Godot 3D Game Development

2D Adventure Games, 3D Maths and Physics, Game Mechanics, Animations, and 3D Game Development

MARIJO TRKULJA

bpb

Godot 3D Game Development

2D Adventure Games, 3D Maths and Physics, Game Mechanics, Animations, and 3D Game Development

MARIJO TRKULJA

Godot 3D Game Development

2D Adventure Games, 3D Maths and Physics, Game Mechanics, Animations, and 3D Game Development

Marijo Trkulja

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: 2023

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-224-6

www.bpbonline.com

Dedicated to

Aryan ancestors

Перун Род Мокош

About the Author

Marijo is a rare author with realized writing potential in different areas and languages. Formally, he has a college technical education and a master's in economics. His sphere of interest is ancient Arian history, the technology of the past and future, and recently (from 2019) game development. He is a writer of many well-known books about the Godot game engine.

Marijo promotes peaceful video games where the player can learn and have fun. He also believes in the power of nature and writes about it in "Living in nature Marijo Trkulja." When people ask for his teachings, he says to read "Spiritual Adventures of young Hakun".

Acknowledgement

Usually, work like this has many contributors, which are planned at the beginning. However, fortunately, the work is the product of one author, which is good for several reasons. The reader has the feeling of a unique educational flow, the level of knowledge is gradually raised, and the method of education is similar throughout the material.

My gratitude goes to the editors from BPB Publications, and many unknown individuals who worked to make this material the best possible.

Preface

The book is the author's master peace about game development with the Godot game engine. Why? Because it's my best knowledge and practice for creating 2D and 3D video games.

Students of this material will learn through real game examples. After every game example, a student will know how to create an initial segment of different video game types.

Material is covered with free code parts in the repository folder and with paid educational material, "Learn to make commercial video games Godot mega tutorial." Students can find games explained in the book "Slavs Make Games itch."

There are **nine chapters** of learning material.

Chapter 1 is an introduction and encouragement with a "Hello World" example for students to be comfortable with the Godot game engine and GD Script.

Chapter 2 will cover GD Script programming concepts as the primary coding tool for 2D and 3D video games in Godot. The student will learn about variables, conditional branches, loops, arrays, and other helpful programming concepts and tools.

Chapter 3 will teach about 2D Math and 2D Physics in games. Computing is essential when you design game objects, and the student will learn about it with 2D game object properties and methods.

Chapter 4 will cover creating a 2D game character, making a 2D game environment, creating game props, troubleshooting a code, coding ethics, and many other game programming tips.

Chapter 5 teaches turn-based game-play, game character movement, 2D Platformer as a quest system, TileMap as background, and coding for a platformer character.

Chapter 6 will introduce 3D games, spatial, vector3, Creating 3D game objects with a GD Script, RigidBody methods, prototyping a game scene, and essential steps in a prototype testing process.

Chapter 7 will cover game planning, character design, props, and environment adding for a 3D platformer video game.

Chapter 8 will teach about 3D RPG adventure planning and design, MRC and NPC creating process, an entire movement in the 3D game space, and animated different movement types.

Chapter 9 will cover the inventory system, save system, and video game publishing process.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/7rif7de

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Godot-3D-Game-Development**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

CHAPTER 1
Introduction

Welcome

You are starting to read an excellent book. This book is about you and how you will shape your future through the world of video games. Do you know that you are a creator? Yes, even when you create nothing, you still make something. And now, you are one step to knowledge about creating fantastic video games. You are in front of life's greatest adventure.

Programming is not so complex as you think. And programming video games can be easy too. You need to go one step at a time.

The knowledge of this book is about Godot. Game engine - Godot. For a beginner, a game engine is a tool to create a video game, and for a game developer, this is a divine device for realizing a vision.

Interestingly, 18 million people search for games on the Internet every day. Godot has more than 14 million content on the internet (2022) and is the number one 2D game engine with promising 3D capabilities. So, now you can see that you are on the right track and in good hands.

As a Godot books writer, this is my fifth book. In the first one, *GD Script*, I taught Godot beginners about Godot's default script for a coding video game. The second one, *Making Games with GD Script* deepens the GD Script knowledge. Then, in the "Autonomous car systems," students learned about data storing and JSON capabilities. Finally, I am focussing on game developers in the *Mastering Godot* book, but beginners can still read it.

In this book, I will still teach some beginners stuff to prepare any Godot beginner for making two 2D video games and one 3D video game. Every game you learn to make here is the basis for a market-ready video game. You can create it for fun or sale on the video game market.

It's good to know that the Godot game engine games can be used or sold without paying any royalty to Godot developers.

Word to game developer

You aren't reading a beginner's book, but any beginner in game development can learn a lot from it. For example, sometimes you compete with other game developers, which game is better? However, if your goal is to improve your video game project, all others are friends, and you can learn from them. This book is an excellent opportunity to learn from fellow game developer. You can see and comprehend different and successful approaches to creating video games. Learning from book teachings, examples, coding methods, and shortcuts saves you a lot of creative time.

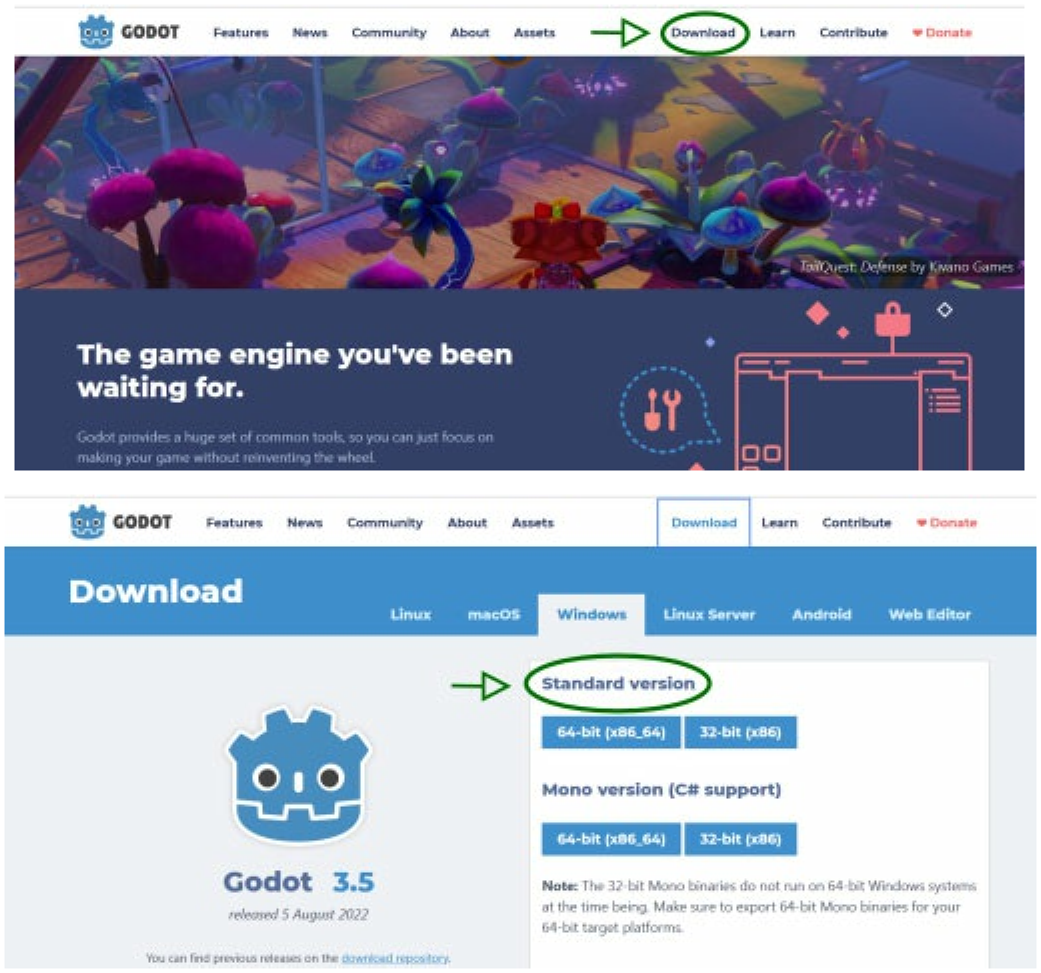
If you, for example, have Python or C# experience, this is helpful in learning and later in game development. Likewise, coding experience can be beneficial, and you will comprehend some coding techniques and solutions quickly. Moreover, with this book, you will get a lot of quality 2D and 3D game objects. Every game developer knows how to validate expensive game resources for free.

I will explain complex programming techniques in a way every beginner can understand. For example, in a first 2D game, there will be a game object generator with the system to remember every interaction. Complex game systems increase playability but need a lot of programming know-how and experience to make. Sometimes, it is good to act like you don't know and, in the learning process, get a lot of new stuff.

Encouraging at the beginning

I would like to encourage beginners with a *Hello World* example with Godot. For this, you will need to perform the following steps:

1. Use your browser, go to **https://godotengine.org/**. Click on the **`Download`** card for the download section to open in a browser (*Figure 1.1(a)*). In the **`Download`** section, select the operating system you use. Options are Linux, macOS, Windows, and Linux server (*Figure 1.1(b)*). For every OS, there is a standard and mono version (image). Then, select the standard version, 64-bit, if you have x86_64 configuration or 32-bit for x_86):



The game engine you've been waiting for.

Godot provides a huge set of common tools, so you can just focus on making your game without reinventing the wheel.

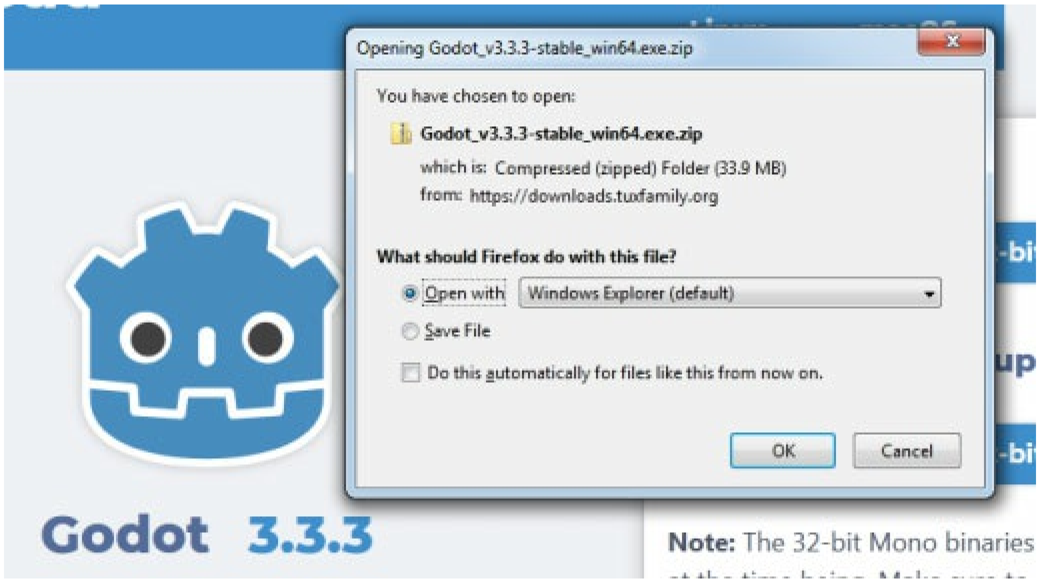
Note: The 32-bit Mono binaries do not run on 64-bit Windows systems at the time being. Make sure to export 64-bit Mono binaries for your 64-bit target platforms.

You can find previous releases on the download repository.

Figure 1.1: *(a) Godot portal (b) Download section*

2. After that, the browser will download a compressed(.zip) file with a game engine. You need to extract it with the application, usually with the option extract.

Note: Codes in the book will work best with the engine versions from 3.1 to 3.5.



Opening Godot_v3.3.3-stable_win64.exe.zip

You have chosen to open:

Godot_v3.3.3-stable_win64.exe.zip

which is: Compressed (zipped) Folder (33.9 MB)

from: https://downloads.tuxfamily.org

What should Firefox do with this file?

(●) Open with: Windows Explorer (default)

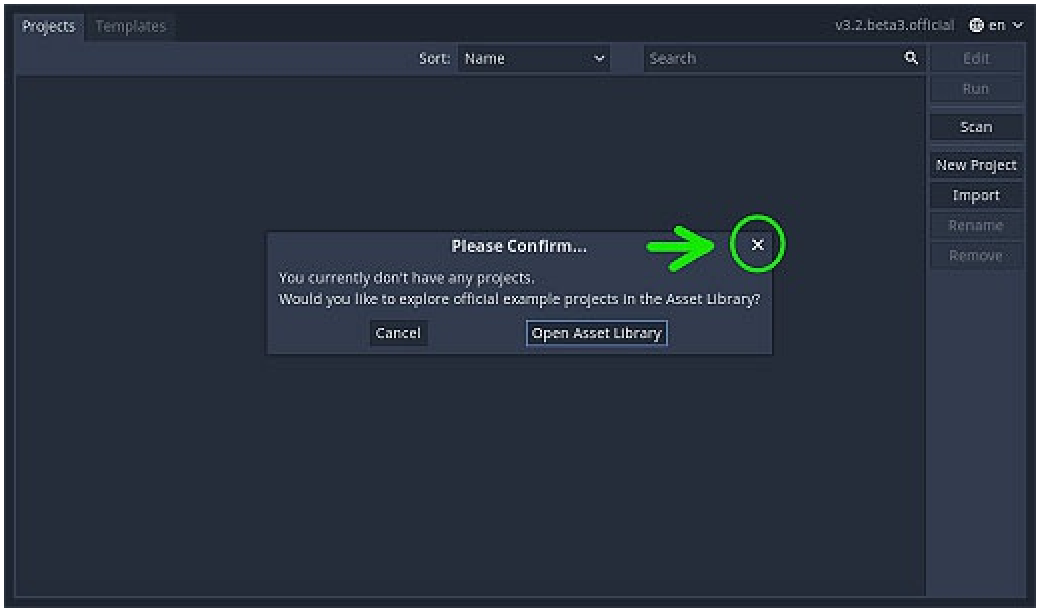
( ) Save File

[ ] Do this automatically for files like this from now on.

Figure 1.2: *Browser open dialog*

3. You will see the Godot file in some of your folders. You can create a desktop shortcut, move the file to another location, or start a game engine.
4. Start the game engine.

Now, you can start the game engine. First, what opens is the **`Projects`** window (Project manager). Then, you can create a new project, select a project for continuous working, or import a project to the project manager. A dialog window will be in the center of a project manager if you open a game engine for the first time, as shown in the *Figure 1.3*. Close the dialog window by clicking on a close micro button in the right-hand upper corner or click on **`Cancel`**.

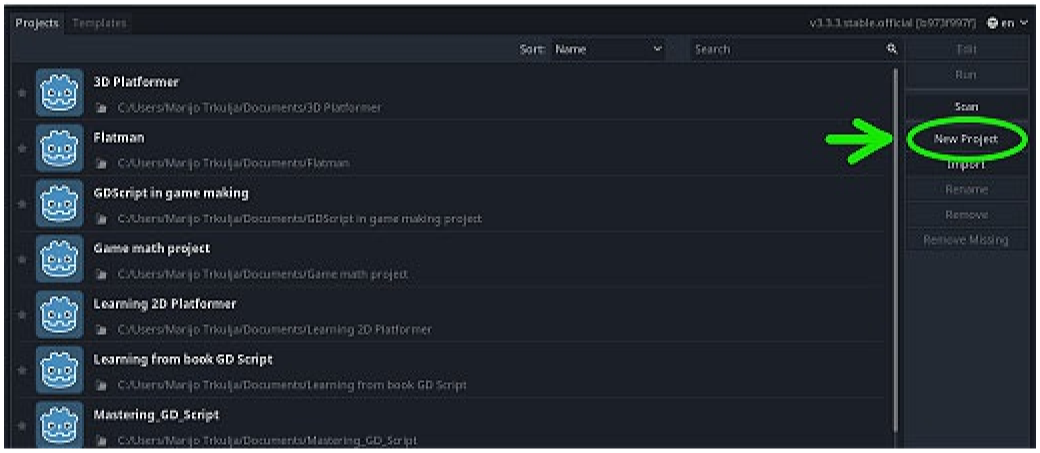


Please Confirm...

You currently don't have any projects.
Would you like to explore official example projects in the Asset Library?

Figure 1.3: Project window

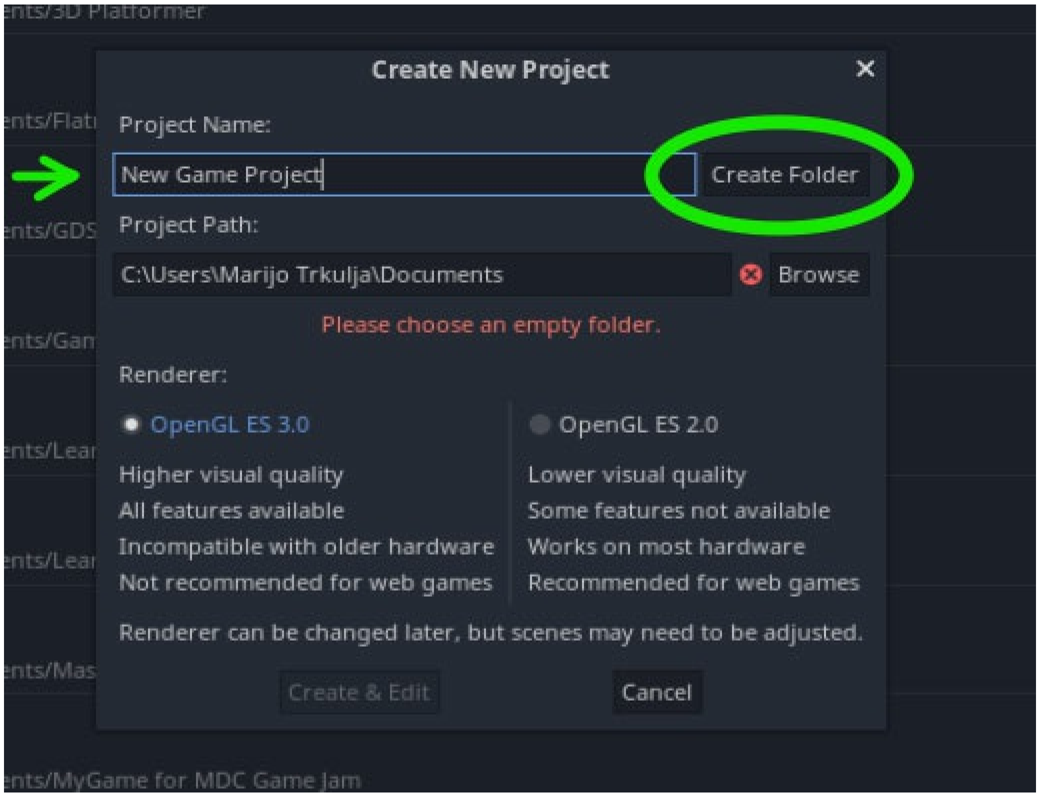
It will be good to start a new project at this stage of learning. It's possible to change the default language for a game engine (upper right), or you can change it later. So, you can select **`New Project`** in the project manager menu (*Figure 1.4*):



3D Platformer
C:/Users/Marijo Trkulja/Documents/3D Platformer
Flatman
C:/Users/Marijo Trkulja/Documents/Flatman
GDScript in game making
C:/Users/Marijo Trkulja/Documents/GDScript in game making project
Game math project
C:/Users/Marijo Trkulja/Documents/Game math project
Learning 2D Platformer
C:/Users/Marijo Trkulja/Documents/Learning 2D Platformer
Learning from book GD Script
C:/Users/Marijo Trkulja/Documents/Learning from book GD Script
Mastering_GD_Script
C:/Users/Marijo Trkulja/Documents/Mastering_GD_Script

Figure 1.4: Project manager menu

After that, the **`New Project`** dialog will open. First, you need to create a new project in an empty folder. For this, select by browsing your data or create a new one (*Figure 1.5*) in a project path. Also, it will be wise to change the default name for a game project. You can additionally select a renderer for a project. For this project, I suggest the OpenGL ES 3.0 renderer. A rendering engine has a Rendering Manager to send data to the GPU (graphic process unit) to initialize the correct shaders. Finally, you can click on **`Create & Edit`** to create a new game project:



Create New Project

Project Name:

New Game Project

Create Folder

Project Path:

C:\Users\Marijo Trkulja\Documents

Browse

Please choose an empty folder.

Renderer:

● OpenGL ES 3.0

Higher visual quality
All features available
Incompatible with older hardware
Not recommended for web games

○ OpenGL ES 2.0

Lower visual quality
Some features not available
Works on most hardware
Recommended for web games

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

Create & Edit

Cancel

Figure 1.5: *New project dialog*

Hello World example

We can create this example with only one game element called Label, but we will also add other things. For this to work, the game scene is essential, so we will create one. First, select a 2D game scene in the game scene menu (*Figure 1.6*). There are other options in the game scene menu (3D scene, user interface, custom node), and we will talk about them later in the book. Whenever you create a new scene, an appropriate root node is added. Later, when you make a new scene, go to the main menu **`Scene | New Scene`**:

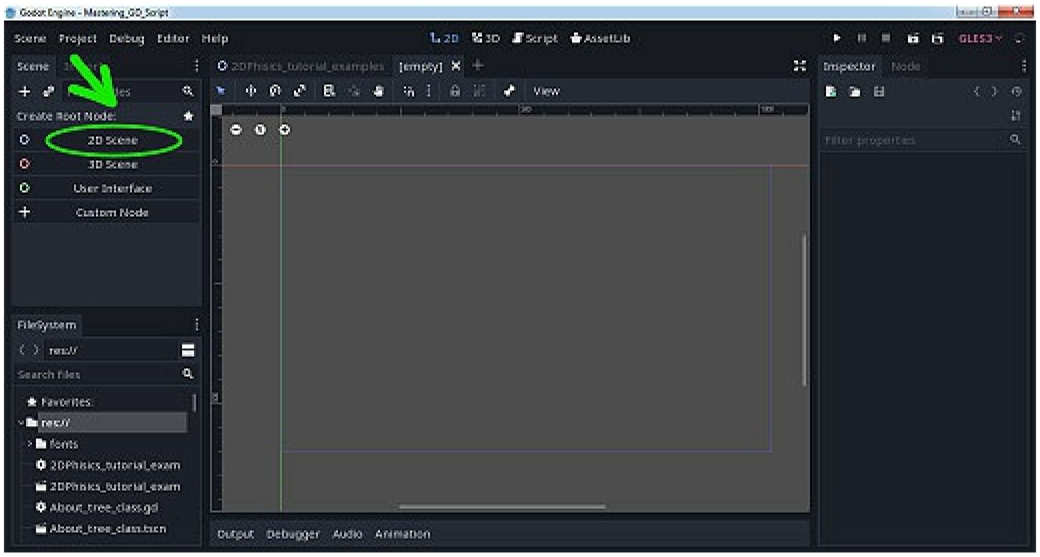


Figure 1.6: *New scene menu*

When you open a project window ([*Figure 1.7*]) for the first time, you will notice other elements too. For example, you will find many options for the Godot integrated developer environment (IDE) in the main menu:

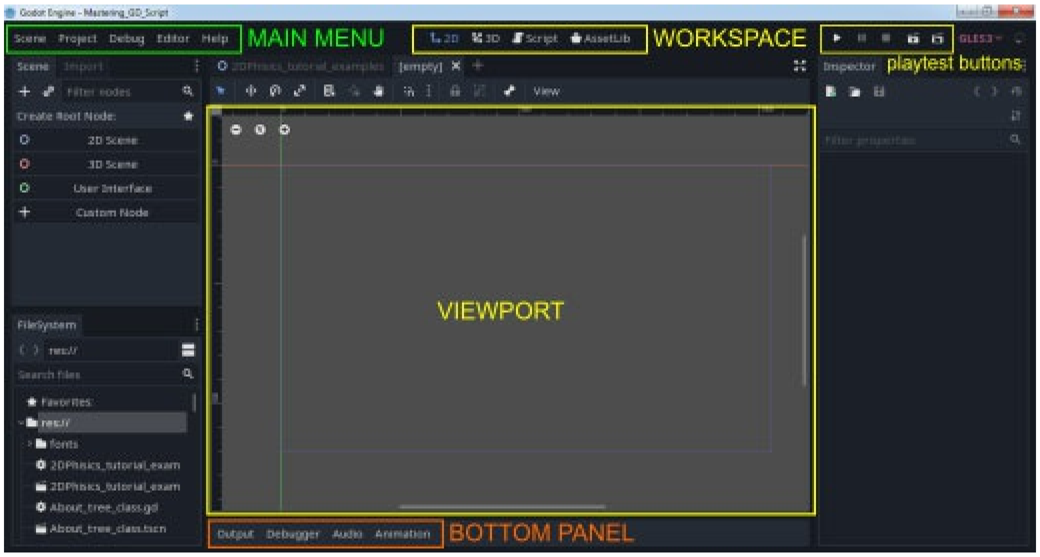


Figure 1.7: *Project window*

And now, let's back to our initial project:

- Add a node (game object), click on a large plus sign ([*Figure 1.8*]) or *Ctrl + A*.
- Write a **`label`** in a **`Search`** field and click on **`Create`**.
- You can add another node called button using the same principle. So,

select the root node, click on the plus sign, write **`button`**, and click on **`Create`**.

Suppose you want to change the node position, hold LMB and move it to the desired position:

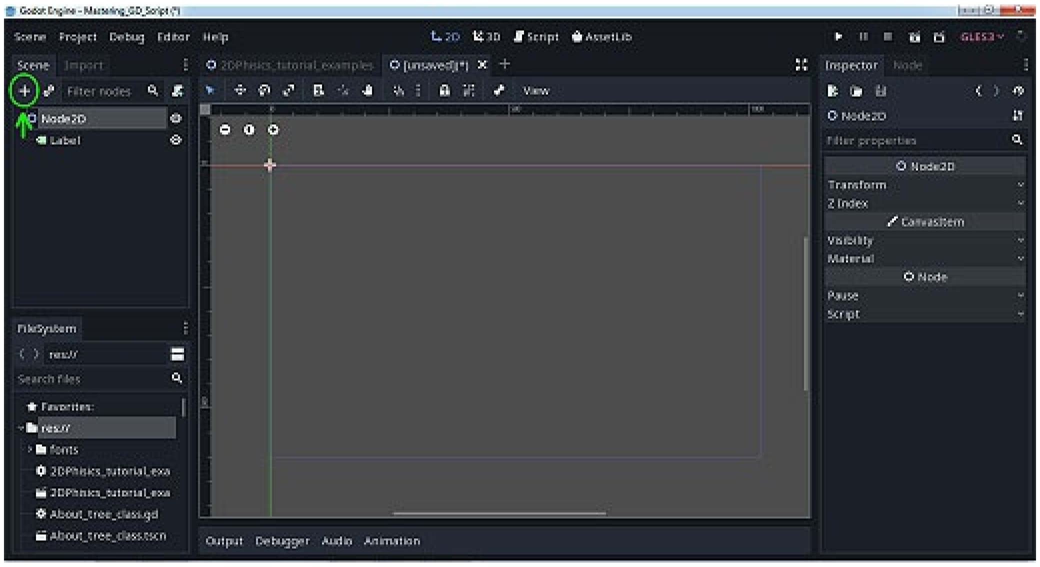


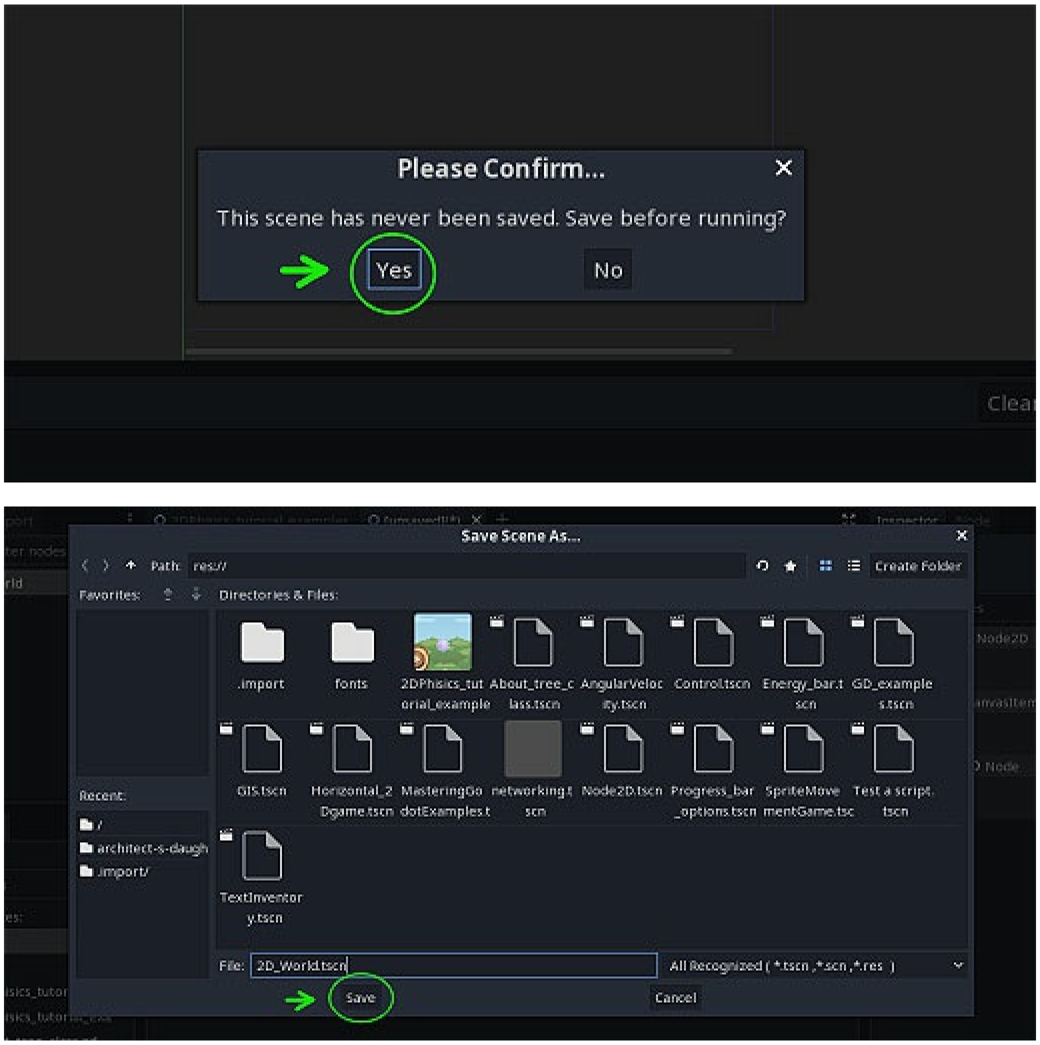
Figure 1.8: *Add a new node*

Now, you have a game project with one 2D game scene. In the game scene, you have three nodes: 2Dnode, label node, and button node.

If you are a game developer, work with a Godot IDE can be easy for you. Write text to label Hello World, create a signal for a button, and create a new script. In the function created by signal, set the text to button as Hello World, hello sky, and that's it.

Too hard (teachers are talking to beginners), do not worry—first, select the label on the left-hand side. Then, find the text property in the inspector window on the right-hand side of the Godot IDE. Write text Hello World, hello sky. and you are ok too.

To start a game scene, click on a **`Play scene`** button (upper right) or click *F6* on your keyboard. You played a scene for the first time, so you need to save it (*Figure 1.9(a)*). Next, accept the proposition by clicking on the **`Yes`** option. When the save dialog opens (*Figure 1.9(b)*), accept saving by clicking on the **`Save`** option. You will save the scene in a default **`res`** project folder. You can see the results of your efforts now:


Please Confirm...

This scene has never been saved. Save before running?

Save Scene As...

Path: res://

File: 2D_World.tscn

All Recognized (*.tscn ,*.scn ,*.res)

Figure 1.9: *(a) Save confirm dialog (b) Save scene dialog*

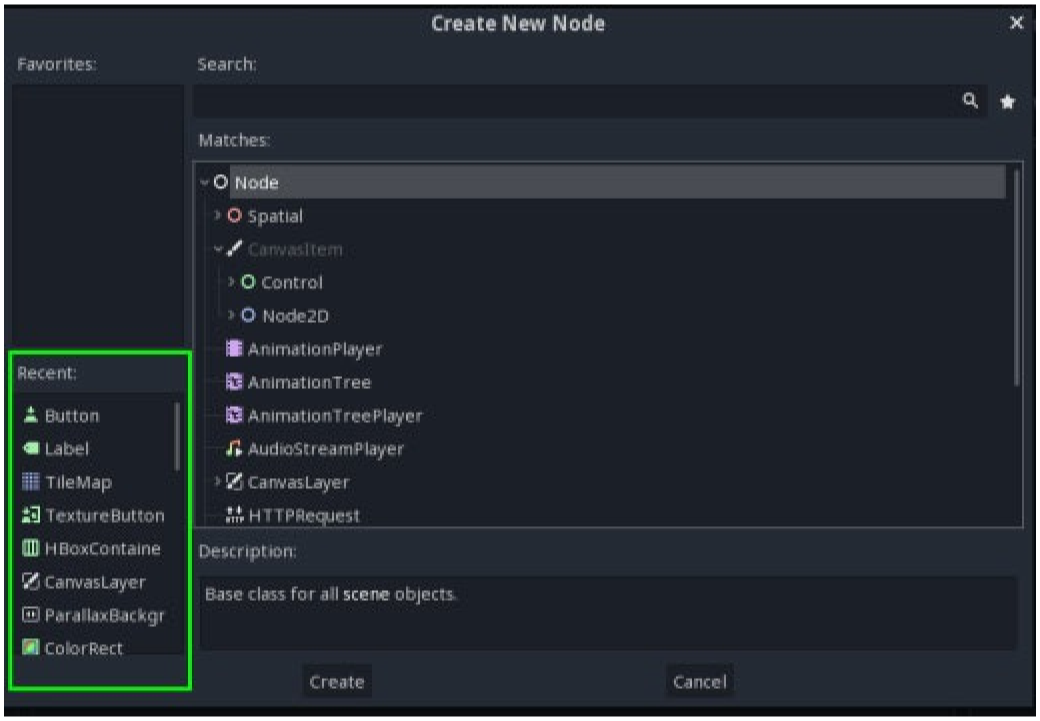
Close the play window, and if you want to change the place of a label or button in the game scene, select it and drag it. If you want to zoom in or out in a viewport, use the mouse scroll wheel. Hold and drag red dots (after selection) around the button node in a viewport to increase the button field.

Godot node system

When you create a game scene, you also get one initial game node. For example, for a 2D scene, you get node2D. When you make a 3D scene, you initially get a 3Dnode, and when you create a user interface scene, you get a control node. From this, you create your game scene, and all added nodes are child nodes.

So, when you add a child node, you can click on RMB (right mouse button) and select **`Add Child Node`** or click on *Ctrl* + *A*.

The created node can be selected in the left part of an **`Add Node`** dialog window as shown in the Figure 1.10. All 2D nodes are blue-colored in the **`Create New Node`** window:



Base class for all scene objects.

Figure 1.10

Your game objects are called nodes in Godot, and you can see them in a scene window. The first upper is called the root node. All others are child nodes of an upper one. All nodes are initially visible, but you can toggle visibility by clicking on a node circle button. For moving, first, select and then drag to the position.

Properties of a node are visible in an inspector window after selection. If you, for example, select the button node, you will see its properties in an inspector window on the left-hand side of a project window (*Figure 1.11*):

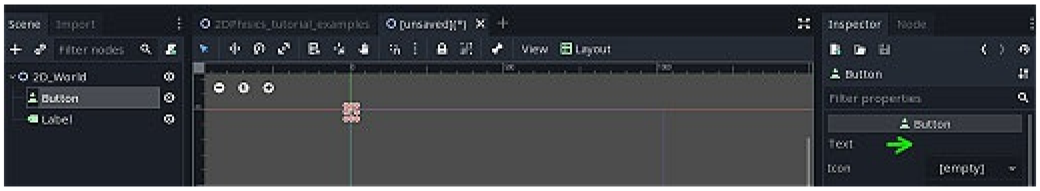
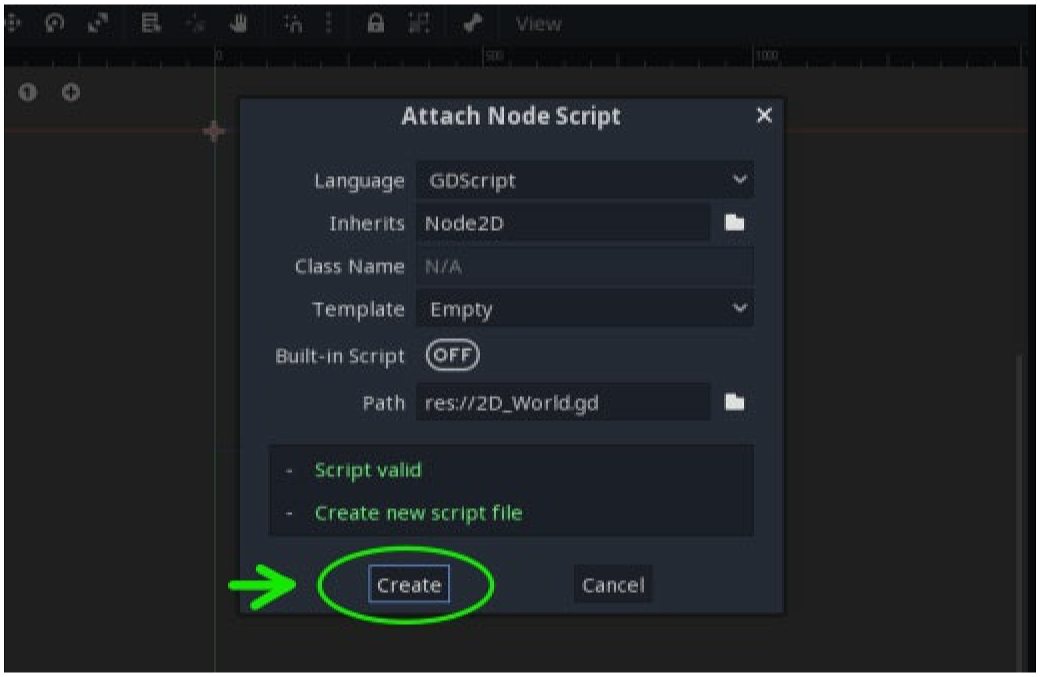


Figure 1.11: *Inspector window properties*

In the previous example, we added two child nodes, set a signal, and wrote a bit of code. Code! Don't panic, please. Coding can be much easy than you think. For this, we can use the GD Script.GD script

GD script

Now, let's change a root's node name. With this, working with many scripts will be much easy. Click twice on a root node and rename it to **`2D_world`**. For this, you can use the RMB and rename option. If you, for example, use RMB on a node, you will have the opportunity to add a script - attach a script. In the dialog window **`Attach Node Script`**, you can find the Nativ Script, GD Script, and Visual Script options (Figure 1.12). The GD Script is the default language for the Godot game engine, and you don't need to add or download anything for his computing. So, we suggest adding a GD script to your root node – node2D:



Language: GDScript
Inherits: Node2D
Class Name: N/A
Template: Empty
Built-in Script: OFF
Path: res://2D_World.gd
- Script valid
- Create new script file

Figure 1.12: *Attach node script dialog window*

Now, you can create a signal for a button node. First, select it and click on the inspector window's second tab - node/signals. Then, select the option **`Pressed`** and click on the **`Connect`** button. After that, in the dialog, click again on the **`Connect`** button.

Select the script in the workspace and change it. Look at the following

example:
```
func _on_Button_pressed():
  $Label.set_text( “Hello World, Hello Sky”)
  # In the above code set_text is a method of a label node
```
Very good till now.

It will be good to delete comments, so your script will as shown in the following example:
```
extends Node2D

func _ready():
  pass

func _on_Button_pressed():
  $Label.set_text( “Hello World, Hello Sky”)
```
Your root node is the extent of a class 2D node. Every class can store many more game objects (nodes) and their properties and methods. You can have a little rest now, and we can later continue with the class 2D node.

Class 2D node

2D node is a fundamental class for other 2D nodes and parent for all added child nodes in the same scene. Class is inherited from the node class and has the position, rotation, scale, and z-index properties.

Properties: **`position`**, **`rotation`**, **`rotation_degrees`**, **`scale`**, **`transform`**, **`global_position`**, **`global_rotation`**, **`global_rotation_degrees`**, **`global_scale`**, **`global_transform`**, **`z_index`**, **`z_as_relative`**.

Position

Position can be set and can be retrieved from the 2D object. For position setting, use **`set_position()`**, and to retrieve the position use **`get_position()`**. The **`Vector2`** variable needs to be defined for the **`set_position()`** method. When you use the variable with the **`get_position()`** method, the variable will be **`Vector2`**.

Example:
```
extends Node2D

func _ready():
  var pos = Vector2( 21, 21)
  self.set_position( pos)
```

Example comment: For example, **`self`** represents an extended node – Node 2D. **`var`** is used to define a variable. **`Vector2`** is a variable type for 2D space with horizontal and vertical space defined.

The first variable in vector2 is horizontal, and the second variable is vertical. **`func`** represents function – part of code. The function ready() automatically starts when you start a game scene, and you don't need to call it.

For example, changing a label position from the Hello World example will be something as shown in the following GD Script code:

```
extends Node2D
func _ready():
  var pos = Vector2( 120, 120)
  $Label.set_position(pos)
  # Put some spaces when using operators(like "=") and any data
  in brackets
```

Rotation

You can set and retrieve (get) rotation data from a 2D object. For rotation setting, use **`set_rotation()`**, and to retrieve rotation use **`get_rotation()`**. When you retrieve the rotation, the variable will be in radians. You can convert radians to degrees. The key is knowing that 180 degrees are equal to pi radians. Then, multiply the measurement in radians by 180 divided by pi. For direct work in degrees, see the rotation degrees property. In calculation, 1 radian = 57.2958.

Example:

```
# example code can be part of ready function
var rot = 1
self.set_rotation( rot)
var rot_deg = rot * 180 / PI
```

Example comment: In this example, you can see the symbol `#`. This symbol is for one-line comment. The example also includes characters for mathematical operations (* for multiplication / for division) and a logo for the pi constant. The constant represents how many times the diameter of a circle fits around its perimeter. It is approximately equal to 3.14159.

Good! Repetition and some tasks can be a good continuation after learning the two (position, rotation) node2D properties.

Repetition

We can create different game scenes in a Godot game engine and add a node (game objects). For example, for a 2D game, we make a 2D scene, and by default, we get node2D. 2Dnode is also a root node. To the root node, we can add child nodes and attach the script. The default Godot script is GD Script based on Python and Lua, but we can use the Native Script with C++ support, C#, or visual scripting with visually-oriented game objects.

In the default GD Script, we get some code, mainly comments, but a ready function also. This function is processed when we play a scene. The **`ready`** function is in many examples for that reason.

Tasks

For the first task, you can code a position for two nodes (label and button). The goal can be to set both positions as the same horizontal but different in a vertical part of a position. It will be good to use signalling and node's properties from the inspector tab.

And for the next task, you can code a rotation for one of your nodes.

Finish your tasks and then go back (after scale property) to see possible solutions.

Scale

For scaling game objects, use the scale property, the **`set_scale ()`** method for a set, and the **`get_scale()`** method to get the scale value. For example, move the default Godot sprite (select and drag with LMB) into a 2D scene for the following example to work. This way, the sprite node will be the child node for the 2D node.

Example:

```
extends Node2D

export var sca = Vector2( 0, 0)

func _ready():
  sca = Vector2( 3, 3)
  self.set_scale( sca)
  print( self.get_scale())
```

Example comment: Export var is good when we want to enable the value of a particular variable to be changed. It can be used when testing the code and initially defining values. For example, the print command prints a value in the output window.

Tasks solutions: I hope you are finished with tasks one and two. Now, you can check it with the provided solutions.

Task 1:

```
extends Node2D
func _ready():
  # Text for label and button
  $Label.set_text( “Hello World and “)
  $Button.set_text( “ Click me!”)
  # Setting position for a label
  var pos = Vector2( 120, 120)
  $Label.set_position( pos)
  # Setting position for a button
  var pos_2 = Vector2( 120, 150)
  $Button.set_position( pos_2)
  # on click button function
func _on_Button_pressed():
  $Label.set_text( “Hello World and Hello sky!”)
```

Task 2:

You need to add **`$. .set_rotation( -0.12)`** after the last code line in a ready function. With this code, your root node rotates for 0.12 radians. See the following code:

```
func _ready():
  $Label.set_text( “Hello World and “)
  $Button.set_text( “ Click me!”)
  var pos = Vector2( 120, 120)
  $Label.set_position( pos)
  var pos_2 = Vector2( 120, 150)
  $Button.set_position( pos_2)
  $”.”.set_rotation( -0.12)
```

With this know-how, youcan code a scale property for a label node text. Scale it at 1.5 factors.

Example:

```
  $Label.set_scale(Vector2( 1.5, 1.5)) # scale for label
  $”.”.set_scale(Vector2( 1.5, 1.5)) # scale for label and button
```

If your project is still non-functional, you can copy-paste solutions from previous examples. Your Hello World can look like *Figure 1.13*:

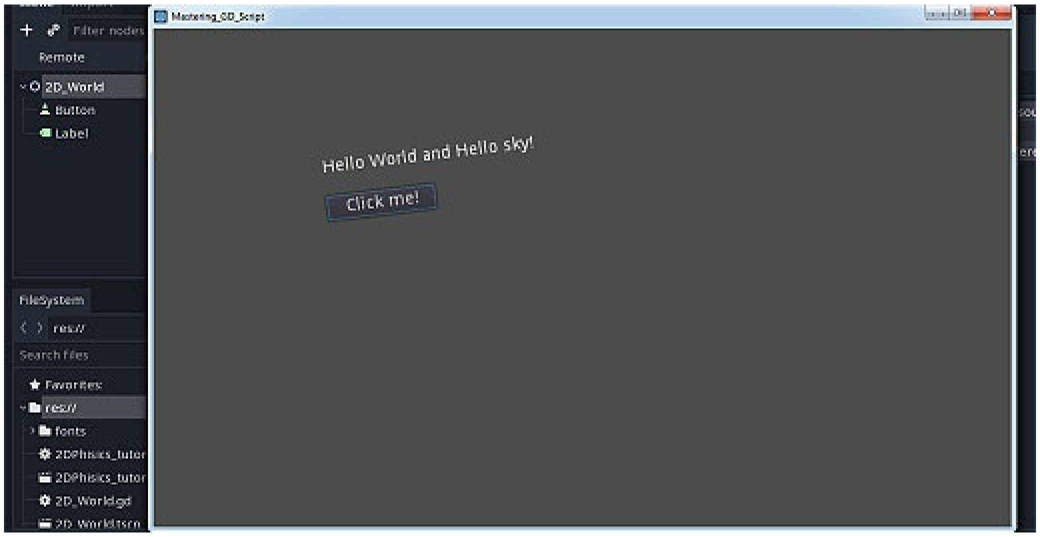


Figure 1.13: *Hello World*

As you can see, we learned a lot about Godot IDE and GD Script. You even know about node properties, and you successfully created your first Godot project. Knowledge about labels, buttons, and signal-making will be helpful in the next chapter.

CHAPTER 2

Towards 2D Game

We made a significant step in the previous chapter, and now we can continue. As a creator, you need tools for shaping your vision. In this situation, tools are properties of node2D and some GD Script syntax. We can continue learning the other properties of node2D, but I would like to suggest some different approaches. First, you will read about essential GD Script coding concepts (while, for, array, if, dictionary). Just read about it with a purpose to widen your knowledge necessary for game projects. After that, we can continue with node2D properties and game projects.

This reading is suitable for experienced developers as a remainder, but for a beginner, it can be challenging at first. You will see codes like this:

```
var arr = [ 0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39,
42, 45]
```

This is an array; a way to temporarily store data for computing. The **`var`** means that this is variable or changeable data. If you, for example, find a variable like the following:

```
var game character = “Strong, confident and almost unbreakable.”
```

Preceding is a string variable, a variable with letters, and other characters. I think this can help in a reading of text about programming essentials.

GDScript in programming

From this chapter you will learn how to program through real-life examples. Every example has a possible implementation. We will start with simple examples, but examples will become more complex over time. This chapter is helpful for beginners and suitable as a reminder for any game developer.

While loop

`while` is a loop that repeats as long as the condition is true.

```
while condition
  some code
```

Let's look at an example of a **`while`** loop.

Example:

```
onready var a = [0,1,3,2,5,7,3,4,5,0,5,6,7,2]

func _ready():
  while typeof(a[n]) == TYPE_INT :
    print(a[n])
    n += 1
```

In this example, code prints the contents of a string if the content type is numeric. The **`typeof`** method is to check the type of content. In addition, you can use the **`typeof`** method to filter large groups of data.

Possible type labels for this method are:

`TYPE_NIL = 0`: Variable is of type **`nil`** (only applied for null).

`TYPE_BOOL = 1`: Variable is of type **`bool`**.

`TYPE_INT = 2`: Variable is of type **`int`**.

`TYPE_REAL = 3`: Variable is of type **`float/real`**.

`TYPE_STRING = 4`: Variable is of type **`String`**.

`TYPE_VECTOR2 = 5`: Variable is of type **`Vector2`**.

`TYPE_RECT2 = 6`: Variable is of type **`Rect2`**.

`TYPE_VECTOR3 = 7`: Variable is of type **`Vector3`**.

`TYPE_TRANSFORM2D = 8`: Variable is of type **`Transform2D`**.

`TYPE_PLANE = 9`: Variable is of type **`Plane`**.

`TYPE_QUAT = 10`: Variable is of type **`Quat`**.

`TYPE_AABB = 11`: Variable is of type **`AABB`**.

`TYPE_BASIS = 12`: Variable is of type **`Basis`**.

`TYPE_TRANSFORM = 13`: Variable is of type **`Transform`**.

`TYPE_COLOR = 14`: Variable is of type **`Color`**.

`TYPE_NODE_PATH = 15`: Variable is of type **`NodePath`**.

`TYPE_RID = 16`: Variable is of type **`RID`**.

`TYPE_OBJECT = 17`: Variable is of type **`Object`**.

`TYPE_DICTIONARY = 18`: Variable is of type **`Dictionary`**.

`TYPE_ARRAY = 19`: Variable is of type **`Array`**.

`TYPE_RAW_ARRAY = 20`: Variable is of type **`PoolByteArray`**.

`TYPE_INT_ARRAY = 21`: Variable is of type **`PoolIntArray`**.

`TYPE_REAL_ARRAY = 22`: Variable is of type **`PoolRealArray`**.

`TYPE_STRING_ARRAY = 23`: Variable is of type **`PoolStringArray`**.

`TYPE_VECTOR2_ARRAY = 24`: Variable is of type **`PoolVector2Array`**.

`TYPE_VECTOR3_ARRAY = 25`: Variable is of type **`PoolVector3Array`**.

`TYPE_COLOR_ARRAY = 26`: Variable is of type **`PoolColorArray`**.

`TYPE_MAX = 27`: Marker for end of type constants.

The following example prints only numeric values if they are non-zero. Numbers from one-dimensional array prints up to zero.

```
while typeof(a[n]) == TYPE_INT && a[n] != 0:
  print(a[n])
  n += 1
```

How would you check if the data type is a vector? For example, you would type vector2 or vector3. Let's look at the following example.

```
while typeof(a[n]) == TYPE_VECTOR2:
# or while typeof(a[n]) == 5:
```

With the **`break`** command, you can stop executing the while loop. You can see the conditional branching (**`if`**) and the **`while`** loop in the example.

Example:

```
func code_test():
  var a = 0
  while a < 7:
    print(a)
    a += 1
    if a == 5:
      print("Make a break at 5")
      break
```

Note: Python allows the else command to be in the while loop.

Use the **`continue`** command to stop the current iteration and continue with the next.

Example:

```
func code_test():
  var a = 0
  while a < 7:
    a += 1
    if a == 5:
     print(“Continue after 5”)
     continue
  print(a)
```

for loop

for loop repeats within the defined range from-to. When using an array or a dictionary, use the name as in the following example:

```
 onready var a = [0,1,3,2,5,7,3,4,5,0,5,6,7,2] # Array

 for i in a:
  print(a[i]) # Prints array elements
```

The following example shows how to add up all the numerical elements of a single array. The example uses a **range** function. The **len** command is to get the number of elements in an array or a string:

```
 var sum = 0
 for i in range( 1,len(a) ):
  sum += a[i]
```

Rank can have a defined step; in the example, every third element in the rank prints.

```
 func code_test():
  var i = 0
  for i in range(0,30,3):
   print(i)
```

Only numbers that we want to use can be defined in a rank. For example, to print only certain elements of a string:

```
 func code_test():
  var i = 0
  for i in [6,12,3,9]:
   print(a[i])
```

In addition, you can type in the string name and use its elements (letters) through the **for** loop:

```
 func code_test():
  var hello = “Hello GD”
  for i in hello:
   print(i)
```

Use the **continue** command to stop the current iteration and continue with the next:

```
 func code_test():
  var hello = “Hello GD”
  for i in hello:
   if i==” “:
     continue
 print(i)
```

If else

Conditional branching using the if command is an old programming technique. In addition to this command, **`elif`** and **`else`** commands are also used. Let’s look at examples.

```
if typeof(a[n]) == TYPE_STRING && a[n]==””:
  n += 1
```

The example uses comparison operators such as the **`==`** character; let’s look at what other comparison operators can be:

`==` Equals

`>` Greater than

`<` Less than

`>=` Greater or equal

`<=` Less or equal

`!=` Not equals

The logic operator **`&&`** is also used in the example; let’s look at the other logical operators:

`||` Or Boolean or

`!` Not Boolean not

Now, let’s explain the current example. Conditional branching checks the data type from a one-dimensional string is a string. It also prevents that the string is content-free.

If the conditions are met, the counter is increased by one. The following example shows the use of an **`elif`** command:

```
func code_test():
  var hi = “Hi GD”
  var ln = len(hi)
  if ln >=5:
  print(“Five or more letters”)
  elif ln <=4:
   print(“Four or less letters”)
```

See also an example for using the **`else`** command:

```
func code_test():
  var hi = “Hi GDScript”
  var ln = len(hi)
  if ln < 6:
   print(“Less than six letters”)
  else:
   print(“Six or more letters”)
```

If and else can be used in conditional expressions, let us see an example.

Example:

```
 var is_nice = true
 var state = “nice” if is_nice else “not nice”
 print(state)
```

Match

Using the **`match`** command, we have the possibility of conditional branching. This type of conditional branching is specific, and there are different ways to use it. Example shows use with numeric values.

Example:

```
 onready var a = [0,1,3,2,5,7,3,4,5,0,5,6,7,2]
 onread var modes = [0,0,0,0,0,0,0,0]

 func ready():
  for i in a:
   match a[i]:
    1:
     modes[1] += 1
    2:
     modes[2] += 1
    3:
     modes[3] += 1
    4:
     modes[4] += 1
    5:
     modes[5] += 1
    6:
     modes[6] += 1
    7:
     modes[7] += 1
  print(“Modes are: “ + str(modes))
```

The program code in the example calculates the mod value for the elements of a one-dimensional array. So if you have the same code for your different match options, you can set it up like this.

Example:

```
  for i in a:
   match a[i]:
    1,2,2:
     print(“Variable 1,2 or 3 found”)
```

The following example will show you how to use a match for counting different types of data in an array.

Example:

```
func code_test():
  var data = [“text”,33,42,6,”e”]
  var string = 0
  var ints = 0
  for i in range(0,len(data)):
   match typeof(data[i]):
    TYPE_STRING:
     string += 1
    TYPE_INT:
     ints += 1
  print(“Strings: “ + str(string))
  print(“Integers: “ + str(ints))
```

Function

In the following examples, we will see how the function is used. Let’s first see how to declare a function with a single parameter. This function will sum all the numeric values in the array:

```
func calc_sum(sum):
  var summ = 0
  for i in sum:
   summ += sum[i]
  print(summ)
```

The function starts via the function name; the parameters for the function are in parentheses:

```
onready var a = [0,1,3,2,5,7,3,4,5,0,5,6,7,2]

func _ready():
  calc_sum(a)
```

The function may contain code without parameters. This can be used when we repeatedly need to perform a specific activity:

```
func calc_sum() → void: # void when return isn’t used
  var summ = 0
  for i in a:
   summ += a[i]
  print(summ)
```

The function can return a value, then the return command is used. See an example.

Example:

```
func _ready():
  var sum = calc_sum()
  print(“Array elements sum is: “ + str(sum))
func calc_sum():
  var summ = 0
  for i in a:
   summ += a[i]
  return summ
```

In the following example, you will see the sum of two numbers; the function returns a numeric int value.

Example:

```
func _ready():
  var sum = calc_sum(21,45)
  print(“Sum is: “ + str(sum))

func calc_sum(a,b) ->int :
  return a + b
```

Let’s look at other types of data to restore function values:

`Void` When function didn’t return value.

`String` Function return string value.

`Array` Array value.

`Dictionary` Dictionary data type value.

Array

A matrix or array can contain different data groups. Let’s look at how to create a one-dimensional matrix:

```
onready var arr = [] # Create all types array
onready var arr = Array() # create all types array
onready var arr = PoolIntArray() # integer array
```

Let’s see what other types of data can be stored by specially designed arrays.

```
PoolByteArray integers from 0 to 255
PoolColorArray color
PoolRealArray float
PoolStringArray string
PoolVector2Array vector2 for 2D space
PoolVector3Array vector3 for 3D space
```

With the append method, an element is in the array. If the string is empty, the method adds a new element at the zero position. If there are elements in the array, a new one adds after the last one:

```
arr.append(6)
```

Example:

```
onready var arr = PoolIntArray()

func _ready():
 arr.append(6)
 print(arr)
```

You can define default values in an array individually or for multiple elements at once.

```
func _ready():
 for i in range(0,15):#multiple elements with for loop
  arr.append(0)
```

Array elements can be defined when declaring the array itself.

```
onready var arr = [39,51,36]
```

The **`count`** command lets you count the number of the same number elements in a row. The example uses **`randomize`** and **`randi`** to generate random numbers:

```
func _ready():
 randomize()
 for i in range(0,101):
  var rnd = randi() % 100
  arr.append(rnd)
 print(arr.count(36))
```

The **`sort`** command is to sort the numeric or string (alphabetical) data:

```
onready var arr = [39,51,36]
func _ready():
 randomize()
 for i in range(0,101):
  var rnd = randi() % 100
  arr.append(rnd)
  arr.sort()
 print(arr)
```

You can use the **`find`** command to find a specific value within an array:

```
func _ready():
 randomize()
 for i in range(0,101):
  var rnd = randi() % 100
  arr.append(rnd)
  var rnd = randi() % 100
 print(arr)
 if arr.find(rnd):
  print(“Number “ + str(rnd) + “ is found in an array!”)
```

There are many other commands for working with an array. Let’s look at the comments and examples.

Example:

```
onready var arr = [39,51,36]
func _ready():
  randomize()
  arr.insert(1,93) # insert element into array
  print(arr[1]) # print element at position
  arr[2] = 33 # declare value of element
  print(arr)
  arr.shuffle() # randomly change elements in array
  print(arr)
```

You can define an array with different element types:

```
onready var arr = [39,51,36,”some text”,”&”,15]
```

Different type arrays can be filtered into arrays containing only one data type:

Example:

```
onready var arr = [39,51,36,”some text”,”&”,15]
func _ready():
  var int_arr = []
  var str_arr = []
  for i in range(0,len(arr)):
   if typeof(arr[i]) == TYPE_INT:
    int_arr.append(arr[i])
   if typeof(arr[i]) == TYPE_STRING:
    str_arr.append(arr[i])
  print(int_arr)
  print(str_arr)
```

Note: When working with larger data groups, it is good that the data belong to the same type.

Dictionary

Dictionary allows you to store different types of data. Each data or data group is under a specific key. When declaring, the key is the first data, then the colon, followed by the value:

```
onready var dict = {1: 9, 2: 24, 3: 9}
```

Let’s look at an example that prints the contents of a dictionary:

```
onready var dict = {1: 9, 2: 24, 3: 9}
func _ready():
 print(dict)
```

The following example lists the size, keys, and values of a dictionary:

```
onready var dict = {1: 9, 2: 24, 3: 9}
func _ready():
 print(dict.size()) # how many elements are in
 print(dict.keys()) # list of keys
 print(dict.values()) # list of values
```

You can change the key-value if the key exists, or you can add a new key and value after the last entry:

```
func _ready():
 dict[1] = 6 dict[“new_key”] = “Text value”
 print(dict)
```

One dimensional matrix can be part of the dictionary. Let’s see an example of it.

```
onready var dict = {1: 9, 2: 24, 3: [9,18,24,30]}
func _ready():
 print(dict[3]) # print values of key
 print(dict[3][2]) # print element of array under key
```

See how you can create default values for a dictionary:

```
onready var dict = {}
func _ready():
 for i in range(0,9):
  dict[i] = 0
```

Creating a numeric array within the dictionary:

```
onready var dict = {}
func _ready():
 for i in range(0,9):
  dict[i] = 0
  dict[3] = [3,9,15,18,21]
 print(dict)
```

Creating a string array within a dictionary.

```
onready var dict = {}
func _ready():
 for i in range(0,9):
  dict[i] = 0
  dict[4] = [“alfa”,”beta”,”delta”]
 print(dict)
```

In the following example, you will see how to create a dictionary in the dictionary

Example:

```
onready var dict = {}
func _ready():
  for i in range(0,9):
    dict[i] = 0
    dict[5] = {“first”:”very good”,”second”: “good”, “third”:
    “acceptable”}
  print(dict)
```

The following example shows how to access values within an array or dictionary part of a unique dictionary.

Example:

```
onready var dict = {}
func _ready():
  for i in range(0,9):
    dict[i] = 0
    dict[3] = [3,9,15,18,21]
    dict[4] = [“alfa”,”beta”,”delta”]
    dict[5] = {“first”:”very
    good”,”second”:”good”,”third”:”acceptable”}
  print(dict[3][3])
  print(dict[4][0])
  print(dict[5][“first”])
```

Class 2D node (scale, transform, and global)

The 2D Node is Godot’s primary game object for all 2D child nodes. Properties of 2D node are scale, transform, and global.

Scale

You can scale a game object in 2D space with the scale property. The **`set_scale ()`** method is for the set, and the **`get_scale ()`** method gets the scale value. For the following example to work, move the default Godot sprite into 2D scene space. Select it in a filesystem micro window, and drag it to the game scene. This way, the sprite node will be the child node for the 2D node.

Example:

```
extends Node2D

export var sca = Vector2( 0, 0)
func _ready():
  sca = Vector2( 3, 3)
  self.set_scale( sca)
  print(self.get_scale())
```

Example comment: Use the export var when you want to enable the value of a particular variable to be changed. It can be used when testing the code and initially defining values. For example, the print command prints a value in the output window.

Transform

Transform allows you to define the position and rotation in 2D space. This property uses the transform2D variable defined by two parameters. Rotation(in radians) and position in 2D space. Use **`set_transform()`** method to set transform, and **`get_transform()`** to retrieve transform.

Example:

```
extends Node2D

func _ready():
  transform_object(1,Vector2( 60, 60))
func transform_object(rot,pos):
  var trans = Transform2D(rot,pos)
  self.set_transform(trans)
  var for_print = self.get_transform()
  print("Transform2D: " + str(for_print))
```

- **Example comment:**
- In the example, we have a user-defined function. The function initializes from the **`ready`** function. The User function has two parameters later set in the transform2D. When you want to print some text via the **`print`** command, the text should be in quotation marks. Use the **`+`** character to connect parts of the text. Use the **`str()`** command to translate numeric values into text.

Global position

Position in scene can be set and can be retrieved from the 2D object. For position setting, use **`set_global_position()`**, and to retrieve position use **`get_global_position()`**. Vector2 variable needs to be defined for the **`set_global_position()`** method. When you use the variable with the **`get_global_position()`** method, the variable will be **`vector2`**.

Example:

```
extends Node2D
const global_pos = Vector2( 18, 18)
```

```
func _ready():
  self.set_global_position(global_pos)
  var get_pos = self.get_global_position()
  print(get_pos)
```

Example comment:

The **Const** are good to declare a constant variable.

Z index

Z index or rendering order determines in which layer a particular object is. To set the z index, use the **`set_z_index()`** method. Use the **`get_z_index()`** method to get the z index values. The default value is 0, higher values show objects above, and lower values show objects below default values.

Example:

```
extends Node2D
func _ready():
  var get_zin = self.get_z_index()
  match get_zin:
    0:
     print(“Default value.”)
    1:
     print(“This object is above”)
    -1:
     print(“This object is below”)
```

Example comment:

In this example, we use a conditional branch – the match. When using this conditional branching, you first define the variable (for example, **`get_zin`**) and then the options. If the option is valid, then you can enter a specific code (for example, **`print`**).

Repetition

In this chapter, you learn about various programming techniques and concepts. Some of GD Script commands are often in-game making, others are not.

For example, **`IF`** as the GD Script command is part of every video game coding. “Array” command is also often in game making. In the following example, **`IF`** conditional branching determines the solution path for some conditions:

```
var materials = [ “initial value”, “planks”, “tools”]
var have_material = [ “initial value”, “yes”, “yes” ]
if have_material[ 1] == “yes” and have_material[ 2] == “yes”:
   # have materials for a build
   print(“You have “ + materials[ 1] + “ and “ + materials[ 2] +
   “ for build.”)
```

In the example, data are stored in an array and checked with **`IF`** conditional branch.

One of the next explained commands is the **`FOR`** loop. When you combine **`if`** and **`for`**, a lot of data can be filtered easily. Let’s take a look at this example:

```
  var materials = [ “initial value”, “planks”, “tools”, “wood”,
  “metal”]
   for f in range( 1, 4):
    if materials[ f] == “wood”:
     print(“You have wood in materials list.”)
```

The **`Else`** is part of the **`IF`** conditional branch and also can be helpful. First, you set a condition with an **`IF`** part and everything that doesn’t compute with **`IF`** will be resolved in the **`ELSE`** part:

```
var materials = [ “initial value”, “planks”, “tools”, “wood”,
“metal”]
   for f in range( 1, 4):
    if materials[ f] == “wood”:
     print(“You have wood in materials list.”)
    else:
     print(“Non wood material is “ + materials[f])
```

Many video games have few commands only, but it’s good to have a broader programming knowledge.

It’s wise to create the functional code with fewer code lines. Functions can make every code compact and readable. When you repeat a part of code more than twice, consider using a function.

We also explain a match conditional branch. Although, many game developers don’t use this command, the **`match`** still has many advantages.

Let’s look at the following example:

```
var materials = [ “initial value”, “planks”,
“tools”,”wood”,”metal”]
  for f in range( 1, 5):
   match materials[ f]:
    “planks”:
     print(“Can do many things with it.”)
    “tools”:
     print(“What can I do without a tools.”)
    “metal”:
     print(“Useful material too.”)
```

With all this information, you are ready for some actual game coding in the next chapter. You will learn about 2D game math and game objects called nodes. So take a little rest, and we will continue with some exciting parts of our learning journey.

CHAPTER 3

Making 2D Games

2D Math

This chapter can start with some good news. First, with a dedicated 2D engine Godot is one of the best 2D game engine from 2020. Second, the author (Marijo Trkulja) of these lines is Godot teacher with 4 books already written (2021) and more than 12 courses made. The tutorial "Learn to make commercial video games – Godot mega tutorial"(2022) is currently the largest tutorial on the subject with more than 24 hours of video content.

What is a video game math?

Game math is all about computing like mathematical operations (see the following example).

Example:

```
var sword = 12
var shield = 12
var warriors
warriors = (sword + shield) / 2
```

But what is 2D games math?

2D Game Math is also about computing, but computing in 2D space using vector2 and other variable types.

Every vector2 variable has two operators, one (X-axis) for horizontal and the other (Y-axis) for the vertical part of the vector2 variable. For example, you can set the position for a sprite image we put into the 2D scene:

```
$icon.set_position( Vector2( 120, 120))
```

To change the position, set the position again, or use the **`translate`** property for vector2 translation (vertical/horizontal):

```
$icon.translate( Vector2( 100, 0))
```

Controlled movement

With this know-how, you can create a controlled movement. For example, you can create a timer game node. You can set the wait time to 0.3, and check auto-start. After that, you can make a signal for the **`timeout`** property. The code for the generated function is as follows:

```
func _on_Timer_timeout():
  $icon.translate(Vector2( 3, 0))
  if $icon.get_position().x > 150:
   $Timer.stop()
```

The code means that the sprite image translates horizontally for 3 pixels every 0.3 seconds until the position is greater than 150 pixels.

If you use **`set_position`**, the code looks like the following:

```
onready var n = 0
func _on_Timer_timeout():
  $icon.set_position(Vector2( 120 + n, 120))
  n += 1
  if $icon.get_position().x > 150:
   $Timer.stop()
```

It has one more option just to see how many possibilities you have for the same coding activity:

```
func _on_Timer_timeout():
  $icon.move_local_x(3)
  if $icon.get_position().x > 150:
   $Timer.stop()
```

You can create the controlled movement with the initial rotation using transform2D.

Example:

```
func _on_Timer_timeout():
  var trans = Transform2D( 0.1, Vector2( 120 +n, 120))
  $icon.set_transform(trans)
  n += 3
  if $icon.get_position().x > 150:
   $Timer.stop()
```

First game props

And now, something interesting. First, let's create a function for making game props—sprites, in our situation. For this, we can create one user function with a **`Vector2`** parameter:

```
func create_props( pos_var):
  var sprite = Sprite.new()
  sprite.set_texture( texture)
  sprite.set_scale( Vector2( 0.3, 0.3))
  sprite.set_position( pos_var)
  .add_child( sprite)
```

You can call this function from **`ready`**:

```
func _ready():
  create_props(Vector2( 100, 10))
Let’s create 12 props with the exact distance between them:
for i in range( 10, 360, 30):
  create_props(Vector2( 10 + i, 10))
```

And what if we, for example, want to create a different distance between game props? We can use random number generators, so let’s learn about them.

Rand range

The rand range method generates a random float value based on parameters. Two parameters determine the range for generating a random numeric value.

Example:

```
func _ready():
  var rnd
  for i in range( 0, 9):
   rnd = rand_range( 0, 9)
   print( rnd * 100)
```

Rand seed

Random from seed: We can pass a seed and an array with both the number and new seed in return. The seed here refers to the internal state of the pseudo-random number generator. The internal state of the current implementation is 64 bits.

Example:

```
func _ready():
  var random_seed = rand_seed( 9)
  seed( random_seed.hash())
  print( randi())
```

Randf

Randf returns a random floating-point value on the interval [0, 1].

Example:

```
func _ready():
  print(randf())
```

Randi

Randi returns a random unsigned 32-bit integer.

Example:

```
func _ready():
  randomize()
  print(randi()) # random integer between 0 and 2^32 – 1
  print(randi() % 100 + 1) # random integer between 0 and 100
```

Randomize

Randomize randomizes the seed (or the internal state) of the random number generator. The current implementation reseeds using a number based on time.

Example:

```
func _ready():
  randomize() # Randomizes the seed
  print(randi() % 27) # random integer between 0 and 26
```

Good. We can now create a different distance between game props with a random number generator:

```
randomize()
  for i in range( 10, 360, 30):
   var rnd_no = randi() % 11 + 1
create_props(Vector2( 10 + i + rnd_no, 10))
```

2D Physics

The physics in 2D game space under a Godot 2D engine depends on the dedicated 2D engine.

Developers can use different types of 2D bodies and tile-maps to make 2D games. For static game objects, StaticBody2D is good; for a moveable character, KinematicBody2D is an option, and for a game object with 2D Physics capabilities, RigidBody2D. So, let’s learn about 2D bodies game elements.

Class static body 2D

StaticBody2D is for 2D Physics, not intended to move, and is suitable for objects in the environment such as walls or platforms.

Its properties are as follows: constant linear velocity, constant angular velocity, friction, bounce and physics material override.

Constant linear velocity

Method constant linear velocity for the 2Dbody does not move the body but affects colliding bodies. Add the **`StaticBody2D`** node for this example (write StaticBody2D in the **`Add node`** dialog window). Also, add **`CollisionShape2D`** as a child node. You need to set the collision shape to a rectangle shape (inspector window, shape property). Use **`RigidBody2D`** to test examples (add it as a node). Add the circular collision shape for a rigid body. Don't forget to enable visible collision shapes (main menu Debug).

Example:

```
 func _ready():
  $StaticBody2D.set_position( Vector2( 0, 300))
  $StaticBody2D.set_constant_linear_velocity( Vector2( 9, 9))
  $StaticBody2D/CollisionShape2D.shape.set_extents( Vector2( 240,
  12))
```

Constant angular velocity

The 'constant angular velocity' method does not move the body but affects colliding bodies.

Add the StaticBody2D node to this example. Also, add **`collisionshape2D`** as a child node. Set the collision shape to a rectangle shape. Use RigidBody2D to test examples. Add the circular collision shape for a rigid body. Don't forget to enable visible collision shapes (menu Debug). Try different examples, each with other properties.

Example:

```
 func _ready():
   $StaticBody2D.set_constant_angular_velocity( 0.3)
   #$StaticBody2D.set_constant_angular_velocity( -0.3)
   #$StaticBody2D.set_constant_angular_velocity( 1)
```

Friction

This is the 2Dbody's friction. The values range from 0 (no friction) to 1 (full friction).

Example:

```
 func _ready():
  #$StaticBody2D.set_constant_linear_velocity(Vector2( 0, -12))
  $StaticBody2D.set_friction( 1)
```

Bounce

This is the body's bounciness. The values range from 0 (no bounce) to 1 (full bounciness).

Example:

```
func _ready():
  #$StaticBody2D.set_constant_linear_velocity(Vector2( 0, -21))
  $StaticBody2D.set_bounce( 0.6)
```

Physics material override

The physics material override method allows you to change the physics material. New material can have definite properties such as friction, rough, bounce, and absorption.

Example:

```
func _ready():
  var NewMaterial = PhysicsMaterial.new()
  NewMaterial.set_bounce( 1)
  $StaticBody2D.set_physics_material_override( NewMaterial)
```

StaticBody2D is OK for a static game object, but you can do these in other ways like tiles with collision shapes. With **`StaticBody2D`**, you can create a dynamic game object. Thanks to properties (constant linear velocity, constant angular velocity, friction, bounce, physics material override). Developers can create game objects like sliders, bounce trampolines, or horizontal elevators (escalators) for 2D games. It's advisable to learn about **`StaticBody2D`** methods too. We can't include everything in this book. A game developer can find additional information about the **`StaticBody2D`**, **`KinematicBody2D`**, and **`RigidBody2D`** properties and methods in the book *Mastering Godot* by (Prof. Marijo Trkulja).

RigidBody2D

`The RigidBody2D` reacts to physical impulses in 2D game space. When you, for example, test the **`StaticBody2D`** linear or angular velocity, you need to set a **`RigidBody2D`**. Linear and angular velocity are also properties for **`RigidBody2D`**, but you can use the applied force and torque.

When using **`RigidBody2D`**, collision shapes are almost a must. You can use **`CollisionShape2D`** for collision detection; a **`CollisionPolygon2D`** is an option as well.

In **`RigidBody2D`** properties, you can find the mass, weight, and gravity scale. Remember that **`RigidBody2D`** does not emit collision signals by default. If you need them, set **`contact monitoring`** to **`true`**.

A game developer can find a detailed explanation of the **`RigidBody2D`** properties and methods in the book *Mastering Godot*. If this extensive material needs to be included in a book, it will go beyond its scope and purpose. However, a few examples are given.

Example:

```
# Example: turning monitoring on
$RigidBody2D.set_monitoring( true)

# Example: Code for RigidBody2D creation
extends Node2D
onready var texture = preload(“res://icon.png”)

func _ready():
 create_rb( Vector2( 210, 120))

func create_rb( pos_var):
 var rigid = RigidBody2D.new() # new rigidbody2D
 rigid.set_position( pos_var)
 var coll = CollisionShape2D.new() # new collisonshape2D
 var shape = RectangleShape2D.new() # shape type
 shape.set_extents( Vector2( 33, 33)) # shape extents
 coll.set_shape( shape)
 var sprite = Sprite.new() # new sprite
 sprite.set_texture( texture) # setting texture for sprite
 rigid.add_child( coll)
 rigid.add_child( sprite)
 .add_child( rigid)

# Example: Generator for a random placed rigidbody2D
# use with user created “create_rb” function
func _ready():
 randomize()
 for i in range( 110, 460, 30):
  var rnd_no = randi() % 11 + 1
  create_rb( Vector2( 10 + i + rnd_no, 10))
```

Hope this is an excellent introduction to **`RigidBody2D`**. Now, we will learn about some of its properties.

Mode

A **`RigidBody2D`** has four modes; the default is rigid(0), and the others are a static(1) and kinematic(2) body. The mode character (3) is similar to the rigid but without rotation.

Example:
```
func _ready():
  var rb = RigidBody2D.new()
  var cs = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents(Vector2(120,30))
  rb.set_position(Vector2(60,60))
  cs.set_shape(shape)
  rb.add_child(cs)
  rb.set_mode(1) # Static mode
  self.add_child(rb)
```

Mass, weight and inertia

The property **`mass`** is the **`RigidBody2D`** mass; the default value is **`1`**. Mass and weight are dependable values; weight is adjusted when you change a mass value. The weight value is based on the mass and the default gravity value. Inertia is like mass, but for rotation (for automatic calculation), inertia is 0. You can create a **`RigidBody2D`** with the default name for the code example.

Example:
```
func _ready():
  print($RigidBody2D.get_mass())
  $RigidBody2D.set_mass(3)
  print($RigidBody2D.get_weight())
```

Example:
```
func _ready():
  $RigidBody2D.set_inertia(0)
```

Friction

This is the body’s friction. The values range from 0 (friction-less) to 1 (maximum friction). The default value is 1. If, for example, you set the friction to 1, the **`RigidBody2D`** would slide on an inclined surface of the **`StaticBody2D`**.

Example:
```
func _ready():
  $RigidBody2D.set_friction(0.3)
```

Bounce

This is the body's bounciness. The default value is 0.

Example:
```
 func _ready():
  $RigidBody2D.set_bounce(0.3)
```

Physics material override

Using this feature, you can change the properties of the material. For example, you can change the friction and bounce.

In addition to this, it is possible to change two additional properties of RigidBody2D: rough and absorbent. Let's take a look at the following example.

Example:
```
 func _ready():
  var rb = RigidBody2D.new()
  var cs = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents(Vector2(120,30))
  rb.set_position(Vector2(60,60))
  cs.set_shape(shape)
  rb.add_child(cs)
  self.add_child(rb)
  var pm = PhysicsMaterial.new()
  pm.set_friction(0)
  pm.set_bounce(0.3)
  rb.set_physics_material_override(pm)
```

Gravity scale

The gravity scale multiplies the gravity applied to the body. The body's gravity is calculated from the **`Default Gravity`**; its default value is **`1`**.

Example:
```
 func _ready():
  $RigidBody2D.set_gravity_scale(0.3)
```

Custom integrator

If true, the internal force integration is disabled for this body. Apart from the collision response, the body will only move as determined by the **`_integrate_forces()`** function.

In the following example, the body movement is defined by the physics material override. So, for instance, you need to add a **`RigidBody2D`** node.

Example:

```
func _ready:
  $RigidBody2D.set_use_custom_integrator(false)
  var nm = PhysicsMaterial.new()
  nm.set_bounce(1)
  $RigidBody2D.set_physics_material_override(nm)
```

Continuous cd

The continuous collision detection tries to predict where a moving body will collide instead of moving it and correcting its movement safter the collision. As a result, the continuous collision detection is slower but more precise and misses fewer collisions with small, fast-moving objects. In addition, raycasting and shape casting methods are available.

Modes:

`CCD_MODE_DISABLED = 0`

The continuous collision detection is disabled. This mode is the fastest way to detect body collisions, but you can miss small, fast-moving objects:

`CCD_MODE_CAST_RAY = 1`

The continuous collision detection can be enabled using ray-casting. This mode is faster than shape casting but less precise:

`CCD_MODE_CAST_SHAPE = 2`

The continuous collision detection can be enabled using shape casting. This mode is the slowest CCD method and the most precise.

Example:

```
func _ready():
   $RigidBody2D2.set_continuous_collision_detection_mode(2)
```

Contact reported

The method contact reports are to set max contacts to report. The default value is 0. So, for example, you need to create a **`RigidBody2D`** and a **`StaticBody2D`** node with collision shapes.

Place **`RigidBody2D`** above **`StaticBody2D`**. Also, set the **`RigidBody2D`** signal body shape entered. This signal will create a function.

Example:

```
func _ready():
  $RigidBody2D.set_contact_monitor(true)
  $RigidBody2D.set_max_contacts_reported(4)

func _on_RigidBody2D_body_shape_entered(body_id, body,
body_shape, local_shape):
  if $RigidBody2D.is_contact_monitor_enabled():
    print(body_id)
    print(body)
    print(body_shape)
    print(local_shape)
```

Contact monitor

If true, the body will emit signals when it collides with another **`RigidBody2D`**—default value: false. Take a look at the example for contact reported.

Sleeping

If true, the body is sleeping and will not calculate forces until woken up by a collision or by using **`apply_impulse()`** or **`add_force()`**.

Example:

```
func _ready():
  $RigidBody2D.set_sleeping(true)
  if $RigidBody2D.is_sleeping():
    print("Use apply_impulse() or add_force()")
```

Can sleep

If true, the body will not calculate forces and act as a static body without movement. However, the body will wake up when other forces are applied via collisions or using **`apply_impulse()`** or **`add_force()`**—default value: true.

Example:

```
func _ready():
if $RigidBody2D.is_able_to_sleep():
  $RigidBody2D.set_can_sleep(false)
```

Linear velocity

This is the body's linear velocity.

Example:

```
func _ready():
  $RigidBody2D.set_linear_velocity(Vector2(-120,30))
```

Linear damp

The linear damp method damps the body's **linear_velocity**. You can add the timer node (0.15, autostart) and **RigidBody2D** in this example.

Example:
```
onready var l_damp = 0
func _ready():
  $RigidBody2D.set_linear_velocity(Vector2(-120,0))
func _on_Timer_timeout(): # func created by Timer signal
  if l_damp < 10:
    $RigidBody2D.set_linear_damp(l_damp)
    l_damp += 1
```

Recapitulation

Godot 2D Physics is very rich with 2Dbody's properties and methods. The most commonly used are **RigidBody2D**, **KinematicBody2D**, and **StaticBody2D**. For collision detection, **Area2D** is a good choice.

The StaticBody2D is used for a static object with 2D Physics in the game creation process, and **RigidBody2D** is used for moveable game objects.

You can create these objects by using IDE or by code. For example, let me remind you how to do it.

Adding RigidBody2D with Godot IDE

First, to create a **RigidBody2D** node, use the add child node or keyboard shortcut Ctrl + A. Type in the search field Rigid, select **RigidBody2D**, and click on the **Create** button. Repeat the same (when **RigidBody2D** is selected), type collision, and select **CollisionShape2D**. The **CollisionShape2D** needs to be sub-node. In the **Inspector** window, find Shape and select new **RectangularShape2D**. Use the select mode micro buttons to set the size for the collision shape.

Adding RigidBody2D with GDScript

Type the following GDScript code to create a **RigidBody2D** with **CollisonShape2D**.

Example:
```
extends Node2D
func _ready():
  var rb = RigidBody2D.new()
  var cs = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents(Vector2(120,30))
  rb.set_position(Vector2(60,60))
  cs.set_shape(shape)
  rb.add_child(cs)
  self.add_child(rb)
```

Task

You have enough know-how data to create a spawning point. Use the default texture for the rigid body and create an initial button. Your code needs to make 21 game objects with 2D gravity (rigid body) above the static body (use default or other texture). You can create **`StaticBody2D`** with Godot IDE, but you will have it in the following example if you decide to use coding. The following example is also practical when making a spawning procedure.

Example:

```
func create_stat_body():
  var sta = StaticBody2D.new()
  var col = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents( Vector2( 600, 5))
  col.set_shape( shape)
  sta.add_child( col)
  var sprite = Sprite.new()
  sprite.set_texture( txture)
  sprite.set_scale( Vector2( 12, 0.12))
  sprite.set_position( Vector2( 300, 0))
  sta.add_child( sprite)
  sta.set_position( Vector2( 30, 500))
  self.add_child( sta)
```

It will be good to start working on a task and then go back to see a possible solution. In the following coding example, you can see the solution for your previous task. The answer has two functions. First, you create a static body, and second, you make a rigid body:

```
onready var txture = preload(“res://icon.png”)
func _ready():
  create_stat_body()
  for f in range( 1, 22):
   create_rigid_spawn()

func create_stat_body():
  var sta = StaticBody2D.new()
  var col = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents( Vector2( 600, 5))
  col.set_shape( shape)
  sta.add_child( col)
  var sprite = Sprite.new()
  sprite.set_texture( txture)
  sprite.set_scale( Vector2( 12, 0.12))
  sprite.set_position( Vector2( 300, 0))
  sta.add_child( sprite)
  sta.set_position( Vector2( 30, 500))
  self.add_child( sta)

 func create_rigid_spawn():
  var sta = RigidBody2D.new()
  var col = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  shape.set_extents( Vector2( 12, 12))
  col.set_shape( shape)
  sta.add_child( col)
  var sprite = Sprite.new()
  sprite.set_texture( txture)
  sprite.set_scale( Vector2( 0.39, 0.39))
  sprite.set_position( Vector2( 0, 0))
  sta.add_child( sprite)
  sta.set_position( Vector2( 250, 10))
  self.add_child( sta)
```

Conclusion

The chapter is one of the essential book chapters. In the next chapter, we will create our first game project. Thus, knowledge about 2D bodies will be necessary. For example, we will learn about **`KinematicBody2D`**, and the student will extensively apply its properties in a *Chapter 4, Creating a 2D Game*.

For an exciting gameplay, the concept of a game object called game bodies is included in the Godot game engine. As discussed in this chapter, we use **`StaticBody2D`** in 2D space for static game objects. The moveable game object gives the necessary gameplay and awakens a game player's interest. For this, we use **`KinematicBody2D`** and **`RigidBody2D`**. So, it's wise to go through tasks and examples to understand the body's concept better.

Questions

1. What is the difference between RigidBody2D and StaticBody2D?
2. How will you code craft RigidBody2D?
3. What are the additional game elements for a functional RigidBody2D?
4. If you want to create a game character, when will you use a KinematicBody2D?

CHAPTER 4

Creating a 2D Game

As your game dev's know-how increases, it's a good time to start making a complete 2D game. For this, we use the Godot 2D engine. Of course, we can make any 2D video game, but let's use previously learned about 2D Physics and 2D Math to create one.

2D character is usually the core of any 2D game; for this one, we can use it as simply as possible. But still, the character needs to have movement ability in 2D space, and we can add some animations to it. Therefore, you will have the possibility to create a complex 2D character for the following 2D game project. When you create one, it's good to have a plan. A plan helps you to see better your creative vision. So, let's plan a 2D adventure.

The first part of the plan is an entirely functional game level. So, it would help if you had a 2D Character, 2D game environment, 2D game props, game mechanics, and game graphics. The second part is creating additional game elements like game menu, intro video, music, audio effects, save systems, and so on. After, you can add more game levels and start with game testing.

In this chapter, we will create a 2D game character; make a 2D game environment; create game props; troubleshoot a code; learn about coding ethics; improve game play-ability; and game menu.

2D character

You can start a new game project and a new 2D scene for a 2D character. For example, you can create a KinematicBody2D with **`CollisonShape2D`** and Sprite2D as child nodes. You can set a rectangle as the collision shape. Use **`ball_patform.png`** from the file repository as the sprite2D texture.

In the beginning, we used a simple graphic to create and test. Later on, you can change it with more quality graphics for an eventual game project for a game market. Rename the root node as **`2D_character`** (RMB + rename) or something you like, and add a GD Script to it (RMB + attach script).

Create the process function and add the following code for movement in 2D space:

```
onready var pos = Vector2( 0, 0)
onready var mf = 3

func _process(_delta):
  pos = $KinematicBody2D.get_position()
  if Input.is_action_pressed( "ui_right"):
   $KinematicBody2D.set_position(Vector2( pos.x + mf,pos.y))
  if Input.is_action_pressed( "ui_left"):
   $KinematicBody2D.set_position(Vector2( pos.x - mf,pos.y))
  if Input.is_action_pressed( "ui_up"):
   $KinematicBody2D.set_position(Vector2( pos.x, pos.y - mf))
  if Input.is_action_pressed( "ui_down"):
   $KinematicBody2D.set_position(Vector2( pos.x, pos.y + mf))
```

The code allows the movement in the four directions of 2D space. The variable **`mf`** is the movement factor (character speed) and can be changed before the game starts or in a game with some game bonuses.

In a game project, we use 2D position change for the movement. In the next game project, you will learn about other methods for 2D character movement.

The player moves a bat on a 2D screen to influence a ball's movement in this game. A good idea can be to add a ball game prop in the same scene with a 2D character (bat).

You can add **`RigidBody2D`** with two child nodes (**`CollisonShape2D`**, **`Sprite2D`**). The settings for **`RigidBody2D`** are custom integrator on (true), contact reported 3, and contact monitor on(true). You can find these properties in the **`RigidBody2D`** inspector window. You can set **`circleShape2D`** for collision, and **`small_ball_magenta_transparent.png`** for **`Sprite2D`** texture as shown in the *Figure 4.1*. You can test the character movement and interaction with a ball on this stage if you like. A suggestion is to test it after we add it as an instance in a 2D environment.

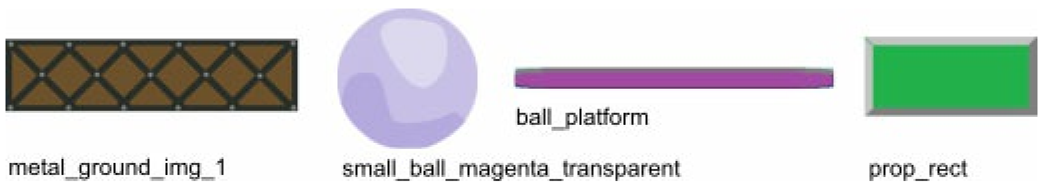


***Figure 4.1:** Graphical props for a game*

2D game environment

Let’s create a new 2D scene for a game environment. For this, let’s use the Sprite2D node and a few static bodies. Our game environment will have passive game elements with the collision and without it. When it is done, we can add some active game elements (game props).

You can add the **ColorRect** node, and set it. You can set the size first, and then put a color in the inspector window. A **ColorRect** is our initial background, but you can change it later with some Sprite2D texture.

Next, add few static bodies, and create a rectangle-shaped 2D space for gameplay. StaticBody’s (game objects) can have set values for linear and angular velocity.

You can create the game space with TileMap, but tiles need to have collision shapes for the game to work correctly. So, when you add the tilemap node, go to the inspector window and set the new tileset. Then, click on it and select **Edit**. New “tiles” is added with a new texture micro button (plus sign). Select a texture for a single tile.

Click on **New single tile**. Then, select the region with a mouse click and drag. If you need a collision shape for a tile, click on the **Collison** button. Next, pick the collision type, and click on the texture image. With this, a single tile is added in tileset.

We can set a bouncing property to **RigidBody2D** (ball) to make things smoother. For this, create a new physics material:

```
func _ready():
  var nm = PhysicsMaterial.new()
  nm.set_bounce(1)
  $RigidBody2D.set_physics_material_override(nm)
```

Creating game props

In the next part, we will learn something interesting. Usually, games have many game props. You can create it one by one, or you can code.

So, we code. The goal is to create a function for generating game props. Because game props are StaticBody’s, we make a function for StaticBody’s creation.

The Function needs **StaticBody2D**, **CollisonShape2D**, shape type, **sprite2D**, and some properties for each node. Let‘s look at a possible code for it:

```
onready var textura = preload(“res://prop_rect.png”)
func create_props( prop_name, posx, posy):
  var statB = StaticBody2D.new()
  var coll = CollisionShape2D.new()
  var shape = RectangleShape2D.new()
  var sprite = Sprite.new()
  shape.set_extents(Vector2( 15, 30))
  coll.set_shape( shape)
  sprite.set_texture( textura)
  sprite.set_scale(Vector2( 0.3, 0.3))
  statB.add_child( sprite)
  statB.add_child( coll)
  statB.set_name( prop_name)
  statB.set_position(Vector2( posx, posy))
  .add_child( statB)
```

It looks complex. Maybe by its first look, but let me explain. You first create elements with a new method, then you set properties. Later, you add node and child nodes.

With set extents, you are forming a collision shape in a 2D space. The set shape defines the collision shape type. When you need to add an image to **`Sprite2D`**, use the "set texture." The set scale makes scaling for a sprite texture.

With the set name property, your instance gets a name. The set position defines the position in a 2D space.

The next part of the code will call the function and create props. The three parameters in the function are prop name, x part of vector2 position, and y part of vector2 position:

```
 func _ready():
  var i = 1 # Used for different prop names
  for f in range( 120, 1040, 40):
   create_props (“prop” + str( i), f, 100)
   i += 1
  for f in range( 120, 1040, 40):
   create_props( “prop” + str( i), f, 120)
   i += 1
  for f in range( 120, 1040, 40):
   create_props( “prop” + str( i), f, 140)
   i += 1
```

The game will have a simple score system. For this, add one label node. When the ball interacts with a game, the prop player gets one point. Then, the ball disappears from the game screen. For this, we will create a signal from **`RigidBody2D`**. Create a signal for the body shape entered. Then, you can add a code in the generated function:

```
onready var score = 0
func _on_RigidBody2D_body_shape_entered(body_id, body,
body_shape, local_shape):
  for f in range( 1, 72):
   if body.name == “prop” + str( f):
   body.queue_free()
   score += 1
   $Label.text = “S C O R E: “ + str(score)
```

If the name of a game object is recognized, the object will disappear (**`queue_free`**).

You can try to play it.

Troubleshooting

It‘s good to know few things about project troubleshooting in Godot IDE. For this to work, set your game scene as the main scene. Use the main menu **`Project | Project Settings`**. Select the card general in the IDE, find the **`Run`** option, and select your scene as the main scene.

For example, you can start a game project (*F6*), and a debugger window pops up with some data.

Parse error: Identifier pos is not declared in the current scope.

To solve this, do not interrupt the application; add the **`pos`** variable (before the ready function):

```
onready var pos = Vector2( 0, 0)
```

Then, continue with the game project (*F5*).

The previous error shows a red dot in the debug window. There are some mistakes with yellow dots (warnings). The application will work if you don’t solve them.

For example, you can get an error like the following:

The argument ‘delta’ is never used in the **`function’_process`**.

This warning can quickly solve. Use delta in your coding or write underscore. Look at the following example:

```
func _process(_delta):
```

You can set debugging options in project settings/debugging. For example, to disable all warnings, find debug/gdscript and uncheck warnings.

Coding ethics

Some rules in coding are not crucial in script working but helps you to solve eventual errors better. For example, make at least two rows accessible between functions. Create space when writing variables and math. Put one free space in front of numbers or important data to see them better. Look at the following code example:

```
extends Node2D
onready var pos = Vector2( 0, 0)
onready var score = 0
onready var textura = preload(“res://prop_rect.png”)
onready var mf = 3

func _ready():
 var i = 1
 for f in range( 120, 1040, 40):
  create_props(“prop” + str(i),f,100)
  i += 1

func _on_Button_button_down():
 $RigidBody2D.set_use_custom_integrator(false)
```

Better game play-ability

The code and design improvements in a game are work after one game level are playable. If this is so, you can continue with the lecture. Otherwise, make one good game level.

Play-ability is superior with better graphics, smooth game object collisions, audio effects, and better game mechanics. But that is not all. When all game elements are improved, play-ability is automatically better.

For example, we can firstly improve the game graphics by changing the quality of sprites. Then, go to data storage and use graphical resources to improve the visual impression of the game screen. Finally, you can change the previously set graphics on all elements except game props.

You can change graphics for game props by coding. In the **`create_props`** function, you have methods for changing a sprite texture. Look at the following example:

```
sprite.set_texture(textura)
```

And the texture is preloaded in the GDScript header, with a line of code and reference:

```
onready var textura = preload(“res://stone background.png”)
```

Be aware of reference; it needs to have a link towards the game resource. As a reminder to do so, select *copy path* after RMB.

Every change in the sprite size can be set rightly with **`extents`** for the collision shape and scale for a sprite size. Look at the previously set code:

```
shape.set_extents(Vector2(15, 30))
sprite.set_scale(Vector2(0.3, 0.3))
```

When you put it right, you will have more quality graphics for the main game prop (*Figure 4.2*).

For a game frame, you can use more quality graphics now (**`frame.png`**, **`frame_decor.png`**). This time, we will use a tilemap with collision shape tiles. For few game objects, **`StaticBody2D`** is OK, but in a situation like this, tilemap is a better solution. Create tiles and set the game frame. Later, you can change the game background color with a 2DSprite texture (**`BG.png`**). Also, you can change the sprite texture for a **`KinematicBody2D`** with more quality one as shown in the figure **`Pad_02_1.png`**.

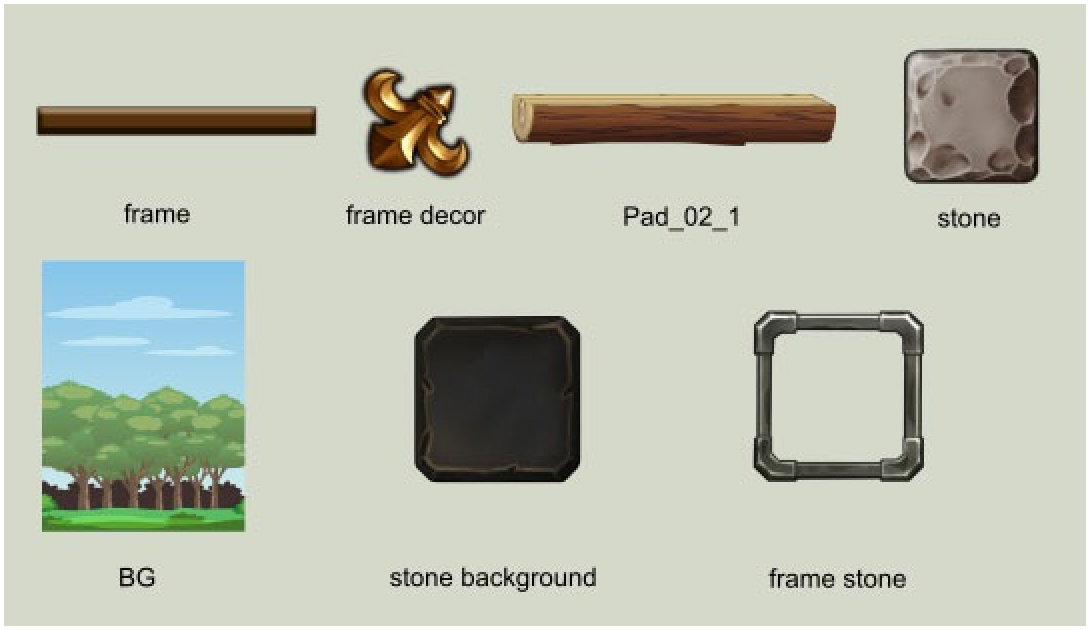


Figure 4.2: *Game props for better play-ability*

Simple game prop

Our game prop is interacting with a rigid body (the game object ball). In the interaction, the prop disappears from a screen, and the player gets the point. First, we can change the texture for this prop. Use stone **`background.png`** (*Figure 4.3*) as explained earlier, and see if you can set the scale and extents correctly. Now, we can make some changes in coding. Add the property v in your **`create_props`** function:

```
 func create_props( prop_name, posx, posy, v):
   var statB = StaticBody2D.new()
   var coll = CollisionShape2D.new()
   var shape = RectangleShape2D.new()
   var sprite = Sprite.new()
```

If the variable v is one, this will be a simple **`prop(prop_a`**). For complex props, you will use other values (2, 3, and so on). Look at the following example:

```
 for f in range(120, 1040, 40):
   create_props("prop_a" + str(i),f,190,1)
   i += 1
```

Complex game props

What if we want to create some complex game props? For example, a ball game object needs to interact more than once with a static game prop as shown in the *Figure 4.3*. After two or more interactions, a game prop will also disappear.

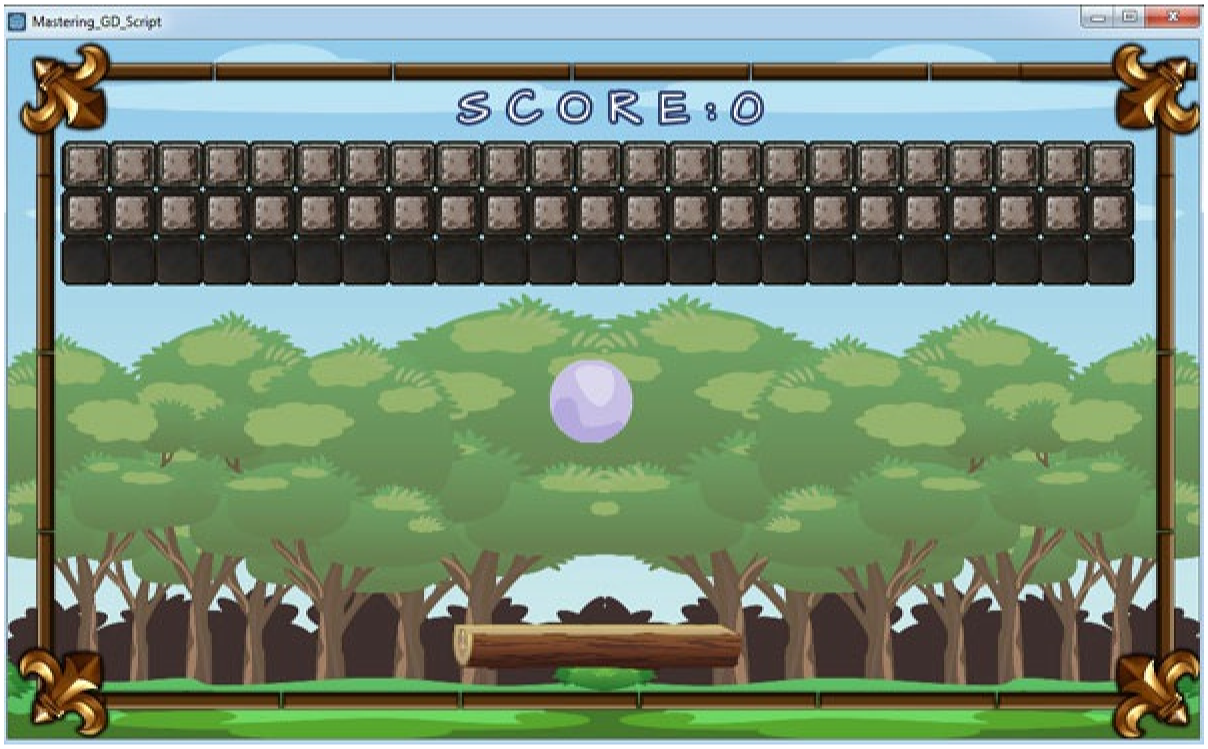


Figure 4.3: *Game-play with static props*

For this, we will remember interactions in a game. First, let's set an array for

interaction placing:
```
 onready var prob_b = []
```
Now, we can set the initial value to the array (put it in a ready function):
```
 func _ready():
   prob_b.append( “initial_value”)
```
As you know, every game object has a name. To set a node’s name, we use **`set`**, and to get it, we use **`get`**. Look at the following example:
```
 for f in range( 1, 72):
   if body.get_name() == “prop” + str( f):
     body.queue_free() # delete object if name is right
```
If we, for example, want to delete a game object after a second interaction, we can count interactions. In a situation where we have more than a few game objects, we will remember the interaction in an array. In the following code, the object will exist after the first and will be deleted after the second interaction:
```
 if body.get_name() == “prop” + str(f):
     for f in range( 0, len( prob_b)):
       if prob_b[ f] == body.get_name():
         # second interaction
         pro = false
         body.queue_free()
       if pro == true:
         # first interaction
         prob_b.append(body.get_name())
 print(prob_b) # list’s objects names in output window
```
For a quality visual effect, use two sprite texture. After the first interaction, the preceding sprite can be deleted. You can add the following or similar code for a sprites textures:
```
 if v == 2:
   var sprite2 = Sprite.new()
   sprite2.set_texture(textura2)
   sprite2.set_scale( Vector2( 0.083, 0.083))
   sprite2.set_name( “sprite2”)
   statB.add_child( sprite2)
```
Adding points can be done after each interaction or after the last one when an object disappears from a game screen:
```
 if pro == true:
   body.get_node(“sprite2”).queue_free()
   prob_b.append(body.get_name())
   score += 1
   $Label.text = “S C O R E: “ + str(score)
```

In the following code, you can see the complete function for two interaction game props:

```
func _on_RigidBody2D_body_shape_entered(body_id, body,
body_shape, local_shape):
   var pro = true
   for f in range( 1, 72):
     if body.get_name() == “prop_b” + str(f):
       for f in range( 0, len( prob_b)):
         if prob_b[ f] == body.get_name():
           pro = false
           var pos = body.get_position()
           body.queue_free()
       if pro == true:
         body.get_node(“sprite2”).queue_free()
         prob_b.append(body.get_name())
         score += 1
         $Label.text = “S C O R E: “ + str(score)
```

Custom game prop

In the previous example, we learned how to create code generated game prop, and now we will make a game prop using Godot IDE. First, add the **`StaticBody2D`** node with **`collisionShape2D`** and **`Sprite2D`** as sub-nodes. Next, add a script to **`StaticBody2D`**, and write the following code:

```
extends StaticBody2D

export var pos_x = 119.071
export var pos_y = 107.644
export var prop_img = preload(“res://4.png”)
export var prop_name = “shine_prop”

func _ready():
$Sprite.set_texture(prop_img)
.set_position(Vector2(pos_x, pos_y))
.set_name(prop_name)
```

Figure 4.4: *Graphical resources for a custom game prop*

Our custom game prop will have a few properties defined as export variables. For a prop position, we use the **pos_x** and **pos_y** variables; for texture, we will use the **prop_img** variable for images shown at *figure 4.4*, and for the game prop name, we use the **shine_prop** export variable.

If we, for example, instantiate the game prop, we will have options for different variable settings. Settings options will allow you to create a few different variants of the same game prop, which is the main characteristic of a custom game prop.

Let's add a few code lines to cover collisions with a prop and scoring. First, create a signal for **body_shape_entered**, and write the following lines of code in the created function:

```
func _on_ShineProp_body_shape_entered(body_id, body, body_shape,
area_shape):
  if body.get_name() == "RigidBody2D":
    score += 9
    $Label.text = "S C O R E: " + str(score)
    .get_node("shine_prop").queue_free()
```

Some score points will be added when the game ball (**RigidBody2D**) has a collision with a prop collision shape, and the game prop is clear from the screen (**queue_free**):

Generated custom prop

One of the great Godot features is scene instancing. Instancing a game scene can be done with IDE or by coding. To code a custom prop, we will use a previously created one. First, let's instance a game prop with a GDScript code:

```
onready var scene = preload("res://ShineProp.tscn")
func _ready():
  var instance = scene.instance()
  add_child(instance)
```

Our custom game prop has some export variables so that we can set them:

```
func _ready():
  var instance = scene.instance()
  instance.pos_x += rnd * 40
  instance.prop_name = "custom_prop_" + str(n)
  add_child(instance)
```

To complete the previous code, we will add random numbers and integer values for instantiate count. The code will look like the following example:

```
func _ready():
  # setting few custom props at random position
  var rnd
  var n = 1
  for f in range(1,4):
   randomize()
   rnd = randi() % 22 + 1
   var instance = scene.instance()
   instance.pos_x += rnd * 40
   instance.prop_name = “custom_prop_” + str(n)
   n += 1
   add_child(instance)
```

The coding used in the “**`for`**” loop iterate three times and instantiate the same number of custom game props. All your custom props need to have different names for collision detection:

```
 instance.prop_name = “custom_prop_” + str(n)
```

We have the option to use other export variables so that we can change the prop image:

```
 onready var img_1 = preload(“res://4.png”)

 func _ready():
  # setting custom prop at random position
  for f in range(1,4):
   randomize()
   rnd = randi() % 22 + 1
   var instance = scene.instance()
   instance.pos_x += rnd * 40
   instance.pos_y -= 40
   instance.prop_img = img_1
   instance.prop_name = “custom_prop_” + str(n)
   n += 1
   add_child(instance)
```

If you want to have similar coding in an example game, look at the following **`GDScript`** code part:

```
 onready var scene = preload(“res://ShineProp.tscn”)
 onready var img_1 = preload(“res://4.png”)
 onready var img_2 = preload(“res://1.png”)

 func _ready():
  # setting custom prop at random position
  var rnd
  var n = 1
  for f in range(1,4):
   randomize()
   rnd = randi() % 22 + 1
   var instance = scene.instance()
```

```
    instance.pos_x += rnd * 40
    instance.prop_name = “custom_prop_” + str(n)
    n += 1
    add_child(instance)
   # setting custom prop at random position
   for f in range(1,4):
    randomize()
    rnd = randi() % 22 + 1
    var instance = scene.instance()
    instance.pos_x += rnd * 40
    instance.pos_y -= 40
    instance.prop_img = img_1
    instance.prop_name = “custom_prop_” + str(n)
    n += 1
    add_child(instance)
    # setting custom prop at random position
   for f in range(1,4):
    randomize()
    rnd = randi() % 22 + 1
    var instance = scene.instance()
    instance.pos_x += rnd * 40
    instance.pos_y -= 80
    instance.prop_img = img_2
    instance.prop_name = “custom_prop_” + str(n)
    n += 1
    add_child(instance)
```

Game menu

Now when we have a functional and playable video game, we can add a few more things. These things will be game buttons for a game menu. So, we are creating a straightforward video game menu.

First, add a node2D as an object pallet and rename it to **`MainMenu`**. The settings for node2D are layer 1 and scale 0.3 (for both x and y). The transform property is as follows:

```
 x 0.3 y 0
 x 0 y 0.3
 x 0 y 0
```

Next, add **`Sprite2D`** and four **`TextureButton`** nodes for menu buttons. The texture for a **`Sprite2D`** node is **`Table.png`**, (look at *Figure 4.5*) and other suggested properties are as follows:

```
 position (765, 1149)
 rotation 0
 scale (1.33, 175)
```

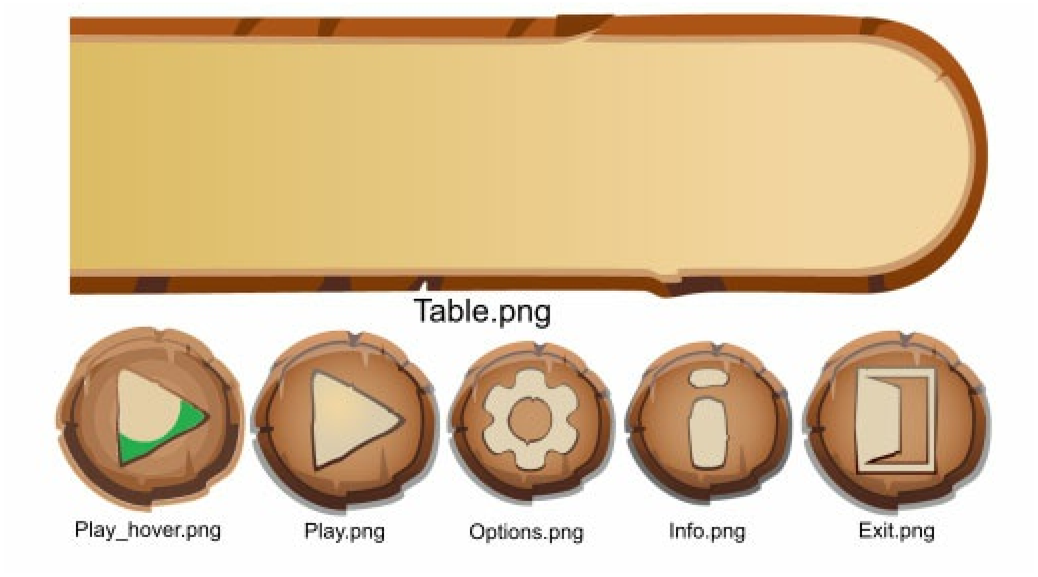


Figure 4.5: *Textures for a sprite and texture button nodes*

Set all game buttons on the Sprite2D, and add textures. For example, the play button has two textures. One is for hovering over a button, and the other is for a normal game button state (see *Figure 4.6*).

And finally, you can code some functionality for these buttons. In the following example, you will see how to code game-play and quit game options.

Example:

```
func _on_Play_btn_pressed():
  $RigidBody2D.set_use_custom_integrator( false)

func _on_Exit_btn_pressed():
  get_tree().quit()
```

Example comment:

When the player clicks on a play button, a custom integrator (for a **`RigidBody2D`**) sets to false. Thus, visually, the ball becomes susceptible to 2D gravity and slowly falls towards the player bat. When a player clicks on the **`Exit`** button, the game quits. Now, you have your first playable video game as shown in the figure 4.6, and this is a good step towards some complex game project.

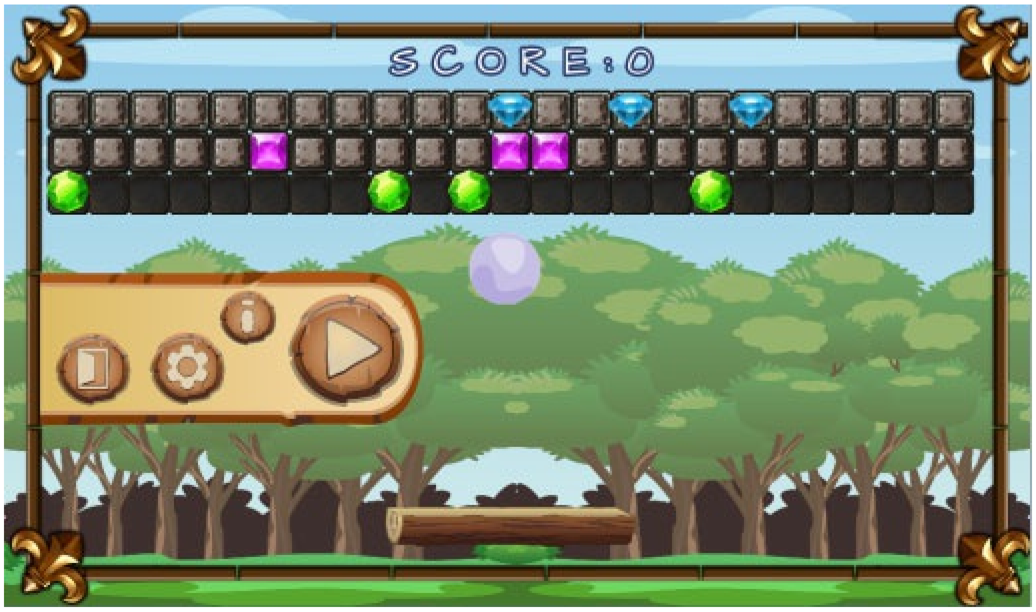


Figure 4.6: *Game menu and game elements*

Conclusion

By creating our first playable 2D video game, you learned a lot. First, you learned additional info about Godot IDE, but you must become more experienced with GD Script codding. Then, for a variety of gameplay solutions, coding is vital. Finally, you had the opportunity to apply knowledge about 2D bodies and combine it with quality coding.

Also, you learned how to implement random numbers and how to apply the for-loop and if branch combination.

In the next chapter, we will conclude our 2D game programming. Students will learn how to make a turn-based 2D video game.

Questions

1. Did you understand game props?
2. What kind of game props are explained in this chapter?
3. Why are game props an essential part of a playable video game?

CHAPTER 5
2D Adventure

It will be suitable for the student to learn gradually until this chapter. In this chapter, you will have a game developer's level of understanding. So, things previously explained wouldn't be repeated here. But still, we will like to help you in the learning process. Therefore, I will put some references to previously taught know-how. Adventure is one of the most desirable game-play types. Therefore, we will create one 2D adventure. Of course, we will use some of Godot IDE-defined game elements in the creative process, but still many things will be solved by coding in a GDScript. Our adventure is a turn-based game adventure, and players will use mouse and keyboard keys for playing. So, one of the first things to learn can be a system for turn-based game-play. So, let's start.

In this chapter, students will learn about turn-based game-play, game character movement, 2D Platformer as a quest system, TileMap as background, and coding for a platformer character.

Turn-based game-play

Initially, we can use buttons for this. With the Button node, we can quickly solve mouse interaction and in-game field graphics. For mouse interaction, we will use signalling, and for field graphics, we can use TileMap.

First, we will add the Button node to the new scene (2D_Adventure). We will set size parameters in the **`rect`** property (inspector window) as 64×64. Create a few more of them and put them like in a lane. We created six of them in a vertical manner (*Figure 5.1*):

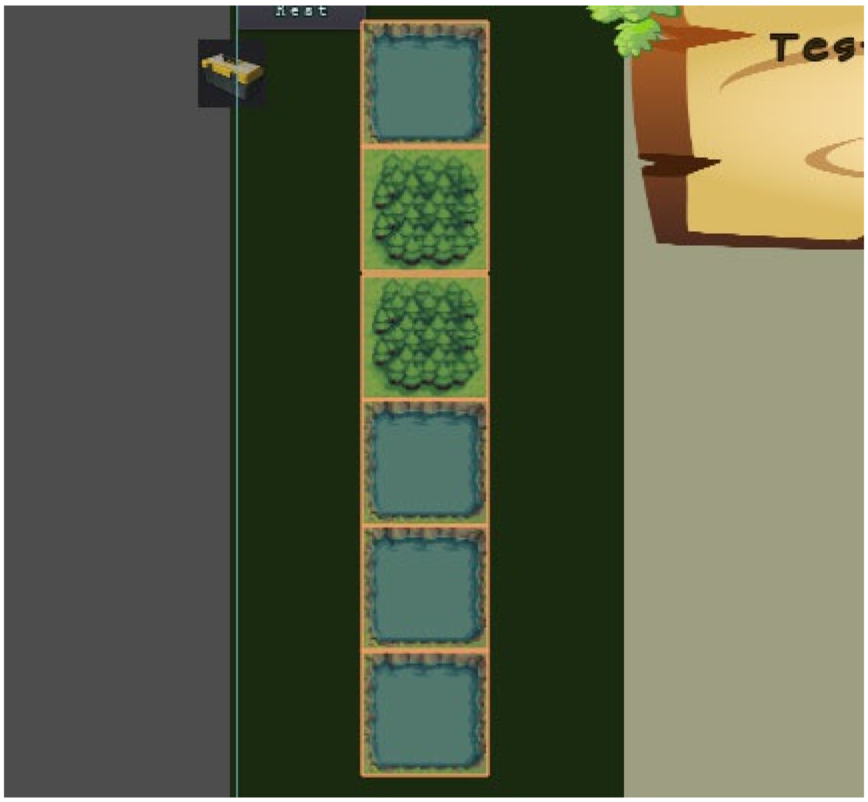


Figure 5.1: *Buttons for turn-based game*

It‘s good to know that we can combine tile-map images as background graphics for the standard button node in this game solution. Naturally, therefore, students need to put nodes in good 2D positions for quality visual performance.

Second, we will add a tile-map node as a game element. A tile-map node doesn‘t have a TileSet, so you will need to create one. We will create a tile-set in an inspector window as a drop-down option New TileSet. Next, we will add some images to TileMap. Possible default graphics for TileMap are visible in *Figure 5.2*:

Figure 5.2: *Possible images for a TileMap*

The intention in a turn-based game is to simplify the game-character movement. Therefore, we will create it as a Sprite node with the ability to change the position graphically. For example, we will have a forward-backward movement or up-down in a game as different players perceive this scene. So, add a sprite node. For texture, use the default character image (Figure 5.3).

Parameters for a texture are vframes 4, hframes 4, and frame 10 in an animation property.

Game character movement

Now, we will start a coding movement for a game character. But, first, let‘s define the character position with an array:

```
 extends Node2D

 var space = [0,0,0,0,0,0,1,0]
```

The array has an initial field and seven more areas. Because we have six buttons, this will be six positions. Therefore, our game character is in the sixth position, represented with the number 1 in the “space”array.

Figure 5.3: *Game character*

Note: I hope you remember how to add the GD script to the initial node. If not, see the introduction.

Your coding will change the character position after each move. So, it will be fine if you create signals for all buttons. Also, you need to define one user function and call it in all six button-created functions.

Example:

```
func _on_Button3_pressed():
  uni_func( 1 )

func _on_Button4_pressed():
  uni_func( 2 )

func _on_Button5_pressed():
  uni_func( 3 )

func _on_Button6_pressed():
  uni_func( 4 )

func _on_Button7_pressed():
  uni_func( 5 )

func _on_Button8_pressed():
  uni_func( 6 )

# universal function for movement resolve
func uni_func(no):
  pass
```

As you can see, every function will send a number to the **`uni`** function. The number will represent an in-game position field. So, we can add some code to define game field names (put in the user function and call it from a ready function):

```
$Button8.set_name( “Field6”)
$Button7.set_name( “Field5”)
$Button6.set_name( “Field4”)
$Button5.set_name( “Field3”)
$Button4.set_name( “Field2”)
$Button3.set_name( “Field1”)
```

Now, we can quickly get the field position for the game character movement:

```
var pos = .get_node(“Field” + str( no)).get_global_position()
```

And, we can set and tune position for a character:

```
pos += Vector2( 32, 32) # tune position
$Sprite.set_position( pos) # set character pos
```

So, the code in the **`uni`** function will be as follows:

```
func uni_func(no):
  var pos = .get_node(“Field” + str( no)).get_global_position()
  pos += Vector2( 32, 32)
  $Sprite.set_position( pos)
```

The explanation of the user-defined functions is given in chapter three, but some helpful code for button names can be as follows:

```
func _ready():
  micro_menu()
func micro_menu():
  $Button8.set_name( “Field6”)
  $Button7.set_name( “Field5”)
  $Button6.set_name( “Field4”)
  $Button5.set_name( “Field3”)
  $Button4.set_name( “Field2”)
  $Button3.set_name( “Field1”)
```

Now, start a game (*F5*). But, first, check the code, and remember that the **`set_position()`** method defines the position of the Sprite node.

So, we can continue with something about gameplay. In this adventure, the game character can move only to the first nearest field. Therefore, we will add some coding:

```
func uni_func(no):
  if space[ no + 1] == 1 or space[ no - 1] == 1:
    var pos = .get_node(“Field” + str( no)).get_global_position()
    pos += Vector2( 32, 32)
    $Sprite.set_position( pos)
    if space[no + 1] == 1:
      space[no] = 1
      space[no + 1] = 0
    if space[no - 1] == 1:
      space[no] = 1
      space[no - 1] = 0
```

The code will verify if the game character is near to the click field (button). Then, only if the game field is above (no+1) or below (no-1), the movement will be possible. After that, the code will set a new character position and define an empty game field.

Example:

```
if space[no + 1] == 1: # field above
    space[no] = 1 # setting g.character pos.
    space[no + 1] = 0 # define empty field
if space[no - 1] == 1: # field below
    space[no] = 1
    space[no - 1] = 0
```

Gameplay

Our 2D adventure is a turn-based video game, and this means the game has turn-based gameplay elements.

First, we will add action points. We can use a global Godot variable called singleton for this. Simply put, a singleton is a script file with settings in the Godot IDE. So, create one script file, click on the **`Script`** tab and select **`New script`** in the **`File`** menu. Write the following code lines, and save the file:

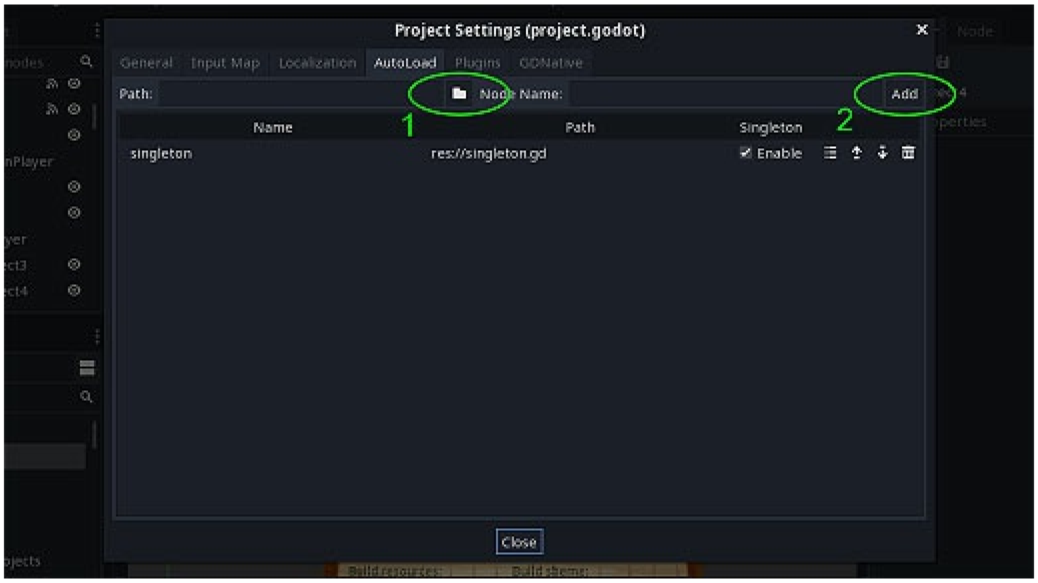
```
extends Node
onready var action_points
func _ready():
  action_points = 2
```

Open the **`Project settings`** option in the project (main menu). Select the **`AutoLoad`** tab (Figure 5.4). Select the path to the **`singleton.gd`** script file (opt 1 in Figure 5.4), and set the name for singleton (opt 2 in Figure 5.4). We use singleton as the name for the singleton file. The Enable option needs to be true, so you need to check it.

Now, we have action points as global variables. To use them, write the code line as shown in the example.

Example:

```
$”/root/singleton”.action_points = 2
```



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

Figure 5.4: *AutoLoad tab of the project settings*

Second, let's add the **`period`** variable to singletons:

```
onready var period # write in the header
period = 0 # in the ready function
```

We can create a textured button for the next turn. When the player clicks on it, the period will increase by one, and action points will get the initial value. The texture for the **`next turn`** button is **`Button_1.png`** (you can use **`Button_2.png`** for the hover button state). Create a signal, and write the following code in a generated procedure:

```
func _on_TextureButton_pressed():
  $"/root/singleton".action_points = 2
  $"/root/singleton".period += 1
```

For the background, add two colored rectangles in a **`CanvasLayer`** node (*Figure 5.5*).

Note: To remind you. First, add CanvasLayer as a node, and then add two child nodes to it. Child nodes are ColoredRectangle nodes.

If you put it as suggested, you will visually get something as shown in *Figure 5.5*.

It's good when action points are visible. Therefore, add one **`Label`** node, and rename it as the **`AcionP`**. Set the **`Label`** position in the middle of the blue background as shown in the Figure 5.5—Code user function as follows:

```
func writte_action_points():
  $ActionP.set_text("Action points: " +
  str($"/root/singleton".action_points))
```

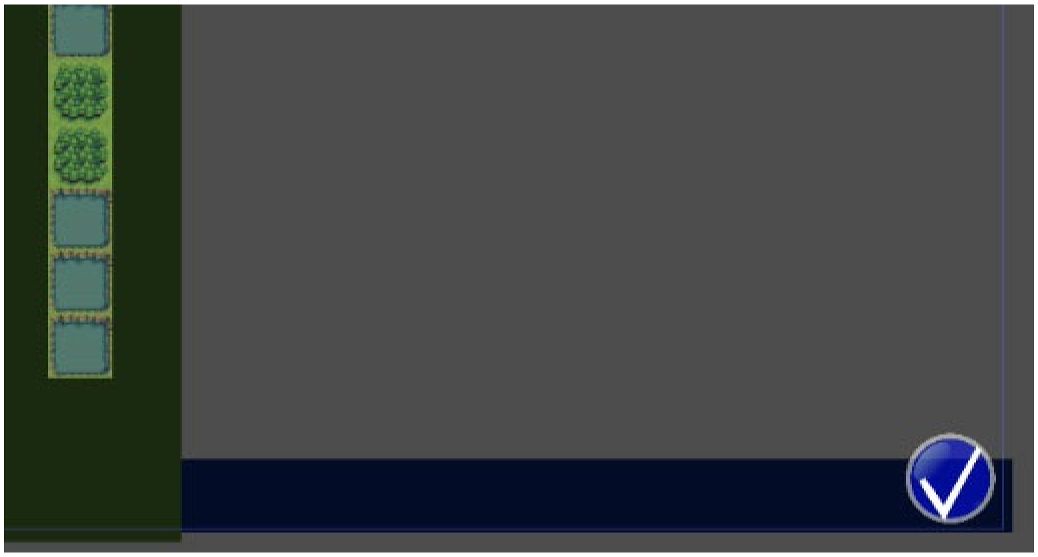


Figure 5.5: *Colored backgrounds and next turn button*

And call it from the **`next turn`** function:

```
 func _on_TextureButton_pressed():
   $"/root/singleton".action_points = 2
   $"/root/singleton".period += 1
   writte_action_points()
```

It will be good to create a label node for a **`period`** variable. So, add a label node, and rename it to **`PeriodLabel`**. Set a label position on the left-hand side of a blue background. Add a code line in the **`writte_action_points`** function:

```
 $PeriodLabel.set_text("Period: " +
 str($"/root/singleton".period))
```

Good, now gameplay has the action point and period system set. So, a player can make two actions, and then go to the next period. But we need to put some coding first if we want this to work without negative action points. Let's add one conditional branch to the **`uni`** function before the move calculation:

```
 func uni_func(no):
   if $"/root/singleton".action_points > 0: # conditional branch
     if space[ no + 1] == 1 or space[ no - 1] == 1: # move calc.
```

Now, the game character can move when it has action points. Otherwise, the player needs to start a new period.

Textual description of the area

Do you know what the main characteristic of 2D adventures was? Yes, you are right; it’s was textual description of a game area. First, there was text alone in the early adventure era, then came the combination with a pixel image, and finally animations. When the player goes to some new game area, the area description shows a textual narration. For example, if the player goes to the next large room in some game adventure, then area descriptions pop up.

Area description example:

```
"You are entering a large room. There are items on a table and a
wall. You see: silver sword, silver cross, holy water, and
hawthorn stake."
```

One of the ways to create a game area description is an array. So, we will create one:

```
var field_text = [“”, # put description code lines in a script
header
“You are at some small plain scape, and the water surface is
around.”,
“You are in the forest. Beautiful nature fills you with optimism
and energy.”,
“Green forest surrounds you with the sound of nature and
beautiful view.”,
“Water surface is almost all you see from this small plain
scape.”,
“You are at some small plain scape, and the water surface is
around.”,
“You start the adventure from some plain scape. Large water mass
is visible around.”]
```

Next, we need a text label for the area description. A suggestion is to use the **`RichTextLabel`** node for it. With this node, we can set the font and size for a text. Also, **`RichTextLabel`** is very good with a large portion of text. So, add one in the upper part of a game screen. Put the default text for it. For example, the default text can be: *You are in some unknown forest land full of intact nature with colorful natural sounds*.

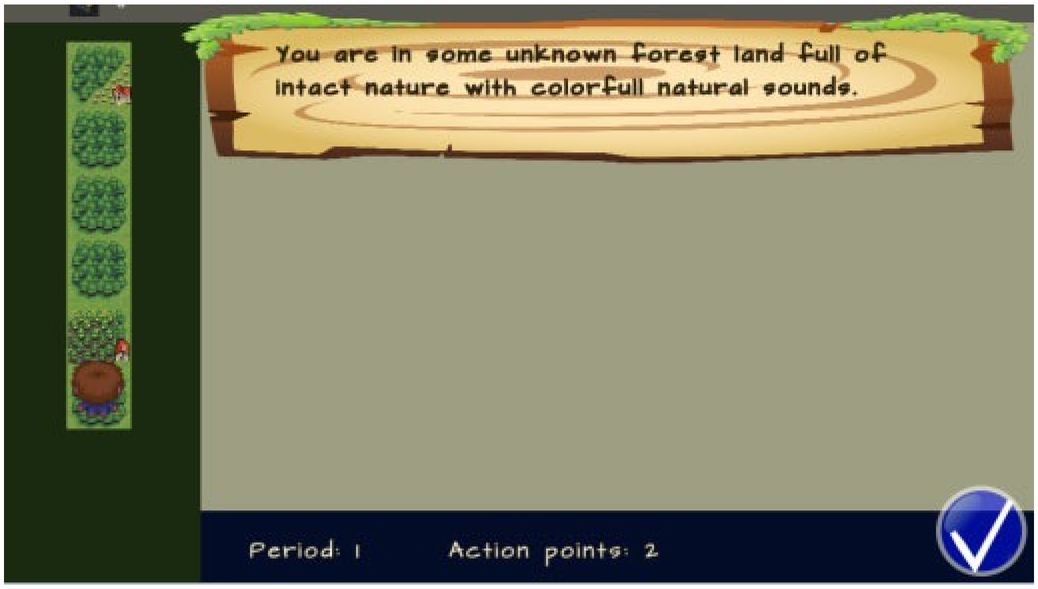
It will be nice to have a background for it. Therefore, add a sprite node with a texture (**`bt_text.png`**) as a background, shown in the *Figure 5.6*. Also, adding additional **`ColoredRectangle`** to a **`CanvasLayer`** will be good.

Note: When you arrange colored rectangles, note that the upper one in the IDE is below in a game screen.

Add one line in the **`uni`** function for a functional coding as shown in the example:

Example:

```
 $RichTextLabel.set_text( field_text[ no])
```



You are in some unknown forest land full of intact nature with colorfull natural sounds.

Figure 5.6: *Game area description with RichTextLabel node*

The different text will be in a **`RichTextLabel`** for each game field.

2D platformer as part of a quest system

We have a good gameplay so far, and now let's continue. Game quests are part of many different game types. A player can mostly find it in adventures and **RPG** (**Role Played Games**). So, we can create a quest system. A game quest will pop up as optional when game characters go to the unexplored field.

The quest system will have many micro platformers for playing. So, we will create a new 2D scene, 2D character, background, and game props. All this will be part of every micro quest as a 2D platformer game.

We will use **`KinematicBody2D`** as the main 2D character, TileMap for background, and Area2D for game props. After creation, we will instantiate a scene in the main game scene. Instantiation will be done with coding when the player goes to the unexplored game field.

TileMap and background for a platformer:

First, we will create a new 2D scene (name it **`2D_landscape.tscn`**). Then, we will add a TileMap node. Start adding tiles in a new tile-set. Our goal is to create a game background (*Figure 5.7*).

Note: Tiles can have a collision shape. Every tile for the game character path needs to have it. TileSet size 32×32, and quadrant size 16. When you add enough tiles (Figure 5.8a) to a tile-set, start placing them in a game scene. Try to create a similar game scene as shown in *Figure 5.7*:

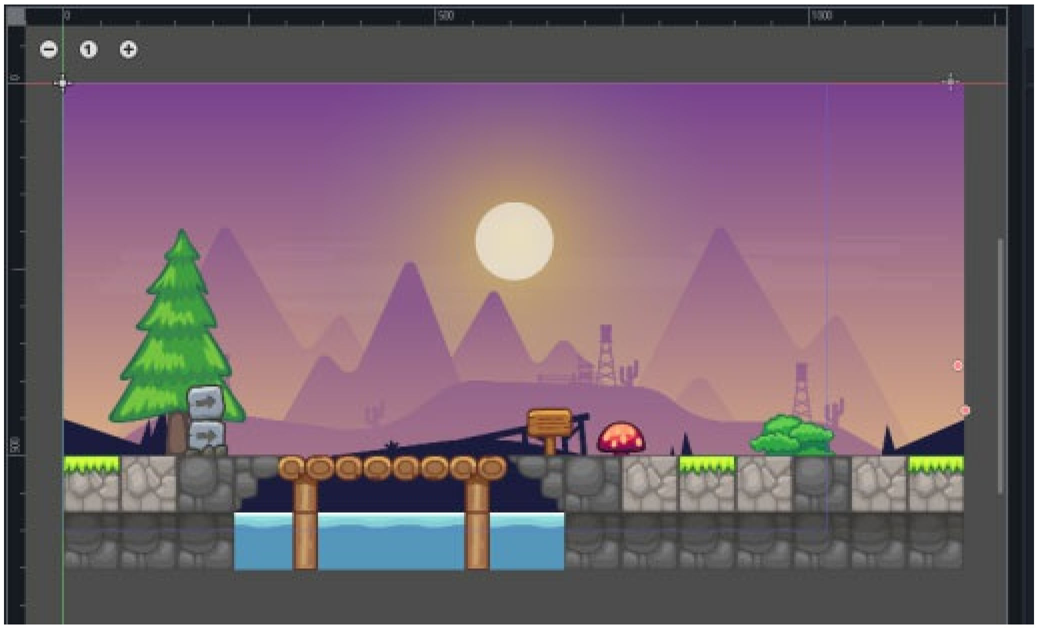


Figure 5.7: *2D platformer game environment*

Our game character needs a path for movement creation. For this, some tile-set tiles need to have a collision shape. For example, you will set collision shapes for 01, 02, 03b, 04a, bridge-2, and similar, but not for **`obj_direction`**, signboard, spring-1, and like (see image description in *Figures 5.8a and 5.8b*):

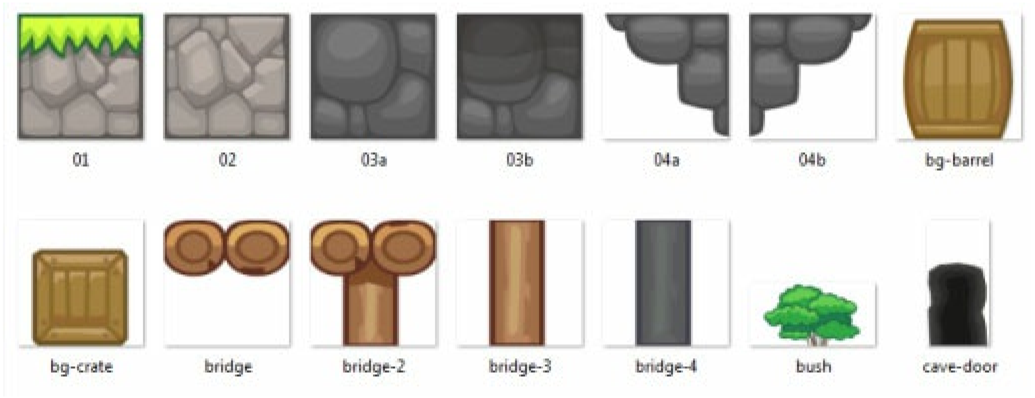


Figure 5.8a

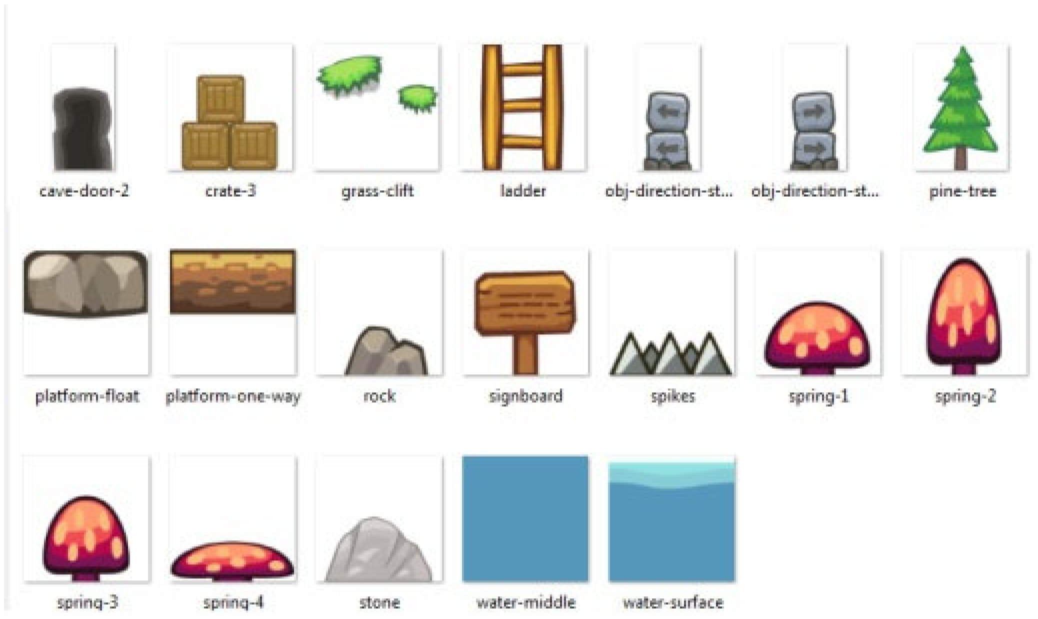


Figure 5.8b: *Images for a tiles*

When adding tiles to a game scene, try to have a rectangular shape of all scene objects. For this, you will may be need another TileMap node. So, right click on a root node, and add TileMap. Then, create a new tile-set with two tiles (texture **`Repeated.png`** and **`Water-surface.png`**).

Note: Tile-set cell size 64×64, quadrant size 16.

Now, we will define a game space for this scene. First, you need to add a **`StaticBody2D`** node with **`CollisionShape2D`** as a sub-node. Then, create additional and similar **`StaticBody2D`**, and put them as a game scene border on both the sides. Collision shapes of the **`StaticBody2D`** nodes will define the movement space for our game character.

We are almost ready to create a game character. Therefore, for a game background, we need to create two sprite nodes and a **`RichTehtLabel`**. Then, we need to put them in an upper-middle part of a game scene—set sizes accordingly (Figure 5.9). Finally, we will use them as info panels when a player finds resources in a game. Info panel game objects are not visible when the game starts. So, you can set its visibility in an inspector window.

Note: Find visibility in an inspector window and uncheck the visible property for info panel game objects.

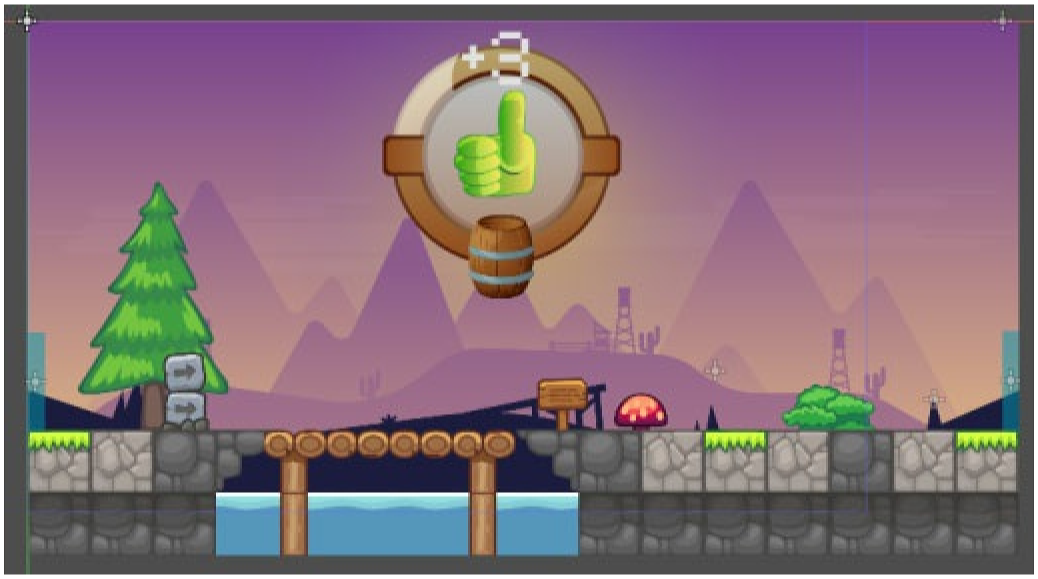


Figure 5.9: *Info panel in a game scene*

Player character

Now, let’s create a player character. For this, we will use **`KinematicBody2D`** with the collision shape and animated sprite. Animated sprite will give a necessary illusion of movement, and we have already created a good background. The player will use keyboard keys for it, right arrow key for right movement, and left arrow key for left movement.

So, let’s create. First, we will create a new scene named **`PlayerCharacter`**. Add a **`KinematicBody2D`**. Next, add a **`CollisionShape2D`** (**`CapsuleShape2D`** as child node) and animated sprite (child node).

Note: We use 2D game elements, so select blue AnimatedSprite.

Now, we will set a walking movement animation. For this, select **`AnimatedSprite`** and create new Sprite Frames (1 – Figure 5.10) in the inspector window (frames property). Then, click on **`SpriteFrames`** and create **`NewAnimation`** (2 – Figure 5.10, rename it to **`Walking`**). Load resource (3 – *Figure 5.10*, walking animation).

Note: Press shift, select first, and then the last images to select all walking resources (pictures). If you do it all right, frame images will be visible with a frame number.

Set scale for Animated Sprite to 0.12×0.12, and speed scale to 6.

Coding for a player character:

We need to code in a **`PlayerCharacter.tscn`** and **`2D_landscape.tscn`** scene. So, let's start with a current one (**`PlayerCharacter.tscn`**). Save it.

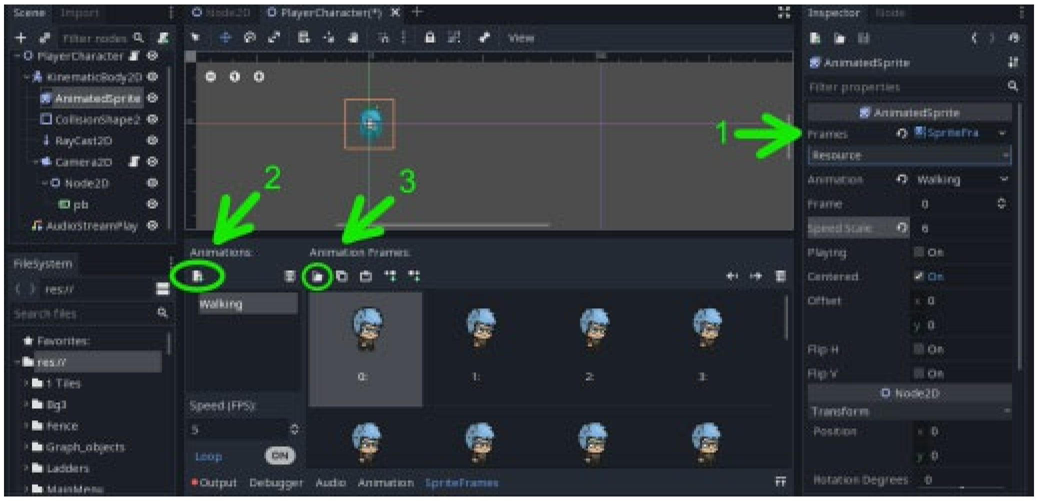


Figure 5.10: *AnimationSprite dialog window*

There will be two game character movement, move left and move right at this game stage. So, I suggest using the **`move_and_collide`** method in a **`process`** function.

Example:

```
func _process(_delta):
  if Input.is_action_pressed("ui_right"):
    $hero.move_and_collide( Vector2( speed, 0))
```

We will also need to define a character speed and code for a stopping pattern. A stoping code pattern can be realized with a combination of two methods, **`is_action_pressed`** and **`is_action_just_released`**.

Example:

```
extends Node2D
# Setting initial variables
var speed
func _ready():
  $KinematicBody2D.set_name("hero")
  speed = $"/root/singleton".speed
# Character movement
func _process(_delta):
  if Input.is_action_pressed("ui_right"):
    $hero.move_and_collide( Vector2( speed, 0))
  if Input.is_action_just_released("ui_right"):
   $hero.move_and_collide( Vector2( 0, 0))
```

Note: The default value for a speed is 1. It will be good to set it in a singleton script.

A similar code will be for a left character movement.

Example:

```
   if Input.is_action_pressed(“ui_left”):
    $hero.move_and_collide( Vector2( -speed, 0))
    $hero/AnimatedSprite.set_flip_h( true)
   if Input.is_action_just_released(“ui_left”):
   $hero.move_and_collide( Vector2( 0, 0))
```

Note: We added a line with a set_flip_h method for a character image flip when moving to another side. You can add a similar code line for a movement right with a false value.

Next, let’s use our **`Walking`** animation. Look at the following example.

Example:

```
func _process(_delta):
  if Input.is_action_pressed(“ui_right”):
   $hero.move_and_collide( Vector2( speed, 0))
   $hero/AnimatedSprite.set_flip_h( false)
   $hero/AnimatedSprite._set_playing( true)
  if Input.is_action_just_released(“ui_right”):
   $hero.move_and_collide( Vector2( 0, 0))
    $hero/AnimatedSprite._set_playing( false)
```

As you can understand, we use the **`_set_playing`** method with two different values for animation play.

Usually, in a 2D platformer, jumping is an option. Therefore, we will code craft a 2D gravity.

Example:

```
   # put it in a process function
   $hero.move_and_collide( Vector2( 0, 1)) # 2D gravity
```

Our game character scene may be functional. So, we need to instantiate it in a game scene. For this, open a **`2D_landscape.tscn`** scene, and put a few lines of code:

```
var character = preload(“res://PlayerCharacter.tscn”)

func _ready():
  var s = character.instance()
  s.set_position(Vector2(130, 450))
  .add_child( s)
```

Note: Set the position value accordingly to your 2D surroundings.

In the previous code example, the scene is firstly preloaded and then instanced as the variable (var s). After that, the position is set and then added as a child node.

So, what are you waiting for?

Let's check whether your mumbo-jumbo work has effects!

Play a **`2D_landscape`** game scene.

Of course, it will be marvelous if all works fine, but that is very rare in a world of game codding. So, check your coding first. In this stage of work, you can use finished and workable files from a book file repository. And learn from them.

Simple quest system:

We have a turn-based game part (**`2D_Adventure.tscn`**) and a platformer part (**`2D_landscape.tscn`**). It will be good to instantiate the platformer part somewhere (layer node, for example) in a turn-based game part.

Note: Add a layer node and rename it to "Platformer".

Instantiation can be done when the player presses a button.

So, add a button with the text **`Start a quest`**. Set the signal procedure, and add the following code.

Example:

```
onready var platf_scene_1 = preload("res://2D_landscape.tscn")
func _on_quest_btn_button_down():
  var platform = platf_scene_1.instance()
  $Platformer.add_child( platform)
```

Therefore, when the player presses a quest button, the platformer scene is visible in a layer part of a game screen. Layer nodes have good settings for a position, so use them wisely.

Conclusion

This chapter strengthens your knowledge about making 2D games. We learned about turn-based games and platformer 2D games. We also learned about new nodes like **`AnimationSprite`**, **`RichTextLabel`**, **`CollisionShape2D`** types, and so on. I hope your GD Script coding is better due to this chapter's coding challenges. With this chapter, we are concluding our 2D game programming, and in the next one, we will start learning about 3D video games programming. Students will have the opportunity to learn about the Godot 3D environment and start making their first game prototypes.

Questions

1. What are singletons, and why do we use them?
2. Which node do we use to simulate the turn-based game, and why?
3. Explain the advantages of the RichTextLabel node.

CHAPTER 6

3D Math and 3D Physics

The 2D game making is behind us, and we see a vast untouched space of the 3D game making. Everything you learn counts because crafting 3D games is not so complicated when you know to make 2D video games.

Creating a 3D game is a joyous process when you know. But, first, what is wise to learn about 3D video game making? Of course, the main characteristics are 3D graphics, so you can guess! 3D Physics is one of the main things to learn. And when you have 3D thrills in-game, you will need something to compute all this, which is 3D Math.

So, prepare yourself because a new chapter of your learning adventure starts soon.

Making a 3D game for fun is not so expensive, but still you need a modern computer with average capabilities. And when you create it by yourself, you will need to obtain 3D assets, which can be expensive than 2D. In this book, you will have some 3D assets for games, but this teaches you how to do something in a 3D game environment.

If you decide to make a commercial 3D game, you will need a quality computer with an excellent GPU and speed memory. Also, making a 3D game by yourself is a gigantuous project, so the developer's team is one of the best suggestions I can give you.

So, let's go!

In this chapter, students will learn about introduction to 3D games, spatial, vector3, creating 3D game objects with a GD Script, 3DMath, StaticBody, RigidBody, RigidBody methods, prototyping a game scene and game props, and important steps in a prototype testing process.

Introducing the 3D game IDE

We will start with practical things we already know. But, first, we will create a new project in a project manager window. Then, we will give it a name, for example, **`My first 3D world creation`**. Next, we will select a higher visual quality renderer (OpenGL ES 3.0 in ver.3).

When the project starts, create a 3D scene. With a 3D scene, the root node will be Spatial.

Your developer's environment is now in 3D. Use the MMB wheel for zooming, hold RMB for panning a scene, and you can slowly move through 3D space with W-S-A-D keyboard keys when holding RMB. Hold MMB and move the mouse for a viewpoint change. We will use the default 1-viewport perspective scene view. You can change this to an orthogonal view with more viewports (top, left, right, front, rear, bottom).

Now, you are using red 3D nodes. So, let's add one node. First, add a **`StaticBody3D`** to the scene. Next, add the Mesh instance as a subnode, and additional subnode - **`CollisionShape3D`**. And now, settings for nodes. If you, for example, open transform for StaticBody, you will see 9 parameters. Three parameters are for translation, an additional three for rotation, and 3 parameters for a scale. Why? A 3D space has a **`vector3`** variable with one additional parameter compared to a 2D space. So, now you are working with **`x`**, **`y`**, and **`z`**. But if you still need a 2D explanation, let me say **`x`** is length (red), **`z`** is the width (blue), and **`y`** is the height (green).

Transform settings for a StaticBody are: translation 0x0.549x0; rotation 0x0x0; scale 9x0.3x9; and matrix transform 9x0x0; 0x0.3x0; 0x0x9; 0x0.549x0. When writing settings, first input transform, and then just check the matrix transform (local object settings).

The static body in 3D space is similar to a **`StaticBody2D`** and doesn't move. Methods are almost the same: **`friction`** (float), **`bounce`** (float), **`physics_material_override`** (PhysicsMaterial), **`constant_linear_velocity`** (Vector3), and **`constant_angular_velocity`** (Vector3).

Note: A mesh is a collection of vertices, edges, and faces that defines the shape of a 3D geometry object.

Add a new CubeMesh to a MeshInstance node, and set translation 0x-1.82x0.

And now, we need to define the collision space for this game ground (MeshInstance). So, we need to add a new BoxShape to a CollisionShape, and set it accordingly. Move the object with the mouse (press W), rotate (E), or scale (R).

Add a directional light node. Next, add a camera node. Camera settings: current on, translation 0x11.39x23.17, and rotation -21.3x0x0.

In 3D games, you need to have at least one camera (current). So, one of them needs to be set as current when you have more. You can save and test this scene. It will be good to set the camera so the game object is visible as a central one.

Creating a 3D game object with a script

You can code a 3D game object with a GD Script. The procedure is similar to making a 2D game object. So, the 3D game Math is almost the same as 2D game math, but there are differences.

We will start with a RigidBody node. Look at the following example.

Example:
```
var rigid = RigidBody.new()
```

A rigid body will need a collision shape, and a shape type needs to be defined. The size setting for the collision shape is with a **`set_extents`**.

Example:
```
var coll = CollisionShape.new()
var shape = BoxShape.new()
shape.set_extents( Vector3( 1, 1, 1))
coll.set_shape( shape)
```

Now, let's set RigidBody's (translation parameter) size and combine coding in a function.

Example:
```
func create_cube_mesh( n, m, j, x):
  var rigid = RigidBody.new()
  rigid.set_translation( Vector3( n, m, j))
  var coll = CollisionShape.new()
  var shape = BoxShape.new()
  shape.set_extents( Vector3( 1, 1, 1))
  coll.set_shape( shape)
  # mesh code will go there
  rigid.add_child( coll)
  .add_child( rigid)
```

Note: In the 3D space, we use 3D game objects. Meshes can be basic 3D game objects.

Now, we will add coding for a cube mesh.

Example:

```
var mesh = MeshInstance.new()
var mesh_shape = CubeMesh.new()
var type = SpatialMaterial.new() # mesh material
mesh.set_mesh( mesh_shape)
mesh.set_material_override( type)
rigid.add_child( mesh)
```

In 3D game making, we can define different materials for game objects. In addition, 3D Physics usually renders other light interactions depending on the surface material.

So, let’s add a script to the root node (if you didn’t) and add the following code:

```
extends Spatial
var cube_colore = [ 0, Color.red, Color.aliceblue, Color.azure,
Color.blue, Color.brown]

func _ready():
  var x = 1
  for i in range( 1, 4, 1):
    for n in range( -4, 6, 3):
      create_cube_mesh( n, 12, i, x)
      x += 1
      if x == 6:
        x = 1
func create_cube_mesh( n, m, j, x):
  var rigid = RigidBody.new()
  rigid.set_translation( Vector3( n, m, j))
  var coll = CollisionShape.new()
  var shape = BoxShape.new()
  shape.set_extents( Vector3( 1, 1, 1))
  coll.set_shape( shape)
  var mesh = MeshInstance.new()
  var mesh_shape = CubeMesh.new()
  var type = SpatialMaterial.new()
  type.set_albedo( cube_colore[ x])
  mesh.set_mesh( mesh_shape)
  mesh.set_material_override( type)
  rigid.add_child( mesh)
  rigid.add_child( coll)
  .add_child( rigid)
```

You have a more less known code for generating game objects with the nested **`for`** loop. For example, the **`set_albedo`** method will set a color for a spatial material from an array of different colors. You can try to see how this works (*Figure 6.1*):

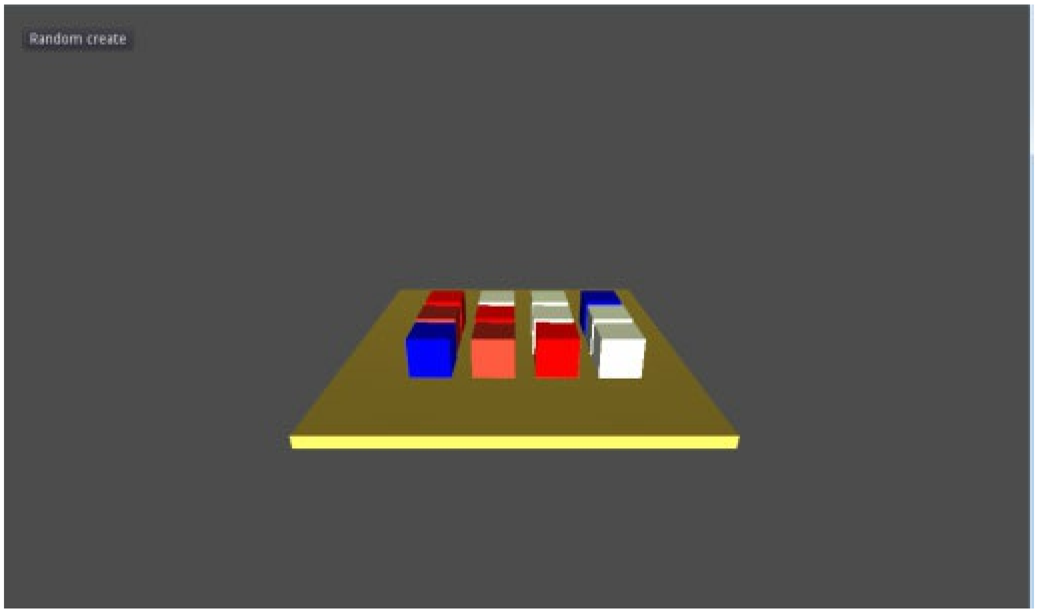


Figure 6.1: *First 3D game scene*

We will add one function with computing and random numbers to see additional 3D math examples. First, we will add a button, and set a button signal. Then, we will add the following GD Script code:

```
func random_create():
  var x = 1
  for i in range( 1, 4, 1):
    for n in range( -4, 6, 3):
      randomize()
      var o = rand_range( 1, 21)
      create_cube_mesh( n, o, i, x)
      x += 1
      if x == 6:
        x = 1
func _on_Button_pressed():
  random_create()
```

Again, you can test to see the generated game object after a button press. You can change the range in the first `for` loop from 1,1 to 1,6 to see a difference. The suggestion is to work a little with your first 3D scene to become more comfortable with a 3D IDE and GD Script coding in a 3D game space.

Task: Increase the size of StaticBody (ground game object) by scaling. Then, add a RigidBody with the collision shape and cylinder mesh. Set it similar to the *Figure 6.2*.

Note: Try to use F for focusing the object and hold Ctrl for the grid object movement.

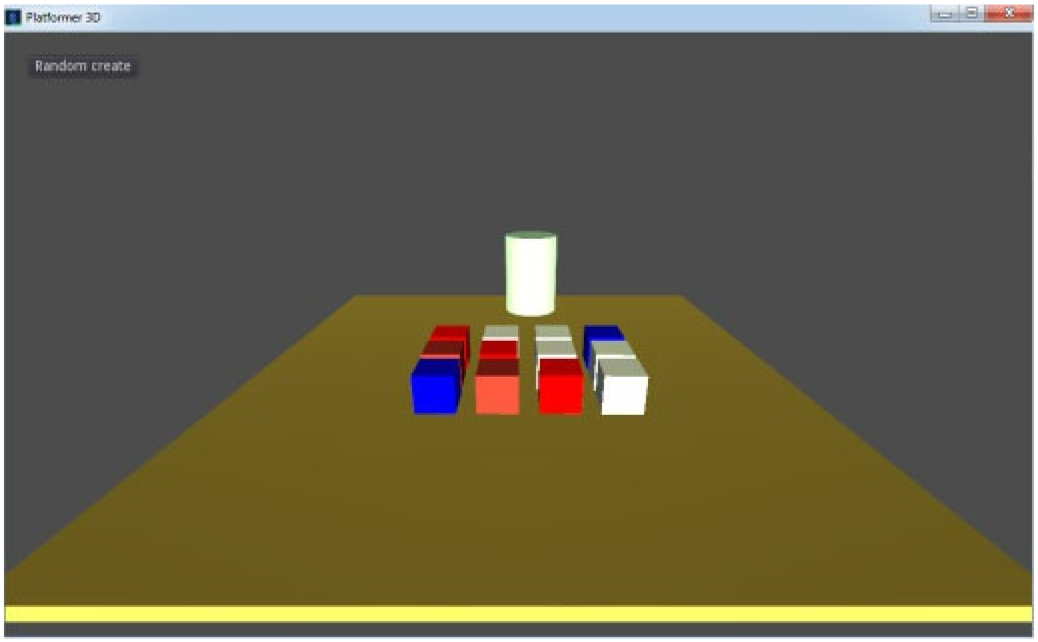


Figure 6.2: *3D task*

RigidBody has a lot of similar methods for the 3D game object movement. The developer can **`add_central_force`**, **`applay_centar_impulse`**, or **`set_axis_velocity`** for a similar 3D activity. With the **`set_axis_velocity`** method, we can create temporary velocity for an object in defined ways (see the effect in *Figure 6.3*):

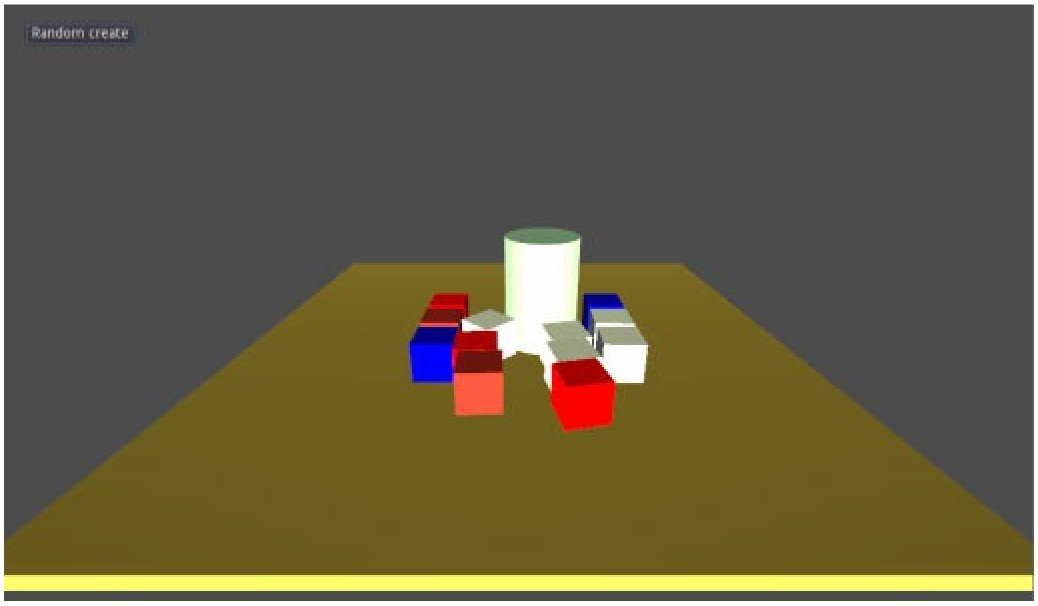


Figure 6.3: *Effect of set_velocity method*

3D models in a game scene

The main characteristic of an excellent 3D game is a game model. Game models are usually created in specialized software and prepared for game engines. Nevertheless, there are some things worth knowing about game models. When you, for example, obtain a 3D game model, you will need to know information about polygons, vertices, textures, materials, rigged status, game ready, UV mapped status, and unwrapped UVs status. These terms can be new for you, so let me explain them.

Polygon is a plane figure with many lines connected and closed together (polygonal chain or circuit). The segments of a polygonal circuit are called its edges or sides. The points where two edges meet are the polygon's vertices or corners.

A texture map is an image mapped to a shape or polygon (bitmap or procedural).

A material controls how a 3D object appears on the screen, which means that most materials take a texture as a parameter.

In our example of the 3D model, we will use a compact one (polygon 0, vertices 0) with texture and material.

Rigged is necessary when you have a skeletal and animated game object for object internal movement of rigged (connected) parts. Making a 3D game ready object is a process that produces low-poly or high poly game graphics. Low-poly uses texture filtering, and high poly uses polygons to determine the surface detail of the game object.

UV mapping is the 3D modeling process of projecting a 2D image to a 3D model's surface for texture mapping. Always use UV mapped game objects in your game projects. Our first 3D model is UV mapped. The process of creating a UV map is UV unwrapping, and the result can be overlapped or not overlapped UV map.

You can now prepare a game scene for a game character.

Task: Find the `.obj` file (The Limited 4.obj) in the file repository and add it to the resource folder. You can move the file by dragging and dropping it from the file folder to the resource folder. Add a `KinematicBody` node (name it **`Character`**) with a collision shape.

Drag the **`.obj`** file to the scene. Put it as a child node to the **`KinematicBody`**.

Scale for a mesh (**`.obj`** object) is 0.15x0.3x0.15. It will be good to set colors for a mesh. Use a spatial material and set the albino color for game character parts as shown in *Figure 6.4*:

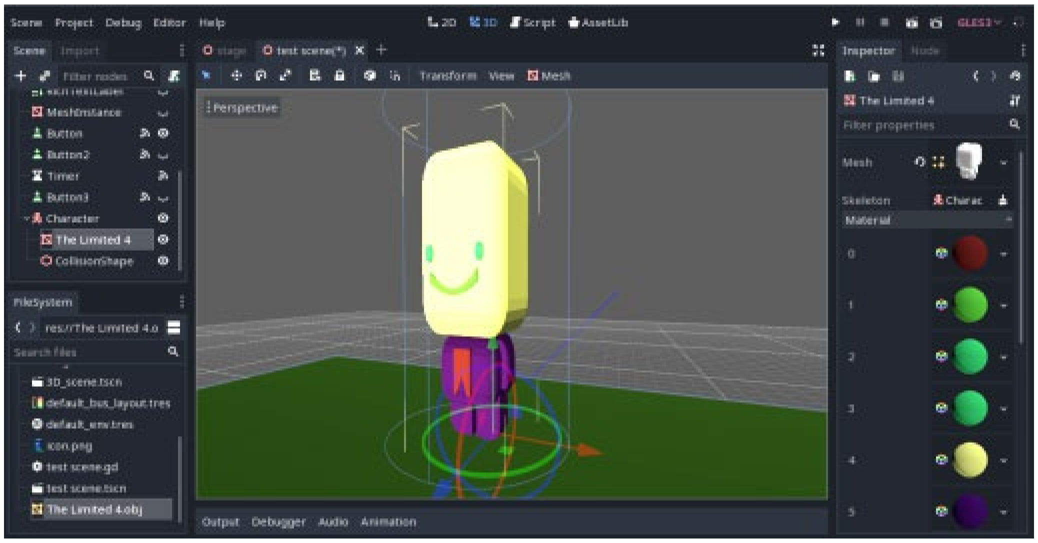


Figure 6.4: *Game character material(colors)*

We can add some coding to the button press function:

```
func _on_Button_pressed():
  $Character.move_and_collide( Vector3(0,0,1))
  # random_create()
```

The move and collide method moves the object till collision occurs with another game object. In our situation, we will have a short-distance movement. Then, the game object will repeat the action as shown in *Figure 6.5*, with each button click:

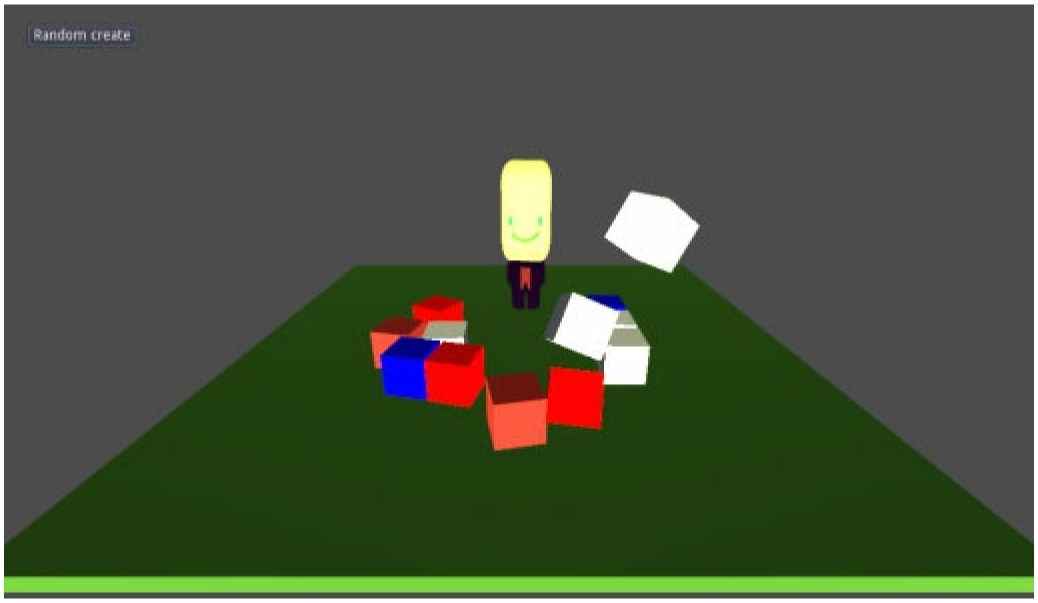


Figure 6.5: *Game character movement*

Material texture

As learned earlier, one of the game material definitions is texture. So, how can we add one? First, search for the albedo color type and add a texture for the spatial material. Next, find a **`www.cadnav.com_Plastic_0033.jpg`** file, and add it to a resource folder. Then, add a file as the material texture for one of the game character parts. For example, the fifth material is a character body material, and you can add a texture to it.

If you start a game scene, you wouldn't see a texture on a material. Therefore, you can add a texture to a ground mesh or use coding. A suggestion is to code some simple zoom as shown in Figure 6.6 for a camera to see the material's texture:

```
func _on_Button_pressed():
  $Character.move_and_collide( Vector3(0,0,1))
  $Camera.set_fov(21)
  # random_create()
```

Note: The fov is the camera field of view in degrees when using the perspective projection as shown in the *Figure 6.6*.

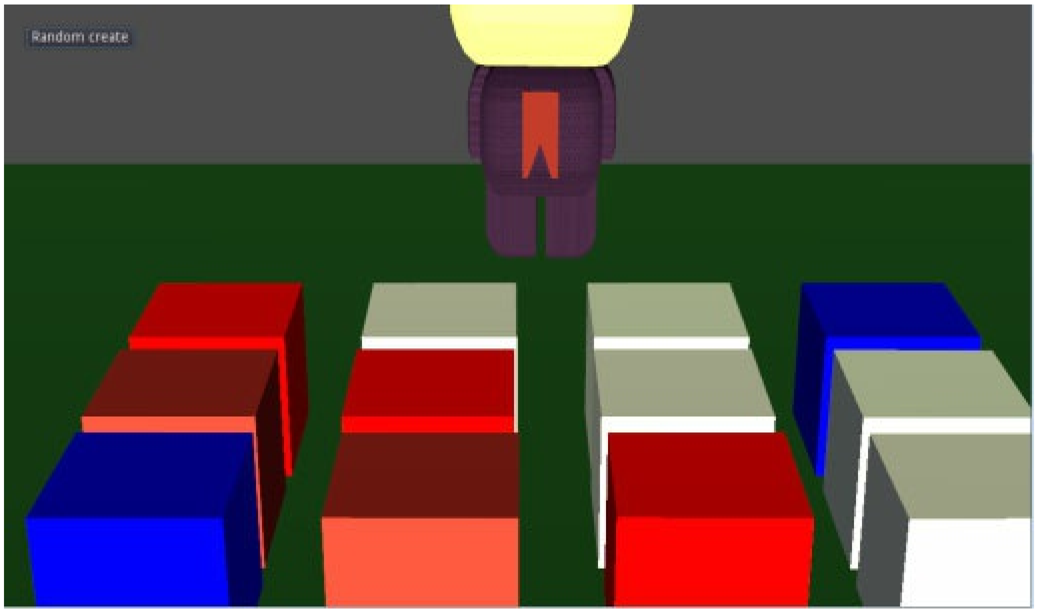


Figure 6.6: *Spatial material with texture in a game character*

Prototyping a 3D video game

You are reading an essential topic for saving a lot of your time, money, and creative energy. However, the fact is that making a 3D game is expensive, time-consuming, and energy-consuming. So, you will do this in a complex way, or you will understand and implement game prototyping.

Prototyping is making a functional video game with as low-quality graphics as possible. So, don't think about materials, textures, lighting, shader, and so on because they are not crucial for a functional video game.

Your prototype video game can be something to discard after testing. With testing, you need to comprehend all concepts of functionality and game-play. After that, you can rebuild the game structure in a limited time, such as 9 hours of standard working time. If you need more than 3-9 hours for rebuilding, your prototyping is gone too far.

Continuing with the topic, I will use the intro scene term to explain prototyping. Therefore, we will design a room with walls, windows, table, chairs, bed, and locker for our intro scene. We will also need a game character and script for character movement.

We wouldn't add any game assets, unique materials, or textures in the intro scene. However, we can use spatial materials with albino color properties. Coloring will give us some look but wouldn't drain time and energy.

Prototyping with CSG

The **CSG** stands for **Constructive Solid Geometry**. You can create various 3D prototyping objects with different CSG nodes. For example, shapes can be a Box, Cylinder (and cone), Sphere, Torus, Polygon, and Mesh. In addition, a developer can combine primitives under a **`CSGCombiner`** node. Every primary CSG node has three shape operations: union, intersection, and subtraction.

Note: Shape operations work when nodes are under CSGCombiner.

So, let's continue with an example. We will create a wall with two window spaces for two sides of the game scene and one blank wall on the third side. First, we will make a blank wall with a CSGBox node. Therefore, we will add a CSGBox node.

It will be good to change the scale for our ground object (StaticBody) to 21x1x21. Rename the created CSGBox to **`A_wall`** and set: width 0.9, height 15, depth 39, operation union, and collision to on. Then, move it to one side of the scene (behind a game character), or use translation as -20x7x0.

Next, we will create a wall with windows opening. In this, we will use a combiner CSG node, so add a **`CSGCombiner`** node (rename it to **`wall_with_windows`**). A combination will be with three CSGBox nodes. So, duplicate (Ctrl + D) an **`A_wall`** node, and put it under the **`wall_with_windows`** node.

Now, we can add the CSGBox node as a window. Node settings are width 2, height 6, depth 6, operation subtraction, and collision to off. Move a node (hold Ctrl), and put it somewhere in the wall. Subtraction will be visible in the IDE. Duplicate the same node, and put it as another window. And now, you can duplicate a **`wall_with_windows`** and put it on another side of a game scene. Set the use collision to on for both **`wall_with_windows`** combined objects.

A few things more, and we can test a scene.

But, first, let the set camera follow our game character's movement. So, move the camera node and put it under the game character node (as subnode). Then, change a camera position behind and above a game character. Finally, use the camera preview to see what you are doing as shown in Figure 6.7.

Now, we need movement coding (movement with arrow keyboard keys):

```
func _process(_delta):
   if Input.is_action_pressed(“ui_down”):
    $Character.move_and_collide(Vector3(0,0,-0.1))
   if Input.is_action_just_released(“ui_down”):
    $Character.move_and_collide( Vector3(0,0,0))
   if Input.is_action_pressed(“ui_up”):
    $Character.move_and_collide( Vector3(0,0,0.1))
   if Input.is_action_just_released(“ui_up”):
    $Character.move_and_collide( Vector3(0,0,0))
   if Input.is_action_pressed(“ui_right”):
    $Character.move_and_collide(Vector3(-0.1,0,0))
   if Input.is_action_just_released(“ui_right”):
    $Character.move_and_collide( Vector3(0,0,0))
   if Input.is_action_pressed(“ui_left”):
    $Character.move_and_collide( Vector3(0.1,0,0))
   if Input.is_action_just_released(“ui_left”):
    $Character.move_and_collide( Vector3(0,0,0))
```

Also, put a **`ready`** function content under a comment:

```
func _ready():
   pass
   # var x = 1
   # for i in range( 1, 4, 1):
   # for n in range( -4, 6, 3):
   # create_cube_mesh( n, 12, i, x)
   # x += 1
   # if x == 6:
   # x = 1
```

And now, you can test a game scene as shown in the *Figure 6.7*.

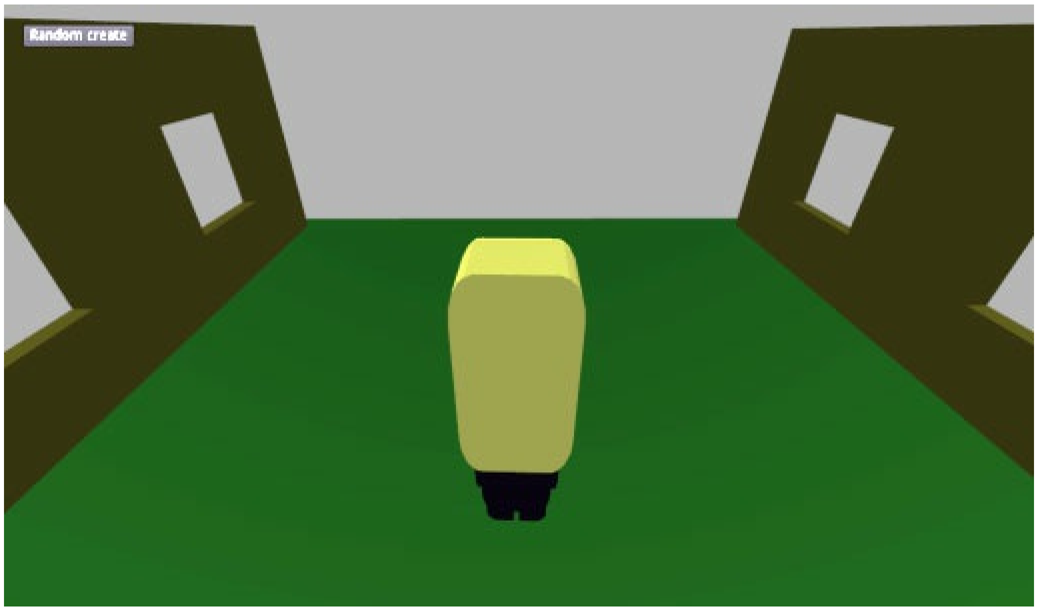


Figure 6.7: *Game scene prototyping*

Game props for the intro scene

What are game props? You know it; a game object with a purpose. Simple ones have only a collision set, and others have more functions. We will first have game props in our intro scene with a collision. I would like you to create it! In addition, make it with CSG nodes.

A table can be your first 3D game prop. You can create it with CSGBox nodes (union operand) and CSG combiner. I would like to give an example, create a table leg, duplicate and set position for a pair, and duplicate again. After you set table legs, add a tabletop.

When you are confident with this, create a chair game prop.

Note: You will now understand how game developers talk about screens like give me more and bigger ones. Nevertheless, you will learn to use views and keyboard shortcuts for better 3D editing.

Bed as a game prop is your next challenge, and after that, a locker. Again, the suggestion is to use different operators because it's not wise to solve all with the union CSG operator. So, maybe you will need to use subtraction with spheres to create a pillow. Moreover, when creating a locker, you can make something with subtraction and an intersection.

When you finish any game props, enable collision in a root one (usually a combiner node). Additionally, one reminder! Don't spend too much time creating game props. No need for perfect ones; they are for testing. Refer to *Figure 6.8*:

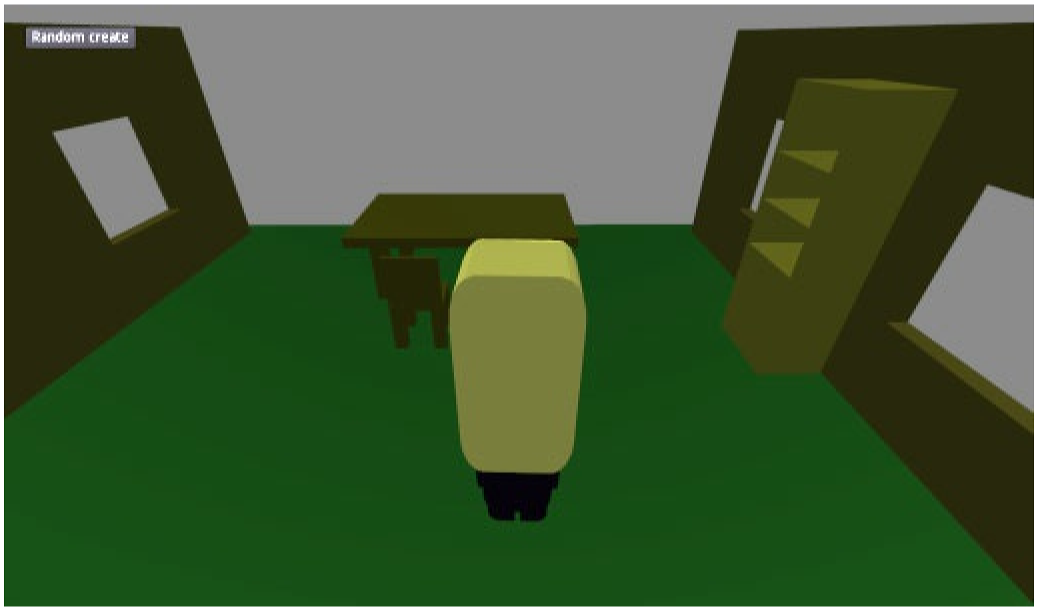


Figure 6.8: *Testing game scene*

Purpose of an intro scene

The intro scene shown in the Figure 6.8 is functional, and you can start testing. So what are the wise things to do?

- Check collision and collision shapes; Test how game characters interact with a collision of game objects.
- Work to improve vital parts of the game code.
- Test main game concepts and game-play; See how input parameters for moving compute.

We can, for example, improve the input code sequence in the process function. Let’s see the following code:

```
func _process(_delta):
if Input.is_action_pressed(“ui_down”) or
Input.is_action_pressed(“move_backwards”):
   $Character.move_and_collide( Vector3(0,0,-0.1))
if Input.is_action_pressed(“ui_up”) or
Input.is_action_pressed(“move_forward”):
   $Character.move_and_collide( Vector3(0,0,0.1))
if Input.is_action_pressed(“ui_right”) or
Input.is_action_pressed(“move_right”):
   $Character.move_and_collide( Vector3(-0.1,0,0))
if Input.is_action_pressed(“ui_left”) or
Input.is_action_pressed(“move_left”):
   $Character.move_and_collide( Vector3(0.1,0,0))
```

In all situations, when we use the **`move_and_collide`** method, we need to set a vector for stopping a movement. Look at the following code example:

```
if Input.is_action_just_released(“ui_down”):
   $Character.move_and_collide( Vector3(0,0,0))
if Input.is_action_just_released(“ui_up”):
   $Character.move_and_collide( Vector3(0,0,0))
if Input.is_action_just_released(“ui_right”):
   $Character.move_and_collide( Vector3(0,0,0))
if Input.is_action_just_released(“ui_left”):
   $Character.move_and_collide( Vector3(0,0,0))
```

I would like to suggest code improvement. For example, we can create an array and process the same input parameters in a **`for`** loop. One of the standards for input keys is WASD, and we will also use it. So, for example, W is in the input map as **`move_forward`**. This code improvement will look like the following:

```
# part for a code header
onready var act_released = [0, “ui_down”, ”move_backwards”,
”ui_up”, “move_forward”, “ui_right”, “move_right”, “ui_left”,
“move_left”]

# part for a process function
func _process(_delta):
if Input.is_action_pressed(“ui_down”) or
Input.is_action_pressed(“move_backwards”):
   $Character.move_and_collide( Vector3(0,0,-0.1))
if Input.is_action_pressed(“ui_up”) or
Input.is_action_pressed(“move_forward”):
   $Character.move_and_collide( Vector3(0,0,0.1))
if Input.is_action_pressed(“ui_right”) or
Input.is_action_pressed(“move_right”):
   $Character.move_and_collide( Vector3(-0.1,0,0))
if Input.is_action_pressed(“ui_left”) or
Input.is_action_pressed(“move_left”):
   $Character.move_and_collide( Vector3(0.1,0,0))
for i in range(1,8):
   if Input.is_action_just_released(act_released[i]):
     $Character.move_and_collide( Vector3(0,0,0))
```

Note: Delete the whole prototyping project.

Conclusion

In this chapter about 3D game making, we learned about using the Godot

IDE for game development in the 3D environment. Nevertheless, we also learned about game prototyping and how prototyping is a quality way of preparing a good video game. In the next chapter, we will deep dive into 3D game making and understand the primary process in creating a game character.

Questions

1. What does CSG stand for?
2. Why is game prototyping so important?
3. What is an intro scene?

CHAPTER 7

Project: 3D Platformer

In the early days of 3D games, developers started with closed 3D spaces like dungeons, underground shelters, or caves. We will not take this antiquated approach. Instead, as the chapter title says, we will start with a 3D platformer as our first 3D game project.

Because every good project starts with a plan, we will plan a 3D platformer. After planning, you will create a prototype scene (optional), and then we can create a 3D character. After a game character is designed and tested, we can add props and a 3D game environment.

Before we continue, explaining a 3D platformer concept will be good. In a 3D platformer game, game character move in the 3D space with different height platforms, obstacles, and bonus props. The two types of 3D platformers are standard and 2D simulated. When you play a game with 3D props made for 3D space in a 2D space view, you play a 2D simulated 3D platformer.

In this chapter, you will learn planning, character design, props, and environment adding for a 3D platformer video game.

Planning a 3D platformer

In modern 3D games, planning is an ordinary initial action. Developer teams always give a lot of time solving problems and questions before they appear in a development. Planning is also a vital part of game design for a solo developer.

What type of game do we create? What resources do we need for realization? Do we have a quality team and suitable equipment for 3D game development?

Nevertheless, what is game planning for a solo developer when we put developers aside? What will be parts of your planning if you make a 3D game for fun?

You can have two approaches. Plan everything in front, or create it step by step. If you decide for a “step by step” approach, you will create a new project, and then you will again determine what will be the next creative step.

Intuitive planner

So, we will start with a new project and then with a new 3D scene. Setting the ground and camera will be the next activity. Then, some number of meshes will define the ground.

After making a game ground generator, we can add a 3D asset as a game character and start a coding movement.

Note: Usually, in 3D games, every asset is individuality put in a 3D environment. But we can’t do it now because we are solo developers, and we learn game development—also, we want to have fun when creating a 3D project.

We will take a wise approach with ground generators, then adding a new game space will be easy. The ground generator is suitable for a 3D platformer we create, but it’s not ideal for other game types like RPG adventures.

When we finish with a movement for the main character, it will be suitable to test it. Then, we will use the test-improve approach to create a good game character. Finally, the game character will use a skeletal mesh to create various movements in a 3D space.

Adding game assets will then help in creating exciting game-play. For example, the game character can walk, run, or jump. We can slowly put game assets in a scene. It’s not necessary to set all 3D environment props at once. The game menu, audio props, music, and animations we can add after the game have good functionality (game-play).

Character design

We will have a first-time game character with 3D skeletons in this game. As Godot documentation says, support for the skeleton node is quite rudimentary (in Ver 3.1), but we can have enough manoeuvring space for our project. If we, for example, use a humanoid 3D asset as a game character, we can set and animate basic movements like walking, running, or jumping.

So, let’s find a suitable low poly 3D game character. But first, I would like to remind you that we need a rigged 3D model. Low poly, rigged, and similar terms are explained earlier in the topic 3D models in a game scene, see *Chapter 6, 3D Math and 3D Physics*.

Preparing a game scene

Now, we will prepare a game scene for a game character. So, create a new project (high visual quality) with a name like **`Game_Project_3D_platformer`**. Then, select a 3D scene, and rename the spatial node to World. Next, select the spatial node, and add a GDScript file. Finally, delete all comments, but leave the ready and process function.

You can add an underscore to the delta property for now. So, the code will look like the following:

```
extends Spatial

func _ready():
  pass

func _process(_delta):
  pass
```

It will be good to add a camera node and save a scene (**`File | Save`**).

Good, select the root node, and add a MechInstance node. Set the new CubeMesh in the inspector window. Set size as 3x0.3x3, and watch for transform values (need to be the default). Create a MeshInstance (as a child of a static body) with coding, as shown in the following example.

Example:

```
func _ready():
    var stat = StaticBody.new()
    var coll = CollisionShape.new()
    var shape = BoxShape.new()
    shape.set_extents( Vector3( 3,0.3,3))
    var mesh = MeshInstance.new()
    var mesh_type = CubeMesh.new()
    mesh_type.set_size( Vector3( 3,0.3,3))
    mesh.set_mesh( mesh_type)
    stat.add_child( mesh)
    stat.add_child( coll)
    .add_child( stat)
```

A mesh instance previously made in the IDE can be hidden or deleted. So, set a camera, and test a scene. In our camera settings, we, for example, have

0x3x4.5 for translation and -12x0x0 for rotation degrees.

Note: It’s wise to set the world.tscn as the main scene (Project settings | Run).

We need a vast space for character movement testing, and for that, we will add a few code lines. Our character will move in one direction; for example, from left to right. Our code will be helpful in this movement because we can generate ground meshes from it.

Look at the following example:

```
func _ready():
  for n in range(-12,15,3):
    # put code from the previous example
```

Set camera translation to 0x7x10, and test a game scene.

We can continue with the game scene preparation for a game character if everything is fine.

Next, we can add a material for a mesh instance. Look at the following example for game material creation:

```
var image = Image.new()
image.load(“res://texture_plastic.jpg”) # file is in ch.7
resource folder
var material = SpatialMaterial.new()
var new_texture = ImageTexture.new()
new_texture.create_from_image(image,0)
material.albedo_texture = new_texture
material.albedo_color = Color.greenyellow
mesh.set_material_override( material)
mesh.material_override.set_texture(0,new_texture)
mesh.set_mesh( mesh_type)
mesh.set_visible( true)
stat.add_child( mesh)
```

As seen in the preceding example, first, the image texture is defined. Later, we create a new spatial material and image texture. Then, the albedo material, albedo color, and mesh type are set. Finally, a mesh is a subnode to a static body.

Adding robotic game character

In this video game, we will have a moveable robotic game character. You can find it as a created game scene (**`player.tscn`**). Instantiate the player scene, and add a script file to it. The player scene has a collision capsule, robotic mesh instance, and animation player. There are animations defined with an animation player like walking, running, and jumping.

Use the move mode to put the robotic character on the early created path. Then, experiment with a camera preview till you are satisfied.

Now, we can start with coding. First, let's create gravity for a character as shown in *Figure 7.1*. Look at the following example:

```
func _physics_process(_delta):
  $”.”.move_and_collide( Vector3( 0,-0.1,0))
```

Next, we will add one side movement:

```
if Input.is_action_pressed(“ui_right”):
  $”.”.move_and_collide( Vector3( 0.06,0,0))
if Input.is_action_just_released(“ui_right”):
  $”.”.move_and_collide( Vector3( 0,0,0))
```

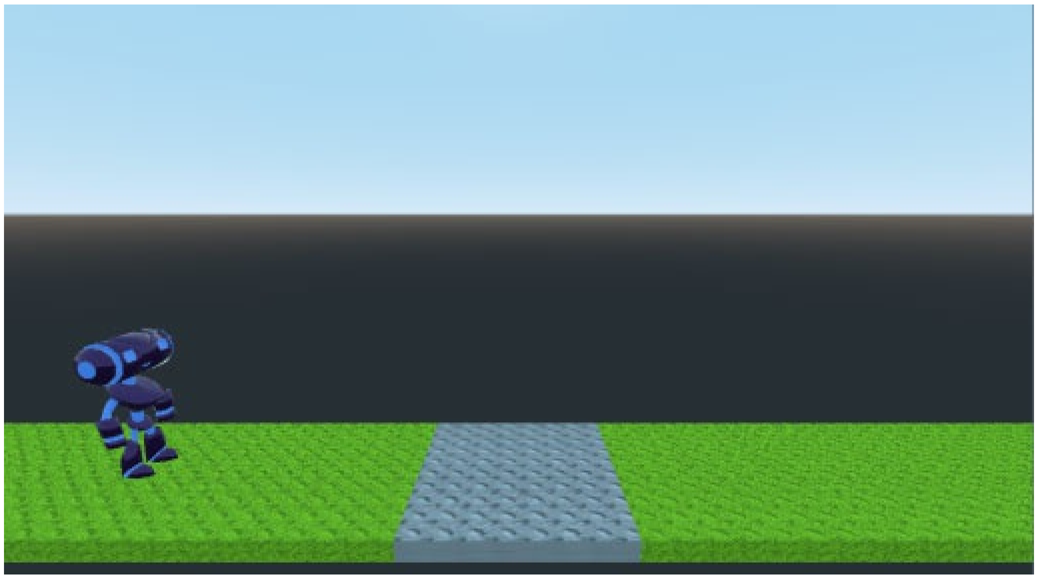


Figure 7.1: *Robotic game character*

We need to set a proper position for the character and game ground.

We can rotate the character when the movement starts. We can use predefined animations for the walking and idle states. For this, look at the following example:

```
if Input.is_action_pressed(“ui_right”):
   $”.”.set_rotation_degrees(Vector3(0,90,0))
   $”.”.move_and_collide( Vector3( 0.06,0,0))
   $AnimationPlayer.play(“walk-cycle”)
if Input.is_action_just_released(“ui_right”):
   $”.”.move_and_collide( Vector3( 0,0,0))
   $AnimationPlayer.play("idle")
 if Input.is_action_pressed("ui_left"):
   $".".set_rotation_degrees(Vector3(0,270,0))
   $".".move_and_collide( Vector3( -0.06,0,0))
   $AnimationPlayer.play("walk-cycle")
 if Input.is_action_just_released("ui_left"):
   $".".move_and_collide( Vector3( 0,0,0))
   $AnimationPlayer.play("idle")
```

Environment for a 3D platformer

Usually, 3D games have thousands of 3D game assets. So, first, assets are expensive, and second, we don't have a learning space for this approach. So, what will we do about the 3D game environment?

We can use scene instantiation and assets prototyping. First, we will create a game scene with prototyping meshes, and then we will instantiate the scene *n* times.

Nevertheless, we can create a scene with CSG nodes. For example, a house with windows, doors, stairs, and a fence will be one of our game scenes. You can add a game scene (**`3D_asset.tscn`**) with Godot IDE or use the GD Script. But, first, load a scene:

```
 var scene = load("res://3D_asset.tscn")
```

Next, add a few code lines for scene instantiation. Look at the following example:

```
 func _ready():
   var scene_instance = scene.instance()
   scene_instance.translate(Vector3( 5, 1, -5))
   add_child(scene_instance)
```

And finally, we can add many instantiate game scenes in a row. To do this, the for loop is used, as shown in the following example:

```
 func _ready():
  for i in range( 5, 35, 15):
   var scene_instance = scene.instance()
   scene_instance.translate(Vector3( i, 1, -5))
   add_child(scene_instance)
```

Coding will create a few game scenes with a house and game surrounding, as shown in *Figure 7.2*, in a row close to the game character path.

Now, we can make some changes and additions for coding.

Let's study the following code example:

```
 var scene = load("res://3D_asset.tscn")
```

```
var mov_cam = false
func _ready():
   var m = 0
   for n in range(-12,21,3):
    game_path( n,m,Color.silver)
   for n in range(-12,21,3):
    game_path( n,-3,Color.greenyellow)
   for i in range(5,35,15):
    var scene_instance = scene.instance()
    scene_instance.translate(Vector3(i,1,-5))
    add_child(scene_instance)

func game_path(n,m,color):
   var image = Image.new()
   image.load("res://texture_plastic.jpg")
   var stat = StaticBody.new()
   var coll = CollisionShape.new()
   var shape = BoxShape.new()
   shape.set_extents( Vector3( 3,0.3,3))

   var mesh = MeshInstance.new()
   var mesh_type = CubeMesh.new()
   mesh_type.set_size( Vector3( 3,0.3,3))

   var material = SpatialMaterial.new()
   var new_texture = ImageTexture.new()
   new_texture.create_from_image(image,0)
   material.albedo_texture = new_texture
   material.albedo_color = color
   mesh.set_material_override( material)
   mesh.material_override.set_texture(0,new_texture)
   mesh.set_mesh( mesh_type)
   mesh.set_visible( true)
   stat.add_child( mesh)
   stat.add_child( coll)
   .add_child( stat)
   stat.set_translation(Vector3(n,0,m))
```

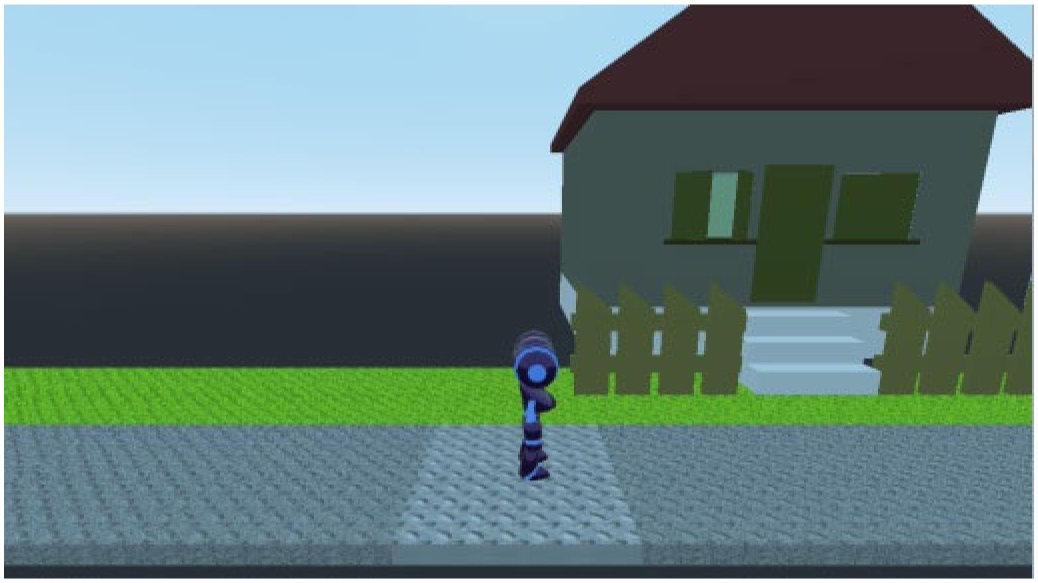


Figure 7.2: *3D platformer game environment*

Changing the camera frame

Our camera is stationary. It will be wise to create some coding for the camera frame change. For example, we can change the camera node position depending on the game character:

```
extends Spatial

var scene = load(“res://3D_asset.tscn”)
var mov_cam = false
func _process(_delta):
    if $Player.get_translation().x > 6.5 and mov_cam == false:
      var get_cam_pos = $Camera.get_translation()
      $Camera.set_translation(get_cam_pos + Vector3(6.5,0,0))
      mov_cam = true
```

First, let’s define a **`boolean mov_cam`**. Then, we need to check the player position and a Boolean value in a process function. Second, the camera node position is received. Finally, we will change the camera node position. The change depends on a previous position and some vector3 variable. Don’t forget to change the Boolean value.

So, when the game character passes some position, the camera frame will change. Now, we can add a few code lines when the game character is going back. Let’s look at the following example:

```
if $Player.get_translation().x < 1.5 and mov_cam == true:
    var get_cam_pos = $Camera.get_translation()
    $Camera.set_translation(get_cam_pos + Vector3(-6.5,0,0))
```

```
   mov_cam = false
```

It’s good to understand that the camera node changes position only when the **`mov_cam`** value is true, which means the game character is in the next camera frame.

Jump option for the game character

We can now create a jump option for our robotic game character. For now, we have solved movement to the right and left. So, we needed a jump option for a good gameplay.

The simplest solution will be to use the move and collide method with some oposite value to game gravity. You can see this in the following example:

```
 if Input.is_action_pressed(“ui_up”):
   $”.”.move_and_collide( Vector3( 0,0.21,0))
   $AnimationPlayer.play(“jump-up-cycle”)
 if Input.is_action_just_released(“ui_up”):
   $AnimationPlayer.play(“idle”)
```

Nevertheless, this solution is more like flying. To create a jump effect, we need to use something for ground detection.

So, we will use the RayCast node. Add a RayCast node to the root node in the player scene. Also, rename a node to a JumpHeight. A node needs to be enabled, and the translation values are -0.05x0.056x0.009.

Put the RayCast node below a game character if you create using your values. It will work as a collision detector. Set the cast parameter to 0x-1.5x0.

Check it visually in 3D view, and then add some coding. You can test colliding with few code lines. Look at the following example:

```
 if $JumpHeight.is_colliding() == true:
  print(“Colliding”)
 else:
  print(“Not Colliding”)
```

Now, we can add code lines for a jump option. Look at the following example:

```
 if Input.is_action_just_pressed(“ui_up”) and
 $JumpHeight.is_colliding() == true:
  $”.”.move_and_collide( Vector3( 0,1.5,0))
  $AnimationPlayer.play(“jump-up-cycle”)
 if Input.is_action_just_released(“ui_up”):
  $AnimationPlayer.play(“idle”)
```

Please test it and set jump parameters by game preference.

Game prop

Gameplay becomes the next logical task when you have a functional game level. So, we will add some game props to our environment. But, first, let’s create a new scene (GameProp) and add some nodes.

Add RigidBody to root node (name it also GameProp) with CollisionShape and CSGBox (0.6x0.6x0.6). Set a proper collision shape scale (0.3x0.3x0.3). Rigid body will be game object for picking or avoiding.

Next, add a script to the root node and Area node with the collision shape. Set the collision shape scale to be a little wider (0.45x0.45x0.45) than a rigid body’s.

Last, add a signal (**`body_shape_entered`**) to the Area node.

Now, we have a scene with a simple game prop. We need to instantiate it in the main scene and add code for functionality.

The game prop will be instanced with code lines in the main scene (World). Look at the following example:

```
extends Spatial

# Loading game prop scene
var game_prop = load(“res://GameProp.tscn”)

func _ready():
# Instancing a game prop
  var prop_instance = game_prop.instance()
  prop_instance.translate(Vector3( 3,0.5,0))
  add_child( prop_instance)
```

If testing is OK, you can add some color to the game prop to look like a cardboard box. Also, some changes in the instantiating code will be acceptable to generate more props. For example, look at the following code lines:

```
#Instancing a game prop
for i in range( 3,13,5):
   var prop_instance = game_prop.instance()
   prop_instance.translate(Vector3( i,0.5,0))
   add_child( prop_instance)
```

Go to the player scene script and add a player name line:

```
extends KinematicBody

func _ready():
  $AnimationPlayer.play(“idle”)
  # Setting a player name
  .set_name(“Player”)
```

Nevertheless, we can now solve a code for game prop picking. In a GameProp scene script, add the following lines:

```
func _on_Area_body_shape_entered(body_id, body, body_shape,
area_shape):
  # Picking a prop
  if body.get_name() == “Player”:
    .queue_free()
```

You can test your game and prop picking solution.

Initial gameplay

In 3D games, we have depth in graphics and the possibility for visual quality, but still for every game, the gameplay is one of the essential elements. Therefore, we can create some initial gameplay for the current learning project - 3D platformer.

Our game projects are peaceful, so picking a game object and avoiding a game object can be our option. However, when creating a game object for picking, we need a defined collision shape and detection script. When avoiding a game object is made, a collision shape is an option.

So, we can use the previously made game prop as a picking object.

And now, what can be our avoiding game prop? You can guess! Avoiding the game prop will be where our game character can use the jump ability. As you know, jump abilities are created and added to the player character.

Now, we can start creating a new game prop. But first, duplicate a **`GameProps.tscn`** scene. To do this, open the scene and save it with a different name. Use **`Save scene as`** from the Godot IDE main menu. Next, open a newly created game scene and start with some necessary changes. Clear the script for the root node and change a root node name to **`GameProp_jo`**. After that, you need to add a new script.

Next, change a prop type from rigid body to static body. To do this, use the context menu with the **`Change Type`** option. The collision shape needs to be after an object’s body and with a different size as shown in *Figure 7.3:*

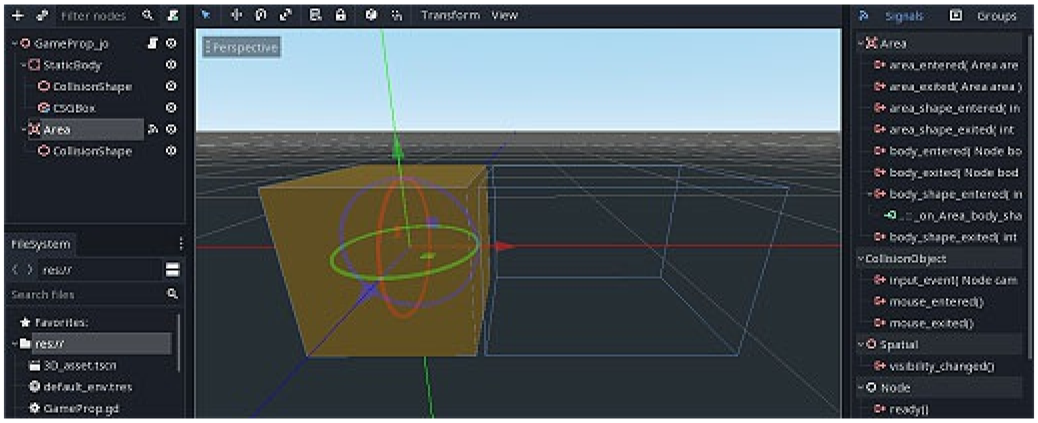


Figure 7.3: *Collision shape of an StaticBody*

Check the signal for the Area node; you can disconnect it and connect it afterwards. And finally, add some lines to the script. Look at the following example:

```
extends Spatial

func _on_Area_body_shape_entered(body_id, body, body_shape,
area_shape):
  if body.get_name() == “Player”:
    .queue_free()
    #$”/root/singleton”.pts += 30
```

The code under comment can be enabled later when we set a scoring system.

Of course, you need to test the working of this newly added game prop. For this to work, you can use the instantiation method explained earlier.

For the sake of initial gameplay, we can create a combination of two props. Look at the following code example:

```
# Instancing a game props
  for i in range( 3,13,5):
    var prop_instance = game_prop.instance()
    prop_instance.translate(Vector3( i,0.5,0))
    add_child( prop_instance)
    var prop_instance2 = game_prop_jo.instance()
    prop_instance2.translate(Vector3( i + 2,0.5,0))
    add_child( prop_instance2)
```

Don’t forget to load the necessary scenes for this code to work. Look at the following code example:

```
extends Spatial

var scene = load(“res://3D_asset.tscn”)
var mov_cam = false
var game_prop = load(“res://GameProp.tscn”)
var game_prop_jo = load("res://GameProp_jumpover.tscn")
```

Using texture for game objects

You need to make or obtain every asset for the 3D game, like a texture for a game object. In the game prototyping, it's ok to have colored objects, but later on, textured objects become imperative. Therefore, let's explain how to change a colored game object to a textured one.

First, you need to put the texture file (**`ORANGE_BRICK_DIFF.png`**) into the resource folder for the current project. Then, we need to register this image into video hardware by loading it properly. But before, we need a game object. We can add one in the 3D_asset scene. So, we can create a CSGBox node with 15 as width, 12 as height, and two as depth properties.

The node will be a wall game object because our texture simulates the wall pattern. Next, add a new spatial material to the material property. Then, go to the albedo sub-property and add an image for texture. After that, click on load and select the PNG image from the resource folder. Check *Figure 7.4*, and adjust game objects properly:

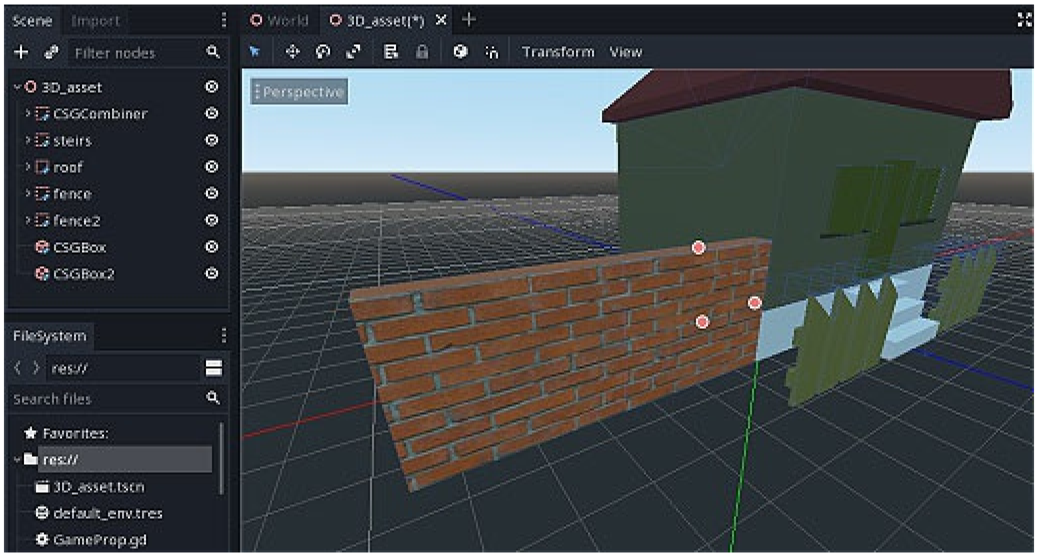


Figure 7.4: *Textured game object with the wall pattern*

Score system

Let's create a scoring system for our 3D game. First, we need a global variable, text label, and a way to calculate points. For a global variable, we will use singletons.

So, create a new code script file. Put these predefined lines as shown in the following example:

```
extends Node
var pts

func _ready():
   pts = 0
```

We have a global variable for points, so let’s add a point calculation part. We will add a code line (in signal function) in the props scene to do this. Look at the following example:

```
extends Spatial
func _on_Area_body_shape_entered(body_id, body, body_shape,
area_shape):
   print(“Entered: “ + str(body))
   if body.get_name() == “Player”:
     print(“Entered by player”)
     .queue_free()
     # Calculating points with global var
     $”/root/singleton”.pts += 10
```

And now, we only need to add one label node to the main scene. Select the 2D space tab, and add a label node. Put it somewhere in the top part of a screen. Also, add code lines to a process function. Look at the following example:

```
func _process(_delta):
   if $”/root/singleton”.pts > 0:
     $Score.set_text(“Points: “ + str($”/root/singleton”.pts))
```

We have all the necessary parts for a functional 3D video game. You can test it and add some additional game space and game props.

Conclusion

We learned a lot about 3D game making. When you know how to create a game character and game environment for a 3D game, you are ready for the next chapter. In the next chapter, we will put our current knowledge to a higher level. We will learn about level creation, 3D character making, NPC making, and gameplay.

Questions

1. How can we code generate a 3D scene?
2. What are the elements of a 3D game environment?
3. Can you describe 3D prop making?

CHAPTER 8
3D RPG Adventure

Welcome! In this chapter, we are going a step forward with making a 3D adventure. But, what does this mean?

First, in a role-playing game (RPG), the main character needs to have space for leveling. Leveling is directly related to a quest system. When the story obstacle (quest) is overcome, the main character gets experience points and can level up.

Second, the adventure game includes **Non-Playable Characters** (**NPC**) with limited abilities and options.

The preceding means that the developer needs at least two 3D game models for the playable character (PC) and NPC. We will use the **Main-Role Character** (**MRC**) as PC due to synonymous with a personal computer.

In developing an RPG, MRC is always visible, so it needs to be created. When obtaining a game model for MRC or NPC, you can choose low or high-poly models. In the current RPG adventure, we will use high-poly models. High-poly models are 3D assets with a high polygon count. Information about them is explained in the *Chapter 5: 2D Adventure*. A lot of high-poly models will drain the computer performance of every computer, so consider low-poly for beginner's game projects. We will be fantastic with a few moveable and a few more static 3D assets in our project.

And now, let's try to understand the pros and cons of low and high poly 3D game assets. With low-poly, you can make larger functional 3D maps with more focus on animations and music. On the other hand, when you create a game with high-poly assets, you need to think about hardware performance, make smaller maps for functional game-play, and always consider proper texture and lighting. And now, after this intro, let's start with game planning.

Structure

In this chapter, we will discuss the following topics:

- Game level creation
- Main role character
- Non-player character creation and coding

Objectives

After studying this unit, you should be able to create an initial 3D game level for an RPG adventure.

Game planning

We will plan three segments of RPG adventure: story, landscape, and characters. In addition, the goal will be to plan elements for one playable game map. The game story will tell us about MRC, what will be the adventure area (landscape), and NPC. When creating a landscape, prototyping will create desirable adventure areas easily. Characters can have the same model base but different appearances for better game computing.

Story

In every story, characters are things to work on it. So, we will work on the main character. Our MRC will be a human with two options for players: male and female. He (she) will be someone who does not try to fit but expresses himself (herself). We will not follow the cliches of games MRC, which obliges every demand from NPCs. Our MRC will be free and will allow others be free.

For example, MRC can have ideas for non-system living and how to realize it. He (she) will then have some interaction with NPCs and logical puzzles for resolving it. So, we will not use the avoid-obstacles method for a game-play, but something like having connections and solving puzzles.

After interactions and puzzle-solving, MRC will receive experience points and have foundations for a leveling up.

Level creation

And now, after a bit of theory, we will start with the Godot IDE and coding for a functional game level. So, you know the drill; now, create a new game project (3D RPG adventure) in ES 3.0 and the 3D game scene.

As you know, 3D game assets have 3 dimensions in the game space, and there are many different formats for 3D game models. So, if we, for example, want to put some quality 3D game objects in a game scene, we need to know about these formats.

The game engine fully supports the glTF. This format holds data about, for example, vertex and normals, but also some complex rendering data like animations. When developers need size-light and simple objects to process, the .glTF is a good choice. The glTF is a short form of Graphics Language Transmission Format.

The fbx file format is not supported by Godot's documentation, but there are some experimental stages like fbx file formats importer. In addition, data is stored as binary by fbx, thus making the format fast and efficient. It's good to know that the format holds data about model rigging and animation information.

You can use the material property with the spatial material to add some colors to game models, as shown in *Figure 8.1*:

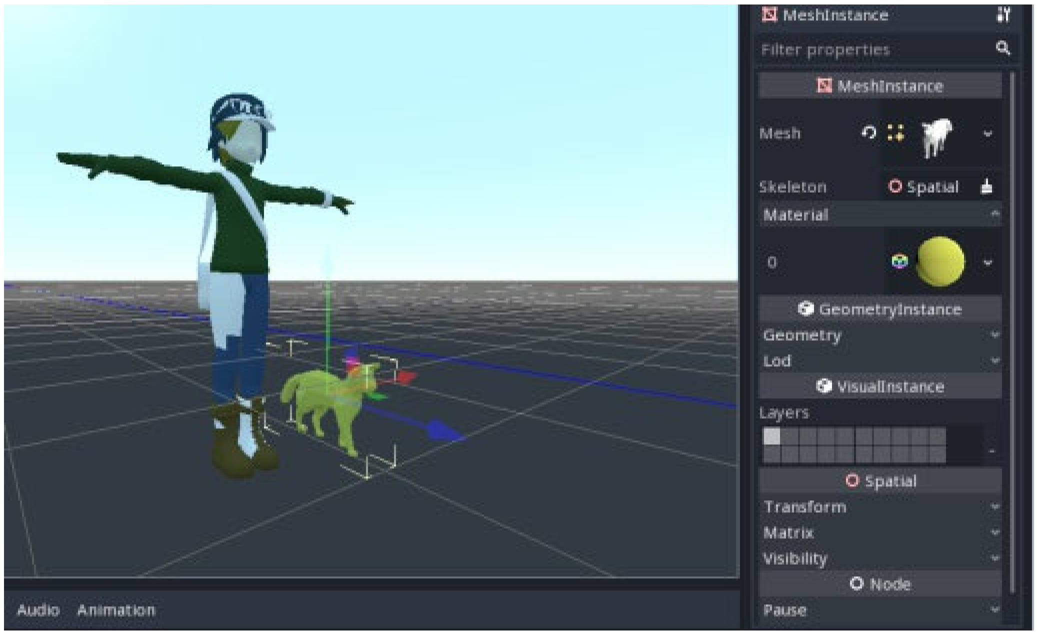


Figure 8.1: *Material property*

The **obj** is the format with info about the vertex, normals, and UV data.

The .**stl** is similar to **.obj** but with slightly fewer vertex data capabilities.

In our learning curve with 3D game assets, we will start with the **`.obj`** game models. Go to the repository folder the 3D RPG Adventure and find the **`GameCharacter`** obj file.

We will put it in the resource folder by drag and drop. Remember that two windows must be visible for drag and drop to work correctly.

You can put the model in the newly created 3D scene by drag and drop. But it's good to know how to put a model with a mesh instance. So, first, add a **`MeshInstance`** node to the scene. Then, select a node and use load mesh (you can use **`cat_3d_model.obj`** file) to add a 3D game model to the scene.

Used .obj models are suitable for learning, but we can do more with some other formats.

Game character

In the 3D RPG adventure, we will use human-like 3D game characters. We can obtain 3D assets, or we can create them. A long learning path is creating 3D game assets, so that we will start with this approach.

What do you need to know about 3D game model making?

Game models are made from model base mashes, and model base mashes are made from base mashes, like cubes or spheres.

To do this, you need an application for 3D modeling. A model base mesh is created from a cube (base mesh) in such an application. From this asset base, we created a game character.

Remember creating a 3D game model is not an easy task. You will need to spend a lot of your time and energy learning 3D modeling applications.

Our game assets will be dynamic, especially for a game character and NPCs. So, you need to know a few things about animations in the 3D game space. There are parts in 3D modeling software for animation, or you can use specialized programs.

In our creating path, we will use a game character with animation. This means we first rigged a game model and added some animations. Usually, rigging a game model defines a skeletal structure for animation (skeletal animation).

Now, let's practice as explained. First, go to the repository folder the 3D RPG Adventure and find **`Young Woman - Better Collada - For Godot`**

`Engine-.dae`. Then, open the scene as new inherited. Next, add the camera node, and set it in preview mode. Finally, save the scene as **`scene_first_ok`**. Test the scene.

The game asset is rigged and has animations so that we can set it. Create a new script file, and add the following code:

```
func _ready():
   var anim =
   get_node(“AnimationPlayer”).get_animation(“young_woman_walk”)
   anim.set_loop(true)
   $AnimationPlayer.play(“young_woman_walk”)
```

It’s advisable to watch other animations stored with the **`AnimationPlayer`** node. For example, select **`AnimationNode`**, select animation, and play animations with micro buttons. Test it.

We can add more 3D assets to the game scene. First, create CSGBox as the ground and color it (you can also color a game character). After that, add a **`Trees2.tsc`** scene to the current one (*Figure 8.2*):

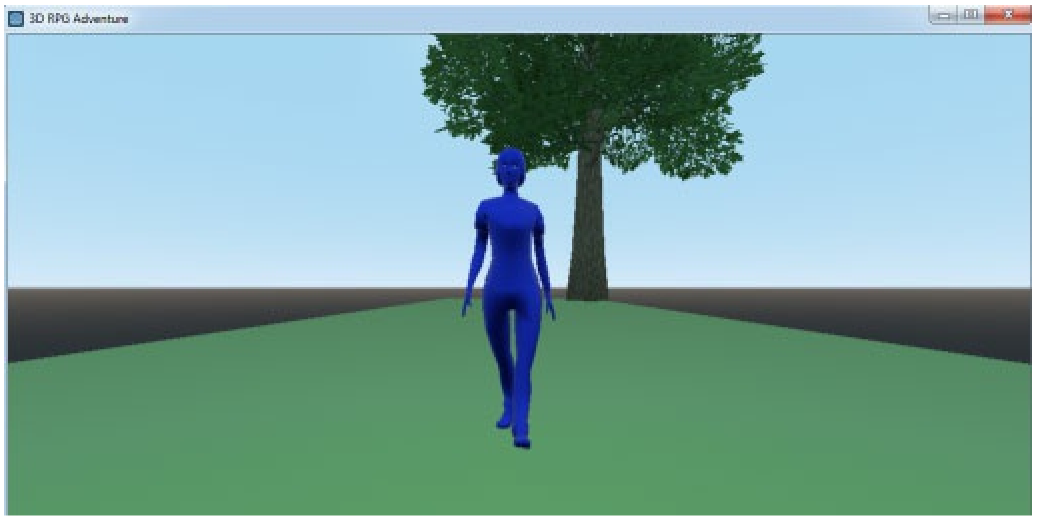


Figure 8.2: *Young girl game model with added game assets*

Intro scene

So, we can start creating an intro game scene. The scene will be dynamic, and we already have active elements. But, first, we can make movement illusions by changing the assets’ positions. We need to learn how to change a position in the 3D game space for this to happen.

All the data about game element positions is stored as transform in a 3x3 data type. All data is **`vector3`** data for translation, rotation, and scale. Preceding means that if we want to access data about the position in 3D space, we will use the translation parameter.

Let's see the following code example:

```
$trees.translation.z = -10
```

The resource (trees scene) will be given a translatory position on the Z-axis by value. Then, we can put it in the ready function. So, let's take a look at the following example:

```
func _process(_delta):
    if $trees.get_translation().z > -22:
      $trees.translation.z += -0.01
    else:
      $AnimationPlayer.play("young_woman_idle")
```

In this example, our tree scene will change the translation in delta time for some value. The change will happen if the condition is fulfilled. In a situation when the condition isn't, some animation will play. The best way to understand an example is to see it, so test this.

As you can see, the game element changes its position, creating an illusion of the MRC movement.

Now, we can add some text to the 3D scene. Writing text in a 2D design window will add it. So, select the 2D micro button, and add the sprite node. Next, find game_text.png in the repository folder, and set a scale correctly.

We will add a game button. Buttons for 3D games are added through 2D design. So, add a textured button node, and set textures for normal (btn_play.png) and hover states (btn_play_hover.png). Now, it's an excellent time to test and set this intro scene as shown in *Figure 8.3:*

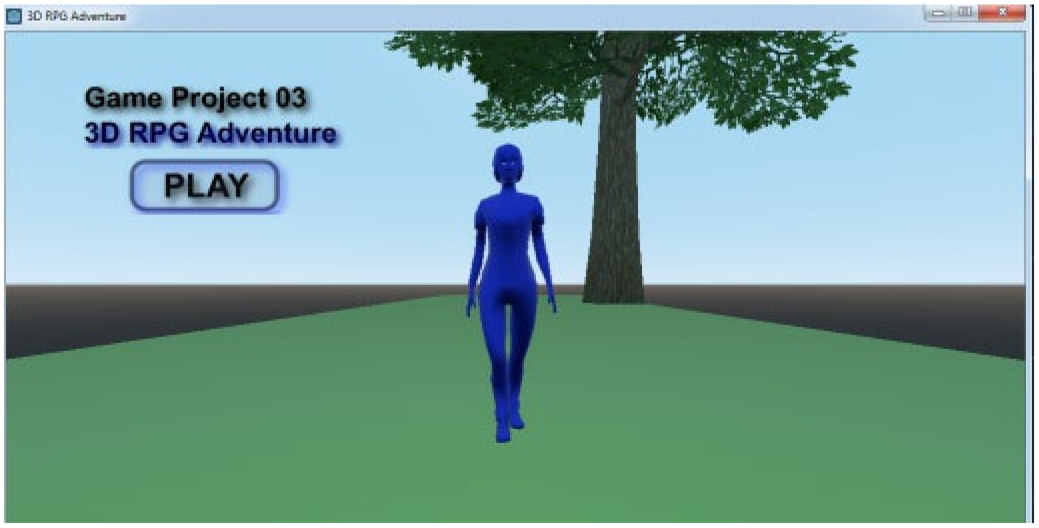


Figure 8.3: *Intro scene*

Character movement

We will use the move and slide method for character movement. We will explain two necessary parameters for the process to work. The first will define the direction, and the second will define the up position in 3D game space, as shown in the following example:

```
$KinematicBody.move_and_slide( direction, Vector3.UP)
```

Now, let’s CharOverride-12define the direction. We will use the Z-axis for the coding movement, and the value of 1 will be forward, and the value of -1 will be backward. Go to the project settings input map and define two keys: one for forward (W) and one for backward (S).

So, we can add some coding now:

```
onready var direction

func _ready():
   # initial direction
   direction = Vector3(0,0,0)
func _process(_delta):
   if Input.is_action_pressed(“forward”):
     direction.z = -1
   if Input.is_action_just_released(“forward”):
     direction.z = 0
   if Input.is_action_pressed(“backward”):
     direction.z = 1
   if Input.is_action_just_released(“backward”):
     direction.z = 0
```

We have the direction defined, so we can set object rotation while going in a different direction. For this, let's see the following example:

```
if direction.z == 1:
   $KinematicBody.set_rotation_degrees(Vector3(0,0,0))
if direction.z == -1:
   KinematicBody.set_rotation_degrees(Vector3(0,180,0))
```

When you need data about rotation in degrees, use the following code part:

```
var ro = $KinematicBody.get_rotation_degrees()
print (ro)
```

And the next is rotation; for this to happen, add the following code:

```
if direction.z == 1 or direction.z ==-1:
  anim.set_loop( true)
  $AnimationPlayer.play("young_woman_walk")
else:
  anim.set_loop( true)
  $AnimationPlayer.play("young_woman_idle")
```

With all this correctly put, we can test a game scene.

You can create a movement to the left and right now. But, first, you will define keys in settings, A for left and D for right. So, now you will have a WASD scheme for the movement keys.

Next is coding, but now you need to define data for the X-axis. For this, see the following example:

```
if Input.is_action_pressed("left"):
  direction.x = -1
if Input.is_action_just_released("left"):
  direction.x = 0
if Input.is_action_pressed("right"):
  direction.x = 1
if Input.is_action_just_released("right"):
  direction.x = 0
```

After that, you need to put code lines for character facing with rotation. See the following example:

```
if direction.x == -1:
   $KinematicBody.set_rotation_degrees(Vector3(0,-90,0))
if direction.x == 1:
   $KinematicBody.set_rotation_degrees(Vector3(0,90,0))
```

And, one more thing, animation setting. For this to be properly done, let's take a look at the following example:

```
if direction.z == 1 or direction.z ==-1 or direction.x == 1 or
direction.x == -1:
    anim.set_loop( true)
    $AnimationPlayer.play("young_woman_walk")
  else:
    anim.set_loop( true)
  $AnimationPlayer.play("young_woman_idle")
```

So, now you have solved movement in 3D space with walking animations.

RPG third-person movement style

We will continue working on 3D movement. So, let's learn how to create a 3rd person view and improve the game movement for MRC. There are many approaches for this, with many code lines, but let's try something compact.

First, we will resolve 3rd person view with a camera and the **`SpringArm`** node. Select MRC (**`KinematicBody`**) and add **`SpringArm`**. Next, add a camera node to **`SpringArm`**. You need to set the camera as current. Move **`SpringArm`** behind MRC, and put a camera above with slight down rotation, as shown in Figure 8.4. It's wise to check the camera view with the camera preview option. For example, the transform for **`SpringArm`** is 0,1.468 and -5. With this correctly done, we have the third-person view:

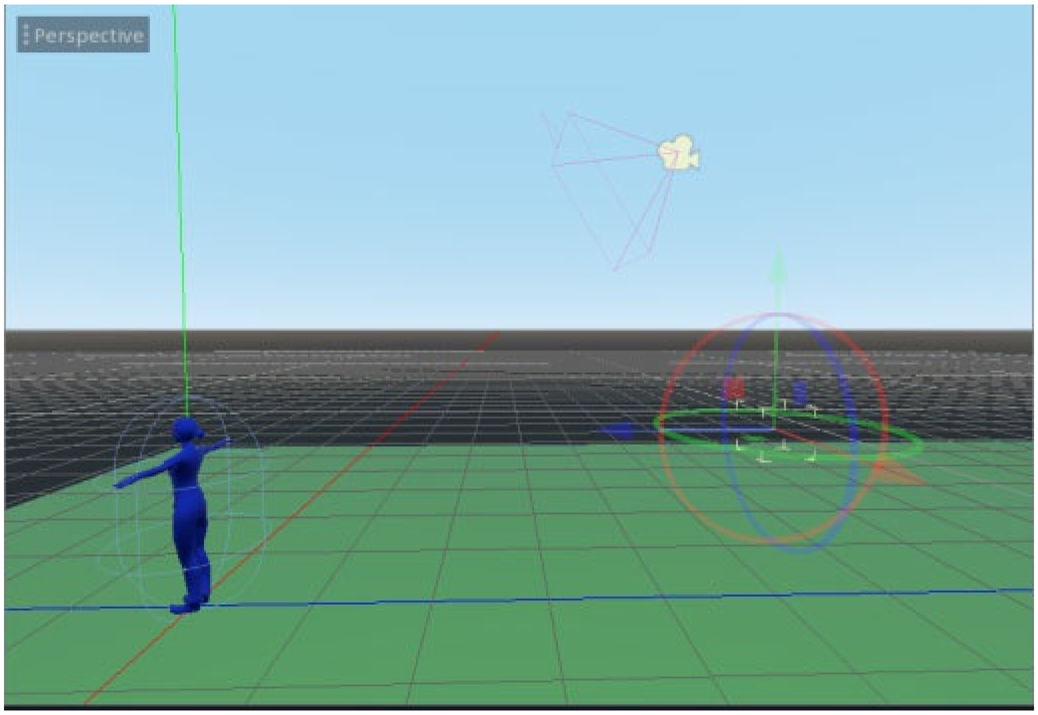


Figure 8.4: *SpringArm and camera*

Second, we will change the code lines. Let's take a look at the following example:

```
func _process(_delta):
  get_input()
  direction = direction.normalized()
$KinematicBody.move_and_slide( direction, Vector3.UP)
```

In the process function, we will normalize the direction, and use the move and slide method. All the input will be resolved in a **`get_input`** function. The previous code can be put under comment if you like to keep it or delete it.

As shown in the following example, we will create a user input function:

```
onready var speed = 1
onready var anim =
get_node(“AnimationPlayer”).get_animation(“young_woman_walk”)
onready var direction

func get_input():
    direction.z = 0
    direction.x = 0
    var diry = direction.y
    direction = Vector3()
    if Input.is_action_pressed(“forward”):
      direction += $KinematicBody.transform.basis.z * speed
    if Input.is_action_pressed(“backward”):
      direction += -KinematicBody.transform.basis.z * speed
    if Input.is_action_pressed(“left”):
      $KinematicBody.rotate_y(0.03)
    if Input.is_action_pressed(“right”):
      $KinematicBody.rotate_y(-0.03)
    if direction.z != 0 or direction.x != 0:
      anim.set_loop( true)
      $AnimationPlayer.play(“young_woman_walk”)
    else:
      anim.set_loop( true)
      $AnimationPlayer.play(“young_woman_idle”)
    direction.y = diry
```

We will use local game object values with **`rotate_y`** and **`transform.basis`** for creating the movement. The rotation for the movement is on the Y-axis. The rotation data is stored in the **`diry`** variable. So, the game character direction will be with local space z values as shown in the following example:

```
direction += $KinematicBody.transform.basis.z * speed
```

The animation will play if any axis data is above 0; look at the following example:

```
if direction.z != 0 or direction.x != 0:
    anim.set_loop( true)
```

```
   $AnimationPlayer.play(“young_woman_walk”)
```

Now, you can test a game scene.

We have a functional code for the third-person movement needed for our RPG adventure game project. If you would like to use a mouse for rotation, you need to add the following code example:

```
func _unhandled_input(event):
  if event is InputEventMouseMotion:
   KinematicBody.rotate_y(-lerp(0,0.1,event.relative.x/10))
```

In this situation, you can create a strafe left and strafe right as shown in the following example:

```
if Input.is_action_pressed(“left”):
   direction += $KinematicBody.transform.basis.x * speed
   #$KinematicBody.rotate_y(0.03)
if Input.is_action_pressed(“right”):
   direction += -$KinematicBody.transform.basis.x * speed
   #$KinematicBody.rotate_y(-0.03)
```

Initial game-play

Let’s start with a game-play creation. We already have a play button so that we can add coding. First, create a button press function with a signal, and add the following code:

```
func _on_TextureButton_pressed():
  $KinematicBody/SpringArm/CameraChar.make_current()
```

Second, make the initial camera the current from the IDE. Next, we need to set a singleton variable for enabling or disabling the movement of an MRC. Create a **`singleton.gd`** file. You can add some code lines, as shown in the following example:

```
extends Node
onready var game_start

func _ready():
   game_start = false
```

And now, you can add this variable to coding files, as shown in the following example:

```
func _process(_delta):
   if $”/root/singleton”.game_start == true:
     get_input()
     direction = direction.normalized()
     $KinematicBody.move_and_slide( direction, Vector3.UP)

func _on_TextureButton_pressed():
   $KinematicBody/SpringArm/CameraChar.make_current()
   $"/root/singleton".game_start = true
```

Test this. We can start with NPC creation now because we have the correct initial gameplay.

NPC for adventure

We will use one of the previously created 3D game characters. Find the **`npc.tscn`** file in the repository folder for this game project. In the **`npc`** game scene, we will have a game character with idle animation and a few game props. The set has a script for enabling idle animation and CSG elements for game props.

Instance the **`npc`** game scene on our main stage, as shown in *Figure 8.5*:

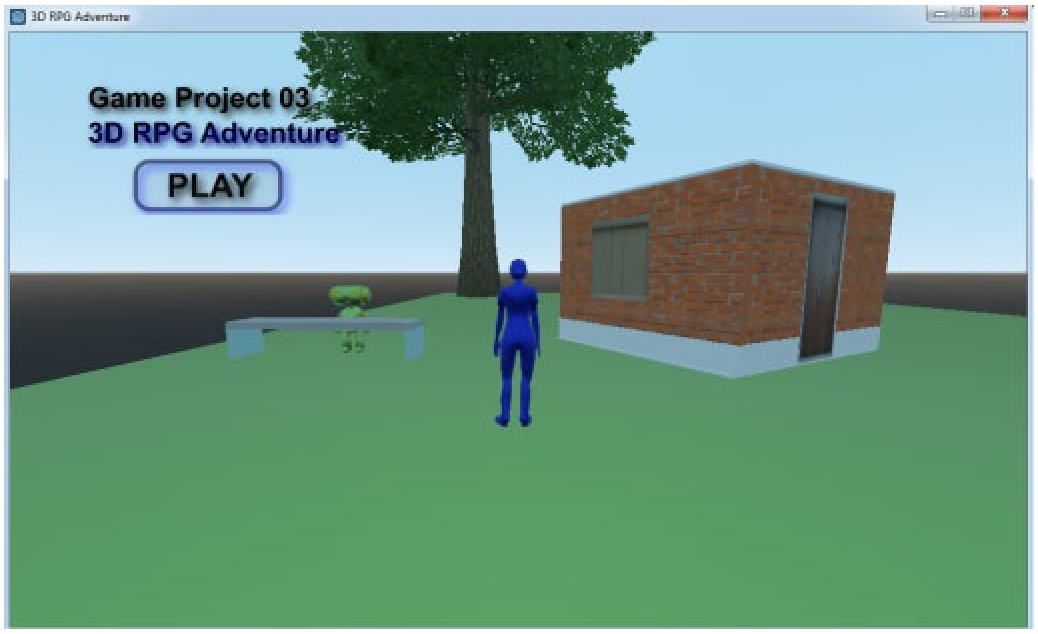


Figure 8.5: *Instanced npc in game scene*

The NPC in the game will have an idle animation and few actions when MRC interacts. Game characters and props in the **`npc`** scene have a collision shape, but you can check it by playing in it. We will use the area3D node for interaction. You can add the area node and box collision shape as a subnode. Put the collision shape in front of NPC so the interaction can happen. You can add a few code lines to test the collision shape, but don't forget to create a function with a signal. See the interaction test in the following example:

```
func _on_Interaction_area_body_shape_entered(body_id, body,
body_shape, area_shape):
  print(“Something entered!”)
```

This interaction script is good when you have one character moving around. However, it’s good to use game object names in situations with more moveable game objects.

MRC interaction

Now, we have an MRC and one NPC. So, we can create exciting interactions. Talking between game characters is one of the fundamental interactions in every RPG game. We will make speaking interactions with a few options. For example, we can use greetings.

First, set your Godot IDE to the 2D view and start with nodes adding. Next, find an **`interaciton_panel.png`** file in the project repository, and use it as texture for the sprite node. Don’t forget to add a sprite2D node first. Set this image file to cover the game screen. Click on visibility once (need to be hidden). Next, add two flat buttons as subnodes. Set the text for the first button as Hello and Thank you. Goodbye for the second. Also, add a label above, as shown in *Figure 8.6*. So, you will have the **`interaction_panel`** sprite node and three subnodes:

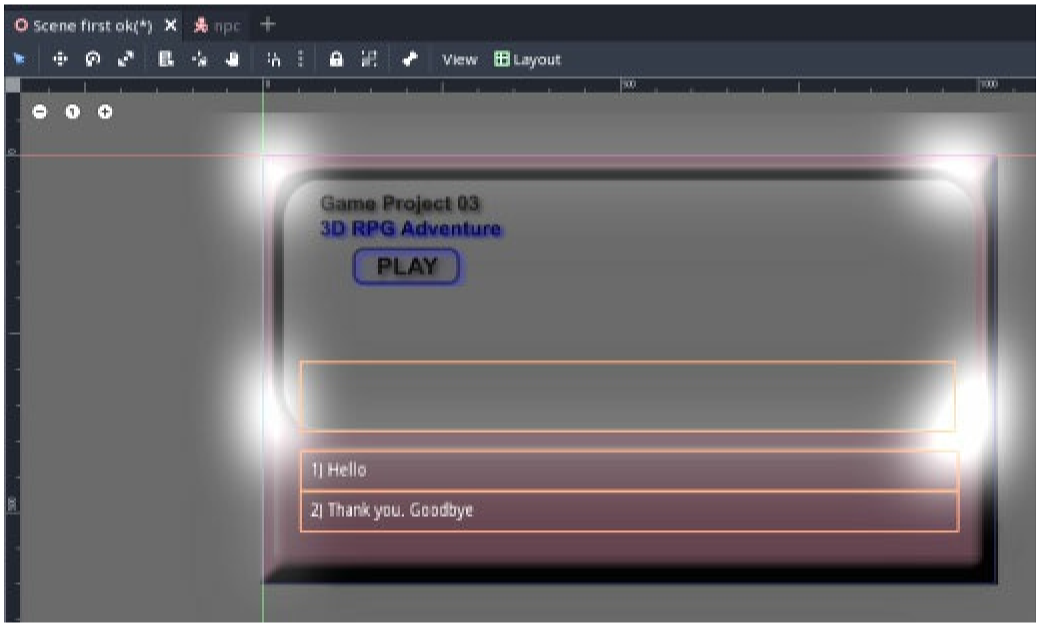


Figure 8.6: *Labels position for interaction texts*

Second, we need to do coding for the interaction panel and buttons. We will enable and disable panel visibility, and we can create signals for switches.

So, let's create one singleton variable for the interaction panel. For example, the variable **`interaction_panel`** can be used, as shown in the following code example:

```
extends Node
onready var game_start
onready var interact_panel

func _ready():
   game_start = false
   interact_panel = false
```

Next, we can add code lines for collision with the interaction game object. Go to the **`npc`** scene script, and set the code as shown in the following example:

```
extends KinematicBody

func _ready():
  $AnimationPlayer.play("idle")

func _on_Interaction_area_body_shape_entered(body_id, body,
body_shape, area_shape):
  print("Something entered!")
  $"/root/singleton".interact_panel = true

func _on_Interaction_area_body_shape_exited(body_id, body,
body_shape, area_shape):
  $"/root/singleton".interact_panel = false
```

Now, go back to the **`first scene ok`**, and create a signal for buttons. After that, we need to add code lines to the script, as shown in the following example:

```
func _process(_delta):
   if $"/root/singleton".game_start == true:
     get_input()
     direction = direction.normalized()
     $KinematicBody.move_and_slide( direction, Vector3.UP)
     #$lbl_Rotation.set_text(
     str($KinematicBody.get_rotation_degrees()) )
   if $"/root/singleton".interact_panel == true:
     $Interact_panel.set_visible(true)
   else:
     $Interact_panel.set_visible(false)
func _on_btn_hi_pressed():
   $Interact_panel/lbl_text.set_text("Hello")
func _on_btn_gb_pressed():
   $Interact_panel/lbl_text.set_text("Goodbye")
```

So now, you can test the game interaction. If everything is put right, you will have the solution for talking interaction in an RPG game. The interaction panel is visible only when MRC approaches NPC, and in-game talking is done with a button node and one label node.

Conclusion

You learned about MRC and NPC creating a process for an RPG adventure-type video game. We made an entire movement in the 3D game space and animated different movement types. Chapter know-how will help us create more exciting and complex game elements such as inventory and save systems in the book's next chapter - *Game Systems in the 3D RPG Adventure. Readers* will have the opportunity to learn about game systems such as game save system and game inventory systems. Mentioned game systems will give the necessary elements for a good 3D game.

Questions

1. Why do we create MRC at the beginning of the 3D game?
2. What is NPC in a 3D RPG game?
3. How to create an interaction in the 3D game space?

CHAPTER 9

Game Systems in a 3D RPG Adventure

A 3D RPG game is technically demanding. Therefore, this chapter will explain two game systems for the previously created game. First, we will learn how to create a game inventory system. Later, the save method will be described, and finally, we will introduce the know-how to create a standalone game project.

So, what is a game inventory system? One of the popular traits for RPG is a game object inventory. Things needed for players and quest solving are put in different game container types like bags, chests, or shelves. Then, the player can arrange them or transfer them from one container to another.

As you can guess, this chapter will introduce some of the most advanced uses of the GD Script. We will work with different variables, arrays, loops, and conditional branching to create a game inventory system and save system for our 3D RPG adventure.

You will need a lot of resource files from the repository folder of this game project. So, can you find it before we start with coding?

Structure

In this chapter, we will discuss the following topics:

- Game inventory system
- Game inventory functionality
- Inventory system implementation
- Save system
- Game publishing

Objectives

After studying this unit, you should be able to design and code a game inventory system and save system for a video game. You will also learn how to publish video games.

Game inventory system

First, let’s explain how we will create an exciting inventory system. We will use grid container nodes and textured buttons as a base for the game inventory system. We will code almost everything, so the only things you can do in the IDE are make a new scene and add two nodes.

So, add a grid container and a sprite2D node.

All coding will be in a root node. Add a script.

First, let’s see the necessary variables in the following example:

```
extends Spatial
onready var texture = preload(“res://invent_field.png”)
onready var fields =
[0,”if_1”,”if_2”,”if_3”,”if_4”,”if_5”,”if_6”,”if_7”,”if_8”,”if_9’
onready var btn
export var fields_in_line = 4
export var no_of_fields_MAX_12 = 4
export var vert_pos = 360
onready var inv_elelent_1 = preload(“res://inv_element_1.png”)
onready var inv_elelent_2 = preload(“res://inv_element_2.png”)
onready var inv_elelent_3 = preload(“res://inv_element_3.png”)
onready var eleinfield_1 = preload(“res://eleinfield_1.png”)
onready var eleinfield_2 = preload(“res://eleinfield_2.png”)
onready var eleinfield_3 = preload(“res://eleinfield_3.png”)
onready var field_state =
[“0”,”knive”,”solar”,”bottle”,”nt”,”nt”,”nt”,”nt”,”nt”,”nt”,”nt”,
onready var mouse_pos
onready var calc_pos = false
onready var click_state = 0
onready var moving_obj
onready var moving_obj_image
```

Few variables will be used as texture links as shown in *Figure 9.1*:

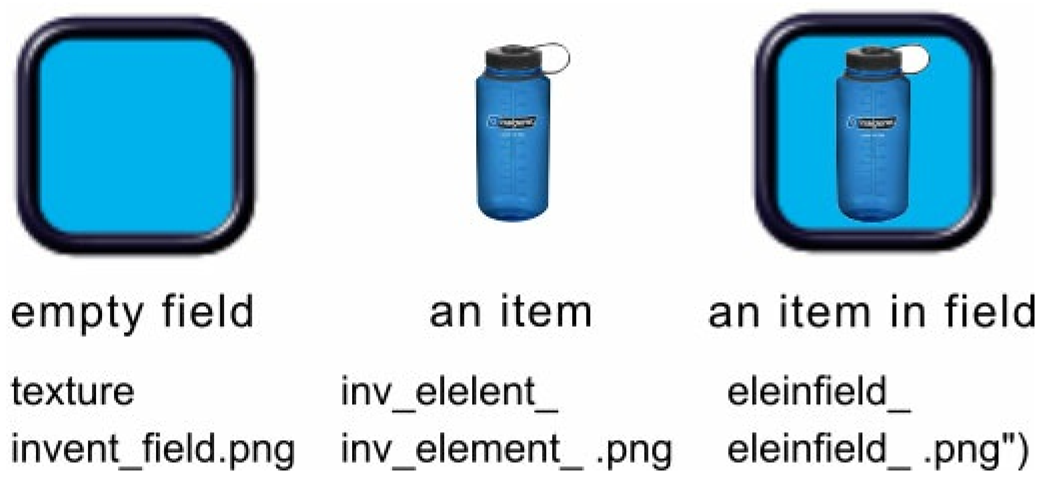


Figure 9.1: *Texture for textured buttons as item containers*

Now, we can explain these variables. First, preloaded image files will be used as a texture for textured button nodes. The initial texture for a button will be from the var **`texture`** when an item is in the inventory field; we will use var **`eleinfield`**, and to move an object, we will use the var **`inv_elelent`** as shown in Figure 9.1.

Secondly, every button we create will have a different name. Button names are in the fields array. The maximum number of fields is in an export variable - **`no_of_fields_MAX_12`**. Therefore, we can now see the code for button creation in the following example:

```
 for i in range( 1, no_of_fields_MAX_12):
    btn = TextureButton.new()
    btn.set_position(Vector2( i, 0))
    btn.set_normal_texture( texture)
    btn.set_name( fields[i])
    var btn_txt = “pressed_” + str(fields[i])
    btn.connect(“button_down”,self,btn_txt)
    $GridContainer.add_child( btn)
```

We use a grid container node with a defined number of columns in the **`field_in`** line var. So, now our ready function looks like the following example:

```
 func _ready():
    if no_of_fields_MAX_12 > 12:
     fields_in_line = 12
    $GridContainer.set_columns(fields_in_line)
    $GridContainer.set_position(Vector2(0,vert_pos))
    print_tree()
    # generate textured buttons as inventory fields
    for i in range( 1, no_of_fields_MAX_12):
```

```
    btn = TextureButton.new()
    btn.set_position(Vector2( i, 0))
    btn.set_normal_texture( texture)
    btn.set_name( fields[i])
    var btn_txt = “pressed_” + str(fields[i])
    btn.connect(“button_down”,self,btn_txt)
    $GridContainer.add_child( btn)
```

And now, we can add items to the inventory, as shown in the following example:

```
# Adding some element in the inventory
$GridContainer/if_1.set_normal_texture( eleinfield_1)
field_state[1] = “knive”
$GridContainer/if_2.set_normal_texture( eleinfield_2)
field_state[2] = “solar”
$GridContainer/if_3.set_normal_texture( eleinfield_3)
field_state[3] = “bottle”
```

We use the previously generated names of buttons to address item inputs.

Game inventory functionality

The game system functionality! This part can be tricky to solve and explain, so we will go slow.

First, we will add defined functions when the buttons are pressed:

```
func pressed_if_1():
   about_field(1)
func pressed_if_2():
   about_field(2)
func pressed_if_3():
   about_field(3)
func pressed_if_4():
   about_field(4)
func about_field(no):
print(“In field no “ + str(no) + “: “ + str(field_state[no]) )
```

Next, we will use a conditional branch to check the state of a field and var for the movement state defined as the **`click_state`** var:

```
# Check if an item is in the item field
if field_state[no] !=”” and click_state == 0:
   if field_state[no] == “knive”:
   .get_node(“GridContainer/” +
   str(fields[no])).set_normal_texture( texture)
   moving_obj = “knive”
   $inv_image.set_texture( inv_elelent_1)
```

The previous code example checks the item, and if an item is appropriate, the field becomes empty by changing the button texture. Later, the moving object is defined, and the moving image has adequate texture. Sprite textures are shown in *Figure 9.2*:

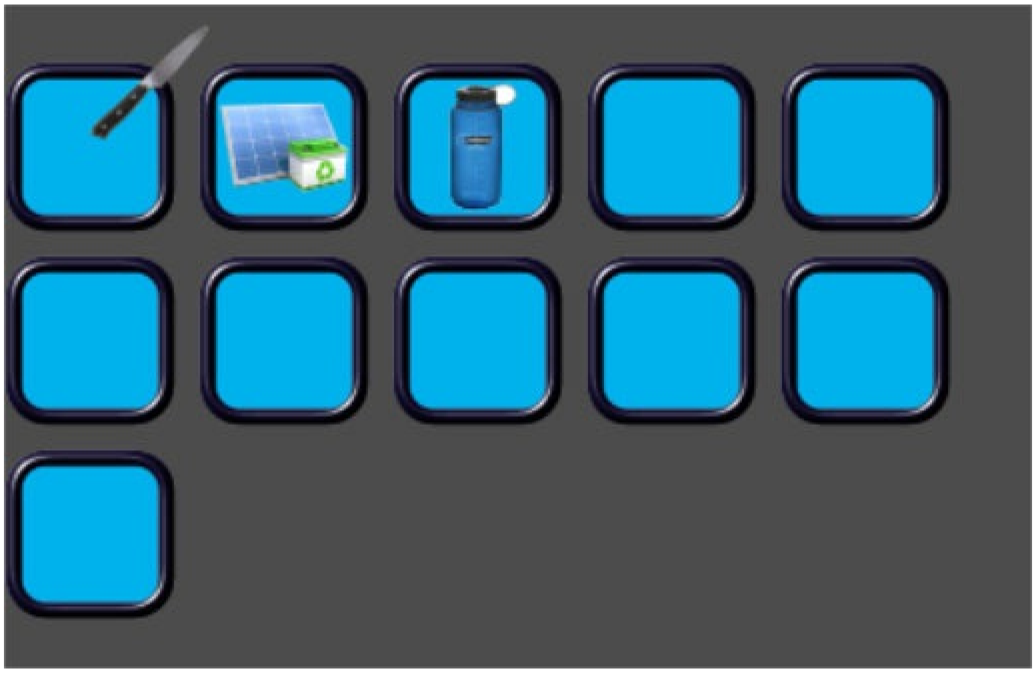


Figure 9.2: *Sprite texture for moving item image*

We also need to set the moving image visibility to calculate the position of the moving item, and define the field state. So, our code part now looks like the following example:

```
if field_state[no] !=”” and click_state == 0:
   if field_state[no] == “knive”:
     .get_node(“GridContainer/”
     +str(fields[no])).set_normal_texture( texture)
     moving_obj = “knive”
     $inv_image.set_texture( inv_elelent_1)
   $inv_image.set_visible( true)
   calc_pos = true
   click_state = 1
   field_state[no] = “nt”
```

When we calculate the mouse position (**`calc_pos`** var) for a moving item, the process function is essential. Look at *Figure 9.2*. Therefore, we set the code as shown in the following example:

```
func _process(_delta):
   if calc_pos == true:
     mouse_pos = get_viewport().get_mouse_position()
     inv_image.set_position(Vector2(mouse_pos.x,mouse_pos.y))
```

Our **`click_state`** var is now **`1`**, and we can code the script for setting a moving item at the chosen inventory field. So, let’s see the following example:

```
 if field_state[no] == “nt” and click_state == 1:
    if moving_obj == “knive”:
      .get_node(“GridContainer/” +
      str(fields[no])).set_normal_texture(eleinfield_1)
      field_state[no] = “knive”
    $inv_image.set_visible(false)
    calc_pos = false
    click_state = 3
    moving_obj = “nt”
```

If the field is empty (var value is **`nt`**) and if an object is defined (**`knive`**), we put some parameters. First, the texture of the field is changed to a moving object. After that, the field state is changed (to a moving object). Moving an object is not visible because the movement is done. Now, our code will look like the following example:

```
 func about_field(no):
    print(“In field no “ + str(no) + “: “ + str(field_state[no])
    )
    print(“Click state is: “ + str(click_state))
    if field_state[no] == “nt” and click_state == 1:
      if moving_obj == “knive”:
        .get_node(“GridContainer/” +
        str(fields[no])).set_normal_texture(eleinfield_1)
        field_state[no] = “knive”
      $inv_image.set_visible(false)
      calc_pos = false
      click_state = 3
      moving_obj = “nt”
    if field_state[no] !=”” and click_state == 0:
      if field_state[no] == “knive”:
        .get_node(“GridContainer/” +
        str(fields[no])).set_normal_texture( texture)
        moving_obj = “knive”
        $inv_image.set_texture( inv_elelent_1)
      $inv_image.set_visible( true)
      calc_pos = true
      click_state = 1
      field_state[no] = “nt”
    if click_state == 3:
    click_state = 0
```

We can add a few more code lines to resolve the movement of other item types. For example, let’s see the following code part:

```
if field_state[no] == “nt” and click_state == 1:
   if moving_obj == “solar”:
    .get_node(“GridContainer/” +
    str(fields[no])).set_normal_texture(eleinfield_2)
    field_state[no] = “solar”
   if moving_obj == “bottle”:
    .get_node(“GridContainer/” +
    str(fields[no])).set_normal_texture(eleinfield_3)
    field_state[no] = “bottle”
   $inv_image.set_visible(false)
   calc_pos = false
   click_state = 3
   moving_obj = “nt”
```

A similar code can be added when the **`click_state`** is **`0`**. So, let’s take a look at the following example:

```
if field_state[no] == “solar”:
   .get_node(“GridContainer/” +
   str(fields[no])).set_normal_texture( texture)
   moving_obj = “solar”
   $inv_image.set_texture( inv_elelent_2)
if field_state[no] == “bottle”:
   .get_node(“GridContainer/” +
   str(fields[no])).set_normal_texture( texture)
   moving_obj = “bottle”
   $inv_image.set_texture( inv_elelent_3)
```

We have the necessary elements for the item inventory script, and you can see them in the following example:

```
extends Spatial

onready var texture = preload(“res://invent_field.png”)
onready var fields =
[0,”if_1”,”if_2”,”if_3”,”if_4”,”if_5”,”if_6”,”if_7”,”if_8”,”if_9’
onready var btn
export var fields_in_line = 4
export var no_of_fields_MAX_12 = 4
export var vert_pos = 360
onready var inv_elelent_1 = preload(“res://inv_element_1.png”)
onready var inv_elelent_2 = preload(“res://inv_element_2.png”)
onready var inv_elelent_3 = preload(“res://inv_element_3.png”)
onready var eleinfield_1 = preload(“res://eleinfield_1.png”)
onready var eleinfield_2 = preload(“res://eleinfield_2.png”)
onready var eleinfield_3 = preload(“res://eleinfield_3.png”)
onready var field_state =
[“0”,”knive”,”solar”,”bottle”,”nt”,”nt”,”nt”,”nt”,”nt”,”nt”,”nt”,
onready var mouse_pos
onready var calc_pos = false
onready var click_state = 0
onready var moving_obj
onready var moving_obj_image
func _ready():
   if no_of_fields_MAX_12 > 12:
    fields_in_line = 12
   $GridContainer.set_columns(fields_in_line)
   $GridContainer.set_position(Vector2(0,vert_pos))
   print_tree()
   # generate textured buttons as inventory fields
   for i in range( 1, no_of_fields_MAX_12):
    btn = TextureButton.new()
    btn.set_position(Vector2( i, 0))
    btn.set_normal_texture( texture)
    btn.set_name( fields[i])
    var btn_txt = “pressed_” + str(fields[i])
    btn.connect(“button_down”,self,btn_txt)
    $GridContainer.add_child( btn)
   # Adding some element in the inventory
   print_tree()
   $GridContainer/if_1.set_normal_texture( eleinfield_1)
   field_state[1] = “knive”
   $GridContainer/if_2.set_normal_texture( eleinfield_2)
   field_state[2] = “solar”
   $GridContainer/if_3.set_normal_texture( eleinfield_3)
   field_state[3] = “bottle”
func _process(_delta):
   if calc_pos == true:
    mouse_pos = get_viewport().get_mouse_position()
    $inv_image.set_position(Vector2(mouse_pos.x,mouse_pos.y))
   if $”/root/singleton”.gridc_visible == true:
    $GridContainer.set_visible(true)
   else:
    $GridContainer.set_visible(false)
func pressed_if_1():
   about_field(1)
func pressed_if_2():
   about_field(2)
 func pressed_if_3():
   about_field(3)
func pressed_if_4():
   about_field(4)
func about_field(no):
   print(“In field no “ + str(no) + “: “ + str(field_state[no])
   )
   print(“Click state is: “ + str(click_state))
   if field_state[no] == “nt” and click_state == 1:
    if moving_obj == “knive”:
```

```
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture(eleinfield_1)
    field_state[no] = "knive"
   if moving_obj == "solar":
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture(eleinfield_2)
    field_state[no] = "solar"
   if moving_obj == "bottle":
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture(eleinfield_3)
    field_state[no] = "bottle"
   $inv_image.set_visible(false)
   calc_pos = false
   click_state = 3
   moving_obj = "nt"
  if field_state[no] !="" and click_state == 0:
   if field_state[no] == "knive":
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture( texture)
    moving_obj = "knive"
    $inv_image.set_texture( inv_elelent_1)
   if field_state[no] == "solar":
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture( texture)
    moving_obj = "solar"
    $inv_image.set_texture( inv_elelent_2)
   if field_state[no] == "bottle":
    .get_node("GridContainer/" +
    str(fields[no])).set_normal_texture( texture)
    moving_obj = "bottle"
    $inv_image.set_texture( inv_elelent_3)
   $inv_image.set_visible( true)
   calc_pos = true
   click_state = 1
   field_state[no] = "nt"
  if click_state == 3:
   click_state = 0
```

Inventory game system implementation

It's good to add an inventory system scene to our main scene. We can do it with the IDE. Also, we need to add a variable for inventory system visibility. We add a **`gridc_visible`** var to the singleton script. In-game implementation is shown in *Figure 9.3:*

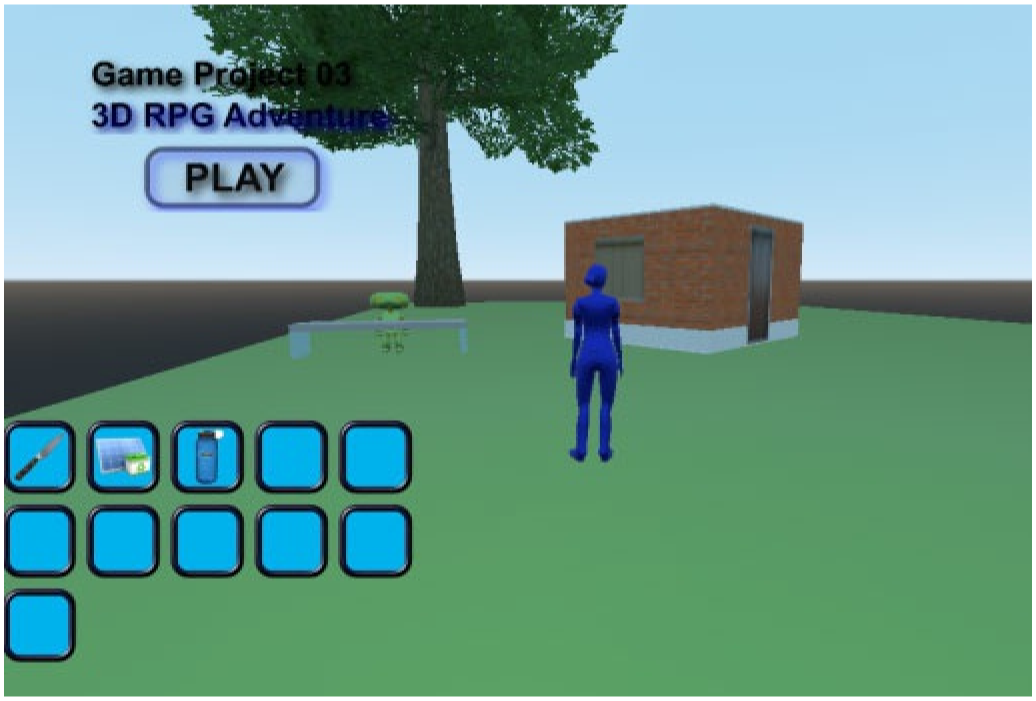


Figure 9.3: *Implemented game inventory system*

Now, set the start value in the **`ready`** function, as shown in the following example:

```
 func _ready():
    direction = Vector3(0,0,0)
    anim.set_loop(true)
    $AnimationPlayer.play(“young_woman_idle”)
    $”/root/singleton”.gridc_visible = false
  Also, add a value defined in the start game function, as shown
  in the following example:
 func _on_TextureButton_pressed():
    $KinematicBody/SpringArm/CameraChar.make_current()
    $”/root/singleton”.game_start = true
    $”/root/singleton”.gridc_visible = true
```

And now, you can start with testing and troubleshooting the inventory system. After the intro scene, we will have a functional inventory system when the coding is correct.

Game save system

Usually, a video game has an option to save the player’s progress so that the player can continue later. There are big games with many save slots. We will create a save system for different data types. Our system will have save slots.

Let’s start with the new user interface scene. First, rename the root node to save system. Then, add a script to the root node. Add the center container and vertical box container as a subnode. Later, you can add one label and a few button nodes. Set custom constants separation to 9 at the inspector window of the vertical box container.

And now, coding. We will temporarily store data in an array, and for the save system, we can use a **`.dat`** or **`.txt`** file. As shown in the following example, let’s set some variables for file node, save path, and save content:

```
extends Control

var file
var content = [0,3,”text”,9,”Some test text.”]
var path
First, we will create code for saving content data.
func _ready():
  file = File.new()
  path = “C://save_game.dat”
  # user://save_game.dat path is at user appdata Godot project
  data
  # C://save_game.dat can be used at Windows system at root
  directory
  file.open(path, File.WRITE)
  for i in range(0,len(content)):
   file.store_line(str(content[i]))
  file.close()
```

As you can see, the array data is in different lines of a **`.dat`** file. So, when using a file node, it’s imperative to use the close method in the end. So, we first open a file in a defined path in the previous example. Later, data is stored with the **`store_line`** method, and we close a file in the end.

The next step is retrieving data. Let’s look at the following example:

```
file.open(path, File.READ)
   for _i in range(0,len(content)):
    var pr = file.get_line()
    print(pr)
   file.close()
```

We open the file, read it line by line, and print it in the output window. You can code to put each line in array fields.

Now, we will add the save script to one of the buttons. Select the first button, create a signal, and put the previously created script as shown in the following example:

```
func _on_save_slot_1_button_down():
```

```
file = File.new()
path = “C://save_game.dat”
file.open(path, File.WRITE)
for i in range(0,len(content)):
 file.store_line(str(content[i]))
file.close()
```

Set the script according to your system. Later, you can add the scene to the main scene. As a game developer, you will understand what and how in the world of video game creation.

Game publishing

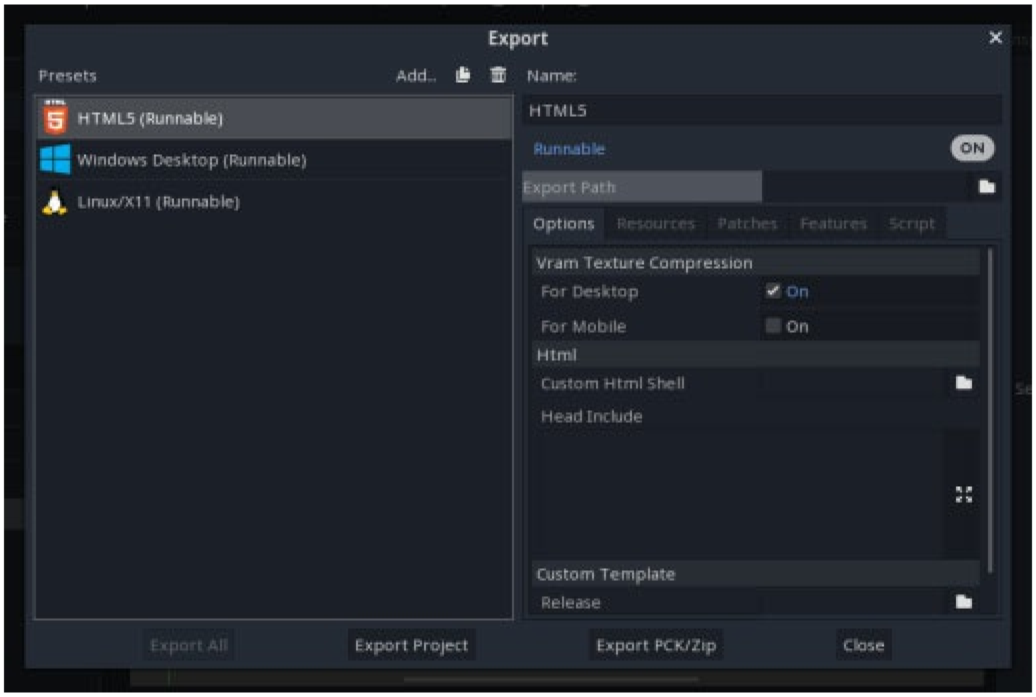
One of the great Godot traits is game publishing. You only need to define a system for gameplay, and other things are primarily automatic.

First, the game needs to have the initial scene defined. For this, go to project settings and set the main scene in the run tab. Then, go to the config file and put the game icon. Optionally, you can set (width/height) gametext.png as a bootsplash image.

Second, go to the project export. Add the operating systems. Next, go to manage export templates if you didn’t set it before. You will have the option to download the necessary files.

And that’s all, set the project name and export location and click on the export project.

Exported files will work on a defined operating system. If you decide to create a web game with an HTML5 exporter, you will need additional settings in the web location. The game publishing dialog is shown in *Figure 9.4*:



HTML5 (Runnable)
Windows Desktop (Runnable)
Linux/X11 (Runnable)

Name: HTML5
Runnable ON
Export Path

Vram Texture Compression
For Desktop ✓ On
For Mobile On
Html
Custom Html Shell
Head Include
Custom Template
Release

Figure 9.4: *Game publishing dialog*

When exporting files, you can optionally set if you need the debug files or not.

Conclusion

The 3D RPG Adventure project teaches how to create some of the most demanding video games for the Godot game engine. The engine can solve many 3D objects and movements in 3D space, but it's wise to understand the limits. The inventory and save system are a vital part of many quality video games, so it's good to understand them and modify them for the project needs.

This learning and playing journey are close to its end. But every ending is a new beginning. You can continue with learning paths from my other books or start with a game project. A suggestion is to combine learning (books, video tutorials) with game creation. Start with a 2D game, and when you create it entirely, work on improvements. You can learn a lot in the 2D game creation process. Then, with a large know-how base, you will have many options in 2D and 3D game development with the Godot game engine.

Questions

1. What are the essential nodes for the inventory system we used?
2. What’s the way to create many nodes?
3. Which method do we use to reference user-defined nodes?

Index