetitively based on a condition. So, the process is usually as follows:

- Start the loop.
- Perform actions.
- Check for a condition.
- Exit the loop if the condition is fulfilled or keep looping.

Sometimes the condition is performed at the start of the loop, some other times it is performed at the end of the loop.

Let's take the following example that is using a **while** loop.

```
var x:int = 0;
while (x<10):
print("x"+str(x))
x += 1
```

In the previous code:

- We set the value of the variable **x**.
- We then create a loop that starts with the keyword **while**.
- We set the condition to remain in this loop (i.e., **x** < **10**).
- Within the loop, we increase the value of the variable **x** by **1** and print its value.

So effectively:

- The first time we go through the loop: the variable **x** is increased to **1**; we reach the end of the loop; we go back to the start of the loop and check if **x** is < 10; this is true in this case (**x** = 1).
- The second time we go through the loop: **x** is increased to **2**; we reach the end of the loop; we go back to the start of the loop and check if **x** is <10; this

is true in this case (**x** = **2**).

- ...
- The 10th time we go through the loop: **x** is increased to 10; we reach the end of the loop; we go back to the start and check if **counter** is < 10; this is now false in this case (**counter** = 10). As a result, we then exit the loop.

So, as you can see, using a loop, we have managed to increment the value of the variable **x** iteratively, from 0 to 10, but using less code than would be needed otherwise.

Another variation of the code could be as follows:

```
for x in range(0,10):
print ("x"+str(x));
```

In the previous code:

- We declare a loop in a slightly different way: we say that we will use a variable called **x** that will go from 0 to 9 (we exclude the upper boundary).
- This variable **x** will be incremented by 1 every time we go through the loop.
- We remain in the loop as long as the variable **x** is less than 10.
- The test for the condition, in this case, is performed at the start of the loop.

Loops are very useful to be able to perform repetitive actions for a finite number of objects, or to perform what is usually referred to as recursive actions. For example, you could use loops to create (i.e., instantiate) 100 objects at different locations (this will save you some code :-)), or to go through an array of 100 (or more) elements.

CLASSES

When coding in GDScript with Godot, you will be creating code that includes your own classes or uses built-in classes. So what is a class?

As we have seen earlier, GDScript is an Object-Oriented Programming (OOP) language. Put simply, a GDScript programme will consist of a collection of objects that interact amongst themselves. Each object has one or more attributes, and it is possible to perform actions on these objects using what are called **methods or functions**. Also, objects that share the same properties are said to belong to the same **class**. For example, we could take the analogy of a bike. There are bikes of all shapes and colors; however, they share common features. For example, they all have a specific number of wheels (e.g., one, two or three) or a speed; they can have a color, and actions can be performed on these bikes (e.g., accelerate, turn right, turn left, etc.). So in object-oriented programming, the class would be **Bike**, speed or color would be referred to as member variables, and accelerate (i.e., an action) would be referred to as a member method. So if we were to define a common type, we could define a class called **Bike** and for this class define several member variables and attributes that would make it possible to define and perform actions on the objects of type **Bike**.

This is, obviously, a simplified explanation of classes and objects, but it should give you a clearer idea of the concept of object-oriented programming, if you are new to it.

DEFINING A CLASS

So now that we have a clearer idea of what a class is, let's see how we could define a class. So let's look at the following example.

```
class_name Bike
var speed:float
var color:String
var name:String
func accelerate():
speed+=1
func turn_right():
print("Turning Right")
```

```
func calculate_distance():
    print("Calculating Distance")
```

In the code above, we have defined a class, called **Bike**, that includes two member variables (**speed** and **color**) as well as two member methods/functions (**accelerate** and **turn_right**). Let's look at the script a little closer; you may notice a few things:

- The name of the class is preceded by the keywords **class_name**.
- Three variables called speed, color and name are defined; they are called member variables because they are declared before any function and are therefore accessible throughout the class.
- Two functions are declared: **turn_right** and **calculate_distance**.

If you have worked with other programming languages such as C#, you may be used to define the access modifiers for both the member methods and variables; in GDScript, these are set to public by default.

ACCESSING CLASS MEMBERS AND VARIABLES

Once a class has been defined, it's great to be able to access its member variables and methods. In GDScript (as for other object-oriented programming languages), this can be done using the **dot notation**.

The dot notation refers to **object-oriented programming**. Using dots, you can access properties and functions (or methods) related to a specific object.

Once a class has been defined, objects based on this class can be created. For example, if we were to create a new **Bike** object, based on the code that we have seen previously, the following code could be used.

```
var Bike = load("Bike.gd")
var my_bike:Bike = Bike.new();
```

In the previous code, we load the class called **Bike** and we then instantiate a new object called **myBike**. So, this code will effectively create an object based on the "template" **Bike**. You may notice the syntax:

```
var variable_name: data_type  = data_type.new()
```

By default, this new object will include all the member variables and methods defined earlier. So it will have a color and a speed, and we should also be able to access its **accelerate** and **turn_right** methods. So how can this be done? Let's look at the next code snippet that shows how we can access these.

```
var b:Bike = Bike().new();
b.accelerate();
b.color = "Blue";
```

In the previous code:

- The new bike **b** is created.
- The speed is then increased after calling the **accelerate** method. This function can be called using the dot notation because it is a member function.
- We also set the color for the bike.
- Note that to call an object's method we use the dot notation.

CONSTRUCTORS

As we have seen in the previous sections, when a new object is created, it will, by default, include all the member variables and methods. To create this object, we can use the name of the class, followed by an opening and closing round bracket, as per the next example.

```
var my_bike : Bike = Bike.new();
my_bike.accelerate();
```

In fact, it is possible to change some of the properties of the new object created when it is initialized. For example, instead of setting the speed and the color of the object as we have done in the previous code, it would be great to be able to set these automatically and pass the parameter accordingly when the object is created. Well, this can be done with what is called a **constructor**.

A constructor literally helps to construct your new object based on parameters (also referred to as arguments) and instructions. So, for example, let's say that we would like the color of our bike to be specified when it is created; we could modify the **Bike** class, by adding the following method:

```
func _init (new_color:String):
color = new_color;
```

Note that if you don't need to pass any parameter when the construction is called, then you don't need to define the function **init**.

This is a new constructor and it takes a **String** as a parameter; so after modifying this constructor (as per the code above), we could then create a new bike object as follows:

```
var my_bike:Bike = Bike.new(“Blue”);
```

This being said, we could also modify the constructor so that if we call the constructor without parameters, that a default value is assigned for the color, as follows;

```
func _init (new_color:String=”Blue”):
color = new_color;
```

In the previous code, we specify that by default, if no parameters are entered, the color will be blue; however, if a parameter has been entered, then the new parameter will be used as the new color.

Furthermore, we could create a constructor that includes several parameters, all with default values as follows:

```
func _init(new_name:="A New Bike", new_color:="Blue",new_speed:=0):
color = new_color
speed = new_speed
name = new_name
```

In the previous code the define a constructor with the following features:

- It takes three parameters for the **name**, **color**, and **speed** of the bike.
- All parameters have a default value: "**A New Bike**", "**Blue**", and **0** respectively.
- If parameters were entered they are then used to initialize the member variables **name**, **color** and **speed**.

We would then call this constructor as follows:

```
var my_bike:Bike = Bike.new("Fast Bike","Red",10)
var my_bike2:Bike = Bike.new()
```

In the previous code:

- We create two different bikes.
- For the first bike, we call the constructor by passing parameters for the name, color and speed or the bike
- We also create a second bike but with no parameters this time; this means that this bike will have the default values assigned to its member variables name, color and speed (i.e., "**A New Bike**", "**Blue**", and **0**).

STATIC MEMBERS OF A CLASS

In GDScript, a member function can be declared as static (however, member variables can't) and in this case, it can be called without the need to instantiate an object of this class. This can be very useful if you want to create and avail of methods that can be used without the need to instantiate the class.

For example, following the previous example for the class Bike, we could create a static member method called **say_hello** as follows inside the class called **Bike**:

```
static func say_hello():
print("Hello");
```

In the previous code, we declare the function **say_hello** as static using the keyword **static**; this function prints **Hello** in the **Output** window when called.

To be able to use this function, we could then do the following inside a different file:

```
Bike.say_hello();
```

You may notice that, in the previous code, we have called the method **say_hello** without having to instantiate a new Bike. This is because the method **say_hello** is static.

INHERITANCE

I hope everything is clear so far, as we are going to look at the concept of inheritance, which is very important in object-oriented programming.

The main idea behind inheritance is that objects can inherit their properties from other objects (their parent). As they inherit these properties, they can remain identical or evolve and overwrite some of these inherited properties. This is very interesting because it makes it possible to minimize your code by creating a parent class with general properties for all objects sharing similar features, and then, if need be, overwrite and customize some of these properties for the children.

Let's take the example of vehicles: vehicles would generally have the following properties:

- Number of wheels.
- Speed.
- Number of passengers.
- Color.
- Capacity to accelerate.
- Capacity to stop.

So we could create the following class for example:

```
class_name Vehicle
var nb_wheels:int
var speed:float
var nb_passengers:int
var color:int
func  accelerate():
speed+=1
```

These features could apply for example to cars, bikes, motorbikes, or trucks. However, all these vehicles also differ; some of them may or may not have an engine or a steering wheel. So we could create a subclass called **MotorizedVehicles**, based on **Vehicles**, but with specificities linked to the fact that they are motorized.

These added attributes could be:

- Engine size.
- Petrol type.
- Petrol levels.
- Ability to fill up the tank.

The following example illustrates how this class could be created.

```
extends Vehicle
class_name MotoredVehicle
var engine_size:float;
var petrol_type:int;
var petrol_levels:float;
func fill_up_tank():
petrol_levels += 10;
```

- At the beginning of the file we specify that the current class inherits from the class **Vehicle**. So it will, by default, avail of all the methods and variables already included in the class **Vehicle**.
- We then define the name of the class (i.e., **MotoredVehicle**).
- We have created a new member method for this class, called **fill_up_tank**.
- In the previous example, you may notice that the methods and variables that were defined for the class **Vehicle** do not appear here; this is because they are implicitly added to this new class, since it inherits from the class **Vehicle**.

When using inheritance, the parent is usually referred to as the **base class**, while the child is referred to as the **inherited class**.

Now, while the child inherits "Behaviors" from its parents, these can always be modified or, put simply, overwritten. In GDScript, a function with the same name as a function defined in the parent will override the latter.

This is illustrated in the following code snippets.

```
class_name Vehicle
```

```
var nb_wheels:int;
var speed:float;
var nb_passengers:int;
var color:int;
func  accelerate():
speed += 1;
```

In the previous code snippet, we define the class called **Vehicle**; as you may have noticed, it includes member variables **nb_wheels**, **speed**, **nb_passengers**, and color. It also includes a member function called **accelerate** that increases the **speed** by one when it is called.

The next snippet illustrates the definition of a class called **MotoredVehicle**. As you will see it includes additional member variables, as well as a function called **accelerate**; however, contrary to the accelerate function defined for the parent class (**Vehicle**), this function increases the speed by **10** (and not 1).

```
extends Vehicle
class_name MotoredVehicle
var engine_size:float;
var petrol_type:int;
var petrol_levels:float;
func fill_up_tank():
petrol_levels+=10;
func accelerate():
speed += 10
```

Finally, in the next code snippet we create two types of objects: an instance of the class **Vehicle** (**v1**) and an instance of the class **MotoredVehicle** (**v2**).

```
var v1:Vehicle =  Vehicle.new();
v1.accelerate();
print ("V1 Speed: " + str(v1.speed));
var v2:MotoredVehicle = MotoredVehicle.new();
v2.accelerate();
print ("V2 Speed: " + str(v2.speed));
```

As you may have noticed:

- **v1** is created as an instance of the class **Vehicle**.
- The function **accelerate** is called; because this object is an instance of the class **Vehicle**, the function **accelerate** that is a member of the class **Vehicle** will be called; as a result, the speed should be increased by 1.
- **v2** is created as an instance of the class **MotoredVehicle**.
- The function **accelerate** is called; because this object is an instance of the class **MotoredVehicle**, and because this class has its own function **accelerate**, the function **accelerate** that is a member of the class **MotoredVehicle** will be called; as a result, the speed should be increased by 10.

If we were to run the last snippet, the following would be displayed in the **Output** window.

```
V1 Speed: 1
V2 Speed: 10
```

METHODS AND FUNCTIONS

As mentioned earlier, the terms method and function will be used interchangeably in this book. Methods or functions can be compared to a friend or a colleague to whom you gently ask to perform a task, based on specific instructions, and to return the information to you then. For example, you could ask your friend the following: "**Can you please tell me when I will be celebrating my 20th birthday given that I was born in 2000**". So you give your friend (who is good at Math :-)) the information (date of birth) and s/he will calculate the year of your 20th birthday and give this information back to you. So in other words, your friend will be given an input (i.e., the date of birth) and return an output (i.e., the year of your 20th birthday). Methods work exactly this way: they are given information (and sometimes not), perform an action, and then (sometimes, if it is needed) return information.

In programming terms, a method (or function) is a block of instructions that performs a set of actions. A method is executed when invoked (or put more simply **called**), or when an event occurs (e.g., the player has clicked on a button or the player collides with an object; we will see more about events in the next section). As for member variables, member functions or methods are declared and they can also be called.

Methods are very useful because once the code for a method has been created, it can be called several times without the need to rewrite the same code over and over again. Also, because methods can take parameters, a method can process these parameters and produce or return information accordingly; in other words, they can perform different actions and produce different information based on the input. As a result, methods can do one or all of the following:

- Take parameters and process them.
- Perform an action.
- Return a result.

A method has a syntax and can be declared in at least two ways:

```
func name_of_the_function():
Perform actions here...
```

Please note that any statement part for a specific method needs to be indented.

In the previous code, the method does not take any input, neither does it return an output. It just performs actions.

OR

```
func name_of_the_function(parameter1):
Perform actions here...
```

In the previous code snippet, the method takes one parameter and then performs actions.

Let's look at the following method for instance.

```
func calculate_sum(a,b):
return (a+b)
```

In the previous code:

- The function is declared and is called **calculate_sum**.
- The method takes two parameters.
- The method returns the sum of the two parameters which are referred to as **a** and **b** within this method.

A method can then be called using the **()** operator, as follows:

```
name_of_the_function1 ();
name_of_the_function2 (parameter1);
var test = name_of_the_function3 (parameter2);
```

In the previous code, a method is called with no parameter (line 1), or with a parameter (line 2). In the third example (line 3), a variable called **test** will be set with the value returned by the method **name_of_the_function3**.

DEFAULT PARAMETERS AND RETURN TYPES FOR FUNCTIONS

Now that you know a bit more about functions, we will see how the definition of functions can be refined to include a return type, a type for the parameters, or default parameters.

Let's start with the parameters; using GDScript, you can specify a type for the parameters passed to the function, as illustrated in the next code snippet.

```
func calculate_sum(a:int,b:int):
return (a+b)
```

In the previous code:

- We declare a function called **calculate_sum**.
- This function takes two parameters.
- Each parameter (referred to as **a** and **b**) should be integers. This means that passing parameters that are not integers (e.g., float or string) will result in an error.
- The function then returns the sum of the two integers passed as parameters.

Now that we have seen how to specify the type of the parameters passed to a function, let's see how we can specify default values; in GDScript, you can specify default values for each of the parameters that should be passed to a function, as illustrated in the next code snippet.

```
func calculate_sum(a:int=0,b:int=0):
return (a+b)
```

In the previous code:

- We declare a function called **calculate_sum**.
- This function takes two parameters, each of them are integers.
- The default value for the first parameter is **0**.
- The default value for the second parameter is **0**.
- As a result, if no parameters are specified when calling this function, it will return **0** (i.e., 0 + 0)

Let's illustrate this with the following code:

```
var sum1 = calculate_sum(1,2);
print("Sum1: " + str(sum1))
var sum2 = calculate_sum()
print ("Default Sum: " + str(sum2))
```

In the previous code:

- We create a variable called **sum1**; we then call the function **calculate_sum**, passing the values **1** and **2**, and what is returned by the function (**1 + 2**) is then stored in the variable **sum1**.
- We then print the value of the variable **sum1**.
- We create a variable called **sum2**; we then call the function **calculate_sum**, passing no parameters (i.e., empty round brackets), and what is returned by the function is then stored in the variable **sum2**. In that case, because no parameters were passed to the function, the default value for the parameters will be **0**; as a result, the function will return **0** and this value will be stored in the variable called **sum2**.
- We then print the value of the variable **sum2**.

If we were to run the previous code snippet, the **Output** window should display the following:

```
Sum1: 3
Default Sum: 0
```

Last but not least, let's look at the return type for a function. As we have seen previously, functions can return values, and it is also possible to specify the return type for a function; this is especially useful to ensure that the correct type of data is returned by the function or that this function doesn't return any value if it is not meant to.

Let's look at the following code snippet to see how this can be done:

```
func calculate_sum(a:int=0,b:int=0) -> int:
return (a+b)
```

In the previous code, we declare the function **calculate_sum**, as we have done before; however, we add “**-> int”** just before the colons to indicate that this function should return an integer. This means that returning a value that is not an integer will trigger an error.

We can also specify that the function should not return any data; in that case, the return type should be **void**, as illustrated in the next code snippet.

```
func display_full_name(f_name:String="John", l_name: String="Murphy")
->void:
print ("Full Name: " + f_name + " " + l_name)
```

in the previous code:

- We declare a function called **display_full_name**.
- This function takes two **String** parameters.
- By default, these parameters will be **John** and **Murphy**.
- We add **-> void** just before the colons to indicate that this function should return no values.
- The function then prints the full name based on the first and second parameters passed to this function.

SCOPE OF VARIABLES

Whenever you create a variable in GDScript, you will need to be aware of its scope so that you can use it accordingly.

The scope of a variable refers to where you can use this variable in a script. In GDScript, we usually make the difference between **member variables** and **local variables**.

When you create a class definition along with member variables, these variables will be seen by any function within your class.

Member variables can be used anywhere in your class. These variables need to be declared at the start of the class (using the usual declaration syntax) and outside of any method; they can then be used anywhere in the class as illustrated in the next code snippet.

```
class_name Bike
var speed:float
var color:String
var name:String
func accelerate():
speed+=1
```

In the previous code we declare the variable **speed** as a member variable and access it from the method accelerate.

Local variables are declared within a method (or function) and are to be used only within this method, hence the term local, because they can only be used locally, as illustrated in the next code snippet.

```
func function1():
var a = "Hello"
func function2():
var b = "World"
}
```

In the previous code, **a** is a local variable to the function **function1**, and can only be used within this function; **b** is a local variable to the function **function2**, and can only be used within this function.

EVENTS OR SIGNALS

Throughout this book and in GDScript, you will employ events that can be compared to a notification that you may receive on your phone. For example, when an event occurs, such as the alarm going off (event), we can either get up (action) or decide to go back to sleep. When you receive a message (event), you can decide to read it (action), and then reply to the sender (another action).

In computer terms, events are quite similar, although the events that we will be dealing with will be slightly different. So, in a game, we could be waiting for the user to press a key (event) and then move the player character accordingly (action), or wait until the user clicks on a button on screen (event) to load a new scene (action).

In Godot, whenever an event occurs, a function (or method) is usually called; the method, in this case, is often referred to as a handler, because it "handles" the event. You have then the opportunity to modify this function and to add instructions (i.e. statements) that should be followed, should this event occur.

To take the analogy of daily activities: we could write instructions to a friend on a piece of paper, so that, in case someone calls in our absence, the friend knows exactly what to do. So an event handler is basically a set of instructions (usually stored within a method) to be followed in case a particular event occurs.

Sometimes information is passed to the handler about the particular event, and sometimes not. For example, when the screen is refreshed the method **_process** is called. When the game starts (i.e., when a particular script is enabled), the method **_ready** is called. When a button is pressed, the method **_pressed** is called.

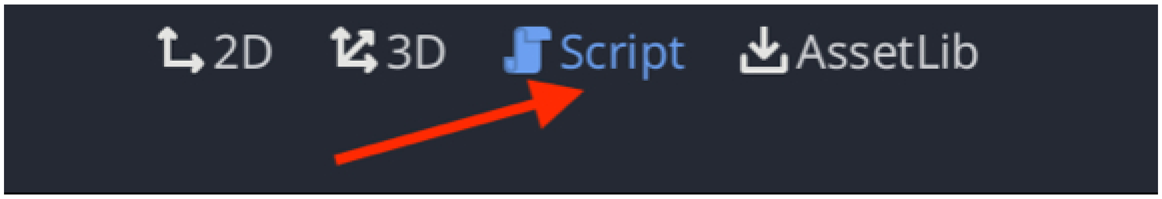
As you can see, we can deal with a wide range of events and signals in our game, and we will get to do that later on. In this book, we will essentially be dealing with the following events:

- **_ready**: when a node is available in the scene.
- **_process**: when the screen is refreshed (e.g., every frame).
- **_pressed**: when the user clicks on a button.

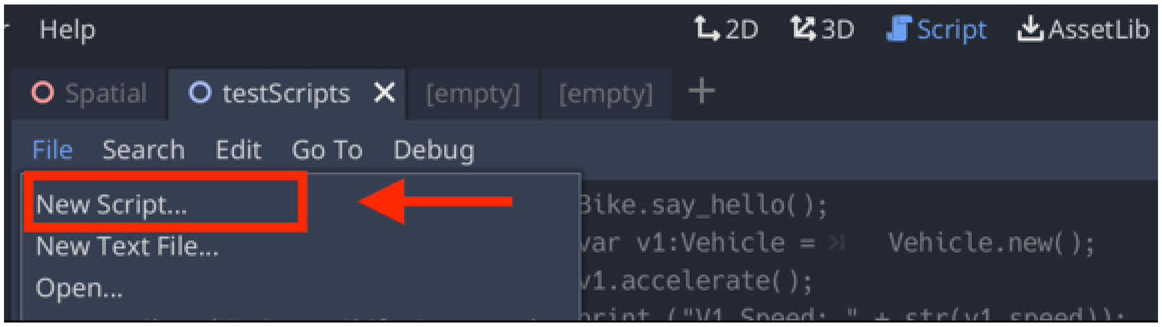
WORKFLOW TO CREATE A SCRIPT

There are many ways to create and use scripts in Godot, but generally the process is as follows:

- Select the **Script** workspace by pressing the corresponding button at the top of your screen.

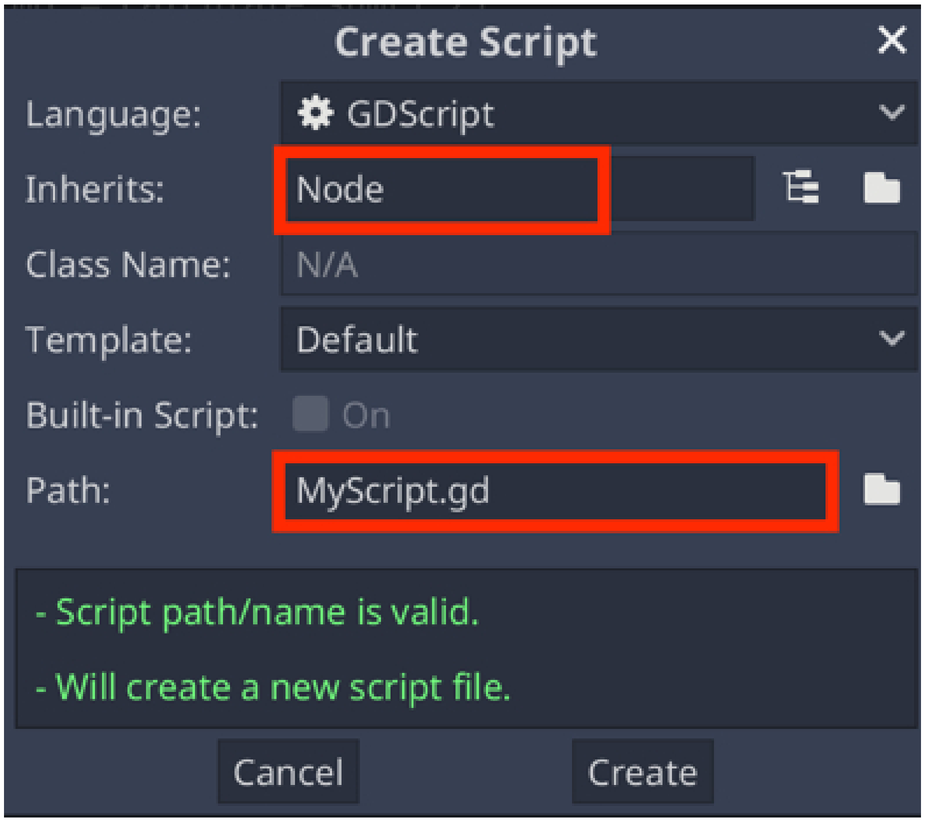


- Select: **File | New Script** from the left menu.



```
Bike.say_hello();
var v1:Vehicle =     Vehicle.new();
v1.accelerate();
print ("V1 Speed: " + str(v1.speed));
```

- Specify a name and parent type for your script; by default, your script will inherit from the class **Node** and it will be saved in the root folder (**res**), as illustrated in the next figure.


Create Script

Language: GDScript

Inherits: Node

Class Name: N/A

Template: Default

Built-in Script: On

Path: MyScript.gd

- Script path/name is valid.

- Will create a new script file.

- Attach the script to a node.
- Check in the **Output window** that there are no errors in the script.
- Play the scene.

When you create your script, by default the following code will usually be created within (specifically if the script is linked to a **Node**):

extends Node

# Declare member variables here. Examples:

# var a = 2

# var b = "text"

# Called when the node enters the scene tree for the first time.

```
    func _ready():
    pass # Replace with function body.
```

```
# Called every frame. 'delta' is the elapsed time since the previous frame.
    #func _process(delta):
    # pass
```

By default, when you create a new **GDScrip**t file, it is a class and it includes member functions. In the previous code, we can see that by default the new file (or class) inherits from the class **Node**, and it includes two methods that can be modified: **_ready** and ***_process***. You can of course remove the first line of code *(i.e.,* "**extends node**"), which means that the file that you have created will not inherit from the class Node (and subsequently, it won't inherit its member methods and variables).

CODING CONVENTION

When you are coding, there are usually naming conventions based on the language that you are using. These often provide better clarity for your code and depend on the language that you will be using. In **GDScript**, you will usually need to do the following:

- Name **classes** using Pascal Casing (i.e., a script): In Pascal casing the first letter of each word included in a name is capitalized; for example: **MyClass**.
- Name **variables** using Snake Casing: In Snake casing the words included in a name are using lower-case and are separated by an underscore; for example: **my_variable**.
- Name **functions** using Snake Casing: In Snake casing the words included in a name are using lower-case and are separated by an underscore: for example, **my_function**.
- When you create functions or blocks of instructions, the code within needs to be indented.
- Use, as much as possible, one statement per line.

Once you feel comfortable with **GDScript** and want to know more about the official naming scheme, you may look at Godot official naming guidelines:

http://docs.godotengine.org/en/stable/getting_started/ script-ing/gdscript/gdscrip-t_styleg-uide.htming-con-ventions

A FEW THINGS TO REMEMBER WHEN YOU CREATE A SCRIPT (CHECK-LIST)

As you create your first scripts in the next chapter, there will be, without a doubt, errors and possibly hair pulling :-). You see, when you start coding, you will, as for any new activity, make small mistakes, learn what they are, improve your coding, and ultimately get better at writing your scripts. As I have seen students learning to code, there are some common errors that are usually made. These don't make you a bad programmer; on the contrary, it is part of the learning process.

We all learn by trial and error, and making mistakes is part of the learning process.

So, as you create your first script, set any fear aside, try to experiment, be curious, and get to learn the language. It is like learning a new foreign language: when someone from a foreign country understands your first sentences, you feel so empowered! So, it will be the same with GDScript, and to ease the learning process, I have included a few tips and things to keep in mind when writing your scripts, so that you progress even faster. You don't need to know all of these by now (I will refer to these later on, in the next chapter), but just be aware of it and also use this list if any error occurs (this list is also available as a pdf file in the resource pack, so that you can print it and keep it close by). So, watch out for these :-).

- Indent your code properly for functions or blocks of instructions.
- All variables are written consistently (e.g., spelling and case). The name of each variable is case-sensitive; this means that if you declare a variable **my_-vari able** but then refer to it as **my_Variable** later on in the code, this may trigger an error, as the variable **my_Variable** and **my_variable**, because they have a different case (upper- or lower-case **V**), are seen as two different variables.
- All variables are declared before being used.

- The type of the argument passed to a function is the type that is required by this function.
- The type of the argument returned by a function is the type that is required to be returned by this function.
- Built-in functions are spelled with the proper case (e.g., **_ready**).
- Use **Snake casing** for variables and functions and or **Pascal casing** for file and class names.
- When calling a function, the exact name of this function (i.e., case-sensitive) is used.
- When referring to a variable, it is done with regards to the access type of the variable (e.g., member or local).
- Local variables are declared and can be used within the same function.
- Member variables are declared outside functions and can be used anywhere within the class.

LEVEL ROUNDUP

Summary

In this chapter, we have become familiar with GDScript and different programming concepts. We also looked into object-oriented programming. In the next chapter, we will harness these skills to be able to create (and execute) our first script.

Quiz

It is now time to test your knowledge.

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following statement will print the text **Hello World** in the **Output window**.

```
print("Hello World");
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **c** in the following statement will be **3**.

```
var a:int=1
var b:int=1
var c:int = a + b;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **full_name**, in the following code snippet, will be **JohnPaul**.

```
var fName:Sring = "John";
var lName:String = "Paul";
var full_name:String = fName + lName;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code snippet will print **I will not go sailing**.

```
var wind_is_strong:bool
wind_is_strong = true;
if (wind_is_strong): print ("I will not go sailing");
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code snippet will print **I will not go sailing**.

```
var weather_is_sunny:bool = true
var wind_is_strong:bool = false
var i_will_go_sailing:bool
if (weather_is_sunny && !wind_is_strong ): print ("I will go sailing")
if (!weather_is_sunny || wind_is_strong ): print ("I will not go sailing")
```

1. Spot three coding mistakes in the following snippet.

```
var test:int
var test2:int;
test3 = 0;
test 3 = test1 + test2;
```

1. Consider the method described in the next code snippet, and select the correct way to call it (i.e., A, B, or C):

   a) **display_A_message()**
   b) **displayAMessage()**
   c) **display_a_message()**

```
func display_a_message ():
print (“Hello”)
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The value of the variable **counter** in the following code snippet will be **3** after the code has been executed.

```
var counter:int
counter = 0;
counter = counter + 1;
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code will print the message **Hello**

```
func print_message():
print (“Hello”);
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

A local variable can be used from any part of a script.

Answers to the Quiz.

1. TRUE.
2. FALSE.
3. TRUE.
4. TRUE
5. FALSE.
6. Spot three coding mistakes in the following snippet.

```
var test:int
var test2:int;
test3 = 0;#test3 was not declared
test 3 = test1 + test2;#no space between test and 3
```

1. **C.**
2. FALSE (should be 1).
3. TRUE.
4. FALSE.

Checklist

If you can do the following, then you are ready to proceed to

- Understand the concept of classes.
- Know how to call a method.
- List and understand at least three types of variables in GDScript.
- Answer at least 7 out of 10 of the questions correctly in the quiz.

Chapter 2: Creating your First Script

In this section, we will start to create GDScript code in Godot. Some of the objectives of this section will be to:

- Introduce GDScript in Godot.
- Explain some basic scripting concepts.
- Explain how to display information from the code to the **Output window**.

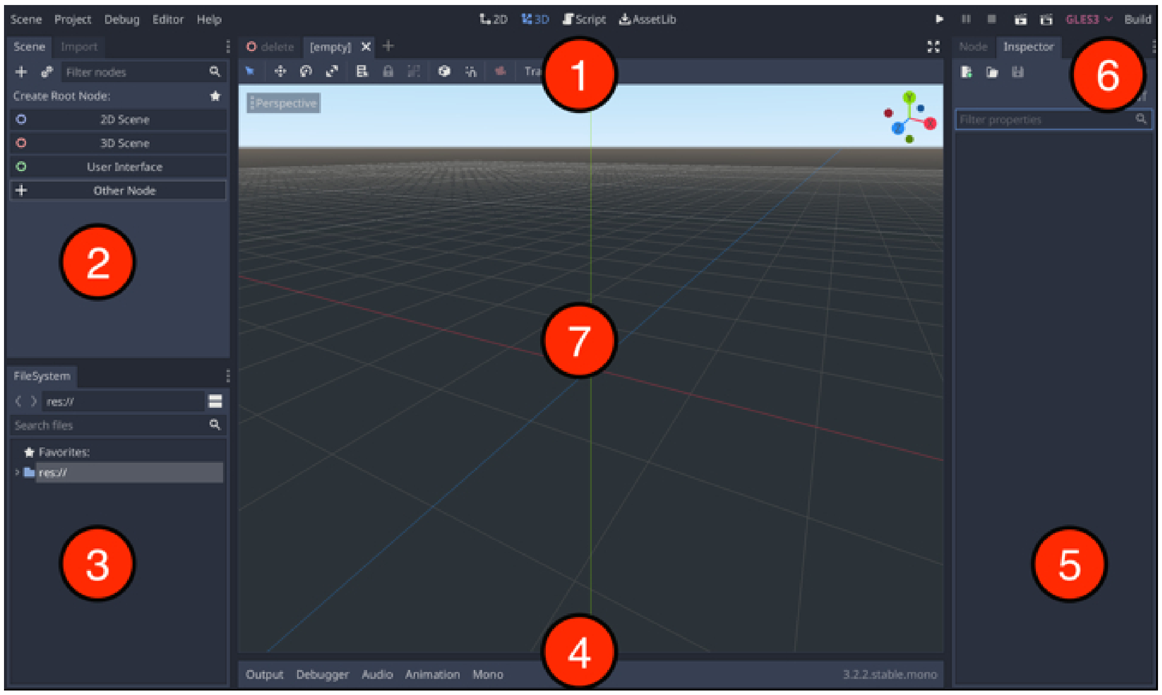
After completing this chapter, you will be able to:

- Understand basic concepts in GDScript.
- Understand the best coding practices.
- Code your first script in Godot.
- Create classes, methods and variables.
- Instantiate objects based on your own classes.
- Use built-in methods.
- Use conditional statements.

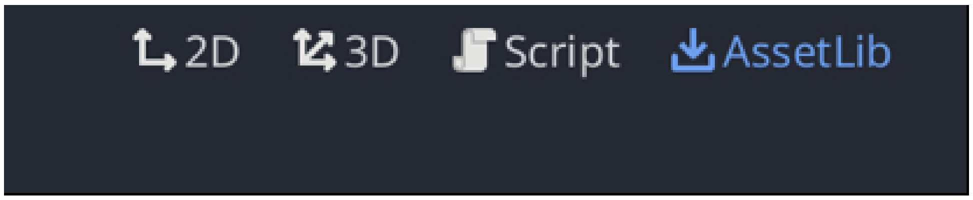
You can skip this chapter if you are already familiar with GDScript or if you have already created and used GDScript scripts within Godot.

Quick overview of the interface

Godot includes several windows organized in a (default) layout. Each of these windows includes a label in their top-left corner. These windows can be moved around and rearranged, if necessary, by either changing the layout (using the menu **Editor | Editor Layout | ...**) or by dragging and dropping the corresponding tab for a window to a different location. This will move the view/panel (or window) to where you would like it to appear onscreen. In the default layout, the following views appear onscreen.

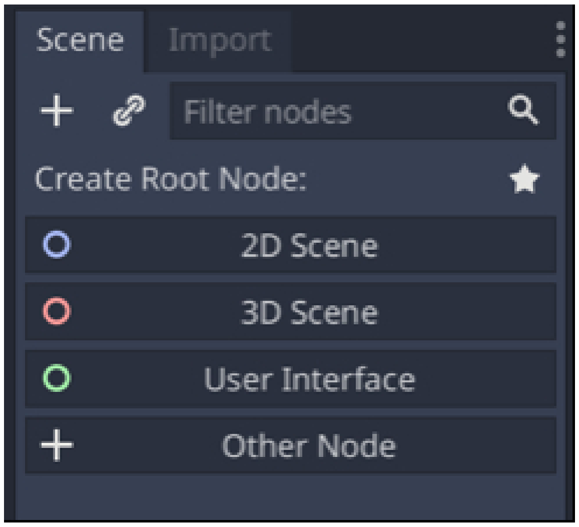


1. The top tabs: these workspaces are used to visualize a 3D scene, or a 2D scene, the scripts included in your scene, and the different assets that you can avail of for your project.

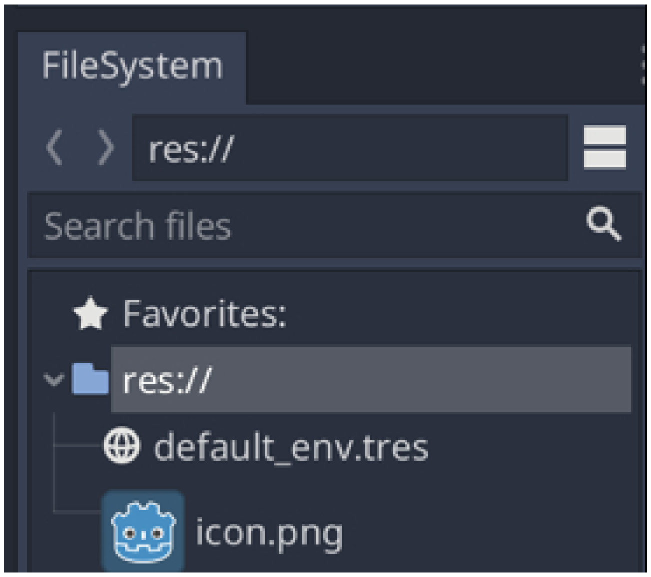


1. The **Scene** tab: this window or view lists all the nodes currently present in your scene. These could include, for example, basic shapes, 3D characters,

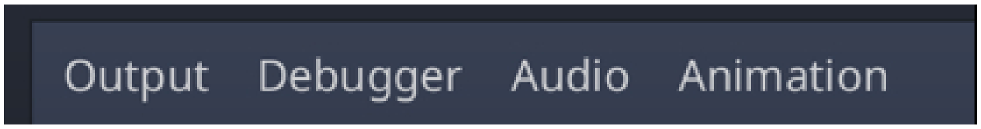
or terrains. This view also makes it possible to identify a hierarchy between nodes, and to identify, for example, whether an object has children or parents (we will explore this concept later).



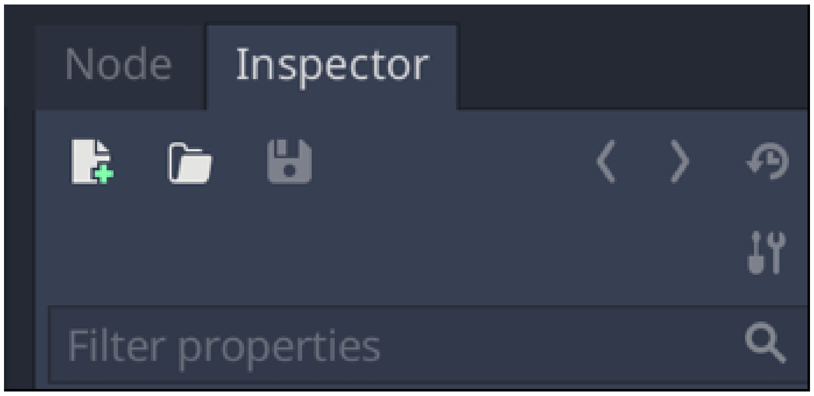
1. The **FileSystem** tab: this window or tab includes all the assets available and used for your project, such as 3D models, sounds, or textures.



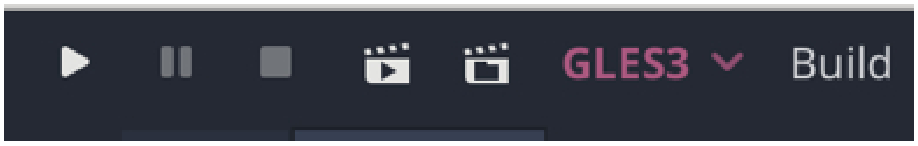
1. The bottom tabs: these tabs include information related to your actions in Godot, as well as compile errors, amongst other things. More specifically information will be related to animation, audio, compilation, messages from your code, and actions in Godot.



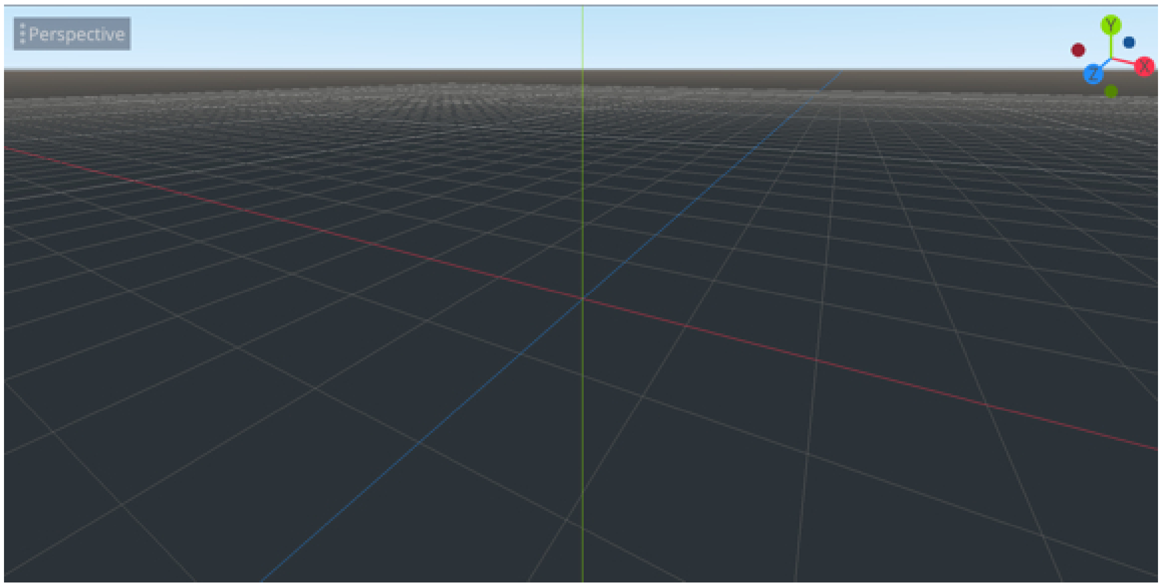
1. The **Inspector** tab: this tab displays information (i.e., the properties) on the asset or the node that is currently selected.



1. The **Play-Test** buttons (located in the top right corner): these buttons make it possible to play/pause/stop/build the current project or scene.



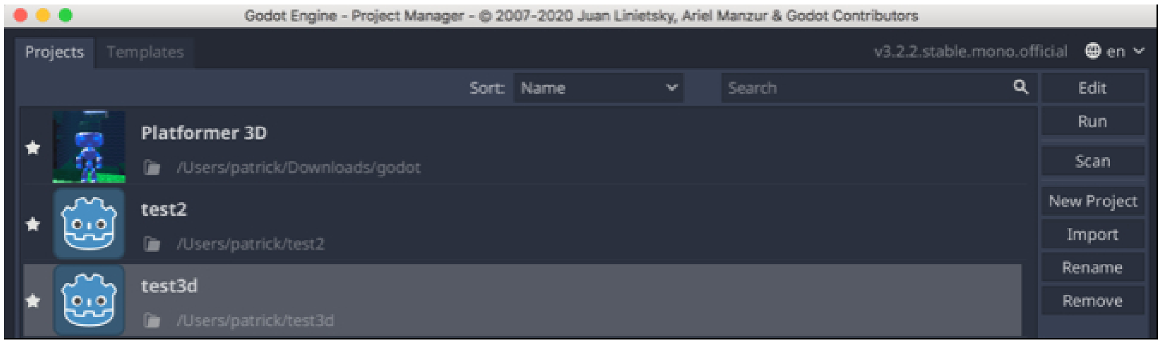
1. The **Viewport**: this tab located in the middle of the screen displays the content of a scene (or the item listed in the **Hierarchy** view) so that you can visualize and modify them accordingly (e.g., move, scale, etc.).



Getting started

In Godot, a script is usually linked to an object; although it can also be used as a standalone class to be instantiated later. Generally, for your script to be executed, it will need to be linked to an object. So to start with, we will create an empty object, create a script, and link this script to the object.

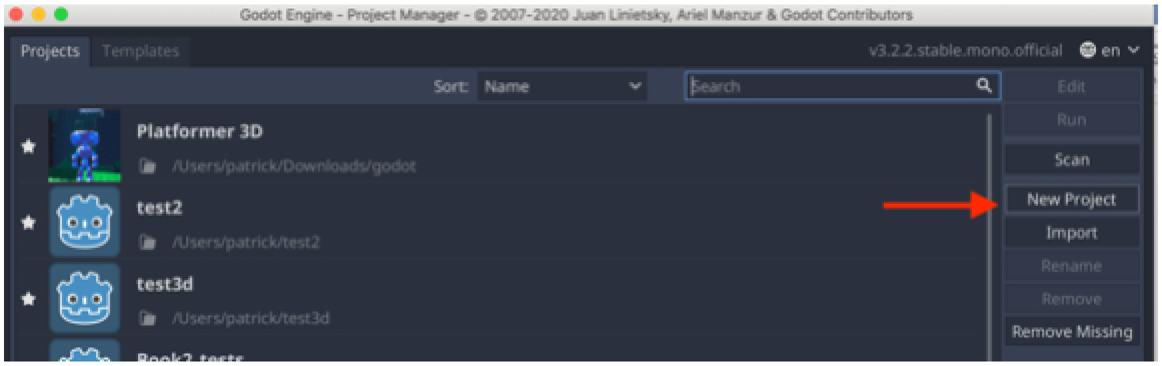
- Please launch Godot.
- After launching Godot, a new window will be displayed as follows.


Godot Engine - Project Manager - © 2007-2020 Juan Linietsky, Ariel Manzur & Godot Contributors
Projects
Templates
v3.2.2.stable.mono.official
en
Sort:
Name
Search
Edit
Run
Scan
New Project
Import
Rename
Remove
Platformer 3D
/Users/patrick/Downloads/godot
test2
/Users/patrick/test2
test3d
/Users/patrick/test3d

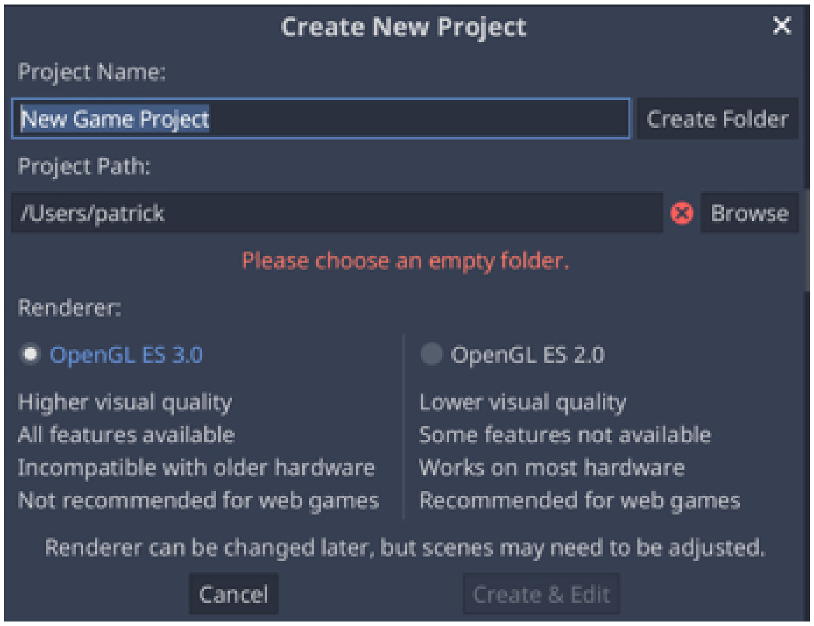
This window includes two tabs: a tab called **Projects** that lists a default project called "**Platformer 3D**" and your current projects (if any), and a tab called "**Templates**" where you will be able to find demo projects and templates for different game genres and their associated features.

For now, we will create a new project to become familiar with the interface.

- Please click on the button labelled **New Project**.



- A new window will appear as illustrated in the next figure:


Create New Project

Project Name: New Game Project

Project Path: /Users/patrick

Please choose an empty folder.

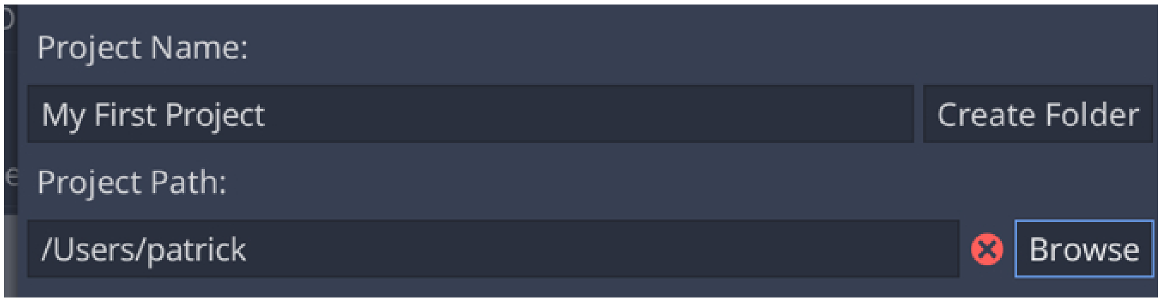
Renderer:

● OpenGL ES 3.0	○ OpenGL ES 2.0
Higher visual quality	Lower visual quality
All features available	Some features not available
Incompatible with older hardware	Works on most hardware
Not recommended for web games	Recommended for web games

Renderer can be changed later, but scenes may need to be adjusted.

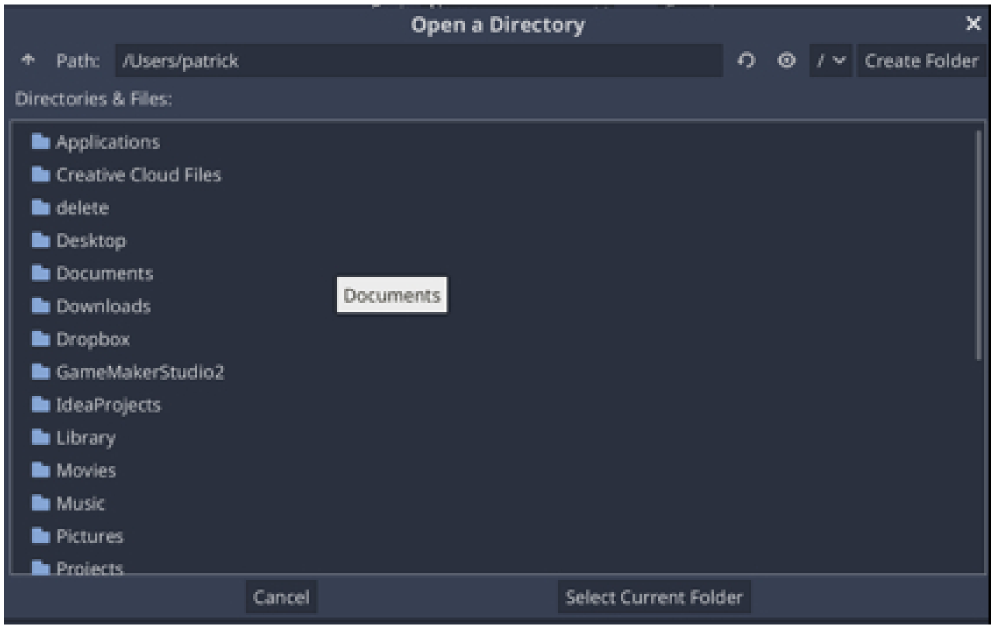
As we will see in the next steps, this window will make it possible for you to provide a name and a location for your project.

- Please enter a name for your project in the field labelled "**Project Name**", for example "**My First Project**".
- By default, your project will be saved in your home folder; this being said, if you prefer to save it in a different location, please click on the button labelled "**Browse**" to select the location of your choice.


Project Name: My First Project

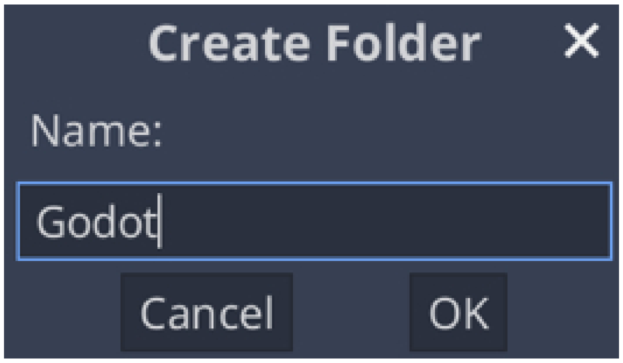
Project Path: /Users/patrick

- This will open a new window where you will be able to select a folder for your project.

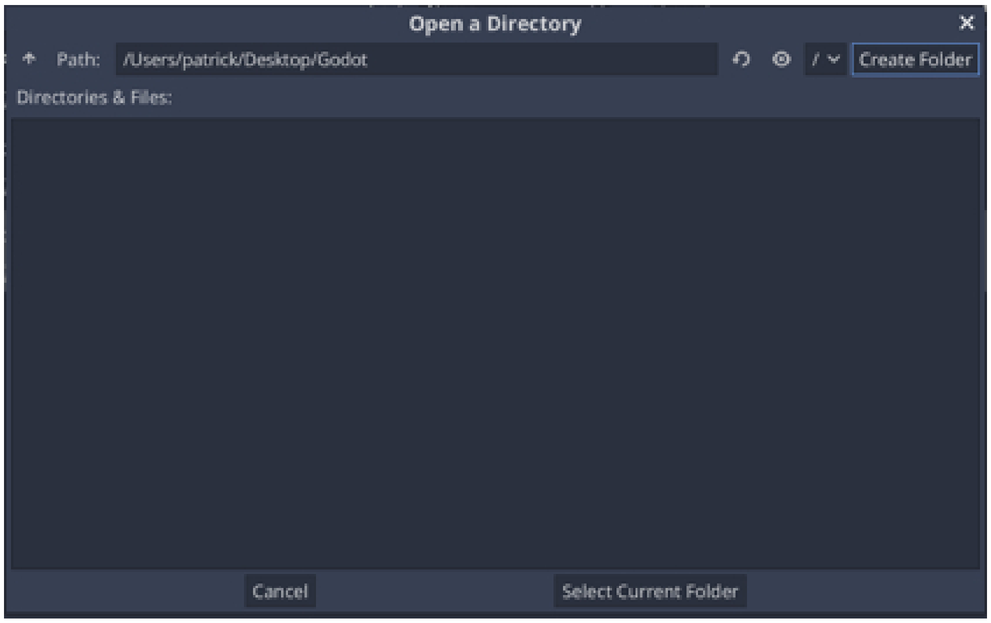


Let's create a folder called "**Godot**" on our desktop (you can create a folder in another location if you wish) so that the project can be saved within that folder.

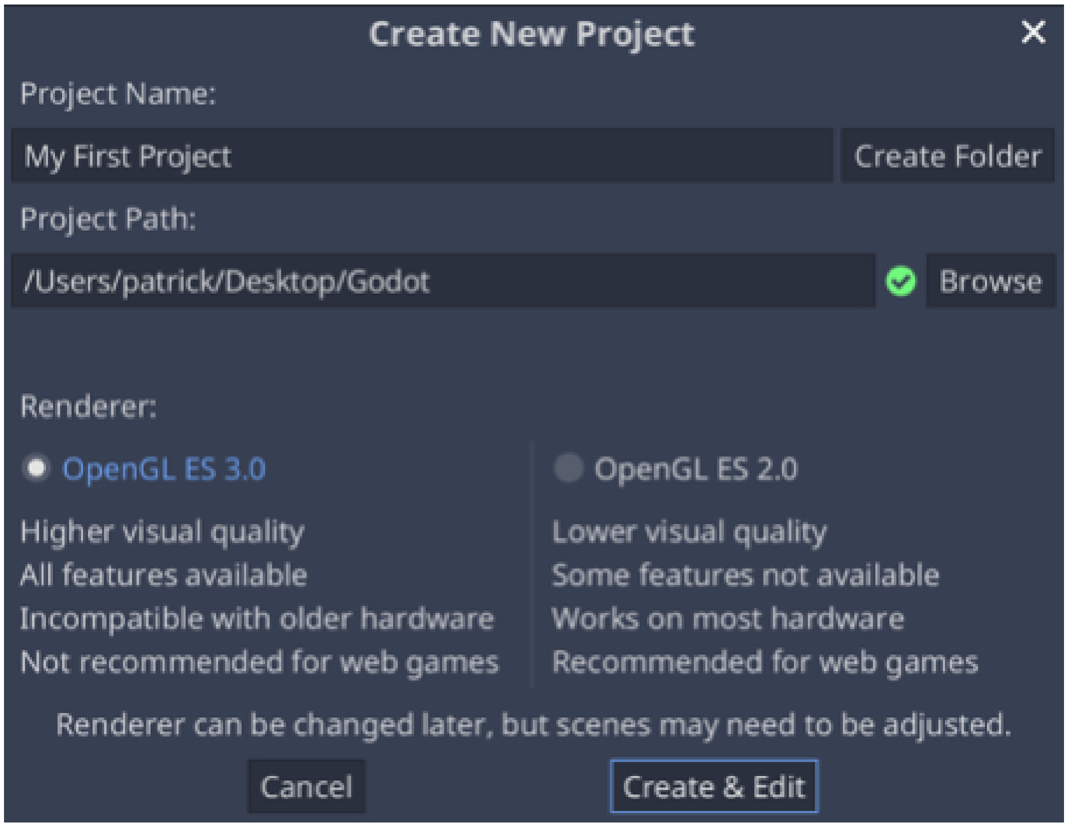
- Please select the folder of your choice from the list (in my case this is the folder called **Desktop**) by double-clicking on it.
- Click on the button labelled "**Create Folder**" to create a folder within the folder called "**Desktop**" that you have just selected. This will open a new window where you can specify the name of the new folder.



- Please type the name of the new folder and click on the button labelled "**OK**".
- Once this is done, you can click on the button labelled "**Select Current Folder**" so that the folder that you have just created is used for your project.



- In the new window, you can then click on the button labelled "**Create and Edit**".


Create New Project

Project Name: My First Project

Project Path: /Users/patrick/Desktop/Godot

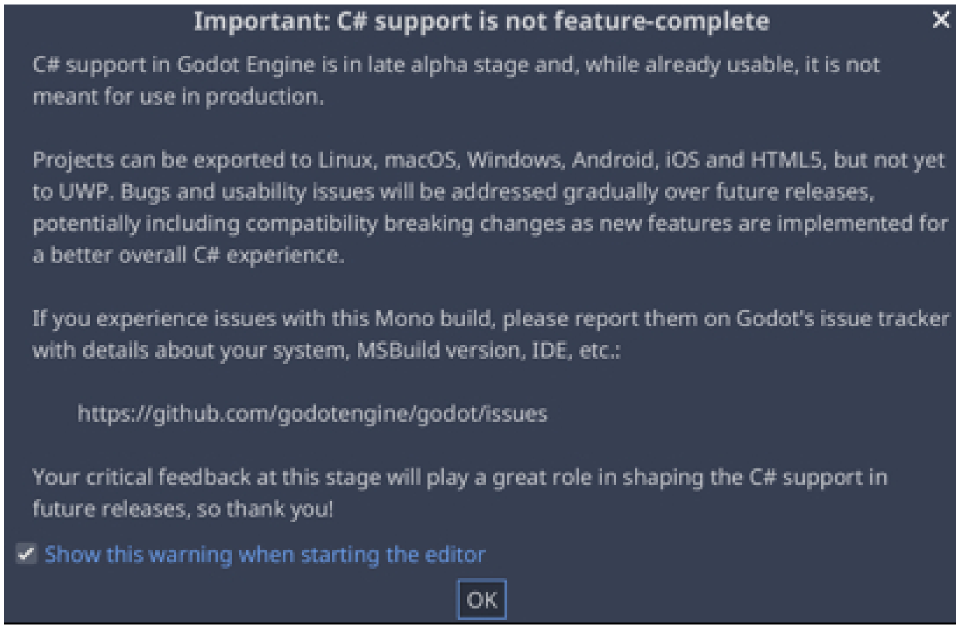
Renderer:

● OpenGL ES 3.0	○ OpenGL ES 2.0
Higher visual quality	Lower visual quality
All features available	Some features not available
Incompatible with older hardware	Works on most hardware
Not recommended for web games	Recommended for web games

Renderer can be changed later, but scenes may need to be adjusted.

- At this stage your new project should open.

- If the following message appears, please click "**OK**".



Important: C# support is not feature-complete

C# support in Godot Engine is in late alpha stage and, while already usable, it is not meant for use in production.

Projects can be exported to Linux, macOS, Windows, Android, iOS and HTML5, but not yet to UWP. Bugs and usability issues will be addressed gradually over future releases, potentially including compatibility breaking changes as new features are implemented for a better overall C# experience.

If you experience issues with this Mono build, please report them on Godot's issue tracker with details about your system, MSBuild version, IDE, etc.:

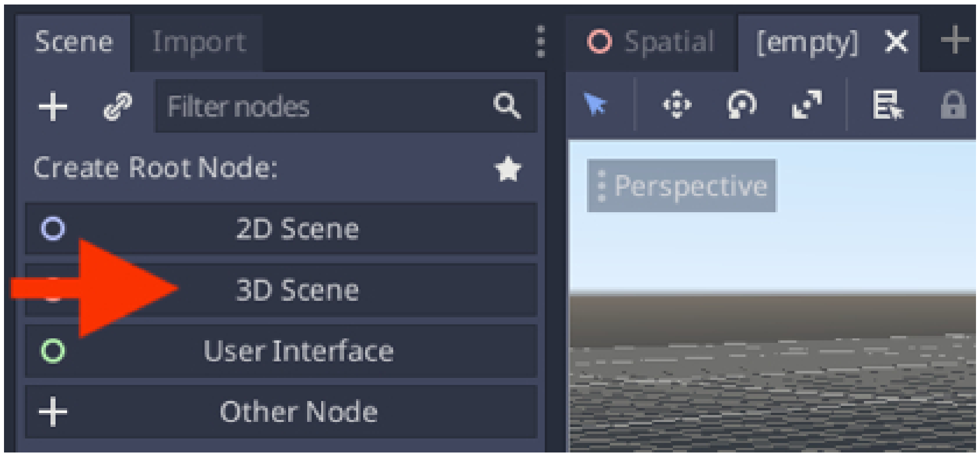
https://github.com/godotengine/godot/issues

Your critical feedback at this stage will play a great role in shaping the C# support in future releases, so thank you!

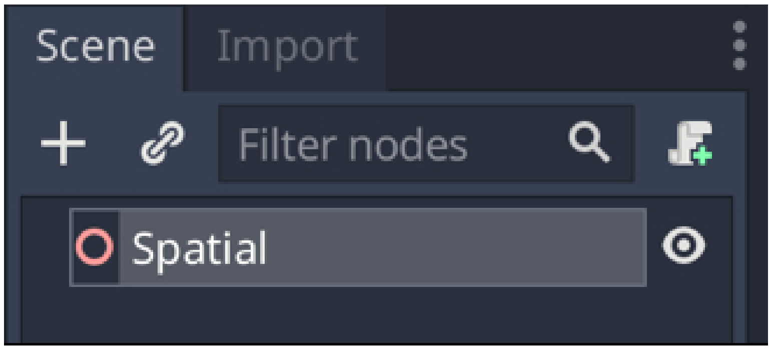
✓ Show this warning when starting the editor

OK

- Create a new scene (**Scene | New Scene**).
- Select the option **3D Scene** from the **Scene** tab, as per the next figure.



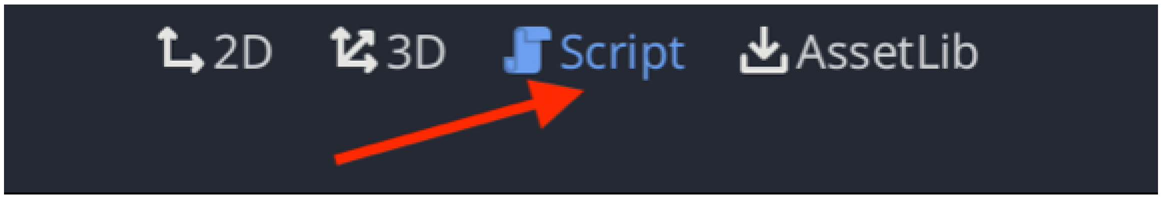
- By default, this should create an empty scene with a **Spatial** Node.



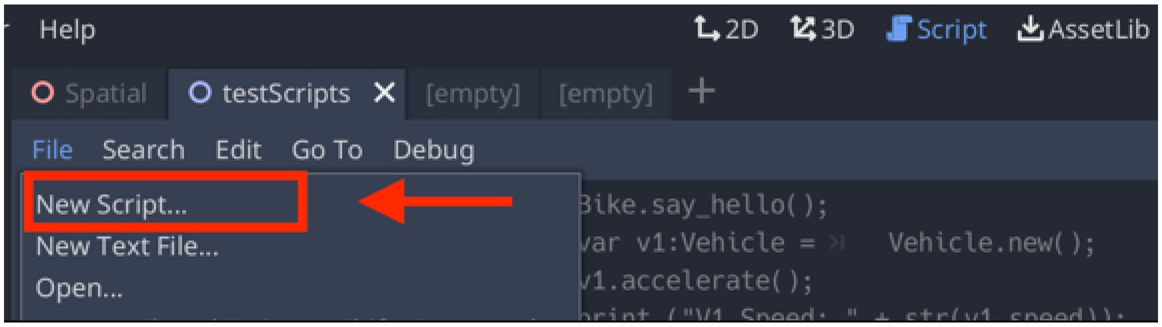
- Rename this node **example_for_scripting** in the **Scene** tab. To do so, you can either right-click on this node and then select the option **Rename**, from the contextual menu, or select the node **Spatial Node** (i.e., click once on it) and press ***CTRL*** + ***Enter***.

Now that this node has been renamed, let's create a new script:

- Select the **Script** workspace by pressing the corresponding button at the top of your screen.

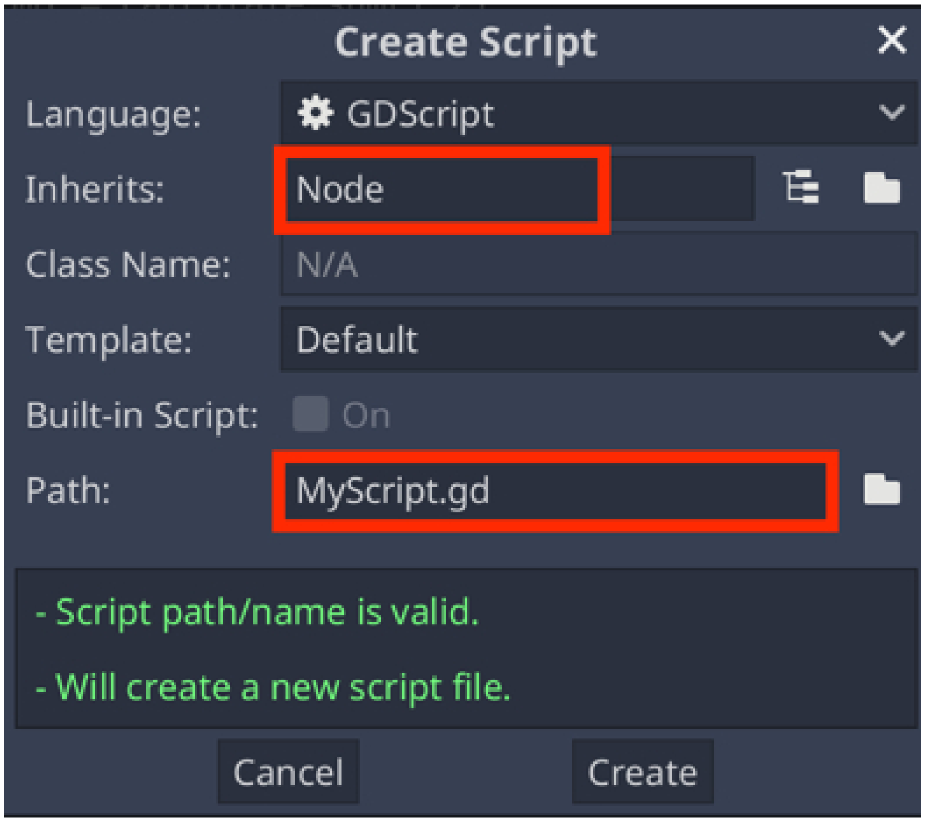


- Select: **File | New Script** from the left menu.



```
Bike.say_hello();
var v1:Vehicle =    Vehicle.new();
v1.accelerate();
print ("V1 Speed: " + str(v1.speed));
```

- In the new window, specify a **Path** (i.e., a name), for example "**MyScript**" for your script; by default, your script will inherit from the class **Node** and it will be saved in the root folder (res), as illustrated in the next figure.



Create Script

Language: GDScript

Inherits: Node

Class Name: N/A

Template: Default

Built-in Script: On

Path: MyScript.gd

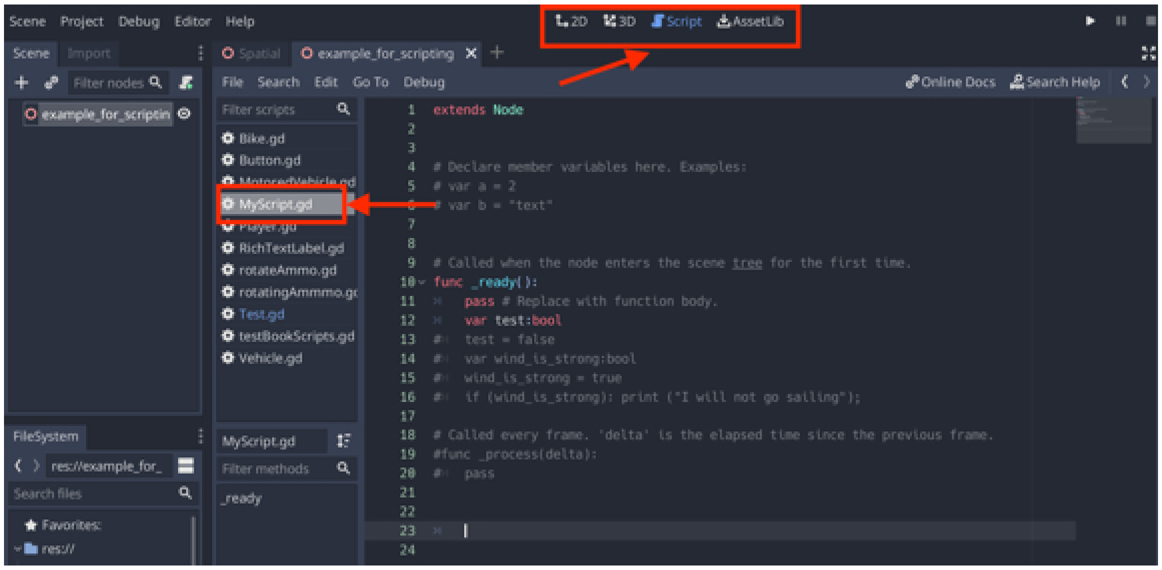
- Script path/name is valid.

- Will create a new script file.

- Click on the button labelled “**Create**”.

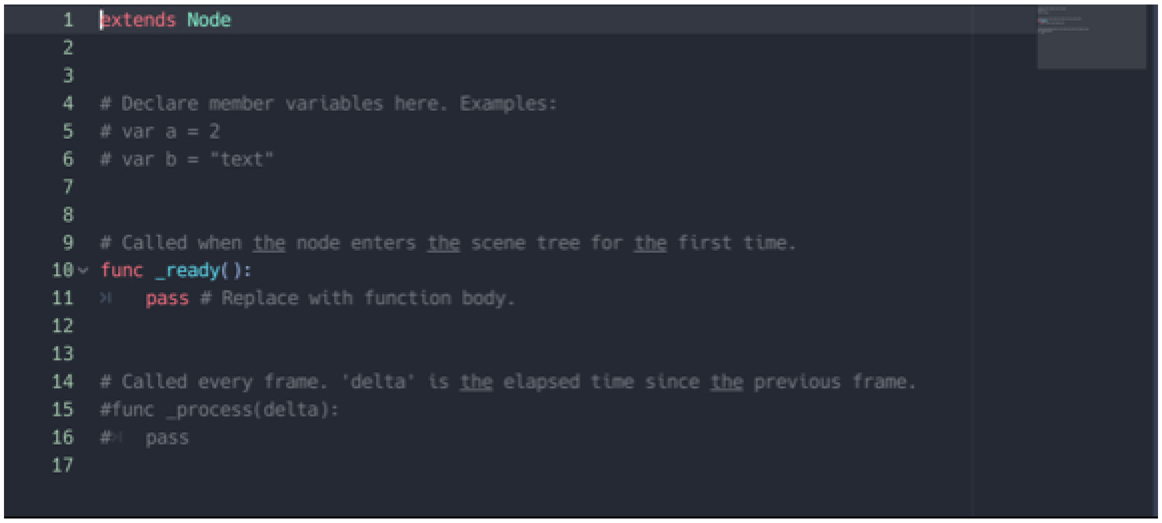
At this stage our script has been created, so let’s open it to modify it:

- Click on the button labelled **Scrip**t in the top menu.
- Then click on the name of your script in the left-hand side column.



```
extends Node


# Declare member variables here. Examples:
# var a = 2
# var b = "text"


# Called when the node enters the scene tree for the first time.
func _ready():
	pass # Replace with function body.
	var test:bool
#	test = false
#	var wind_is_strong:bool
#	wind_is_strong = true
#	if (wind_is_strong): print ("I will not go sailing");

# Called every frame. 'delta' is the elapsed time since the previous frame.
#func _process(delta):
#	pass



```

- This should display the content of the script as follows.



```
extends Node


# Declare member variables here. Examples:
# var a = 2
# var b = "text"


# Called when the node enters the scene tree for the first time.
func _ready():
	pass # Replace with function body.


# Called every frame. 'delta' is the elapsed time since the previous frame.
#func _process(delta):
#	pass

```

So let's start coding.

First let's create a variable of type **integer** called **number** as a member variable.

- In the editor, please type the following code just after the line that says "**extends Node**:

var number:int

- As you can see, the variable is declared outside any method but inside the class **Class/File**, which means that it is a member variable.
- Then, we can declare a **String** variable called **my_name**. So please type the following code just after the previous declaration.

var my_name:String

- Remove the line that starts with the code "**pass**" in the **_ready** function.
- Then type the following code inside the method **_ready** (i.e., use tab to indent your code).

number = 12

- This code sets the variable **number** to **12**; this variable was declared at the start of the script, as a member variable and it can be accessed from anywhere within the class **File/Class**, including from inside the method **_ready**.

- Then type the following code inside the method **_ready** after the previous statement (you can replace the word **Patrick** with your own name if you wish):

```
my_name = “Patrick”
```

- As you type this line, make sure that the name of the variable is spelled properly with proper case (i.e., lower-case letters).

If you happen to copy/paste this code, please ensure that you are using straight double quotes otherwise you may get an error.

- Then type the following code after the previous statement to display a message in the **Output window**.

```
print ("Hello " + my_name + " , your number is "+ str(number))
```

- This should print the message **“Hello Patrick, your number is 1”** in the **Output window** in Godot after the code has been executed. This window displays error messages from Godot or messages from your code. You may notice the quotes around the word **Patrick**, this means that the text **Hello** will be displayed and we will add the value of the variable **my_name** to it. So these two strings will be concatenated (i.e., grouped) to form a dynamic sentence (i.e., a sentence for which the content will vary) for which the content will depend on the value of the variables **my_name** and **number**. You may also note that we use the code **str(number)**, and this is to convert the number to a **String** so that it can be concatenated to the rest of the sentence.

So at this stage, your code should look as follows (and if it doesn’t, you can use the next code snippet as a template):

```
extends Node
var number:int
```

var my_name:String

# Declare member variables here. Examples:

# var a = 2

# var b = "text"

# Called when the node enters the scene tree for the first time.

func _ready():

number = 12

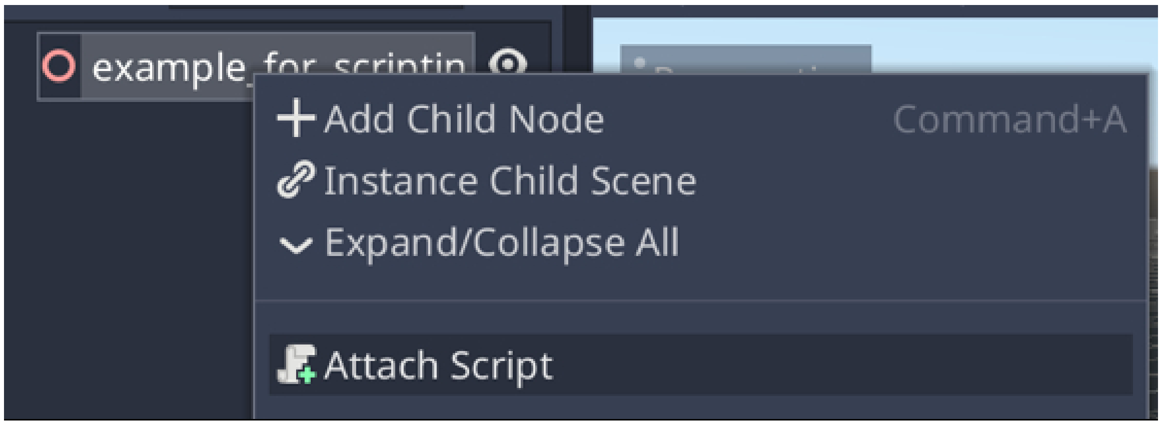
my_name = "Patrick"

print ("Hello " + my_name + " , your number is "+str(number))

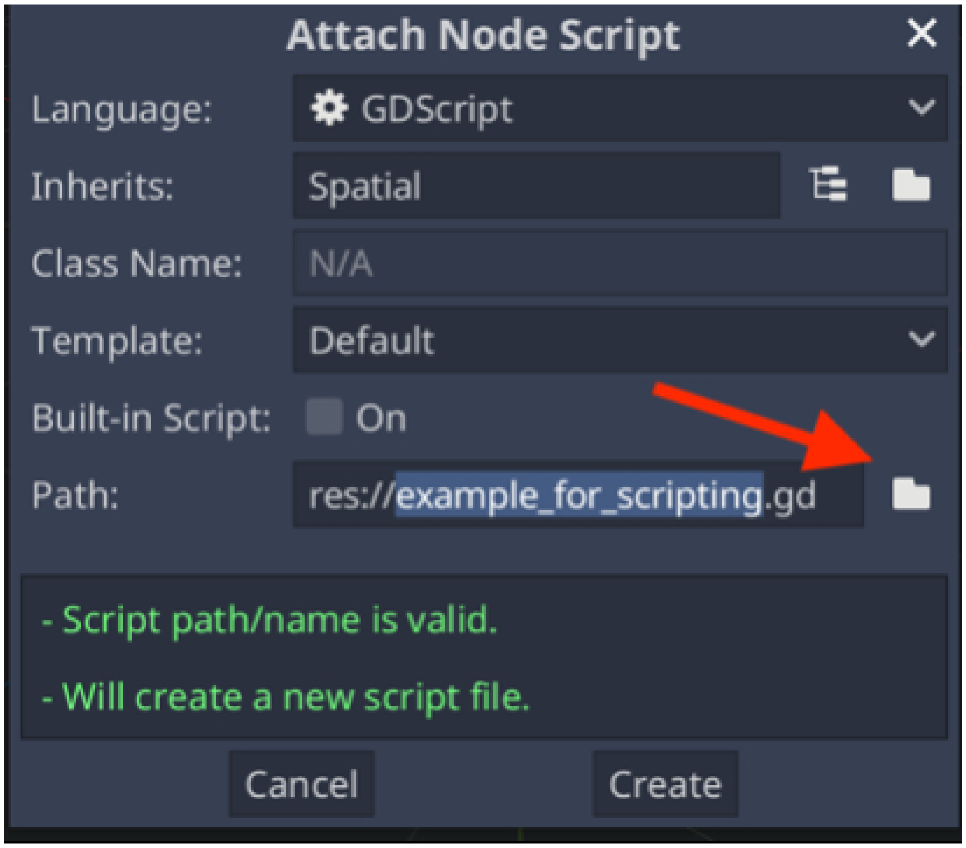
- At this stage, we can save our script (**File | Save As...**) as **MyScript.gd**. By default, and based on how we created the script, it will be saved in the root folder of the project.

We now need to attach this script to an object:

- In Godot, right-click on the node called **example_for_scripting**, and select the option "**Attach Script**" from the contextual menu, as described in the next figure.



- in the new window, please select the location of the script by clicking on the folder icon to the right of the label called **Path**, as illustrated in the next picture.



Attach Node Script

Language: GDScript

Inherits: Spatial

Class Name: N/A

Template: Default

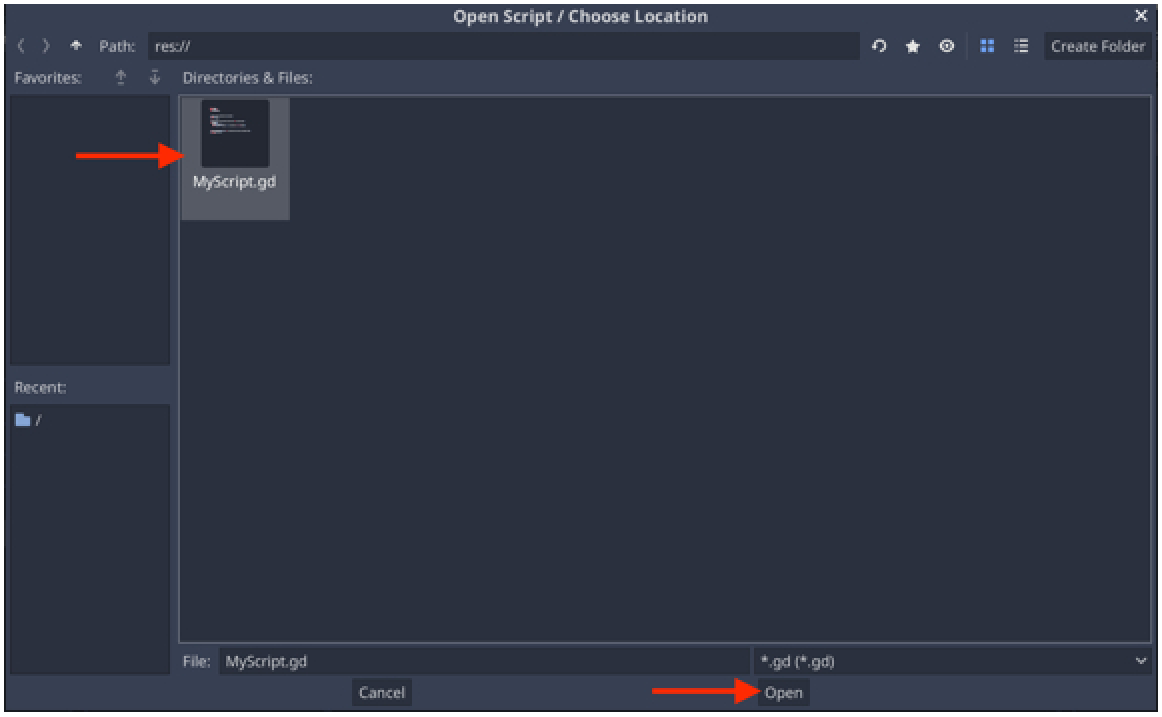
Built-in Script: On

Path: res://example_for_scripting.gd

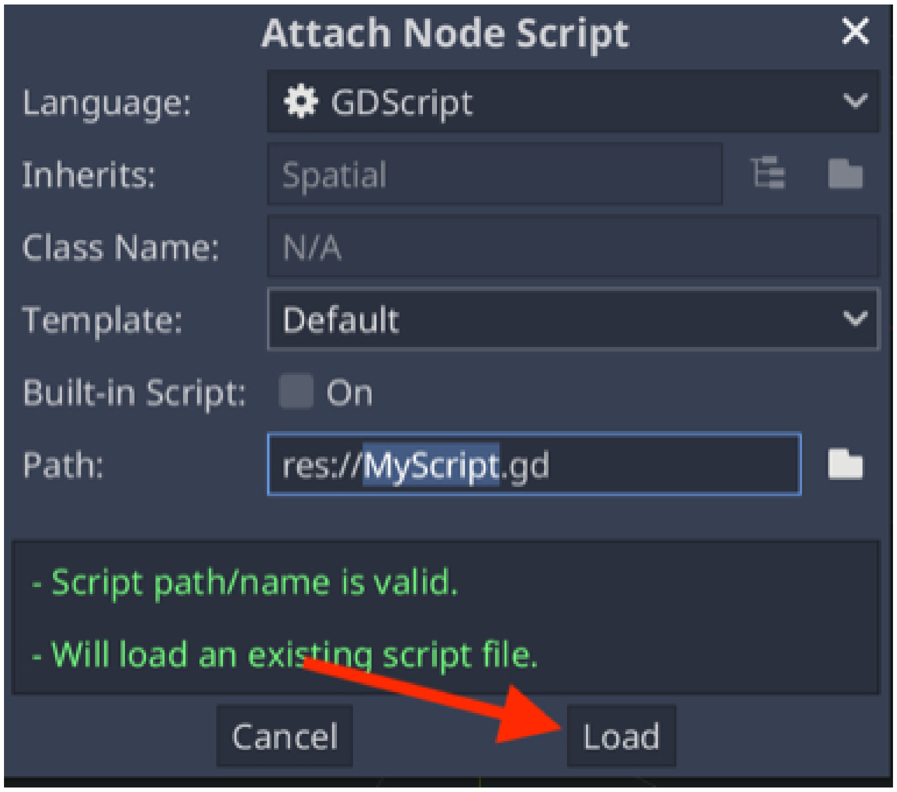
- Script path/name is valid.

- Will create a new script file.

- This will open a new window where you can select the script that you want to attach to the node.



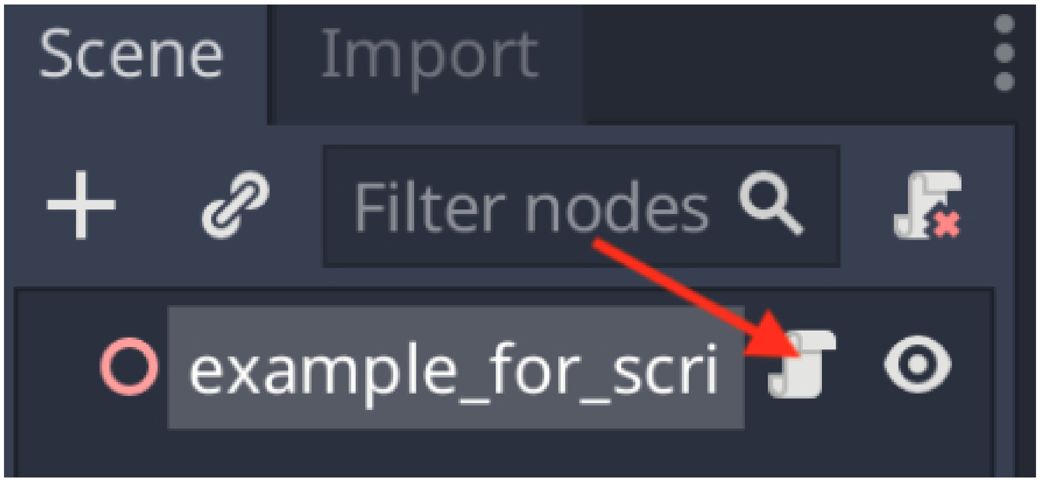
- Please select the script **MyScript.gd** and click on the button labelled "**Open**".
- This will reopen the previous window.
- You can then click on the button labelled "**Load**".



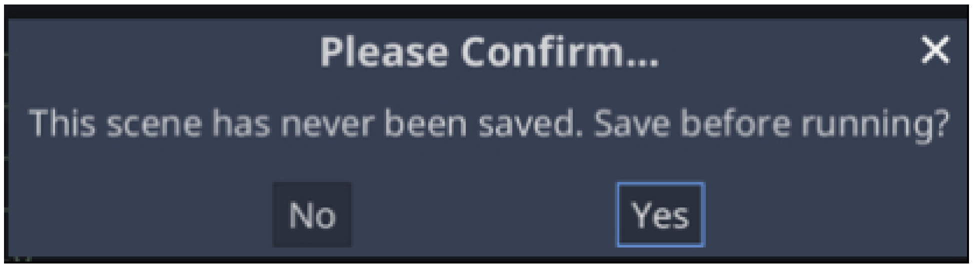
Language: GDScript
Inherits: Spatial
Class Name: N/A
Template: Default
Built-in Script: On
Path: res://MyScript.gd

- Script path/name is valid.
- Will load an existing script file.

- At that stage, you should see that a script logo is now featured to the right of the node, as illustrated in the next figure.

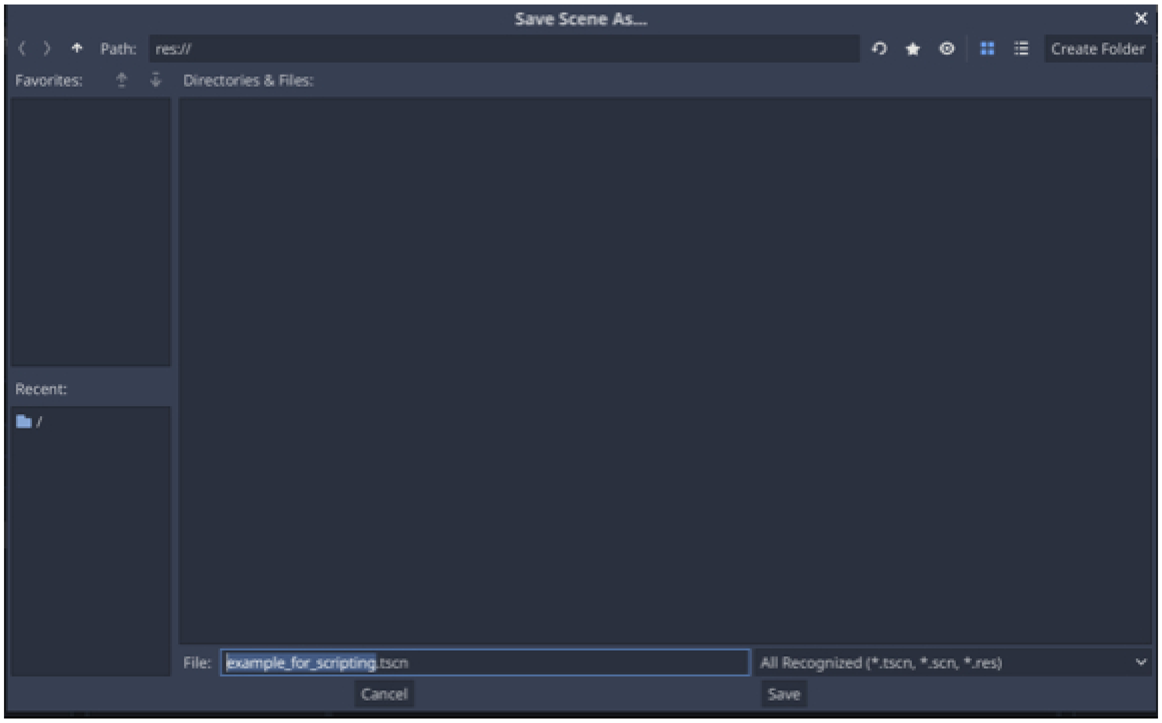


- Look at the **Output window** to see if there are any errors.
- We can now play the scene (***F6 FOR WINDOWS OR CMD + R FOR MACOS***).
- Godot may ask you to save the scene and you can confirm by clicking "**Yes**".

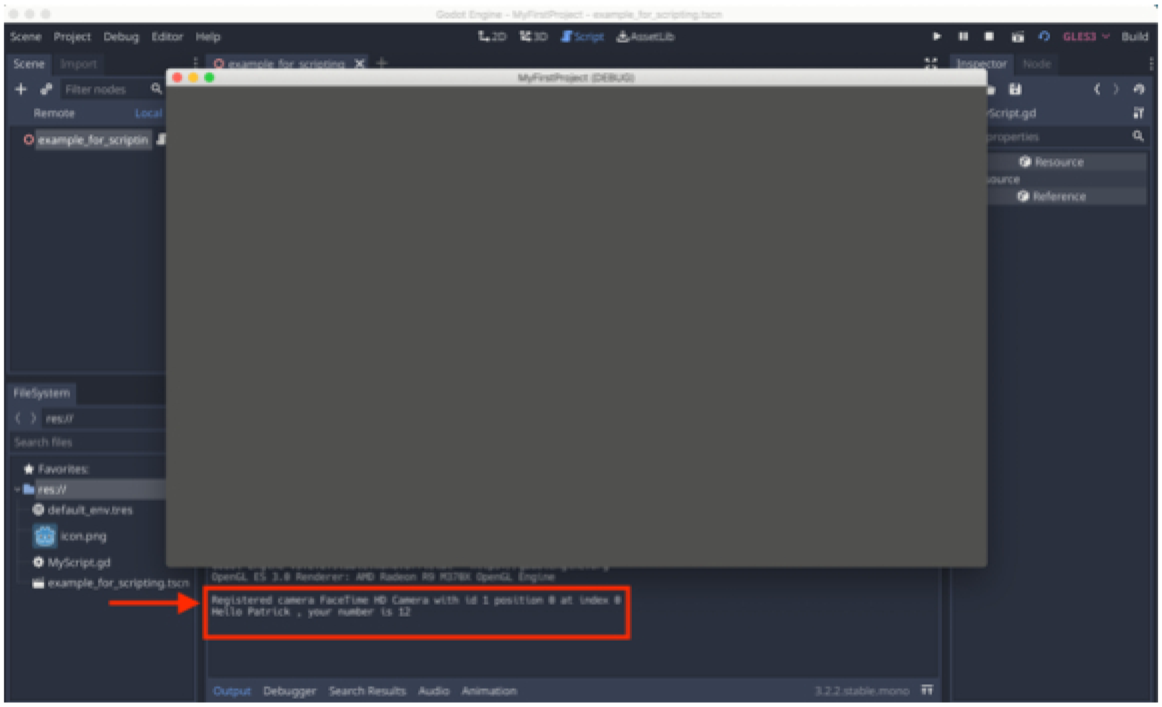


This scene has never been saved. Save before running?

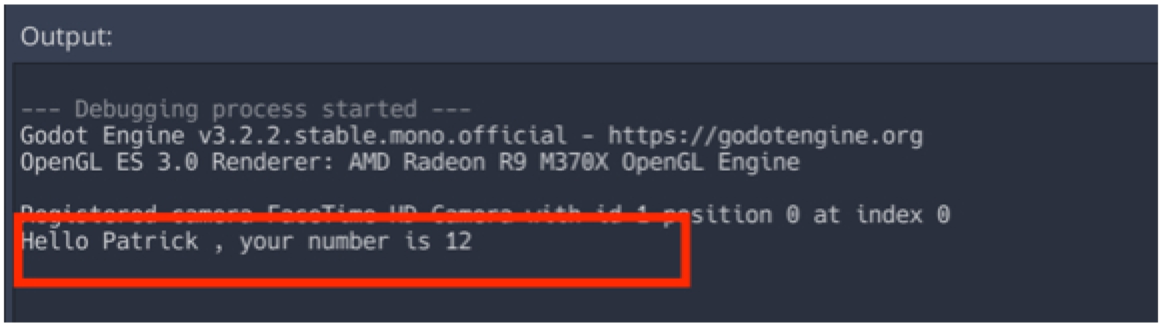
- You can save your scene as **example_for_scripting.tscn** (default name), or any other name of your choice.



- As we play the scene and look at the **Output window**, we should see the message “**Hello Patrick, your number is 12**”.



```
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine
Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Hello Patrick , your number is 12
```



```
Output:

--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Hello Patrick , your number is 12
```

This is it! We have created our first script using the built-in method **_ready** and some variables, more specifically member variables. These variables are of type **integer** and **String**. Again, these variables are member variables as they were declared at the start of the class and outside any method. They can, as a result, be used across the class. The full script should look as described in the next code snippet.

```
extends Node
var number:int
var my_name:String
# Declare member variables here. Examples:
# var a = 2
# var b = "text"
```

```
# Called when the node enters the scene tree for the first time.
    func _ready():
    number = 12
    my_name = "Patrick"
    print ("Hello " + my_name + " , your number is "+str(number))
    pass # Replace with function body.
```

```
# Called every frame. 'delta' is the elapsed time since the previous frame.
    #func _process(delta):
```

# pass

You can stop playing the scene by either closing the grey window or by switching to Godot and pressing **CTRL + Period**.

Using the _process Function

Let's now use the method **_process** to display another message in the **Output window**. Again, this method is called every frame (i.e., every time the screen is refreshed), so any message printed within this method will be displayed indefinitely and every frame.

- Please switch to the **Script** workspace.
- Open the script **MyScript** if it is not already open.
- Modify the function **_process** as below

# Called every frame. 'delta' is the elapsed time since the previous frame.

func _process(delta):

print(my_name)

- Save your code (**File | Save** or **ALT + CTRL + S**).
- Play the Scene (**F6 FOR WINDOWS OR CMD + R FOR MACOS**).
- You should see that the message **Patrick** (or your own name) is displayed indefi nitely in the **Output** window.

Creating local variables

At this stage, the code is working well, and we have created two member variables: **number** and **my_name,** that are accessible throughout our class. However, we could also create variables that are only accessible from within a function (i.e., local variables).

- Please switch back to your code.
- Delete or comment the code that we have just created in the **_ready** method. To comment code, you can use hashtags, as described in the next code snippet.

```
# number = 12
# my_name = "Patrick"
# print ("Hello " + my_name + " , your number is "+str(number))
```

- Modify the function **_process** as follows:

```
# print(my_name)
pass
```

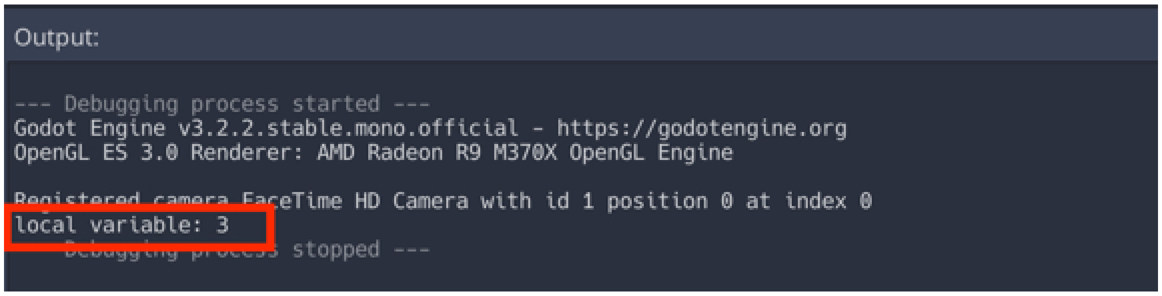
In the previous code, we commented the code that prints the value of the variable **my_name** and we also add the keyword pass. This is because the function once it has been declared needs to include instructions. The keyword pass does just that without actually doing anything, and is often used for empty function.

- Add the following code to the function **_ready**, ensuring that your code is indented (new code in bold).

```
func _ready():
# number = 12
# my_name = "Patrick"
# print ("Hello " + my_name + " , your number is "+str(number))
var local_variable:int = 3;
print("local variable: " + str(local_variable))
```

With the first statement, we declare a variable that should only be used locally, that is to say, within the function ***ready****. We then print the value of this variable and display a message that includes the string* ***"local variable"*** *that will be followed by (or appended to) the value of the variable* **local_variable**; in our case, this should display **"local variable: 3"**.

- Check the code that you have written and ensure that it is error-free.
- Save your script.
- Play the scene.
- As we play the scene, the **Output window** should look like the following:


```
Output:

--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
local variable: 3
--- Debugging process stopped ---
```

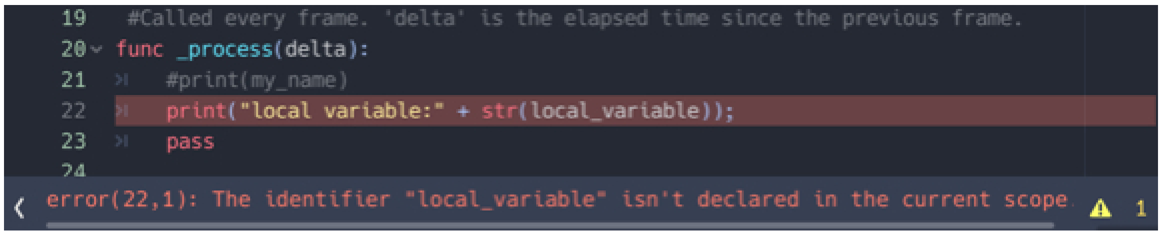
- We can see the message from the **_ready** function that we have just created.

Now, just to demonstrate the importance of variable scope, we will make an error on purpose; we will try to use the variable **my_variable** (which is a local variable) outside the method **_ready**, where it has initially been declared. As you may have guessed, this should trigger an error.

- Please type or copy and paste the following code inside the method **_process**.

print(“local variable: “ + str(localVariable));

- Save your script (*CTRL* + *S*).
- Switch back to Godot. Before you can try to play the scene, you will notice an error in the **Output window** as follows.


```
#Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
	#print(my_name)
	print("local variable:" + str(local_variable));
	pass

error(22,1): The identifier "local_variable" isn't declared in the current scope.
```

By displaying this message, Godot is telling us that it does not recognize the variable **local_variable** in the context where it is being used. This is because it was declared locally in the **_ready** function and then used outside this function (i.e., outside its original scope or context). So if you see similar messages as you code your game, always check the scope of your variable. This should save you some headaches. :-)

We can now comment this code, as it was only created to illustrate possible errors linked to the scope of variables.

- Please comment or delete the line that we have just created in the **_process** method.

```
#print("local variable:" + str(local_variable))
```

In GDScript you can comment a line of code by adding **#** to the start of the line. This means that the code will be part of the script, but it will not be executed.

Creating a simple counter

In this section, we will create a simple counter to practice declaring and assigning values to variables. This timer will just count from **0** onwards and use a variable for which the value will be increased over time (i.e., every frame).

- Please add the following line at the top of our class (i.e., **MyScript**), just before the function **_ready** to declare our counter (just after the declaration for **my_name**).

```
var counter:int
```

- Then, initialize the variable **counter** to **zero** by adding the following code within the **_ready** function.

```
counter = 0
```

Note that the variable **counter** is initialized in the **_ready** function; as a result, the initialization for this variable is done only once, at the start of the scene. If we were to add this code to the **_process** method instead, the variable would be initialized constantly (i.e., every frame), so this would not be suitable.

- Finally, please add the following code at the end of the **_process** method so that we add one to the variable **counter** every frame (i.e., every time the screen is refreshed) and display its value.

```
counter += 1
print (“counter=” + str(counter))
```

- After you have made these modifications, the code should look as follows:

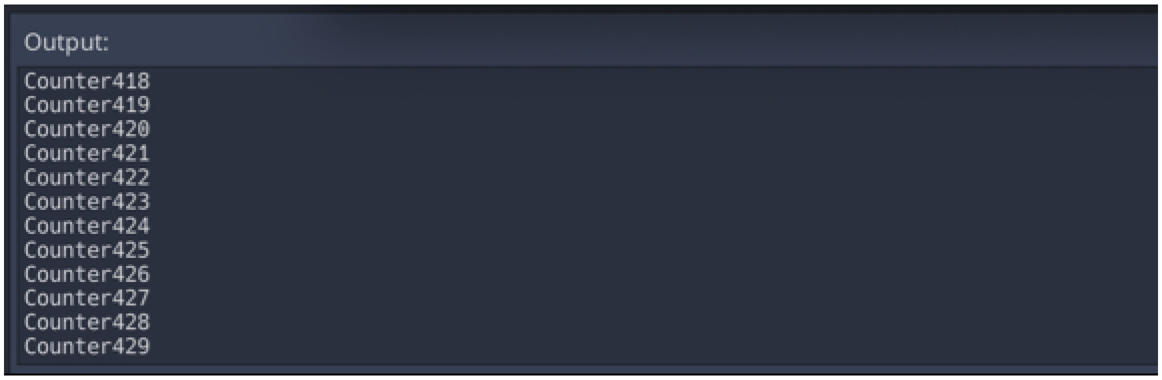
```
extends Node
var number:int
var my_name:String
var counter:int;
# Declare member variables here. Examples:
# var a = 2
# var b = "text"moo
```

```
# Called when the node enters the scene tree for the first time.
    func _ready():
    # number = 12
    # my_name = "Patrick"
    # print ("Hello " + my_name + " , your number is "+str(number))
    counter = 0;
    var local_variable:int = 3;
    print("local variable: " + str(local_variable))
```

```
#called every frame. 'delta' is the elapsed time since the previous frame.
    func _process(delta):
    counter += 1;
    print("Counter"+str(counter))
    pass
```

- Play the scene (**F6 FOR WINDOWS OR CMD + R FOR MACOS**).
- Look at the **Output window**: it should not display any error (i.e., provided that you have commented or deleted the code that we created to generate an error on purpose).
- Play the scene (**F6 FOR WINDOWS OR CMD + R FOR MACOS**), look at the **Output window**, and you should see that the value of the counter is displayed

and that it is increasing over time, as per the next figure.



```
Counter418
Counter419
Counter420
Counter421
Counter422
Counter423
Counter424
Counter425
Counter426
Counter427
Counter428
Counter429
```

At this stage we know about local and member variables, so let's look into functions and create our very first function.

Creating your first function

So what is a function? A method or a function is usually employed to perform a task outside the main body of the game. I usually compare methods to a friend or a colleague to whom you gently ask to perform a task for you. In many cases you will call them and they will agree to perform the task. Sometimes they will need additional information to perform the task (e.g., a number to be able to call someone on your behalf); some other times, they will call you back to give you the information that they found, but in other cases, this may not be necessary, and they will perform the task without contacting you afterwards.

So there are essentially three types of methods:

- Methods that just perform actions with no parameters.
- Methods that perform actions with parameters.
- Methods that perform actions (with or without a parameter) and return a result.

Declaring a function

In GDScript a function declaration usually includes the type of the parameters passed to this function. You can also specify the type of data returned by the function, but this is optional.

The syntax to declare a method is as follows:

- The keyword **func**.
- The name of the method.
- Opening round brackets.
- The type of the parameter and their names.
- Closing round brackets.
- Colons.
- Any action (i.e., statement) performed by this method will be added after but indented from the function definition.

In the next sections, we will see examples of how functions can be declared.

Functions that don't return or take any parameter

In this case, the function is called with no parameters; it will then perform an action. This is the simplest form of method. The syntax sequence is as follows: the keyword **func**, followed by the **name of the function**, followed by **opening and closing round brackets**, followed by **colons**. Any action (i.e., statement) performed by this function will be added after the line after the colons and indented, as illustrated in the next code snippet.

```
func my_first_function():
print("Hello World")
```

When called, this method will print the message **"Hello World"** to the **Output window**.

At this stage we have just defined the function **my_first_function**; in other words, we have specified what the function should do when it has been called. So once the function has been defined, we can call it using the syntax: **name_of_the_-function()** for example, to call **my_first_function** from the function **_ready** we could write the following statement at the end of the **_ready** function:

```
my_first_function()
```

So that this message stands out in the **Output window**, we can comment all other **print** statements inside the **_ready** and **_process** methods so that the code of your script looks like this (the changes are highlighted in bold):

```
extends Node
var number:int
var my_name:String
var counter:int;
# Declare member variables here. Examples:
# var a = 2
# var b = "text"moo
```

```
# Called when the node enters the scene tree for the first time.
```

```
func _ready():
# number = 12
# my_name = "Patrick"
# print ("Hello " + my_name + " , your number is "+str(number))
counter = 0;
var local_variable:int = 3;
#print("local variable: " + str(local_variable))
my_first_function()
```

```
#called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
counter += 1;
#print("Counter"+str(counter))
func my_first_function():
print("Hello World")
```

Note that the location of the function in the script (i.e., at the end or the start) does not matter, as long as it is declared within the class (**MyScript**) and outside any other function: so you need to declare your function outside of any other methods; we could have easily written this function at the start or middle of the class, resulting in no errors.

Now that you have written your first method, please do the following:

- Check that your code is written properly (i.e., error-free).
- Save your code (**ALT + CTRL + *S***).
- Check that there are no errors in the **Output window**.
- Play the scene and check that the message **"Hello World"** appears in the **Output** window.

Defining a function that takes parameters

So far, we looked at functions that would not take or return any parameters. For now, we will create a function that still doesn't return any data, but that takes one or several parameters to perform calculations.

So to borrow the previous example, you call someone, give them some information, and ask them to perform an action based on your instructions. To illustrate this concept, let's create a new method that will display a message based on a parameter passed as an argument.

- Please type the following code at the end of the class.

```
func my_second_function(name:String):
print("Hello, your name is " + name)
```

- In the previous code, we have created a function called **my_second_function**. It takes a parameter called **name** of type **String** (i.e., text). So when we call this method and include a String variable within the brackets, this variable will be referred to as **name** within this method.
- Let me illustrate with the following code.

```
my_second_function("Patrick");
```

If we were to type the previous code inside the ***_ready*** function, the method **my_second_function** would set the variable **name**, that is used in the method **my_second_function**, with the string **Patrick**, and then display the message "**Hello, your name is Patrick"**. The variable **name** is a local variable to the function **my_second_function**.

If you have not already done so, please add the following code to the **_ready** function. You can replace the word **Patrick** with your own name.

```
my_second_function("Patrick");
```

- Save your code, and check the **Output window** for any error and play the scene.
- You should see, amongst other messages, the message **"Hello, your name is Patrick"**.
- You could now change the call to this method and pass your own name as a

parameter and see the result as you play the scene.

Note that we could have created a method that takes many other parameters. For example, we could have created a method that takes the first and last name as parameters, as follows.

```
func my_third_function (f_name:String, l_name:String):
print("Hello, your name is "+f_name+ " " + l_name)
```

Defining a method that takes parameters and returns information

So far, we know how to declare a function that takes parameters; however, we have not seen yet how we could define a function that also returns information.

This type of function will, in addition to possibly taking parameters and processing this information, return information to where it was called.

In the following example, we will create a method that does all three: it will be called; it will then take the **year of birth** as a parameter, and then calculate and return the corresponding **age** (based on the current year).

- Please add the following code at the end of the class.

```
func calculate_age(YOB:int):
var age:int = 2021-YOB
return (age)
```

In the previous code:

- The function called **calculate_age** is declared.
- The function called **calculate_age** takes a parameter called **YOB** (short for Year Of Birth).
- The function **calculate_age** then subtracts **YOB** from the current year and returns the result.

Please add the following code to the method **_ready** (please make sure that you code is indented at the same level as the other statements within that function).

```
var my_age:int = calculate_age(1998)
print ("Your age is " + str(my_age))
```

In the previous code:

- The function **calculate_age** is called once; it returns the calculated age, and this (returned) value is saved in the variable called **my_age**.
- This variable **my_age** is then printed in the **Output window**.

Now that you have written the code for your new function, please do the following:

- Save your code.
- Check that there are no errors in the **Output window** and play the scene.
- The console should display, amongst other things, the message **"Your age is 23"**.

As you can see, there are different types of methods that you can create, depending on your needs. They may or may not take parameters, and they may or may not return values.

Note that we could have specified the type of the data to be returned by the function by modifying its definition as follows (new cold in bold):

func calculate_age(YOB:int)->int:

var age:int = 2021-YOB

return (age)

We could also have specified a default value for the parameter DOB as follows (new cold in bold):

func calculate_age(YOB:int = 2000)->int:

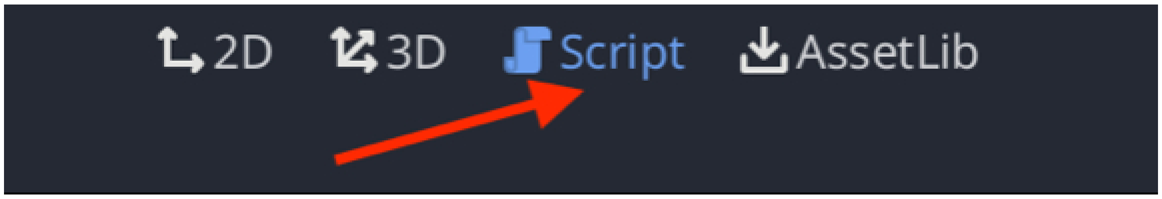
var age:int = 2021-YOB

return (age)

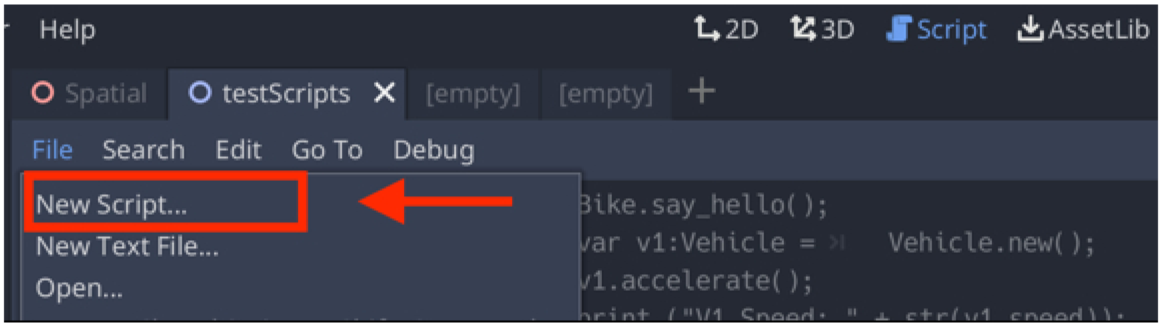
CREATING YOUR OWN CLASS

To finish this chapter, it would be great to see how you could create and use your own class, and we will simply create and use a class for a bike.

- Please select the **Script** workspace by pressing the corresponding button at the top of your screen as per the next figure.

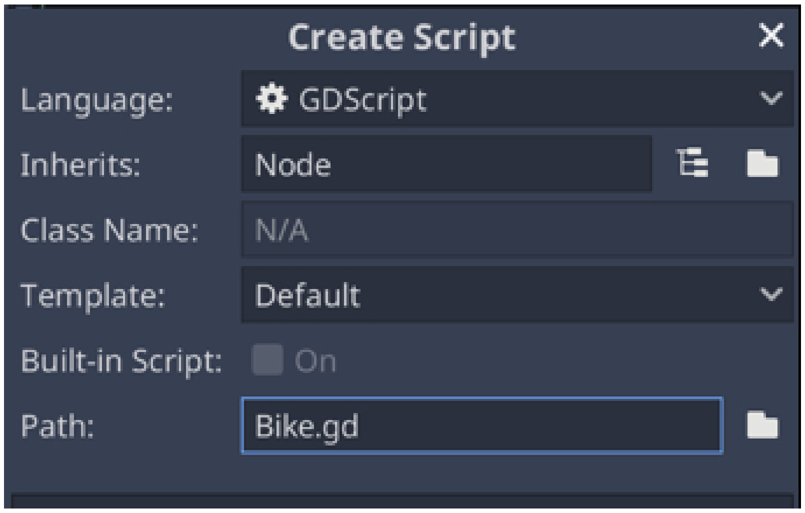


- Select: **File | New Script** from the left column.



```
Bike.say_hello();
var v1:Vehicle =     Vehicle.new();
v1.accelerate();
print ("V1 Speed: " + str(v1.speed));
```

- Specify a name for your script; by default, your script will inherit from the class **Node** (and you can leave this default setting) and it will be saved in the root folder (res), as illustrated in the next figure.



Create Script

Language: GDScript

Inherits: Node

Class Name: N/A

Template: Default

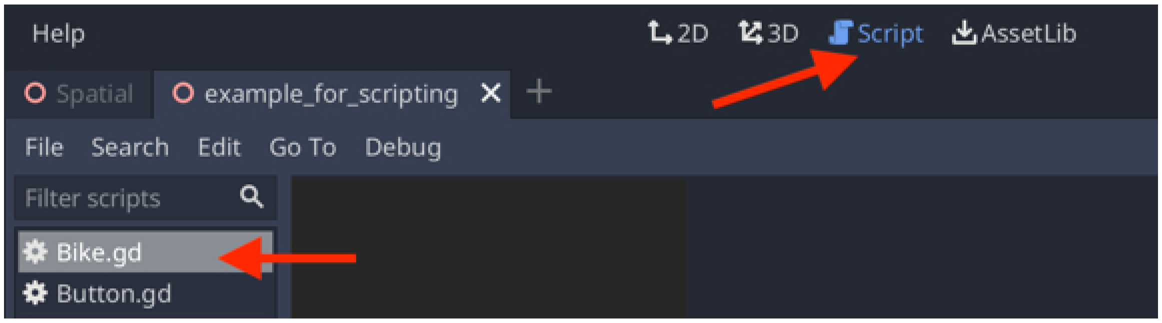
Built-in Script: On

Path: Bike.gd

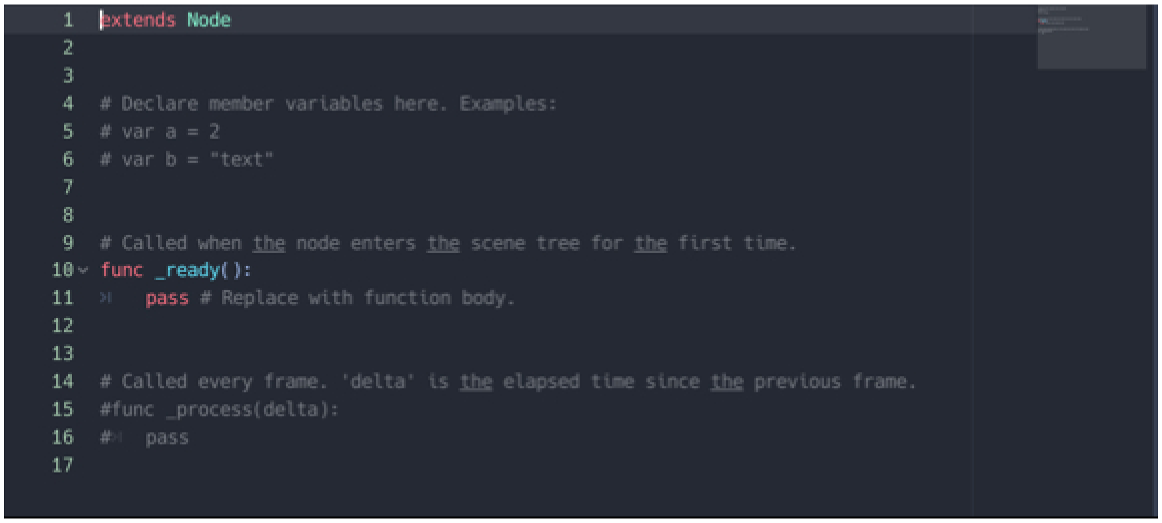
- Press the button labelled **Create**.

At this stage our script has been created, so let’s open it:

- Click on the button labelled **Script** in the top workspace.
- Then click on the name of your script in the left-hand side column.



- This should display the content of the script as follows.



```
extends Node


# Declare member variables here. Examples:
# var a = 2
# var b = "text"


# Called when the node enters the scene tree for the first time.
func _ready():
	pass # Replace with function body.


# Called every frame. 'delta' is the elapsed time since the previous frame.
#func _process(delta):
#	pass

```

When this is done, let's edit this script to add some features to our bike.

- We can delete the code "**extends Node"**. This is because our class will be used as a standalone class and not inherit from the **Node** class.
- Then we can delete the methods **_ready and _process**, as we will create our own methods for the class **Bike**.
- So, you can delete the entire code from the file so that the file is totally empty.

At this stage we have a blank canvas that we can use for our new class.

- Please add the following code at the start of the class.

class_name Bike

var speed:float

var color:String

var name:String

- In the previous code, we declare the name of the class as well as three mem- mem- variables of type **String** and **float**. These will be used to identify the name, speed, and number of wheels for new bikes.

We will now define a constructor for our class, to define the feature of each new bike created.

- Please add the following code within the class, after the previous code:

```
func _init(new_name:="A New Bike", new_color:="Blue",new_speed:=0):
color = new_color
speed = new_speed
name = new_name
```

- In the previous code, we create a constructor that takes three parameters which are used to initialize the class's member variables **color**, **speed** and **name**. Default values are also defined for these parameters (i.e., "**A New Bike**", "**Blue**" and **0**).

Next, we will define a few member functions for the class **Bike**; so please add the following function to the class **Bike**:

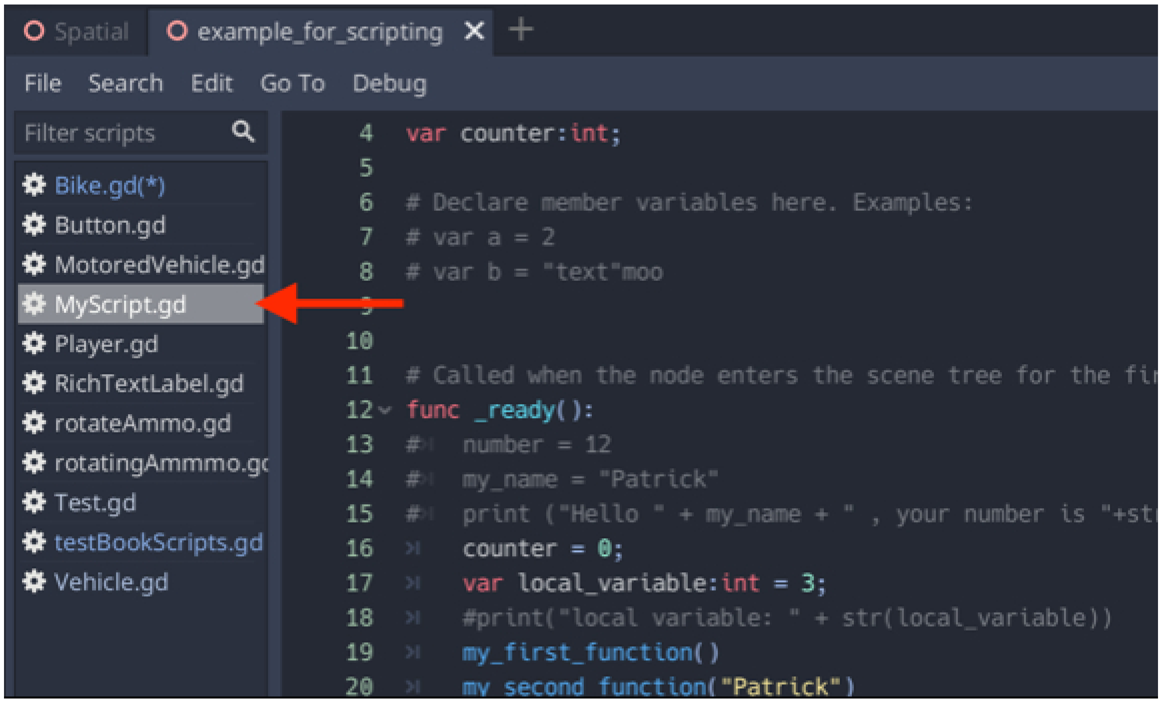
```
func display_name():
print("Name: "+name)
func display_color():
print("Color is: " + color)
func accelerate():
speed+=1
print("New speed: "+str(speed))
```

- In the previous code, we define three functions named **display_name**, **display_ color** and **accelerate**.
- The first function displays the name of the bike.
- The second function displays the color of the bike.
- The third function increases the bike's speed by one and displays it.

You can now save this file (From the left-hand menu: select **File | Save As**) as **Bike.gd**. By default, the root folder is used to save your file; however, you can create and/or select another folder to do so.

Once the class **Bike** has been saved, it is now time to use it by creating instances of this class. To do so, we will use the file **MyScript** that we have already created and add code, in the **_ready** function, that will instantiate new bikes.

- Please open the script **MyScript** by selecting it on the left side of the screen.



```
var counter:int;

# Declare member variables here. Examples:
# var a = 2
# var b = "text"moo


# Called when the node enters the scene tree for the fi
func _ready():
#	number = 12
#	my_name = "Patrick"
#	print ("Hello " + my_name + " , your number is "+st
	counter = 0;
	var local_variable:int = 3;
	#print("local variable: " + str(local_variable))
	my_first_function()
	my_second_function("Patrick")
```

- Once the file is open, please comment all the code in the **_ready** function: select the code within the function and press **CTRL + K**.
- Type the following code, in the **_ready** function, after the code that has been commented (ensuring that the code is indented compared to the first line of the function).

```
var my_bike:Bike = Bike.new();
my_bike.display_color()
my_bike.display_name()
my_bike.accelerate()
```

In the previous code:

- We declare a variable called **my_bike**.
- You may notice that we use the constructor without passing any parameter, so, as a result the default values for this bike's member variables will be

used: A **blue** color, a speed of **0**, and the name "**A New Bike**".

- We then call the member functions **display_color**, **display_name**, and **accelerate**.

We can now test our code: please save your code and play the scene, and you should see the following messages in the **Output** window.

```
Color is: Blue
Name: A New Bike
New speed: 1
```

Now that we know that the code works, we can add more code to test the constructors.

- Please stop the scene (**CTRL + period**).
- Add this code to the **_ready** function.

```
var my_bike2:Bike = Bike.new("My Bike","Red",10)
my_bike2.display_color()
my_bike2.display_name()
my_bike2.accelerate()
```

In the previous code:

- We create a new **Bike**.
- This time we pass parameters to the constructor so that our new bike's member variables **name**, **color** and **speed** are respectively "**My Bike**", "**Red**", and **10**.
- We then display the value of the variables **color** and **name**.
- Finally, we call the function **accelerate** which increases the speed by **1**.

We can now test our code: please save your code and play the scene, and you should see the following messages in the **Output** window.

```
Color is: Red
Name: My Bike
New speed: 11
```

BEST PRACTICES

To ensure that your code is easy to understand and that it does not generate countless headaches when trying to modify it, there are a few good practices that you can start applying as you begin with coding; these should save you some time along the line.

Variable naming

- Use meaningful names that you can understand, especially after leaving your code for two weeks.

```
var my_name:String = "Patrick";//GOOD
var b : String = "Patrick";//NOT SO GOOD
```

- Use Pascal and **Snake** casing consistently.

```
bool test_if_the_name_is_correct;// GOOD
bool testifthenameiscorrect; // NOT SO GOOD
```

Methods

- Check that all opening brackets have a corresponding closing bracket.
- Indent your code when it is part of a function or a block of instructions.
- Comment your code as much as possible to explain how it works.
- Use the **_ready** function if something just needs to be done once at the beginning of the game.
- If something needs to be done repeatedly, then the method **_process** might be a better option.

Level roundup

Summary

In this chapter, we became familiar with different programming concepts. We also looked into classes, constructors, and member variables. Finally, we created our first class and experimented with instantiating instances and displaying their properties. In the next chapter, we will harness these skills to bring interactivity to a 3D environment.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available on the next page.

1. Each class has a default constructor.
2. A constructor can include several parameters.
3. A member variable can be accessed from anywhere in your class.
4. When a new instance of an object is created, the corresponding constructor is called.
5. A new GDScript file created with Godot will inherit from the **Node** class by default.
6. The **Script** workspace can be used to open, create and modify scripts.
7. The keyword **class_name** makes it possible to define the name of a class.
8. In **Snake casing** the first character of each word that makes up the name of a variable is capitalized except for the first word.
9. In **Pascal casing** the first character of each word that makes up the name of a variable is capitalized.
10. Functions are declared using the keyword **func**.

Answers to the Quiz

1. TRUE.
2. TRUE
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. TRUE.
8. FALSE
9. TRUE
10. TRUE.

Checklist

If you can do the following, then you are ready to go to the ne

- Create a new GDScript script.
- Attach a script to a node.
- Create a class.
- Create member variables.
- Create member functions.
- Call a constructor.
- Know how to comment your code in a script.
- Answer at least 7 out of 10 of the questions correctly in the quiz.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let's put these to the test.

- Modify your code to create two additional member functions.
- Instantiate objects and call these methods from these objects.

Chapter 3: Adding Interaction with GDScript

In this section we will discover how scripting can be used to add interaction to the game and to provide more control over the game mechanics.

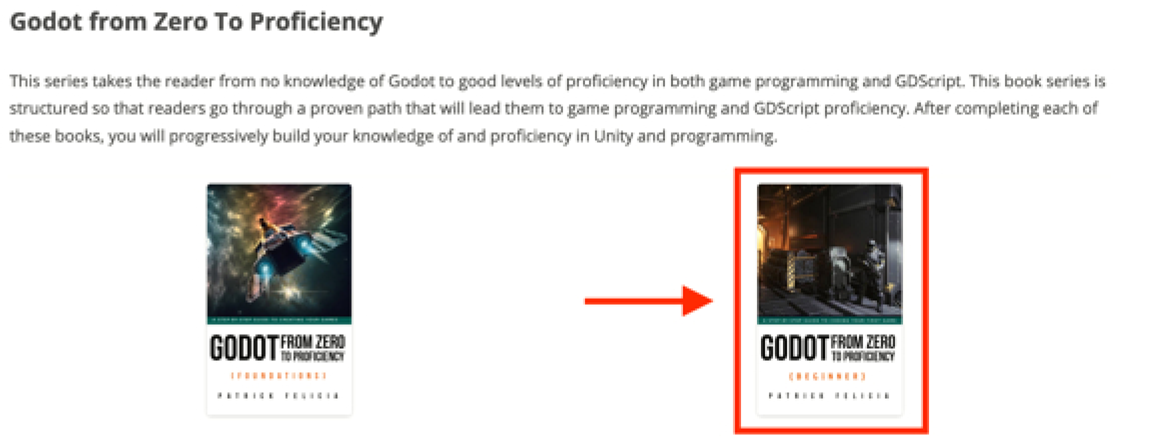
After completing this chapter, you will be able to:

- Detect collisions between the player and other objects.
- Collect objects upon collision.
- Implement a scoring system to keep track of the number of objects collected.
- Change the current level and load a new scene from your code based on the score.

RESOURCES NECESSARY FOR THIS CHAPTER

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



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]
PATRICK FELICIA
GODOT FROM ZERO TO PROFICIENCY
[BEGINNER]
PATRICK FELICIA

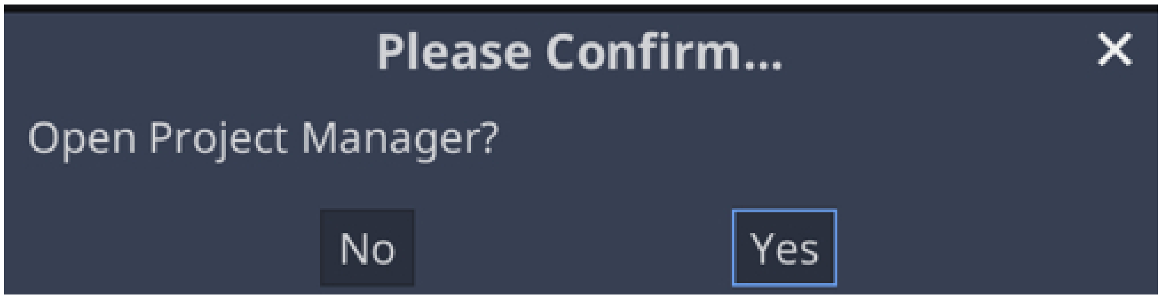
CREATING A SIMPLE SCRIPT TO COLLECT OBJECTS

At this stage, we are becoming familiar with Godot and creating scripts; to use our skills and to be able to interact with the environment we have created, we will learn how to collect objects upon collision, using some of the built-in functions available in Godot, as well as new variables. The workflow will be as follows. We will:

- Create a simple environment.
- Add boxes.
- Add a tag to these boxes (we will see what this means in a few seconds).
- Use a built-in function that is called when the **First-Person Controller** collides with another object.
- Modify this function so that we detect the name or the label of the object involved in the collision and destroy it accordingly (i.e., collect it).
- Initialize and update the score accordingly.
- Load the next scene based on the score.

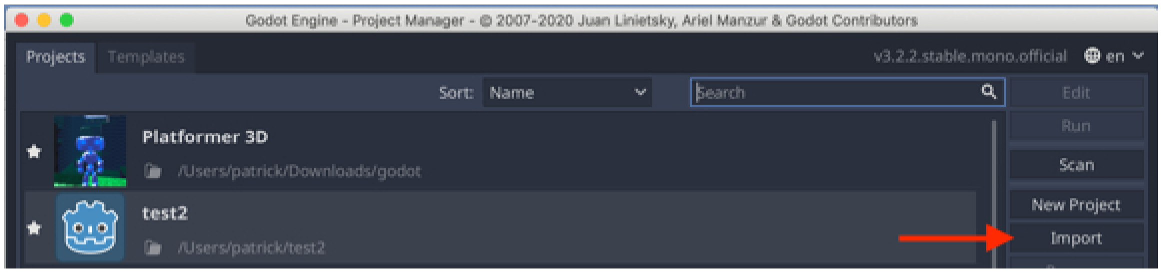
So, let's get started:

- Please open Godot if it is not already open.
- If Godot is already open, then go to the **Project List** by selecting **Project | Quit to Project List** and by pressing "Yes" in the new window to confirm your choice.

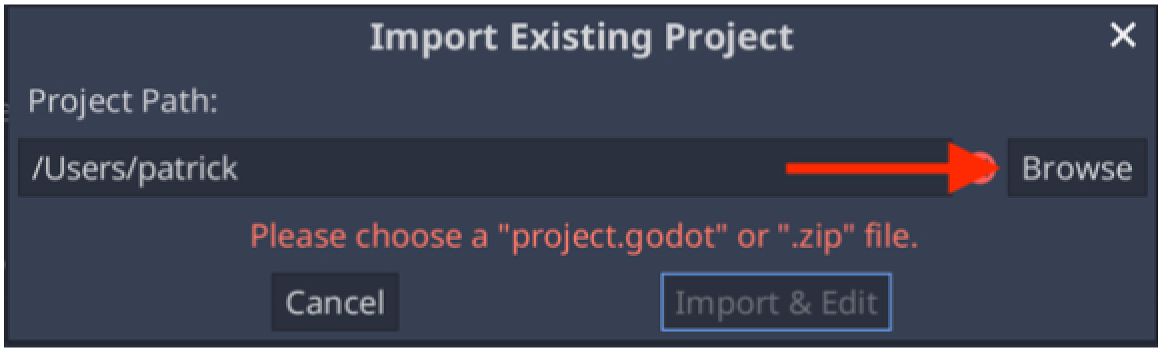


Open Project Manager?

- In the Project list, please click on the button labelled "**Import**".

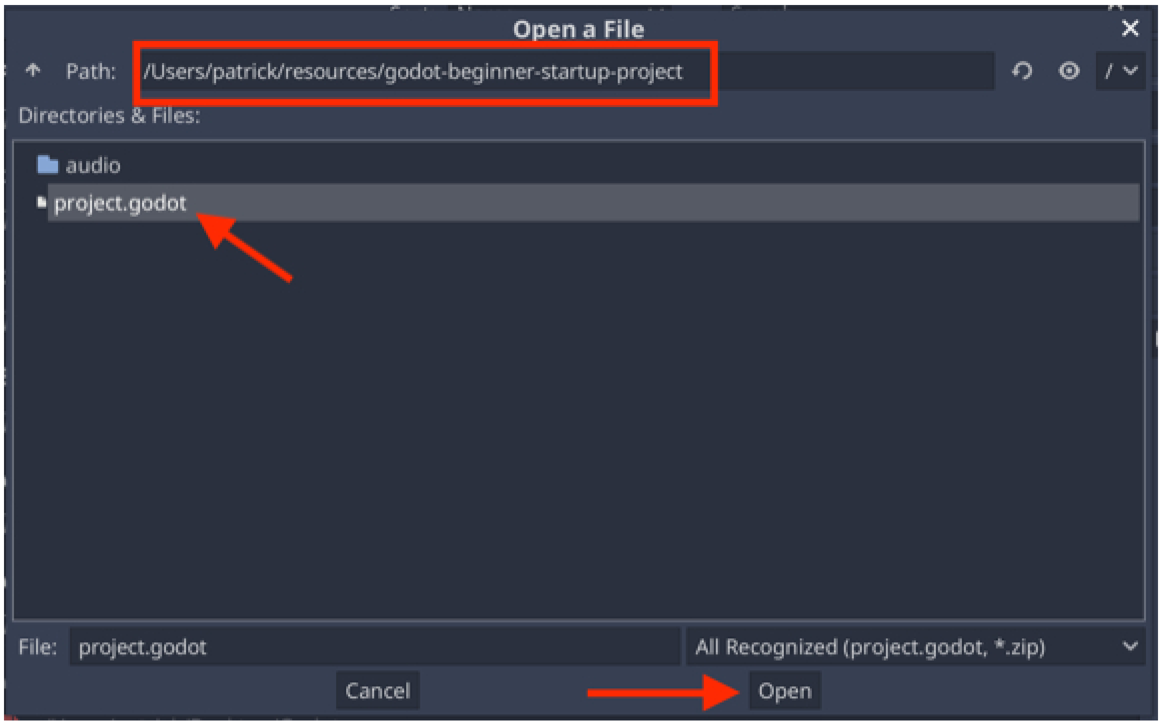

Godot Engine - Project Manager - © 2007-2020 Juan Linietsky, Ariel Manzur & Godot Contributors
Projects
Templates
v3.2.2.stable.mono.official
en
Sort:
Name
Search
Edit
Run
Scan
New Project
Import
Platformer 3D
/Users/patrick/Downloads/godot
test2
/Users/patrick/test2

- In the new window, click on the button labelled "**Browse**" to select the location where you have downloaded and unzipped the start-up project.

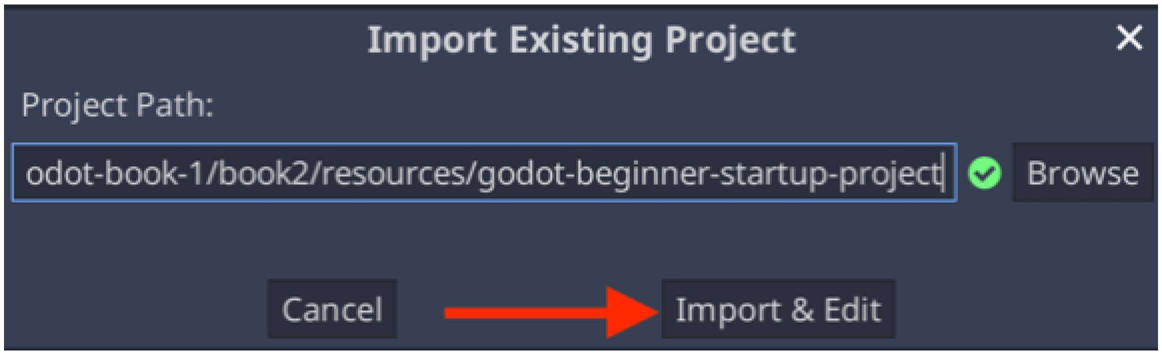


Please choose a "project.godot" or ".zip" file.

- In the new window, please navigate to where the resource pack was downloaded and unzipped, then locate the folder labeled "**godot_beginner_startup_project**", select the file **project.godot** that is within, and click on **Open**.

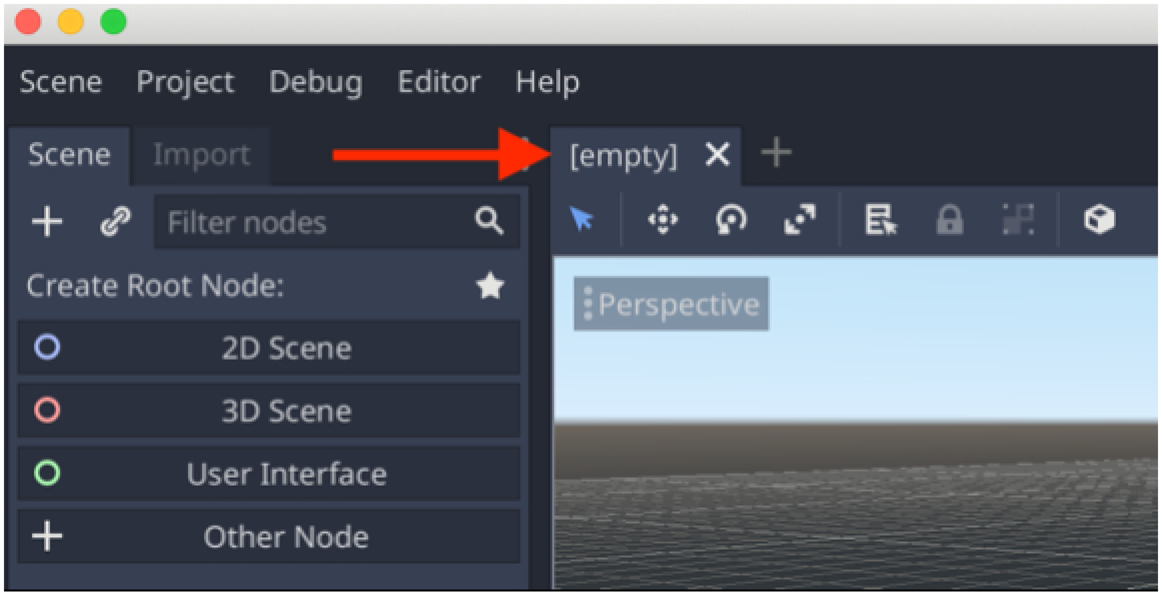

Open a File
Path:
/Users/patrick/resources/godot-beginner-startup-project
Directories & Files:
audio
project.godot
File:
project.godot
All Recognized (project.godot, *.zip)
Cancel
Open

- In the new window, click on the button labelled "**Import and Edit**".

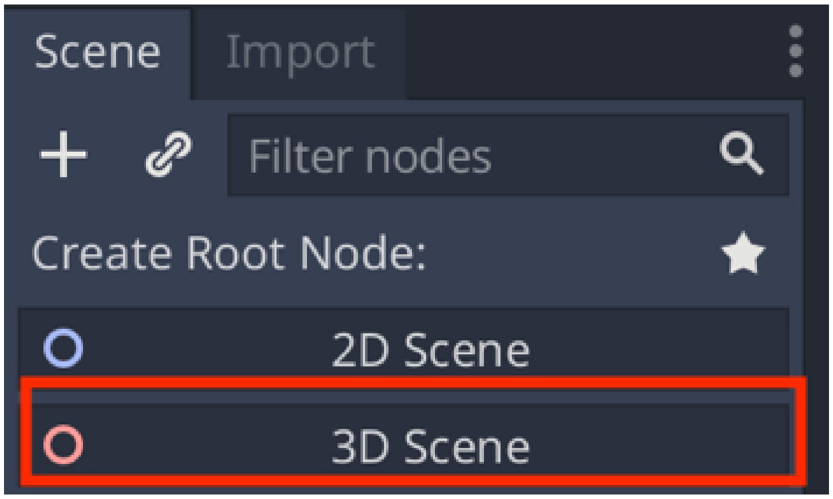


Once the project is open we can then create our new scene and first script:

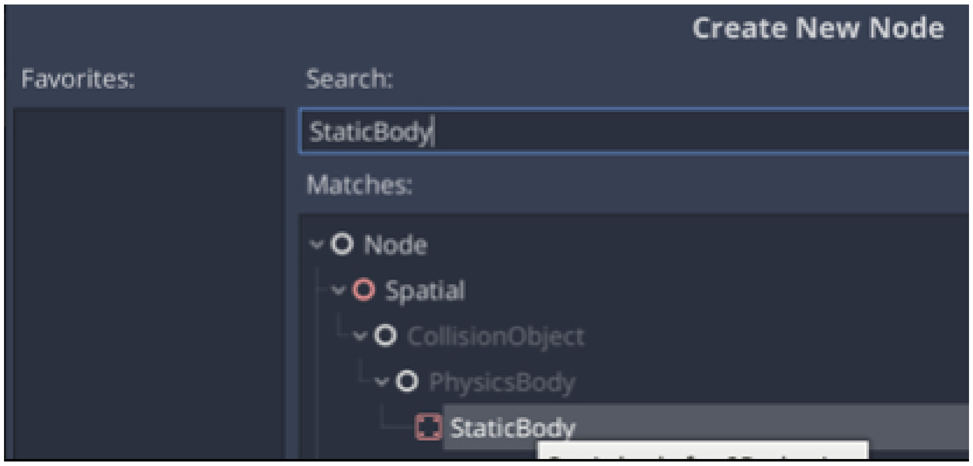
- This should also create an empty scene by default.



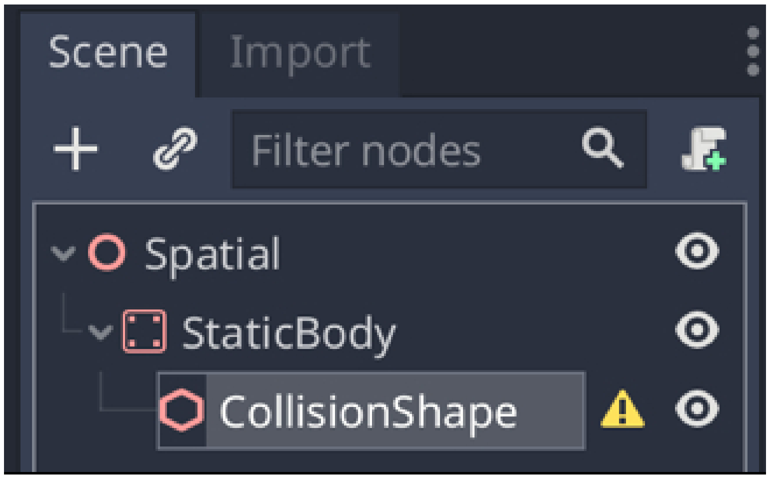
- If not, you can create a new scene (**Scene | New Scene**).
- Select "**3D Scene**" from the left menu, as illustrated in the next figure.



- Save the current scene (**Scene | Save Scene As**) with a name of your choice (e.g., **scene1**).
- Please create a new **StaticBody** node: right-click on the **Spatial Node** already present in the scene, select the option "**Add Child Node**", type **StaticBody** in the **search** field, select the node **StaticBody** and click **"Create"**.



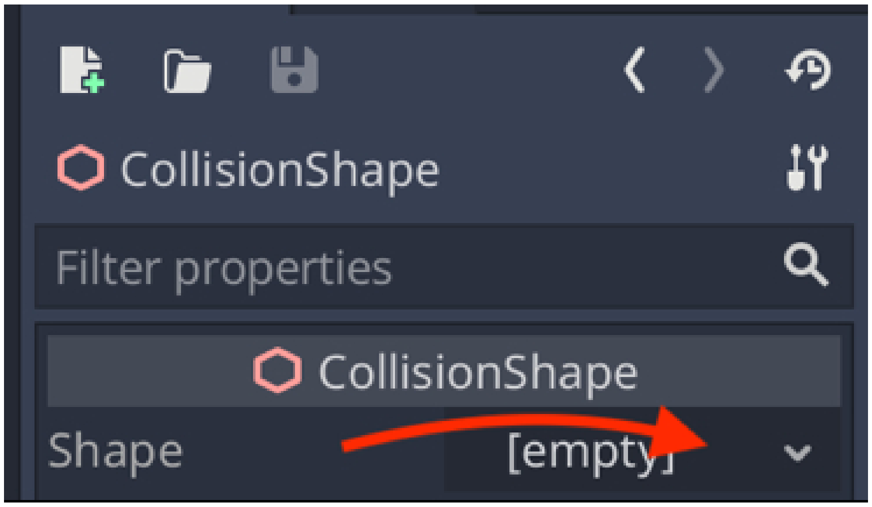
- This will create a new node called **StaticBody**.
- Please select this node in the **Scene** tree, and add a new child node to it of type **CollisionShape** (i.e., right-click on the node **StaticBody** and select **Add Child Node** from the contextual menu).
- At this stage, you should see three nodes in the **Scene** tree as illustrated in the next figure.



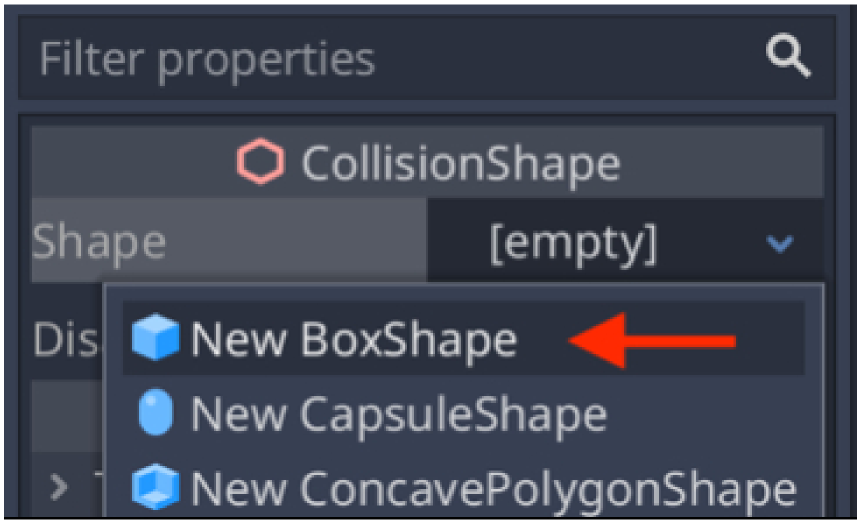
You may notice a warning sign for the **CollisionShape** node: this is because we need to specify a type of shape for the **CollisionShape** node. In other words, we need to specify the shape of the collider for our node.

- Please click once on the node called **CollisionShape**.
- In the **Inspector**, click on the downward-facing arrow that is located to the right of the field called **empty** in the section **Collision Shape | Shape**, as

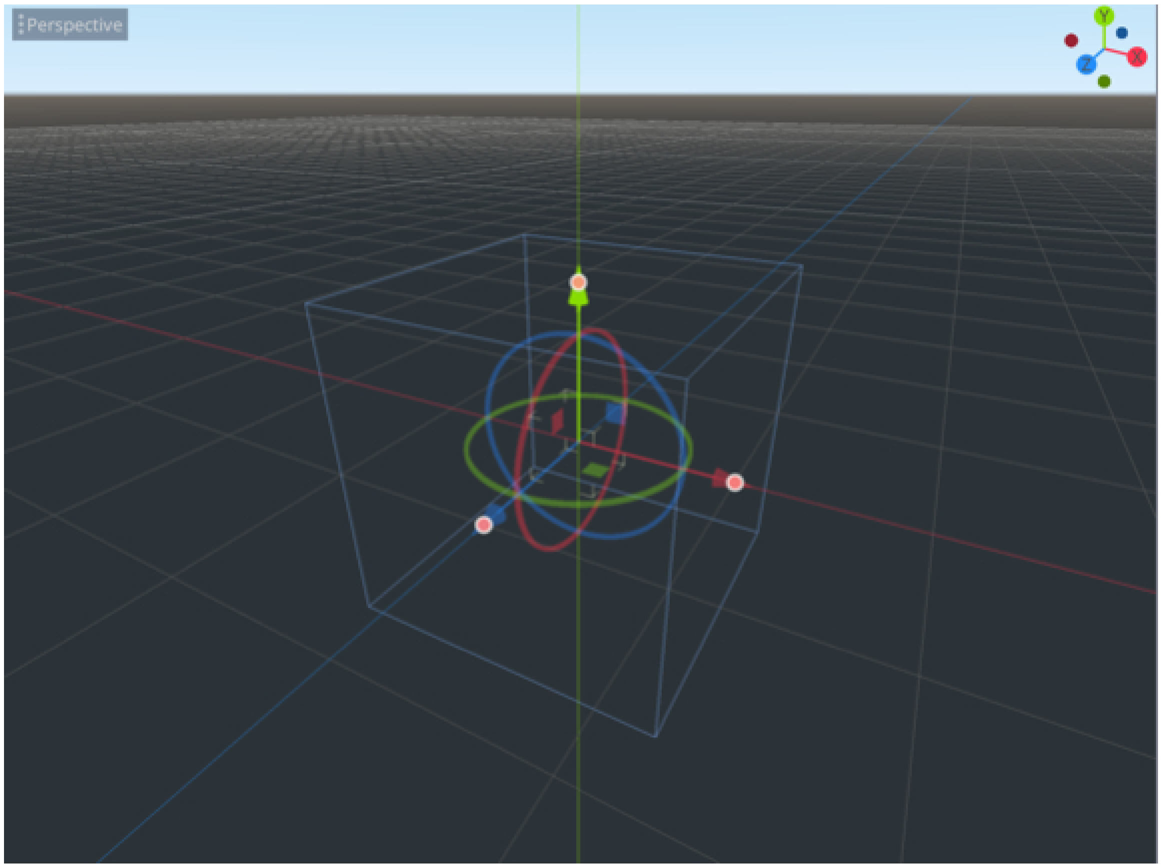
illustrated in the next figure.



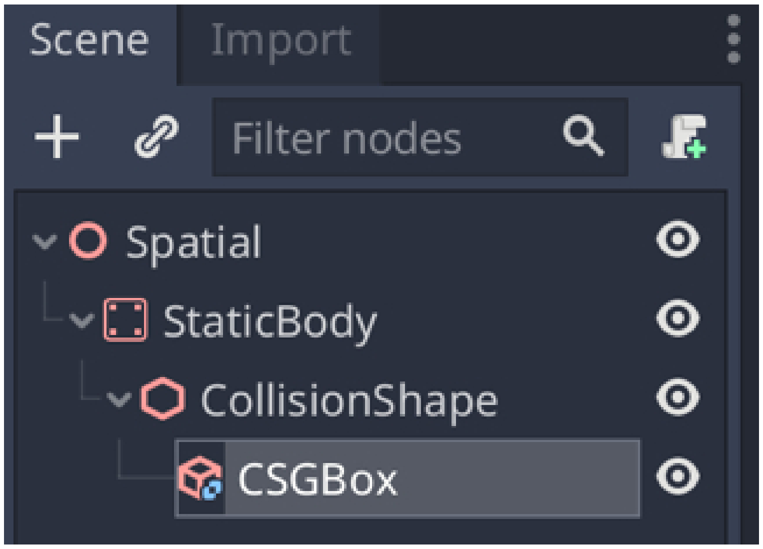
- Select the option **New BoxShape** from the contextual menu.

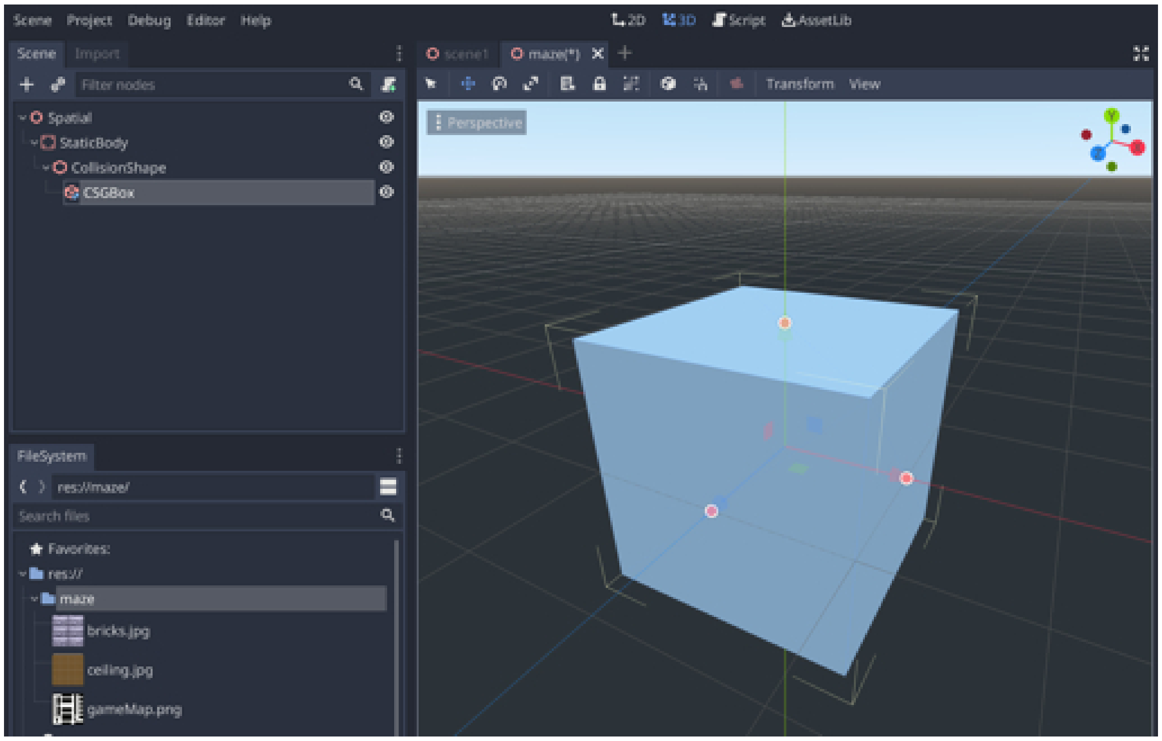


- This will create a blue box in the **Scene** view that represents the collision shape.



- Finally, please select the node called **CollisionShape** and add a new child of type **CSGBox** to it (i.e., right click on the node **CollisionShape**, then select **Add Child Node**, type **CSGBox** in the search field, select the type **CSGBox**, and click on **Create**).





So at this stage, we have a box that can be collided with:

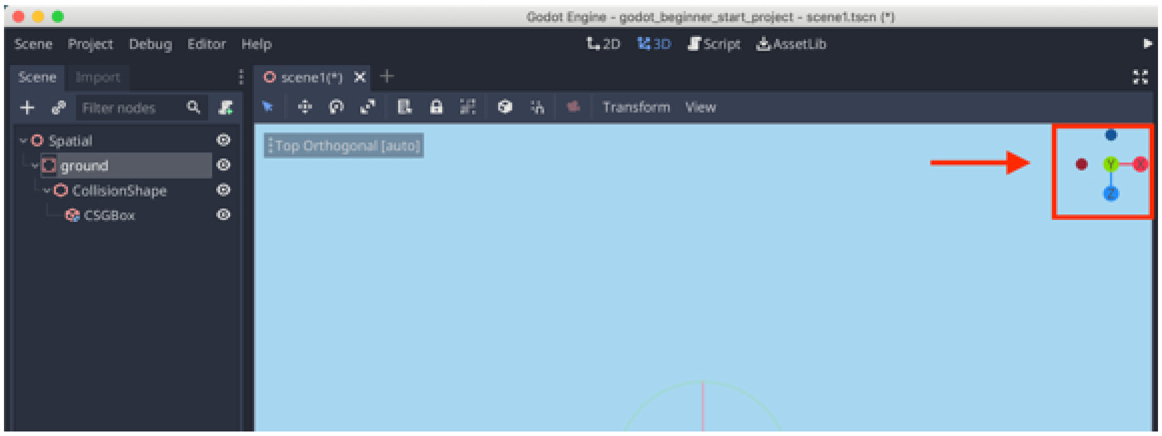
- The **StaticBody** node will ensure that it can be collided with.
- The **CollisionShape** node will determine the boundary that will be applied for collision detection.
- The **CSGBox** node will give an appearance to our box (i.e., shape and color).

Please note that because the **CollisionShape** and **CSGBox** nodes are children of the node called **StaticBody**, any transformation applied to the parent node (**StaticBody**) will also be applied to the child nodes.

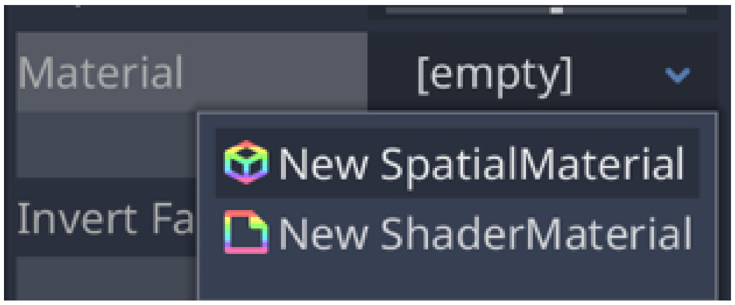
- Please rename the object **StaticBody** to **ground**.
- Using the **Inspector**, ensure that the position of this object is **(0, 0, 0)**.
- Using the **Inspector**, change the scale properties of this object to **(100, 1, 100)** so that it is scaled up along the **x-** and **z-axes**.

We will now apply a color to the **ground** object, and because the child node called **CSGBox** is responsible for the appearance of this ground, we will focus and that node for now:

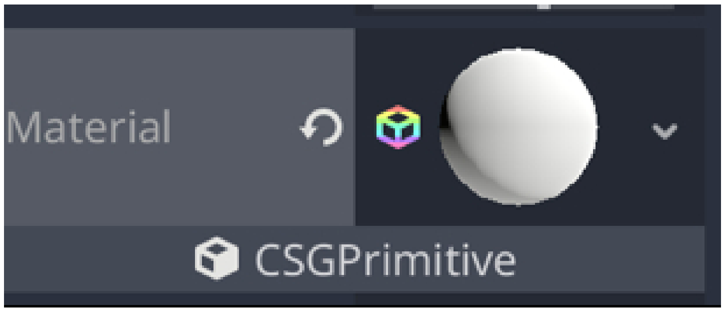
- To make it easier to see the changes, we can look at the scene from the y-axis. This can be achieved, as previously, by using the **Gizmo** located in the top-right corner of the **ViewPort**, and by clicking on its **y-axis**.



- Please click on the object **CSGBox** in the **Scene** tree.
- Using the **Inspector**, navigate to the section **CSGBox**.
- Click on the arrow to the right of the attribute called **Material**, and select the option **New Spatial Material** from the contextual menu.

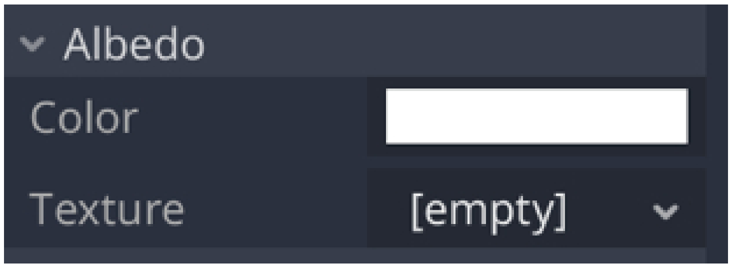


- Click on the white sphere displayed in that section.



- This will list several attributes for the **Material** component.

- Select the option called **Albedo**.



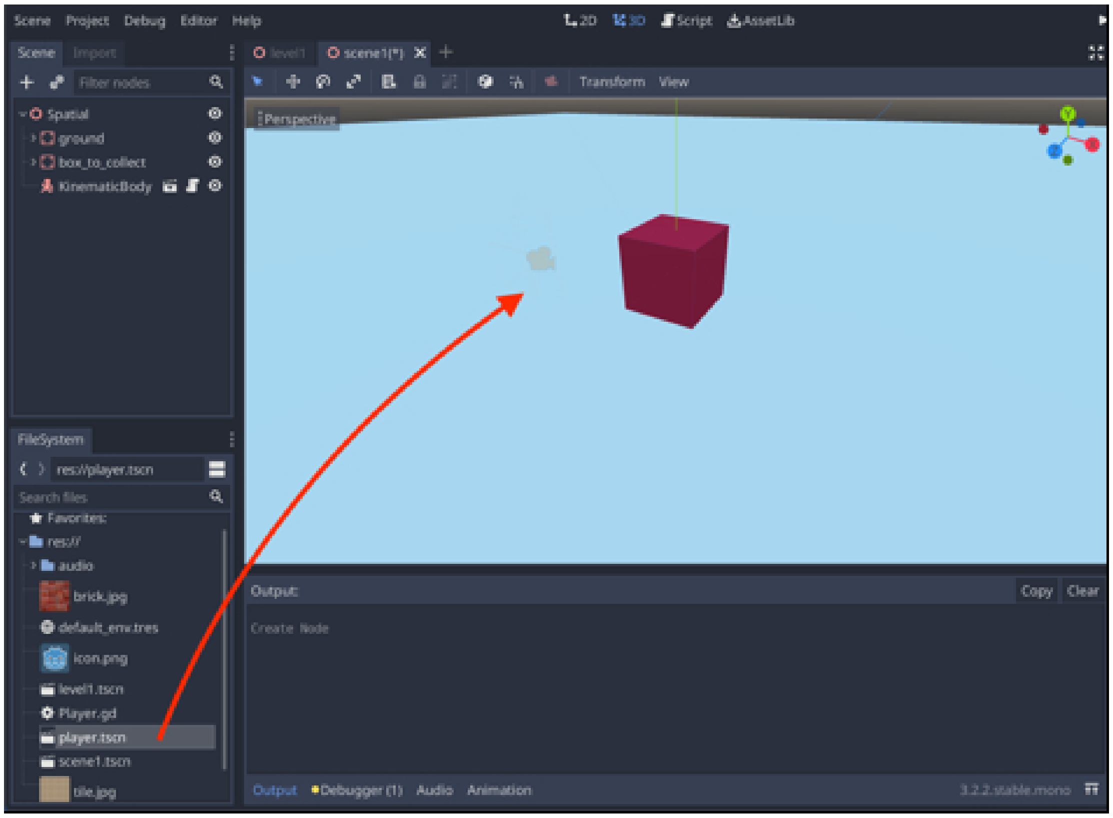
- At this stage, we want to specify a color for the ground.
- Please click on the white rectangle and select a color of your choice for the ground.

So at this stage you should have a ground node that includes both a collision shape and a **CSGBox** node.

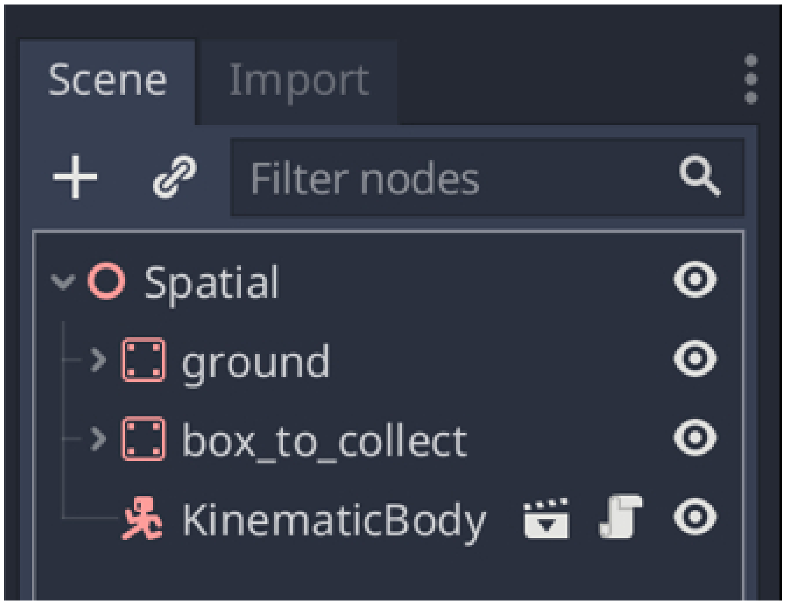
We will now create another cube that we will be collecting upon collision.

- Please duplicate the node **ground** and rename the duplicate **box_to_collect**.
- Change its scale property to **(1, 1, 1)** and its position to **(0, 2, 0)**.
- Chang its color to **red** by creating a new **Spatial Material** for this box.
- Make sure that it is slightly above the **ground** object.
- Add a **First-Person Controller** to the scene: Select the object called **Spatial** in the **Scene** tree (so that the new asset is added as a child of this node), drag and drop the asset **player.tscn** from the **FileSystem** window to the viewport, near the box, as per the next figure.

The asset **player.tscn** was created and added to the start-up project; it includes a First-Person Controller made that makes it possible to navigate through a scene using the arrow keys, the mouse, and the spacebar (to jump). This FPS controller includes a **KinematicBody** node, coupled to a camera, and a **Capsule Collider**.



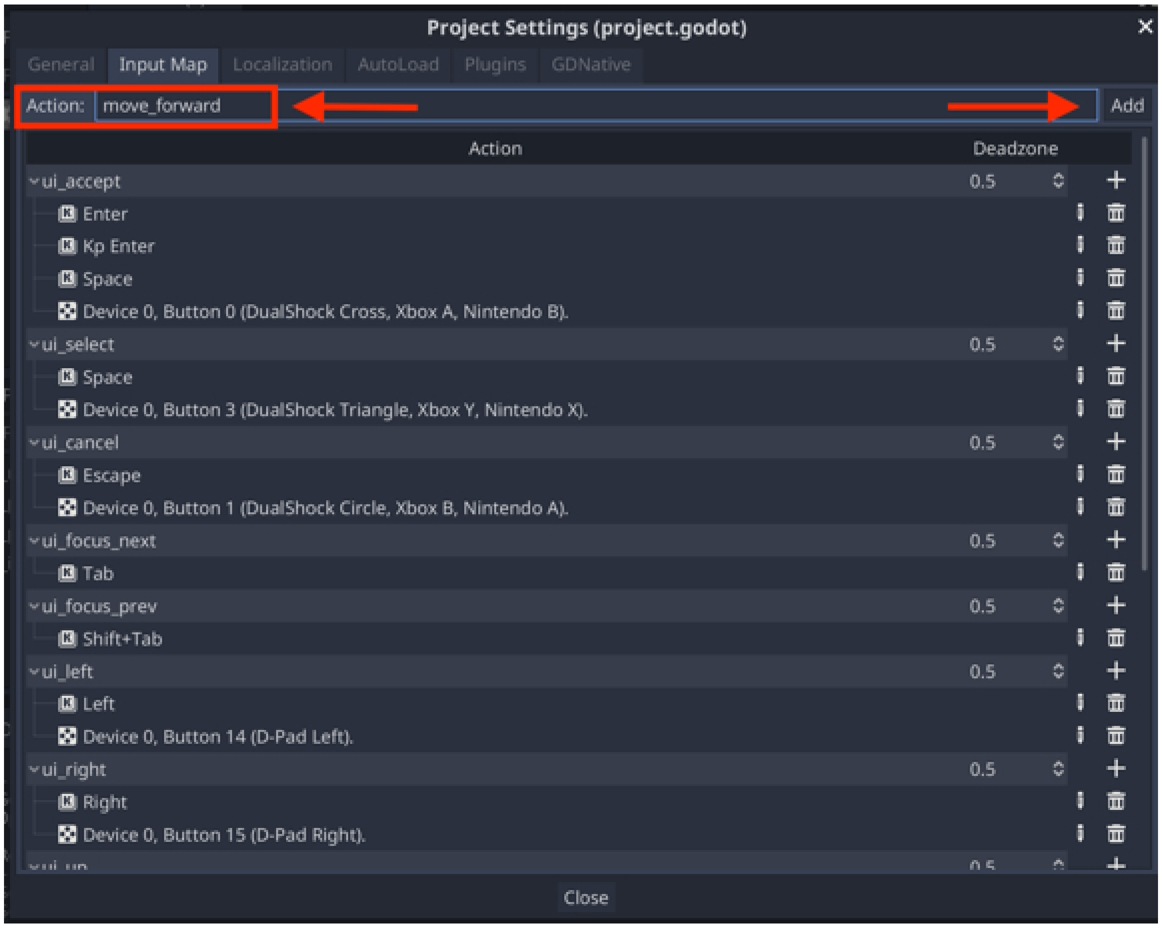
- This will create a node called **KinematicBody** in the **Scene** tree.



- Please rename this object **player** and change its position to **(-3, 1, 2)**.

Before we can play the scene, we just need to map the arrow keys to the movement of the First-Person Controller (i.e., the object **player**) so that you can use these keys to move around the level.

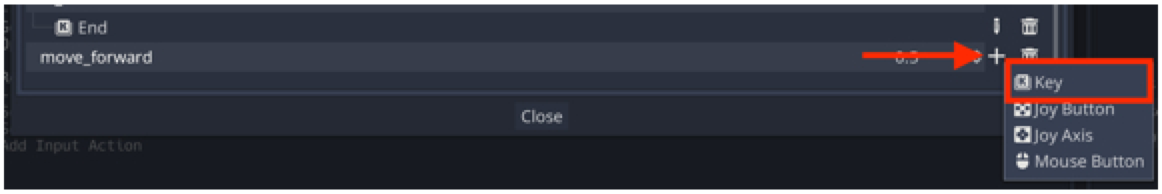
- From the top menu, please select: **Project | Project Settings | Input Map**.
- In the new window, enter the text "**move_forward**" in the top field (i.e., the field labelled "**Action**") and then press the button labelled **Add** (to the right of the field).



Action: move_forward

Action	Deadzone
ui_accept	0.5
Enter	
Kp Enter	
Space	
Device 0, Button 0 (DualShock Cross, Xbox A, Nintendo B).	
ui_select	0.5
Space	
Device 0, Button 3 (DualShock Triangle, Xbox Y, Nintendo X).	
ui_cancel	0.5
Escape	
Device 0, Button 1 (DualShock Circle, Xbox B, Nintendo A).	
ui_focus_next	0.5
Tab	
ui_focus_prev	0.5
Shift+Tab	
ui_left	0.5
Left	
Device 0, Button 14 (D-Pad Left).	
ui_right	0.5
Right	
Device 0, Button 15 (D-Pad Right).	
ui_up	0.5

- Once the key has been added, scroll down, click on the + button to the right of the key called **move_forward** and select "**Key**" from the contextual menu, as per the next figure.



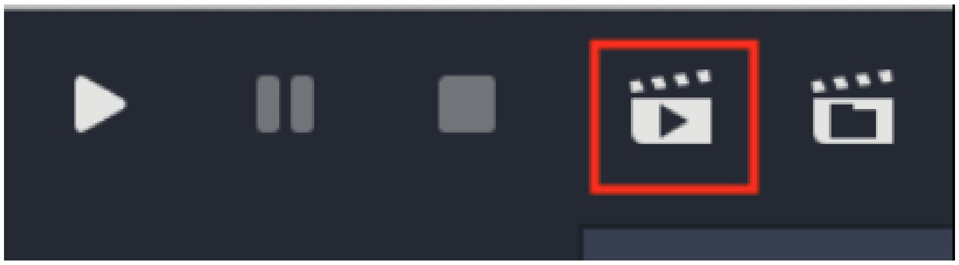
- Press the **Up Arrow** on your keyboard and then press **OK**. This means that the up arrow is now associated with an input that is referred to as **move_forward** in the script that is used to move the FPS controller.

Please repeat the previous steps to add the following settings:

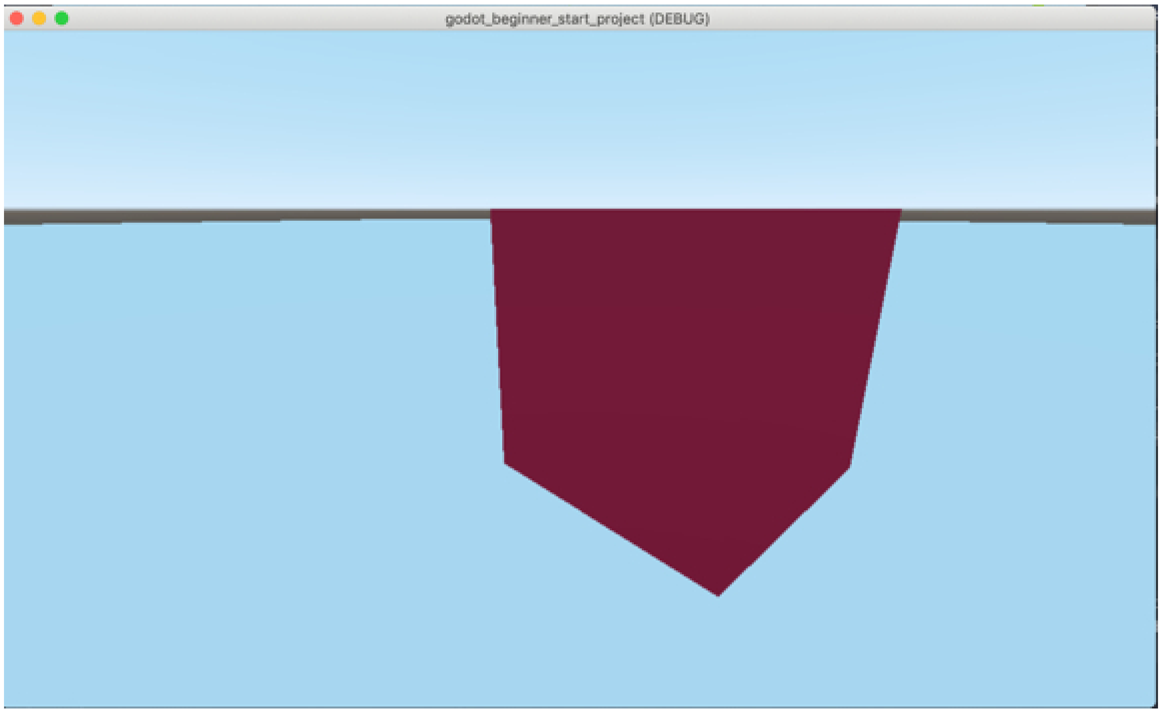
- **move_back** using the **Down Arrow**.
- **move_left** using the **Left Arrow**.
- **move_right** using the **Right Arrow**.
- **jump** using the **Spacebar**.

You can then close the **Project Settings** window.

At this stage, you can test the scene by pressing **F6** for Windows or **CMD + R** for MacOS (this is to play the scene; you can also press the corresponding button in the top-right corner - the fourth from the left.



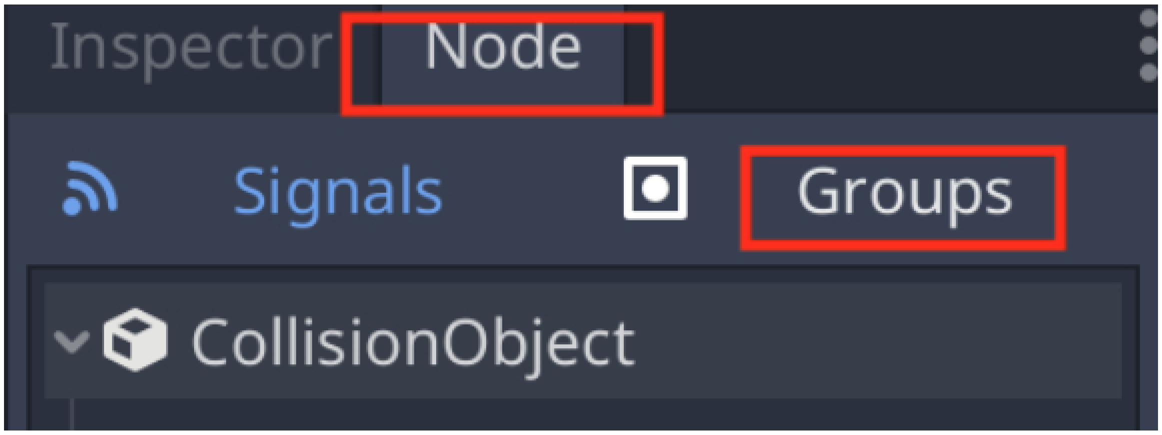
- Play the scene and you should be able to navigate the scene without going through the walls by pressing the arrow keys on your keyboard and the mouse to rotate the view.
- Please ensure that you can walk around the scene and collide with the box labelled **box_to_collect**.



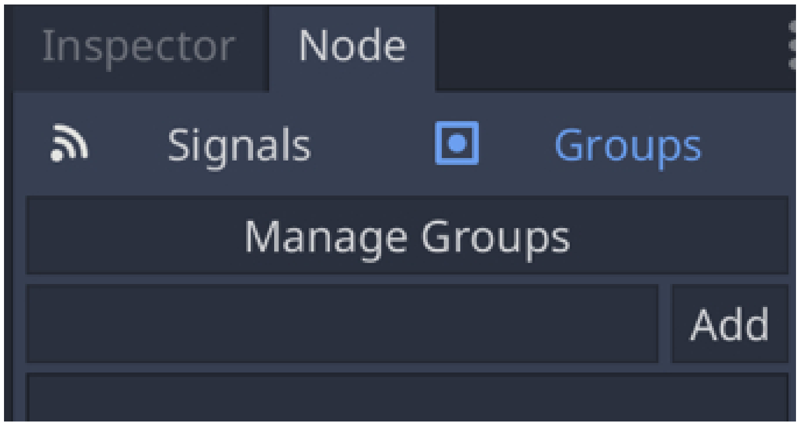
Now that you have created a new cube, we will create a new **group** or **tag** for this object. A **group** is comparable to a label that we can apply to one or several objects. This helps to group and categorize objects with similar behaviors. It is very helpful because it is possible to check the name of the tag applied to an object from a script.

So let's create a group (or tag) for our box:

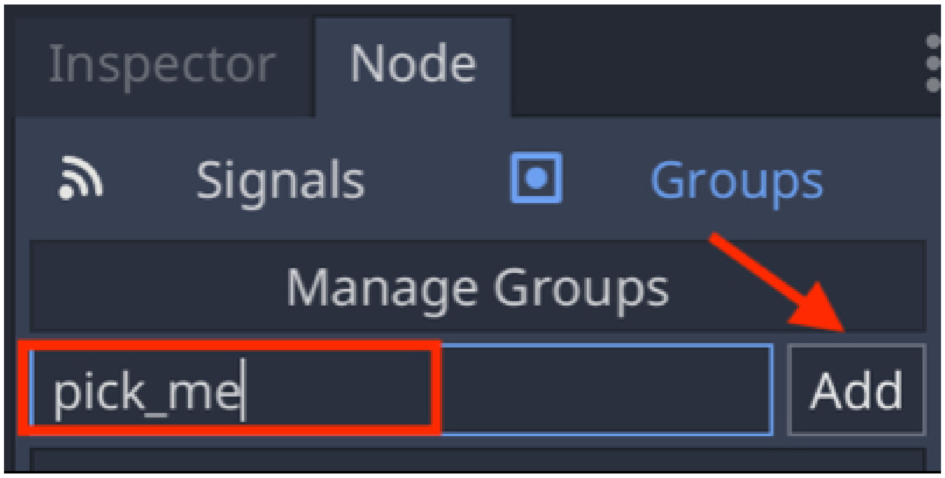
- Please stop to play the scene (**CTRL + Q** for Windows or **CMD + Q** for MacOS).
- Please select the object **box_to_collect** in the **Scene** tree window.
- In the **Inspector** window, you will notice a tab called **Node**, and within this tab a section called **Groups**; so please click on "**Groups**", as per the next figure.



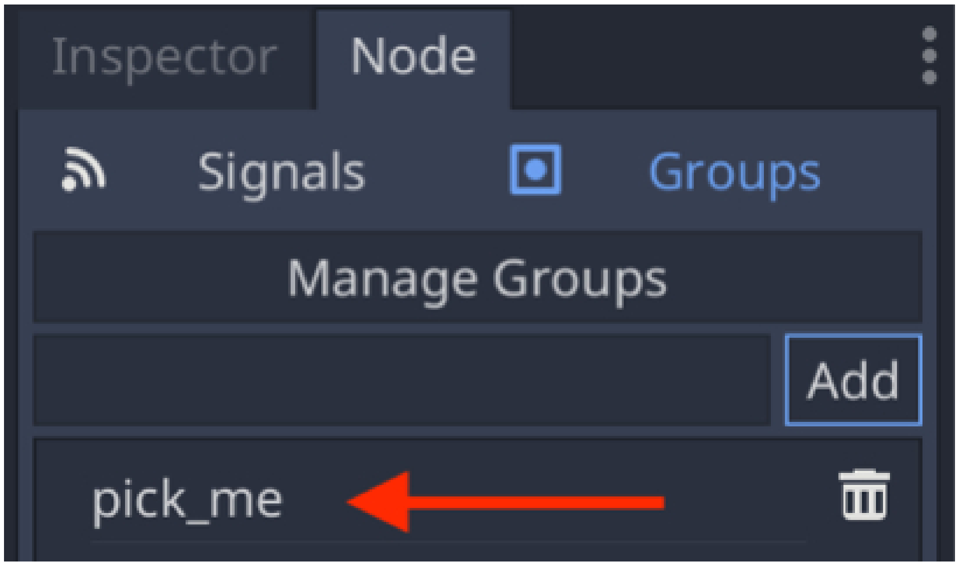
- As you click on the button labelled “**Groups**”. A new window will appear where you can define (and allocate) a new group for the current node.



- Please type the word “**pick_me**” in the field below the label “**Manage Groups**” and then click on the button labeled “**Add**”, as described in the next figure.



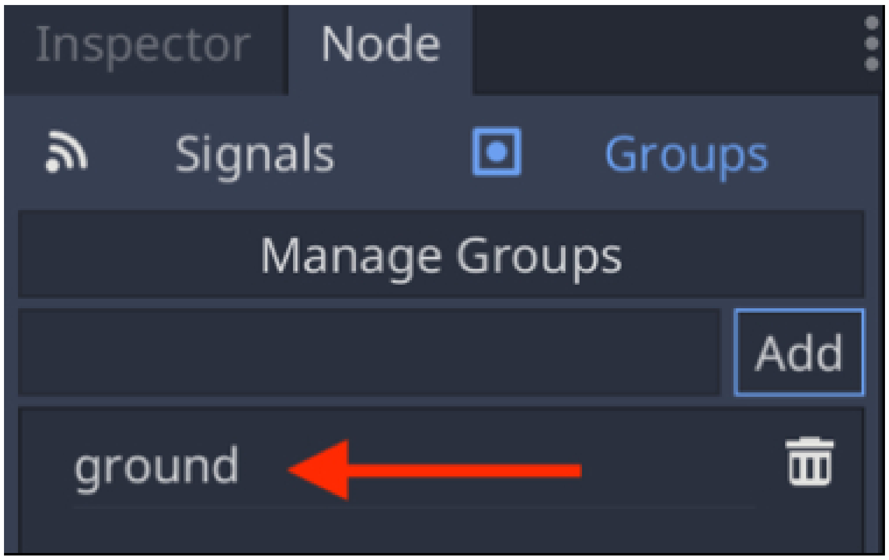
- You should then see that the group has been added.



- So at this stage the group (or tag) called **pick_me** has been created and allocated to the node **box_to_collect**.

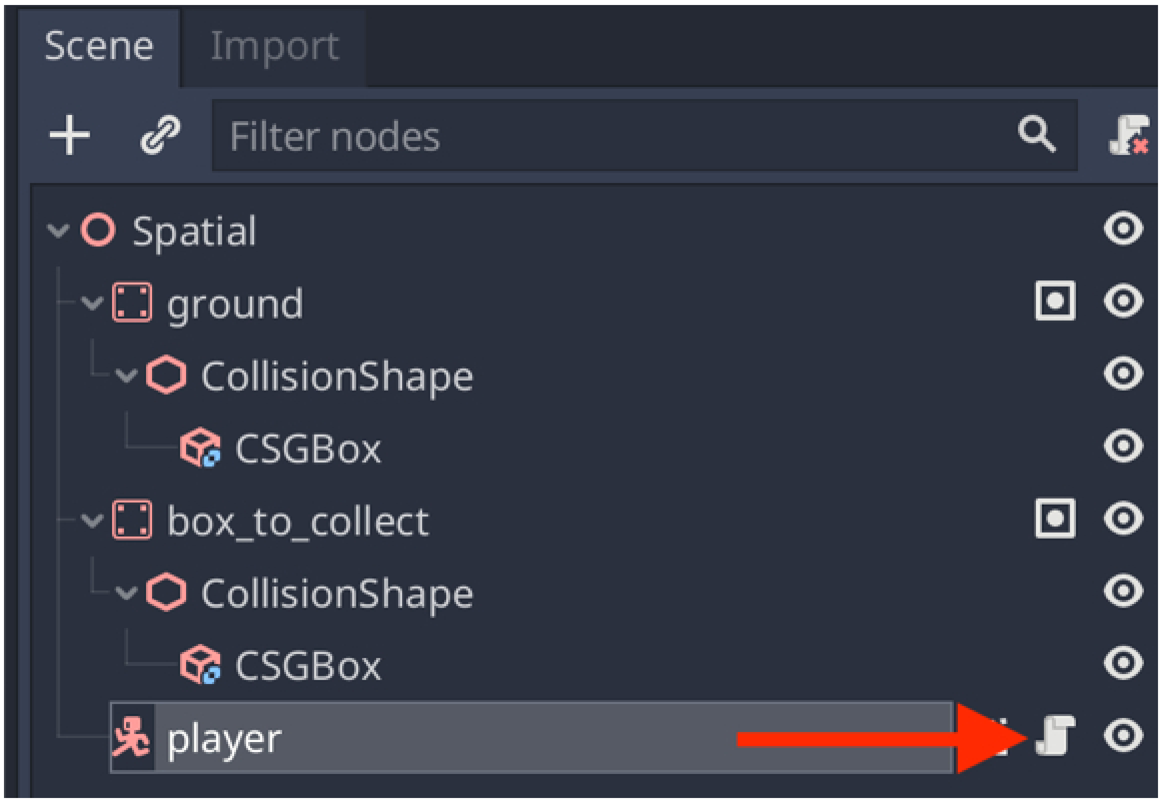
Now that we know how to create a group, we can repeat the previous steps to create and apply a tag called **ground** to the object **ground**. This tag will be used to detect whether we are colliding with the ground. Please select the object called **ground**, and repeat the previous steps to create a tag/group named **ground** and apply it to the **ground** object as follows.

- Please select the node called "**ground**" in the **Scene** tree.
- Using the **Inspector** select **Node | Groups**.
- Click on the button labelled "**Groups**".
- Enter the word "**ground**" in the text field and press "**Add**".
- This should add and allocate the group named "**ground**" to the node called "**ground**".

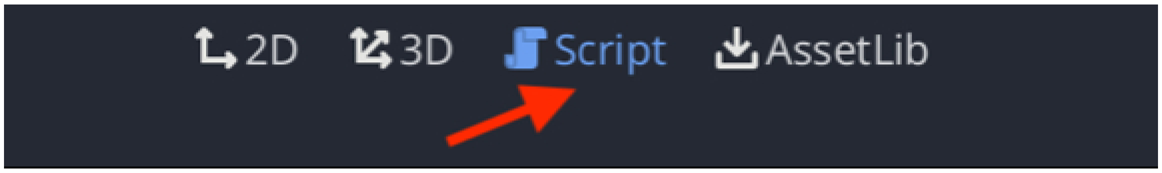


At this stage we are ready to create the logic of the game level in a new script that will be called when a collision occurs between the **First-Person Controller** and other objects.

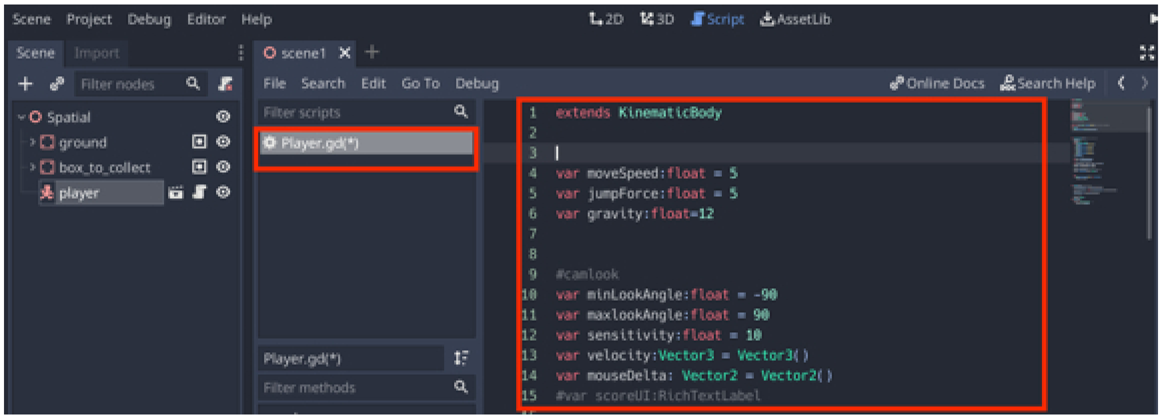
- Please click on the script logo to the right of the node called "**player**" in the **Scene** tree.



- This will switch Godot to the **Script** workspace.



- It will also show the content of the script called **Player** that is already attached to the object player.



```
extends KinematicBody


var moveSpeed:float = 5
var jumpForce:float = 5
var gravity:float=12


#camlook
var minLookAngle:float = -90
var maxlookAngle:float = 90
var sensitivity:float = 10
var velocity:Vector3 = Vector3()
var mouseDelta: Vector2 = Vector2()
#var scoreUI:RichTextLabel
```

This script contains code that makes it possible for the player to move and look around; it includes a set of member variables and functions.

As you scroll through the script, you may notice several functions, including:

- _ready.
- _ physics_process.
- _process.
- _input.

In the next section, we will modify these functions so that the player can also collect items when colliding with them.

- Please add the following code at the end of the function **_physics_process**.

```
for index in get_slide_count():
print ("Collision")
```

In the previous code, we use the function **get_slide_count**; this function usually returns the number of times the **KinematicBody** node used to move our player around has collided with other objects.; we then loop through all the different objects that we have collided with, and every time this happens, we print the message "**Collision**".

Let's execute the script and test if it works as expected.

- Please save the script (**CTRL** + **S**).
- Play the scene (**F6** for Windows or **CMD** + **R** for MacOS).
- You should see the message **"Collision"** displayed several times in the **Output window**; this is because you are colliding with the **ground** constantly.
- You can now stop the scene (**CTRL** + **Q** for Windows or **CMD** + **Q** for MacOS) and switch back to your script.

We will now try to display more information about the object that we are colliding with:

- Please modify the function **_physics_process** so that it looks as follows (changes are highlighted in bold):

```
for index in get_slide_count():
var collision = get_slide_collision(index)
print("Collision with " + collision.collider.name)
```

In the previous code:

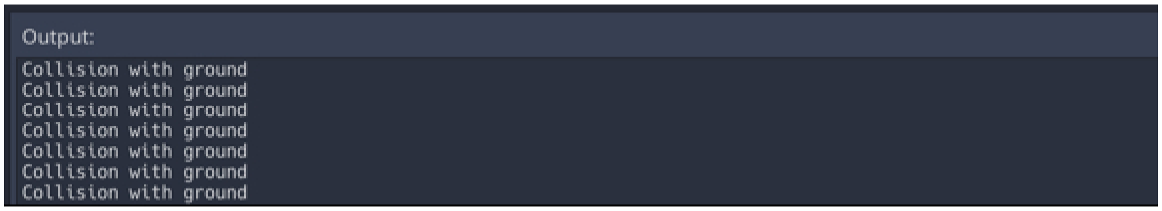
- We use the same loop as previously, so that we can go through all the items that we are currently colliding with.
- Each of the items that we are colliding with has an index.
- We then create a variable called **collision** that will correspond to one of the items that we are currently colliding with, at the current index.
- We then print the name of the collider linked to the object that we are colliding with.
- Since this is a loop, we will loop through all the objects currently in collision with the player and print their corresponding name.

- We then create a new variable of type **String**. This variable is set with the **name** of the object of the collider involved in the collision.

The function **_physics_process** is similar to the function **_process** that we have used in the previous chapters. It was preferred to (and more relevant than) the function **_process** because it deals with physics, and is ideal when managing the movement of kinematic objects, such as our First-Person player.

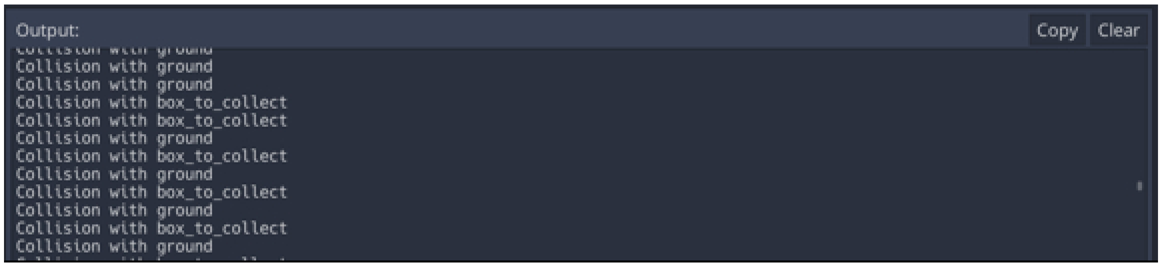
Now that we have amended our script, let's see if and how it works:

- Please save your script.
- Play your scene.
- We should now see a message saying "**Collision with ground**" in the **Output window**.



```
Collision with ground
Collision with ground
Collision with ground
Collision with ground
Collision with ground
Collision with ground
Collision with ground
```

- Move your **FPSController** so that you collide with the box and the Output window should then display "**box_to_collect**", as illustrated in the next figure.



```
Collision with ground
Collision with ground
Collision with box_to_collect
Collision with box_to_collect
Collision with ground
Collision with box_to_collect
Collision with ground
Collision with box_to_collect
Collision with ground
Collision with box_to_collect
Collision with ground
```

We will now introduce a new concept: **Removing nodes**. The idea is that to collect an object (i.e., to make it disappear from the scene) we will destroy it. So we will destroy any object when we are colliding with it if its label is **pick_me**.

Let's go ahead:

- Please stop playing the scene (**CTRL + Q** for Windows or **CMD + Q** for MacOS).

- Please, switch back to the code editor.
- Modify the function **_physics_process** as follows (new code in bold):

for index in get_slide_count():

var collision = get_slide_collision(index)

if (collision.collider.is_in_group("pick_me")):

print("Collision with " + collision.collider.name)

collision.collider.queue_free()

In the previous code:

- We check that the object that we are colliding with is part of the group called "**pick_me**".
- We then print the name of this node.
- Finally, we destroy (or delete this node) using the function **queue_free**.

You can now save this code and test it in Godot. You should see that the box disappears after you collide with it.

collecting several boxes

Now, that the collection system works, we could duplicate the box labeled **box_to_collect** several times and test the scene to ensure that we can pick up all the duplicates:

- Please Switch to the **Godot** Editor.
- Duplicate the box labeled **box_to_collect** three times: Select the object **box_-to_ collect** in the **Hierarchy** window (or the **Scene**) and press **CTRL + D**.
- In the **Scene** view, move the duplicates apart.
- Play the scene and check that each box disappears upon collision with the player.

ADDING A SCORING SYSTEM

As for many games, it is useful to have a scoring system. So, we will create one to count the number of boxes collected.

Please modify your code as follows:

- Switch to the **Script** workspace.
- Add this code at the beginning of the script **Player.gd** (new code in bold).

extends KinematicBody

var score:int = 0

In the previous code, we declare an integer named score for which the initial value is 0;

- Next, please modify the function **_physics_process** as follows (new code in bold).

if (collision.collider.is_in_group("pick_me")):

print("Collision with " + collision.collider.name)

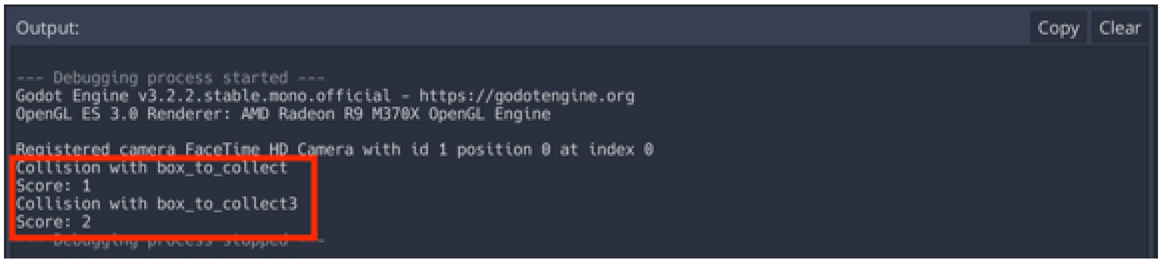
collision.collider.queue_free()

score += 1

print("Score: "+str(score))

In the previous code, whenever we collide with a box that belongs to the group **pick_me**, we destroy it, we increase the score by one, and we then print the score in the **Output** window.

Once you have made these modifications, you can save your code, play the scene, and check that after collecting boxes, the score is updated and displayed in the **Output window**.

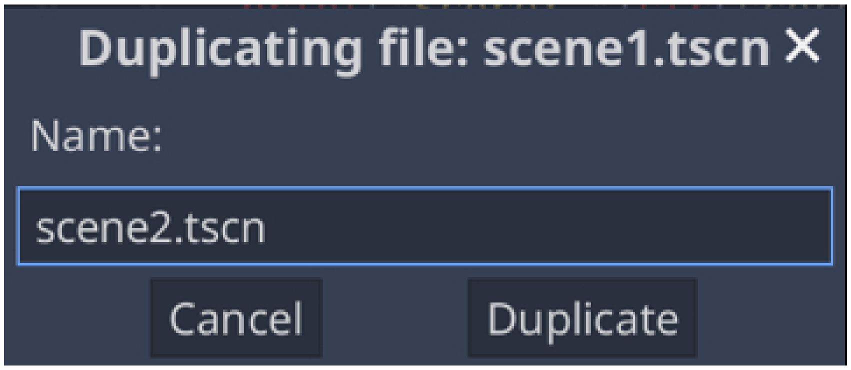


```
--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Collision with box_to_collect
Score: 1
Collision with box_to_collect3
Score: 2
--- Debugging process stopped ---
```

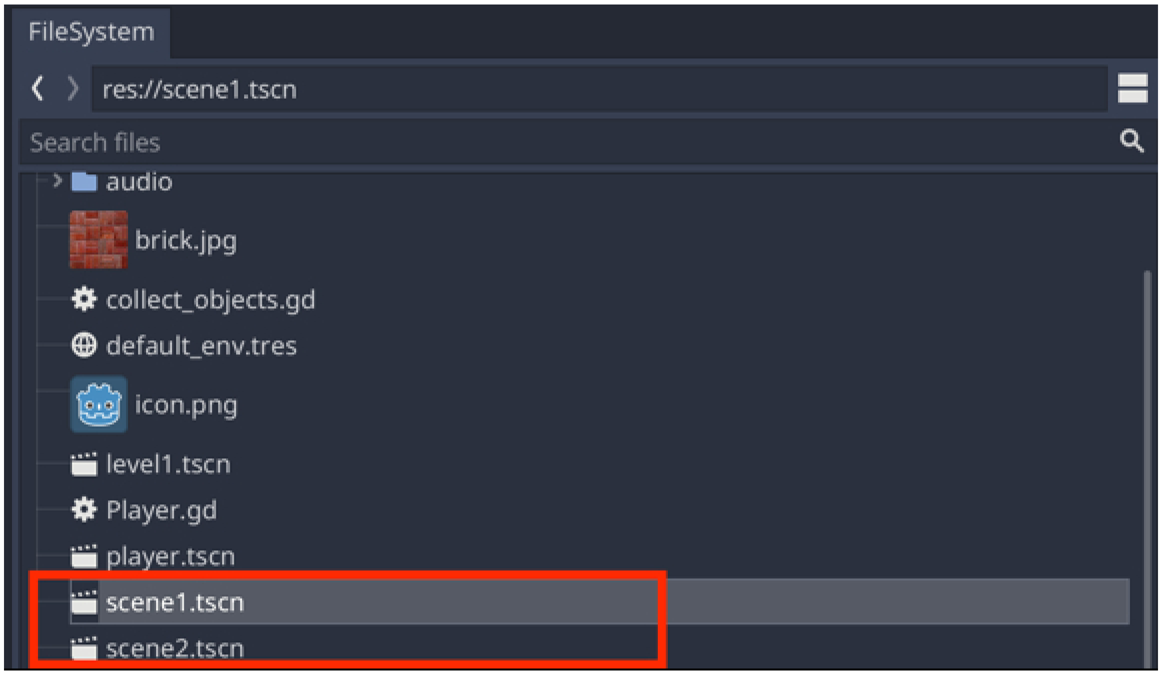
LOADING A NEW LEVEL BASED ON THE SCORE

The last thing we will do is to change level (scene) whenever we have collected four boxes; to do so, we will create a new scene, which will be a duplicate of the current scene, and load it whenever we have collected enough boxes. First let's duplicate the current scene.

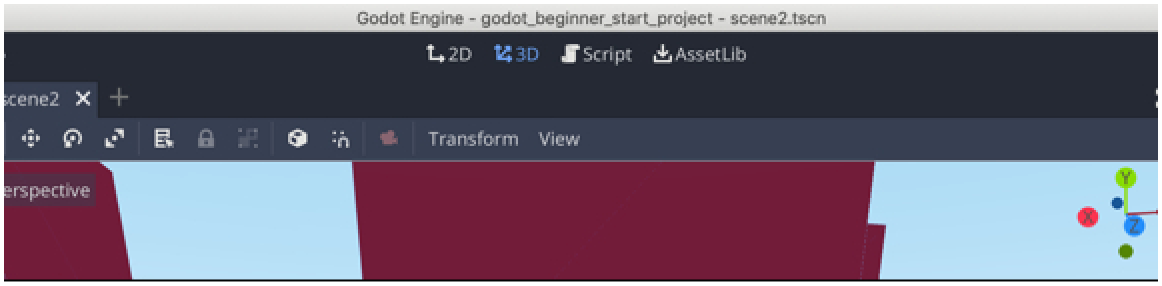
- If you haven't already done so, please save the current scene (**CTRL + S**).
- In the **FileSystem** window, look for your current scene (**scene1.tscn**) by using the corresponding search field and by typing **scene1** in it, if need be.
- Select your current scene (**scene1**) from the search result and duplicate it (**CTRL + D**).
- A new window will appear where you can specify a name for your new scene.



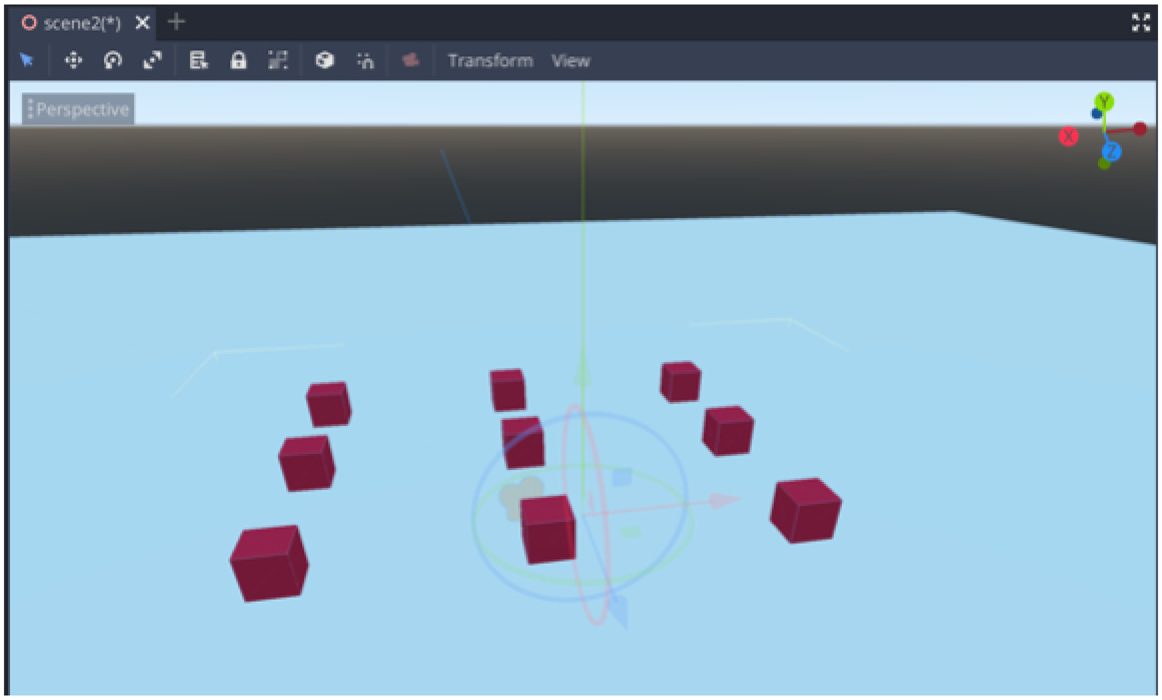
- You can enter a new name, for example **scene2** and then press the button labelled "**Duplicate**".
- The new scene should now appear in the **FileSystem** window, as illustrated in the next figure.



- Once you have located the duplicated scene (i.e., **scene2**), double-click on this scene to open it (you should see that the name of the current scene has changed at the top of the window, as illustrated on the next figure); I have renamed my duplicate **scene2,** but feel free to use a different name.



- Change the layout of the new scene (i.e., **scene2**), so that it looks different from the previous scene and so that you recognize it as it is loaded. For example, you can duplicate boxes and move them as described in the next screenshot.



Once this is done, save your scene (**CTRL + S**), open the previous scene (i.e., the original scene, **scene1**), by double-clicking on it in the **FileSystem** tab or by using: **Scene | Open Scene.**

At this stage, we just need to check that the score is three or more before we load the new scene (**scene2**).

- Please click once on the node called **player** in the **Scene** tree.
- Make sure that the **Script** workspace is active so that you can modify the script called **Player.gd**.
- In the script **Player.gd**, modify the following code in the function **_physics_process** as follows (new code in bold).

```
score += 1
print("Score: "+str(score))
if (score >= 3):
get_tree().change_scene("res://scene2.tscn")
```

In the previous code, if the score is greater than or equal to 3, we load a new scene called **scene2** (assuming that it has been saved at the root of the project, i.e., in **res://**)

You can now save your code and play the current scene (i.e., **scene1)**; you should be able to collect three boxes and be transferred to the next level (i.e., **scene2**) immediately after.

LEVEL ROUNDUP

In this chapter, we have learned about creating a script using GDScript. We also became more comfortable with functions, variables, and their properties. We managed to create and use scripts to detect collisions, to collect objects, increase the score, and load a different level accordingly. So yes, we have made some considerable progress, and we have by now looked at several programming structures as well as common errors that you may come across on your coding journey.

Checklist

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

- Create and apply groups/tags.
- Use and modify built-in functions.
- Call a function.
- Detect collision between the player and other objects.
- Detect a group.
- Attach a script to an object.
- Destroy an object.
- Create messages made of static and dynamic information (i.e., append the two).
- Launch your game using the keyboard shortcut *F6 FOR WINDOWS OR CMD + R FOR MACOS.*

Quiz

It's now time to check your knowledge with a quiz. Please answer the following questions. The solutions are on the next page. Good luck!

1. Please specify whether this statement is **TRUE** or **FALSE:**

The function **get_slide_collision** can be used to detect when the player collides with other objects.

1. Please specify whether the following statement is **TRUE** or **FALSE:**

Given that the object we are colliding with in the next script is named **ground** and that it has been given the tag **pick_me**, the following code will print the message **Collided with pick_me** in the **Output window**.

```
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("pick_me")):
print("Collided with " + collision.collider.name)
```

1. Write the missing line in this code to be able to destroy the object we have collided with.

```
if (collision.collider.is_in_group("pick_me")):
```

<MISSING LINE>

1. Please specify whether the following statement is **TRUE** or **FALSE:**

By default, all scenes included in the current project can be opened from a script.

1. Find the error in the following code.

```
for index in get_the_slide_count():
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("pick_me")):
```

1. Please specify whether the following statement is **TRUE** or **FALSE:**

Any scene selected in the **Scene** tree window can be duplicated using the shortcut *CTRL + D*.

1. Please specify whether the following statement is **TRUE** or **FALSE:**

If the scene **scene4** has been saved at the root of the project (i.e., res://), the following code will load it.

```
get_tree().change_scene("res://scene4.tscn")
```

1. What does this error message most likely mean: **"The identifier score is not declared in the current scope"**.

a) You used a variable that has not been declared yet.
b) You may have forgotten to indent your code.
c) A colon was forgotten at the end of a conditional statement.

1. What does this error message most likely mean: **": expected at the end of line"**.

a) You used a variable that has not been declared yet.
b) You may have forgotten to indent your code.
c) A semi-colon was forgotten at the end of a statement.

1. If the **Output window** shows errors and you can't seem to be able to play your scene, what can you do?

a) Check the code using the error message provided (i.e., script name, error line and column).

b) Correct the error.

c) All the above.

Answers to the quiz

1. **TRUE**.
2. FALSE (it will display the name of the node).
3. Write the missing line in this code to be able to destroy the object we have collided with.

```
if (collision.collider.is_in_group("pick_me")):
collision.collider.queue_free()
```

1. **TRUE**.
2. Find the error in the following code.

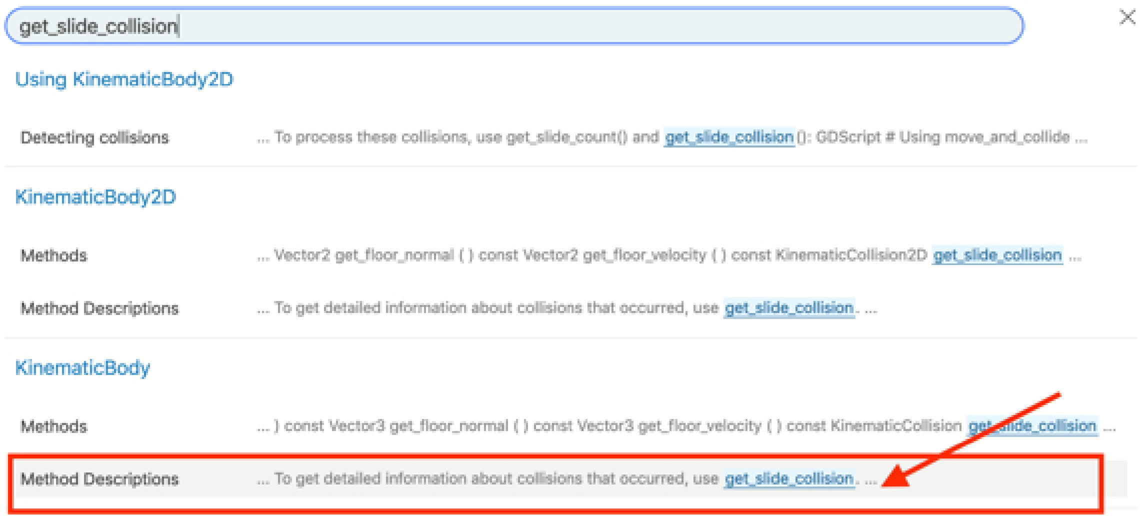
```
for index in get_the_slide_count():#that should read get_slide_count
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("pick_me")):
```

1. **TRUE**.
2. **TRUE**.
3. **a**
4. **c**
5. **c**

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let's put these to the test.

- Open the scripting reference (**https://docs.godotengine.org/en/stable**).
- Enter the word **get_slide_collision** in the search field.
- In the contextual menu, click on the link **get_slide_collision** within the section **Method Descriptions**, as illustrated in the next figure.



get_slide_collision

Using KinematicBody2D

Detecting collisions ... To process these collisions, use get_slide_count() and get_slide_collision(): GDScript # Using move_and_collide ...

KinematicBody2D

Methods ... Vector2 get_floor_normal ( ) const Vector2 get_floor_velocity ( ) const KinematicCollision2D get_slide_collision ...

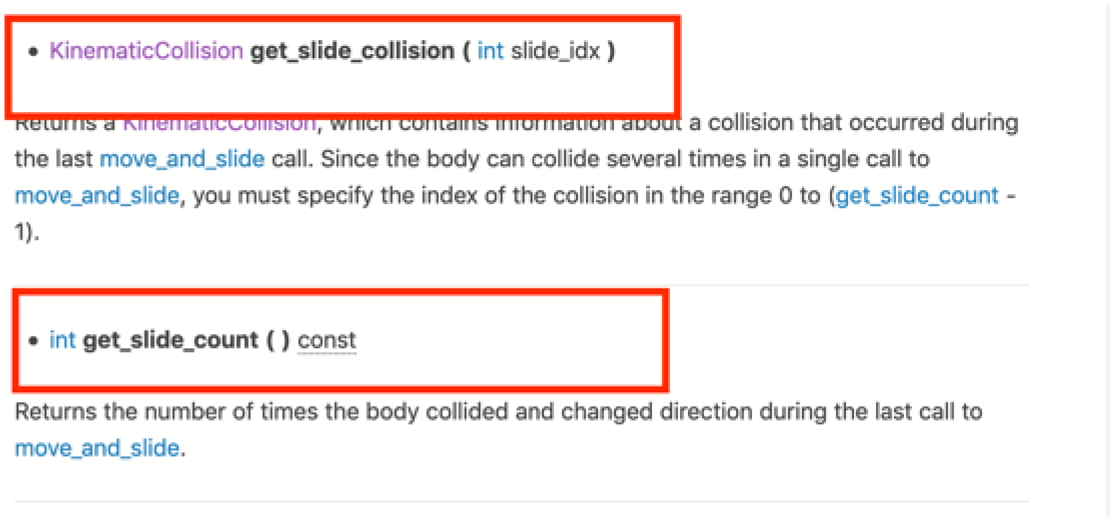
Method Descriptions ... To get detailed information about collisions that occurred, use get_slide_collision. ...

KinematicBody

Methods ...) const Vector3 get_floor_normal ( ) const Vector3 get_floor_velocity ( ) const KinematicCollision get_slide_collision ...

Method Descriptions ... To get detailed information about collisions that occurred, use get_slide_collision. ...

- In the new window, read the information on the function **get_slide_collision** and **get_slide_count**.



- KinematicCollision **get_slide_collision (int slide_idx)**

Returns a KinematicCollision, which contains information about a collision that occurred during the last move_and_slide call. Since the body can collide several times in a single call to move_and_slide, you must specify the index of the collision in the range 0 to (get_slide_count - 1).

- int **get_slide_count ( )** const

Returns the number of times the body collided and changed direction during the last call to move_and_slide.

Challenge 2

It is now time to do a little bit of debugging. Not to worry, this will be relatively easy, but it will get you to progressively become more comfortable with finding and fixing bugs.

- Look for a script called **debug_me.gd** in the **FileSystem** window.
- Open this script in your code editor.
- You will notice that the **Output window** will display error messages.
- Try to solve the errors in this script; there should be five errors in total.

Chapter 4: Creating and Updating a User Interface from Your Code

In this section we will discover how to create and update a simple user interface through scripting. Some of the objectives of this section will be to:

- Explain additional GDScript concepts.
- Explain how to display information from the code to the game's user interface.
- Explain how to load levels and activate objects based on conditions.
- Explain how to display information as part of the user interface.

After completing this chapter, you will be able to:

- Create and display a timer.
- Create a function that displays information onscreen.
- Modify this function so that the message disappears after a few seconds.
- Display messages when the user has collected items.
- Activate and deactivate objects from your script.

In this section, we will create a game with the following gameplay:

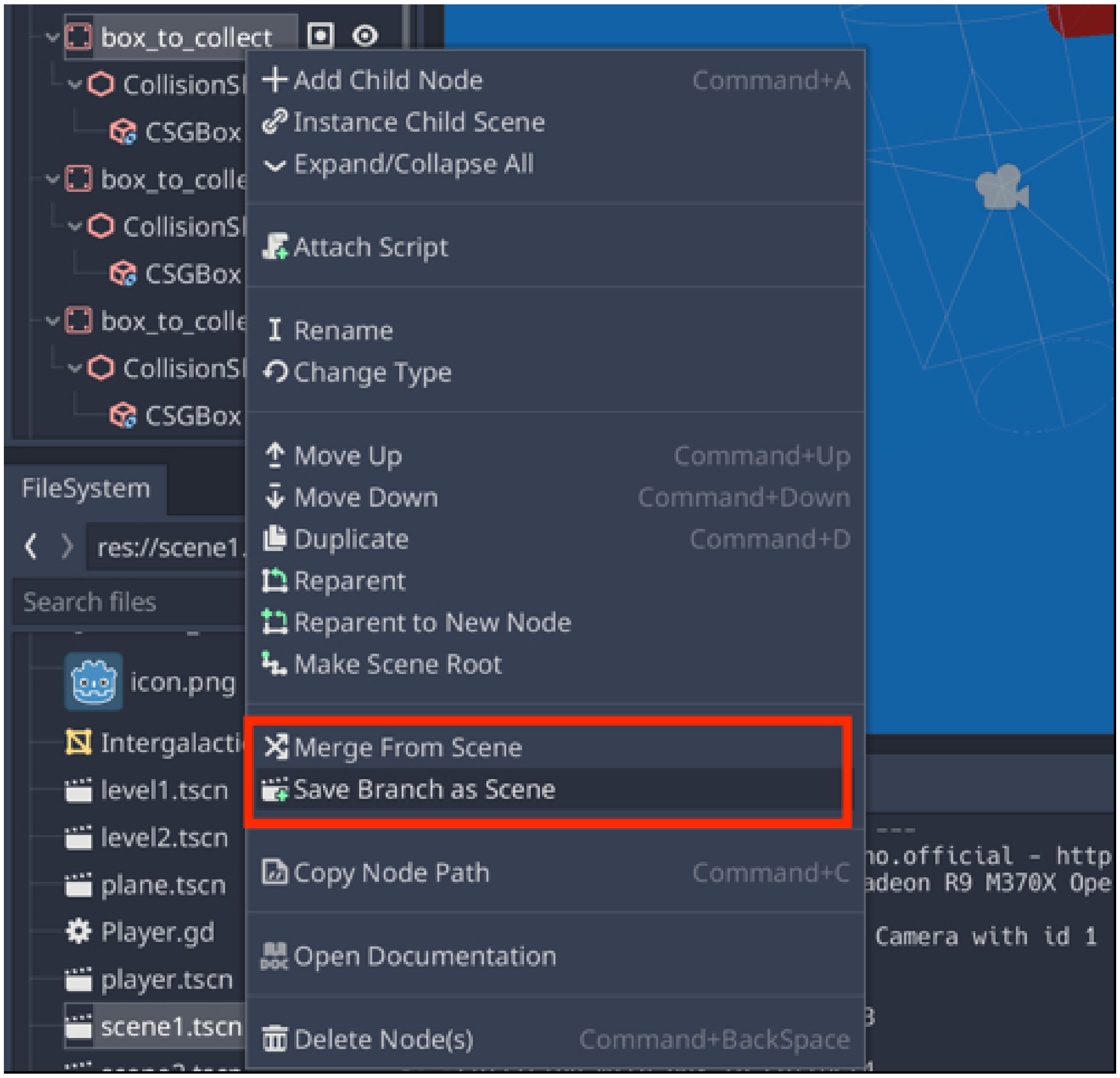
- The player starts in the maze.
- The player has two minutes to collect three boxes.
- The player needs to collect these three boxes to proceed to the next level.
- In the next level, the player needs to collect four petrol cans before s/he can use the plane and escape the level.
- The 3D environments for the two levels are already accessible in the project as **level1** and **level2**.

Setting up the scene

In this chapter we will be working with the levels **level1** and **level2**, two levels included in the start-up project; we will also use the First-Person controller that we have employed in the previous sections to move around the level. Finally, we will also reuse some of the boxes that we have created earlier.

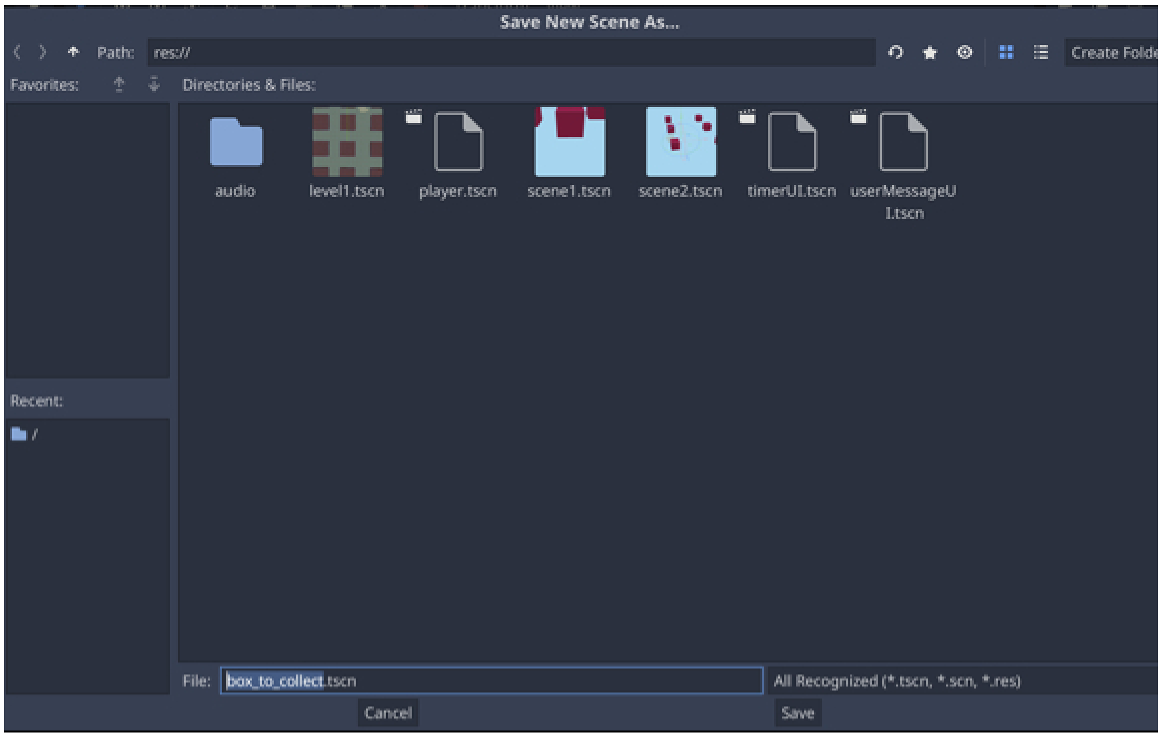
To reuse the boxes, we will first save them as nodes so that we can import them in our own scene.

- Please open the scene **scene1**.
- Select the box named **box_to_collect** in the **Scene** tree.
- Right-click on this node and select the option "**Save Branch As Scene**" from the contextual menu.

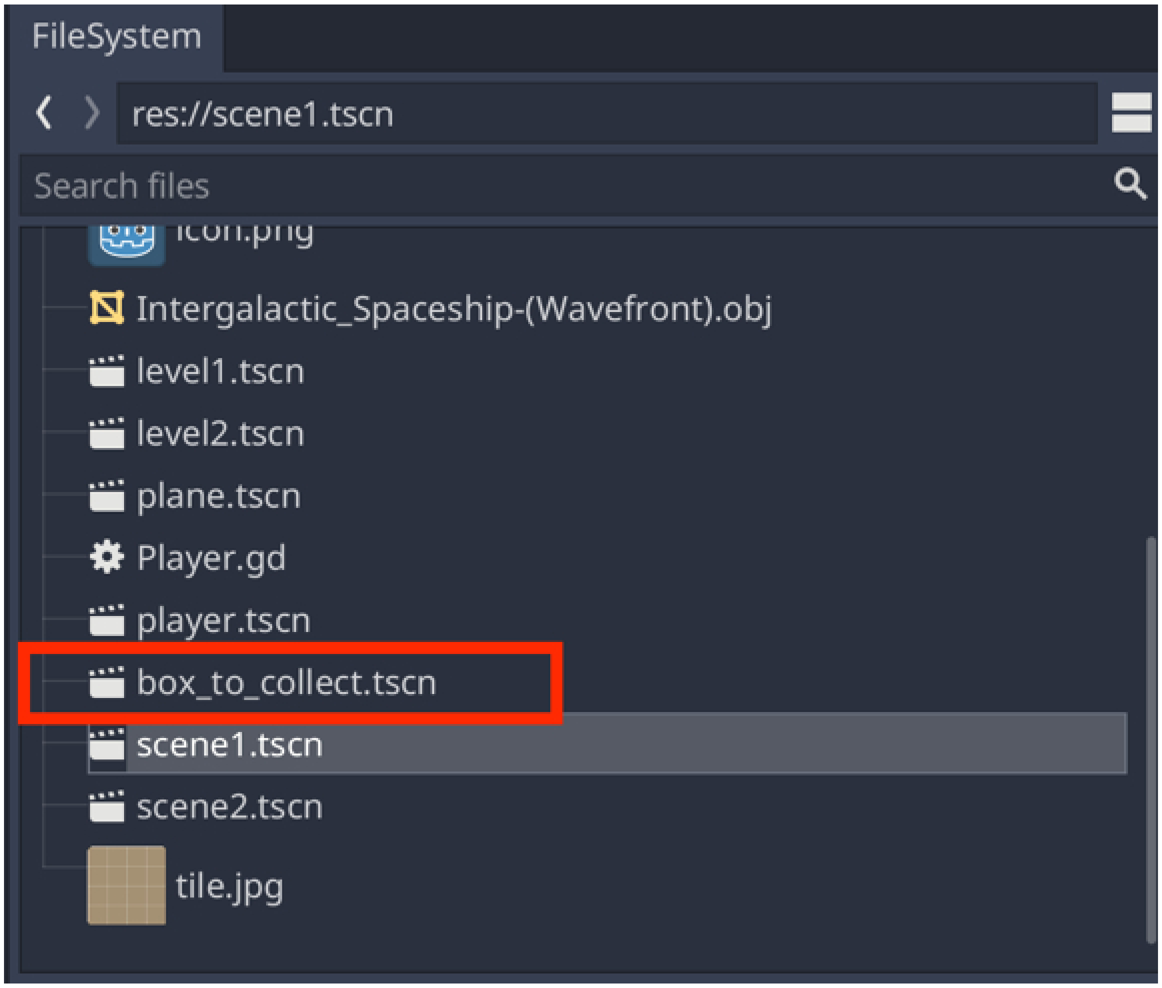


```
no.official - http
adeon R9 M370X Ope
Camera with id 1
```

- In the new window, leave the default options, and click on "**Save**".

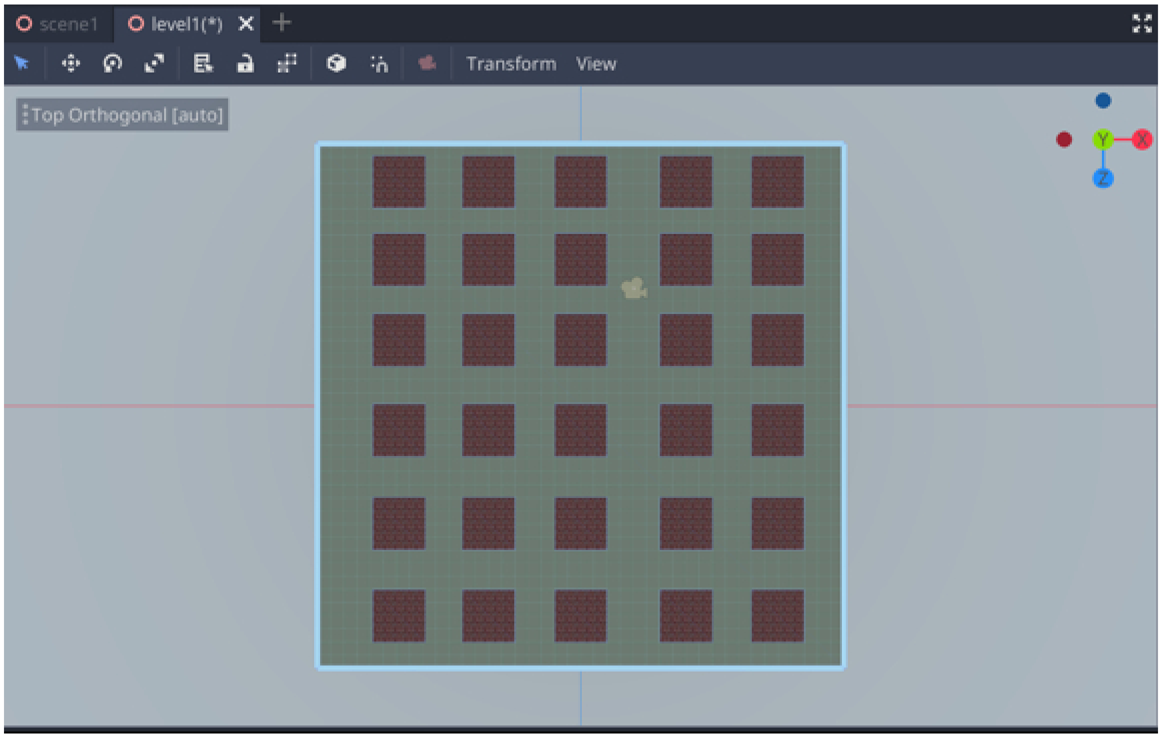


- This should create an asset (or scene) called **box_to_collect.tscn** in the **FileSystem** window. This means that we will be able to reuse this box under the name "**box_to_collect**" in any scene that is part of our project, including the scenes **level1** and **level2** that are already present in the project and that we will use later.



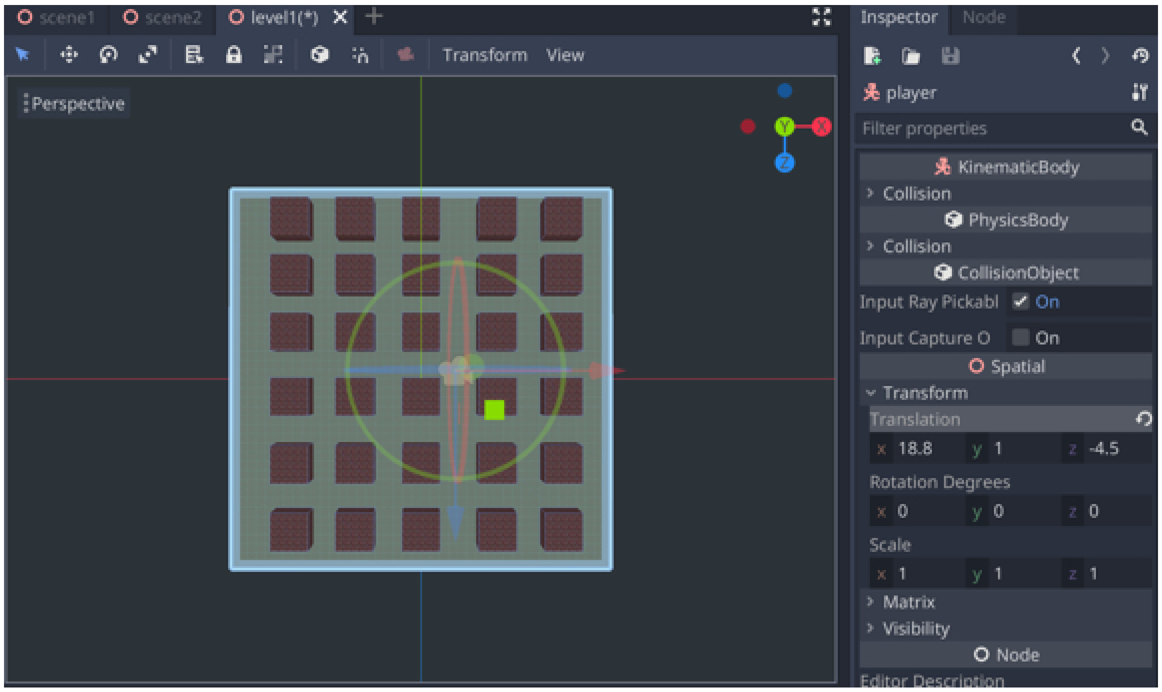
Next, we need to add a **First-Person Controller** to our scene **level1**:

- Please open the scene called **level1**.
- Deactivate the node called **ceiling** for the time being and you should be able to view the maze from above.



Once this is done, it is time to add our **First-Person Controller**:

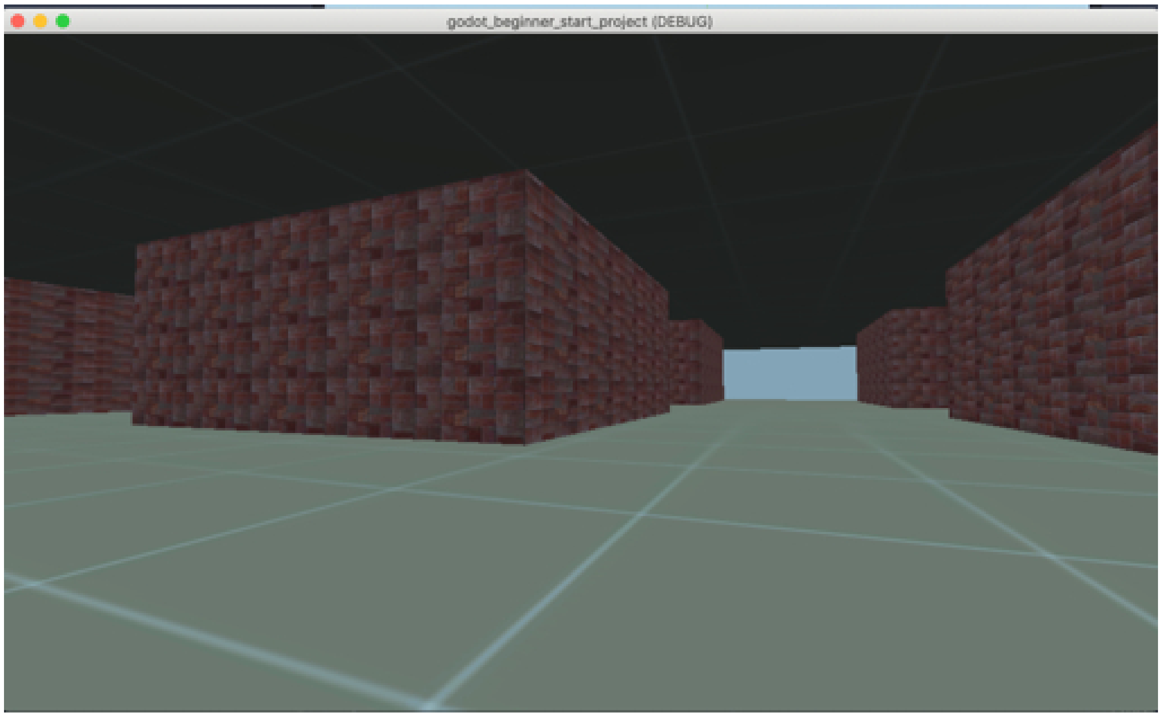
- Select the node called **Spatial** in the **Scene** tree so that the asset that we are about to add becomes a child of that node.
- Please locate the scene called **player.tscn** in the **FileSystem** window.
- Drag and drop this file to the scene.
- This will create a node called **KinematicBody**.
- Looking at the scene from the y-axis, please ensure that this object is within the area defined by the maze.



Input Ray Pickabl ✓ On
Input Capture O On

Translation: x 18.8 y 1 z -4.5
Rotation Degrees: x 0 y 0 z 0
Scale: x 1 y 1 z 1

- Please rename this node **player.** and ensure that its **y** coordinate is **1**, using the **Inspector**.
- Once this is done, you can play the scene and move around using the arrow keys, the mouse and the spacebar (to jump).

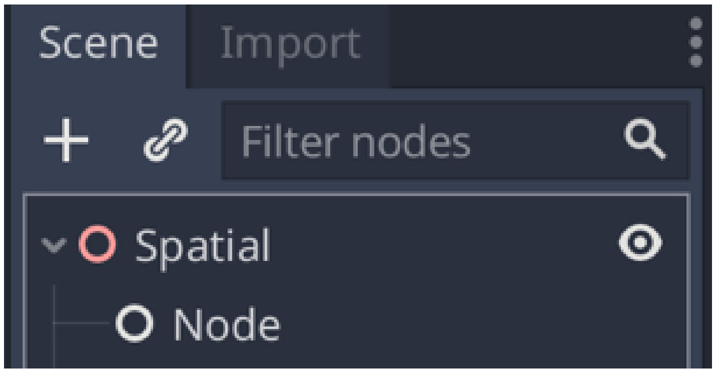


That's it: the level is ready to be used now.

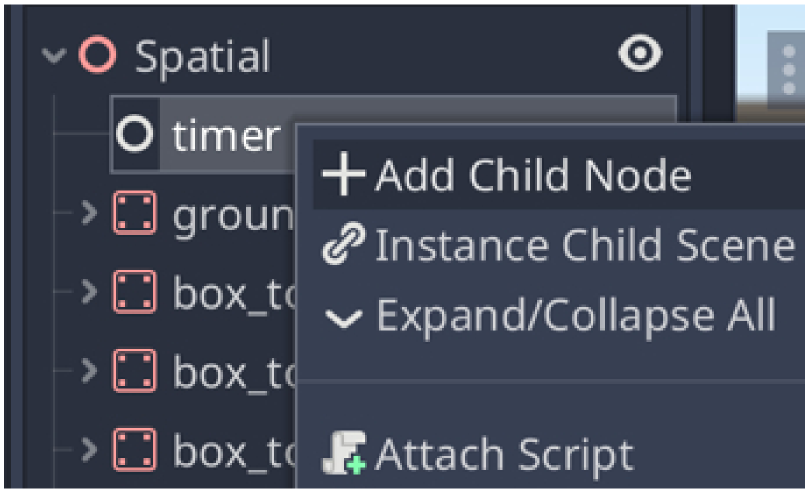
Creating a timer

To create our timer, we will start by creating a new script:

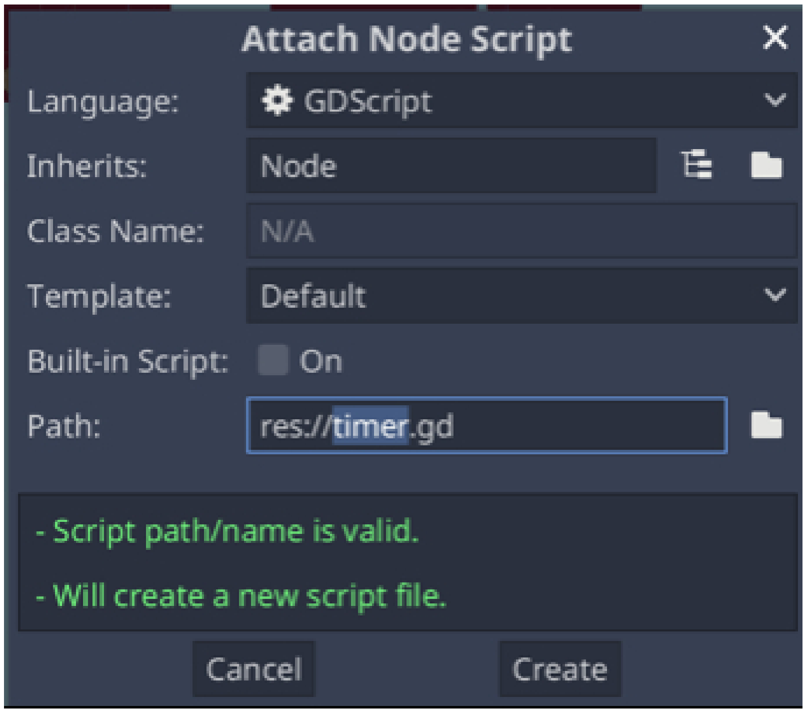
- Please open the scene named **level1,** it consists of a maze made of boxes.
- Create a new node of type **Node** as a child of the node **Spatial Node**.



- Rename this node **timer**.
- Attach a new script to this node.



- In the new window, please leave the default values and click on the button labelled "**Create**".


Attach Node Script

Language: GDScript

Inherits: Node

Class Name: N/A

Template: Default

Built-in Script: On

Path: res://timer.gd

- Script path/name is valid.
- Will create a new script file.

- This will create a new script called **timer**.
- By default, it will include the **_ready** and **_process** functions.
- The script should now be open in the editor.

Once the script is open, we will then create a variable that stores the time.

- Please modify the file as follows (new code in bold):

extends Node

var time:float = 0

var counter:int

func _ready():

pass # Replace with function body.

---

func _process(delta):

time += delta

if (time > 1):

counter+=1;

time = 0;

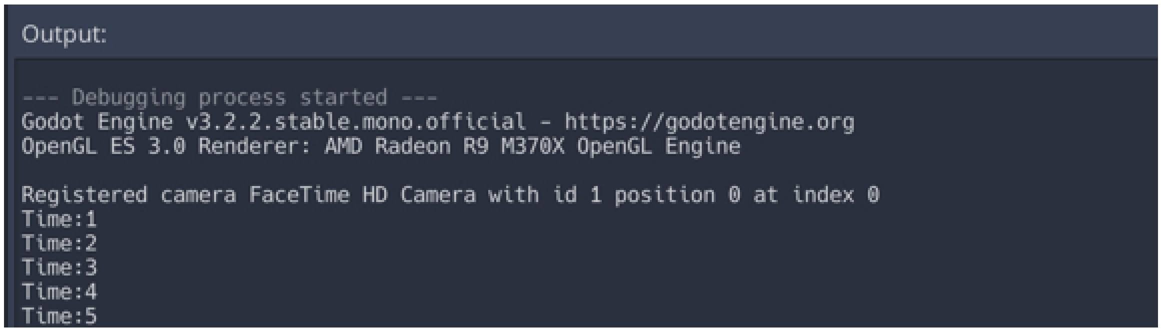
print("Time:" + str(counter))

In the previous code:

- We declare two variables: **time** and **counter**.
- The function **_ready** is left unmodified.
- In the function **_process**: we add **delta** to the variable **time**. Delta is the time in seconds between each frame; since the function **_process** is called every frame we effectively calculate the number of seconds elapsed.
- If **time** is greater than one (i.e., if one second has elapsed) we increase the value of the variable **counter** by one and reset the variable **time** to 0.
- We then print the value of the variable **counter** in the **Output** window.

So that the time is consistent across computers, it is good practice to use the variable **delta**, so that the actual number of seconds elapsed is used, regardless of the frame rate.

- Please save your script (**CTRL** + **S**).
- Check the **Output window** for any error.
- Please fix the errors, if there are any, and play the scene (**F6** for Windows or **CMD** + **R** for MacOS).
- It should display the time and update it every second, as per the next figure.



```
--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Time:1
Time:2
Time:3
Time:4
Time:5
```

We now need to display the number of minutes, following the same format.

- Please switch back to the script **timer.gd**.

- Modify the function **_process** as follows (new code in bold):

func _process(delta):

time += delta

if (time > 1):

counter+=1;

var seconds = counter%60

var minutes = counter / 60

time = 0;

#print("Time:" + str(counter))

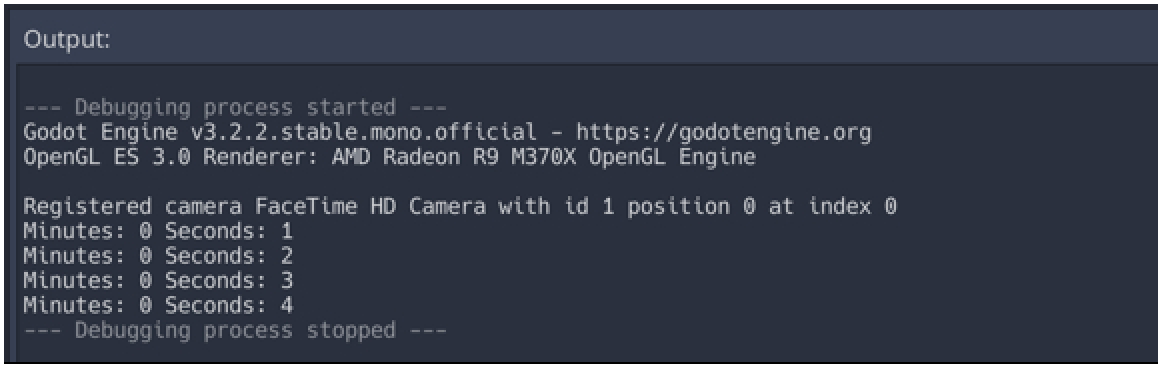
print ("Minutes: %d Seconds: %d" %[minutes,seconds])

In the previous code:

- We declare the variables **seconds** and **minutes**.
- **minutes** is obtained by dividing the variable **counter** by 60.
- **seconds** is obtained by using the **modulo** operator (the remainder of the division of counter divided by 60). This is because we need to reset the number of seconds to 0 every minute, and we use the operator modulo % that will display the remainder of the division. For example, 62%60 is 2 (or, in other words: 62 = 1*60 + 2).

You can now save your code and play the scene.

The **Output window** should look as follows:



```
--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Minutes: 0 Seconds: 1
Minutes: 0 Seconds: 2
Minutes: 0 Seconds: 3
Minutes: 0 Seconds: 4
--- Debugging process stopped ---
```

At this stage, our timer is complete when the number of seconds goes beyond 60, it will be set back to 0. To see how it works, we will just modify the initial value of the variable **counter** so that we can see what happens after one minute has elapsed, without having to wait for 60 seconds:

- Please modify the following code at the beginning of the class (new code in bold).

```
extends Node
var time:float = 0
var counter:int = 55
```

In the previous code, we set the time to 55 seconds so that we can see what happens after one minute has elapsed, without having to wait for 60 seconds.

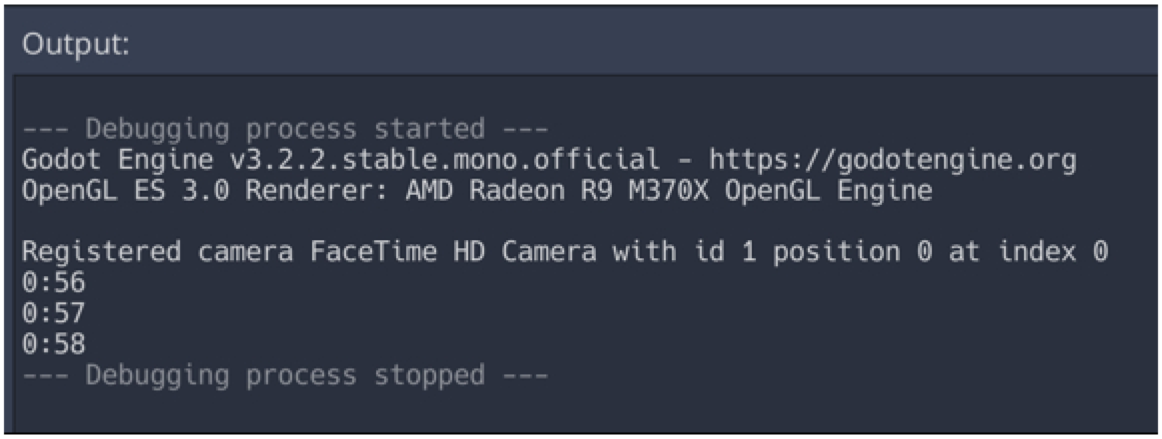
- Please save your code and play the scene.
- You should see that whenever the timer goes over one minute, the number of seconds displayed is reset to 0.

At this stage, we just need to display the time more neatly by showing only the minutes, followed by a colon, followed by the number of seconds elapsed. This will be useful as we display the time as part of the Graphical User Interface (GUI) in the next sections.

- Please modify the code of the script as highlighted in the next code snippet (changes noted in bold):

```
var minutes = counter / 60
time = 0;
#print("Time:" + str(counter))
print ("%d:%d" %[minutes,seconds])
```

- Save your code, switch to Godot, and play the scene to check that the time is displayed properly.


```
Output:

--- Debugging process started ---
Godot Engine v3.2.2.stable.mono.official - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
0:56
0:57
0:58
--- Debugging process stopped ---
```

As illustrated in the previous figure, the time is now displayed correctly.

Reloading the level when the time is up

At this stage, we have managed to display the time neatly. We will now reload the level when the time is up. To do so, we will check when the time is beyond a specific value, display a message accordingly, and then reload the level.

- Please switch back to the code editor (i.e., the last script)
- Add the following code to the function **_process** as follows (new code in bold).

if (counter > 120):

print("Time Up!")

get_tree().change_scene("res://scene1.tscn")

In the previous code above, we check that the variable **counter** is greater than 120 seconds (i.e., 2 minutes). If this is the case, then we print a message in the console window **("Time Up")** and then we reload the scene labeled **maze**.

Once you have made these changes, we just need to switch back to the text editor and change the initial value for the variable **time** to **100**, so that the time starts at 100 and so that we can see the message straight away (without having to wait for too long). So, please change the following line at the beginning of the script (new code in bold):

extends Node

var time:float = 0

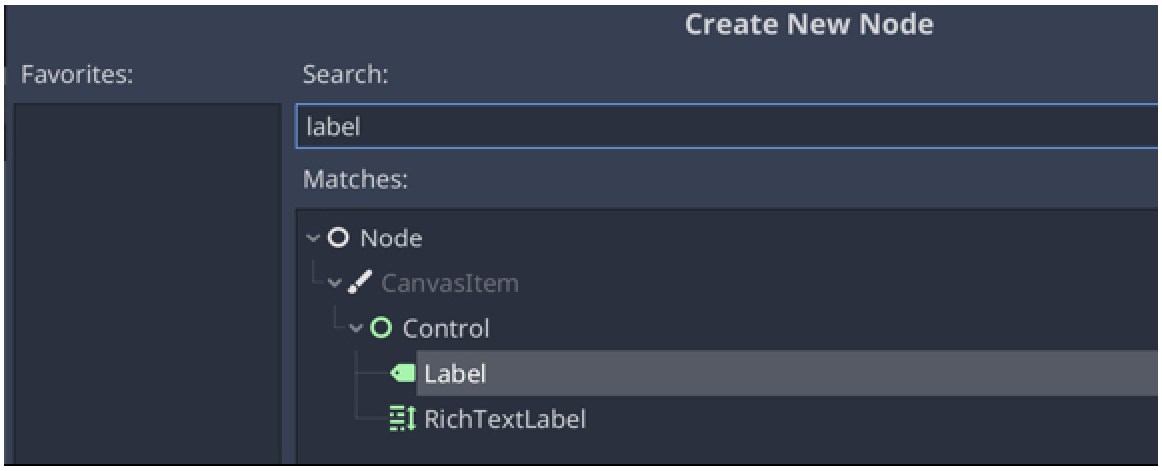
var counter:int = 100

At this stage, after saving our script, we can play the scene. You should see that the time will start at **1:40** to increase up to two minutes. After this threshold, a message should be displayed and the new level should be reloaded.

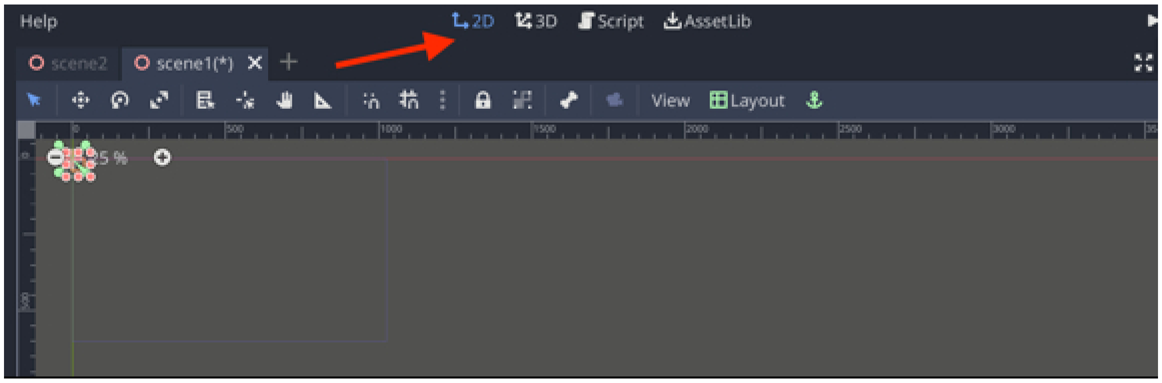
Using the user interface to display messages

Now that we have managed to display our message and reload the level, we will use the user interface for the same purpose so that the user can see messages onscreen.

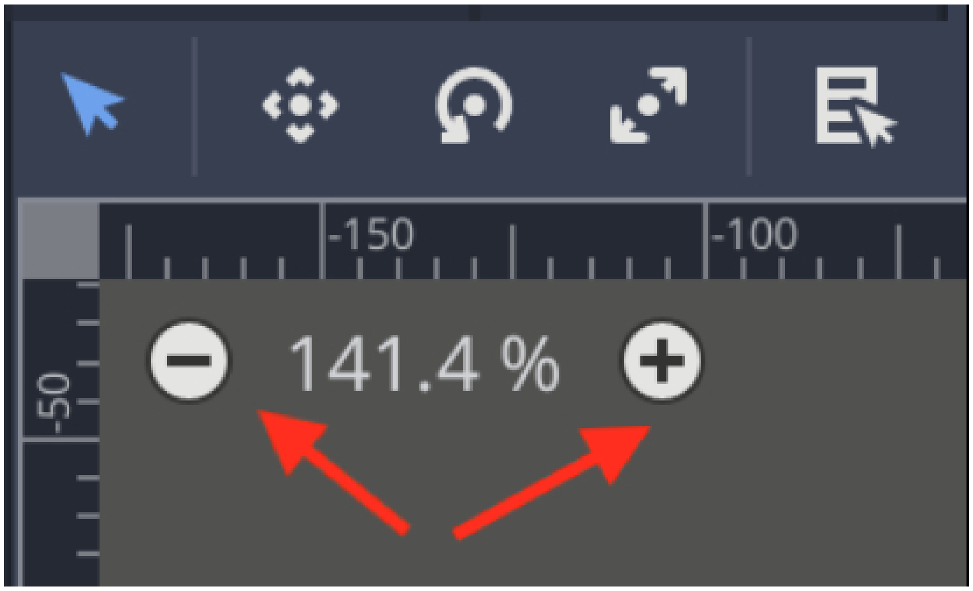
- Using the **Scene** tree, click on the **Spatial** node already present and add a child node of type **Label**.



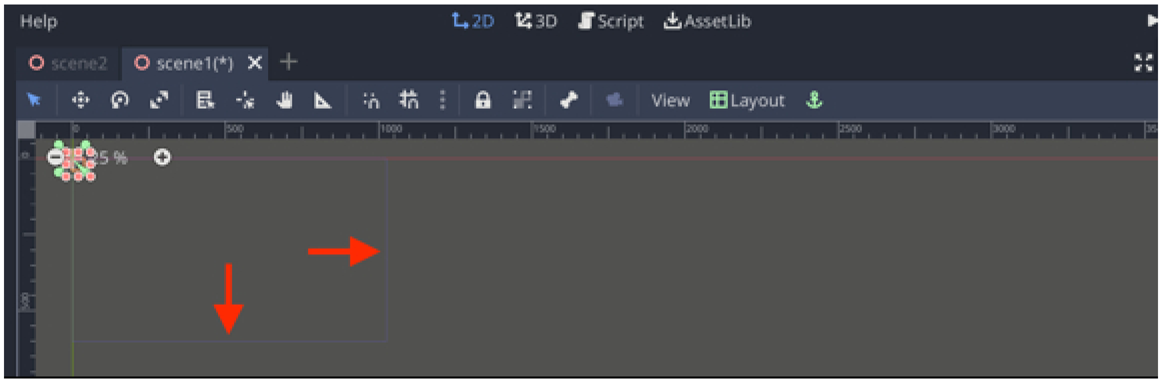
- Godot will automatically switch to the 2D workspace to show you the appearance of the new node that you have just created.



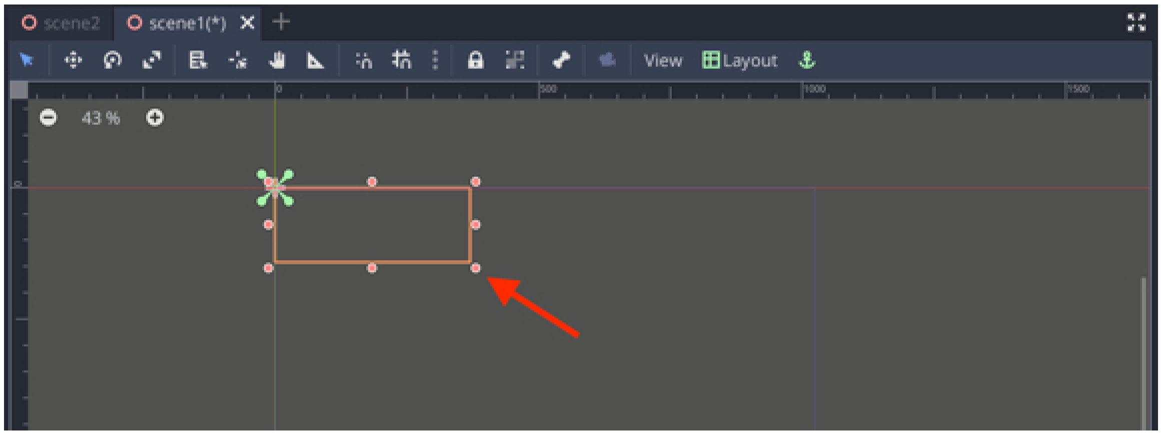
- You can zoom in or zoom out using the corresponding buttons onscreen.



- You will notice a purple rectangle that represents the visible area for the game (or screen size for the game). This outline represents the area that will be viewed onscreen by the player, so it gives an indication of how the different UI elements will be represented onscreen and their relative position.



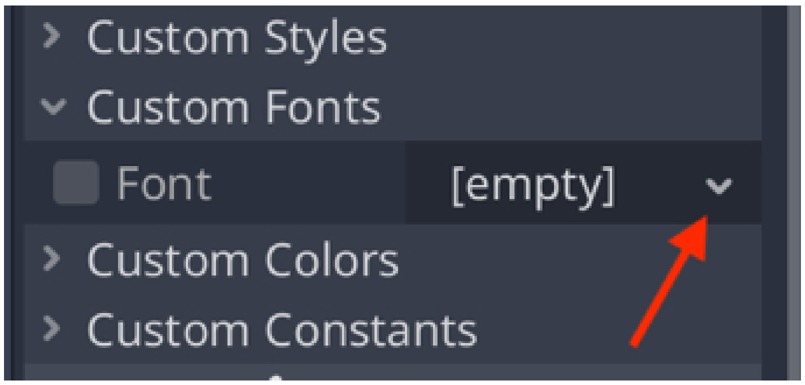
- Please drag the bottom right corner of the rectangle that is located in the top-left corner, and stretch it so that the rectangle occupies about a third of the length and the width of the visible area. This rectangle represents how the label that we have just created will be seen onscreen.



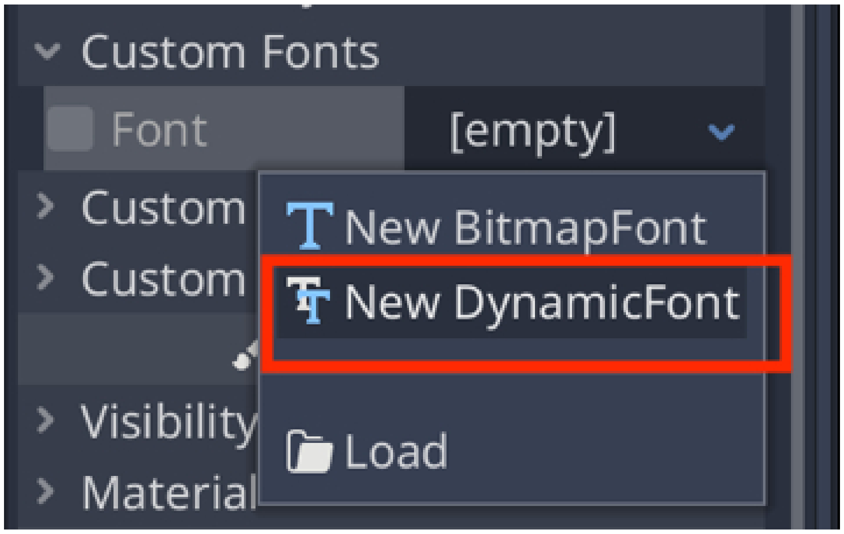
- Because the new node will be used to display the time onscreen, we will rename it accordingly to **timerUI** (right-click + **Rename** or single-click and change the name).

Next, we will need to modify the size of the font for the label that we have created; for this purpose, we will need to load a new font and then modify the font size; so let's proceed:

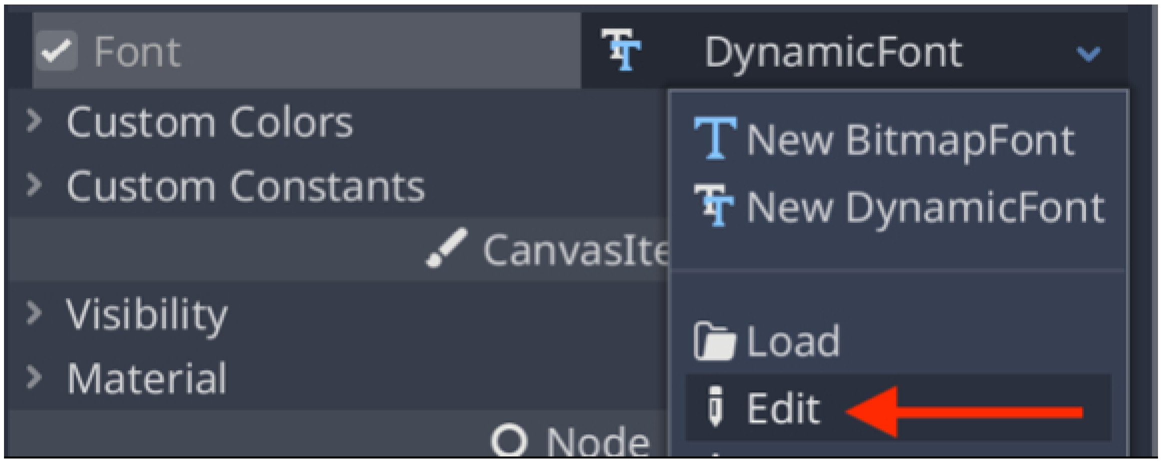
- Please select the node called **timerUI** in the **Scene** tree.
- Using the **Inspector**, scroll down to the section entitled **Custom Fonts** and click on the downward-facing arrow to the right of the label **Font**.



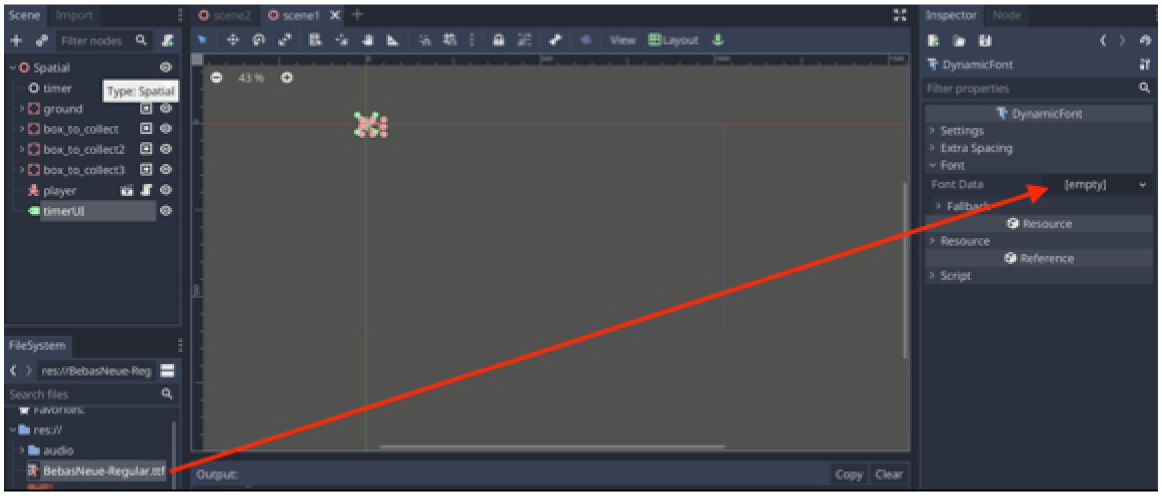
- From the contextual menu, select: **New DynamicFont**.



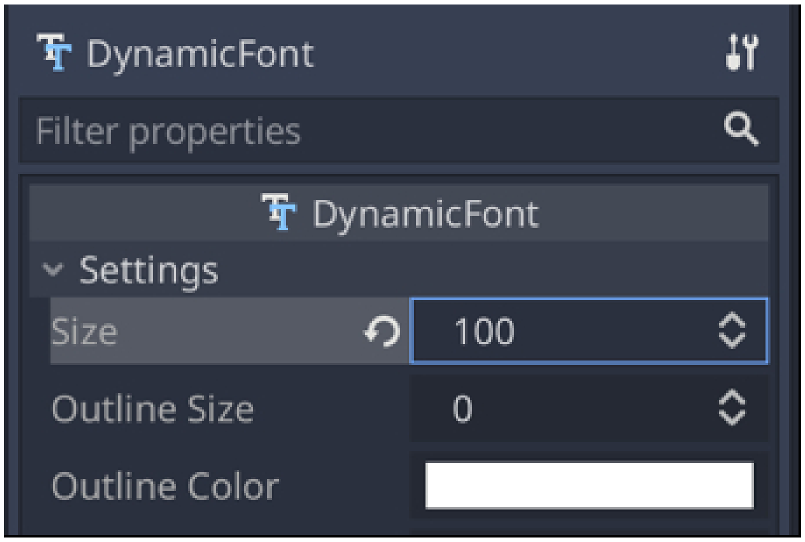
- Then click on the downwards facing arrow to the right of the **DynamicFont** label and select **Edit** from the contextual menu.



- In the new window, expand the section called **Font** and drag and drop the file called **BebasNeue-Regular.ttf** to the section called **Font Data**, as illustrated in the next figure.

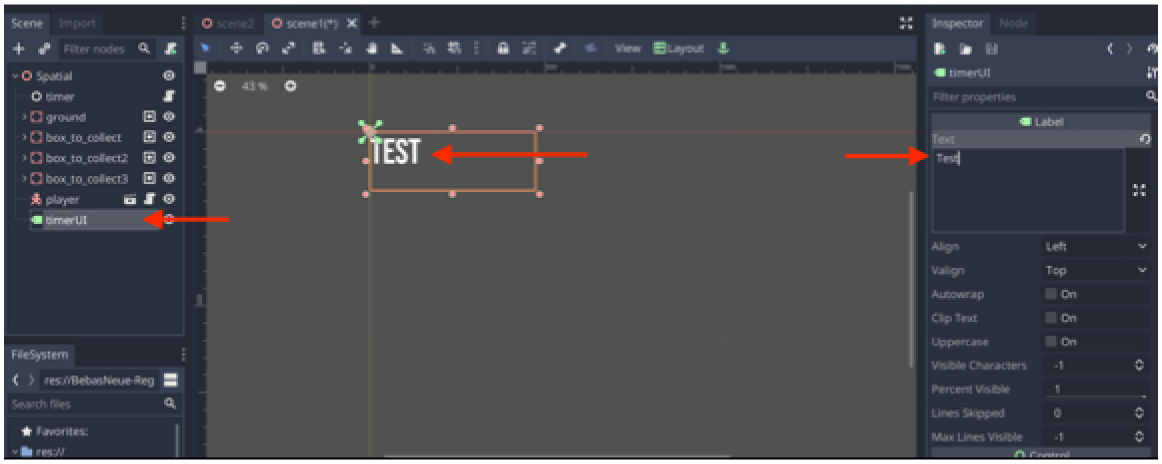


- You can now expand the section called **Settings**, and set the font size.



Size 100
Outline Size 0
Outline Color

- You can click once on the node **timerUI** in the **Scene** tree, and using the **Inspector**, enter the text **Test** in the **text** section and you should see it appearing in the interface, as illustrated in the next figure.



At this stage we need to access this **UI** element so that messages are displayed on the user interface rather than in the **Output window**.

For this purpose, we will need to find this object from the **Timer** script and use a built-in function called **get_node**, which basically looks for a node in your scene, based on its name.

- Please switch to the **Script** workspace.
- Open the script **Timer**.
- Add the following code just before the function **_ready**.

onready var timer_ui:Label = get_node("../timerUI")

In the previous code, we declare a new variable named **scoreUI** of type **Label**.

- Please add the following code to the function **_ready** as follows (new code in bold).

timer_ui.set_text("")

Using this code, we access the **text** attribute in the node **timerUI** and we set this attribute to an empty string. This means that we just clear the text for the node **timerUI** for now.

Following this, we need to modify our code so that the time is displayed on the user interface.

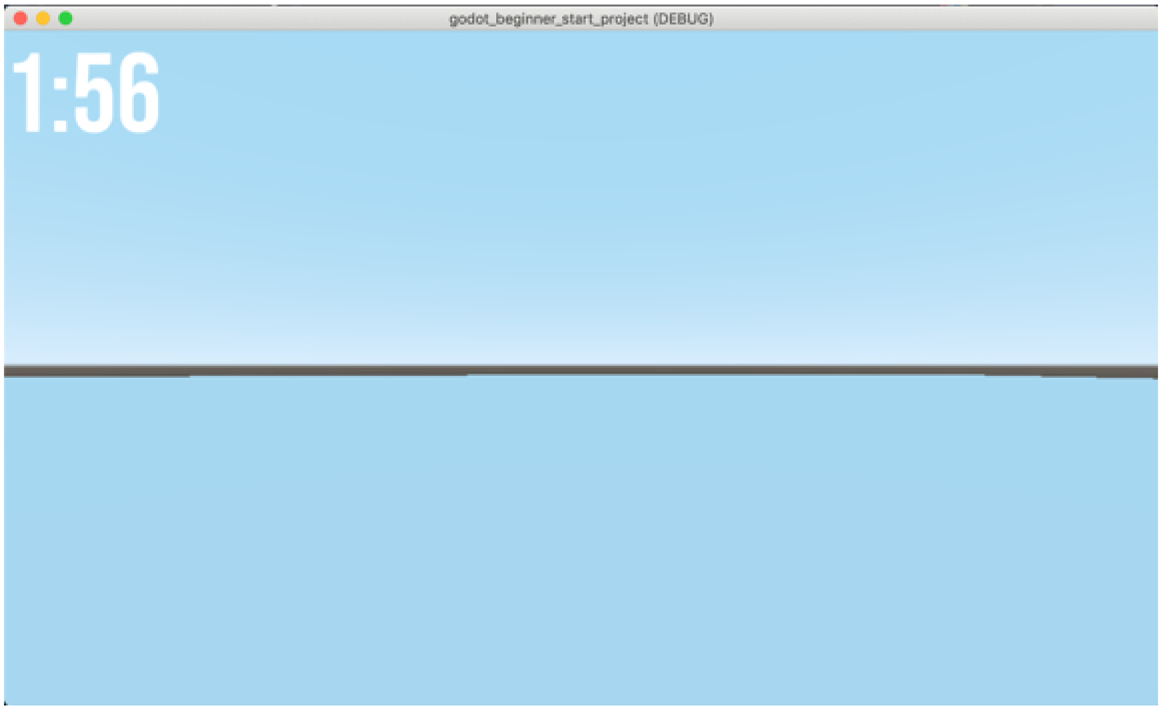
- Modify the function **_process** as follows (new code in bold):

#print ("%d:%d" %[minutes,seconds])

timer_ui.set_text("%d:%d" %[minutes,seconds])

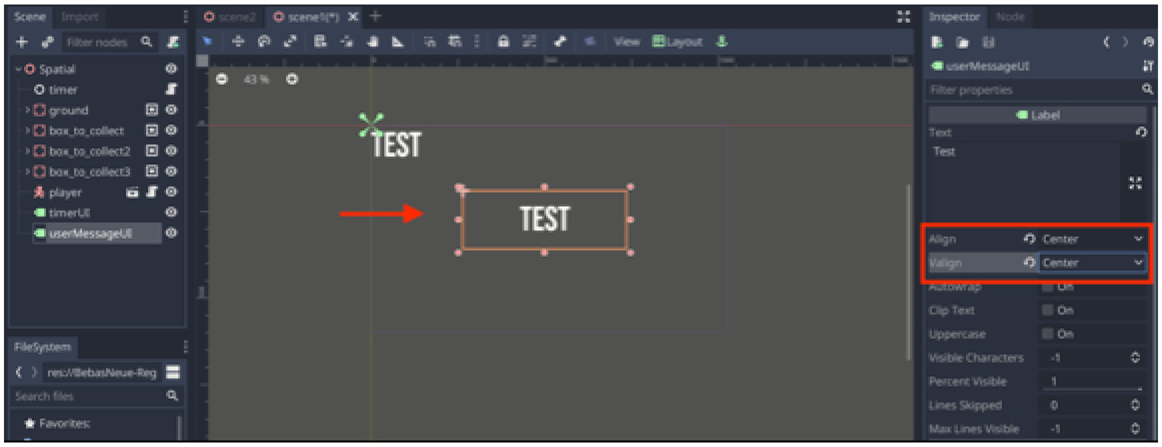
if (counter > 120):

- In the previous code, we look for the node **timerUI**, we then access its **text** attribute and set it so that it displays the number of minutes and seconds.
- You can now save your code and play the scene in Godot.
- You should see the time displayed in the top-left corner of the screen.

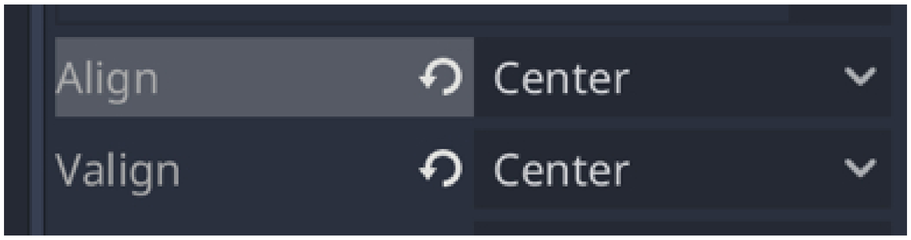


Now that the time is displayed onscreen, we can also create another **Label** node that will display other messages to the user, but this time in the middle of the screen.

- Please stop the scene.
- Duplicate the node called **timerUI** and rename the duplicate **user_message_ui.**
- Move the node **userMessageUI** to the middle of the screen (i.e., drag and drop the corresponding rectangle).



- Using the **Inspector**, change its horizontal (i.e., **Align**) and vertical alignment (i.e., **Valign**) to **Center**.


Align Center
Valign Center

We will now use code to display user messages onscreen rather than in the **Output** window:

- Please switch back to the code editor.
- Open the script **Timer**.
- Add the following code at the beginning of the script, before the function **_ready**.

```
onready var user_message_ui:Label = get_node("../userMessageUI")
```

- Add the following line to the **_ready** function.

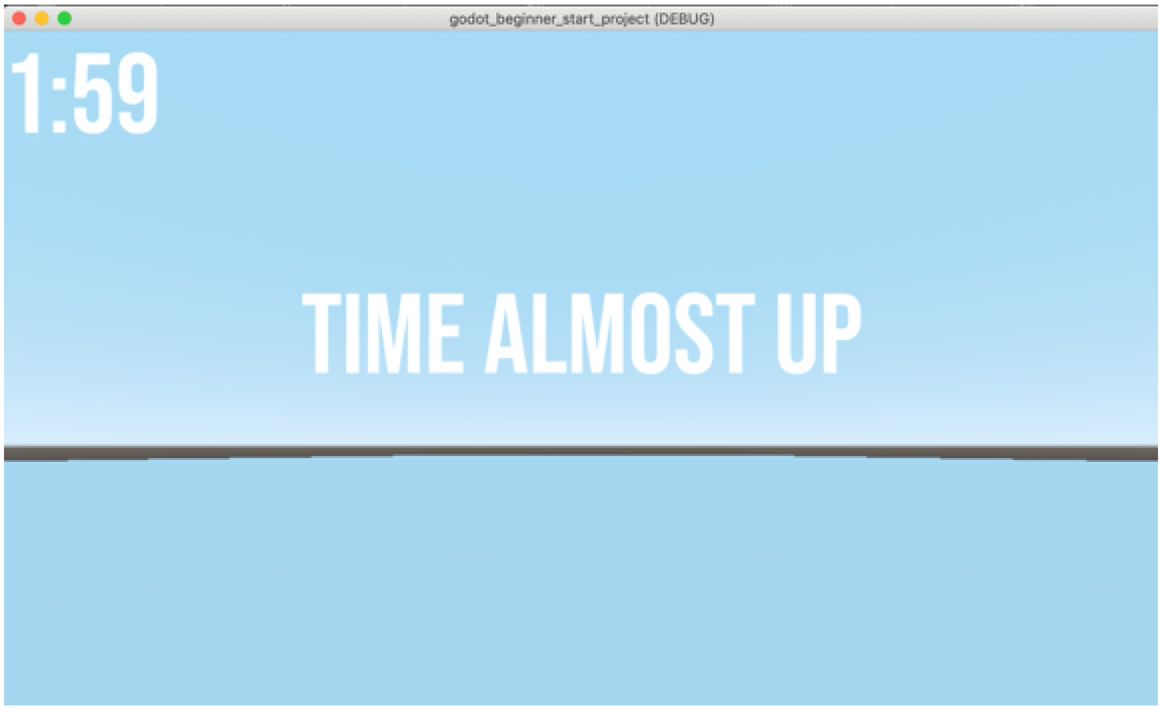
```
user_message_ui.set_text("")
```

- As for the previous statements, in the previous code we declare the variable **userMessageUI**, that will be linked to the node **userMessageUI**, and we then initialize its text to an empty string.

- In the script **Timer**, add the following code to the **_process** function (the new code is highlighted in bold) so that a message is displayed a few seconds before the time is up.

```
timer_ui.set_text("%d:%d" %[minutes,seconds])
if (counter >118) :
user_message_ui.set_text("Time Almost Up")
if (counter > 120):
print("Time Up!")
```

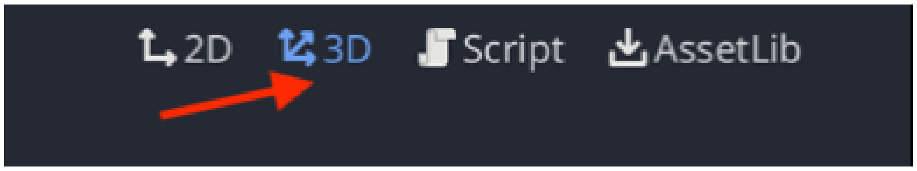
Save your script and play the scene. After a few seconds, the message should be displayed as illustrated on the next screenshot.



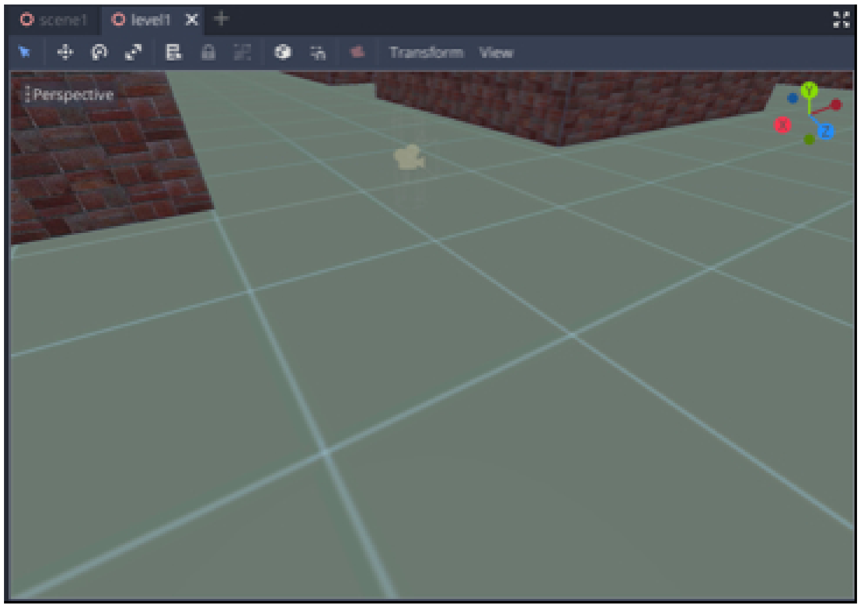
Collecting boxes and displaying messages accordingly

At this stage we will add boxes to our level (i.e., **level1**), collect them, and display messages accordingly onscreen.

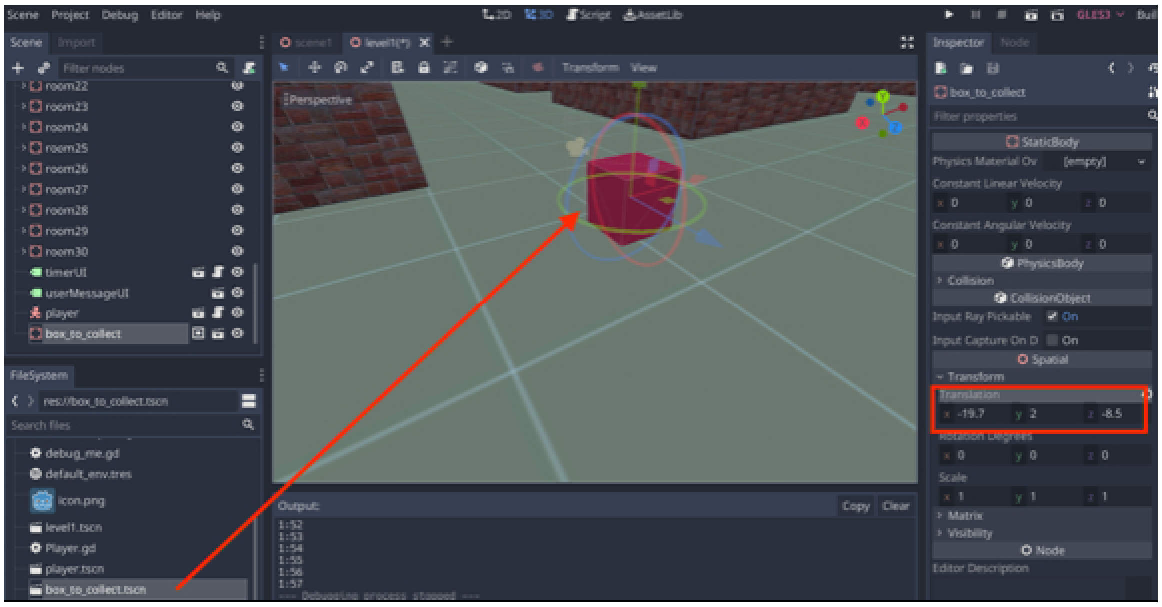
- Please switch back to the 3D workspace by clicking on the corresponding button in the top menu.



- In your current scene, move the view closer to the **player** node.

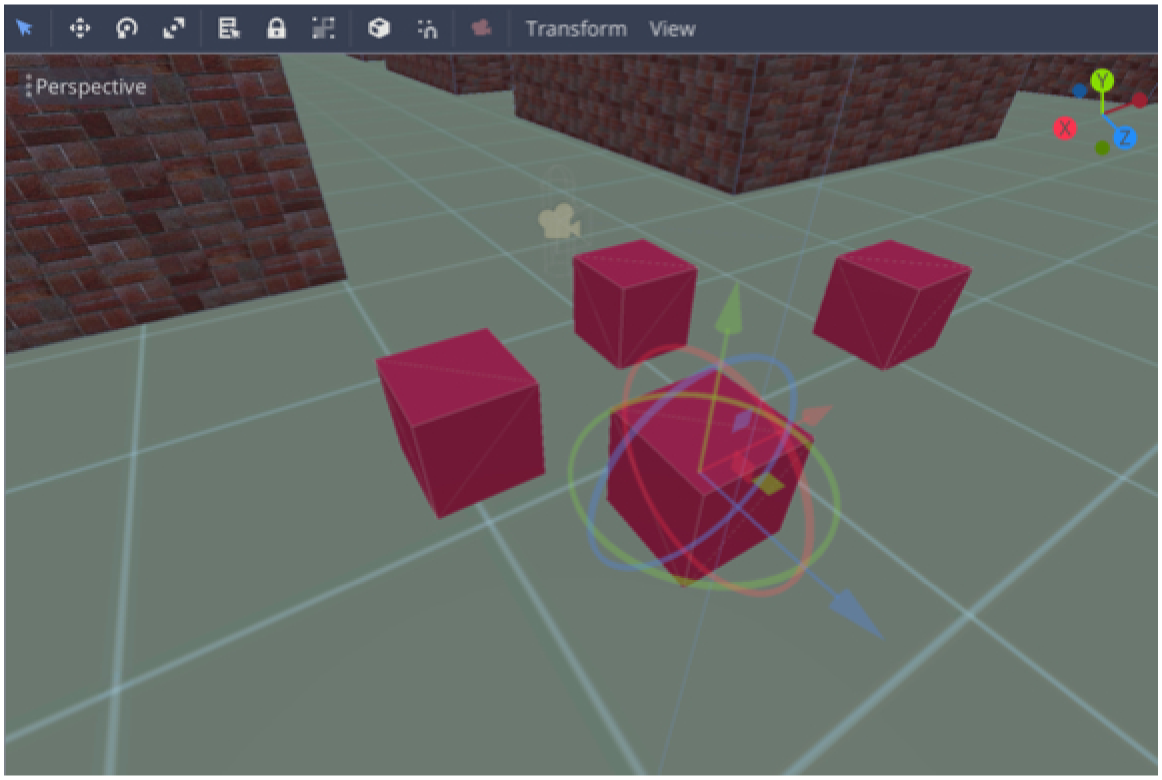


- Select the node called Spatial in the **Scene** tree, so that the new assets added to the scene become children of that node.
- Add four boxes close to the **player** node by dragging and dropping the file **box_to_collect** from the **FileSystem** tab to the viewport, and then by duplicating it three times, ensuring that the **y** coordinate of all four cubes is **2**.



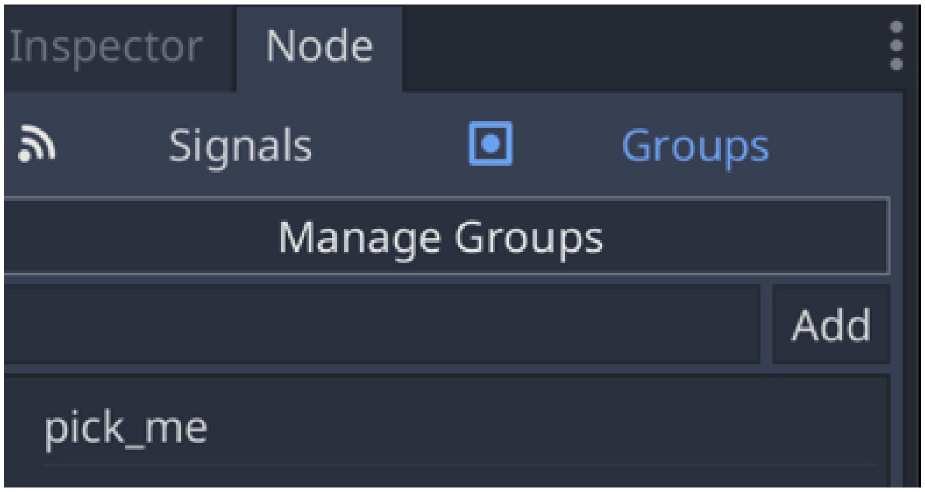
Translation: x -19.7 y 2 z -8.5

- Rename these boxes **box_to_collect1**, **box_to_collect2**, **box_to_collect3,** and **box_to_collect4**.



- For each of these boxes, you can add a texture or a color of your choice (some textures are available in the resource pack).

- By default, all these boxes should already have a tag/group called **pick_me**, and you can check it using the **Inspector**.

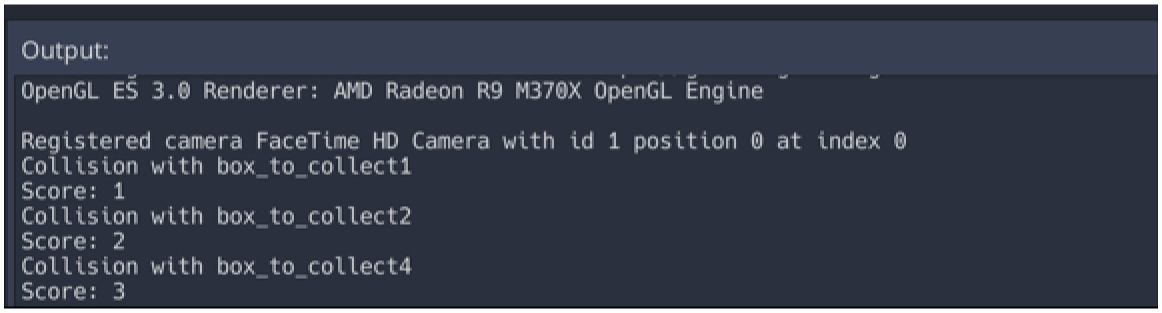


You may temporarily disable the **ceiling** object (if this is not already done), so that you can see your scene more clearly from above.

Also, you may notice that the player node is already linked to the script **Player.gd**; this is because the scene called **player.tscn** included a node that was already linked to the script **Player.gd**; since then we have updated the script and an updated version is now attached to the **player** node.

So, at this stage the boxes have been created and we are pretty much ok to go.

Please, play the scene and check that you can collect the boxes. No onscreen message should be displayed yet, as we will add this functionality in the next section. You should also notice messages from the new script, in the **Output** window, that provide information on the score and the objects collected, as illustrated in the next figure.



```
Output:
OpenGL ES 3.0 Renderer: AMD Radeon R9 M370X OpenGL Engine

Registered camera FaceTime HD Camera with id 1 position 0 at index 0
Collision with box_to_collect1
Score: 1
Collision with box_to_collect2
Score: 2
Collision with box_to_collect4
Score: 3
```

At this stage, it would be great to display more information on the boxes collected. So we will modify our code accordingly.

- Please switch to the code editor.
- Open the script called **Timer**.
- In that script, please comment all the code that displays messages in the **Output** window (i.e., the **print** statements).

```
#print ("TIME UP");
```

We will now modify the script called **Player.gd**, which is attached to the node **Player** and that deals with collisions.

- Please open the script **Player.gd**.
- Add the following code just before the function **_ready**.

```
onready var user_message_ui:Label = get_node("../userMessageUI")
```

- Add this code to the function **_ready**.

```
user_message_ui.set_text("");
```

- Finally, modify the function **_physics_process** as follows (new code in bold):

```
for index in get_slide_count():
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("pick_me")):
print("Collision with " + collision.collider.name)
collision.collider.queue_free()
user_message_ui.set_text("Box Collected");
```

- In the previous code (in bold), we just display the message **“You collected a box!”** onscreen, through the **node** named **user_message_ui**.
- Save your code and play the scene.
- Whenever you collide with an object, a message should be displayed on-screen.

If the onscreen message is truncated, you may need to decrease the font size

used for the node **userMessageUI** or to increase the size of the node.

So, at this stage we can collect boxes and we just need to make sure that after collecting four boxes the player is moved to the scene called **level2**. Whereas in the previous section we used to load a new scene, we will proceed differently here: we will:

Stay in the current scene.

- Keep the player node and the user interface.
- Remove the nodes that we no longer need (i.e., the maze).
- Add a new node for the new environment.

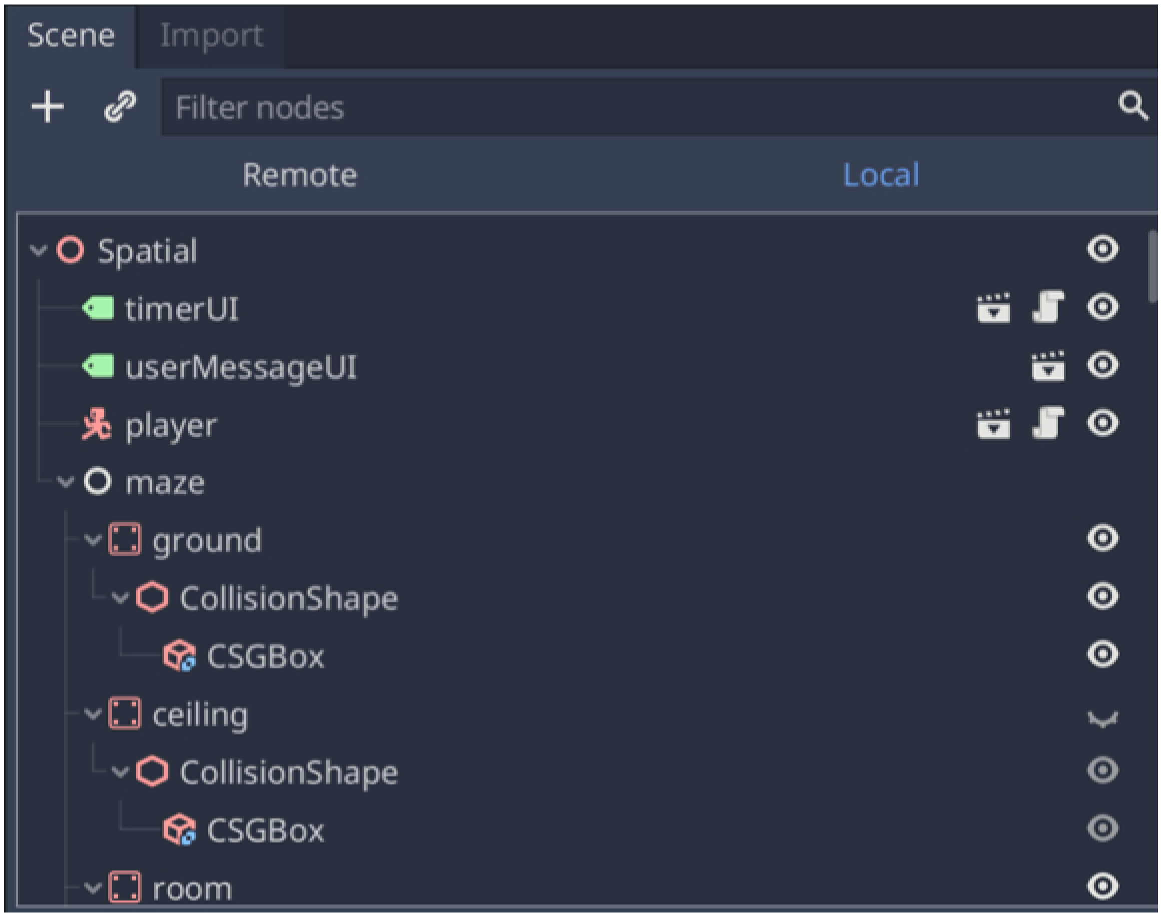
This is because we have created a user interface for our game with a timer and messages for the player and it would be great to keep these across levels. Loading a new scene would mean losing these nodes, hence the choice to only load what we need in the current scene and remove what is no longer needed.

So, what we will do here is to:

- Remove all the objects from the current scene (e.g., the maze and the boxes).
- Keep the player and the user interface (and keep the score as well).
- Load the content of the new scene in the current scene.

So first let's group all the elemental nodes from the current scene:

- Please create a new node of type **Node** as a child of the **Spatial** node (you will need to use an uppercase **N** for **Node** in the search window).
- Rename this node **maze**.
- Please select all the walls and boxes in the scene and drag and drop them on the node called **maze** so that they become children of this node, as illustrated in the next figure.



At this stage all the environmental objects from the maze are grouped and are children of the node called maze. So the next step is to test whether we have collected 3 boxes, and to successively unload the first level by removing these objects, and then load the second level by loading the corresponding node. So let's proceed:

- Please switch to the **Script** workspace.
- Open the script **Player.gd**, if it is not already open.
- Modify the following code in the function **_physics_process** (new code in bold).

```
score += 1
print("Score: "+str(score))
if (score == 3):
get_node("../maze").queue_free()
```

```
var new_scene = load("res://level2.tscn").instance()
get_parent().add_child(new_scene)
get_node("../timer").counter = 0;
```

In the previous code:

- We test whether the score is **3**, and if that's the case we perform several actions.
- First we remove the node called **maze** by using the function **queue_free**. Since all the walls and boxes are children of this note, these will also be removed.
- We then create a new node called **new_scene** and initialize it with the content of the asset (or scene) called **level2** that is present in the **FileSystem**.
- We add this node as a child of the root node of our scene, that is, the node called **Spatial**.
- Finally, we reset the counter of the script **Timer** (that is attached to the node **timer**) to 0.

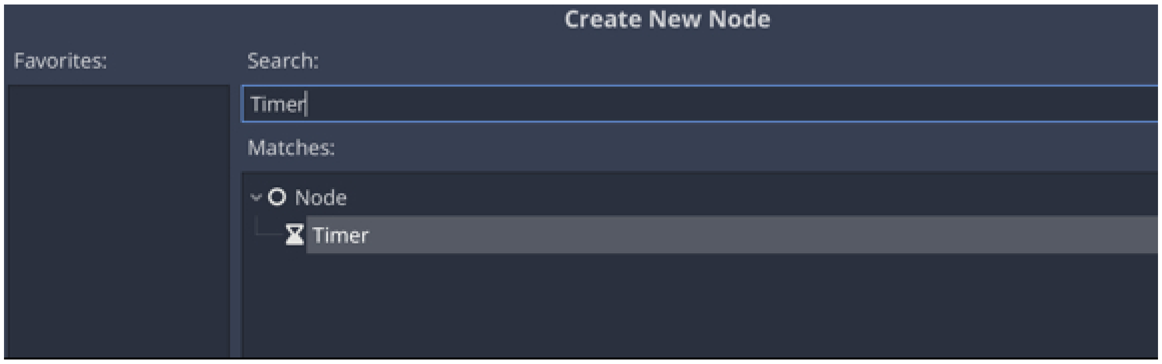
That's it. You can now save your script and play the scene; after collecting 3 boxes, you should see that the player moves on to the next level. Hurray!

Deleting the user messages after a few seconds

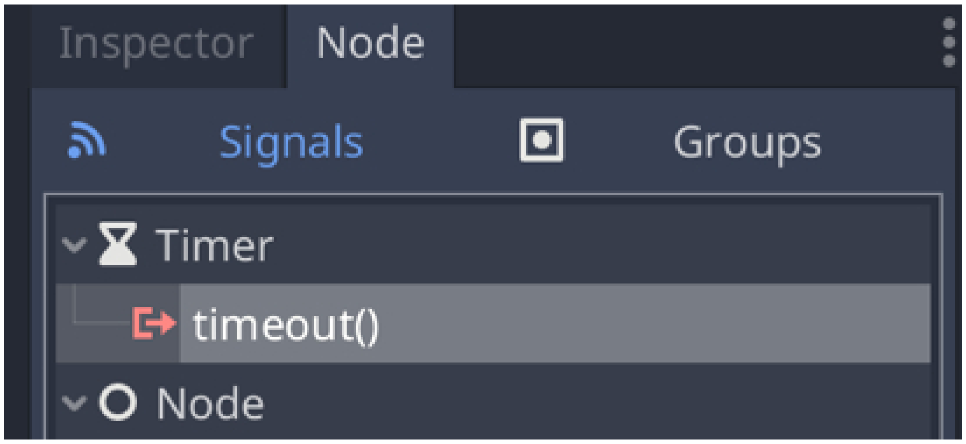
While we have managed to display onscreen messages as well as a timer, it would be great if we could delete the message displayed to the user after a few seconds, so that the screen remains clear afterwards. To do so, we will use another type of node called a **Timer** which basically performs actions at regular intervals.

First let's create a timer:

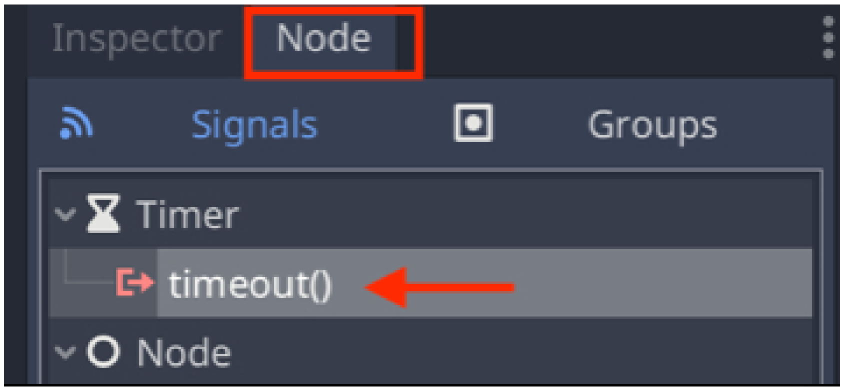
- Please select the node called **Spatial**.
- Add a child of type **Timer** to this node, this should create a new node called **Timer** in the **Scene** tree.



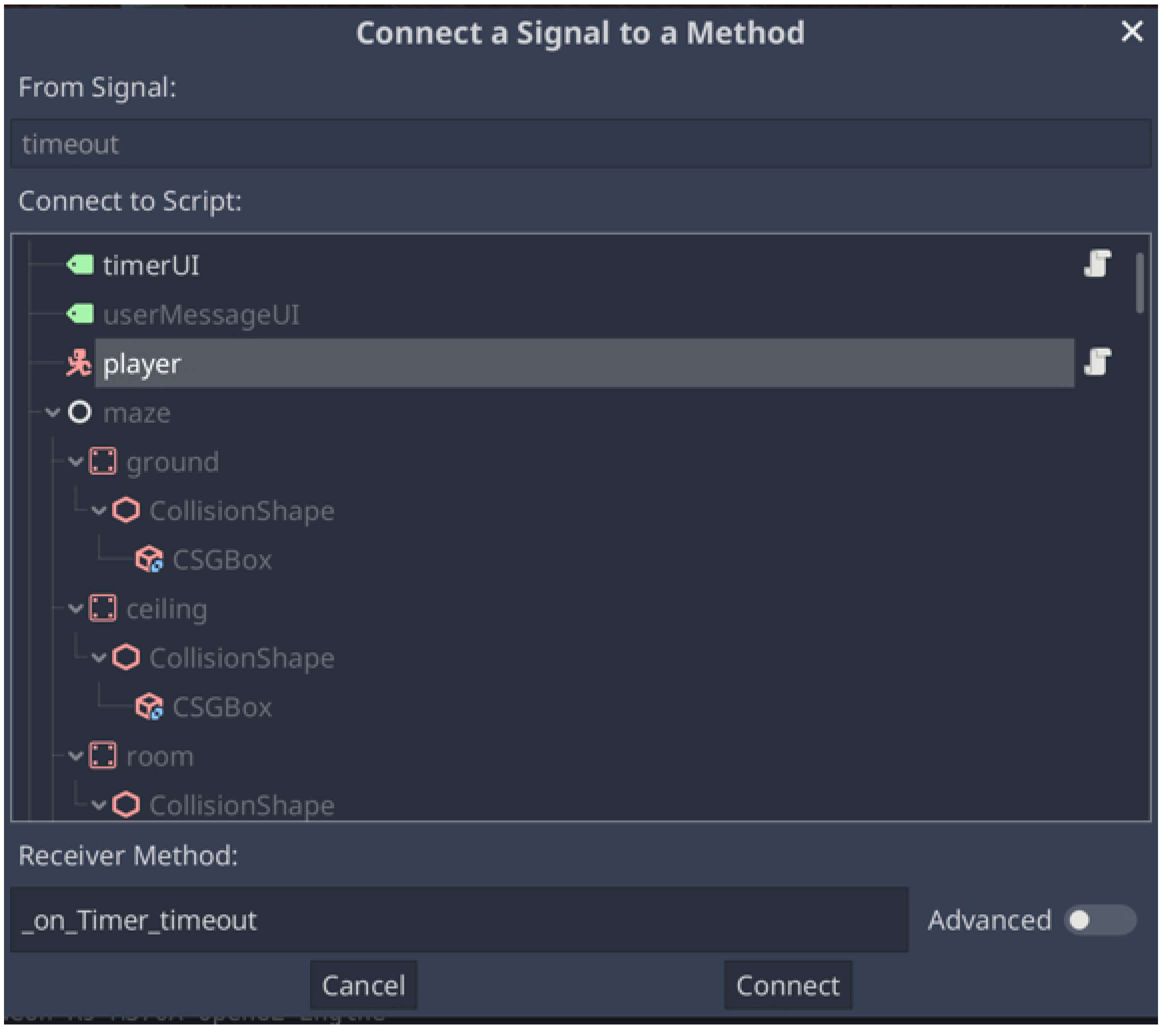
- Once this is done, select the node **Timer** in the **Scene** tree.
- In the **Inspector**, click on the tab called **Node** and then **Signals**.



- You can then double-click on the option (or event) **called timeout()**. This means that we will get to choose what should be done whenever our timer times out.



- Once this is done, a new window will appear, and it will make it possible to select the object and the function that will be called when the timer times out. As you will see in the next figure, it is possible to choose any of the nodes already present in the scene.
- You may also notice a section called **Receiver Method**: this refers to the function (on the selected node) that will be called whenever the timer times out.


Connect a Signal to a Method

From Signal:

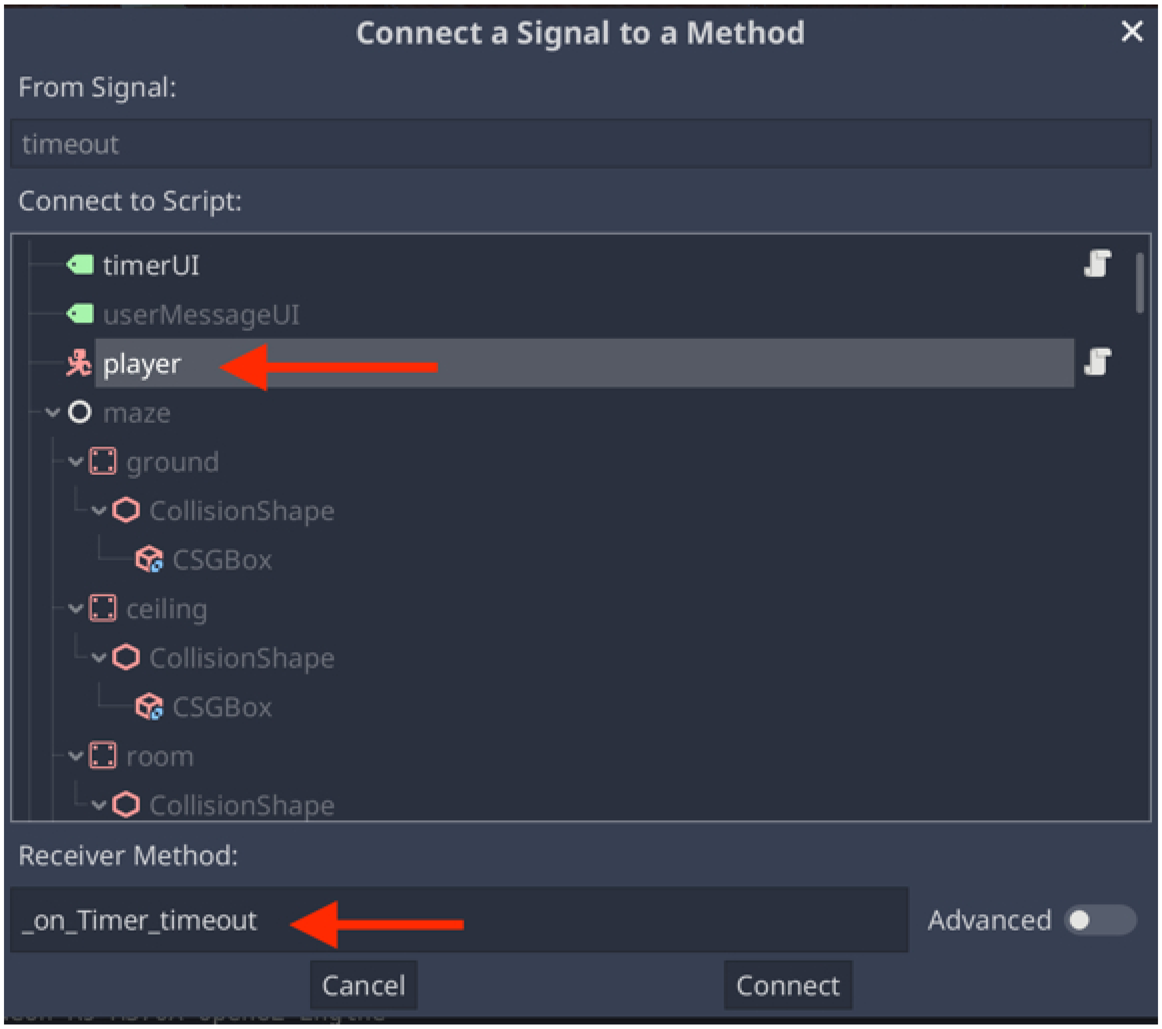
timeout

Connect to Script:

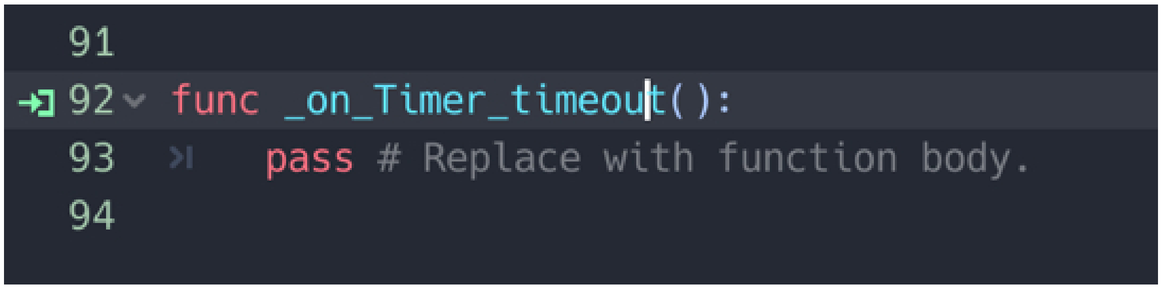
Receiver Method:

_on_Timer_timeout

- So please select (or click on) the node called **player**, leave the other options as default (i.e., **Receiver Method**), and press **Connect**. This means that whenever the timer times out, the function called **on_Timer_timeout** that will be added in a script linked to the node **player (**i.e., **Player.gd)** will be called.



- Once you click on the button "**Connect**", Godot will switch to the script **Player.gd** that is currently linked to the **player** node; you will also notice that a new function called **on_Timer_timeout** has been created, as per the next figure.



```
func _on_Timer_timeout():
	pass # Replace with function body.
```

So at this stage, we have linked the timer to the script **Player.gd**; now, we need to specify that:

- The timer whenever we display a message onscreen.
- The timer times out after 2 seconds.
- When the timer times out, the onscreen message should be deleted.

So let's add these modifications:

- Please add this line just before the function **_ready** in the script **Player.gd**.

```
onready var timer:Timer = get_node("../Timer")
```

In the previous code, we declare the variable **timer** and link it to the node called **Timer** that we have just created.

- Add this code to the function **_ready**.

```
timer.set_wait_time(2)
```

In the previous code we set the duration of the timer to 2 seconds.

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

```
if (collision.collider.is_in_group("pick_me")):
print("Collision with " + collision.collider.name)
collision.collider.queue_free()
user_message_ui.set_text("Box Collected");
timer.start()
score += 1
```

In the previous code we just start the timer.

- Finally, modify the function **_on_Timer_timeout** as follows (new code in bold):

```
func _on_Timer_timeout():
user_message_ui.set_text("")
timer.stop()
```

In the previous code, whenever the timer times out, we delete the onscreen message and stop the timer.

Please save your code and play the scene to see how the code works. You should see that the message displayed (e.g., after picking up a box) disappears after 2 seconds.

DISPLAYING A DYNAMIC MESSAGE FOR THE BOXES COLLECTED

Once this has been checked, we could now modify this script so that the number of boxes collected is also displayed onscreen.

- Please modify the function **_physics_process** in the script **Player.gd** as follows (new code in bold):

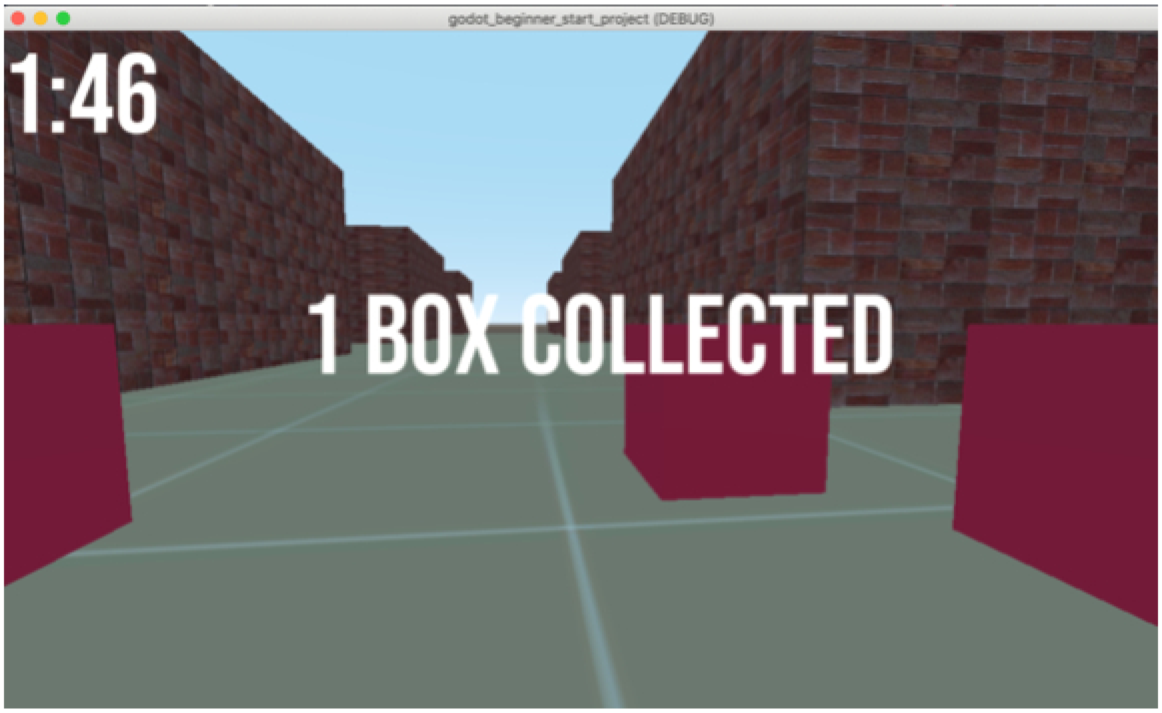
```
timer.start()
score += 1
if (score == 1):
user_message_ui.set_text(str(score) + " Box Collected")
else:
user_message_ui.set_text(str(score) + " Boxes Collected")
```

In the previous code:

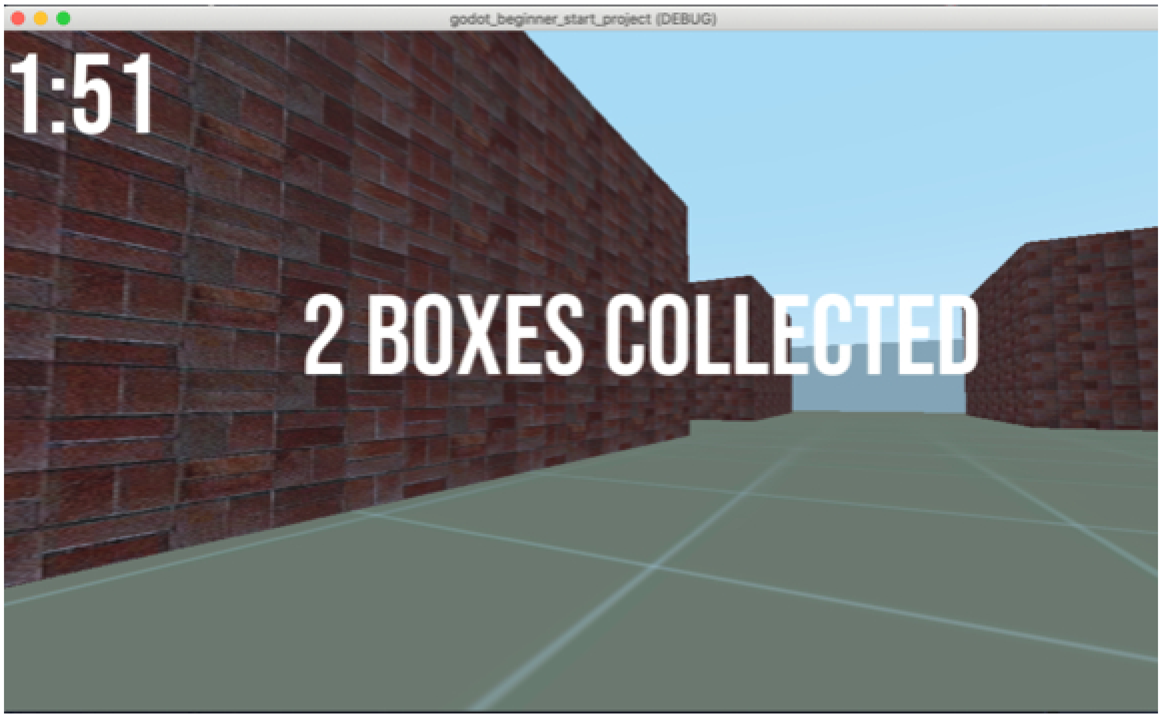
- We display the score as part of the onscreen message.
- We ensure that this is done after the score has been increased.
- If the score is 1 then the message will use a singular version of the word box (i.e., “**Box**”).
- Otherwise we will use a plural version of the word box (i.e., “**Boxes**”).

That’s it. Please save your code and play the scene to see how the code works.

- As you collect the first box, you should see that the message says “**1 Box Collected**”



- As you collect your second box, you should see that the message says "**2 Boxes Collected**".

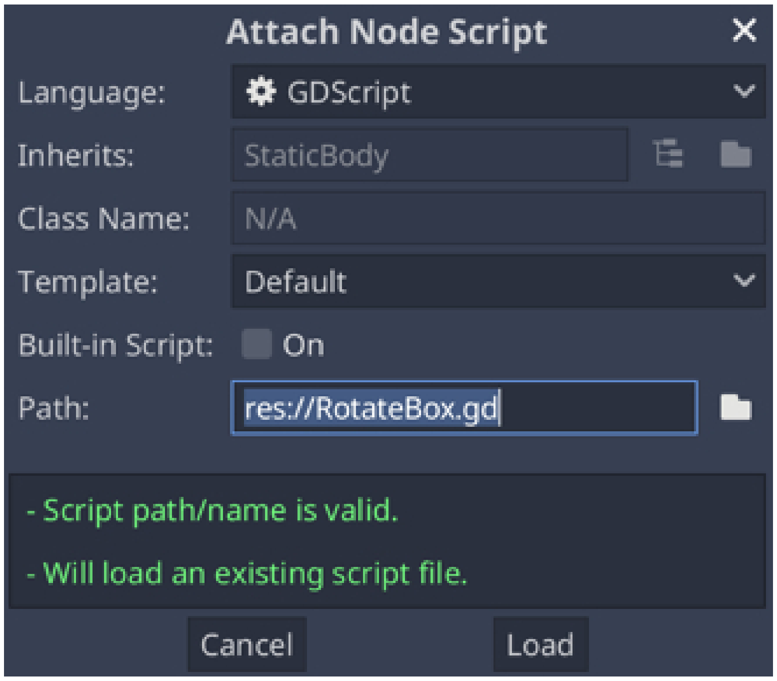


Rotating objects to be picked up

We will now add a mechanism that helps to make the objects to be collected more obvious to the player. We will get them to rotate continuously using a simple script.

First, we will create a new script and link it to the boxes to collect.

- Please right-click on the node **box_to_collect_1** (**or box_to_collect** depending on whether you have renamed the first box **box_to_collect1**).
- Select **AttachNode Script** from the contextual menu.
- In the new window, type **res://RotateBox.gd** for the path attributes, and click on create.



Language: GDScript
Inherits: StaticBody
Class Name: N/A
Template: Default
Built-in Script: On
Path: res://RotateBox.gd

- Script path/name is valid.
- Will load an existing script file.

- This will create a new script called **RotateBox** and attach it to the box.

Once the script is open we can modify it:

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

```
func _process(delta):
rotation_degrees.y +=90*delta
```

In the previous code, we rotate the box at the rate of 90 degrees per second around the y axis (the variable delta here ensures that the rotational speed does not depend on the computer's frame rate).

You can now save your code, and attach the script **RotateBox** to the other boxes, **box_to_collect2**, **box_to_collect3**, and **box_to_collect4** (i.e., right-click on the node, select the option **Attach Script**, use the path **"res://RotateBox.gd",** and click on **Load**).

Once this is done, you can play the scene and you should see that all four boxes are rotating at the same speed.

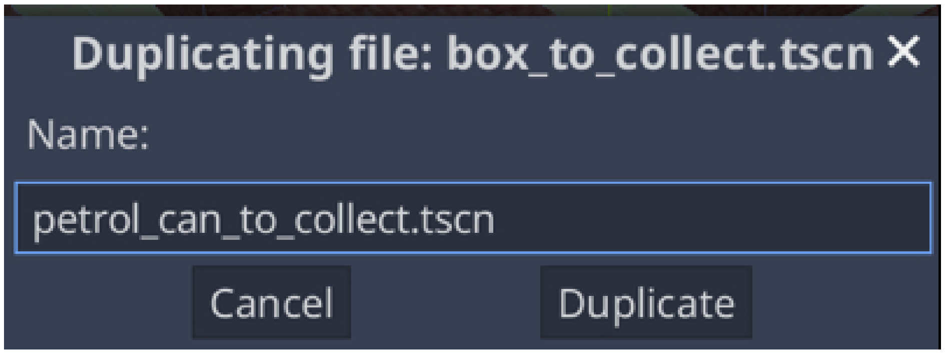
Collecting petrol cans in the second level

At this stage of our game, we have created and managed the conditions to transition from the maze to the second level. We now need to build the logic of the game, so that the player needs to collect three or four petrol cans before s/he can access and pilot the plane to escape the island.

- Please save your current scene (**Scene | Save Scene** or **CTRL + S**).
- Open the **scene** called **level2**.

We will now create several boxes that will represent the petrol cans to be collected; to do so, we will reuse and modify the file **box_to_collect.tscn** that we have already created and that is available in the **FileSystem** window.

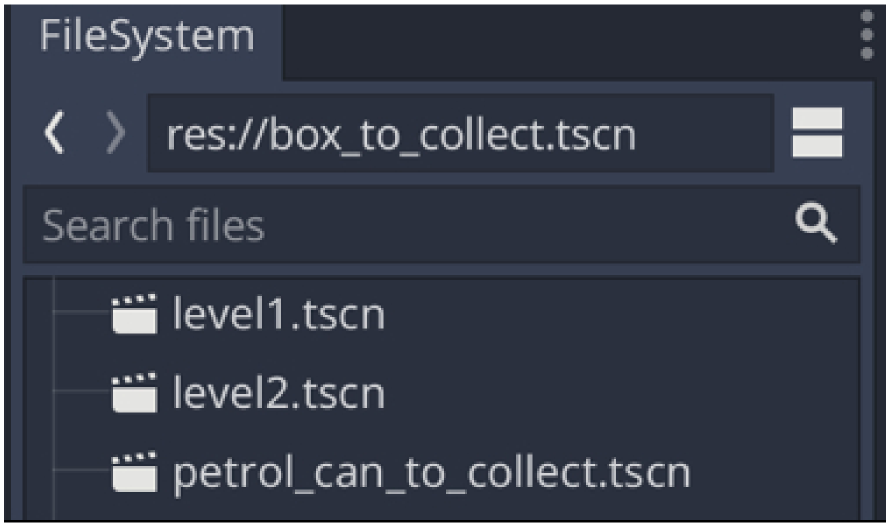
- Please select the file **box_to_collect.tscn** in the **FileSystem** window.
- Duplicate this file by pressing **CTRL + D** or by right-clicking on the file and then selecting the options **Duplicate** in the contextual menu.
- In the new window, type **petrol_can_to_collect**, and press **Duplicate**, as per the next figure.


Duplicating file: box_to_collect.tscn

Name:

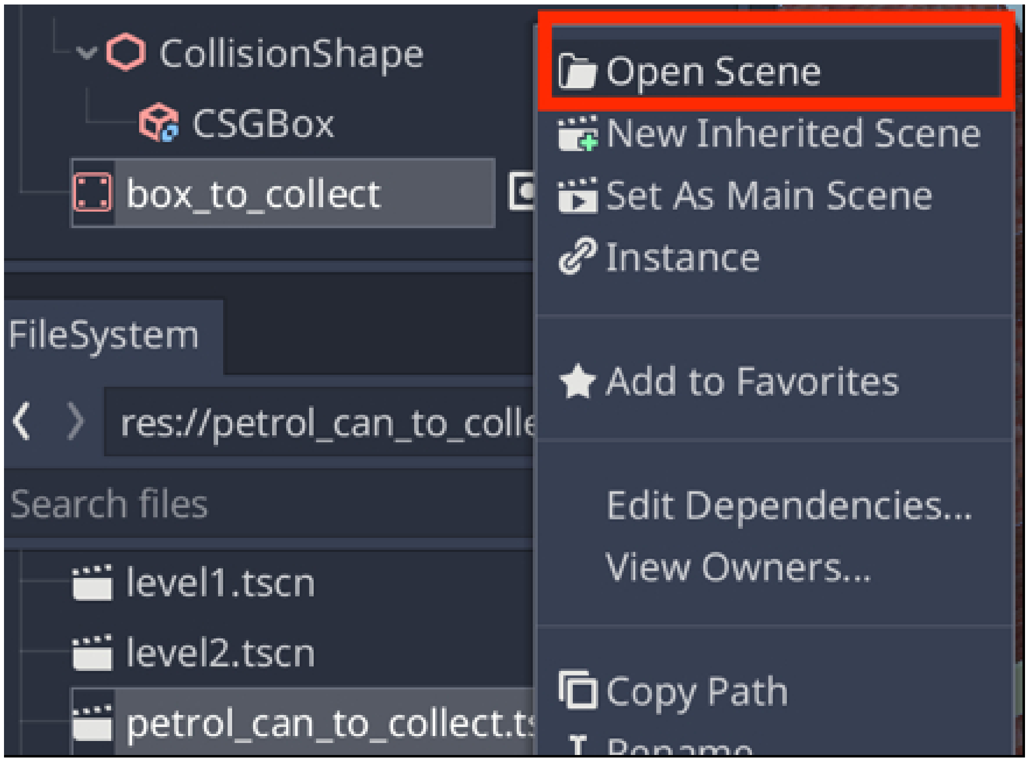
petrol_can_to_collect.tscn

Once this is done, a new file called **petrol_can_to_collect.tscn** should appear in the **FileSystem**.



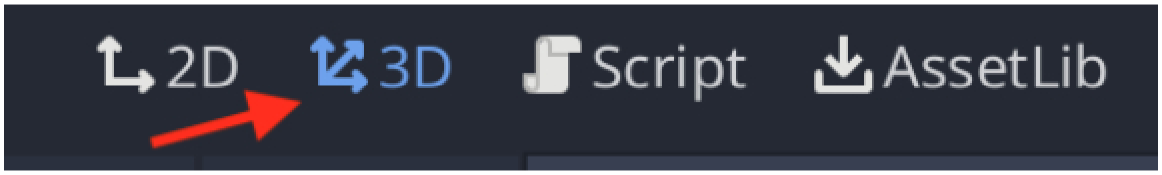
At this stage, we just need to modify this file to use a different group for this object.

- Please right-click on this file (i.e., **petrol_can_to_colled**) and select the option "**Open Scene**" from the contextual menu.

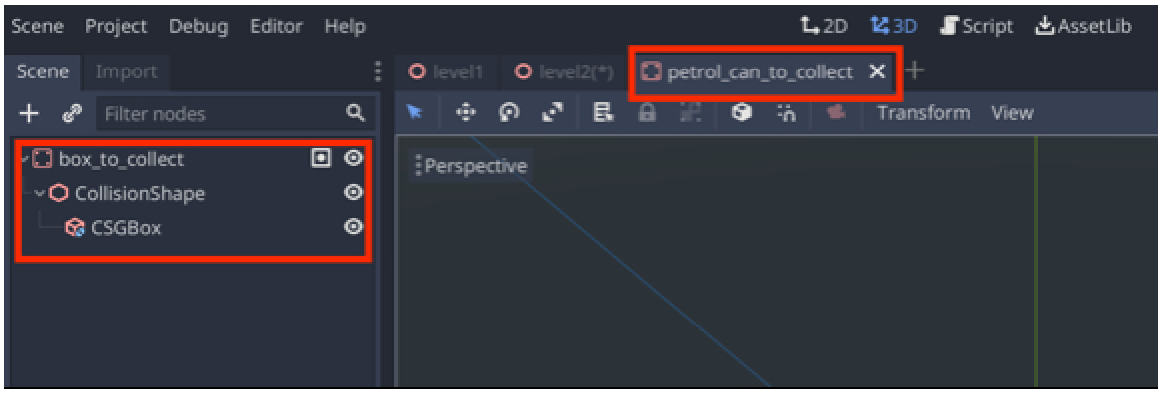


At this stage, you should see that a new scene is open and we will be able to modify it, including the box that will be used to represent the petrol can, along with its properties.

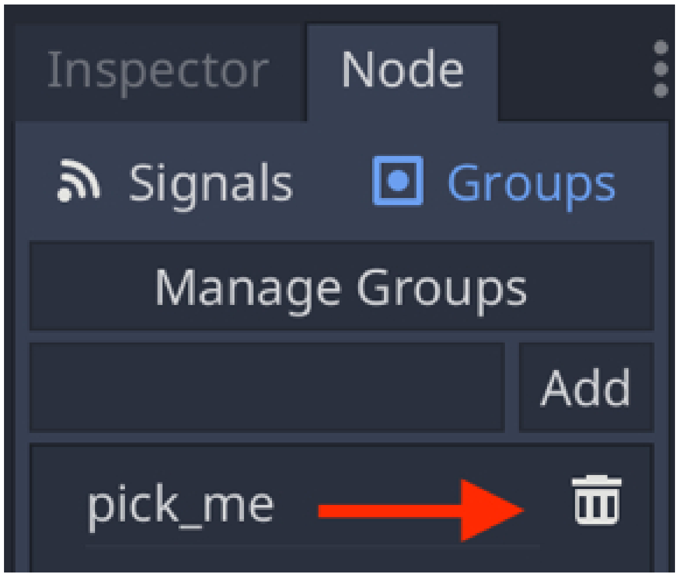
- Please switch to the 3D workspace.



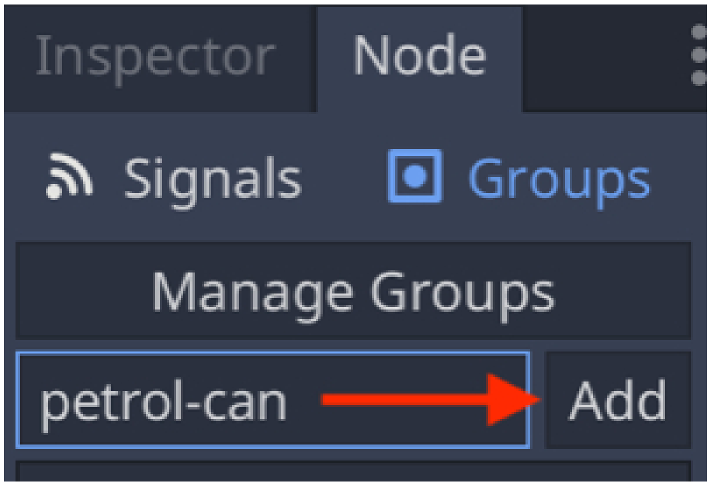
- You will notice three nodes in the **Scene** view.



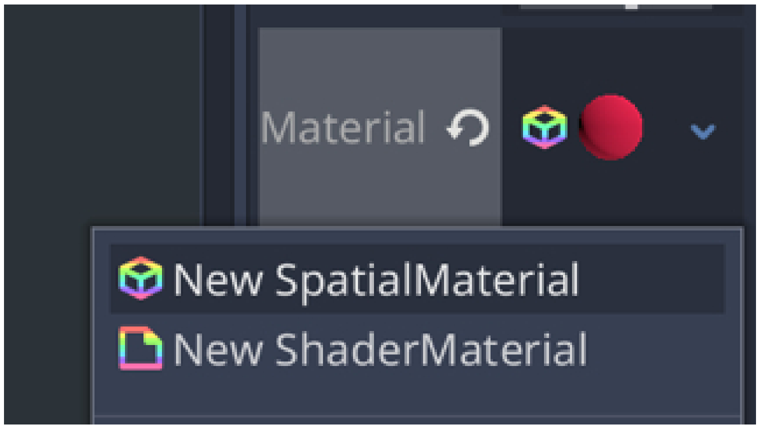
- Please select the node called **box_to_collect** in the **Scene** tab, and rename it **petrol_can_to_collect**.
- Using the **Inspector**, click on the tab **Node | Groups** and delete the group **pick_me** using the icon to the right of the label **pick_me**.



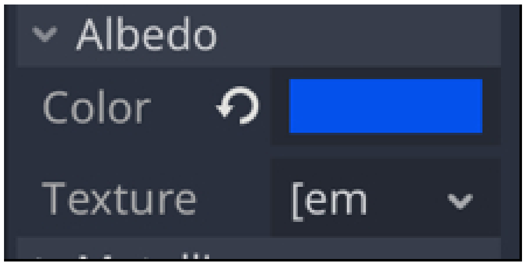
- Add a new group called **petrol-can**: type "**petrol-can**" in the text field and click on **Add**.



- Finally, click on the node called **CSGBox**, then in the **Inspector** tab create a new **Spatial Material**.

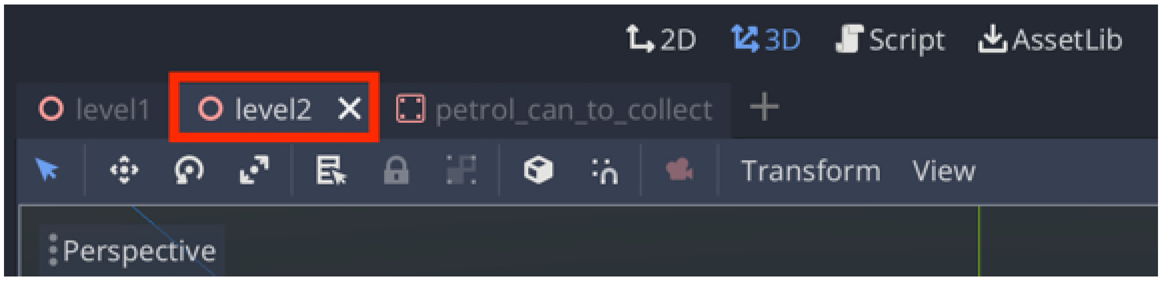


- Modify its **Albedo** feature by selecting a new blue color.

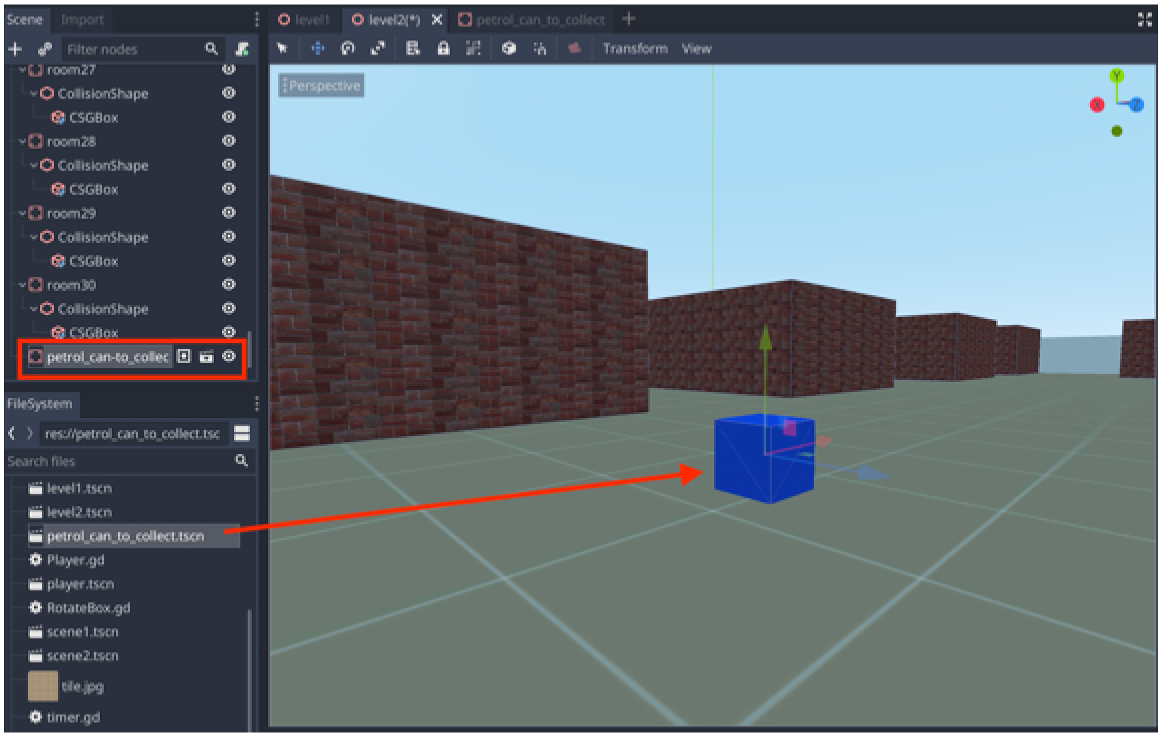


At this stage, we have modified the color and the tag of the file called **petrol_-can_ to_collect** and we will be able to use it to create new petrol cans to be added to the second level.

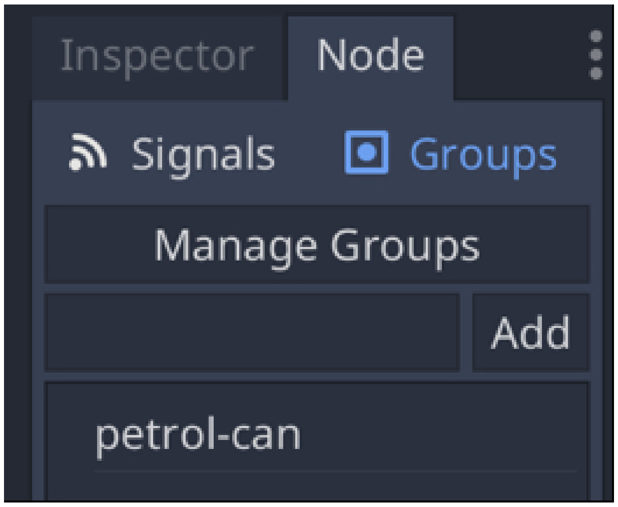
- Please save the file called **petrol_can_to_collect**.
- Switch to the scene called **level2** using the corresponding tab.



- Select the node called **Spatial**, so that new assets added to the scene are children of this node.
- Drag and drop the file **petrol_can_to_collect** from the **FileSystem** to the viewport. This will create a new node called **petrol_can_to_collect.**



- Select this object, and check that its y coordinate is **2**. You can also check that its group is **petrol-can**.



- Please repeat the previous steps (or duplicate the object **petrol_can_to_collect**) to create two new petrol cans.
- Rename these objects **petrol_can1**, **petrol_can2**, and **petrol_can3**.

At this stage we have created and tagged all the petrol cans. We just need to modify our object-collection script so that we can detect and collect the cans.

- Please open the script used in the previous section (**Player.gd**).
- Add the following code to the start of the file (new code in bold).

extends KinematicBody

var score:int = 0

var nb_petrol_can :int = 0

We will then add more code to manage the collision with the petrol cans.

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

if (collision.collider.is_in_group("pick_me") ||collision.collider.is_in_group("petrol-can")):

print("Collision with " + collision.collider.name)

collision.collider.queue_free()

timer.start()

score += 1

```
if (collision.collider.is_in_group("petrol-can")):
nb_petrol_can+=1
if (nb_petrol_can == 1):
user_message_ui.set_text(str(nb_petrol_can) + " Can Collected");
else:
user_message_ui.set_text(str(nb_petrol_can) + " Cans Collected");
else:
if (score == 1): user_message_ui.set_text(str(score) + " Box Collected");
else:  user_message_ui.set_text(str(score) + " Boxes Collected");
```

In the previous code:

- We check whether we have collided with an object that belongs to the group **petrol-can**.
- We then display different messages based on whether we have collected a can or a box.
- In case we collect a can we also increase the value of the variable **nb_petrol_can**.

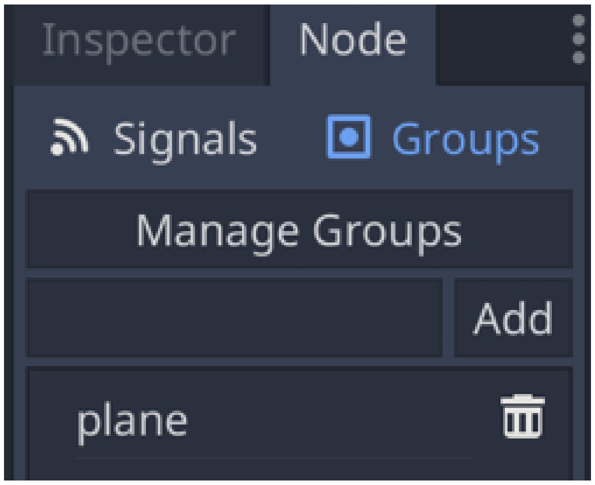
Please save your code and play the scene **level1**; check that the messages are displayed correctly even after progressing to the second level. As you go through the second level, you may also check that you can collect petrol cans, and that you can collide with the spacecraft that is currently in the scene.

At this stage, we can collect petrol cans and we would like to activate the aircraft (or make it available) whenever we have collected a sufficient number of petrol cans. So we need to check that we have collected enough petrol cans before colliding with (and trying to pilot) the aircraft.

If you look at the **Scene** tree, you may notice a node called **plane**; this node is the node for the plane that the player will need to pilot to escape the level; to do so, the player will need to collect enough petrol cans.

So at this stage we need to check the number of cans collected.

- Please select the node called **plane** in the hierarchy, and check that it has a group called **plane** already allocated.



- Please modify the function **physics_process** in the script **Player** as follows (new code in bold):

if (collision.collider.is_in_group("pick_me") ||collision.collider.is_in_group("petrol- can") **||collision.collider.is_in_group("plane")):**

print("Collision with " + collision.collider.name)

if (!collision.collider.is_in_group("plane")):

collision.collider.queue_free()

timer.start()

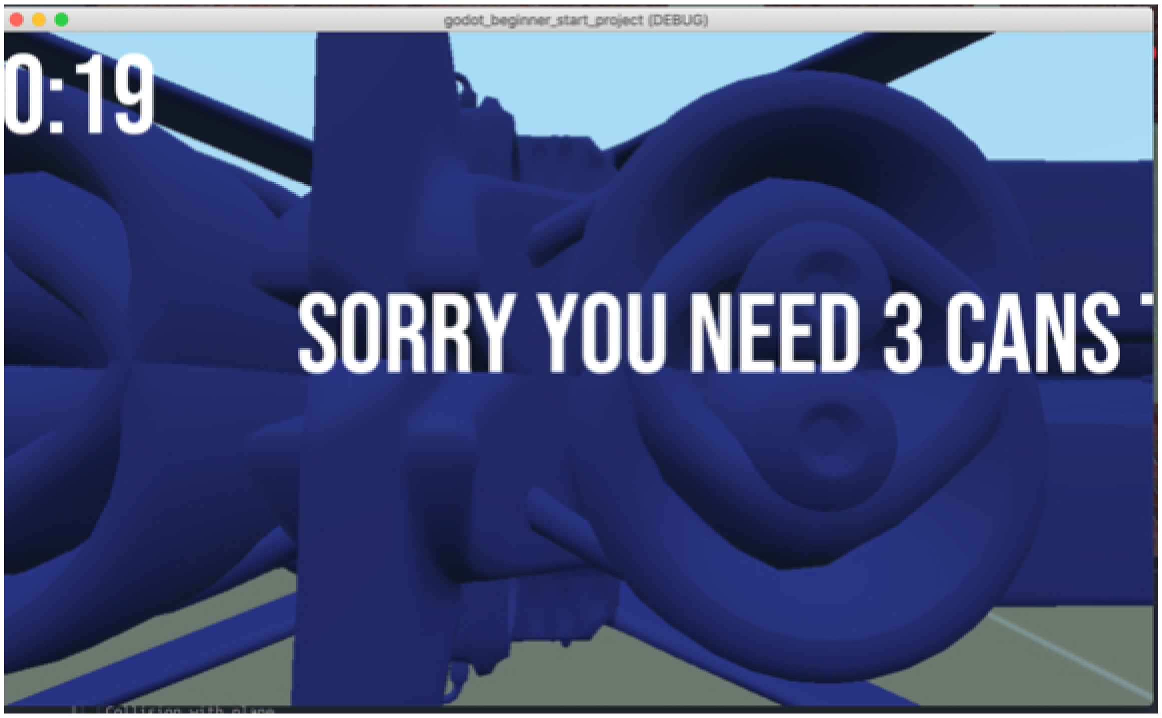
```
score += 1
if (collision.collider.is_in_group("petrol-can")):
nb_petrol_can+=1
if (nb_petrol_can == 1):
user_message_ui.set_text(str(nb_petrol_can) + " Can Collected");
else:  user_message_ui.set_text(str(nb_petrol_can) + " Cans Collected");
elif (collision.collider.is_in_group("plane")):
if (nb_petrol_can <3) :
user_message_ui.set_text("Sorry you need 3 cans to fly the plane");
else:
user_message_ui.set_text("Congratulations, you can fly the plane");
else:
if (score == 1): user_message_ui.set_text(str(score) + " Box Collected");
else:  user_message_ui.set_text(str(score) + " Boxes Collected");
```

In the previous code

- We check whether we are colliding with a plane.
- If that is the case, we check whether we have already collected enough petrol cans.
- If this is not the case, we display a message that says “**Sorry you need 3 cans to fly the plane**”.
- Otherwise, if we have collected enough cans, we display a message that says “**Congratulations, you can now fly the plane**”, we pause for 2 seconds, and we then load the end scene.

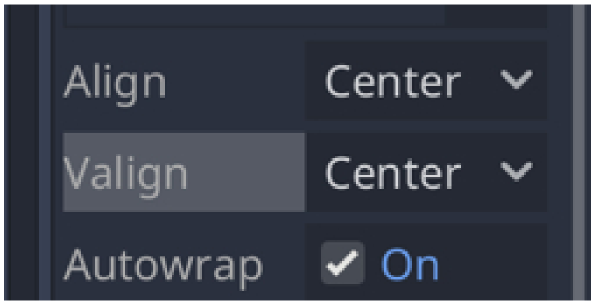
You can now save your code and test the scene, starting with **level1**.

- Collect three boxes in **level1**.
- Collect two cans and go near the plane, you should see the following message:



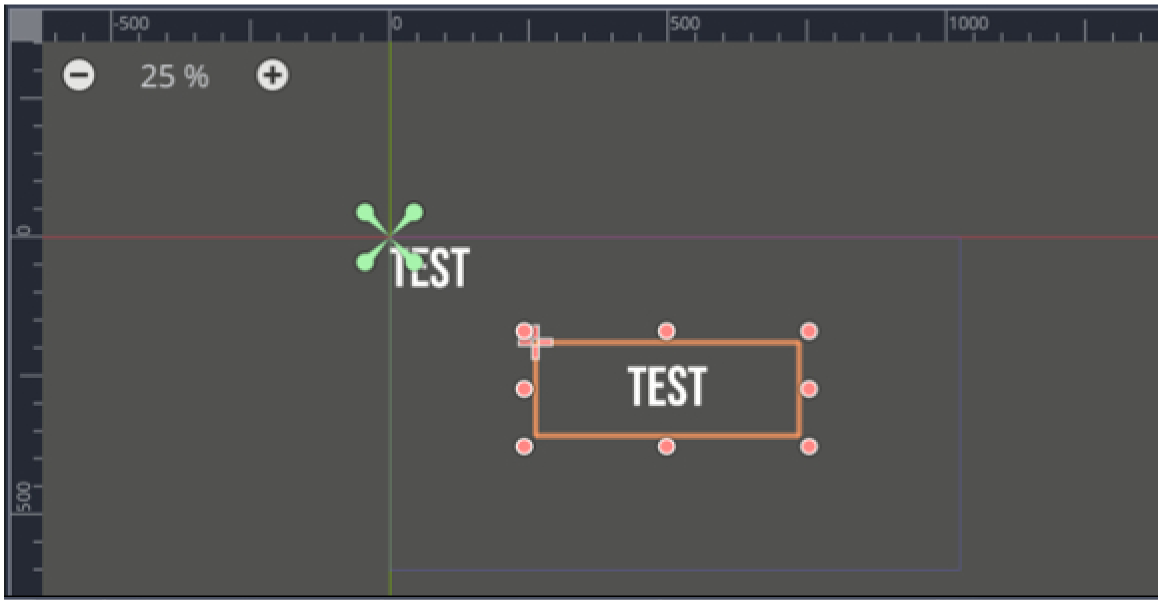
You may notice that some of the messages are truncated; this is because the label used to display these messages is possibly too small, so we could modify them as follows:

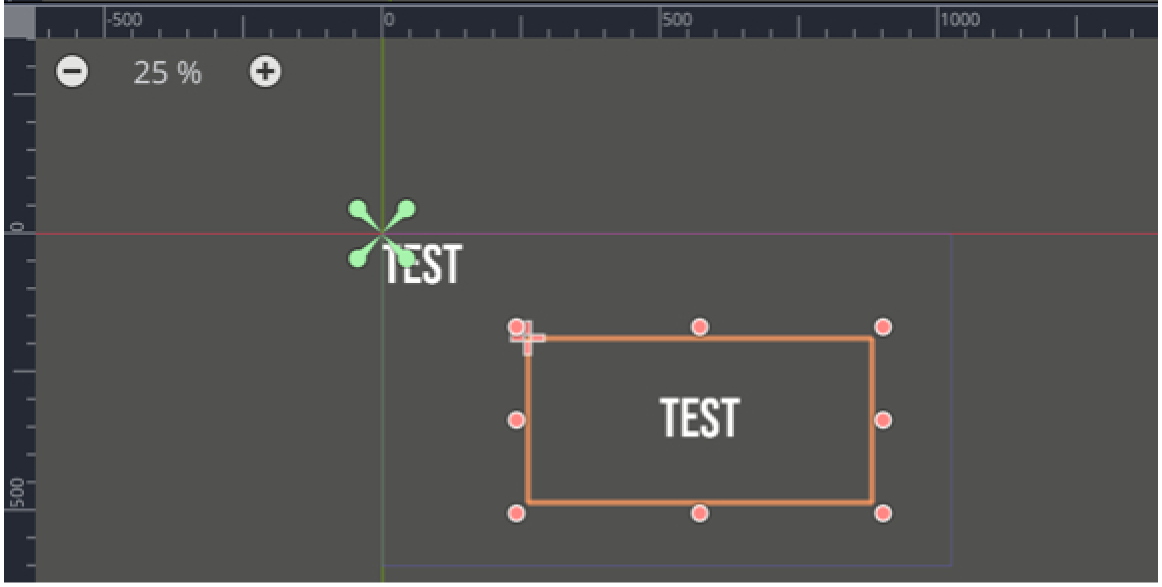
- Please select the node called **userMessageUI.**
- Using the **Inspector** locate the section called **Autowrap**.
- Set the option to **true**.



Align Center
Valign Center
Autowrap ✓ On

- Increase the size of the rectangle that defines the area covered by the label by dragging its handles.





If you replay the scene you should be able to see the full content of the messages. If you collect 3 cans, a new message should say that the player can fly the plane, and the next section will show you how you can create a new screen with more information on the player's achievements.

LEVEL ROUNDUP

In this chapter, we have further improved our skills to learn about how to create and update a user interface. We became more comfortable with the terms **label** and **group**. We managed to create scripts to rotate objects, display messages, and re-move messages after a few seconds. So, again, we have made considerable progress since the last chapter. Well done!

Checklist

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

- Create a **Label** node.
- Move a **Label node** to a specific location onscreen.
- Create a function.
- Use the **_ready** function to rotate an object.
- Access a **Label node** from a script and update its content.

Quiz

Please answer the following questions.

1. Please specify whether this statement is **TRUE** or **FALSE:**

A new label can be added using the node **Label**.

1. Please specify whether this statement is **TRUE** or **FALSE:**

The following code will empty the text node named **userMessageUI** provided that the variable **user_message_ui** has been linked to this node.

```
user_message_ui.set_text(““);
```

1. A **Label** node can be resized or moved.

2. Find the error in the following code.

```
onready var userMessageUI:Label = get_the_node("../userMessageUI")
```

1. Please specify whether this statement is **TRUE** or **FALSE:**

Any scene can be duplicated using the shortcut **CTRL + F**.

1. Please specify whether this statement is **TRUE** or **FALSE:**

If the scene **level4** has been created and saved as **level4.tscn**, it can then be loaded using the following code.

```
var new_scene = load("res://level4.tscn").instance()
get_parent().add_child(new_scene)
```

1. Please specify whether this statement is TRUE or FALSE: “The following code should execute with no errors...”

```
var score:int = 2
print("Score: "+str(score))
```

1. Please specify whether this statement is TRUE or FALSE: “The following

code should execute with no errors…"

```
var score:String = "2"
print("Score: "+score)
```

1. Please specify whether this statement is **TRUE** or **FALSE: "**This code will rotate the object linked to the script by 90 degrees every second**"**

```
func _process(delta):
rotation_degrees.y += 90*delta
```

1. Please specify whether this statement is **TRUE** or **FALSE: "**This code will rotate the object linked to the script by 90 degrees every second**"**

```
func _process(delta):
rotation_degrees.y += 90
```

Answers to the quiz

1. TRUE
2. TRUE
3. TRUE**:**
4. Find the error in the following code.

onready var userMessageUI:Label = **get_the_node**("../userMessageUI")

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

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let's put these to the test.

- Download a font of your choice from the site **http://www.dafont.com.**
- Unzip the file that you have downloaded; this should provide you with a **TTF** file.
- Import this file into Godot.
- Use the font for one of the **Label** nodes present in the scene (**timerUI** or **userMessageUI**).
- Experiment with other fonts.

Chapter 5: Polishing Our Game

In this section, we will finalize the structure and aspect of our game by adding a series of elements that will facilitate the user interaction, and that will also improve its appearance; these will include splash screens, background music, sound effects, and a mini-map.

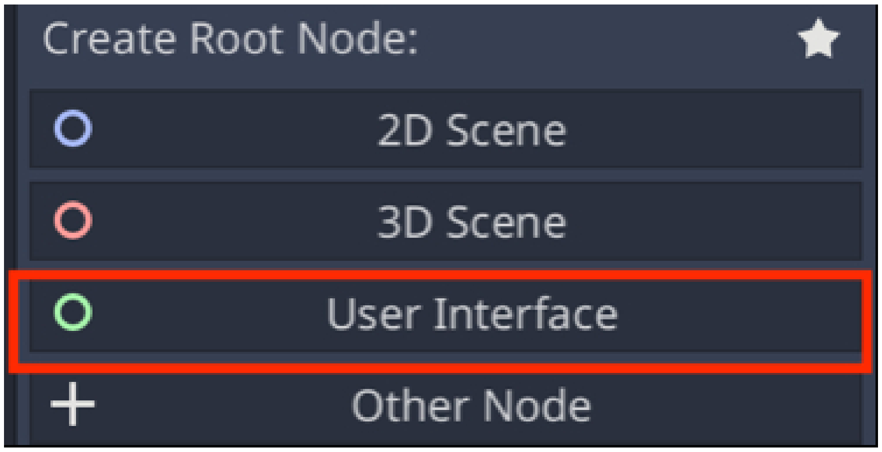
After completing this chapter, you will be able to:

- Create menus and make it possible for the player to navigate between them.
- Create buttons.
- Manage interaction with buttons.
- Link different scenes using buttons.
- Add background music.
- Mute the music using the keyboard.
- Add and configure a mini-map.
- Display the items collected as part of the user interface.

Creating a splash screen for the game

In this section, we will create a splash screen for the game. To do so, we will create a new scene, add a button to this scene, and ensure that the player, after pressing this button, is transferred to the game (i.e., the scene **level1**). So let's go ahead:

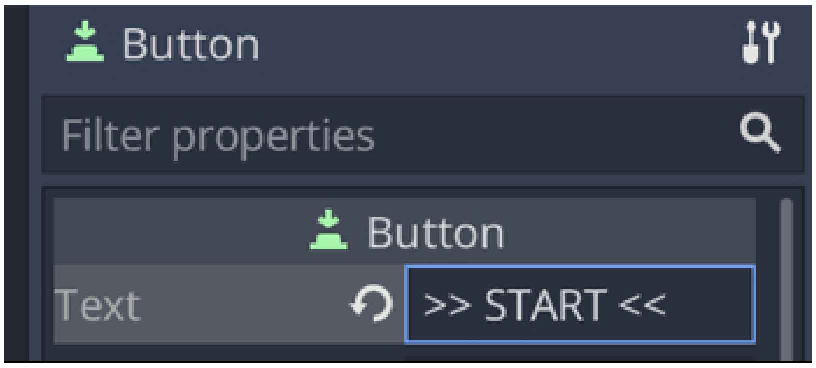
- Please save the current scene (**Scene | Save Scene**).
- Create a new scene: from the project menu, select **Scene | New Scene**.
- In the new window, select the type **User Interface** as we will be designing an interface.



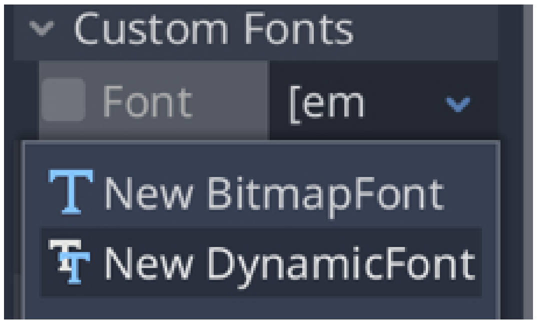
- Rename this scene **starting_scene** (**Scene | Save Scene As**).

We can now start to create a button and manage associated events (i.e., click):

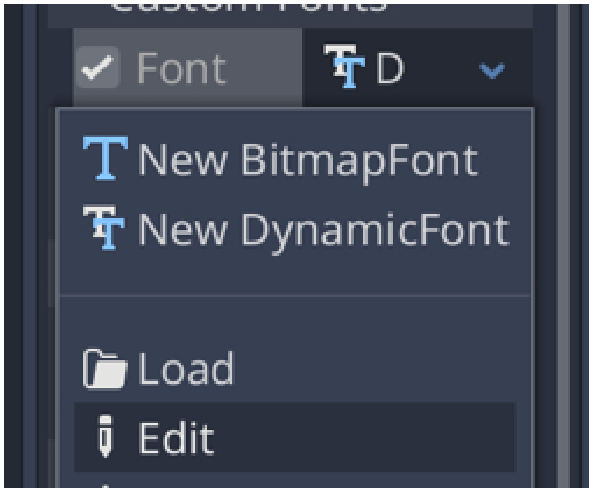
- Please right-click on the existing node called **Control** and add a child of type **Button** to this node.
- This will create a new node called **Button** to the scene.
- Using the **Inspector**, change the **Text** attribute of this button to “>> **Start** <<”.



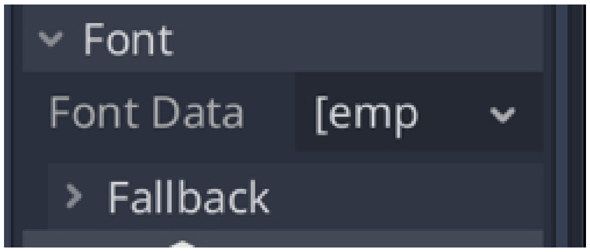
- Scroll down to the section called **Custom Fonts**.
- Click on the downward-facing arrow and select "**New Dynamic Font**" from the list.



- Click again on the drop-down menu and select **Edit**.

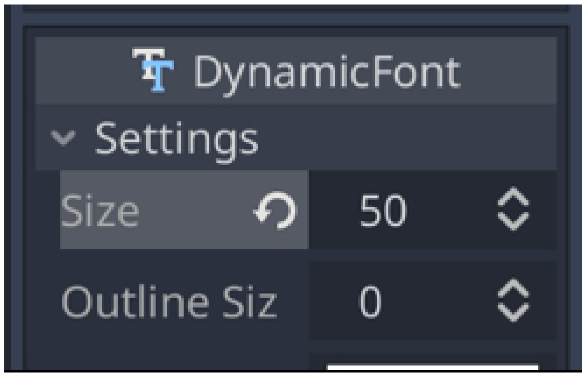


- In the new window, expand the section called **Font**.



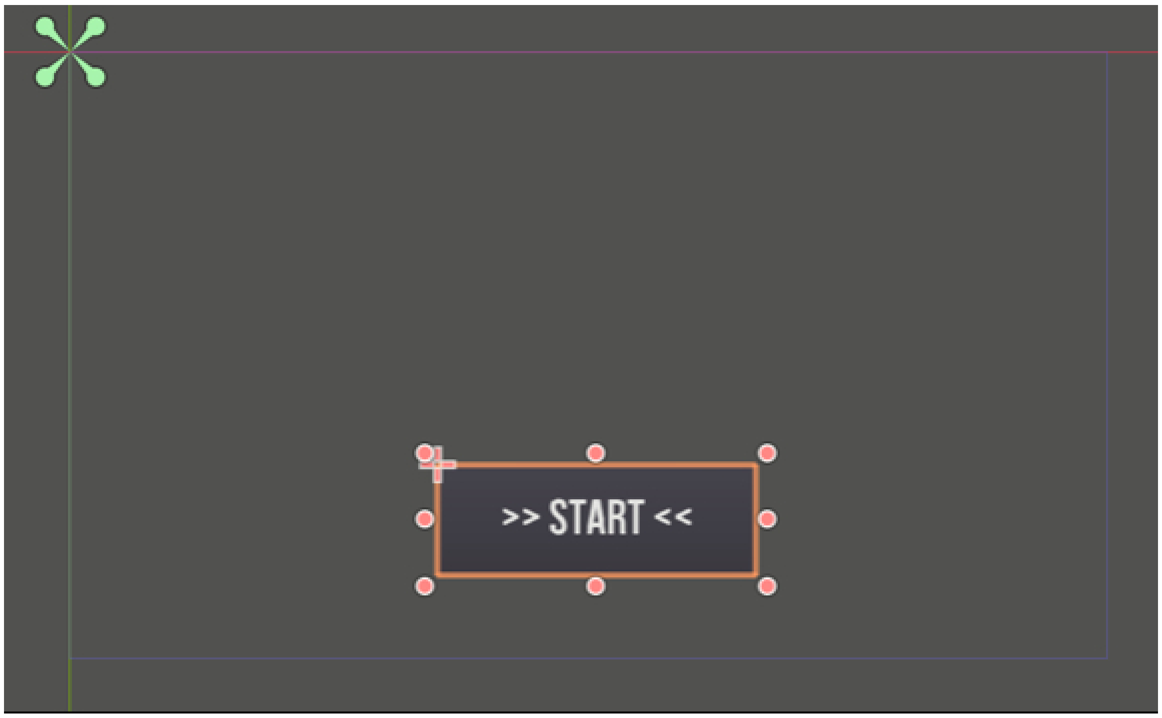
- You can now drag and drop the font **BebasNeue-Regular.ttf** from the **FileSystem** tab to the section **Font Data**.

Once this is done, you can expand the section called settings to increase the font size to



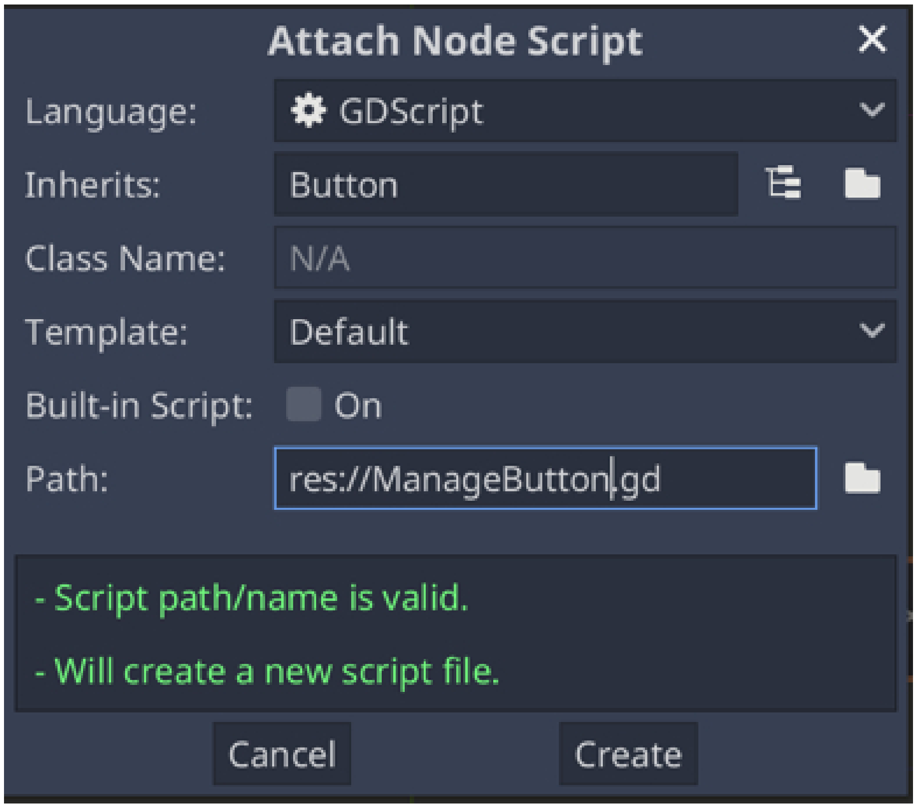
Size 50
Outline Siz 0

- Finally, you can move (drag and drop) the button near the bottom of the square that delimitates the size of the screen.



For an interaction to be created (i.e., to detect a click on this button), we will need to create a script. This script will include a function that will be called whenever the button is clicked.

- Please ensure that the node **Button** is selected.
- Attach a new script called **ManageButton** to this node.


Attach Node Script

Language: GDScript

Inherits: Button

Class Name: N/A

Template: Default

Built-in Script: On

Path: res://ManageButton.gd

- Script path/name is valid.

- Will create a new script file.

- Open this script in your code editor (i.e., double-click on the script).
- Modify the code as follows

```
func _ready():
connect ("pressed",self, "load_level")
```

```
func load_level():
    get_tree().change_scene("res://level1.tscn")
```

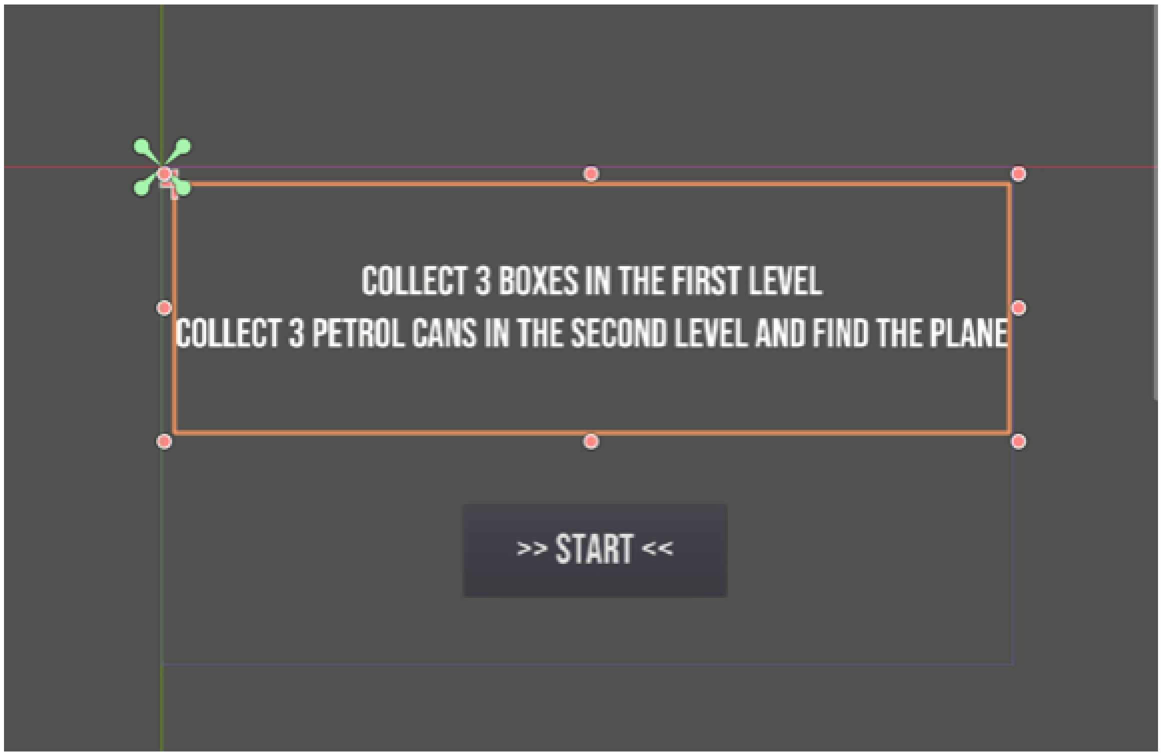
In the previous code:

- In the function **_ready** we connect the "**pressed**" event to the function **load-_level** that is declared in this script, hence the use for the keyword **self**.
- This means that whenever this button is pressed, we will call the function **load_level**.
- In the function **load_level**, we simply load the scene called **level1**.

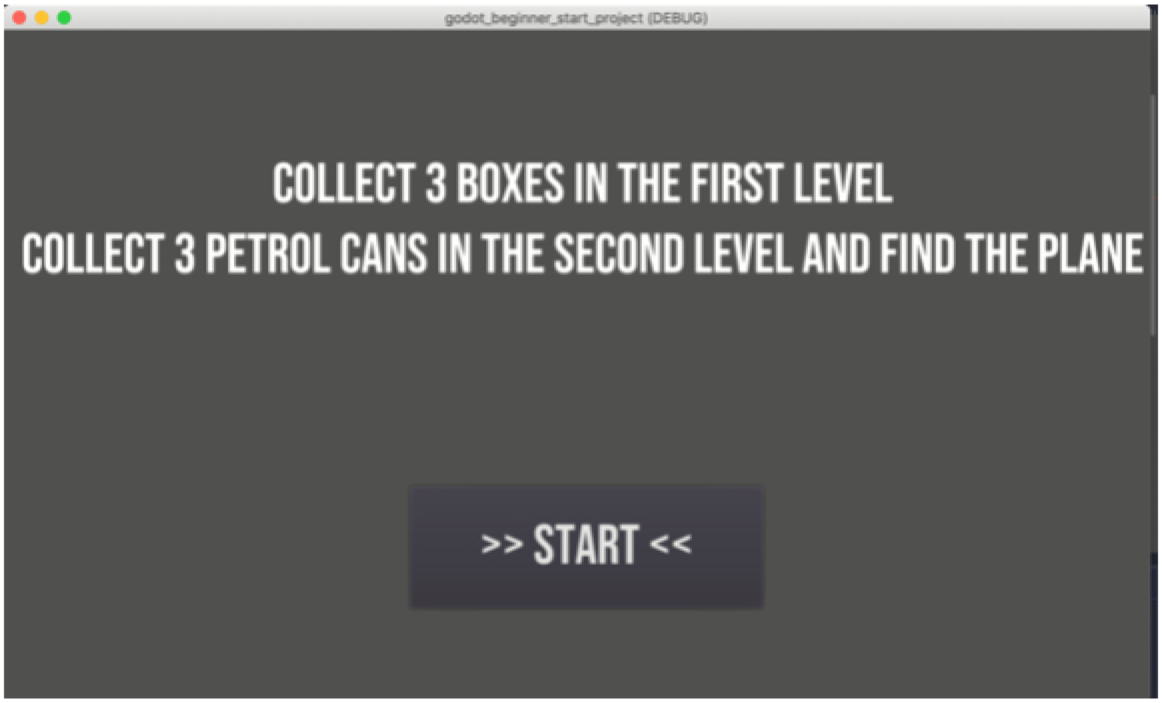
That's it.

Before we can test this splash screen, we will add some text that includes instructions on how to play the game.

- Please open the scene **starting_scene** (if it is not already open).
- Create a new node of type **Label** as a child of the node named **Control.**
- As we have done for the button, create a new **Dynamic Font** for this label.
- Edit this font.
- Use the font **BebasNeue-Regular**.
- Set the font size to **50**.
- Set the option **AutoWrap** to **On**.
- Resize the label.
- Center-align the text vertically and horizontally.
- Set its text to "**Collect 3 Boxes in The first level; Collect 3 petrol cans in the second level and find the plane**".

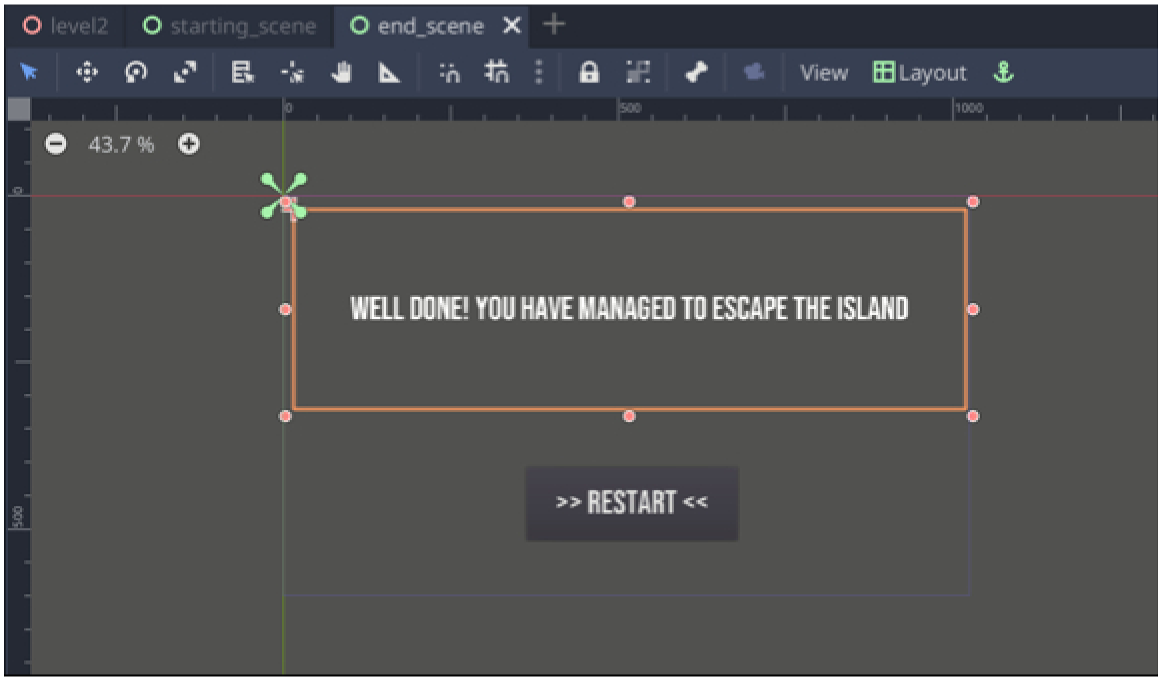

COLLECT 3 BOXES IN THE FIRST LEVEL
COLLECT 3 PETROL CANS IN THE SECOND LEVEL AND FIND THE PLANE
>> START <<

Once this is done, you can play the scene, and check that it works properly by clicking on the button (i.e., check that scene **level1** loads after pressing the button).


godot_beginner_start_project (DEBUG)
COLLECT 3 BOXES IN THE FIRST LEVEL
COLLECT 3 PETROL CANS IN THE SECOND LEVEL AND FIND THE PLANE
>> START <<

So, as we have created the splash screen, we could now create another scene for the end of the game.

- Please save the current scene (**CTRL** + **S**).
- Duplicate it and rename the duplicate **end_scene**.
- Open the new scene.
- Rename the button **RestartButton**.
- Change the label of the button to “>> **RESTART** <<”.
- Modify the text of the label to “**Well done! You have managed to escape the island**”.



- Modify the script **ManageButton** as follows:

```
func load_level():
if (get_name() == "RestartButton"):
get_tree().change_scene("res://starting_scene.tscn")
else: get_tree().change_scene("res://level1.tscn")
```

In the previous code:

- Because the script **ManageButton** is used in two different scenes, we need to tell whether the player is in the starting scene or the ending scene.
- If the button pressed is the **RestartButton**, we load the starting scene.
- Otherwise, we load the first level
- Save your scene and play it to see if it works.

Now that this scene has been created, we can modify our **Player.gd** script so that this scene loads after the player has managed to collect all the petrol cans and to access the plane.

- Please open the script **Player.gd** and add the following code to it (new code in

```
  bold):

elif (collision.collider.is_in_group("plane")):
if (nb_petrol_can <3) :
user_message_ui.set_text("Sorry you need 3 cans to fly the plane");
else:
get_tree().change_scene("res://ending_scene.tscn")
```

In the previous code we load the ending scene when the player has collected enough cans.

Displaying the score in each scene

At present, while our game works properly, it would be great to display the score onscreen. So, first, let's display the score onscreen using the same technique that we have used in the previous sections, that is by creating a **Label** node and by changing its content from the script.

- Please open the scene **level1**.
- Duplicate the node **timerUI** and rename the duplicate **scoreUI**.
- Move it and resize it (if need be) so that it appears in the top-right corner of the screen.

At this stage, we just need to access this **Label** from the script and update its text accordingly every time the score is increased.

- Please switch to the **Script** workspace.
- Open the script **Player.gd**.
- Add this code just before the function **_ready**.

```
onready var scoreUI:Label = get_node("../scoreUI")
```

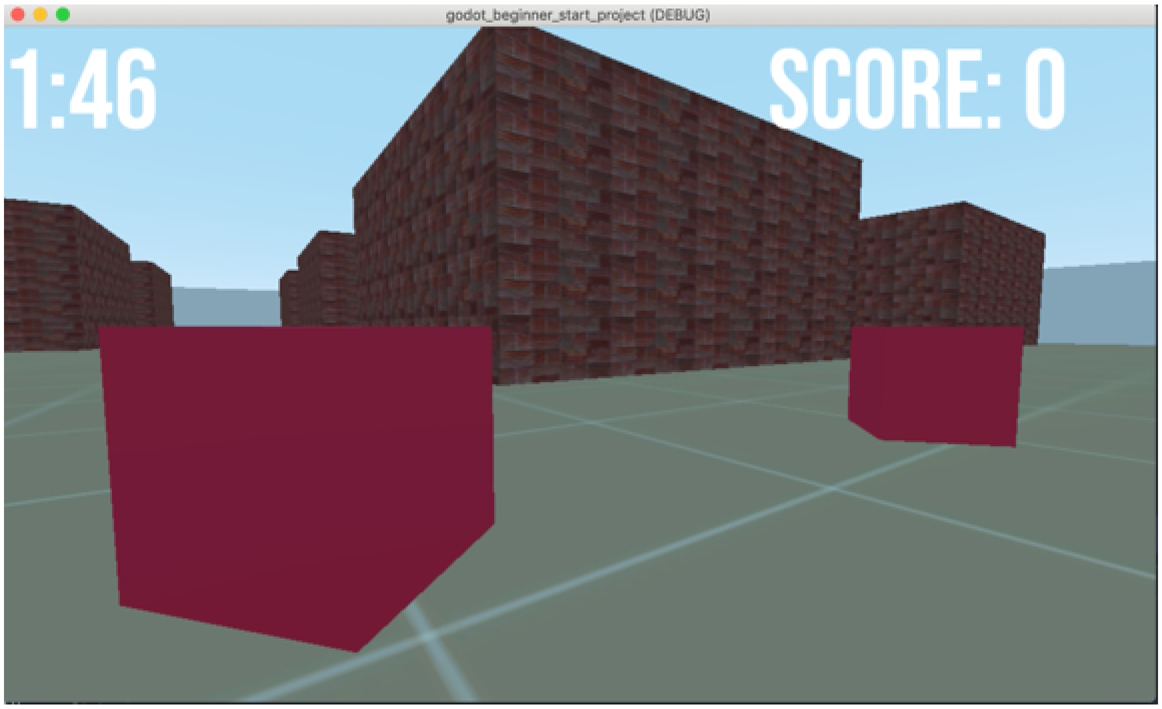
- Add this code to the function **_ready**.

```
scoreUI.set_text("Score: "+str(score))
```

- Modify the function **_physics_process** as follows (new code in bold).

```
score += 1
scoreUI.set_text("Score: "+str(score))
if (collision.collider.is_in_group("petrol-can")):
```

Please save your code and play the scene **level1**, you should see that the score is displayed onscreen.


godot_beginner_start_project (DEBUG)
1:46
SCORE: 0

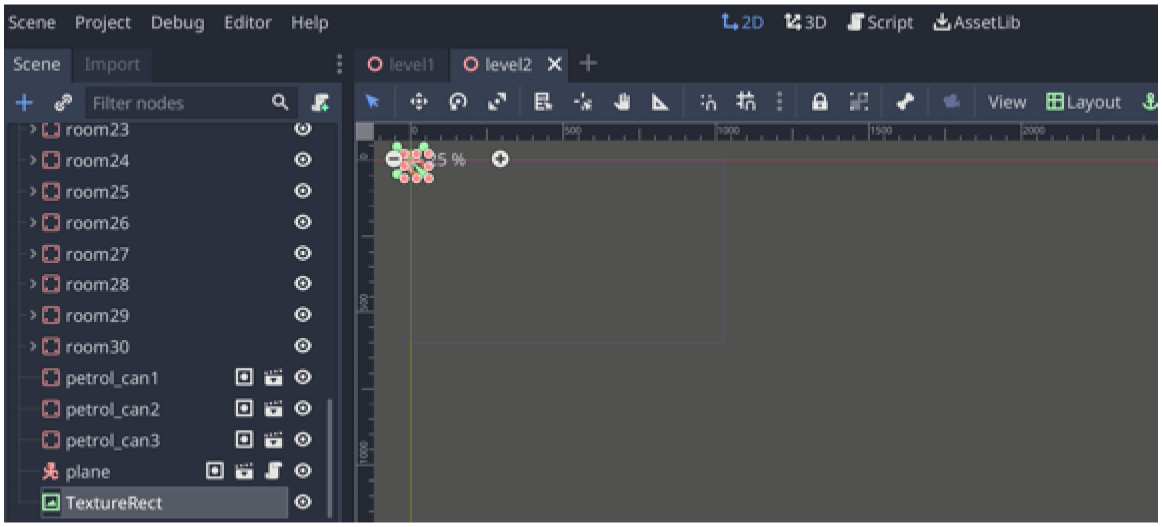
Displaying items collected as part of the user interface using images

At this stage, we have managed to display the score onscreen. However, it would also be great to display the number of petrol cans collected in the outdoor scene, instead of just using text. For this purpose, we will do the following:

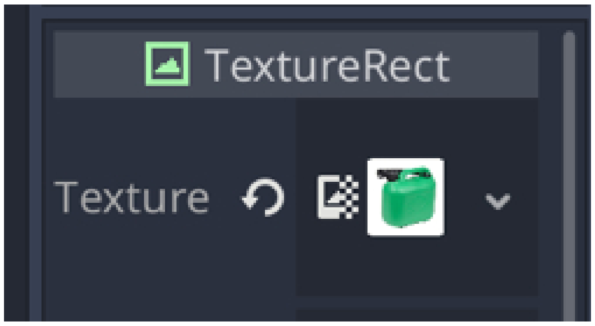
- Use images included in the start-up project.
- Add these images to the Graphical User Interface (GUI).
- Display an additional image for each petrol can that has been collected.

Let's start by creating and adding these images to the user interface:

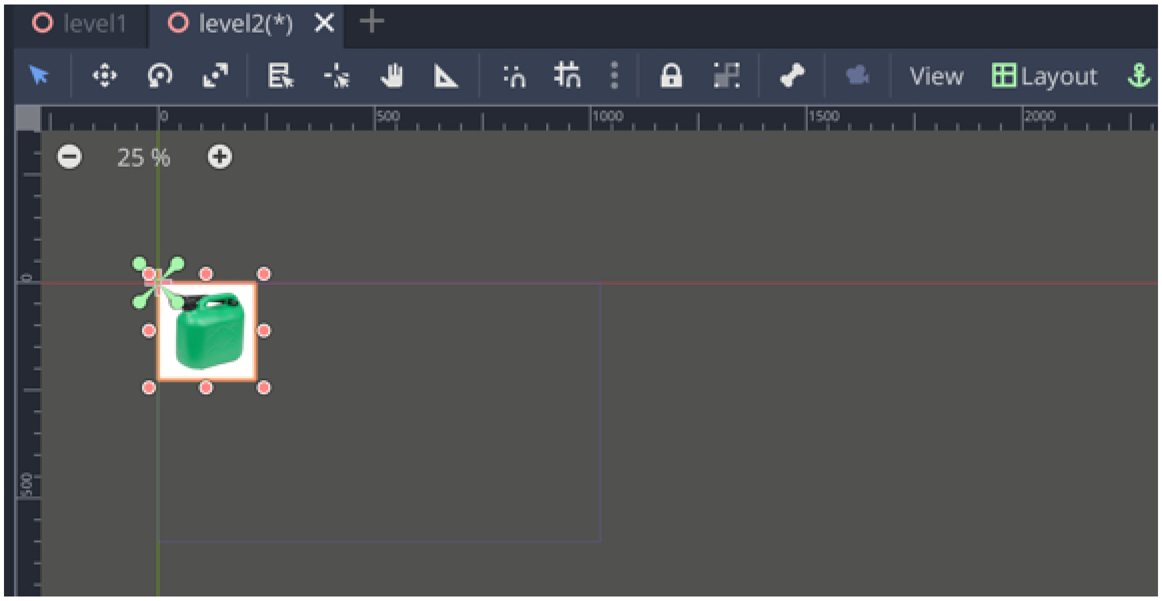
- Please open the scene **level2**.
- Select the **Spatial** node, and add a new child of type **TextureRect** to it. This will cause Godot to switch automatically to the 2D workspace.



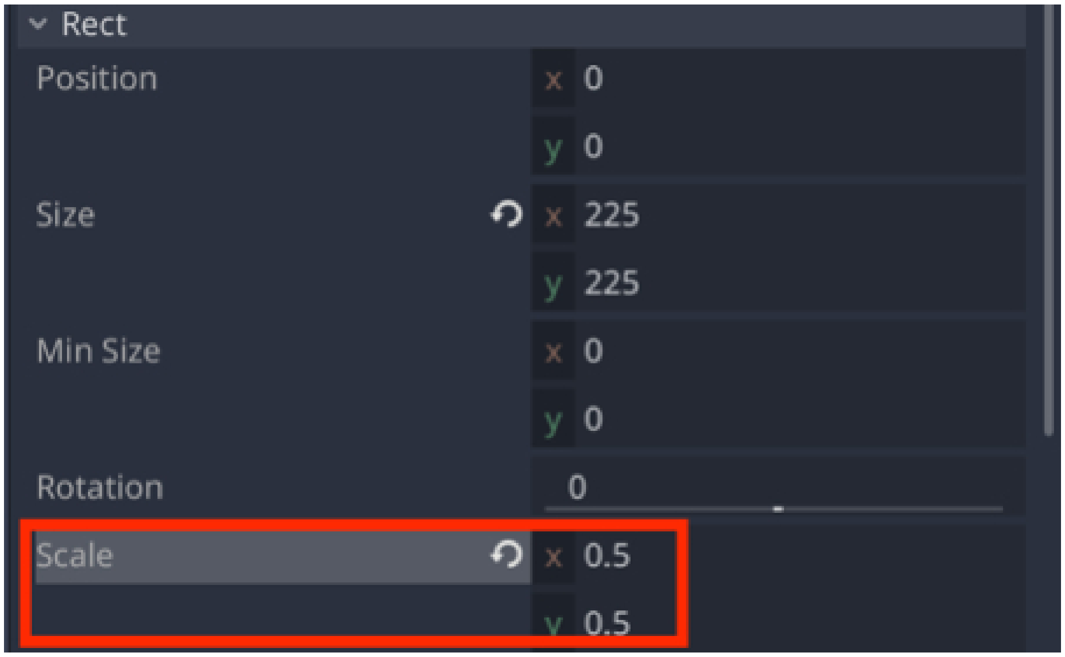
- A new node called **TextureRect** will also be created. Please rename it **petrol_can1_ui**.
- Select this node.
- Please drag and drop the file **petrol_can.jpg** from the **FileSystem** to the section **TextureRect | Texture** in the **Inspector**.



- This will display the petrol can's image on the interface also.



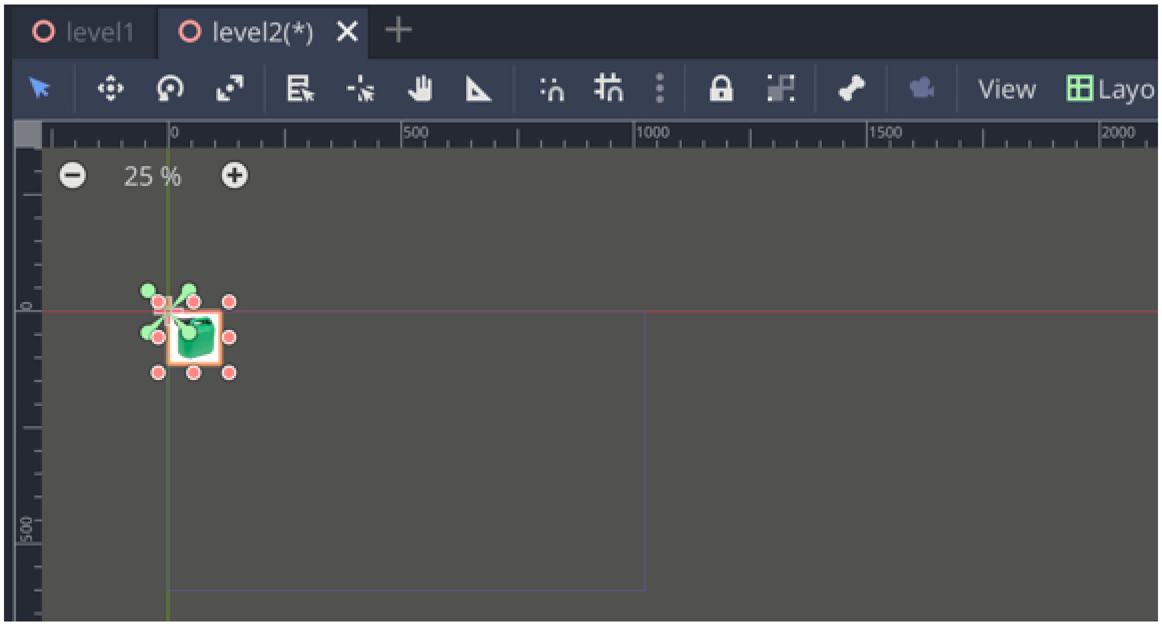
- In the **Inspector**, scroll down to the section called **Rect**, and change the **x** and **y** scale attributes to **.5**, so that the image is scaled down.



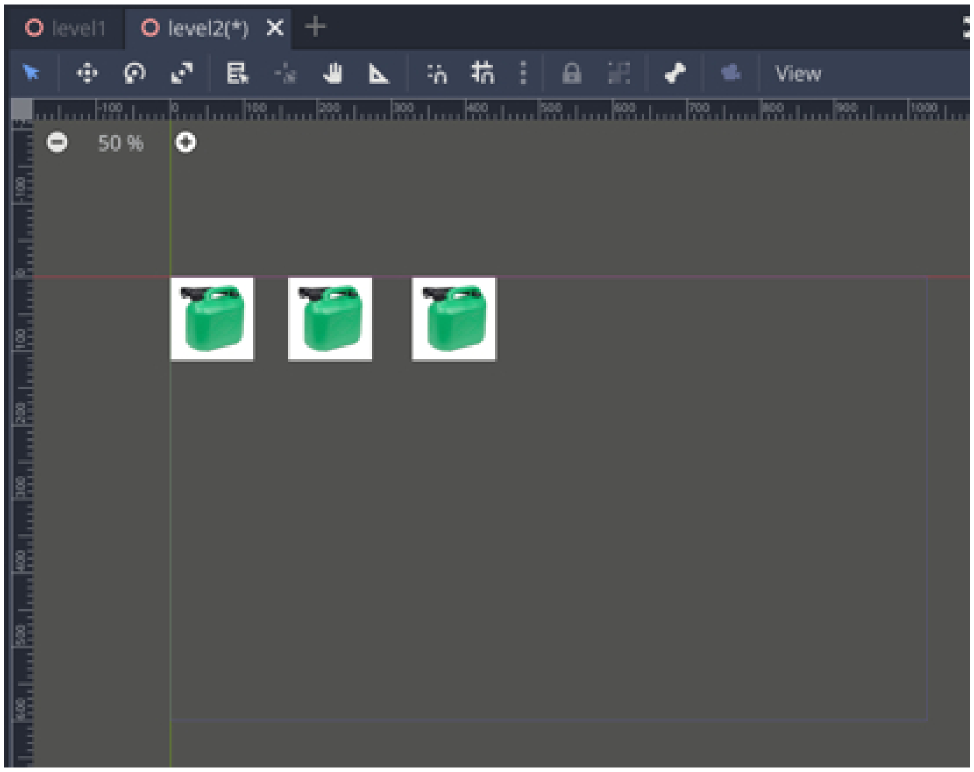
Rect

Position	x	0
	y	0
Size	x	225
	y	225
Min Size	x	0
	y	0
Rotation		0
Scale	x	0.5
	y	0.5

- You should see that the image size has been halved.



- Once this is done, you can duplicate the node **petrol_can1_ui** twice and call the duplicates **petrol_can2_ui** and **petrol_can3_ui**.
- Move the duplicates to the right of the first image, as per the next figure.



Now that we have defined three images, and added them to the user interface, it is time to control when and how they will be displayed using a script.

- Open the script **Player.gd**.
- Add this code just before the **_ready** function.

```
var petrol_can1:TextureRect
var petrol_can2:TextureRect
var petrol_can3:TextureRect
```

In the previous code, we declare three variables that will be used to link to the petrol cans to be displayed on the user interface; because these should only be displayed in **level2**, we will initialize them once the scene **levele2** has been loaded.

- Please create a new function called **init_ui_level2** as follows:

```
func init_ui_level2():
petrol_can1 = get_node("../level2/petrol_can1_ui")
petrol_can2 = get_node("../level2/petrol_can2_ui")
```

petrol_can3 = get_node("../level2/petrol_can3_ui")

petrol_can1.visible = false

petrol_can2.visible = false

petrol_can3.visible = false

In the previous code:

- We create a new function called **init_ui_level2**, that will initialise the petrol cans to be displayed as part of the user interface.
- We initialise the variables **petrol_can1**, **petrol_can2**, and **petrol_can3**.
- We then make sure that they are invisible for now (until the player collects a petrol can).

Finally, we just need to be able to call this function once the scene **level2** has been loaded. So please modify the function **_physics_proces** as follows (new code in bold):

if (score == 3):

get_node("../maze").queue_free()

var new_scene = load("res://level2.tscn").instance()

new_scene.set_name("level2")

get_parent().add_child(new_scene)

init_ui_level2()

In the previous code, we give a name to the new node added to the scene so that it can be accessed in the **init_ui_level2** function, we then call the function **init_ui_level2**.

Finally, we just need to display the petrol cans onscreen one they have been collected; so please add the following code to the function **_physics_process** in the script **Player.gd** (new code in bold).

if (collision.collider.is_in_group("petrol-can")):

nb_petrol_can+=1

match nb_petrol_can:

1:petrol_can1.visible = true

2:petrol_can2.visible = true

3:petrol_can3.visible = true

In the previous code:

We use the match keyword so that we execute different statements based on the value of the variable **nb_petrol_can**.

- If we have collected 1 can, then the first image is displayed onscreen.
- If we have collected 2 cans, then the second image is displayed onscreen.
- If we have collected 3 cans, then the third image is displayed onscreen.

That's it. Please save your code and play the scene **level1**; check that after moving to **level2** the petrol cans are displayed onscreen.

Adding sound effects

In this section, we will just add sound effects whenever the player picks up an object. Providing feedback to users is always a good idea, and this feedback can be provided in many forms, including using audio.

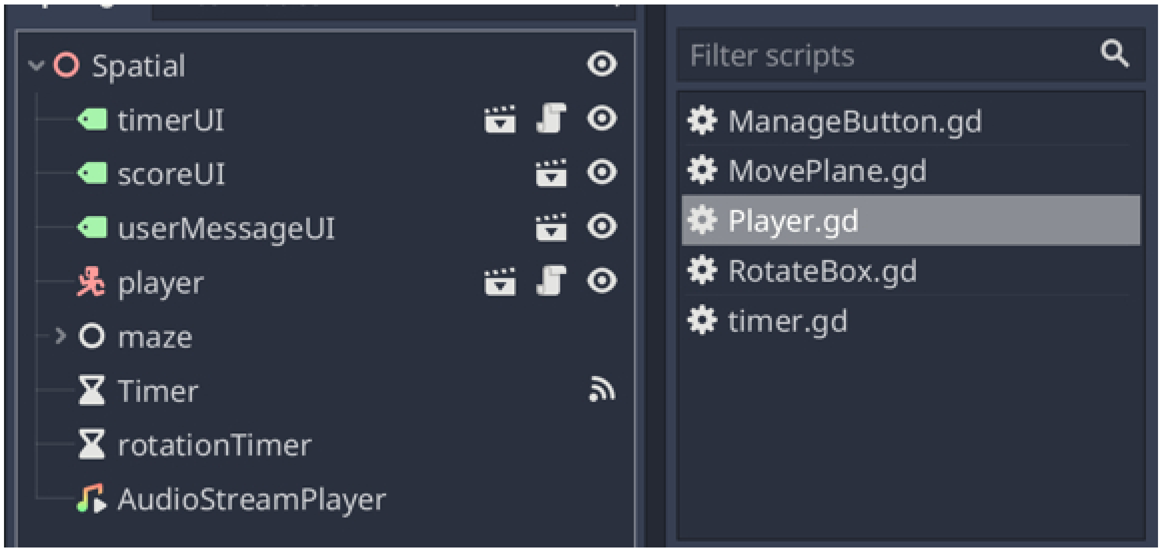
For this we will be using an **AudioStreamPlayer** node. An **AudioStreamPlayer** node is similar to a sound system (or MP3 player) with a charger for different CDs or songs. For this section, we would like to play a sound effect whenever an object has been picked up.

For this, we will do the following:

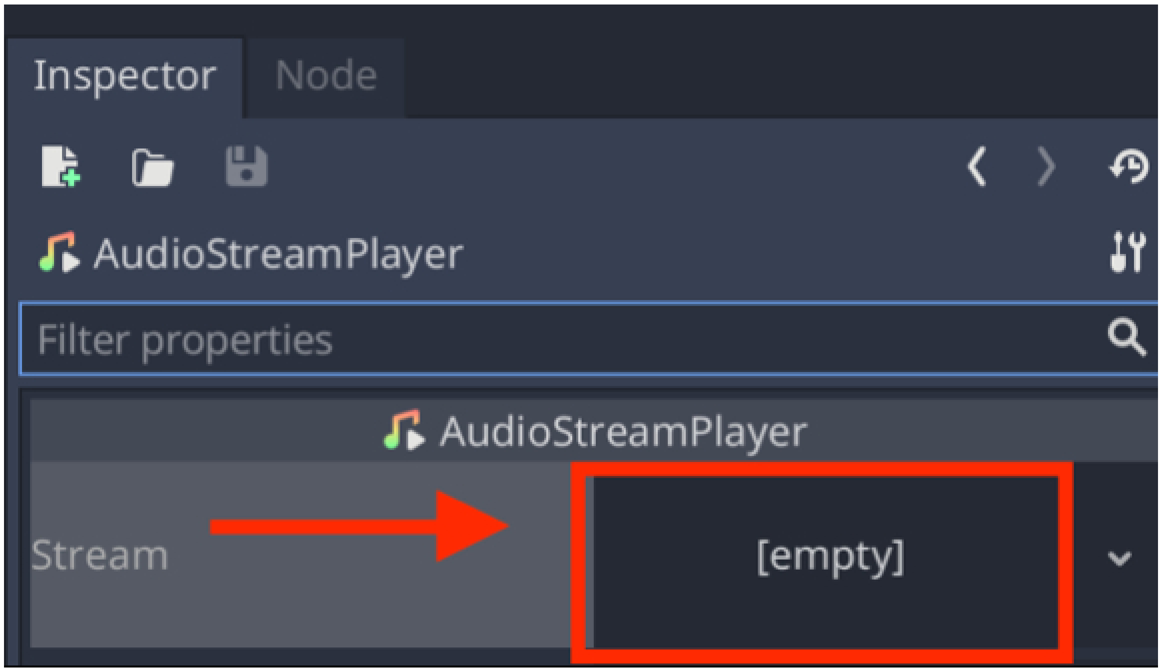
- Create an **AudioStreamPlayer** node.
- Associate an audio file to it, so that this sound can be played on the sound system.
- Play the sound effect whenever the player picks up an object.

So let's get started!

- Please open the scene **level1**.
- Switch to the 3D workspace.
- Add a new node of type **AudioStreamPlayer** to the root node (i.e., **Spatial**).
- This will create a new node called **AudioStreamPlayer**.



- Select this node in the **Scene** tree.
- Drag and drop the sound called **beep**, that is located in the audio folder, within the **FileSystem** tab, to the section called **Stream** in the **Inspector**.



Once this is done, we can modify the code in the script **Player.gd** to play this sound when we collect objects.

- Please open the script **Player.gd**.
- Add the following code just before the **_ready** function

```
onready var beep_sound:AudioStreamPlayer = get_node("AudioStreamPlayer")
```

- This code creates a variable that is linked to the **AudioStreamPlayer** that we have just created.
- Modify the function **_physics_process** as follows (new code in bold):

```
timer.start()
score += 1
beep_sound.play()
```

- In the previous code, we just play the sound included in the

AudioStreamPlayer whenever we collide with a box, a petrol can or the plane.

You can now play **level1** and progress to **level2** and check that the sound is played accordingly whenever you collect items.

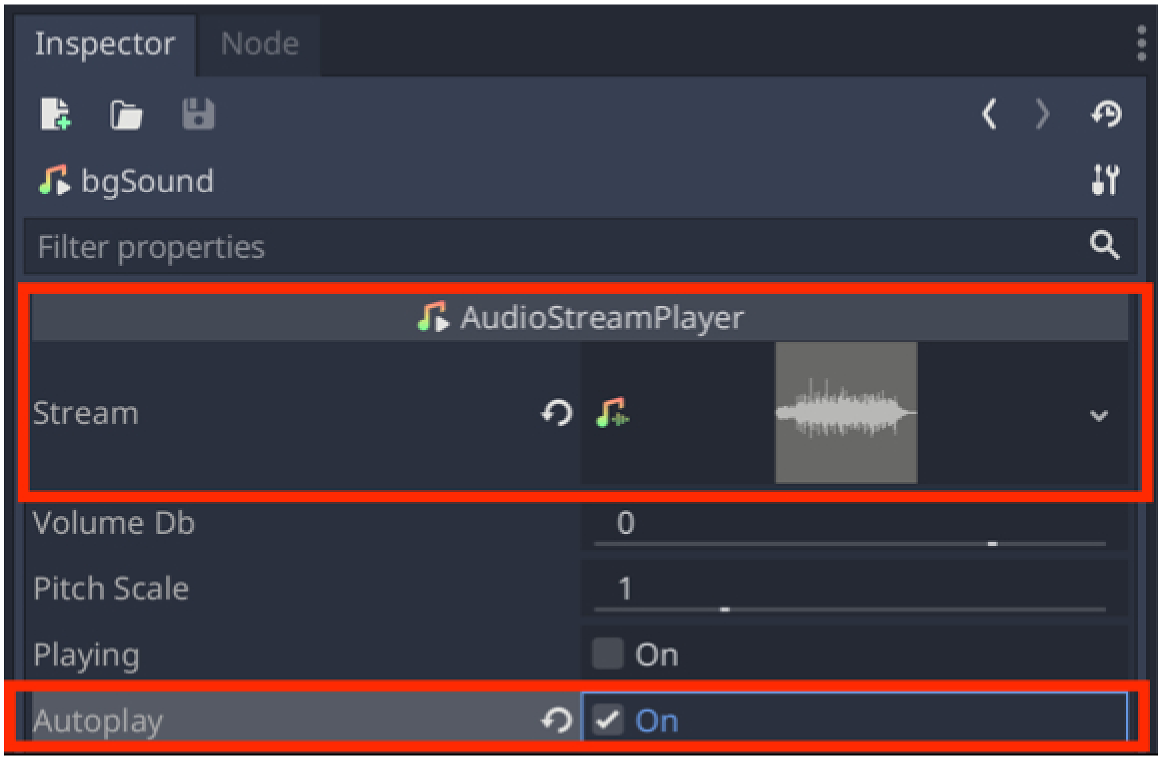
Although you have been using a sound available in the resource pack, you can also create your own sound for your game. The site http://www.bfxr.net makes it possible to create your own sounds easily by just using clicks. Note that this site uses Flash, and that alternatively, you can also use the site https://www.leshylabs.com/ apps/sfMaker/ instead.

Playing background music

At this stage, you are probably very comfortable with adding sounds, and we will perform this task again, but this time, for a background sound. The process will be to create an **AudioStreamPlayer** node, allocate a sound to it, and set the sound to play indefinitely.

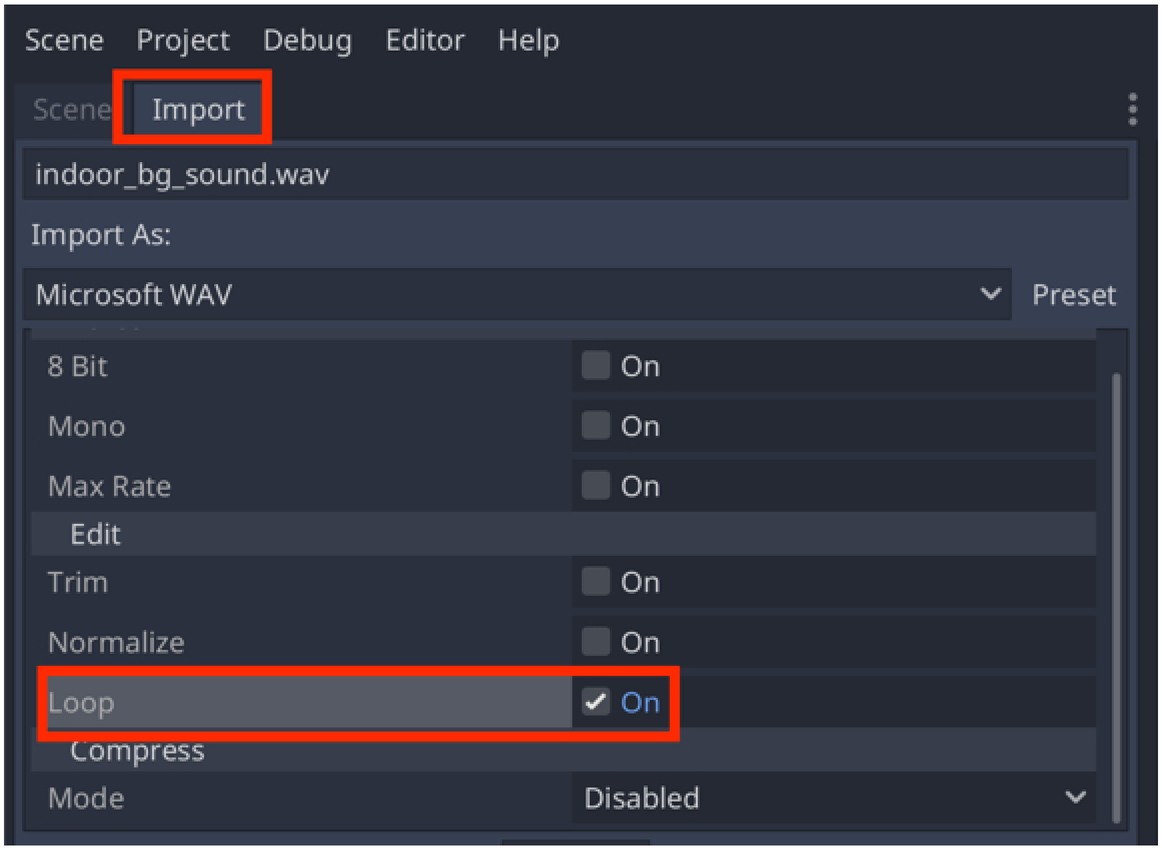
So let's get started!

- Open the scene **level1**.
- Create a new **AudioStreamPlayer** node.
- Rename this node **bgSound**.
- Drag and drop the file **indoor_bg_sound.wav** from the **FileSystem** tab to the **Stream** attribute of the node **bgSound**, and also set the attribute **AutoPlay** to **On**.



Stream
Volume Db 0
Pitch Scale 1
Playing On
Autoplay ✓ On

- Select the file **indoor_bg_sound.wav** in the **Scene** tree.
- Click on the **Import** tab, and set the option **Loop** to **On**.



indoor_bg_sound.wav

Import As:

Microsoft WAV   Preset

8 Bit	On
Mono	On
Max Rate	On
Edit	
Trim	On
Normalize	On
Loop	✓ On
Compress	
Mode	Disabled

- Play the scene and check that the background sound is played.

Note that the sound **indoor_bg_sound** was created by Kevin McLeod and is royalty- free. You can also download and use more sounds from his **official website**:

http://incompetech.com/music/royalty-free/collections.php

At this stage, we have managed to play the background sound; however, it would be great to make it possible for the player to mute this sound. This is a common feature in games and a good design practice to give more choice to players. So, what we could do next is to allocate a key (for example **P**) to the sound, so that whenever the player presses this key, the sound is toggled to the states **ON** or **OFF**. To do so, we need to do the following:

- Detect that the key **P** has been pressed.
- Create a variable that will determine whether the sound should be **ON** or **OFF**.
- Start or stop the sound based on the variable defined above.

So let's get started:

- Please open the script **Player.gd**.
- Add this code just before the function **_ready**.

```
onready var bg_sound:AudioStreamPlayer = get_node("../bgSound")
var sound_is_on:bool = true
```

- Add the following code to the function **_input(event)** (new code in bold):

```
func _input(event):
if event is InputEventMouseMotion :
mouseDelta = event.relative
if Input.is_key_pressed(KEY_P):
if (sound_is_on): bg_sound.stop()
else: bg_sound.play()
sound_is_on = ! sound_is_on
```

In the previous code:

- We are in the function **_input** which processes inputs, including those from the keyboard.
- We check whether the key **P** has been pressed.
- If this is the case, and if the audio is already **ON** we stop playing the background music.
- Otherwise, if the audio is **OFF**, we play the background music.
- In all cases, we change the value of the variable **sound_is_on** from true to false or from false to true.

The operator **!** is a **logical not** operator, which means, in simple terms, the opposite value. Since a Boolean variable can only have two values: if it is not true then it is false, and if it is not false, then it is true.

Please save your code and play the scene. Press the **P** key several times, and check that the background sound is either muted or played.

Adding a mini-map

Now that we have added a background sound, it would be great for our player to know his/her location within the environment, and a mini-map, in this case, would be useful. By mini-map we mean an outline of the scene displayed on top of the game. Mini-maps are usually a simplified representation of the environment, and display the position of the player (usually symbolized by a dot), the position of the NPCs (also symbolized by dots), along with the position of important items such as ammunitions or health packs.

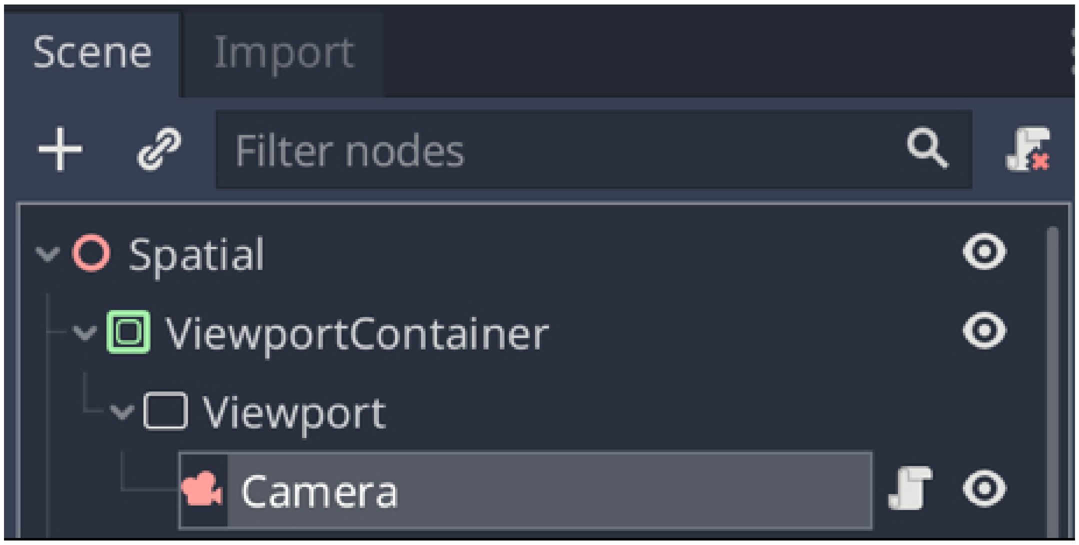
So how do we create a mini-map?

Well, it should be relatively simple, as we have, so far, looked at all the elements that we need to create this map. The process will be as follows:

- Create a camera.
- Display the image captured by this camera in the bottom-right corner of the screen.
- Identify the objects that we would like to be displayed on the mini-map.
- Create corresponding icons (e.g., red or green dots).
- Allocate these icons to a specific layer (we will explore this concept in the next paragraphs).
- Make sure that the camera allocated for the mini-map only displays items that have been linked to that layer: in other words, we are selective as to what we would like to capture (and display) with this camera.

So let's get started:

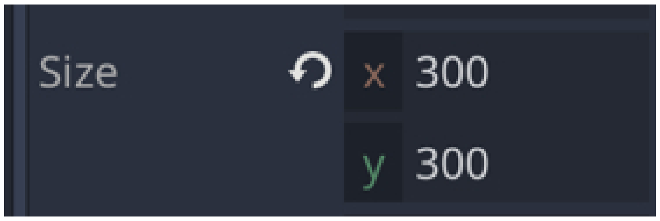
- Open the scene **level1**.
- Create a node of type **ViewportContainer** as a child of the node **Spatial**.
- Add a child of type **ViewPort** to this node
- Finally, add a child of type **Camera** to the **ViewPort** node.
- The **Scene** tree, should now look as follows:



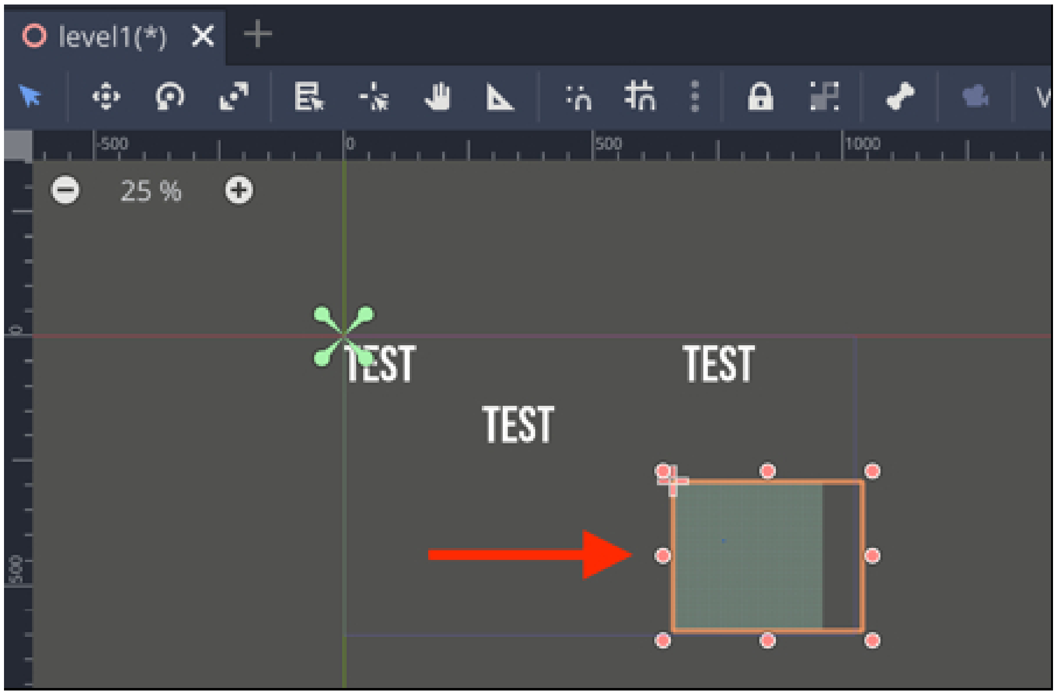
- The content of the camera will be displayed through the **Viewport** node. The content of a camera is usually displayed on the closest parent **ViewPort**.
- The **Viewport** node is held by the **ViewportContainer** node.
- The **ViewportContainer** node can be used to resize the viewport or to locate the **Viewport** onscreen.

Once this is done, it is time to configure these nodes:

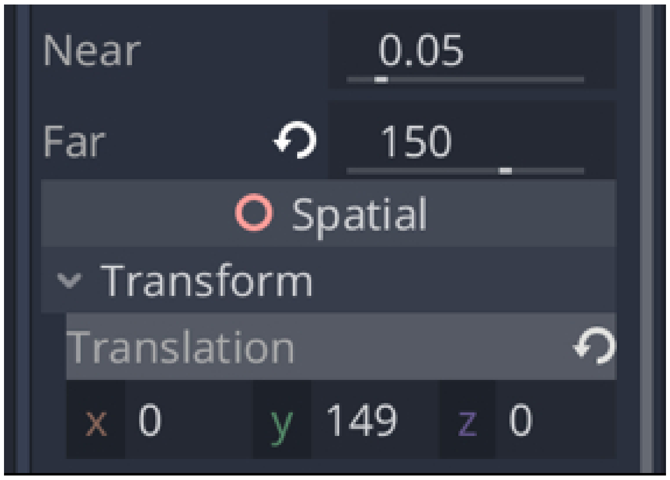
- Select the **Viewport** node in the **Scene** tree.
- In the **Inspector**, change its size to **(300,300)**.


Size x 300 y 300

- Select the **ViewportContainer** node in the **Scene** tree.
- Modify the size of the corresponding rectangle on screen so that it looks like the following:

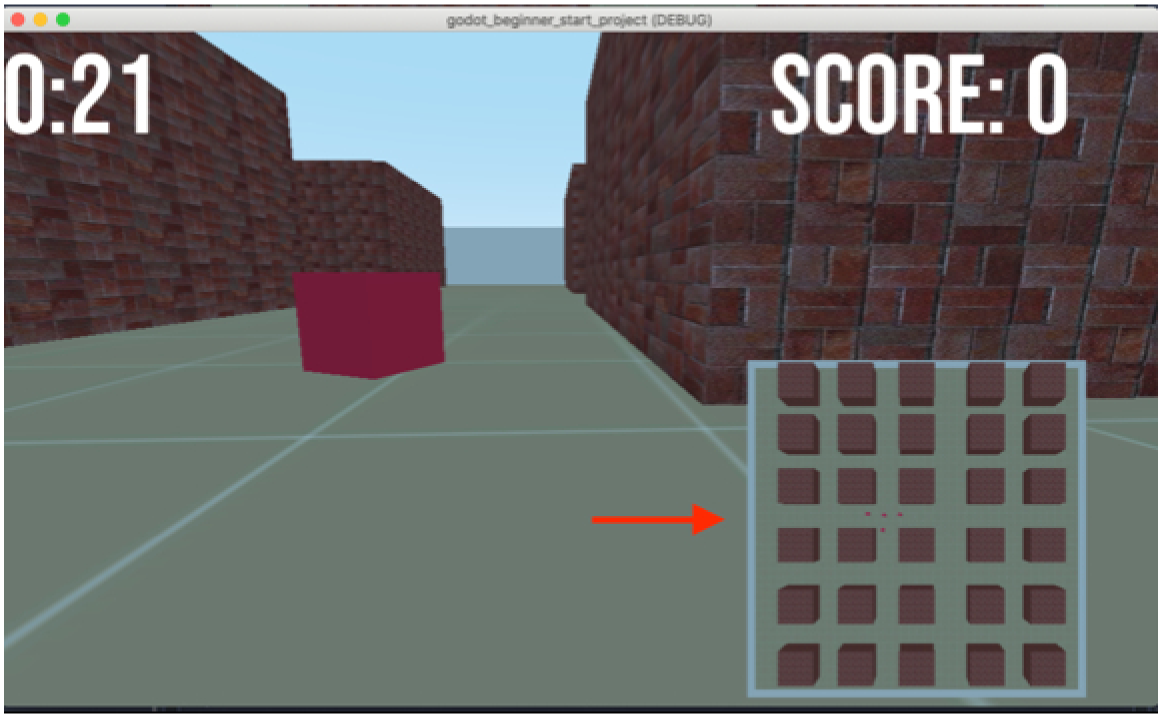


Finally, select the **Camera** node in the **Scene** tree (the child of the object **Viewport**), change its position to **(0, 180, 0)** its rotation to **(-90, 0, 0)**, and its **Far** property to **200**.


Near 0.05
Far 150
Spatial
Transform
Translation
x 0 y 149 z 0

This will elevate the camera along the y axis, ensure that it is looking down, and that objects within 200 meters will be visible and captured by this camera.

You can now play the scene and you should see, as per the next figure, that a min-map appears in the bottom right corner.



At this stage we have a mini-map, located in the bottom-right corner of the screen, that displays the environment around the player. As it is, it works perfectly; however, it would be great to also locate elements on this map with red or green dots, such as the plane or the petrol cans. So, we will add these to the map, and the process will be as follows; we will:

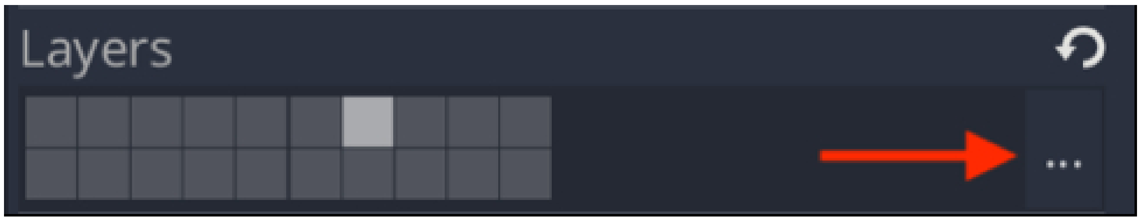
- Create dots that will be placed above each of the important objects.
- Add these dots to a specific layer.
- Make sure that the camera used for the mini-map only displays objects that are on this layer (e.g., dots), in addition to the terrain.
- Make sure that the main camera (the one attached to the **First-Person Controller**) does not display these dots.

First, let's create these dots:

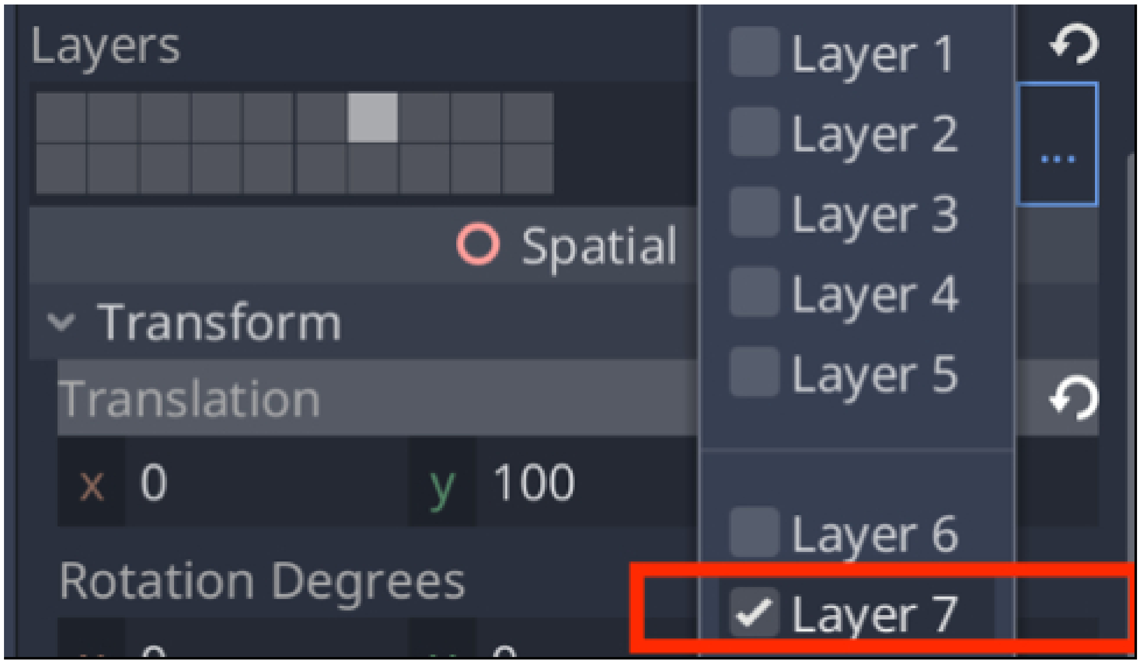
- Please select the node called **player**.
- Add a child of type **CSGSphere** to this node.
- Check that its **scale** attribute is set to **(1, 1, 1)**.
- Create a new **Green** material for this node: from the **Inspector**, click on the

downward arrow to the right of the label **Material**, and select **New SpatialMaterial**, then click on the white sphere in the **Material** section, and go to the **Albedo** section to create an apply a green color.

- Rename this node **green_dot**.
- When this is done, select the object **green_dot**, and change its position to **(0, 100, 0)**. This means that it will be 100 meters above the player.
- In the **Inspector**, in the section called **Layers**, click on three dots to the right of the label Layers.

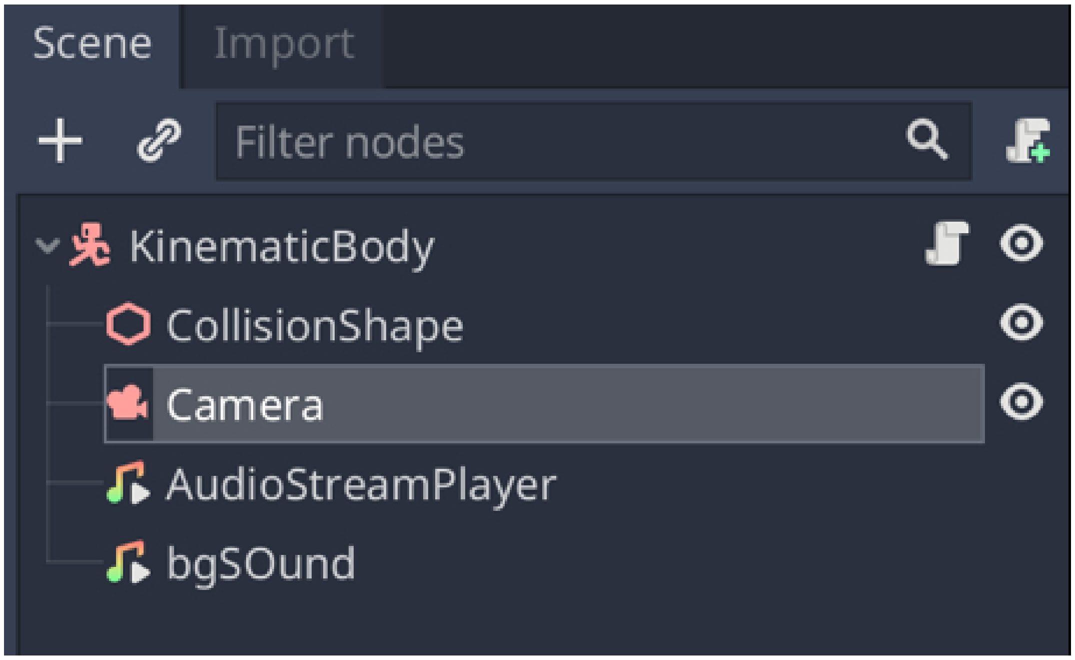


- From the contextual menu, ensure that only **Layer7** is selected.

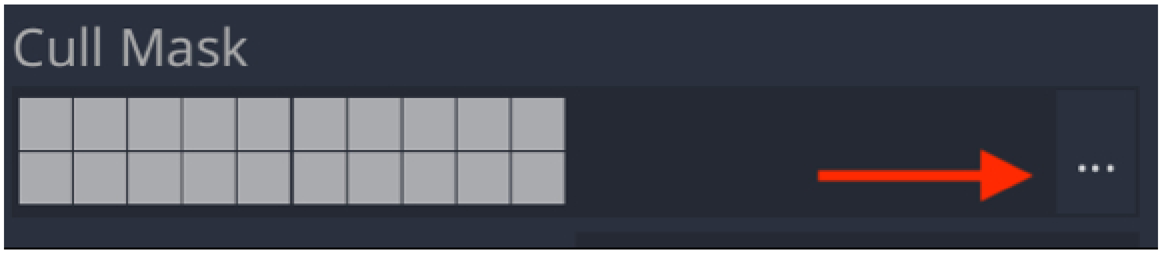


At this stage, we have specified that the node **green_dot** is on the layer **Layer7**. We just need to make sure that it is not rendered by the main camera (i.e., the camera that is attached to the **FPSController** by default); so we will specify that the main camera should render all layers, except from the layer **Layer7** (as nodes on the layer **Layer7** will be displayed by the camera allocated to the mini-map only).

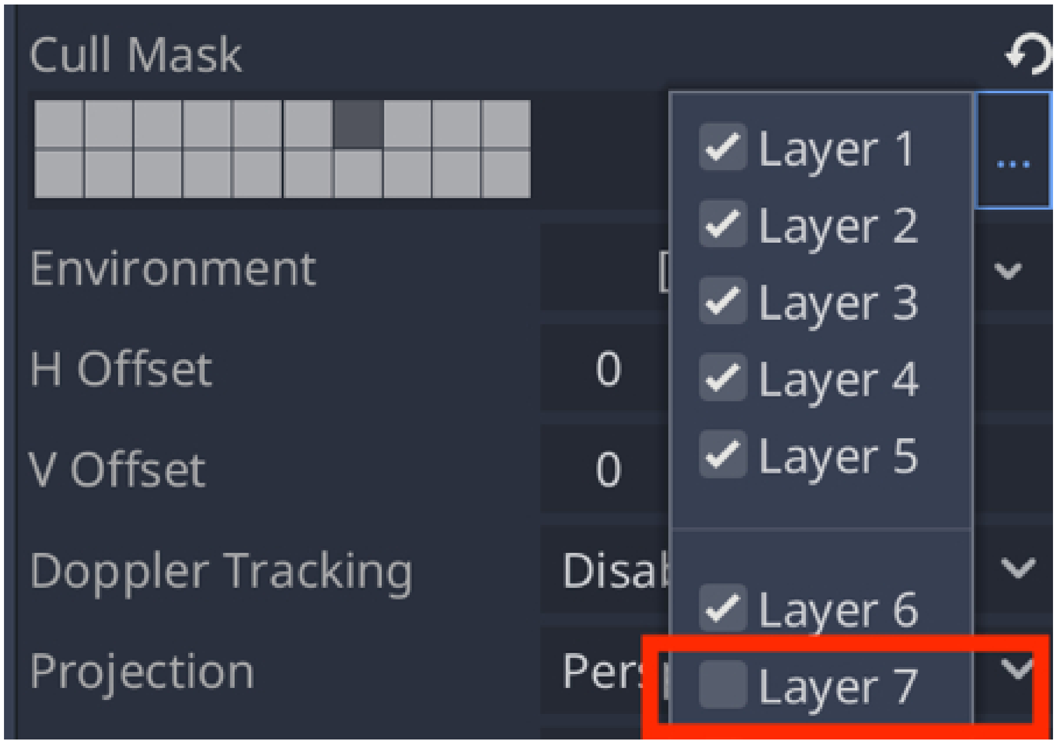
- Please right click on the node called **player**, and select the option "**Open In Editor**" so that we can edit the nodes within the node **player**.
- In the new window, select the object called **Camera**.



- In the **Inspector** window, locate the **Cull Mask** section and click on the three dots to the right of the label **Cull Mask**.



- Once this is done, make sure that the layer **Layer7** is deselected.


Cull Mask

Environment

H Offset 0

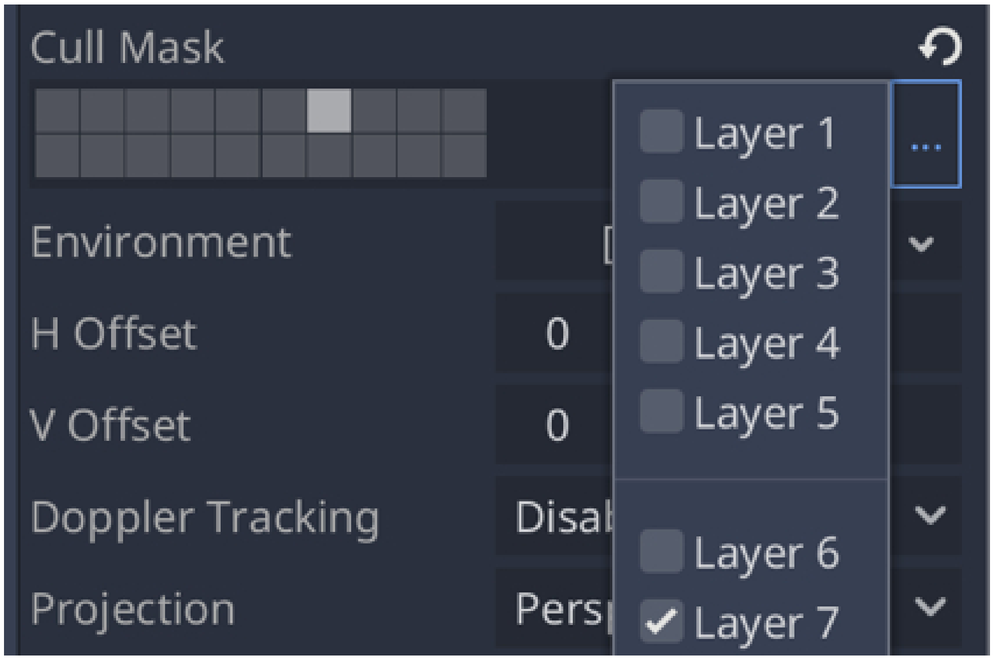
V Offset 0

Doppler Tracking Disab

Projection Pers

We now need to specify that our mini-map should display all objects on the layer **Layer7**; to do so, we will proceed as for the previous steps:

- Please reopen the scene **level1**.
- Select the camera associated with the mini-map.
- Change its **Culling Mask** so that it only displays objects that are on the layer **Layer7**.

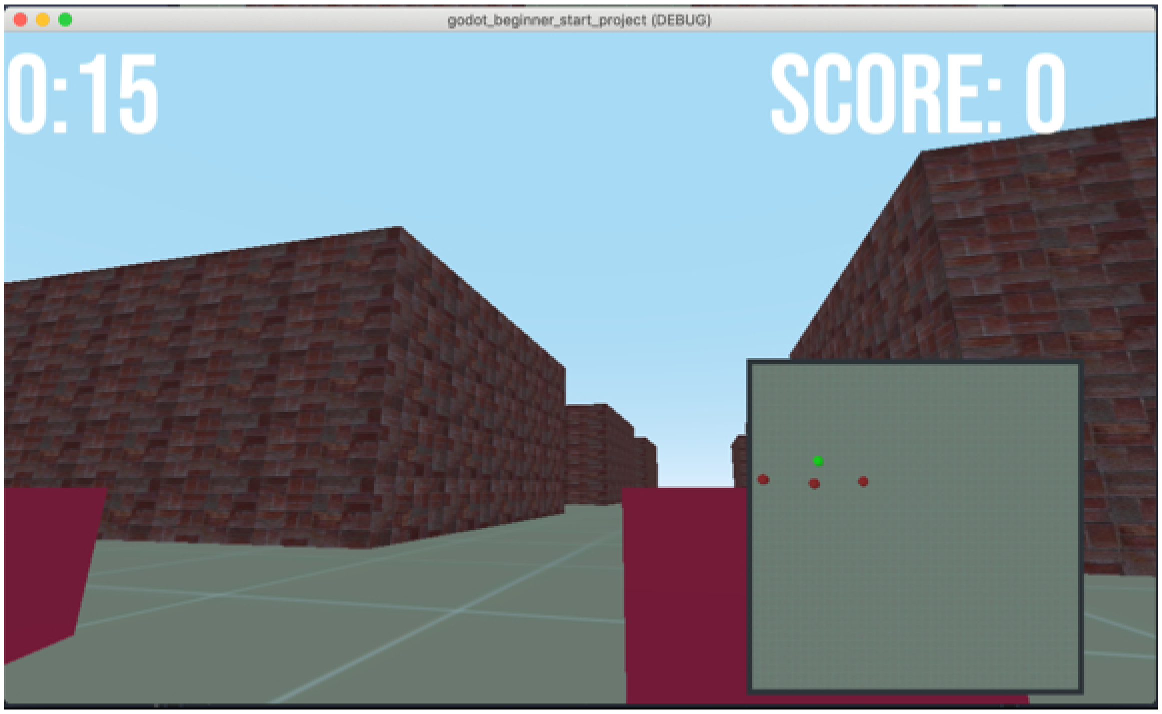


Now that you have managed to set up the dot representing the First-Person Controller on the mini-map, please repeat the previous steps to add dots, this time for the boxes to collect in both levels.

- Open **level1**.
- Select the node **CSGBox** that is a child of the node **ground**; set its layers property to both layers **Layer1** and **Layer7**.
- Create a new sphere.
- Rename it **red_dot1**.
- Create a new **red** material for this sphere.
- Apply this material to the sphere **red_dot1**.
- Duplicate this sphere twice. Rename these duplicates: **red_dot2**, **red_dot3** and **red_dot4**.
- Drag and drop each of the duplicates on each of the boxes in the first level.
- Once this is done, you should have one red dot for each box.
- Change the position of each of these dots to **(0, 100, 0)**. To speed up this process, you can search for the term **red_dot** in the **Scene tree**, select all four objects matching this keyword (i.e., **red_dot1**, **red_dot2**, **red_dot3** and **red_dot4**), and change their x, y, and z attributes simultaneously in the

Inspector window to **(0, 100, 0)** and their layer to **Layer7**.

You can now test the first level and check that the dots appear for the player and each of the boxes.



Note that you can use the same process to display dots in the second level; to avoid having to recreate the dots, you could do the following:

- Open the scene **level2**.
- Select the node **CSGBox** that is a child of the node **ground**; set its layers property to both layers **Layer1** and **Layer7**.
- Select the node **petrol_can1**.
- Add a new **Sphere** node as a child of the node **petrol_can1**.
- Set its color to **red**.
- Sets its position to **(0, 100, 0)** and its layer to **Layer7**.
- Repeat these steps for the remaining petrol cans and the plane.

Now, there is a last cool thing we could do; that is, to be able to toggle the map when the player presses the key *M*. So the idea is to hide or display the map accordingly; for this, we will go through the following steps:

- Detect when the key *M* is pressed.
- Change the value of a Boolean variable called **displayMiniMap**.
- Disable or enable the camera based on the value of this variable (i.e., true or false).

So, let's try this:

- Open the script **Player.gd**.
- Add this code before the function **_ready**.

```
onready var viewport_container:ViewportContainer = get_node("../ViewportContainer")
```

- Add the following code to the function **_input**.

```
if Input.is_key_pressed(KEY_M):
viewport_container.visible = ! viewport_container.visible
```

In the previous code, when the key *M* is pressed, we will enable or disable the node **ViewportContainer**, based on the value of the variable **viewport_container.visible**.

- Please save your script.
- Play the scene and check that pressing the *M* key successively deactivates and activates the mini-map.

Last but not least: when the end screen is loaded, you may not be able to see the mouse because it has been deactivated during gameplay; to ensure that the mouse is visible when the player needs to click on buttons, please add the following code to the function **_ready** in the script **ManageButtons**:

```
Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
```

LEVEL ROUNDUP

In this chapter, we have learned how to polish up our game by adding sound, displaying a simple inventory system, a mini-map, the score, and a splash screen. We became more comfortable with functions, variables, and components' properties. We managed to detect users' key input and to interact accordingly (i.e., by muting the sound or hiding the map). So, again, we have covered considerable ground to produce a relatively polished game with some of the key features that you will find in many 3D games.

Checklist

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

- Detect keystrokes from the script.
- Play and stop a sound from the script.
- Load scenes.
- Display text onscreen using UI elements.
- Attach a script to an object.
- Update UI elements.
- Activate and deactivate a node from a script.

Quiz

Now, let's check your knowledge! Please answer the following questions. The answers are on the next page.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

The following code when linked to a button will call the function **load_level** if the button is pressed.

```
func _ready():
connect ("pressed",self, "load_level")
```

1. Please specify whether the following statement is **TRUE** or **FALSE**.

The following code will load the scene called **level1.tscn**.

```
get_tree().change_scene("res://level1.tscn")
```

1. Find and write the missing code below so that the script can check whether the name of the node linked to the script is **Button**.

```
if (MISSING CODE == "RestartButton"):
```

1. Please specify whether the following statement is **TRUE** or **FALSE**.

By default, a node of type **AudioStreamPlayer** will play a sound automatically when the scene starts.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

An **AudioStreamPlayer** node can play wav files.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

A camera can only display one layer onscreen.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

A node can be on several layers.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

It is possible to display the content of the images captured by a camera using a **ViewPort** and a **ViewportContainer** node.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

It is possible to hide or display a node from a script using the node's attribute called **visible**.

1. Please specify whether the following statement is **TRUE** or **FALSE**.

It is possible to display text onscreen using the node **Label**.

Solutions to the Quiz

1. **TRUE**.
2. **TRUE.**
3. Find and write the missing code below so that the script can check whether the name of the node linked to the script is **Button**.

```
if (get_name() == "RestartButton"):
```

1. **FALSE**.
2. **TRUE**.
3. **FALSE.**
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 improve the flow of your game. So for this challenge, you will be creating an instruction screen and a game-over screen.

- Create two additional scenes: a briefing and debriefing scene for the first scene.
- Link these scenes, so that the player first reads the instructions, then after clicking a button, moves to the briefing screen, then to the first level. After completing the maze scene, a debriefing screen should be displayed, congratulating the player and also explaining what should be done in the outdoor level.
- All screens can be created with UI components (e.g., text, buttons, or images).

Challenge 2

Here, you will modify your game so that if the player runs out of time, s/he should be taken to the game-over screen.

- Create a **game-over** scene.
- In the first level, if the time is over, the **game-over** scene should be loaded.

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.

SCRIPTS

How do I create a script?

In the **Script** workspace, select: **File | New Script**.

How can I check that my script has no errors?

Save your script and any error should be displayed underneath.

What is the dot notation for?

The dot notation refers to **Object-Oriented Programming**. Using dots, you can access properties and functions (or methods) related to a particular node.

INTERACTION WITH ASSETS

How do I detect collisions?

To detect collisions from the **First-Person Controller** (player), and provided that it is based on a **KinematicBody** node (as is the case in this book) you can use the function **get_slide_count**.

How do I destroy objects?

To destroy an object, you can use the function **queue.free**. For example, to destroy a node called maze, you would use the following code:

```
get_node("../maze").queue_free()
```

How can I create a scoring system?

For a simple scoring system, you can create an integer and increase its value by one every time the player has collected an item.

How do I create a text to be displayed onscreen?

Create a **Label** node.

How do I update a text to be displayed onscreen?

You need to find the **Label** node object, and modify its text attribute. So to display the message **“Hello”** using a **Label** node with the name **messageUI**, the following code could be used:

```
get_node("../messageUI").set_text("Hello")
```

How can I empty (i.e., delete) the text onscreen?

You just need to set its text attribute to an empty string; for example, the following code will empty the text field **messageUI**:

```
get_node("../messageUI").set_text("Hello")
```

How can I display the value of a specific variable onscreen?

You just need to access the **Label** node where you need to display this variable and set its text attribute with additional text if need be; for example, the following code displays the text “**Score =**“ followed by the value of the variable **score**:

```
var score:int = 20;
get_node("../messageUI").set_text("Score: "+str(score))
```

AUDIO

What type of node can I use to play audio?

You can use a node of type **AudioStreamPlayer**.

How do I play a sound?

- Create a node of type **AudioStreamPlayer**.
- Assign an audio file to this node.
- Set the attribute AutoPlay to **On**, or call the **play** function from that node.

DETECTING USER INPUTS

How can I detect keystrokes?

You can detect keystrokes by using the function **_input**. For example, the following code detects when the key **E** is pressed.

```
func _input(event):
if Input.is_key_pressed(KEY_E):
```

How can I detect a click on a button?

To detect clicks on a button, you can do the following:

- Create a new **Button** node.
- Create a new script and link it to this object.
- Add this code to the script.

```
func _ready():
connect ("pressed",self, name_of_function_to_call)
```

- Write the function **function_to_call**.

Thank you

I would like to thank you for completing this book; I trust that you are now comfortable with GDScript and that you can create interactive 3D game environments. This book is the second in a series of four books on Godot, so it may be time to move on to the next book for the intermediate level where you will learn more advanced features, including Artificial Intelligence, 3D character animation, and finite-state machines.

The book is currently in the making, and you should be able to access it soon from the official page: **http://www.learntocreategames.com/books/**. If you have subscribed to my mailing list, you should be able to receive a notification.

Please also leave an honest review, this would mean the world to me and it would also help other people to assess whether this book could help them.

So that the book can be constantly improved, I would really appreciate your feedback and hear what you have to say. So, please leave me a helpful review on Amazon 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!

Don't miss out!

Click the button below and you can sign up to receive emails whenever Patrick Felicia publishes a new book. There's no charge and no obligation.

Sign Me Up!

https://books2read.com/r/B-A-NXXC-EPVNB

BOOKS 2 READ

Connecting independent readers to independent writers.