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

GODOT FROM ZERO TO PROFICIENCY

(PROFICIENT)

PATRICK FELICIA

Godot from Zero to Proficiency (Proficient)
First Edition

A step-by-step guide to creating your first 3D Role-Playing Game.

GODOT FROM ZERO TO PROFICIENCY (PROFICIENT)

First edition. February 15, 2022.

ISBN: 979-8201345402

Written by Patrick Felicia.

Table of Contents

PATRICK FELICIA

Godot from Zero to Proficiency (Proficient)

- First published: February 2022.

Published by Patrick Felicia

Credits

Author & Editor: Patrick Felicia

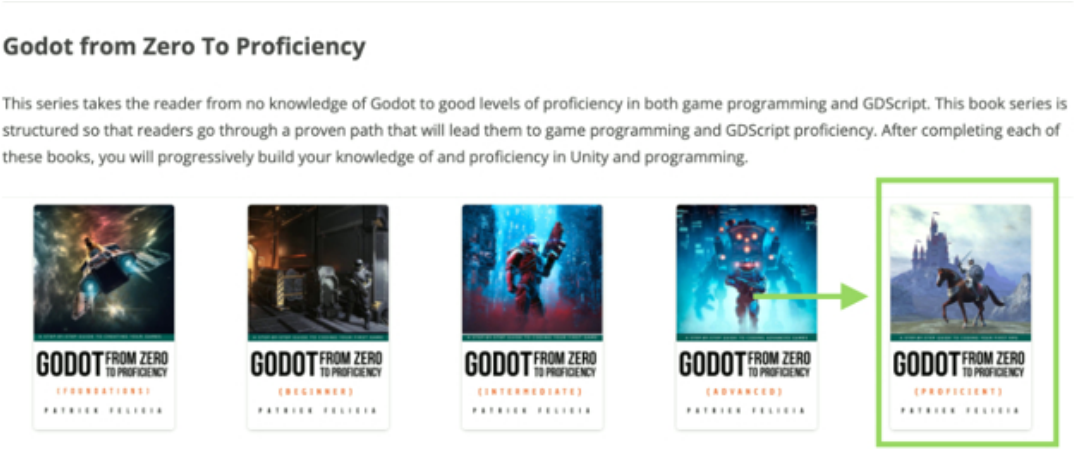
About the Author

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

Support and Resources for this Book

To complete the activities presented in this book you need to download the startup pack on the companion website (if you are already a subscriber, you just need to enter the member area - **http://learntocreategames.com/member-area/**); it consists of free resources that you will need to complete your projects. To download these resources, please do the following:

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



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

This book is dedicated to Mathis

Table of Contents

Preface

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

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

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

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

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

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

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

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

Content Covered by this Book

- Chapter 1: Introduction to the RPG Genre, Key Features and Design Concepts, provides general information about RPGs and the key features that can be found in this type of game. It also explains the features that you will be creating in this book, along with the philosophy behind the code covered.
- Chapter 2: Creating and Animating the Main Character, explains how you can create and animate your main 3D character, add a camera that will follow this character, as well as a mini-map. You will also create the 3D environment for your quest.
- Chapter 3: Creating a Dialogue System, explains how to create a dialogue system from a JSON file, and how to integrate it seamlessly into your game.
- Chapter 4: Creating an Inventory System, explains how you can create a simple inventory system and use it to collect, store, and use items that you will find in your quest. You will also learn to create a health bar.
- Chapter 5: Creating a Shop and a Buying System, shows you how to create a shop where the player can buy items that will then be added to the inventory.
- Chapter 6: Adding Weapons and Protagonists, explains how you can create intelligent NPCs that will challenge the player.
- Chapter 7: Adding a Quest System, explains how you can create a quest system based on a JSON file to manage the objectives for each of your levels.
- Chapter 8: Creating an XP Attribution System, explains how you can create a user interface through which players can use their XPs to increase their skills.
- Chapter 9: Creating The Final Level, explains some interesting features that you can use to increase your game play.
- *Chapter 10, Your Next Steps,* provides tips on how to extend your RPGs.
- *Chapter 11* summarizes the topics covered in this book and also provides useful information if you would like to progress further with this book series.

What you Need to Use this Book

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

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

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

Who this book is for

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

1. Are you a total beginner in RPG games?
2. Would you like to become proficient in the core functionalities offered by Godot?
3. Would you like to teach game programming to students or help your child to understand how to create games and learn coding in the process?
4. Would you like to start creating great RPG games?
5. Although you may have had some prior exposure to Godot, would you like to delve more into Godot and understand its core functionalities in more detail?

Who this book is not for

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

1. Can you already easily create a 3D RPG with Godot?
2. Are you looking for a reference book on GDScript?
3. Are you an experienced (or at least advanced) RPG developer?

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

How you will learn from this Book

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

- A list of the learning objectives at the start of each chapter so that readers have a snapshot of the skills that will be covered.
- Each section includes an overview of the activities covered.
- Many of the activities are step-by-step, and learners are also allowed to engage in deeper learning and problem-solving skills through the challenges offered at the end of each chapter.
- Each chapter ends up with a quiz and challenges through which you can put your skills into practice, and see how much you know.
- The book focuses on the core skills that you need. Some sections also go into more detail. However, once the concepts have been explained, links are provided to additional resources, if and where necessary.

Format of each Chapter and Writing Conventions

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

Special Notes

Each chapter includes resource sections so that you can further your understanding and mastery of Godot. These include:

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

The author's notes appear as described below:

The author's suggestions appear in this box.

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

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

How Can You Learn Best from this Book

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

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

- **Do the exercises.**

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

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

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

- **Export your games early.**

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

- **Learn in chunks.**

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

Feedback

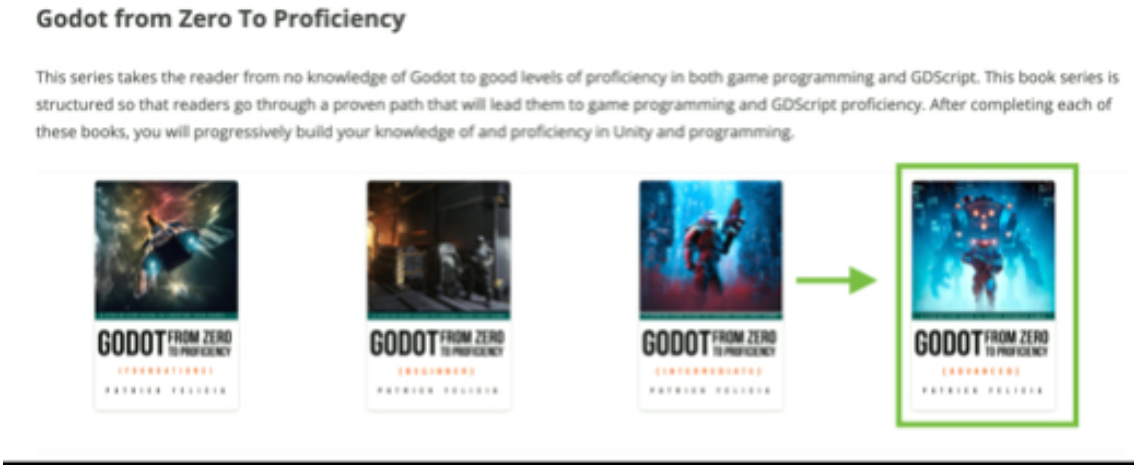
While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback, you can email me at:

learntocreategames@gmail.com.

Downloading the Solutions for the Book

To download the solutions for this book (e.g., code) you need to download the startup pack on the companion website (if you are already a subscriber, you just need to enter the member area - **http://learntocreategames.com/member-area/**); it consists of free resources that you will need to complete your projects and code solutions. To download these resources, please do the following:

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



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

Improving the Book

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

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

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

Supporting the Author

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

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

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

- -

Chapter 1: Introduction to the RPG Genre, Key Features and Design Concepts

In this section, more information will be provided on the RPG genre, in terms of:

- Features.
- Gameplay.
- Format.

You will also be able to see how you will progressively create these features throughout this book.

So, after completing this chapter, you will be able to:

- Understand the RPG genre and its key characteristics.
- Understand the features that you will be creating.
- Understand some interesting design concepts that will be used in this book.

RPG GAMES

Role-Playing games often derive from Dungeon and Dragons (tm) table top games whereby one or several players are required to complete one of several quests to win the game. While each player starts with specific characteristics such as strength, dexterity and special abilities, these features will evolve, and the players' characters will be able to collect loots that can be used to improve their health or their weaponry. Along their way, players will earn e**X**perience **P**oints and use these to improve their skills.

Digital RPGs have used several of these features, including:

- An armed Player Character.
- The ability to collect food and other objects.
- The presence of Non-Player characters with whom they may talk or fight; these NPCs have a wide range of intelligence and skills.
- Sometimes an end-of-level boss is included.
- Randomly-generated scenes.
- Win XPs and level up.

RPG FEATURES IN THE GAME THAT YOU WILL CREATE

Creating an RPG game can be time-consuming as they can include multiple levels and great complexity. This book is attempting to make this process easier for you by helping you to create key components that you can reuse for your games.

Throughout this book, you will learn to do the following:

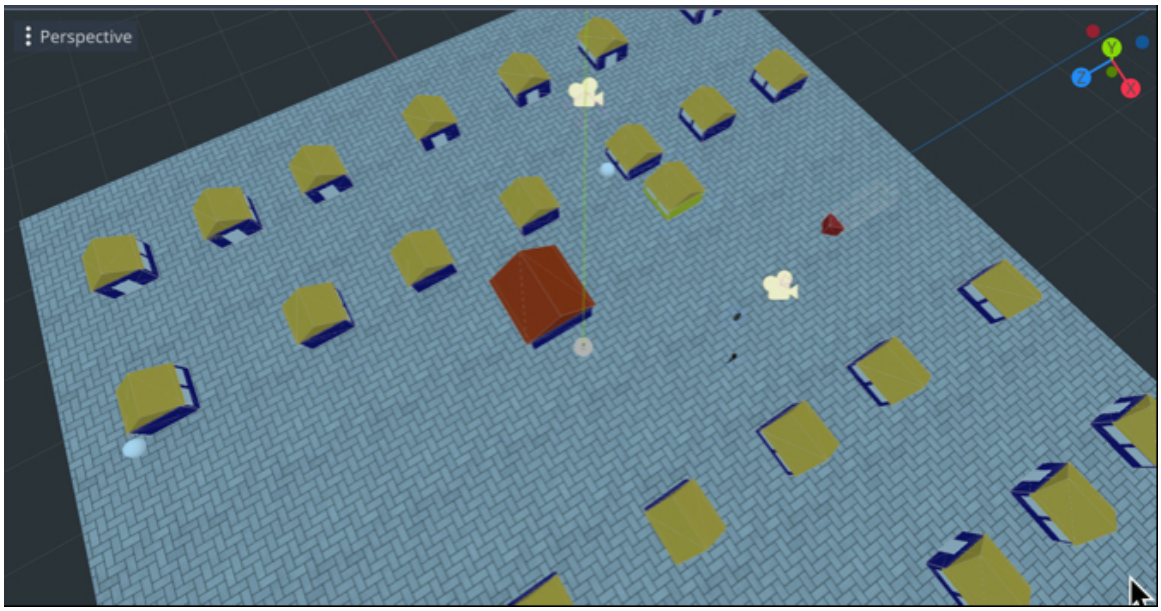
- Create a 3D character that can navigate through a 3D scene.
- Create objects that you can collect.
- Create an inventory to store these objects.
- The ability to use some of these objects (e.g. food), to improve your character's health.
- Provide a weapon to the player character.

Once your character is set up we will then start to work on the environment by doing the following:

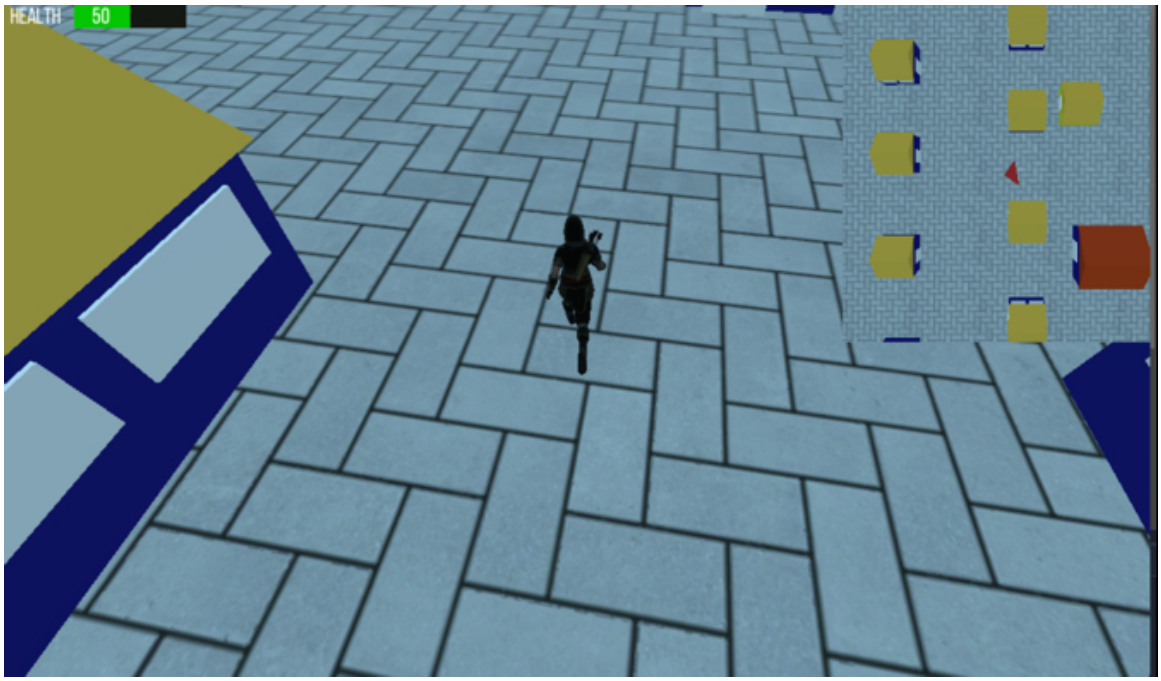
- Create a simple environment using basic shapes.
- Create a mini-map so that it is easier to see where the character is going and highlight objects of interest.
- Add NPCs.
- Add the ability for the player and the NPCs to engage in battles.
- Add NPCs to whom the player can talk.
- Create a shop where the player can buy new items.

Once all these work properly, we will start to create a quest system whereby you can set the objectives for the player using a simple JSON file; this means that you will save yourself some time by just having to modify one file to change the objectives of the quest; we will also create a game manager that will manage the entire game in terms of what should be added or kept between scenes.

- You will create a village as well as all the houses within using **Godot's** built-in 3D editor.

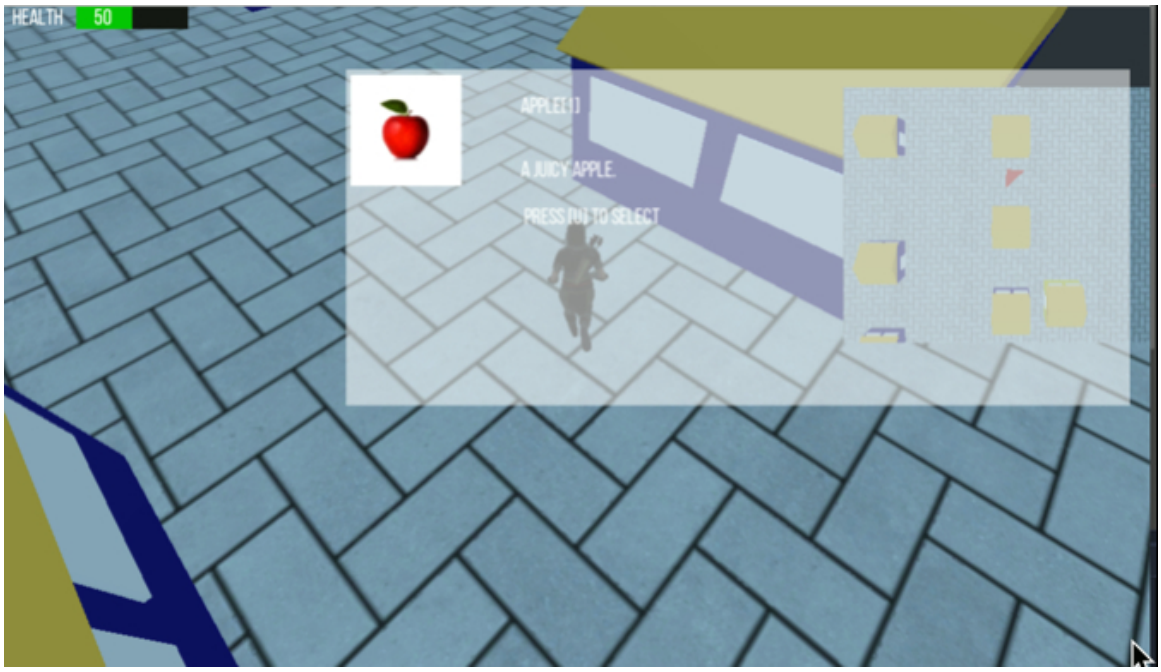


- Your character will be animated and able to use a weapon.

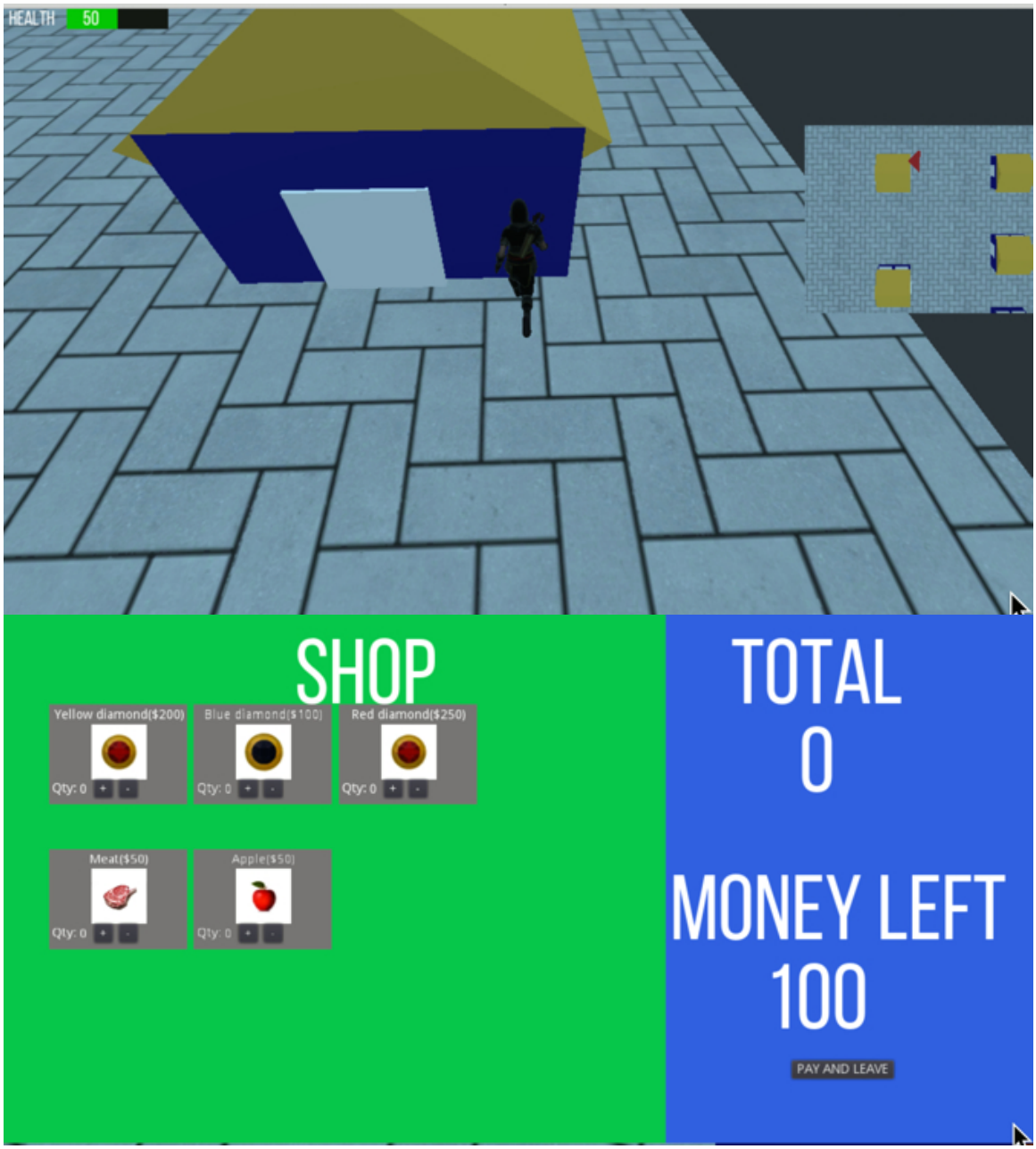


- The player will be able to talk to Non-Player Characters and select questions/ answers so that the outcome of the dialogue depends on their choices.

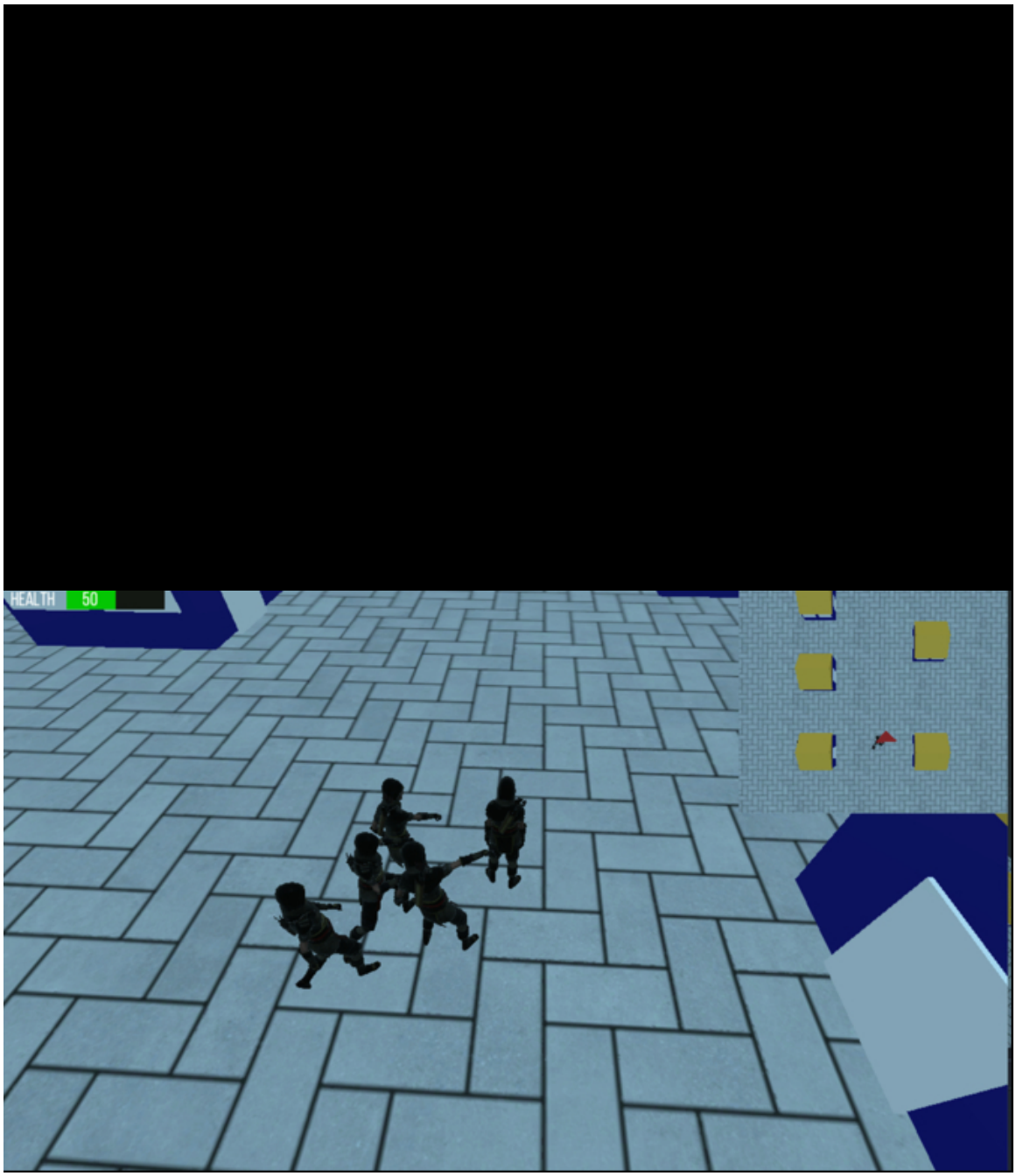
- The player will be able to collect objects and add these to an inventory.



- You will create a shop with both a 3D and a 2D representation from which the player can buy items.

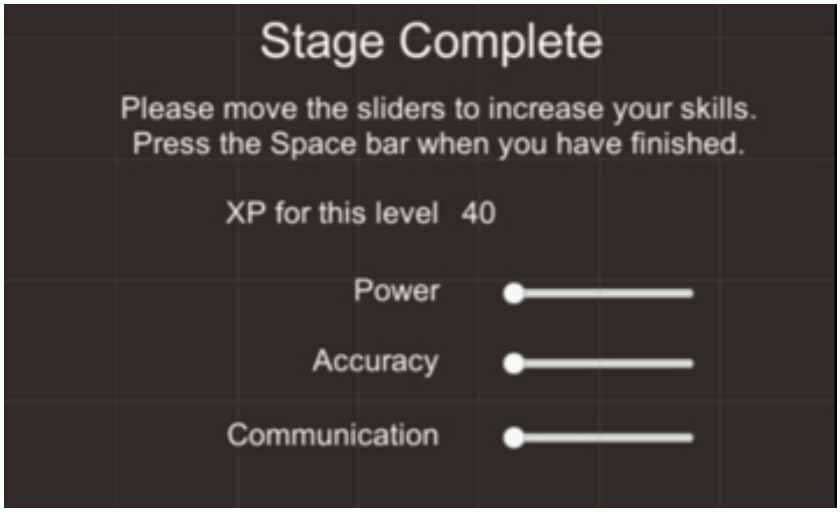


- You will add intelligent NPCs that will be able to detect and attack the player.



- You will create a quest system used to define, display and track the objectives for each level.

- After each level, the player will be able to use their XPs to increase their skills.



THE DESIGN PHILOSOPHY BEHIND THIS BOOK

The idea behind this book is that:

- Creating and modifying your RPG should be made easy.
- All elements in your game should be easily reusable.
- Configuring your quests should be seamless.
- Testing should be made easy.

For this purpose, I have added a few tricks so that: you can create a generic object to be collected and then set its type (e.g., gold, or apple) in the **Inspector** or create new items to be added to the shop or to be collected easily by creating/modifying a few lines of code.

All in all, this book will teach you, not only how to create an RPG, but also how to design your game so that it is easy to expand your code/game without headaches.

Along the way, you will learn about:

- Character Animation.
- Quest Systems.
- Dialogue Systems.
- GDScript.
- Classes.
- JSON files.
- UI design in Godot.
- Artificial Intelligence.
- And much more.

Chapter 2: Creating and Animating the Main Character

In this section, we will start our RPG by creating and animating the main character.

After completing this section, you should be able to:

- Import and animate 3D characters for your RPG.
- Create a third-person view so that your main character is always in full sight of the camera.
- Create a mini-map.

CREATING YOUR CHARACTER

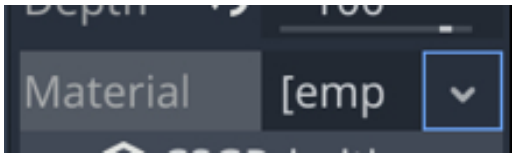
To start with, we will set up and animate our main character. For this book, a character, along with the corresponding animations, has already been created; this being said, if you'd like to use your own character and animations, please feel free to do so.

Let's import the main character and the corresponding animations in Godot.

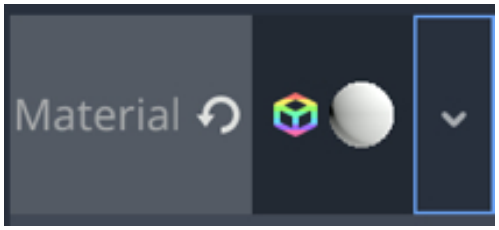
- Please create a new Godot Project (**3D Scene**).
- Open your resource pack (please see the previous sections on how to download the resource pack).
- Open the folder **3D-characters | akai**.
- Drag and drop the files **akai.gltf** to the folder **res://** in the **FileSystem** tab.
- Create the ground from a **Cube**: right-click on the node **Spatial**, select the option **AddChildNode**, and select the type **CSGBox**. This will create a new node called **CSGBox**.
- Rename this node **ground**.
- Using the **Inspector**, change its position (**Transform | Translation**) to (**0, 0, 0**).
- Change its **width**, **height** and **depth** to **100**, **1** and **100** respectively.

Now that the ground has been created, we can apply some texture to it:

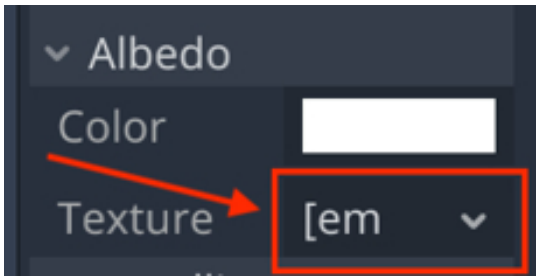
- Drag and drop the texture **pavement.jpg** from the resource pack (i.e., from the folder **textures**) to the folder **res://** in Godot.
- Select the node **ground**.
- Using the **Inspector**, create a new **Spatial Material** by clicking on the downward- fac- ing arrow to the right of the label **Material**.



- Click on the white sphere to the right of the label **Material**.

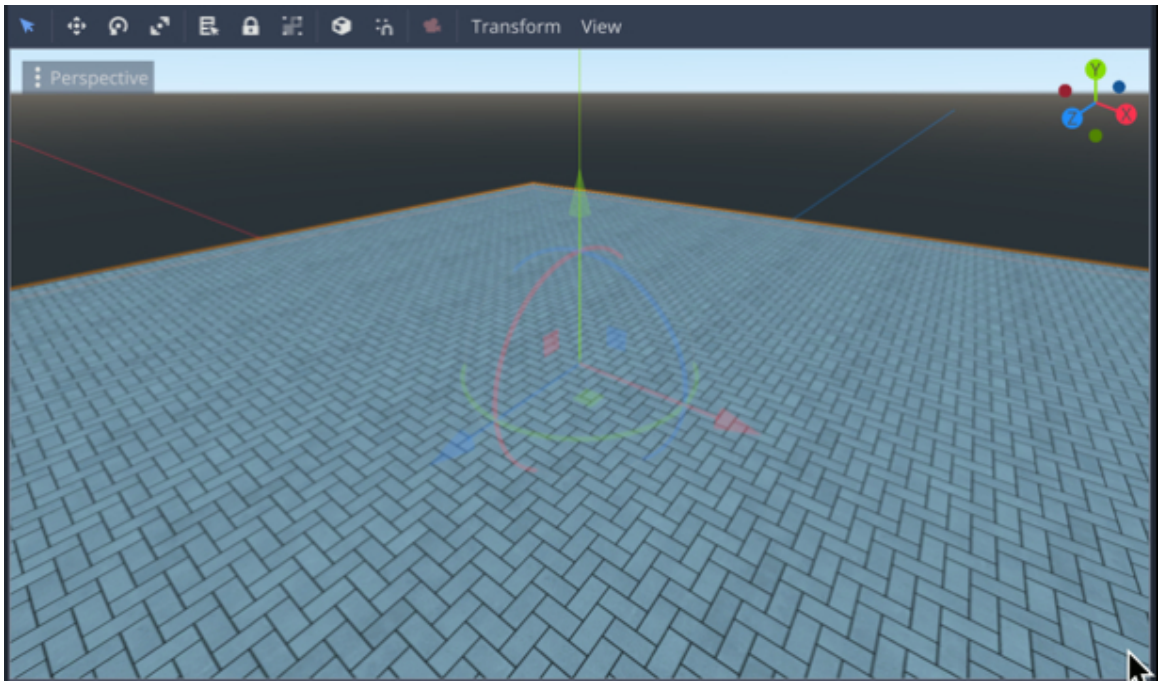


- After expanding the section **Albedo**, drag and drop the texture **pavement.jpg** from the **FileSystem** tab to the attribute called **Texture**.



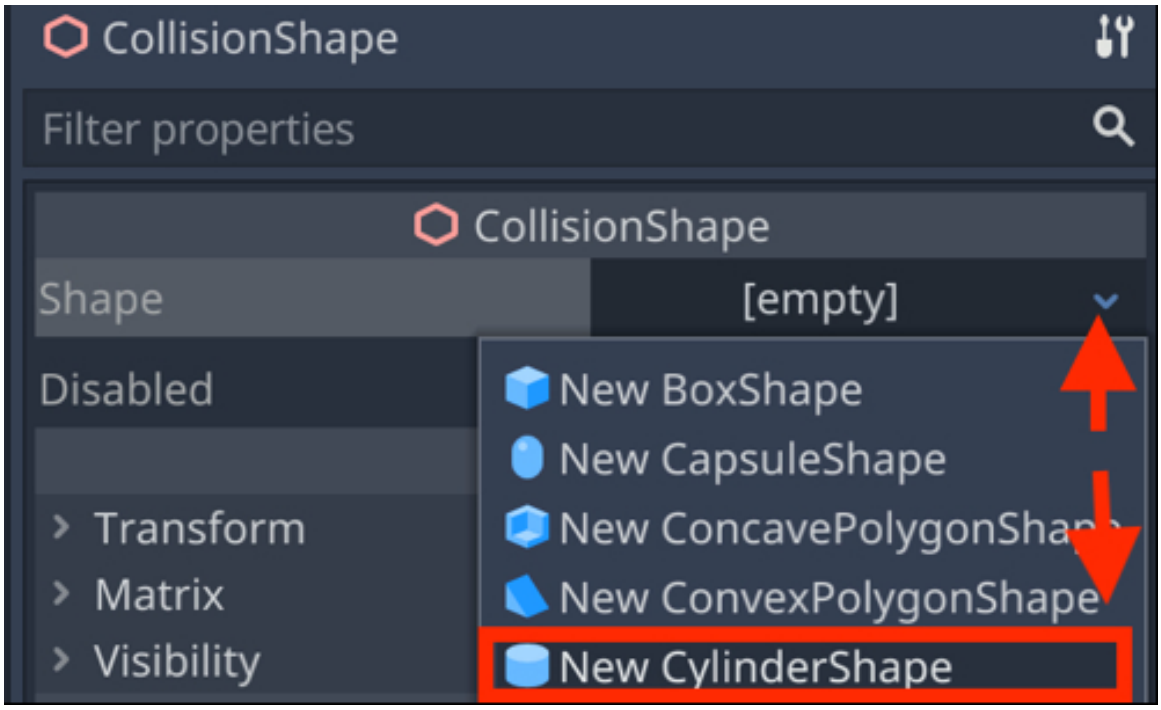
- Once this is done, you can expand the section **UV1**, and change the scale attribute to **(10, 10, 1)**.

- You should now see that the texture has been applied to the ground.

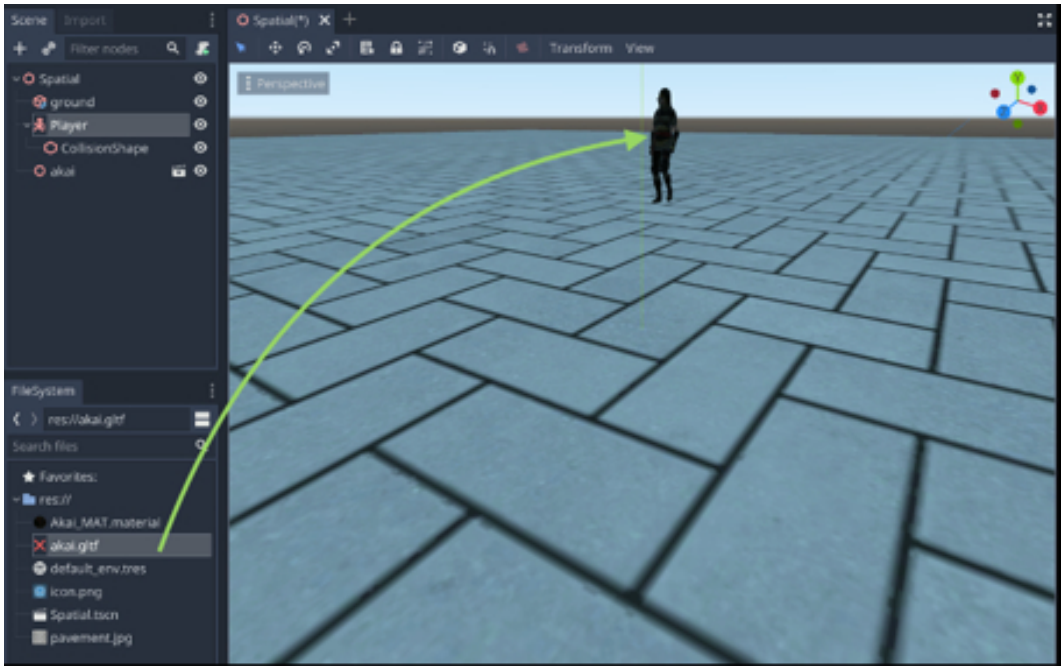


Now that we have created and textured the ground, we will add our character.

- Please create a new **KinematicBody** node as a child of the node **Spatial**, and rename this new node **Player**.
- Change the position of the node **Player** to **(0, 0.5, 0)**.
- Please add a new node of the type **CollisionShape** as a child of the node **Player**.
- Select this new node called **CollisionShape**.
- Using the **Inspector**, click on the downward-facing arrow to the right of the label **Shape** in the section **CollisionShape** and select the option **New Cylinder Shape**.

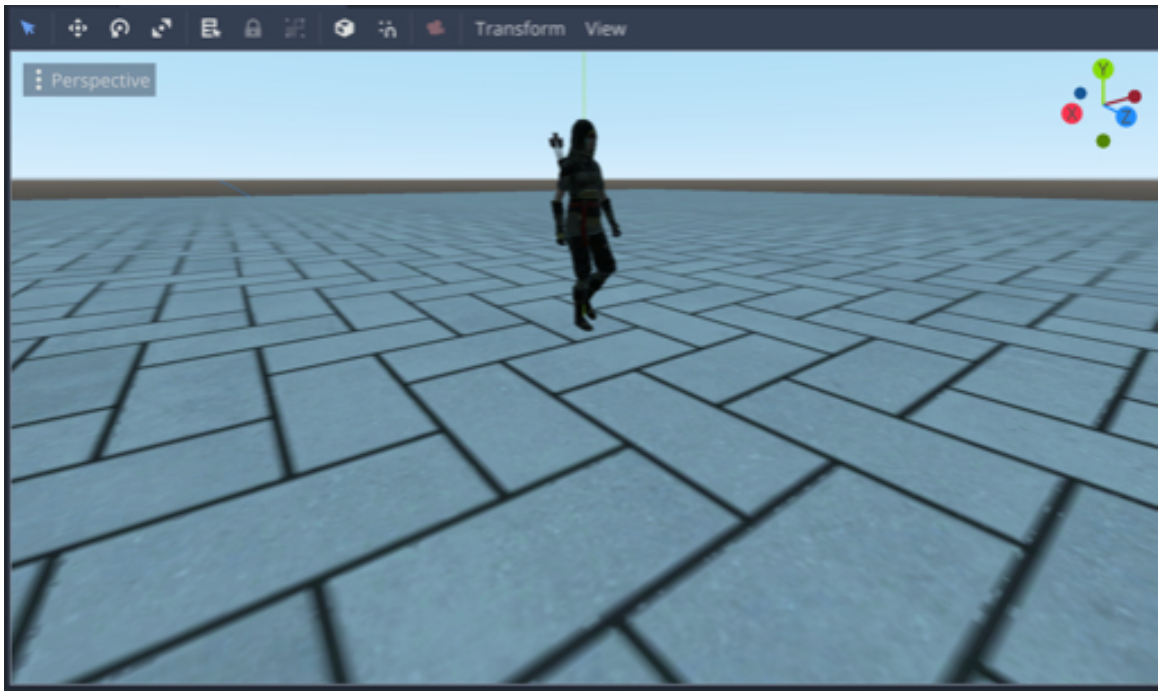


- Change the position of the node **CollisionShape** to **(0, 1.5, 0)**.
- Drag and drop the asset **akai.gltf** from the **FileSystem tab** to the **Scene** view.



- This will create a new node called **akai** in the **Hierarchy Tree** (to the left of the screen).

- Please drag and drop the node **akai** atop the node **Player** so that it becomes a child of the node **Player**.
- Change the position of the node **akai** to **(0, 0, 0)**.
- This will make this node a child of the node **Player**.
- Please rename this node **PC** (for **P**layer **C**haracter).

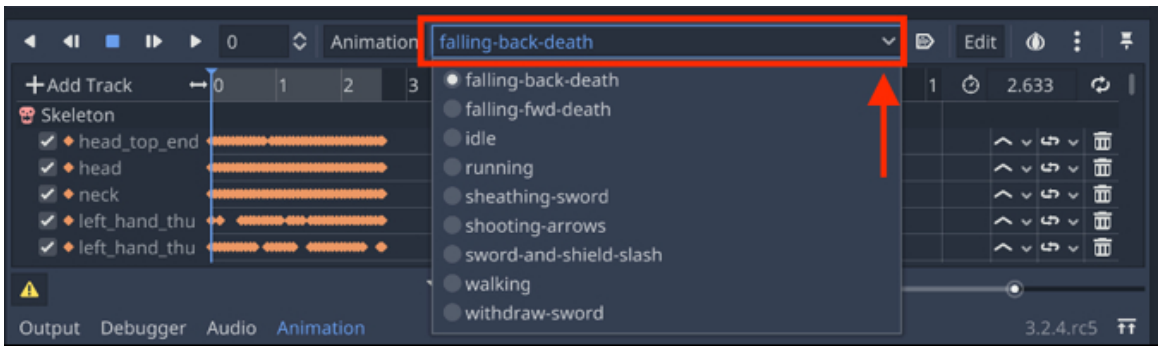


The node **PC** that we have imported contains all the animations that we will need for this game and you can see them by doing the following:

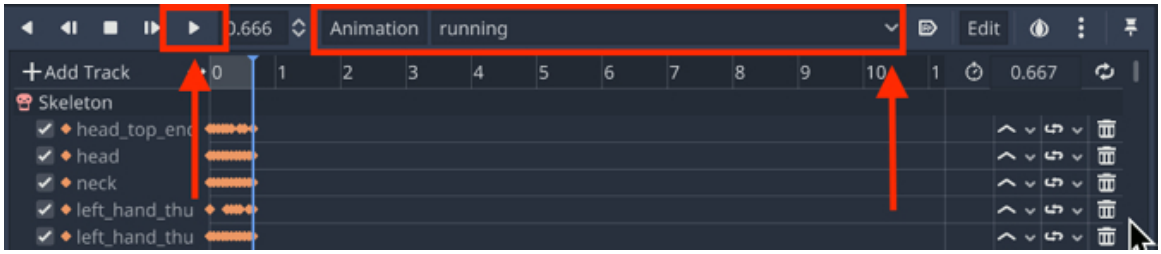
- Right-click on the node **PC** and select the option **Editable Children**.
- This should show the children nodes of the node **PC**.

- Select the node **AnimationPlayer**, and you should see that the **Animation** window is now open.

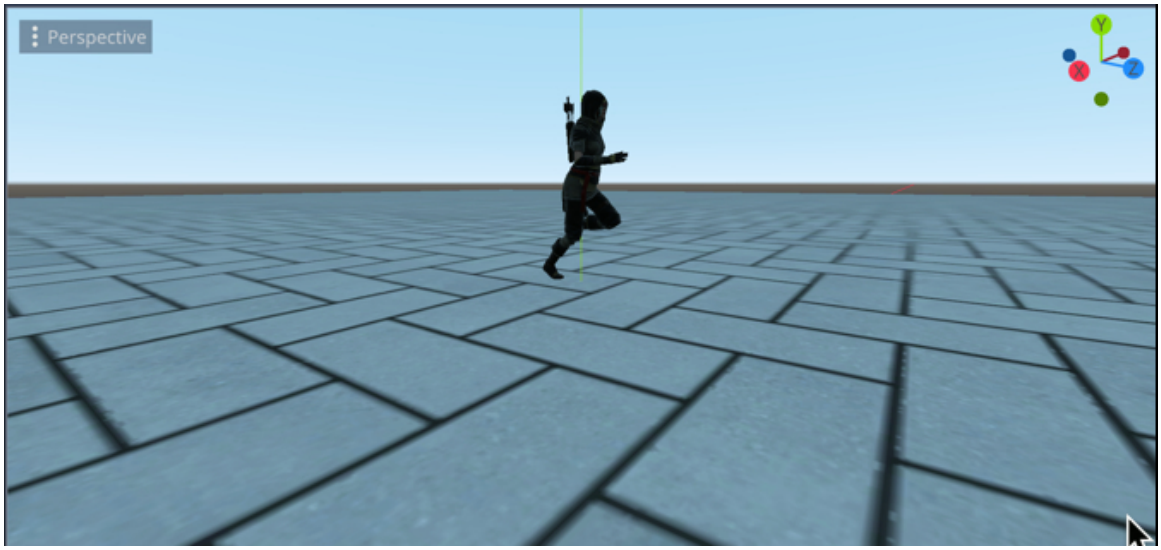
- If you click on the arrow to the left of the label "**Animation**", as described in the next figure, you should see a list of the animations contained in the node.



- If you want to see these animations in action, make sure that your view is focused on the character **akai**, select an animation in the **Animation** window (for example, **running**), and press the **Play** button, as described in the next image.



- After pressing the **Play** button, you should see the character running.

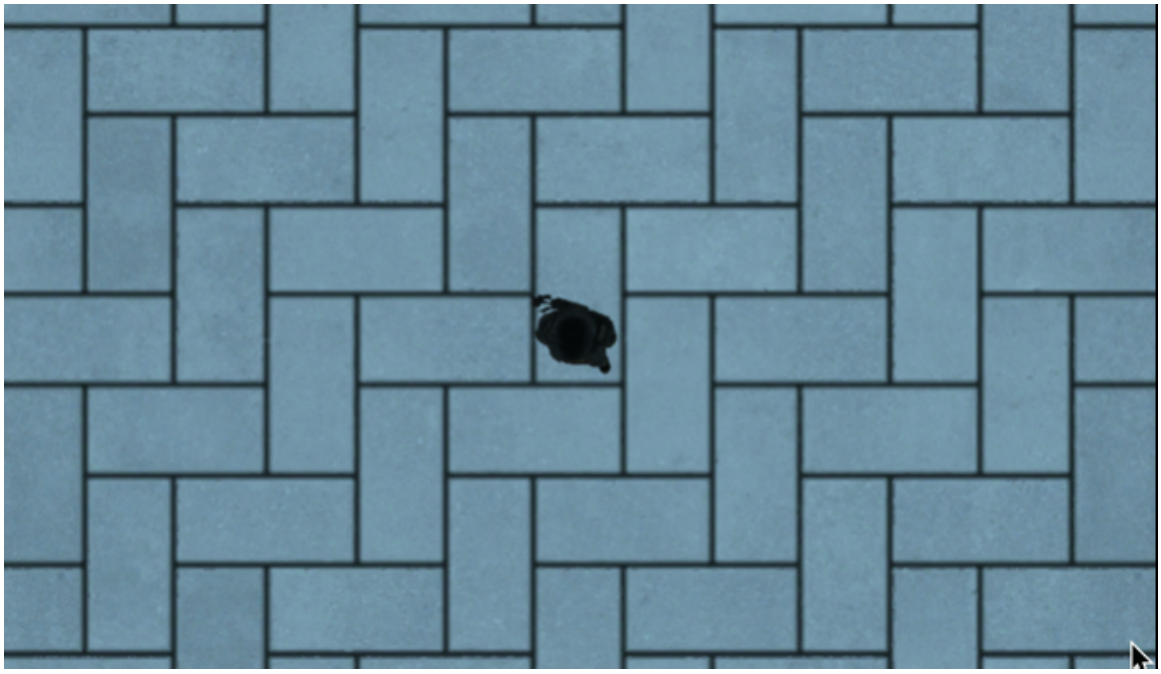


So at this stage, we know that the animations are working properly; so the next phase will consist in controlling our character through a script.

For now, we will add a camera to the scene so that we can see our character (that is not yet animated) when we play the scene:

- Please add a node of the type **Camera** as a child of the node **Spatial**.
- This will create a new node called **Camera**.
- Using the **Inspector**, change the **Rotation** property of this node to **(-90, 0, 0)** and its position to **(0, 5, 0)**.
- You can now play the scene, and you should see the Player Character

appearing just below the camera.



- You can stop playing the scene for now.

CONTROLLING THE CHARACTER FROM A SCRIPT

In this section we will animate our character from a script.

- Please select the node **Player** and add a new script to it (i.e., right-click on the node **Player** and select the option to "**Attach a Script**"). Name this script **manage_player**.
- Open this script.
- Add the following code just before the function **_ready**:

```
enum {IDLE, WALK, WALK_REVERSE, RUN}
var rotation_increment = .2
var current_state = IDLE
var speed = 1
onready var pc_node = get_node("PC/AnimationPlayer")
```

In the previous ode:

- We create several **enums** that will be used to identify the current state of the player character.
- We declare a variable called **rotation_increment** that will be used to determine the rotation speed of the player character.
- We declare a variable called **current_state** that will store the character's current state.
- We define the speed for this character through the variable **speed**.
- We also create a variable called **pc_node** that will be linked to the node **AmimationPlayer** which contains the animations for the character.

Now that we have defined the key nodes and variables for our player character, we just need to initialize the animations that we will be using by specifying whether they should be looping.

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

```
func _ready():
pc_node.get_animation("idle").loop = true
pc_node.get_animation("walking").loop = true
pc_node.get_animation("running").loop = true
```

In the previous code, we specify that all the animations that we will be using and that are contained in the node **AnimationPlayer** (i.e., **idle**, **walking**, and **running**) will be looping over time.

Next, we will specify what should happen in all the states that we have defines earlier (i.e., what animation should be played and what speed should be used depending on whether the character is walking or running).

- Please add the following function:

```
func _process(delta):
match current_state:
IDLE:
WALK:
WALK_REVERSE:
RUN:
```

In the previous code, we define four different states for our character, and the system will switch to one of them depending on the value of the variable **current_state**.

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

```
IDLE:
pc_node.play("idle")
WALK:
speed = 1
pc_node.play("walking")
var forward = global_transform.basis.z;
move_and_slide(forward*speed, Vector3.UP)
```

In the previous code:

- We specify that in the state called **IDLE** we will play the animation called **idle**.
- We also specify that in the state called **WALK**, the speed of the character will be **1**, and that the animation called **walking** will be played.
- We also move the character forward using the function **move_and_slide**; the forward direction used is relative to the character's **z**-axis.

Please add the following code (new code in bold):

```
WALK_REVERSE:
speed = -1
pc_node.play("walking")
var forward = global_transform.basis.z;
move_and_slide(forward*speed, Vector3.UP)
```

In the previous code, we specify what should happen when the player walks backward; the code is similar to the one used for the state **WALK** except that the speed is now **-1**.

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

```
RUN:
speed = 4.0
pc_node.play("running")
var forward = global_transform.basis.z;
move_and_slide(forward*speed, Vector3.UP)
```

In the previous code, we specify what should happen when the player character is running; the code is similar to the one created for the state **WALK**, except that the speed is now **4**.

Finally, we just need to write the code that will detect the keys pressed so that the player can walk, run, and turn. But before we create this code, we just need to map the keys that we will use through the **Input Map**:

- Please select: **Project | Project Settings** from the top menu, and then open the

tab **Input Map** in the new window.

To map a key, you just to type (1) the action in the top-most the field, (2) click on **Add**, and then select this action from the list, (3) click on the **+** sign to the right of this action, then (4) click on the button labeled "**key**", and then type the key (from your keyboard) that you want to associate with this action, and then press **OK**, as per the next figure.

Please do the following:

- Map the **up arrow** key to the action "**move_fwd**".
- Map the **down arrow** key to the action "**move_back**".
- Map the **left arrow** key to the action "**turn_left**".
- Map the **right arrow** key to the action "**turn_right**".

Once this is done, we can return to our script

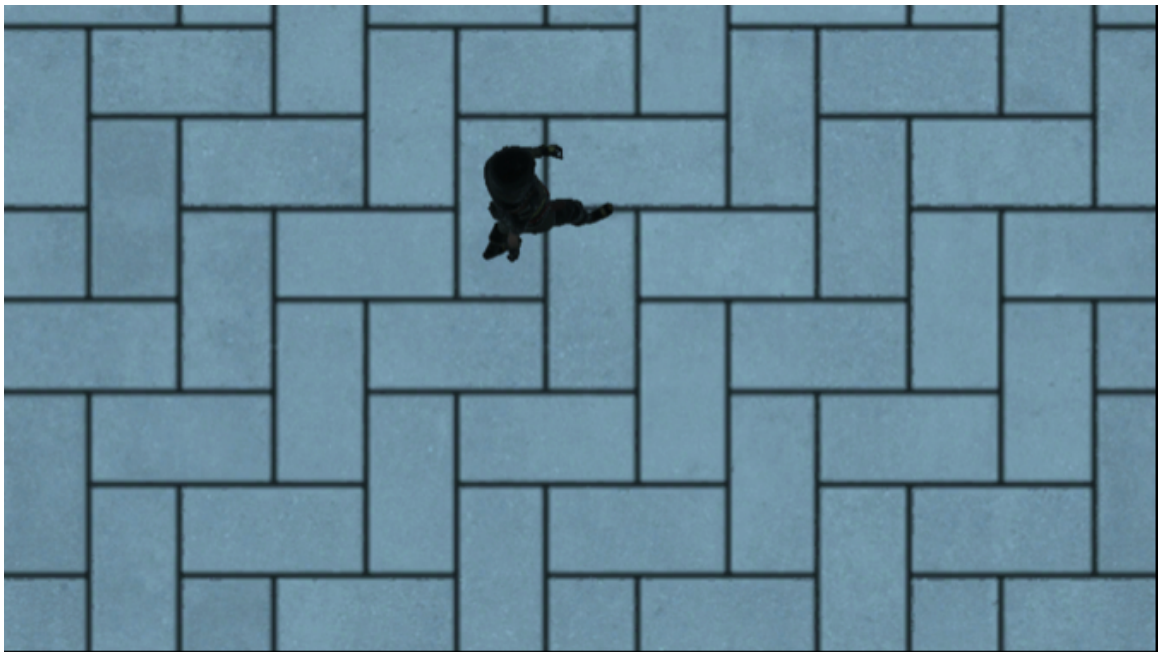
- Please add the following function to the script (i.e., **manage_player**):

```
func _input(event):
if (Input.is_action_pressed("move_fwd")) :current_state = WALK
elif (Input.is_action_pressed("move_back")) :current_state = WALK_REVERSE
else: current_state = IDLE
if (Input.is_action_pressed("turn_left")):
rotate_y(rotation_increment)
if (Input.is_action_pressed("turn_right")):
rotate_y(-rotation_increment)
if (Input.is_key_pressed(KEY_ALT)):
current_state = RUN
```

In the previous code:

- We use the function **_input** that is often employed to detect the user's input.
- If the "**move_forward**" action is detected then the current state is switched to **WALK**.
- If the "**move_back**" action is detected then the current state is switched to **WALK_REVERSE**.
- Otherwise the current state is **IDLE**.
- If the "**turn_left**" action is detected then the character is rotated clockwise.
- If the "**turn_right**" action is detected then the character is rotated anti-clockwise.
- Finally, if the **ALT** key is pressed, the character is switched to the state **RUN**.

You can now save your code and play the scene; you should see that the character can walk, run, and turn as you use the arrow and the **ALT** keys.

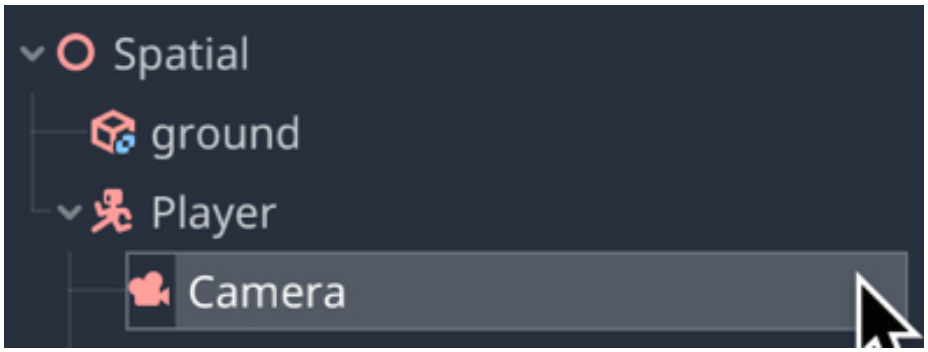


So that the camera is always following the player we will, in the next section create some code that will ensure that the camera is always following and looking towards the player character.

Modifying the Camera to follow the player

In this section, we will make sure that the camera is always following the player. Please do the following:

- Drag and drop the **Camera** node atop the node **Player**, so that is it becomes a child of the node **Player**.



- Select the **Camera** node.
- Using the **Inspector**, change the position of the camera to **(0, 5, -4)** and make sure that its rotation attributes are **(0, 0, 0)** and its scale attributes are **(1, 1, 1)**.
- This is so that the camera is placed above the player (i.e., **y = 5**) and slightly behind it (i.e., **z = -4**).

Note that at this stage the camera is still not focused on the player; so we just need to make sure that it will always be looking at the player; for this purpose, we will create a simple script that makes sure that the camera looks at the player, using a built-in function called l**ook_at**:

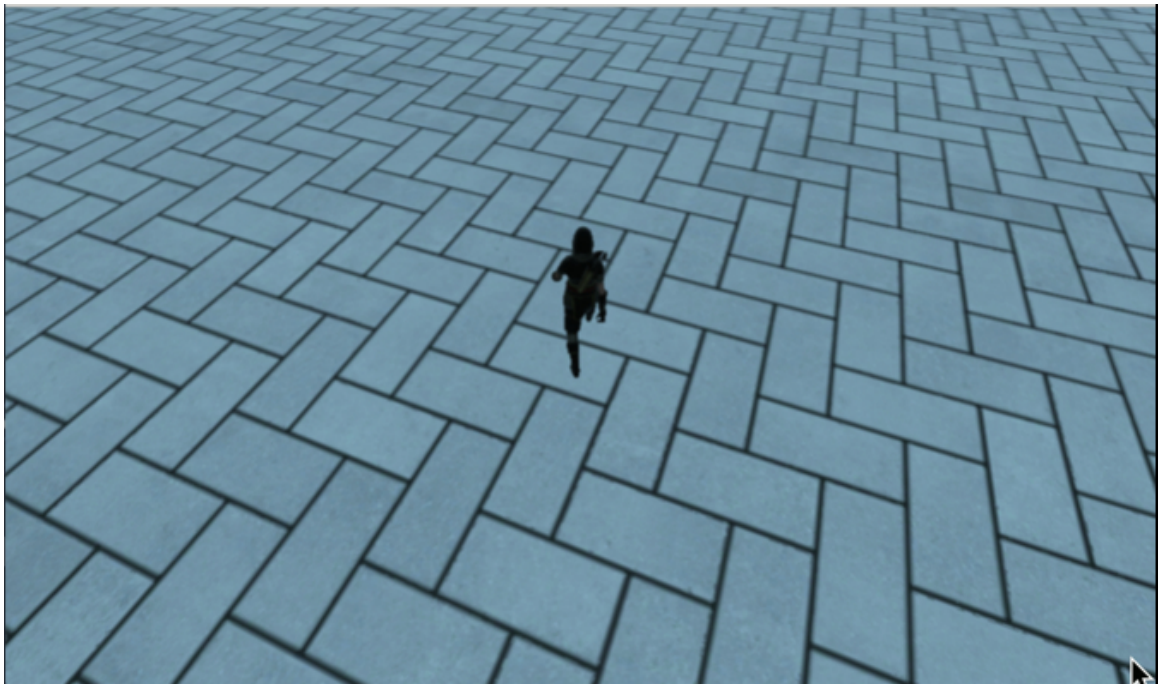
- Please right-click on the **Camera** node.
- Select the option "**Attach Script**" from the contextual menu to create a new script.
- Call the new script **camera_follow_player**.
- Open the script and add the following code to it:

```
func _process(delta):
look_at(get_parent().global_transform.origin,Vector3.UP)
```

In the previous code:

- We use a built-in function called **_process**, which is called every frame.
- In this function, we use the function **look_at** which makes sure that the node linked to this script (i.e., the camera) looks at a specific target.
- In our code, the target is the parent of the current node: the **Player** node.
- So effectively, every frame, we make sure that the camera is pointing towards the player.

You can now save your script and play the scene, and you should see that the camera is effectively following the player.

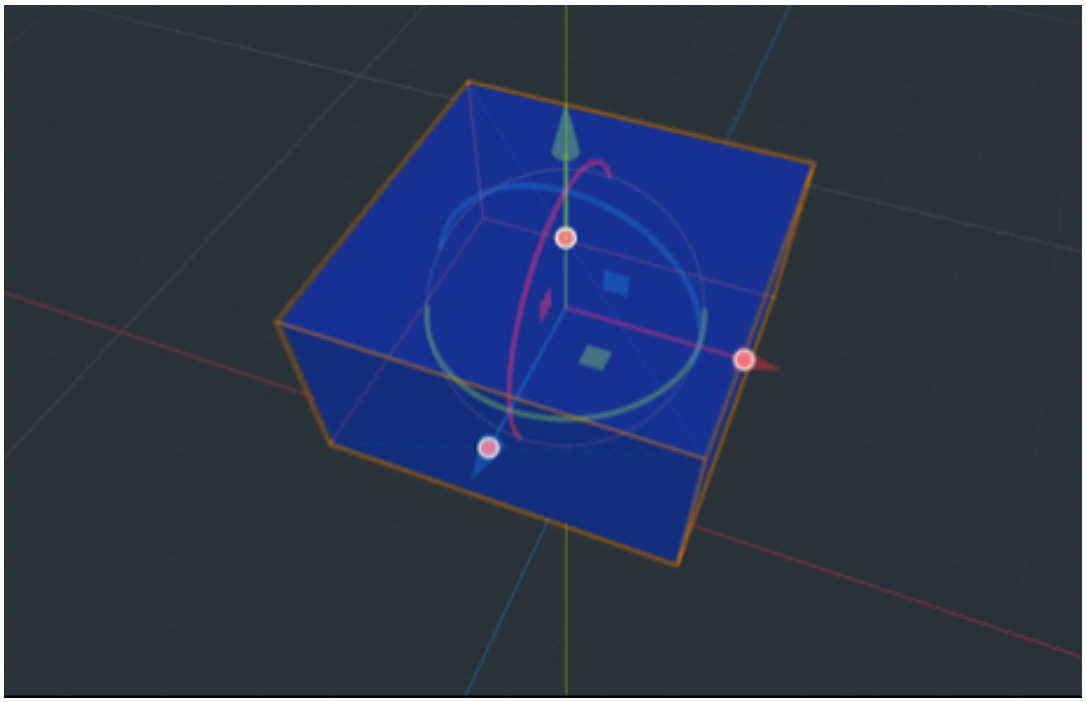


Creating the village

In this section, we will start to create the village that the player will navigate to complete its quests. The village will consist of several houses, and each of them will be based on a template (i.e., a scene that includes a house built with primitive shapes).

So first, we will create the template house:

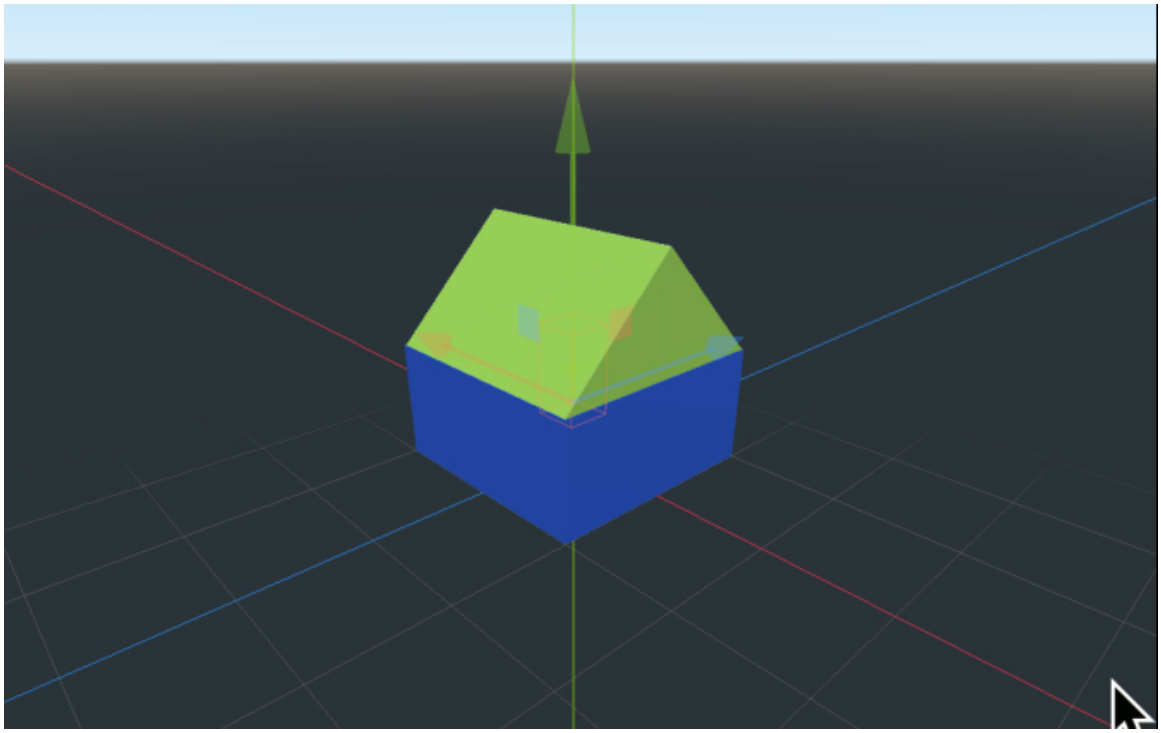
- Please create a new scene: **Scene | New Scene | Other | Static Body**.
- This will create a new scene with a default node of the type **StaticBody**.
- Rename this node **house**.
- Change the position of this node to **(0, .5, 0)**.
- Create a new **CSGBox** node as a child of the node **house**.
- Rename this node **wall**, and check that its **width**, **height** and **depth** are all **2**.
- Apply a new blue **Spatial Material** to this node.



We can now create the roof for this house:

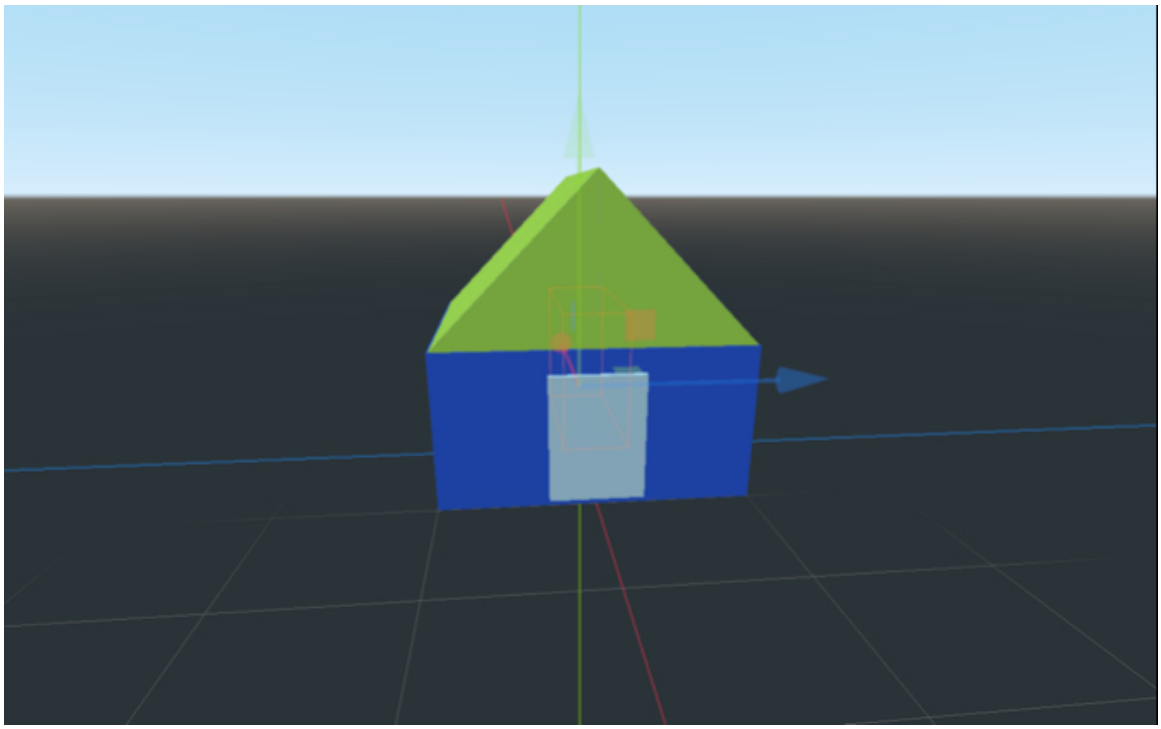
- Please create a new **CSGSBox** node as a child of the node **house** and rename it **roof**.

- Check that the **width**, **depth**, and **height** of this node are all **2**.
- Change its **position** to **(0, .5, 0)**, its **rotation** to **(45, 0, 0)** and its **scale** to **(0.99, 0.7, 0.7)**.
- Apply a new yellow **Spatial Material** to this node.



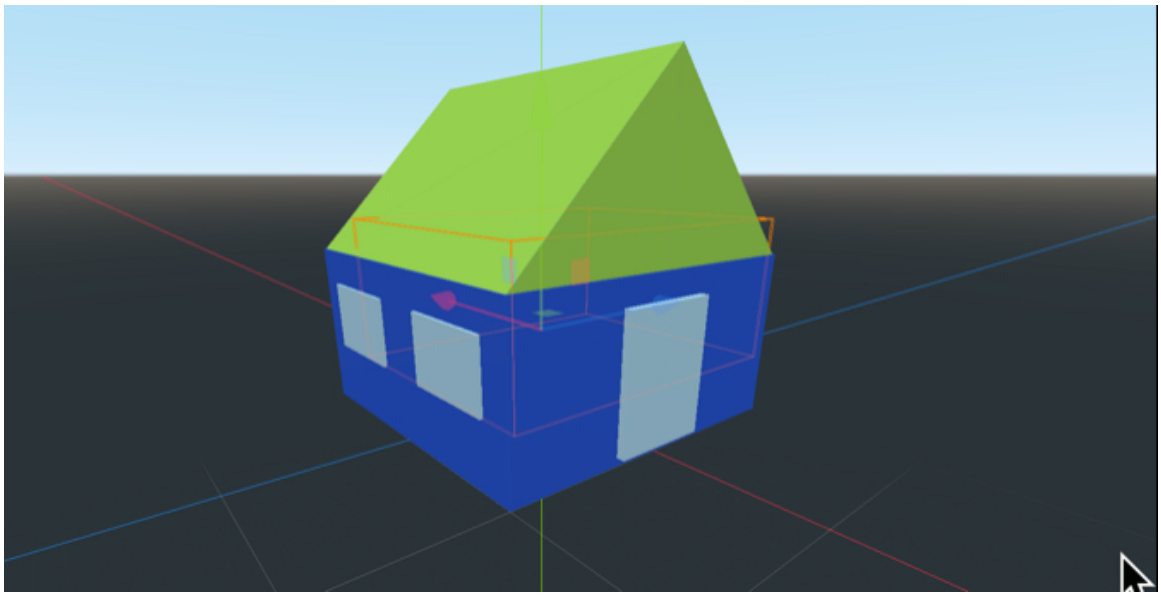
Once this is done, we can start designing the door and windows:

- Please create a new **CSGSBox** node as a child of the node **house** and rename it **door**.
- Check that the **width**, **depth**, and **height** of this node are all **2**.
- Change its **position** to **(-0.95, -0.05, 0)** and its **scale** to **(0.1, 0.4, 0.3)**.



Now that the door has been created, we can complete the house by creating the windows:

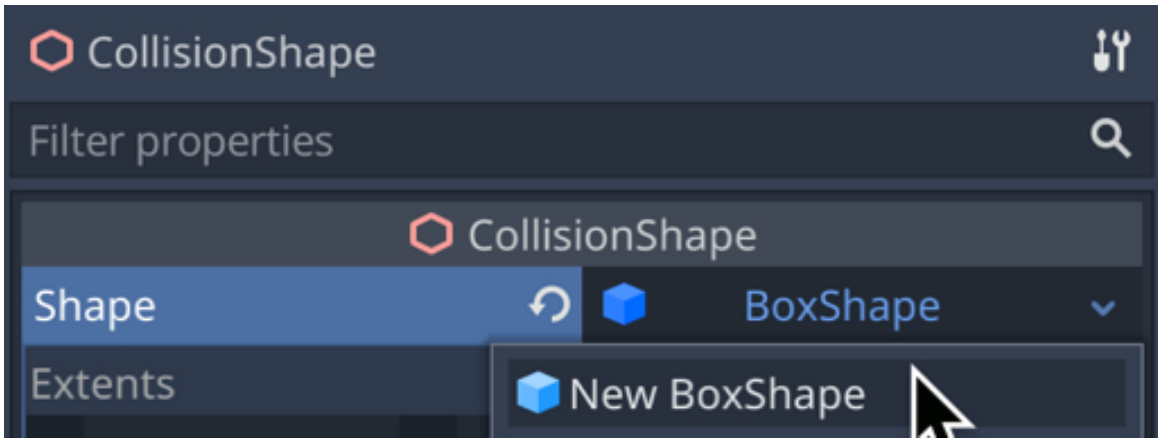
- Please create a new **CSGBox** node as a child of the node **house**, and rename this node **window1**.
- Check that the **width**, **depth**, and **height** of this node are all **2**.
- Change its **position** to **(0.5, 0.1, -0.92)** and its **scale** to **(0.3, 0.2, 0.1)**.
- You can leave this node with the default white color or apply a color of your choice.
- Duplicate this node and rename the duplicate **window2**.
- Change its position to **(-0.5, 0.1, -0.92)**.



- Please duplicate the node **window2** and rename the duplicate **window3**.
- Change its position to **(-0.5, 0.1, 0.92)**.
- Finally, duplicate the node **window3** and rename it **window4**.
- Change its position (i.e., **window4**) to **(0.5, 0.1, 0.92)**.

So, at this stage, you should have a house with walls, a roof, windows, and a front door. The last thing we need to add to this house is a **Collision Shape** node, so that collisions can be detected with other objects (including the **Player Character**).

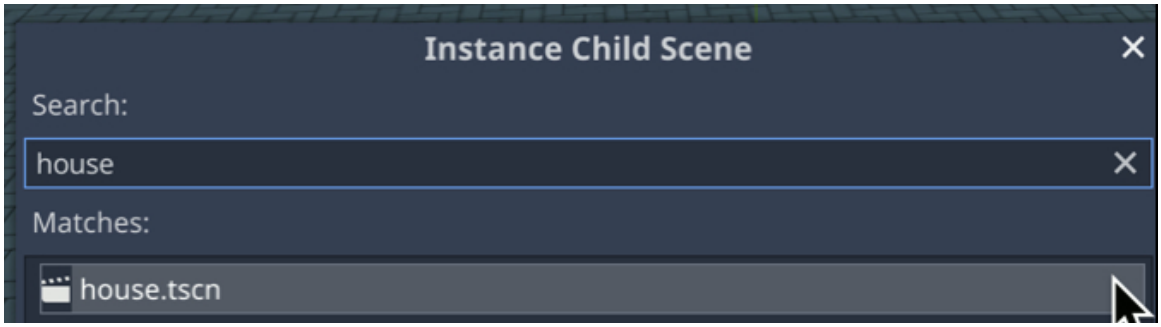
- Please create a new node of the type **Collision Shape** as a child of the node **house**.
- Select this node, and using the **Inspector**, click on the downward-facing arrow to the right of the label "**Shape**", in the section **Collision Section**, and select the option **New Box Shape**.



Now that a collider has been created for the house, we can start to create the village.

So now, we will use this template to create the village:

- Please save the current scene as **house.tscn.**
- Switch back to the previous scene.
- Right-click on the **Spatial** node, and select the option "**Instance Child Scene**".
- In the new window, please type the text **house** in the search field, select the scene **house.tscn** from the result, and press **Open**.



- This will create a new **house** node as a child of the node **Spatial**.
- Change its **scale** attribute to **(3, 3, 3)** and also change its **position** so that it is slightly above the ground and a few meters away from the player character.

- Duplicate this node seven times, as to obtain eight houses in total in a row, as illustrated in the next screenshot.

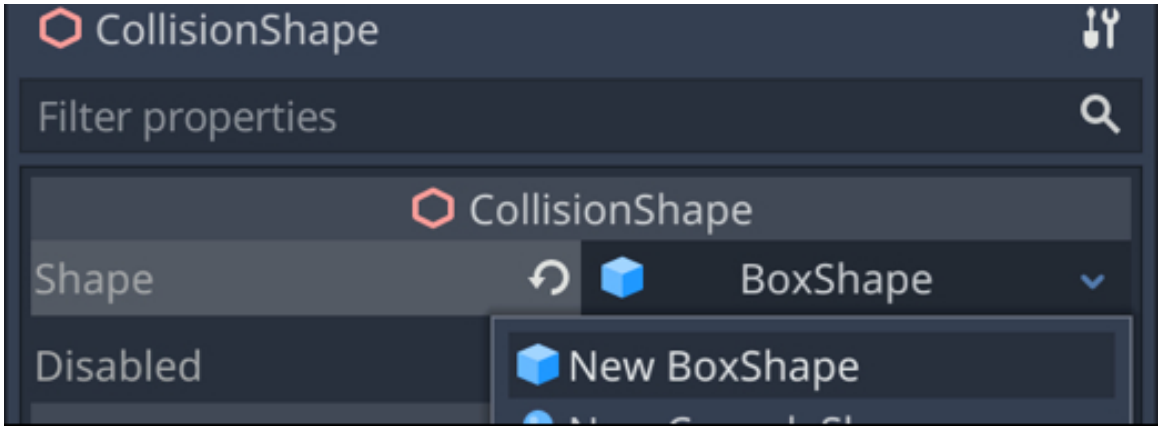
- Duplicate these 8 houses, rotate them about 180 degrees, and place them opposite from the first row, as illustrated in the next figure.

- Duplicate these 16 houses to create the layout illustrated in the next figure.

Next, we will ensure that the player character can't fall off the ground by creating

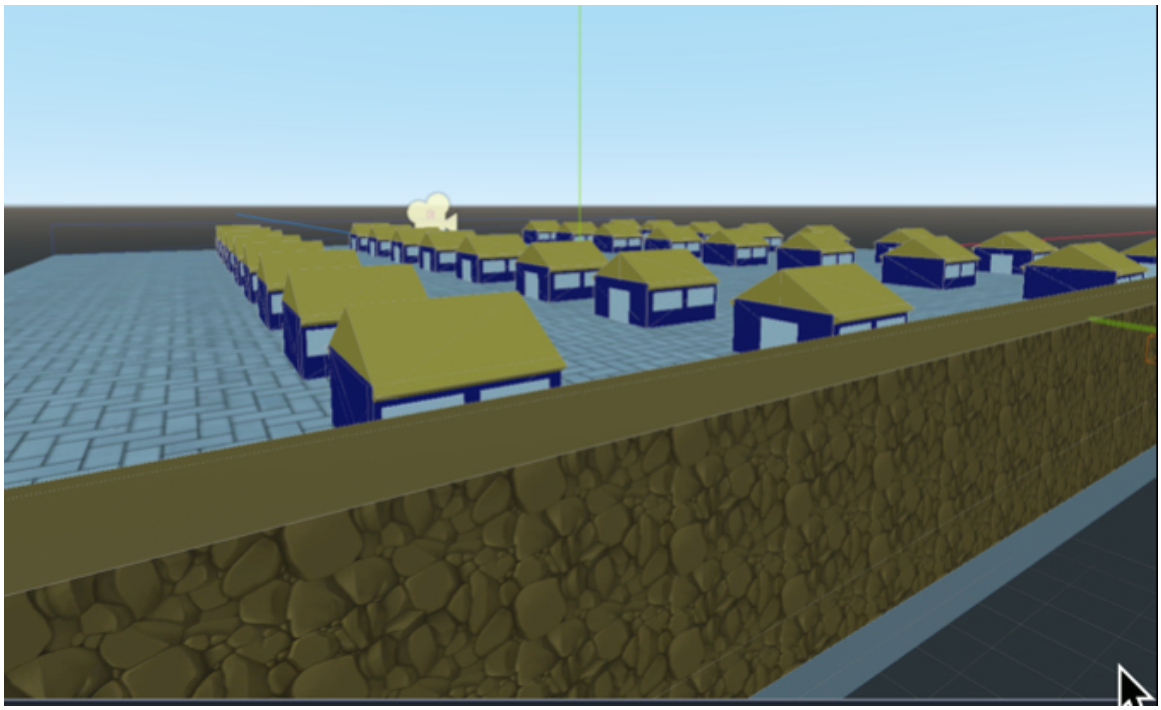
walls around the village. So please do the following:

- Create a new scene of the type **StaticBody**.
- This will create a new scene with a default node of the type **StaticBody**.
- Rename this node **wall**.
- Change its scale attribute to **(50, 2, 1)**.
- Create a new **CSGBox** node as a child of the node **wall**.
- Apply a new **Spatial Texture** to this node, and use the image **wall_picture.jpg** (from the resource pack) for this texture, setting the **UV1 | Scale** attribute to **(50, 2, 1)**.
- Add a new node of the type **CollisionShape** to the node **wall**.
- Using the **Inspector**, click on the downward-facing arrow to the right of the attribute **Collision Shape | Shape** and select the option **New Box Shape**.



- Save this scene as **wall.tscn**.
- Switch back to the previous scene (i.e., the main scene).
- Right-click on the node **Spatial**, and select the option **Instance Child Scene.**
- Select the scene **wall.tscn** in the new window and press "**Open**".

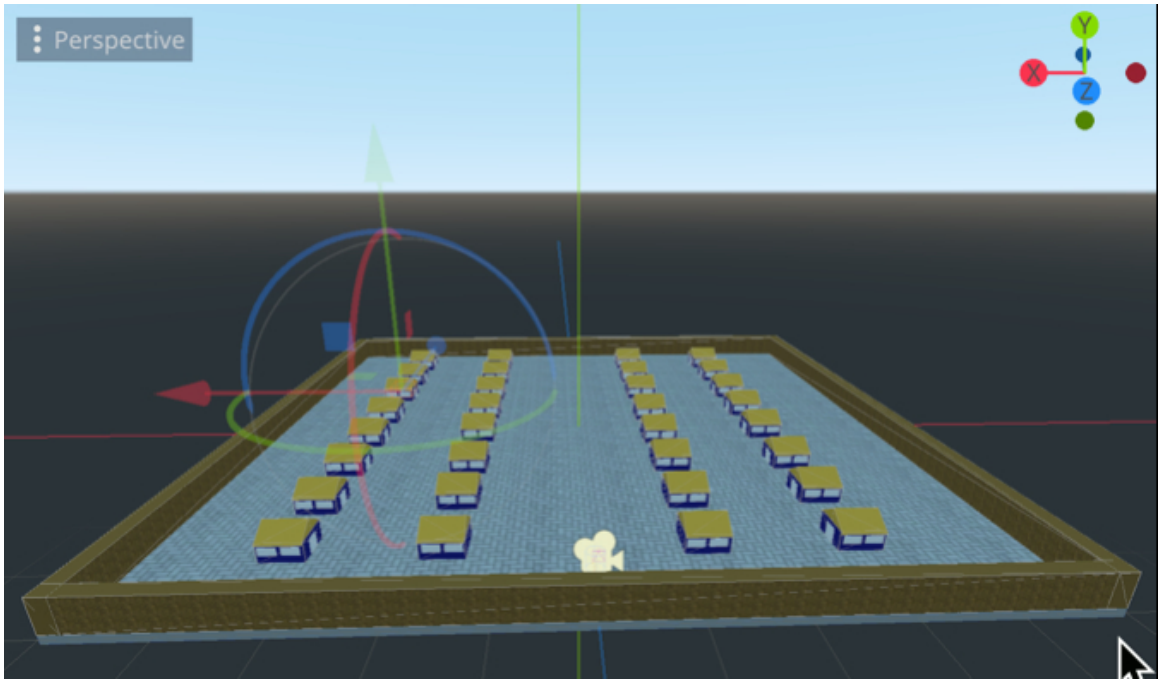
- This will add a new node called **wall** to the scene.
- Rename this node **north_wall**, and change its position to **(0, 2.5, 49)**.



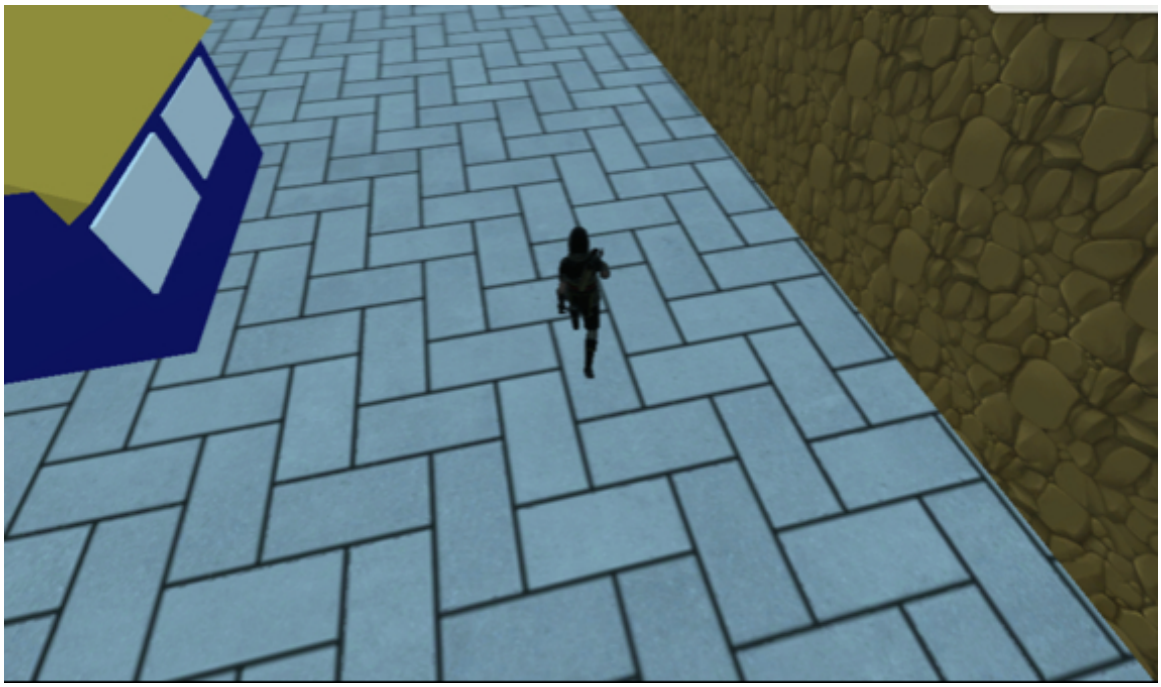
- Duplicate this node (i.e., **north_wall**), rename the duplicate **south_wall**, and

change its position to **(0, 2.5, -49)**.

- Duplicate this node (i.e., **south_wall**), rename the duplicate **east_wall**, and change its position to **(49, 2.5, 0)** and its **rotation** to **(0, 90, 0)**.
- Finally, duplicate this node (i.e., **east_wall**), rename the duplicate **west_wall**, and change its position to **(-49, 2.5, 0)**.



You can now test the scene and ensure that the village's walls are textured properly, and also that the player can collide with them.



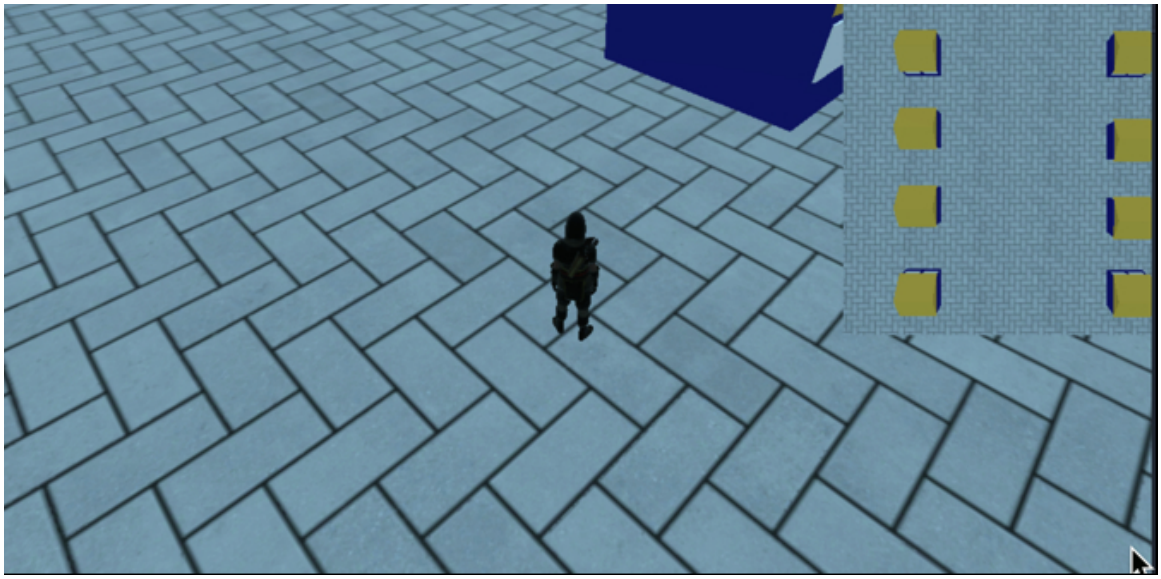
You can save your main scene as **level1** (**Scene | Save Scene As**).

Creating a mini-map

At this stage, our character can move around freely in the village; this being said, it would be great to avail of a map and we will create one in this section.

Please do the following:

- Open the main scene (i.e., **level1**).
- Create a node of the type **ViewportContainer** as a child of the node **Spatial**.
- Select the node that you have just created, and, using the **Inspector**, expand the section **Control | Rect**, and change the attributes **Position** to **(750, 0)**, **Size** to **(300, 300)**, and **Min Size** to **(300, 300)**.
- Add a new node of the type **Viewport** as a child of the node **Viewport-Container** and set its **Size** attribute to **(300, 300)**.
- Add a new node of the type **Camera** as a child of the node **Viewport** and re-name it **mini-map-camera**.
- Change the **position** of this camera to **(0, 35, 0)** and its **rotation** to **(-90, 0, 0)**.
- You can now play the scene and you should see the mini-map in the top-right corner of the screen.



However, you may notice that the map doesn't move with the player; in addition, it is quite challenging to identify the location and the direction of the player; so in this section, we will just ensure that the map is moving with the player and

that an arrow is displayed on screen to represent the direction of the player.

- Please attach a new script called **mini-map-camera.gd** to the node **mini-map-camera**, and add the following code to it:

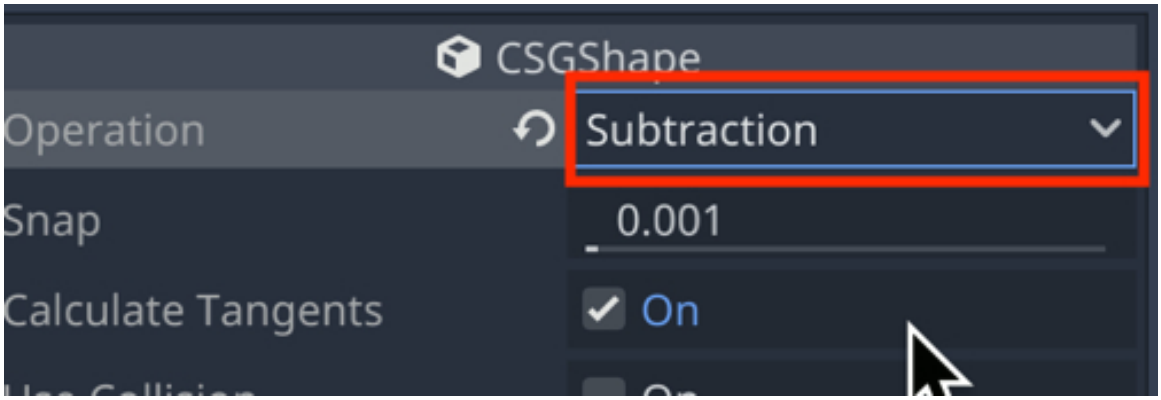
```
onready var player = get_node("../../../Player")
func _process(delta):
transform.origin.x = player.transform.origin.x
transform.origin.z = player.transform.origin.z
```

In the previous code:

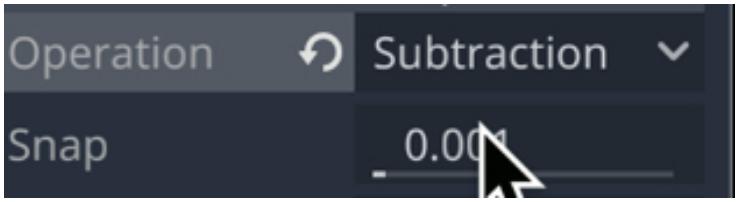
- We create the variable **player** that links to the **Player** node
- We then use the function **_process** that is called every frame.
- In this function, we ensure that the **x** and **z** coordinates of the camera are the same as the ones for the **Player Character**.

Next, we will create an arrow that will be displayed on the mini-map and that indicates the direction of the player.

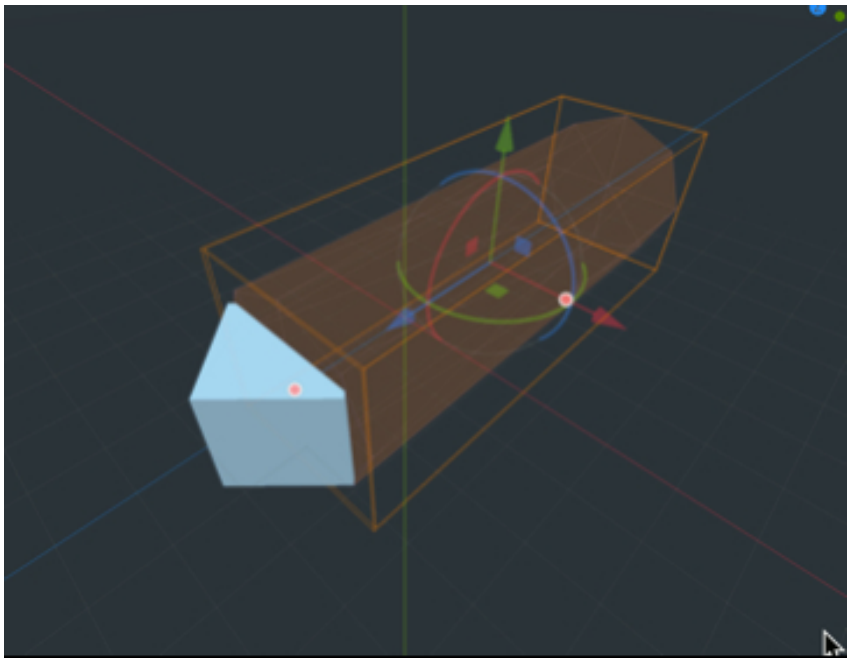
- Please create a new **3D** scene and save it as **top_arrow.tscn**.
- In the new scene, there should be a default node called **Spatial**: please re-name this node **top_arrow**.
- Add a node of the type **CSGCombiner** as a child of the node **top_arrow**.
- Select this node, and using the **Inspector**, select the option **Subtraction** from the drop-down list to the right of the attribute **Operation**.



- Add a node of the type **CSGBox** as a child of the node **CSGCombiner**.
- Make sure that the **height**, **width**, and **depth** attributes of this node are all **2**.
- Change its position to **(0, 0, 2.56)**, its **rotation** to **(0, 45, 90)** and its **scale** to **(0.5, 0.5, 0.5)**.
- You can also apply a **red** or **green** material to it.
- Add a node of the type **CSGCylinder** as a child of the node **CSGBox**.
- Change this node's **position** to **(0, -3.6, -3.6)**, its **rotation** to **(0, -90, -45)** and its scale to **(1.8, 10, 1.8)**.
- Change its attribute **CSGShape | Operation** to **Subtraction**.

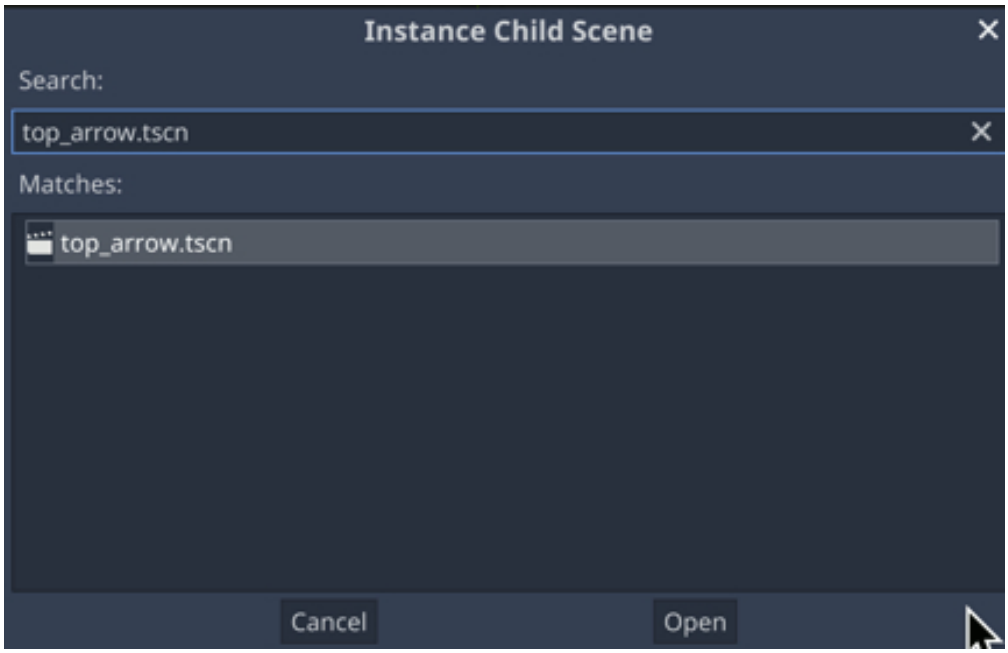


- You should notice that the cylinder is now transparent and that an arrow (i.e., the result of the subtraction between the original box and the cylinder) is visible.



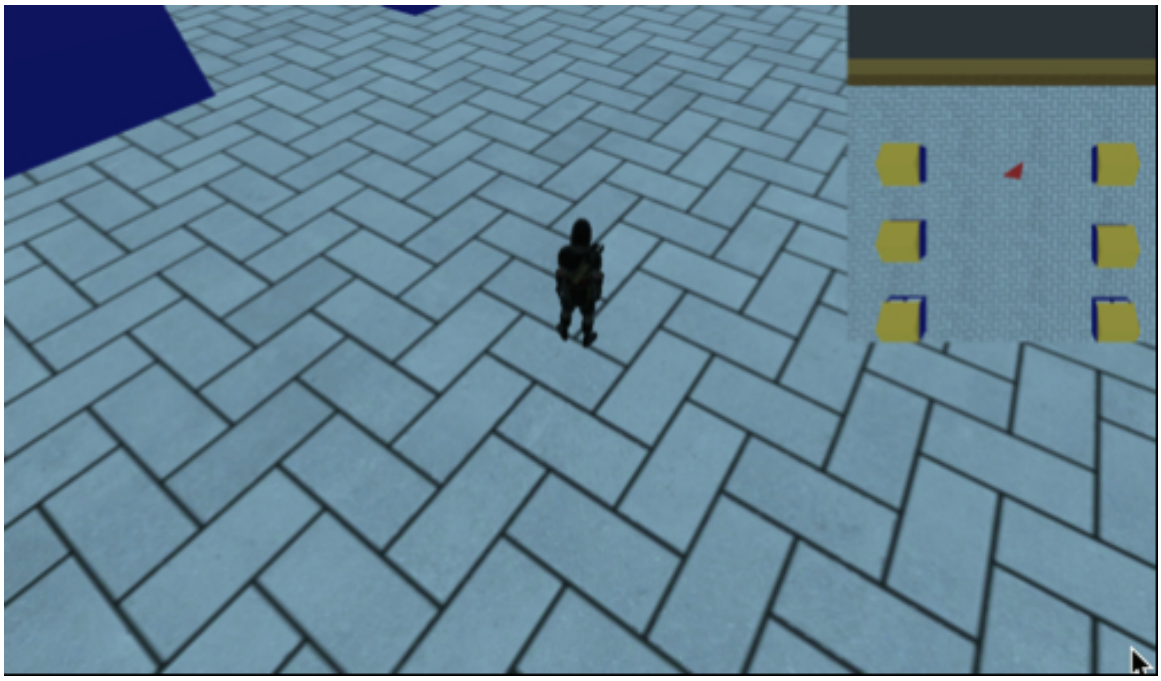
- Please save the current scene.

- Open the main scene (i.e., **level1**).
- Right-click on the node **Player**, and select the option **Instance Child Scene**.
- In the new window, select the scene **top_arrow.tscn** and press **Open**.



- This will create a new node labeled **top_arrow** as a child of the node **Player**.
- Select the node **top_arrow** and change its position to **(0, 20, -2.85)**.

You can now play the scene and you should see an arrow on the mini-map, as illustrated in the next figure.



Level roundup

Summary

Throughout this chapter you have created and animated your own 3D character; you have created a 3D environment and templates that can be re-used later on. Finally, you have created a user interface that includes a mini-map.

Chapter 3: Creating a Dialogue System

This chapter helps you to create a dialogue system whereby the player will be able to talk to NPCs and select his/her answer based on NPCs' sentences. This will be done using JSON and branching structures so that the dialogue is dynamic and also so that it can be amended without a line of code by the game programmer, or a person with no programming background.

After completing this section, you should be able to:

- Understand how JSON files can be used to create a dialogue system.
- Understand the structure of a JSON-based dialogue system.
- Create dialogues for your own game.
- Save your dialogues in a proper format.
- Read these dialogues when the player interacts with a specific character.
- Provide and detect the options chosen by the player during a conversation.
- Detect when the discussion has ended.
- Create a user interface for the dialogues, with an image of the NPC with whom the player is talking.
- Make it possible for some NPCs to talk to the player.
- Detect when the player is near a character that s/he can talk to.

INTRODUCTION

In this chapter, we will create a dialogue system. This will involve:

- Creating a JSON file that includes all the information about each dialogue and the possible answers.
- Reading this file.
- Creating a user interface where the dialogue can be displayed.
- Attaching this **Dialogue System** to a character.
- Detecting when the player collides with this character to be able to start the dialogue.
- Stopping the navigation for the player and detecting its choices for the conversation.
- Detecting when the conversation has ended to hide the user interface and enable the user to resume navigation.

IMPORTING THE JSON FILE FOR THE DIALOGUES

First, we will import a JSON file that includes all the information needed for the dialogues in your game.

- Please create a folder called **dialogues** inside the folder **res://** in Godot (i.e., right-click on the folder **res://** and select the option **New Folder**).
- Drag and drop the file **dialogues.json** from the folder called **dialogues** in the resource pack to the folder **dialogues** in Godot.
- In **Godot**, right-click on the folder **dialogues** and then select "**Open in File Manager**".
- This will open the file system on your computer.
- Inside this folder double click on the file **dialogues.json** to open it with your default text editor. It will look like the following snippet.


```
{
"Diana":
{
"name":"Diana",
"dialogues":
[
{
"id" : "0",
"content": "Hello Stranger! What brings you here!",
"choices":
[
{"content":"Hi there, would you know where I can find the golden globe?", "target":"2"},
{"content": "Hi there, are you from here, Im kind of lost?", "target":"1"}
]
},
```


As you will see:

- The file starts and ends with curly brackets.
- It then includes an object defined by an identifier within double quotes (e.g., “Diana”); followed by a colon, followed by opening and closing curly brackets.
- Within these curly brackets, we define the attribute of the object using an attribute/value pair separated by a comma. The name of the attribute is usually within double quotes, and the corresponding attribute may (if this is a string) or may not use quotes.
- Some attributes can have sub-attributes.
- For example, in this file we an object called “**Diana**”.
- This object has an attribute called **dialogues** which is effectively an array of **objects**.
- Each of these objects includes an attribute called **id**, an attribute called **content**, along with an array of **choices**.
- Each **choice** includes an attribute called **content** along with an attribute called **target**.

So effectively:

- This file consists of nested nodes marked as dialogues.
- This dialogue is associated with a character called "**Diana**".
- The conversation will start with the dialogue with the **id=0**.
- For this dialogue, the player will have two choices.
- Each choice is associated with a target which is the ID of the next dialogue based on the player's answer.
- For example, if the player selects the second answer which is:

" Hi there, would you know where I can find the golden globe?"

- Then the system will branch to the dialogue with the **id=1**, and the NPC will then say:

" Yes, I've been living here for decades, what brings you here? "

- You may notice that answers that end the conversation are marked with the code **target id = -1** which will indicate to the system that the conversation has ended.

Note that this is a text file, and you can modify it if you wish; the beauty of this system is that it can be modified by someone with no coding background and the file can still be used in the game, regardless.

READING THE DIALOGUE FILE

Now that the format of the file is clearer, we will start to create the necessary code to read and use this JSON file.

In the next section, we will start to read this file to create a dialogue system. For this purpose, we will create a new class called **Dialogue** which stores information about dialogues in the game between the player and an NPC.

- In Godot, please click on the tab called **Script** at the top of the window.
- Create a new script called **Dialogue.gd** in Godot (select **File | New Script** from the **Script** workspace).

- Open the script.
- Note that is script is not linked to any node for now.
- Remove all the code from this script.
- Add the following code:

```
var name_of_character : String
var message:String
var array = []
var target_for_response = []
```

In the previous code, since our dialogue system will consist of several dialogues, we create a class for which each instance will store information about a specific dialogue.

- Please save this file (**CTRL + S** or select the option **File | Save** available in the **Script** workspace).

Once we have defined a structure for each dialogue, we can now create our **Dialogue System**, a class that will be used to read the **JSON** file and to proceed with

the conversation between the player and NPCs.

- Please create a new script called **DialogueSystem.gd**.
- Add this code at the beginning of the class (new code in bold):

```
extends Node
var name_of_character
var dialogues = []
var nb_dialogues:int
var current_dialogue_index:int = 0
var waiting_for_user_input:bool = false
var dialogue_is_active
var characters : Dictionary
var current_character = "Diana"
onready var player = get_node("../Player")
```

In the previous code, we create several variables: **name_of_character** will be used to store the name of the NPC, **dialogues** (an array) will store all the different dialogues saved in the JSON file, **nb_dialogues** will provide the number of dialogues for a specific NPC, **current_dialogue_index** will be used to know the ID of the current dialogue, and **waiting_for_user_input** will be used to know whether the NPC is talking or whether the player is required to make a choice. The variable **dialogue_is_ac tive** will be used to know whether the NPC is currently engaged in a dialogue with the player, **characters** include an array of characters that will be able to talk to the player, **current_character** specifies the current character to whom the script is attached, and finally, the variable **player** refers to the **Player** node.

We will now add some code that will make it possible to display the dialogues on screen; some of it refers to nodes that will be created later in this section.

- Please add the following code just after the one that you have created.

```
onready var dialogue_panel = get_node("../dialogue_panel")
onready var dialogue_text = get_node("../dialogue_panel/dialogue_text")
```

```
onready var dialogue_image = get_node("../dialogue_panel/dialogue_image")
```

In the previous code, we create variables that will be linked to the user interface elements that will be created later and that will be used to display the dialogues; these include the variables **dialogue_panel** and **dialogue_text**, **dialogue_image**.

Now that we have defined the key variables for our dialogue system we will create and test a function that will read the JSON file and the related dialogue content.

- Please add the following code to the script:

```
func read_json_file():
var file = File.new()
file.open("res://dialogues/dialogues.json", file.READ)
var json_data = parse_json(file.get_as_text())
var json = to_json(json_data)
characters = JSON.parse(json).result
```

In the following code:

- We create a new function called **read_json_file**.
- We open the file **dialogues.json** stored in the project folder **dialogues**.
- The data within is converted to a JSON format and saved inside the variables called **characters**.

Please add this code just after the code that you have typed.

```
for character in characters:
var name = characters[character].name
print ("Name:"+ name)
for dialogue in characters[character].dialogues:
print ("Message: "+ dialogue.content)
print ("-> Option A: " + dialogue.choices[0].content)
print ("-> Option B: " + dialogue.choices[1].content)
for choice in dialogue.choices:
print ("-> Option: " + choice.content)
```

In the previous code:

- We loop through all the characters included in the file; in our case, there is information only for the character **Diana**.
- We define and save the name of each character and we display the content of each possible dialogue along with the options that will be made available to the player.

Once this is done, we can test this code by doing the following:

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

```
func _ready():
read_json_file()
```

- Please save your code and check that it is error-free
- Create a new **CSGBox** node as a child of the **Spatial** node.
- Rename this box **Diana** so that the corresponding dialogues are used.
- Add the script **DialogueSystem** to this object (i.e., right-click on the node **Diana** and select the option **Attach Script**).

You can now play the scene and look at the **Console** window, and you should see that the **JSON** file has been read properly; the dialogues for the character **Diana** should appear in the **Console** window.

PROCESSING THE USER'S ANSWERS

Now that we know that the dialogues are loaded properly, we will start to trigger the dialogues and make it possible for the player to select their answers by typing a number on their keyboard.

- Please add the following code to the script **DialogueSystem**.

```
func _process(delta):
if (dialogue_is_active):
```

In the previous code, we use the function **_process** that is called every frame and we check whether the dialogue is active.

- Please add this code inside the conditional statement that you have just typed (new code in bold):

```
if (dialogue_is_active):
if (!waiting_for_user_input):
if (current_dialogue_index != -1):
display_dialogue1_for_character(current_character);
else:
dialogue_is_active = false;
waiting_for_user_input = true;
current_dialogue_index = 0
```

In the previous code:

- We check whether we are waiting for the user to press a key.
- If this is not the case, we proceed with the following.
- If the current index is not **-1** (i.e., if the conversation is not finished yet) we call the method **display_dialogue1_or_character** (that we will need to create).
- Otherwise, we reset the variables **dialogue_is_active**, **waiting_for_user_input**, and **current_dialogue_index**.
- We then wait for the user input.

Note that Godot will display an error message in the **Script** window, because we have not yet created the function **display_dialogue1_for_character**, however, this message will disappear as soon as we define this function, and this will be done in the next sections.

We now need to add code to enable the player to select one of the options offered.

- Please add the following code, just after the code that you have typed (new code in bold).

```
current_dialogue_index = 0;
else:
if (Input.is_action_just_released("dialogue_choice_1")):
current_dialogue_index = return_target(0);
waiting_for_user_input = false;
if (Input.is_action_just_released("dialogue_choice_2")):
current_dialogue_index = return_target(1);
waiting_for_user_input = false
```

Note: the first **else** keyword in the previous code should be at the same level of indentation as the code

```
if (!waiting_for_user_input):
```

In the previous code:

- We check whether the player pressed the key **dialogue_choice_1** (for the first option)
- If that's the case, we go to the corresponding dialogue by changing the variable **current_dialogue_index**.
- We specify that we are not waiting for the player's input anymore because it is the NPC's turn to talk.
- We repeat the last three steps if the player selects the second choice.

We will now create the function **return_target**; please add the following code:

```
func return_target(the_choice:int)->int:
var the_target = -1
print("Return target!")
for character in characters:
var name = characters[character].name
if (name == current_character):
the_target = int(characters[character].dialogues[current_dialogue_index].choices[the_choice].target);
return (the_target)
```

In the previous code:

- We create a new function called **return_target**.
- This function, based on the choice selected by the by the player will return the id of the next part of the dialogue.
- This id is determined based on the JSON file **dialogues.json**.

Once this is done, we just need to create the method **display_dialogue1_for_character**.

- Please add the following method to the class file **DialogueSystem**.

```
func display_dialogue1_for_character(the_name:String):
for character in characters:
var name = characters[character].name
if (name == the_name):
print("Message: "+ characters[character].dialogues[current_dialogue_index].content)
print("1: "+ characters[character].dialogues[current_dialogue_index].choices[0].content)
print("2: "+ characters[character].dialogues[current_dialogue_index].choices[1].content)
waiting_for_user_input = true
```

- Add the following function:

```
func start_dialogue():
waiting_for_user_input = false;
dialogue_is_active = true;
```

- Add this code at the end of the **_ready** method:

```
start_dialogue();
```

- You can comment the following code now (as we have it used in the method **start_dialogue**) in the **read_json_file** method (i.e., select the code and press **CTRL/CMD + K**):

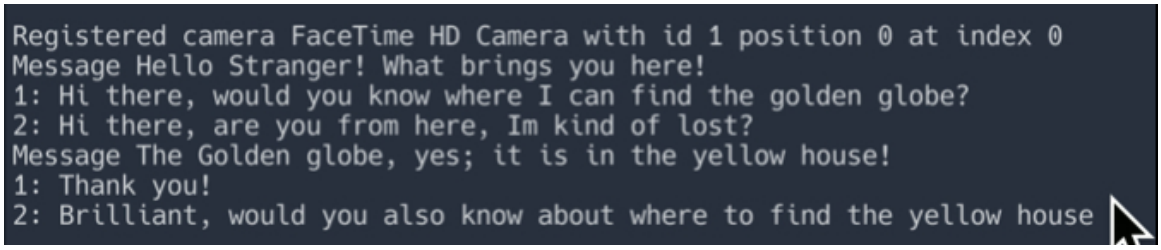
```
for character in characters:
var name = characters[character].name
print ("Name:"+ name)
for dialogue in characters[character].dialogues:
print ("Message: "+ dialogue.content)
print ("-> Option A: " + dialogue.choices[0].content)
print ("-> Option B: " + dialogue.choices[1].content)
for choice in dialogue.choices:
print ("-> Option: " + choice.content)
```

- You can now save the script and, if you haven't already done so, attach it the node object called **Diana**.

Before we can save the scene, we just need to define the actions **dialogue_-choice_1** and **dialogue_choice2**.

- Please Select: **Project | Project Settings | Input Map** and map the actions **dialogue_ choice_1** and **dialogue_choice_2** to the keys **1** and **2** respectively.
- Play the scene, and you should see the dialogue in the **Console** window.
- You can then press the key the keys **1 or 2**, and the next part of the dialogue

will be loaded as per the next figure.



BUILDING A USER INTERFACE

So at this stage, we know that we can create an interactive dialogue between the player and an NPC; this being said, for our game, we will need to display this dialogue through the User Interface (UI), so we will need to create a UI for the dialogue including an image for the character, a background, and text fields where the different sentences and options will be displayed for the player.

- Please create a new node of the type **ColorRect** as a child of the node **Spatial** in the main scene.
- Rename this node **dialogue_panel**.
- Using the **Inspector**, expand the section **Rect**, and change the position (i.e., the attribute **Rect | Position**) to **(15, 363)** and its **size** (i.e., the attribute **Rect | Size**) to **(1000, 200)**.
- Create a new node of the type **RichTextLabel** as a child of the node **dialogue_panel**, rename this node **dialogue_text.**
- Using the **Inspector**, expand the section **Rect**, and change the **position** (i.e., the attribute **Control | Rect | Position**) to **(0, 0)** and its **size** (i.e., the attribute **Control | Rect | Size**) to **(1000, 200)**.
- Finally, create a new node of the type **Sprite** as a child of the node **dialogue_-panel** and rename this node **dialogue_image**.
- Import the image **avatar.jpg** from the resource pack (i.e., drag and drop it from the folder **character-pictures** in the resource pack to the folder **res://** in Godot).
- Using the **Inspector**, drag and drop the image **avatar.jpg** from the **File System** tab in Godot to the right of the label **Texture** in the **Inspector**.
- Move the image so that it is just above the dialogue box, as per the next picture.

So that the text can be seen over the background panel, we can change the color of the background panel:

- Please select the node **dialogue_panel**, and using the **Inspector**, click to the right of the label **Color** (in the section **ColorRect**) and pick a black color for the background.

LINKING THE CODE AND THE JSON FILE TO THE USER INTERFACE

At this stage, the UI has been created and we need to create the code to display the text there for the dialogue, as well as the image that corresponds to the NPC that the player is talking to.

- First, please import the image called **Diana.jpg** from the folder **images** in the resource pack and add it, in Godot, to the folder **res://** (in the **File System** tab).

We can then write the code that will make it possible to display the dialogue on screen.

- Please open the script **DialogueSystem.gd**.
- Please add this code to the **_process** function (new code in bold) in the script **DialogueSystem.gd**.

```
if (dialogue_is_active):
if (!waiting_for_user_input):
dialogue_panel.show()
if (current_dialogue_index != -1):
#display_dialogue1_for_character(current_character);
display_dialogue2_for_character(current_character);
else:
dialogue_is_active = false;
dialogue_panel.hide();
waiting_for_user_input = true;
```

- Finally add the following function to the class:

```
func display_dialogue2_for_character(the_name:String):
for character in characters:
var name = characters[character].name
```

```
var text_to_display = "["+the_name+"]\n";
if (name == the_name):
text_to_display += "Message: "+ characters[character].dialogues[current_dialogue_index].content
text_to_display += "\n-> Option 1: " + characters[character].dialogues[current_dialogue_index].choices[0].content
text_to_display += "\n-> Option 2: " + characters[character].dialogues[current_dialogue_index].choices[1].content
dialogue_text.text = text_to_display
waiting_for_user_input = true
```

In the previous code, the text that was initially displayed in the **Console** window is now displayed in the user interface.

- Please check that your code is error-free, and play the scene.
- You should see a window similar to the next figure.

As you select options (i.e., 1 or 2), the corresponding dialogue will appear on screen and the panel will disappear once the conversation has ended.

TRIGGERING THE DIALOGUE WHEN WE ARE NEAR THE NPC

So the code is working well, the dialogue is displayed and we can go through the different options to answer questions from the NPC; however, the last things we need to do are to display the correct image for the NPC and to trigger this dialogue only when the player is close to the NPC, and we will do this in this section.

- Please open the file **DialogueSystem**.
- Comment the following code in the **_ready** function. This is because we will start the dialogue based on a collision with the NPC.

```
#start_dialogue()
```

- Open the script **manage_player**.
- Add the following code at the beginning of the script (i.e., before the function **_ready**):

```
onready var dialogue_panel = get_node("../dialogue_panel")
onready var dialogue_box = get_node("../dialogue_panel/dialogue_box")
onready var dialogue_image = get_node("../dialogue_panel/dialogue_image")
var name_of_current_npc_talking:String = ""
var is_talking = false;
```

In the previous code, we create three variables that are linked to the nodes dedicated to displaying the dialogues. We also create the variable **name_of_current_npc_talking** that will hold the name of the NPC talking to the player; finally, we define a variable called **is_talking** that will be used to check whether the layer is currently talking.

- Please add this script to the function **_ready** in the script **manage_player**:

```
dialogue_panel.hide()
```

- Please add the following function to the script **manage_player**:

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

```
for index in get_slide_count():
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("NPC_TALK")):
if (collision.collider.name == "Diana" && !is_talking):
dialogue_image.texture = load("res://Diana.jpg");
collision.collider.start_dialogue()
current_state = IDLE
is_talking = true;
```

In the previous code:

- We check whether we are colliding with the an object that belongs to the group **NPC_TALK**.
- We then check the name of this NPC that the player character is not already talking.
- In this case, we start the dialogue.
- We make sure that the player character is in the **IDLE** state while talking.

Please, add this line at the beginning of the function **_input_event** in the script **manage_player**:

```
if (is_talking): return;
```

In the previous code, we ensure that we don't process any keys related to the player's movement during a dialogue.

- Add this new function to the script **manage_player**:

```
func end_talking():
is_talking = false
var forward = global_transform.basis.z;
move_and_slide(-forward*1.5, Vector3.UP)
rotate_y(3.14)
```

In the previous code, we give the control back to the player once the dialogue is finished, so that s/he can move the player character. We also rotate the player

character so that the player can leave in the opposite direction it arrived without the need to walk back.

- Please add this code to the function **_process** in the class script **DialogueSystem** (new code in bold).

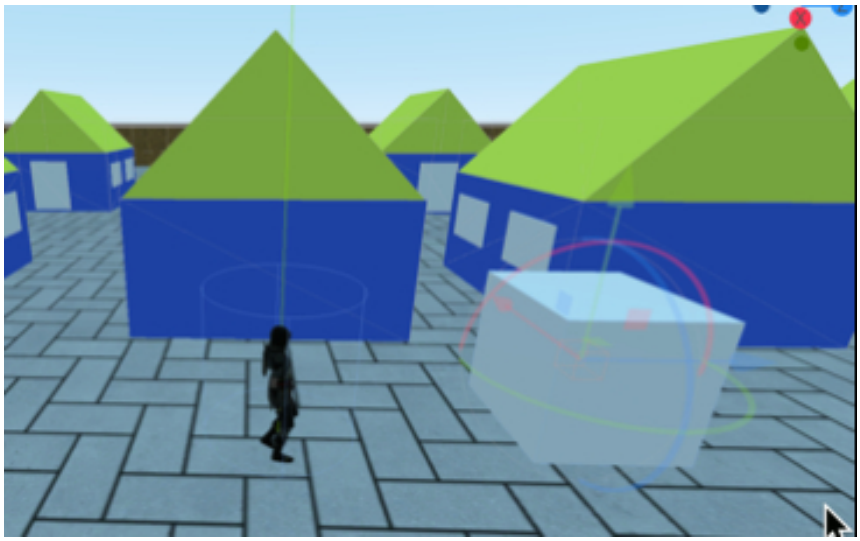
```
else:
dialogue_is_active = false;
dialogue_panel.hide(); #new code
waiting_for_user_input = true;
current_dialogue_index = 0
player.end_talking();
```

In the previous code, we notify the script **manage_player** that the dialogue is finished.

Finally, we just need to ensure that the collision between the player character and the NPC is detected by assigning a group to the NPC and also by adding a collider to it.

Please do the following:

- Right-click on the node **Diana**.
- Select the option "**Change Type**" from the contextual menu, and select the node type "**StaticBody**".
- Add a node of the type **CSGBox** to the node **Diana**.
- Add a node of the type **CollisionShape** to the node **Diana**.
- Select this new node (i.e., **CollisionShape**), and using the **Inspector**, click on the arrow to the right of the label **Shape**, and select the option **New Box Shape**.
- Using the tab **Node | Group**s, assign the group **NPC_TALK** to the node **Diana**.
- Move the node **Diana** so that it is a few meters away from the player.



- Once this is done, you can play the scene, and move the player character close to the NPC; as you get closer to the NPC, the dialogue window should appear, as illustrated in the next figure.

ADDING A 3D CHARACTER FOR THE NPC

At this stage the system is working properly; the next step is to use a 3D character for the NPC. For this purpose, we will re-use the **akai** 3D character, apply an **idle** animation to it, and also link it to a dialogue system.

- Please drag the animation **akai.gltf** atop the node **Diana**, so that it becomes a child of this node and delete the node called **CSGBox** that is a child of the node **Diana** (i.e., right-click on that node and select the option **Delete Node** from the contextual menu).
- The node **Diana** should now have two child nodes, as illustrated in the next figure.

- Please select the node **CollisionShape** that is a child of the node **Diana**, and change its **scale** attribute (i.e., **Transform | Scale**) to **(0.5, 1, 0.5)**.
- Change the position of the node **akai** that is a child of the node **Diana** to **(0, -1.1, 0)**.
- Test the scene, you should see that after walking near the NPC the dialogue should start.

ANIMATING THE NPC

The last thing we will do is to animate this character using the **idle** animation that we have used earlier for the player character.

- Please select the node **akai** that is a child of the node **Diana**.
- Attach a new script called **manage_talking_npc** to the node **akai**.
- Open this script and add the following code to it (new code in bold):

```
onready var pc_node = get_node("AnimationPlayer")
func _ready():
pc_node.get_animation("idle").loop = true
func _process(delta):
pc_node.play("idle")
```

In the previous code, we ensure that animation called **idle** will be played continuously.

- Please save your script and play the scene, and you should see that the character **Diana** is **idle**.

LEVEL ROUNDUP

Summary

That's it, you have managed to create your dialogue system from a JSON file; you created a system whereby dialogues could be stored and read to generate dialogues and possible branching scenarios for this dialogue. Finally, you applied this dialogue to an actual character and implemented a user interface where the player can see the different possible options in order to conduct a conversation with the character. Well done!

Quiz

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

1. A JSON document is a text file.
2. In JSON documents, it is possible to create arrays.
3. In JSON documents, each object cannot have several attributes.
4. It is possible to read a JSON file from GDScript.
5. A node of the type **CollisionShape** can be used to detect collisions between the player and other objects.
6. The event **body_entered** is called whenever a **KinematicBody** node enters an **Area** node.
7. The event **body_exited** is called whenever a **KinematicBody** node exits an **Area** node.
8. The method **file.open** can be used to load files from the folder **res://**.
9. In Godot, a **ColorRect** node can be created to display a 2D panel on screen.
10. By default, a **ColorRect** node will fill the size of the screen, unless specified otherwise.

Solutions to the Quiz

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

Checklist

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

- Modify a JSON file for the dialogues.
- Understand the structure of a JSON file used for dialogues.
- Read a JSON file.

Chapter 4: Creating an Inventory System

In this chapter, we will create an inventory system and make it possible for the player to collect items.

After completing this section, you should be able to:

- Add objects to be collected in the scene.
- Add objects to an inventory.
- Display the inventory.
- Use the objects in your inventory to increase the player's health.

Introduction

In this section, we will create an inventory system whereby the player will be able to collect and use items.

So in this section, the creation of the inventory system will use the following steps:

- We will create a class that will be used to store information about the items that can be collected, such as their name, health benefits (if any), the damage they can inflict (if any), a description, and the maximum number of the items that the player can carry.
- We will then create an inventory system that will be used to display all the items that the player owns on screen; some items will be provided to the player by default at the start of the game, and some other items will be collected throughout the game by the player.
- We will create a system whereby the player can collect items and update the inventory accordingly.
- Finally, we will make it possible for the player to use some of the items in the inventory to increase his/her health.

Creating classes to store information on the items to be collected

So first, let's start with the creation of the class used for each item in the inventory.

- Please switch to the **Script** workspace (i.e., click on the tab **Script** at the top of the Godot window), and create a new script (i.e., **File | New Script**) called **Item.gd**.
- Open the script **Item**.
- Empty its content (**CTRL + A** then **CTRL + X**).
- Add the following line at the beginning of the script:

class_name Item

In the previous code, we are mentioning that this script is a class (that can be instantiated later).

Once this is done we can start to define the different types of items that we can collect, by adding the following code at the beginning of the script:

enum item_type {APPLE = 0, MEAT = 1, GOLD = 2, RED_DIAMOND = 3, BLUE_DIAMOND = 4, YELLOW_DIAMOND = 5,SWORD = 6, BATON = 7}

In the previous code:

- We declare an **enum** called **item_type**, which is an enumeration of different items types all with a name and a corresponding number. Note that **enums** are usually defined in base 0 (i.e., first element is **0**, second is **1**, etc.), so in theory there was no need to include the numbers as we have above (for learning purposes) and we could have written the following code as follows:

enum item_type {APPLE, MEAT, GOLD, RED_DIAMOND, BLUE_DIAMOND, YELLOW_DIAMOND, SWORD, BATON}

- So we will be able to collect apples, meat, gold, red diamonds, blue diamonds, yellow diamonds, a sword, or a baton.

Once this is done, we can start to declare the different types of properties for all

of the items to be collected.

- Please add the following code to the beginning of the class, just after the code that you have just typed.

```
enum item_family_type { FOOD = 0, LOOT = 1, WEAPON = 3}
```

In the previous code, we create an **enum** called **item_family_type** that defines three types of families for the items to be collected, namely: **food**, **loot**, or **weapon**.

- Please add the following line, just after the code that you have just typed:

```
var item_type_names =["APPLE", "MEAT","GOLD","RED_DIAMOND", "BLUE_DIAMOND", "YELLOW_DIAMOND","SWORD","BATON"]
```

In the previous code, we declare an array that stores the names of the different items that can be collected.

- Please add the following just after the code that you have just typed.

```
var name: String
var description: String
var  price:int;
var health_benefits:int
var dammage:int
var type:int
var family_type:int
var nb:int
var max_nb:int
var article:String
```

In the previous code, we specify that each item will have a name, a description, a price, possible health benefits, and possible associated damage (i.e., for weapons). We also specify the items' type and family type (i.e., whether it is food, loot, or a weapon), how many items of this type have been collected, the maximum number of items of this type that can be carried by the player, and an article which

will be used when displaying the item's name (e.g., **an** apple, **a** sword, etc.)

Next, we will create a constructor, a method that will be called whenever we want to instantiate items based on the class **Item**; this constructor will create an item based on its type, and also set the item's properties accordingly (e.g., name, description, etc.).

- Please add the following function after the code that you have just typed.

```
func _init(the_new_type):
match (the_new_type):
```

In the previous code:

- We create a method called **_init** which is a constructor.
- This method will take a parameter which is the type of item to be created.
- We then create a match structure (similar to the **switch** structure in other programming languages) based on the type passed as a parameter.

We will now define how each type of item should be created (or constructed):

- Please add this code within the **match** structure (new code in bold).

```
match (the_new_type):
item_type.APPLE:
name = "Apple";
price = 50;
health_benefits = 10;
dammage = 0;
nb = 1;
max_nb = 5;
description = "A juicy apple.";
family_type = item_family_type.FOOD;
article = "an";
item_type.MEAT:
```

name = "Meat";

price = 50;

health_benefits = 30;

dammage = 0;

nb = 1;

max_nb = 2;

description = "A nice piece of cooked meat, to nurture yourmuscles.";

family_type = item_family_type.FOOD;

article = "some";

In the previous code, we specify the properties of the items of the type **apple** and **meat**; for both these items, we specify their name, their price (that will be used later in the shop system), their health benefits (how much health points the player will gain by using them), the damage caused if used (none in our case), how many these items the player owns whenever it is added to the inventory, the maximum number of this type of item that the player can carry with him/her, a description of the item, the family it belongs to, as well as the article that should be used when describing this item; for example **some** meat or **an** apple (this will be used to write a message whenever the player has found this item).

- Please add the following code just after the code that you have added in the **match** structure.

item_type.RED_DIAMOND:

name = "Red diamond";

price = 250;

health_benefits = 0;

dammage = 0;

nb = 1;

max_nb = 10;

description = "A valuable diamond crafted by the best jewelllers with some known magic properties"

family_type = item_family_type.LOOT;
article = "a";
item_type.YELLOW_DIAMOND:
name = "Yellow diamond";
price = 200;
health_benefits = 0;
dammage = 0;
nb = 1;
max_nb = 10;
description = "A valuable diamond crafted by the best jewelllers with some known magic properties";
family_type = item_family_type.LOOT;
article = "a";
item_type.BLUE_DIAMOND:
name = "Blue diamond";
price = 100;
health_benefits = 0;
dammage = 0;
nb = 1;
max_nb = 10;
description = "A valuable diamond crafted by the best jewelllers with some known magic properties";
family_type = item_family_type.LOOT;
article = "a";
item_type.GOLD:
name = "Gold";
price = 100;
health_benefits = 0;
dammage = 0;
nb = 10;
max_nb = 10;

```
description = "Gold Coins";
family_type = item_family_type.LOOT;
article = "some";
```

- Please add this code after the previous one:

```
item_type.SWORD:
name = "Sword";
price = 100;
health_benefits = 0;
dammage = 10;
nb = 1;
max_nb = 1;
description = "A powerful sword that defeats most opponents.";
family_type = item_family_type.WEAPON;
article = "a";
item_type.BATON:
name = "Baton";
price = 50;
health_benefits = 0;
dammage = 50;
nb = 1;
max_nb = 1;
description = "A simple wooden stick that you can handle easily.";
family_type = item_family_type.WEAPON;
article = "a";
type = the_new_type
```

In the last line of code, we just set the type of the new item created to the type passed as a parameter to the constructor.

Finally, we will also need to have an image that represents the item created, so we will create a method that returns an image (or its location) based on the name

and type of the current item so that it can be displayed in the inventory system accordingly.

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

```
func  get_texture_path():
var tx = ""
if (family_type == item_family_type.WEAPON):
return "res://weapons/" + name.replace(" ","_") +".png"
elif  (family_type == item_family_type.FOOD):
return "res://food/" + name.replace(" ","_") +". png "
elif  (family_type == item_family_type.LOOT):
return "res://loot/" + name.replace(" ","_") +". png "
else: return ""
```

In the previous code, based on the type of the current item, we return the location of the corresponding image. This code assumes that three folders, **weapons**, **loot**, and **food**, have been created in the project folder and that all necessary images have been copied to the corresponding folder, and we will do that in the next sections.

Creating a user interface for the inventory system

We can now start to create an inventory system whereby the player can avail of and collect items. This inventory will be displayed on screen whenever the player wants to see it.

We will now start to design the user interface that displays the inventory, and we will then create the code that will link the items present in the inventory to the user interface.

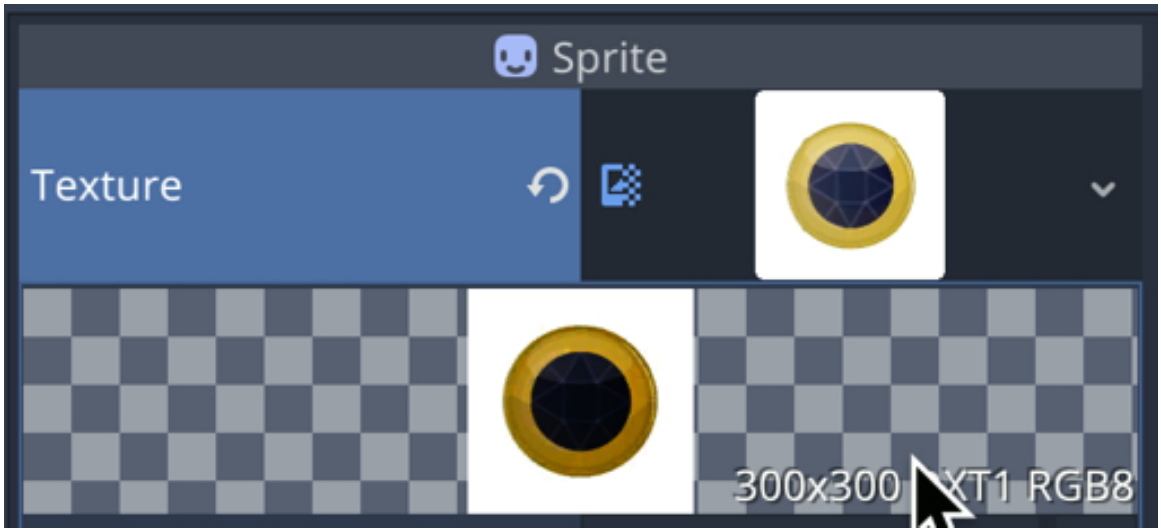
To make things easier, you can temporarily hide the **dialogue_panel** node.

- Please create a new node of the type **Node2D** as a child of the node **Spatial** and rename this node **inventory**.
- Add a node of the type **ColorRect** as a child of the node **inventory**, and re-name this new node **inventory_panel**.
- Using the **Inspector**, click to the right of the label **ColorRect | Color** and change the **alpha** attribute to **63**.

- Using the same node (i.e., **inventory_panel**), expand the section called **Rect** in the **Inspector**, and change its position to **(306, 56)**, and its size to **(700, 300)**.
- Add a node of the type **RicheTextLabel** as a child of the node **inventory** and re-name this new node **inventory_description**.
- Select this new node; using the **Inspector**, expand the section called **Rect**, and change its position to **(435, 125)**, and its size to **(200, 100)**.
- Duplicate the node **inventory_description** and rename the duplicate **inventory_text**.
- In the **Inspector**, expand the section **Rect**, and change the **Position** attribute to

(435, 97) and the **Size** attribute to **(200, 40)**.

- Add a new node of the type **Sprite** as a child of the node **inventory**, and rename this node **inventory_image**.
- Select the node **inventory_image**, change its position (i.e., **Transform | Translation**) to **(359, 110)** and its scale to **(0.329, 0.329)**.
- Finally: create a folder called **loot** in Godot's **FileSystem**, drag and drop the image **blue_diamond.png** from the folder **images** in the resource pack to that folder, and then drag this image from its folder in Godot to the right of the label attribute in the **Inspector** (for the node **inventory_image**).



Displaying and updating the inventory from the code

Once you have created the interface for the inventory, we can then use the code to access these different fields.

- Create a new node of the type **Node** as a child of the node **Player**.
- Rename this new node **inventory_system**.
- Attach a new script called **InventorySystem** to the node **inventory_system**.
- Open this script.
- Add this code at the beginning of the script.

```
var player_inventory;
var current_inventory_index = 0;
var is_visible = false;
var path = "../../inventory/"
onready var inventory_text = get_node(path + "inventory_text")
onready var inventory_image = get_node(path + "inventory_image")
onready var inventory_description = get_node(path + "inventory_description")
onready var inventory_panel = get_node(path + "inventory_panel")
```

In the previous code, we create an array called **player_inventory** that will be used to store all the objects that the player owns. We set the **inventory_index** variable to **0**; this variable will specify the current item displayed on screen; the variable called **is_visible** is used to check whether the inventory is visible on screen. We create a variable called path that stores the path from the current script to the **inventory** node that is used to display the inventory. Finally, we create four variables **inventory_text**, **inventory_image**, **inventory_description**, and **inventory_panel** which are linked to the corresponding nodes.

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

```
display_ui(false);
player_inventory = []
var new_item = Item.new(Item.item_type.GOLD)
```

```
player_inventory.push_back(new_item)
new_item = Item.new(Item.item_type.MEAT)
player_inventory.push_back(new_item)
player_inventory[1].nb = 300;
```

In the previous code:

- We call the method **display_ui** that we have yet to create to empty the text in all the text field created earlier.
- We create a new item of the type **GOLD**.
- We add this item to the inventory.
- We repeat the last steps to add meat to the inventory.
- We initialize the array **player_inventory**.
- We also specify that the player will have 300 gold coins.

Note that the **Script** window will display an error because we are calling a function **display_ui** that we have not defined yet, but this will be done in the next sections.

Once this is done, we could check that our inventory has been created properly by creating a simple method that displays its content.

Let's create this method:

- Please add the following method to the class **InventorySystem**.

```
func check_inventory():
for i in range(player_inventory.size()):
print(player_inventory[i].item_info());
```

In the previous code, we go through each item present in the player's inventory and call the method **item_info** (that we yet have to create) and that displays information about a specific item.

- Please add this method to the class **Item**.

```
func item_info()->String:
```

var info = "name:" + str(name) + ", health benefits:" + str(health_benefits) + ", dammage:" + str(dammage) + ", nb:" + str(nb) + ", max Nb:" + str(max_nb) + ", type:" + str(item_type_names[type]);

return info;

- Once this is done, we need to call the method **check_inventory** from the **_ready** method in the class **InventorySystem**, by adding this line at the end of the function **_ready**.

check_inventory();

- For testing purposes, please comment this code in the function **_ready** that is in the script **InventorySystem**.

display_ui (false);

You can now do the following save your code and play the scene: you should see the following in the **Console** window.

name:Gold, health benefits:0, dammage:0, nb:10, max Nb:10, type:GOLD

name:Meat, health benefits:30, dammage:0, nb:300, max Nb:2, type:MEAT

Now that we know that the inventory is saving information about our items, it is time to display this inventory on screen.

- Please remove the comment from this code in the file **InventorySystem**.

display_ui(false);

The first thing about this inventory is that it should not be displayed until the player presses the key **I**. Before or after pressing this key, images and/or text for this inventory should be hidden. This will be done through a new method called **display_ui** that will hide/display the interface.

- Please add the following function to the class **InventorySystem**.

func display_ui(toggle:bool):

if (toggle):

```
inventory_text.show();
inventory_panel.show();
inventory_image.show();
inventory_description.show();
else:
inventory_text.hide();
inventory_panel.hide();
inventory_image.hide();
inventory_description.hide();
```

In the previous code, we pass a Boolean variable to the method that will be used to determine whether each object that is a child of the object **inventory** should be displayed.

Next, we will be dealing with the display of the UI for the inventory based on the user's input. This will be done in the function **_process**.

First, we need to check whether the UI is displayed:

- Please add this method to the file **InventorySystem**.

```
func _process(delta):
if is_visible:
#some code will be added here in the next section
else:
if (Input.is_action_just_pressed("inventory")): is_visible = true;
```

In the previous code, we use the variable **is_visible** (that is initially set to **false**) to decide on whether the UI should be displayed. We also check that if the UI is not currently displayed, it can become visible when the player presses the key mapped to the action **inventory** (we will map this key in the next sections).

Next, we need to display the UI and also select which items will need to be displayed (i.e., the items that are owned by the player).

- Please add this code to the function **_process**, within the portion of the code that deals with when the UI is visible.

```
display_ui(true)
var current_item = player_inventory[current_inventory_index];
inventory_text.text = current_item.name + "[" + str(current_item.nb) + "]";
inventory_description.text = current_item.description +"\n\n Press [U] to Select";
inventory_image.texture = load(current_item.get_texture_path());
if (Input.is_action_just_pressed("inventory")):
current_inventory_index +=1
if (current_inventory_index >= player_inventory.size()):
current_inventory_index = 0;
is_visible = false
display_ui(false)
```

In the previous code:

- We display the UI.
- We specify that the current item to be displayed is the item at the position **current_inventory_index** in the inventory.
- We display how many of these specific items are owned by the player.
- We display the description for this type of item, along with a picture.
- We then check whether the key mapped to the action **inventory** has been pressed, and we increase the value of the variable **current_inventory_index** accordingly so that we can display the next item in the inventory.
- We also ensure that the inventory is not visible once we have gone through all the items within the inventory.
- For the image: assuming that we have stored the image in the project folder (we will do that in a few seconds), we navigate to this folder, pick the image and set it as the image to be displayed for the current item on screen).
- This assumes that we have created the folders **weapons**, **food**, and **loot** in the folder **res://** and that we have copied the image for each possible item in their folders also.

Let's create these folders and copy these images:

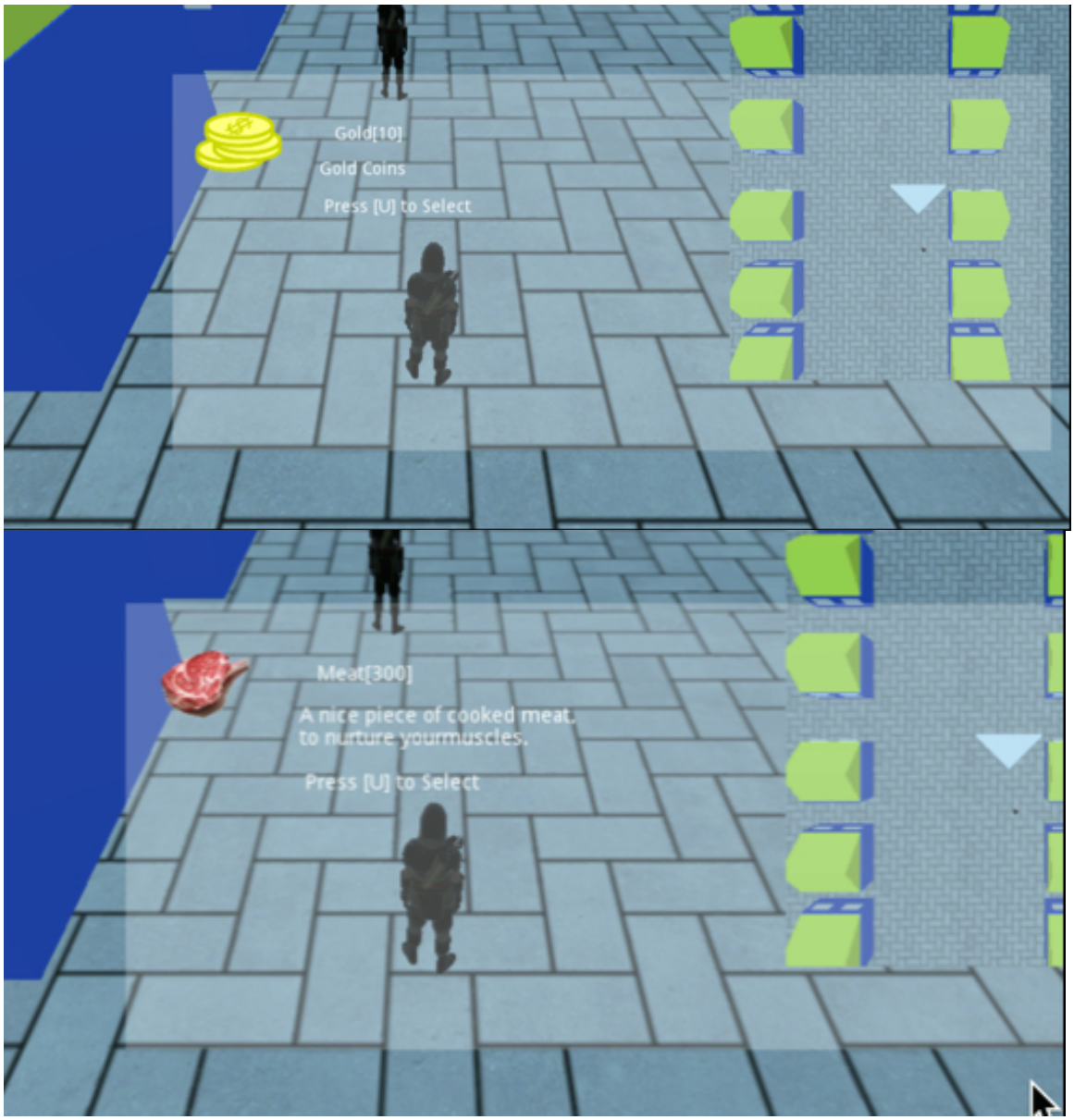
- Please save your code.
- In the **FileSystem** tab, navigate to the folder called **res://**.
- Create two folders **food** and **weapons**. You have already created the folder **loot** so there is no need to do it again here
- Locate the folder called **images** in the resource pack.
- Drag and drop the image **apple** and **meat** from the resource pack (in the folder **images**) to the **food** folder that you have created in Godot.
- Drag and drop the images for the **diamonds (i.e., green_diamond, red_diamond,** and **yellow_diamond)** and **gold** to the **loot** folder in Godot. You have already imported the file **blue_diamond** so there is no need to do it again here.
- Drag and drop the images **sword** and **baton** to the **weapons** folder in Godot.

At this stage, we can check that the UI works properly; beforehand, please uncomment this line in the **_ready** function in the script **InventorySystem** so that the inventory is initially hidden (if you have not already done so).

```
display_ui(false);
```

Before we can play the scene, please map the action **inventory** to the key **I** using the **Input Map** (i.e., **Project | Project Settings | Input Map)**.

You can play the scene and press the **I** key, and you should see information about the **meat** and the **gold** that the player will own by default; for more testing, you could modify the file **Item.gd**, so that the attribute **nb** is **2** or **3** for the **meat**, for example.



You may have noticed that we ask the player to select the object but there is no mechanism yet in place to do so, and we will cover this feature in the next sections.

Creating objects to be collected for the inventory

While we were able to set up and display an inventory in the previous section, we will now enable the player to pick up objects and to add them to his/her inventory.

The main steps will be as follows:

- We will detect which objects are available near the player.
- A message will be displayed to the player to indicate that s/he can pick up an object.
- The players will be able to choose to collect the object; if they decide to do so, the object will be destroyed from the scene and added to the player's inventory accordingly; if the player decides not to collect this object, then the previous message will be hidden, and the player will need to move away from the object (at least 1 meter away) to be able to try to collect it again. The message will automatically be hidden if the player has walked one meter beyond the object.

First, we will implement a template for the objects that can be collected. This will save you time when you create your own game, as you will be able to add a generic object to the scene, and to then, using the **Inspector**, choose the type of this object (e.g., apple, meat, etc.).

- Please create a new scene of the type **Area** (**Scene | New Scene | Other Node | Area**).
- This will create a new scene with a default node called **Area**.
- Please rename this node **object_to_collect**.
- Add a node of the type **CollisionShape** as a child of the node **object_to_collect**.
- Select this new node and after clicking to the right of the label **CollisionShape | Label** in the **Inspector**, select the option **New Box Shape**.
- Once this is done, click on the label called **BoxShape** to the right of the attribute **Shape**, and set the **Extents** attribute to **(.3, .3, .3)**.

- Save this scene as **object_to_collect.tscn**.
- In the same scene, add a node of the type **CSGBox** as a child of the node **object_to_collect**, and change the **scale** attribute (i.e., **Transform | Scale**) of this node to **(0.2, 0.2, 0.2)**. This node is there to help you visualize where the object to be collected will be in the editor; this being said, its appearance will be changed using the proper texture at run-time.

Once this is done, we just need to create the script that will handle the collision with this object.

- Please attach a new script called **object_to_be_collected** to the node **object_to_collect**.
- Add the following code to the script, before the function **_ready**.

```
export (PackedScene) var baton
export (PackedScene) var sword
export (PackedScene) var apple
export (PackedScene) var meat
export (PackedScene) var gold
export (PackedScene) var red_diamond
export (PackedScene) var yellow_diamond
export (PackedScene) var blue_diamond;
export(int, "APPLE", "MEAT","GOLD","RED_DIAMOND", "BLUE_DIAMOND", "YELLOW_DIAMOND","SWORD","BATON") var type
var item;
```

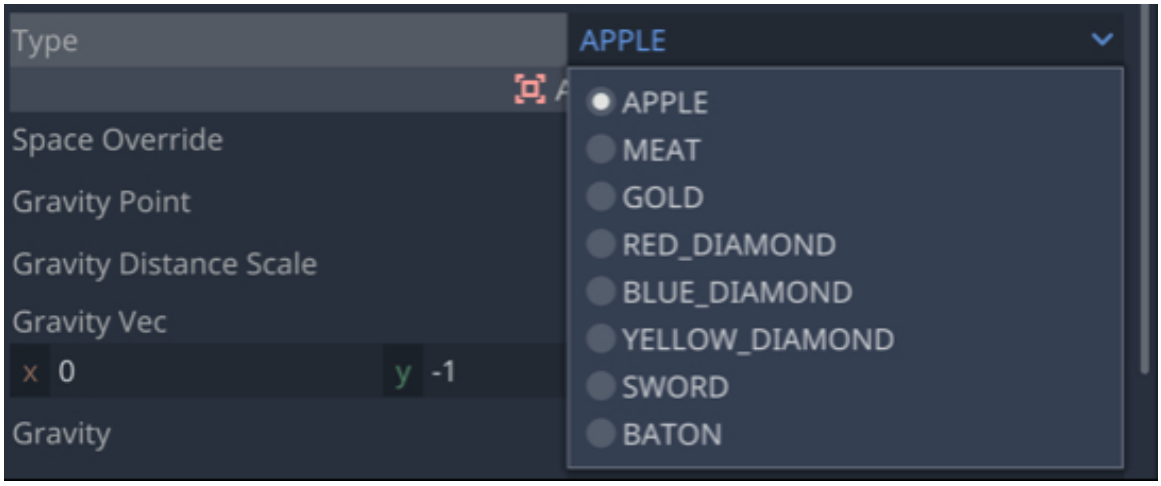
In the previous code, we create placeholders for the 3D nodes that will be used to represent the items to be collected. We also define a variable called **type** so that these items can be configured by the game designer (you) later on when the level is set-up. Finally, we define the variable **item** and its corresponding type.

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

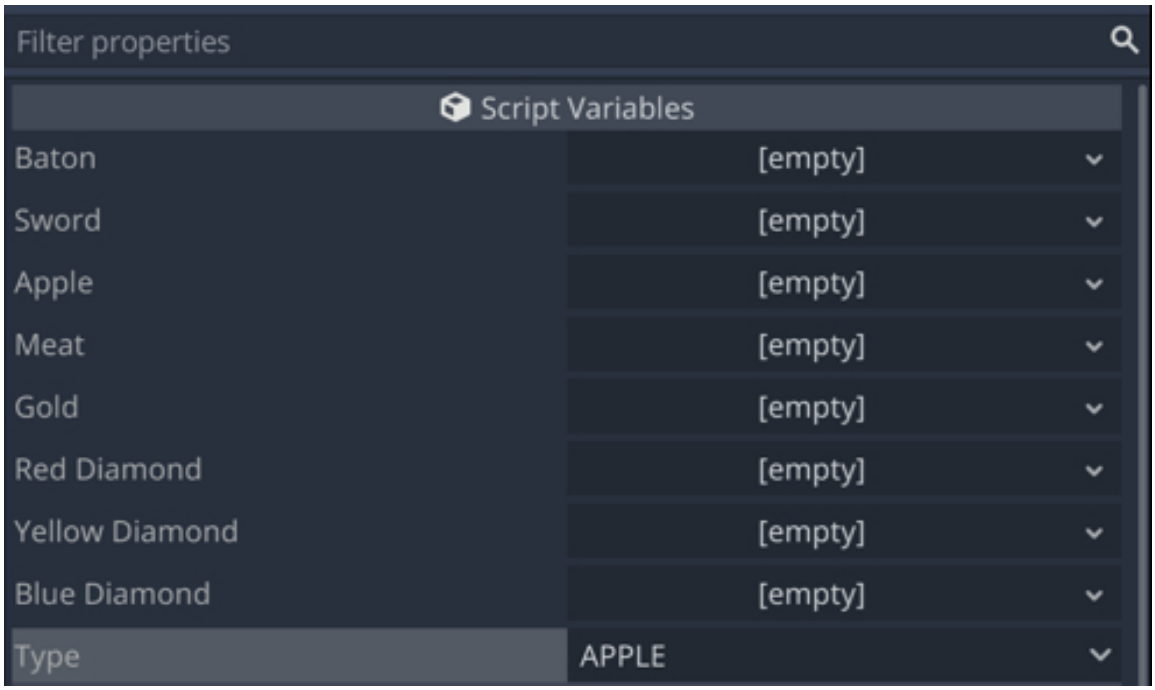
```
item = Item.new(type)
var new_object_to_collect
new_object_to_collect = baton.instance()
match type:
Item.item_type.BATON:
new_object_to_collect = apple.instance()
Item.item_type.SWORD:
new_object_to_collect = sword.instance()
Item.item_type.APPLE:
new_object_to_collect = apple.instance()
Item.item_type.MEAT:
new_object_to_collect = meat.instance()
Item.item_type.GOLD:
new_object_to_collect = gold.instance()
Item.item_type.RED_DIAMOND:
new_object_to_collect = red_diamond.instance()
Item.item_type.YELLOW_DIAMOND:
new_object_to_collect = yellow_diamond.instance()
Item.item_type.BLUE_DIAMOND:
new_object_to_collect = blue_diamond.instance()
_:
pass
add_child(new_object_to_collect)
new_object_to_collect.global_transform.origin = global_transform.origin
```

In the previous code, we create a new instance of an **Item** that will be linked to the object to be collected. We also instantiate the 3D object that will represent the object to be collected.

Please save your code, and look at the **Inspector**, you should see that a field called **type** has appeared for the section **ScriptVariables**.

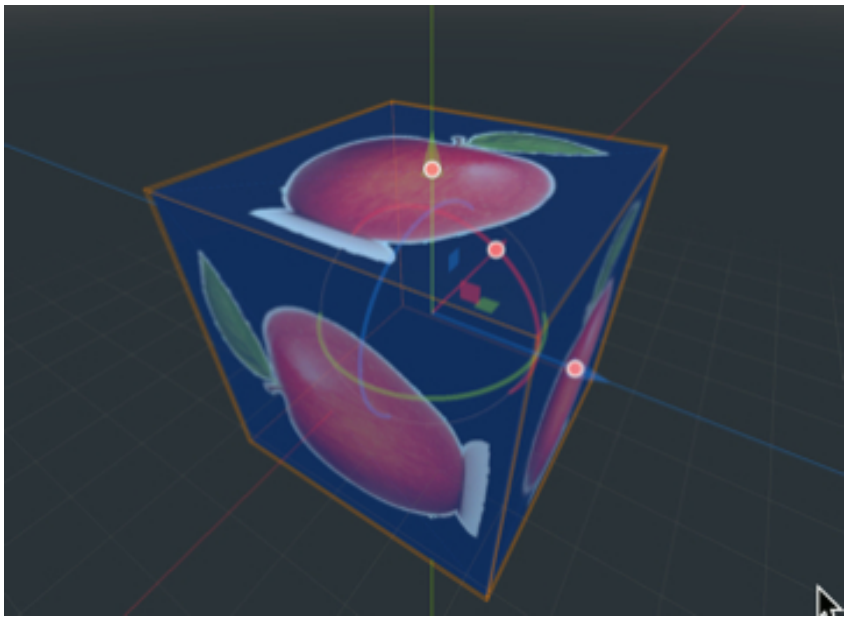


You will also see eight empty fields called **baton**, **sword**, **apple**, **meat**, **gold**, **red diamond**, **red diamond**, **blue diamond**, **and yellow diamond**. At this stage, these empty fields correspond to templates/scenes (to be created) that are the 3D representations of each of the possible 3D objects to include in the inventory.

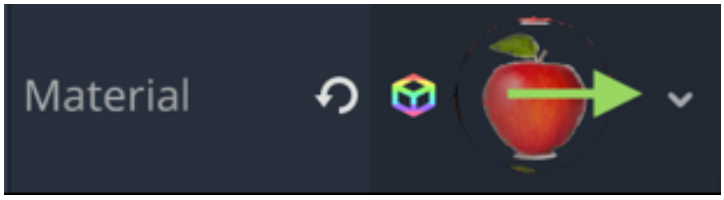


We will now need to create these templates.

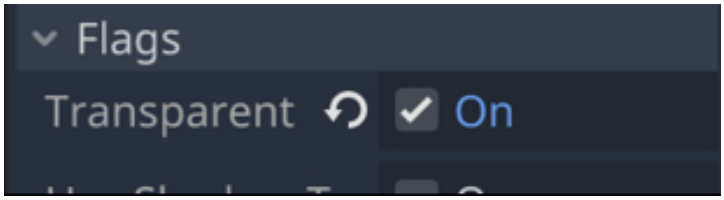
- Please create a new scene of the type **CSGBox**.
- This will create a new scene with a box called **Spatial**.
- Change the **scale** (i.e., **Transform | Scale**) to this node to **(.3, .3, .3)**.
- Using the section material, create a new **Spatial Material** and use the image **apple.png** (located in the **food** folder) as a texture for this node (i.e., use the section **Albedo | Texture**).



- Click on the downward facing arrow to the right of the label material, as per the next figure, and select the option to "**Edit**".



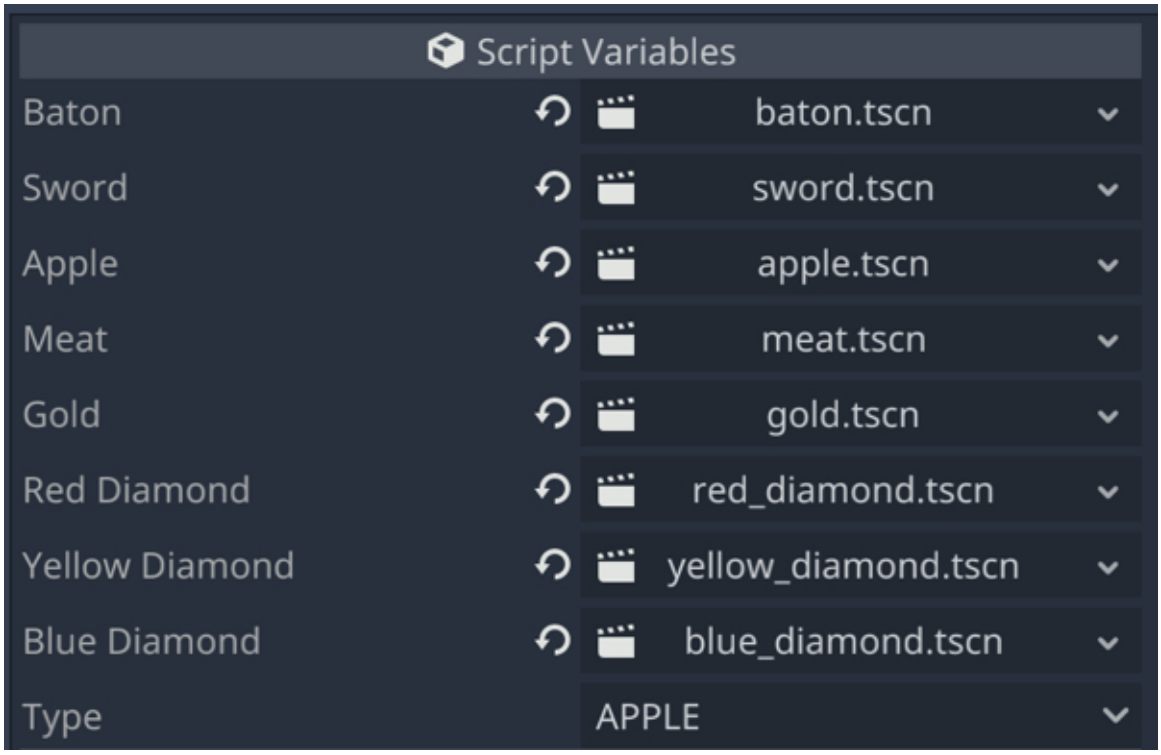
- In the new window, expand the section called **Flag** and set the attribute **Transparent** to **On**.



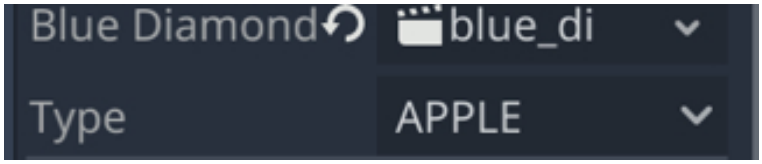
- Save this scene as **apple.tscn.**
- Duplicate this scene **seven** times (i.e., select the scene on the **FileSystem** tab and press **CTRL + D** or **CMD + D**) and rename the duplicates: **baton.tscn**, **sword.tscn**, **meat.tscn**, **gold.tscn**, **blue_diamond.tscn**, **yellow_diamond.tscn**, **green_diamond.tscn**.
- Open each of these scenes and modify the texture used for the **CSGBox** node

to the correct texture (i.e., **baton.png**, **sword.png**, etc.).

- Save each scene.
- Please open (or switch to) the scene **object_to_collect**.
- Select the node **object_to_collect**.
- Drag and drop the scenes that you have created from the **FileSystem** tab (i.e., **apple.tscn, meat.tscn**, etc.) to their corresponding placeholders for the script **object_to_be_collected** that is linked to the node **Area**.



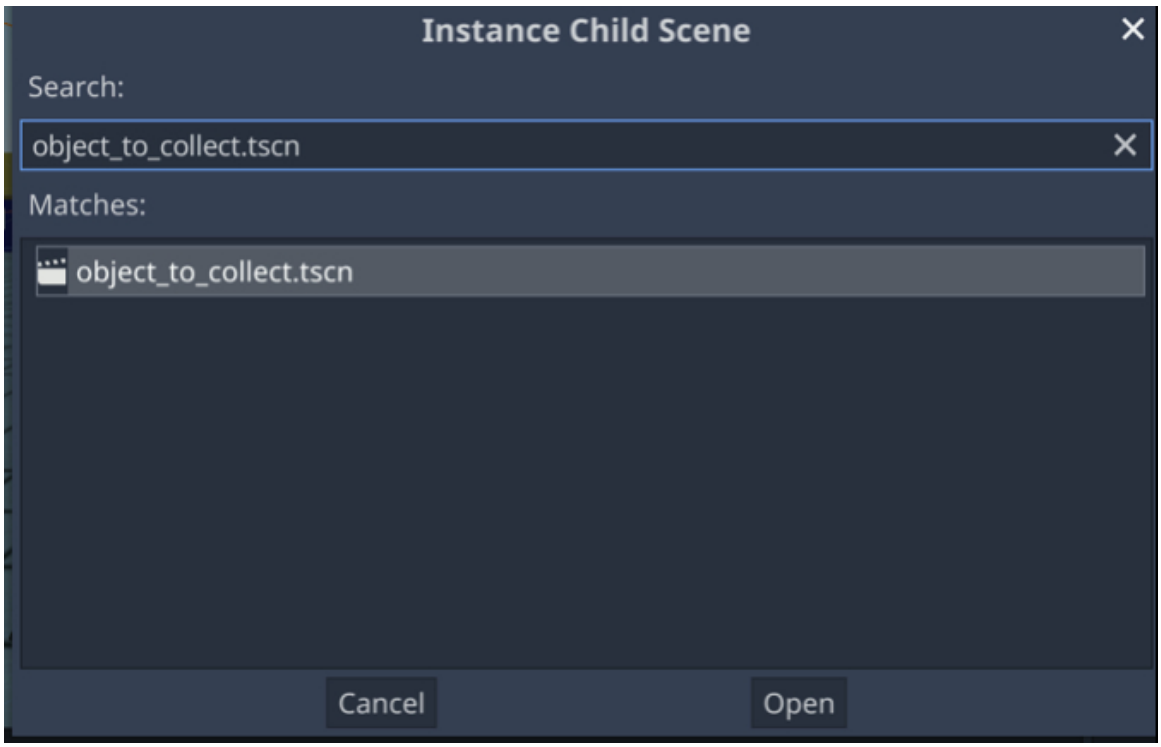
- Select the type **APPLE**.



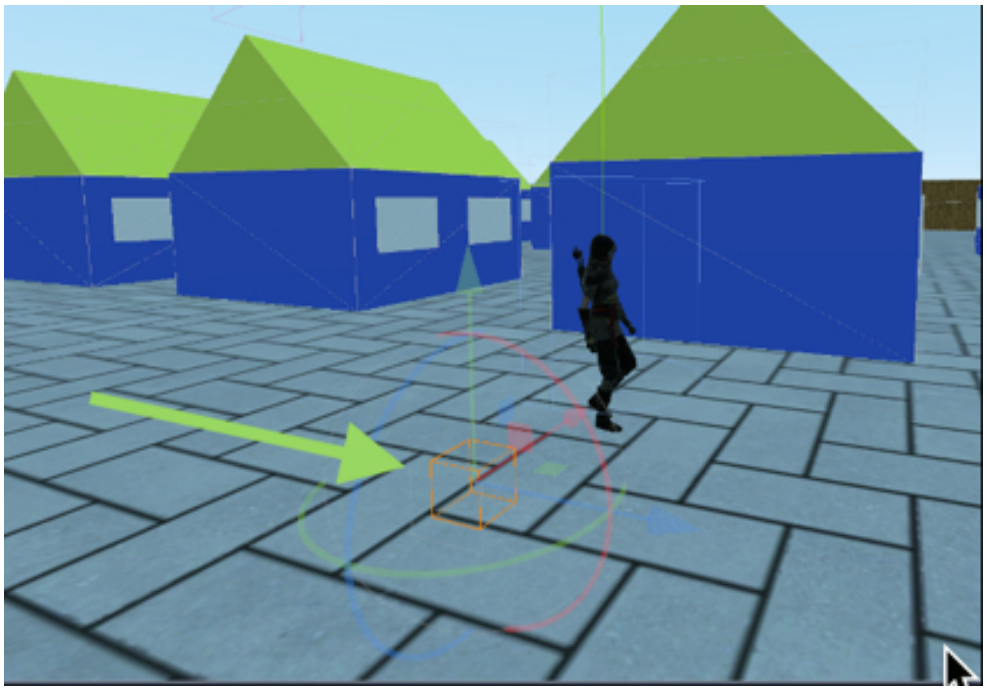
Once this is done, you can do the following:

- Save the scene.
- Open the main scene (i.e., **level1**).
- Right-click on the node **Spatial**, and select the option **Instance Child Scene**.

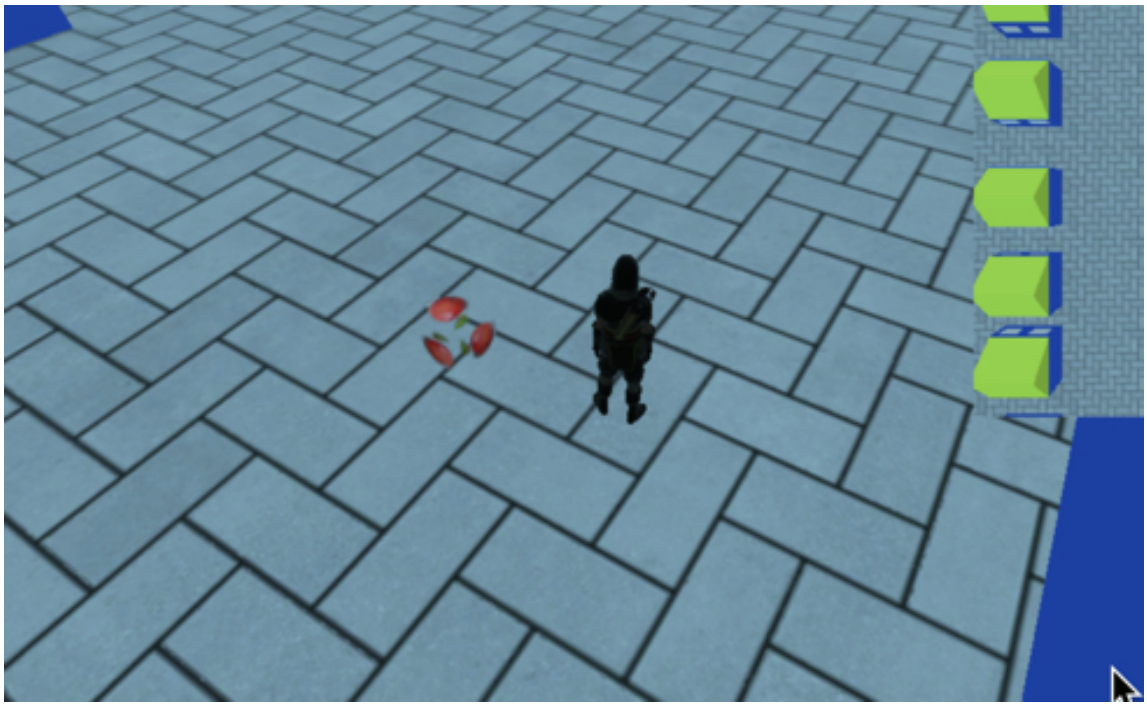
- Type the text **object_to_collect.tscn** in the text field and click on **Open**.



- This will create a new node called **object_to_collect**.
- Please move this object above the ground and a few meters away from the player character, as per the next figure.



- Play the scene and you should see that the apple is now displayed in the scene as an object to collect.



Next, we will need to detect collisions between the player and the objects to be collected.

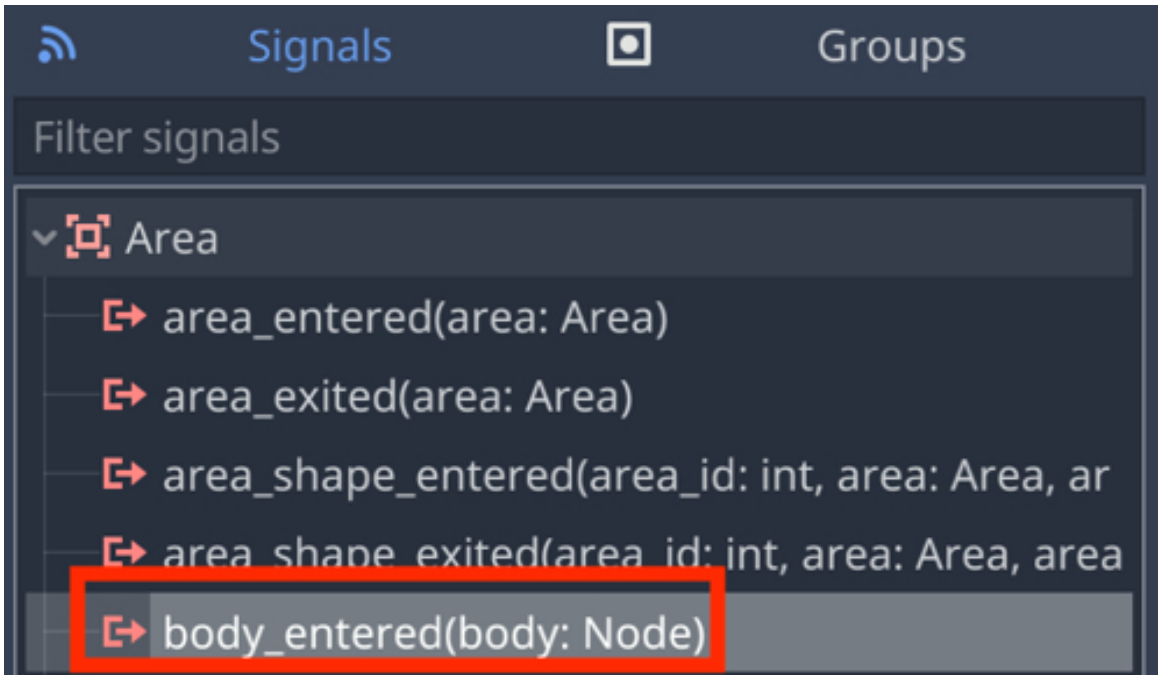
- Please open the script **object_to_be_collected**.
- Add this function to the script.

```
func body_entered(body):
pick_up_object1()
```

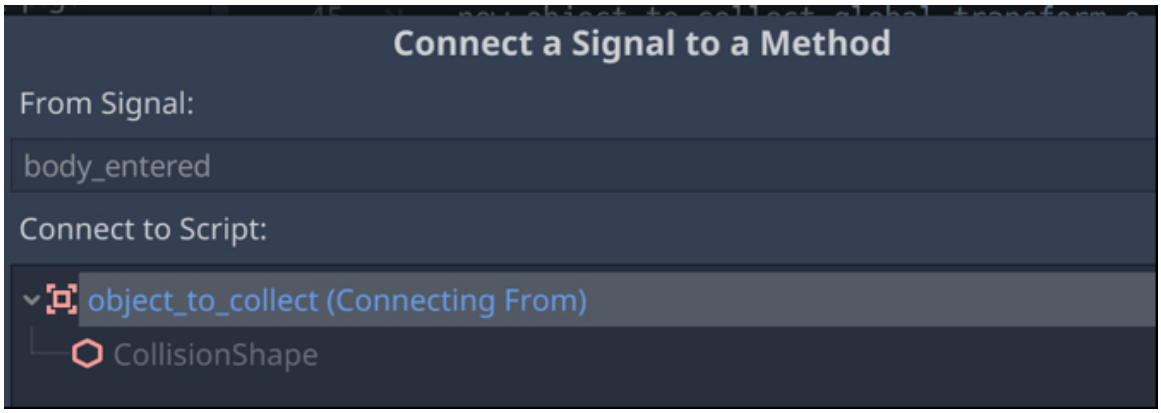
In the previous code, we create a function called **body_entered** that will call the function **pick_up_object1**.

Now we just need to make sure that this function is called whenever the player enters the area defined for the object to be collected.

- Please open the scene **object_to_collect**.
- Select the node **object_to_collect**.
- Using the **Inspector**, select the tab **Node | Signals**.
- Double-click on the event **body_entered**.



- In the new window, make sure that the node **object_to_collect** is selected.



- In the section **Receiver Method**, please type the text **body_entered** and click on "**Connect**".
- So effectively, the function **body_entered** will be called whenever the area defined for the current object has been entered by the player.

We now need to define the method **pick_up_object1**.

- Please add this function to the script **object_to_be_collected**

```
func pick_up_object1():
get_node("../Player").item_to_pickup_nearby = true
var can_pick_up = get_node("../Player/in-ventory_system").update_in-tory(type,1)
if (can_pick_up):
get_node("../Player").item_to_pickup_nearby = false
queue_free()
```

In the previous code:

- We create a function called **pick_up_object1**.
- We notify the player that there is an item nearby to be collected.
- We add the new object to the inventory (if that' possible) using the function **update_inventory**.
- Then when this has been done, we notify the player that the object has been collected, and the latter is then destroyed.

Before we can test the scene, we just need to add a few more things:

- Please add this code at the beginning of the script **manage_player**.

```
var item_to_pickup_nearby
var item_to_pickup
```

- Finally, please add this function to the script **InventorySystem**.

```
func update_inventory(type_of_item:int, nb_items_to_add:int):
var found_similar_item = false
for i in range (0, player_inventory.size()):
if (player_inventory[i].type == type_of_item):
if (player_inventory[i].nb + nb_items_to_add <= player_inventory[i].max_nb):
player_inventory[i].nb += nb_items_to_add;
found_similar_item = true;
break
else: return false
if (!found_similar_item):
var new_item = Item.new(Item.item_type.GOLD)
player_inventory.push_back(Item.new(type_of_item))
player_inventory[player_inventory.size() - 1].nb = nb_items_to_add;
return true;
```

In the previous code:

- We go through the player's inventory.
- If an item of the same type is already in the inventory, we then check that we can add this item without going over the limit set for this type of item.
- If that is the case, then the item is added to the inventory which is updated.
- Otherwise, if a similar item was not found in the inventory, we add it to the inventory.

Please save your code and test the scene, and you should be able to pick up the

item as you collide with it; it should also be added to your inventory (you can press **I** to see the inventory).

Allowing the player to choose the objects to collect

As it is, the system works properly and it is possible to collect an item and add it to the inventory.

Now it would be good to make it possible for the player to accept or refuse to collect an item. So, in the next sections, we will create a menu that will appear when the player is near an object to collect and that will provide the options to collect the item if the player wishes to do so.

- Please open the main scene (i.e., **level1**) and add a new node of the type **Label** as a child of the node **Spatial** and rename this node **user_message**.
- Change its position (i.e., **Rect | Position**) to **(425, 190)** and its size (i.e., **Rect | Size**) to **(400, 100)**.
- Add this code at the beginning of the script **manage_player**.

```
onready var user_message = get_node("../user_message")
```

- Add this code to the function **_ready** in the script **manage_player**:

```
user_message.hide()
```

- Modify the code in the function **body_entered** in the script **object_to_be_collected** as follows (new code in bold).

```
func body_entered(body):
#pick_up_object1()
get_node("../Player").item_to_pickup_nearby = true
pick_up_object2()
```

In the previous code, we simply comment the call to the method **pick_up_object1** and we add a call to a method named **pick_up_object2**.

- Please add the following function to the script **object_to_be_collected**.

```
func pick_up_object2():
var article;
```

```
if (is_a_vowel(item.name [0])): article ="an"
else: article = "a"
var message = "You Just Found " + article + " "+item.name +"\n\n Collect y/n"
get_node("../user_message").show()
get_node("../user_message").text = message
```

In the previous code, we define the article to be used for the item to be collected and this is done through the function **is_a_vowel** (that we yet have to define); for example, we will use "**an**" if it is an apple, or "**some**" if it is gold; we then create a sentence that will be displayed and that offers the player the choice to pick up or to leave this item.

- Please add the following function to the same script:

```
func is_a_vowel(the_letter) :
the_letter = the_letter.to_lower()
var is_a_vowel = false
for i in "aeiou":
if (the_letter == i):
is_a_vowel = true
return (is_a_vowel)
```

In the previous code:

- We create a function called **is_a_vowel**.
- This function takes a letter as a parameter and checks whether this letter is a vowel.
- The function returns **true** (i.e., that the letter passed as a parameter is a vowel) if this is the case, and **false** otherwise.

Once the message has been displayed, we just need to check the decision made by the player.

- Please add the following code to the script **object_to_be_collected**.

```
func _process(delta):
if (get_node("../Player").item_to_pickup_nearby):
if (Input.is_key_pressed(KEY_Y)):
pick_up_object1()
elif (Input.is_key_pressed(KEY_N)):
get_node("../user_message").hide()
```

In the previous code:

- We check that there is an actual object nearby to be collected.
- If that's the case, we check whether the player has pressed the key **Y** (for Yes) or **N** (for No).
- If the player has indicated that s/he wants to collect this item, we call the function **pick_up_object1**.
- Otherwise, we just clear the screen from the message.

We now need to check whether the player has walked far away from the object to be collected.

- Please add the following function to the script **object_to_be_collected**:

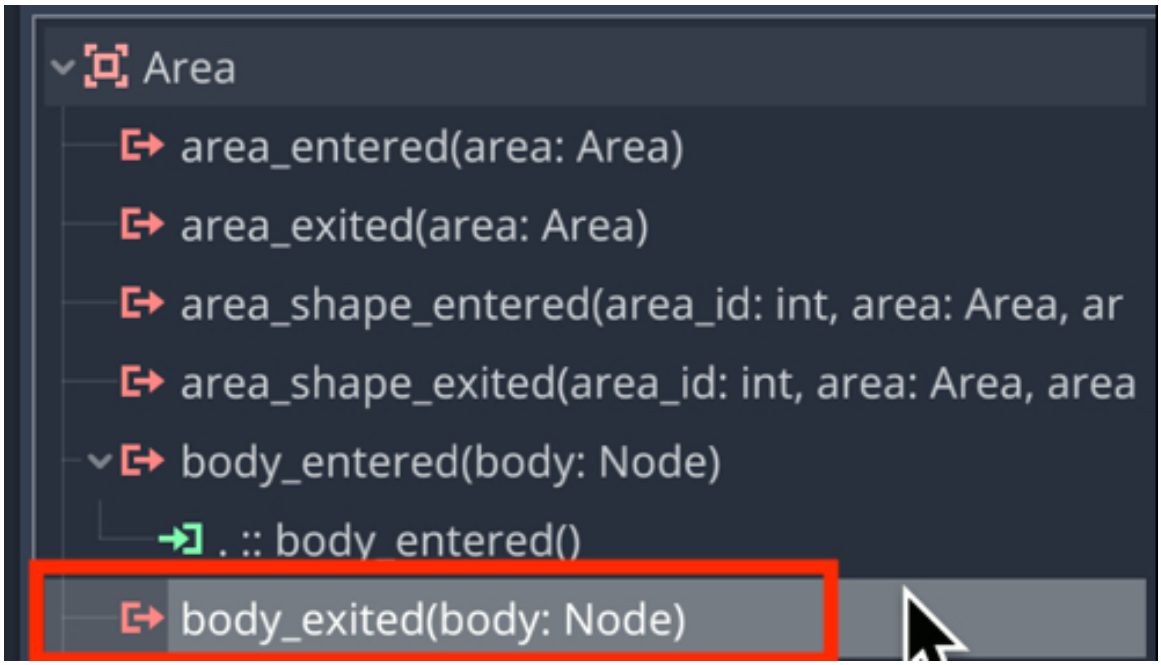
```
func body_exited(body):
get_node("../Player").item_to_pickup_nearby = false
if (get_node("../user_message").is_visible()):
get_node("../user_message").text = ""
get_node("../user_message").hide()
```

In the previous code, we just clear the previous message from the interface once the player has exited the area that was originally defined for the object to collect. In other words, once the player is far away from the object, we don't provide him/her the option to pick the object up, unless s/he goes back to this object.

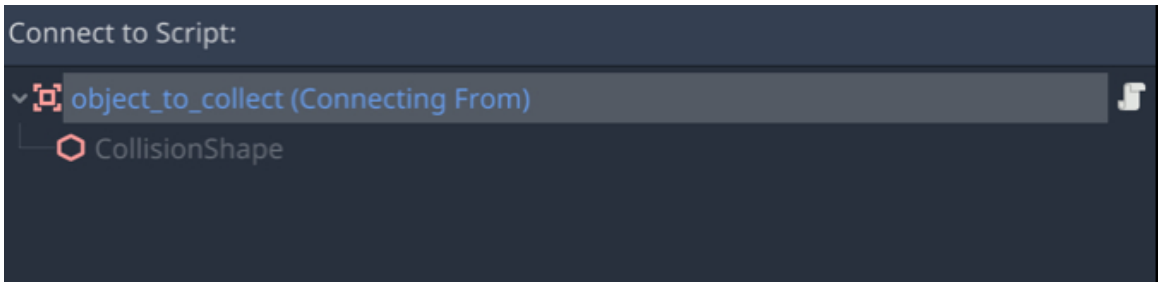
Once this has been done, we just need to link the event triggered when the player exits this area to the function that we have just created as follows:

- Please open the scene **object_to_collect.tscn**.

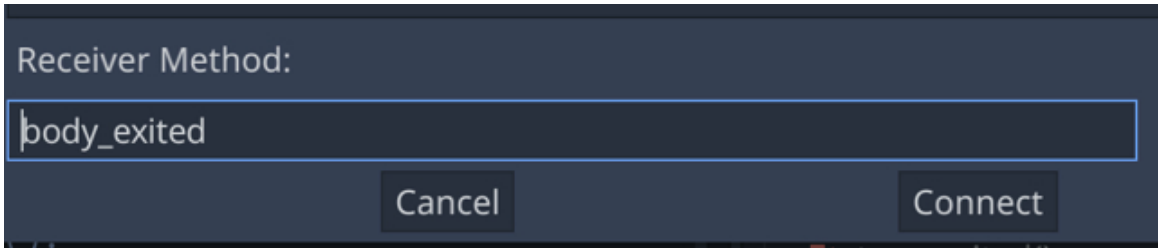
- Select the node **object_to_collect**.
- Using the **Inspector**, select the tab **Node | Signals**.
- Double-click on the event **body_exited**.



- In the new window, please make sure that the node **object_to_collect** is selected.



- Type **body_exited** in the field **Receiver Method** and then press **Connect**.



Last but not least, we will need to check that it is possible to add an item to the inventory before we can pick it up:

- Please modify the method **pick_up_object1** (new code in bold) in the script **object_to_be_collected**.

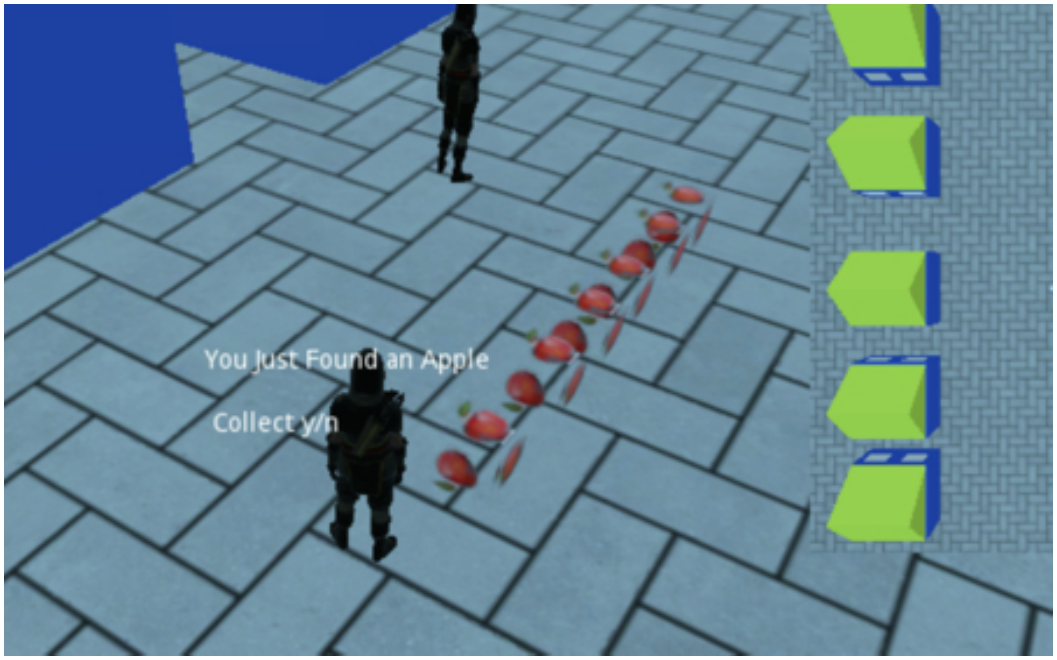
```
if (can_pick_up):
get_node("../Player").item_to_pickup_nearby = false
get_node("../user_message").hide()
get_node("../user_message").text = ""
queue_free()
else:
var message = "Sorry, you can't pick up this item"
get_node("../user_message").text = message
```

In the previous code: if it is possible to add the object to the inventory, we empty the message previously displayed to the user and we also deactivate the panel that is used to display the message; otherwise, we display a message that indicates that the player can't collect this object.

- You can now save your code.
- You can now reactivate the objects **dialogue_panel** and **inventory** in the scene **level1**.

To perform the test, please play the scene, go towards the apple, collect it (i.e., press the key **Y** after the message asking you whether you want to pick up this object is displayed), and then check your inventory that should now include **meat**, **gold**, and the **apple** that you have just picked up.

To complete the test, you could copy the object **object_to_collect** five times, move the duplicates apart, and try to collect all of them, as per the next figure.



As you try to collect the last one, the following message should be displayed:

" You can't pick up this item"

This is because we have specified in the class **Item** that the maximum number of apples to be carried is **5**, as per the next code snippet.

```
match (the_new_type):
item_type.APPLE:
```

name = "Apple";

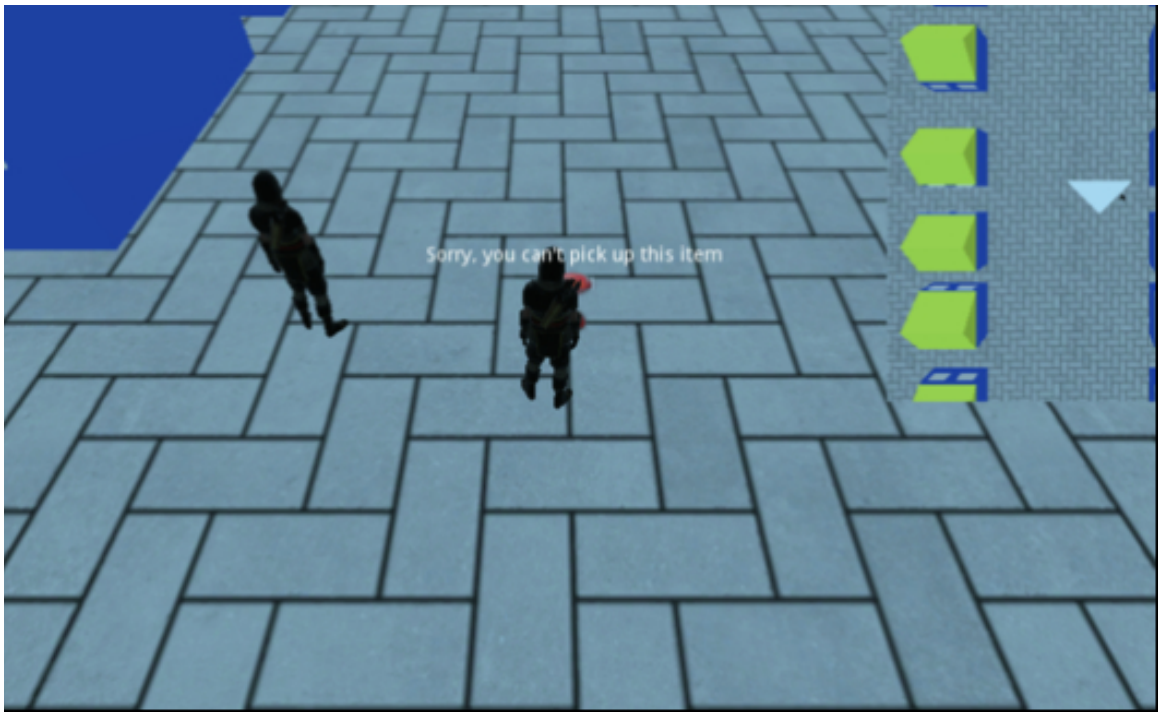
price = 50;

health_benefits = 10;

dammage = 0;

nb = 1;

max_nb = 5;



You can delete the extra objects to collect once you are happy with your tests.

Using an item from the current inventory to increase the players' health

At this stage, the inventory works well, and you can pick up objects up to a maximum set for the type of the item selected; if you remember well, we have, in the class called **Item**, specified attributes such as **health_benefit** for each item; the idea is that for example, you should be able to increase the player's health by using (i.e., eating) some of the items in the inventory. So in this section, we will add the possibility for the player to use the different types of food present in the inventory to increase its health.

For this purpose, we will:

- Create a variable for the player's health.
- Set this variable.
- Increase the health of the player whenever s/he eats food according to the health benefits associated with this food.
- Display the health levels on screen.

First, let's define the health of the player.

- Please open the script **manage_player**.
- Add the following code at the beginning of that script.

```
export(int, 1,100) var health
```

In the previous code, we define the variable called **health**; it will be accessible through the **Inspector** and its range will vary from **1** to **100**.

We will now add a method to increase the player's health:

- Please add the following function to the script **manage_player**:

```
func increase_health(amount:int):
health += amount
if (health >= 100): health = 100
print("Health="+str(health))
```

In the previous code:

- We create a new function called **increase_health.**
- In this function, we increase the **health**.
- We also cap this number to **100**.
- We then print the value of the variable **health** in the **Console** window.

Finally, please open the script **InventorySystem** and add the following code in the function **_process** (new code in bold).

```
if (Input.is_action_just_pressed("inventory")):
current_inventory_index +=1
if (current_inventory_index >= player_inventory.size()):
current_inventory_index = 0;
is_visible = false
display_ui(false)
if (Input.is_action_just_pressed("use_inventory")):
if (player_inventory[current_inventory_index].family_type == Item.item_family_type.FOOD):
get_node("../../Player").increase_health(player_inventory[current_inventory_index].health_benefits);
player_inventory.remove(current_inventory_index);
current_inventory_index = 0;
is_visible = false
display_ui(false);
```

In the previous code:

- We check whether the player has pressed the key **U**.
- If that's the case, we check that this item is from the group called **FOOD**.
- If the item selected is part of the **FOOD** group, we increase the health of the player; the increase is equal to the health benefit provided by this item and previously defined in the file **Item**.
- Once the food has been used (i.e., eaten), we remove it from the inventory.

- We set the current index of the inventory to **0**, and we then hide the interface for the inventory.

You can now save your code and test it as follows.

- Set the **health** variable to **50** using the **Inspector** for the object **Player**.
- Play the scene.
- Display your inventory.
- Select the meat.
- Press the key **U** to use (i.e., eat) the meat.
- You should see the following message in the **Console** window.

Health:80

- Pick up and use the **apple**, and the following message should be displayed as follows.

Health:90

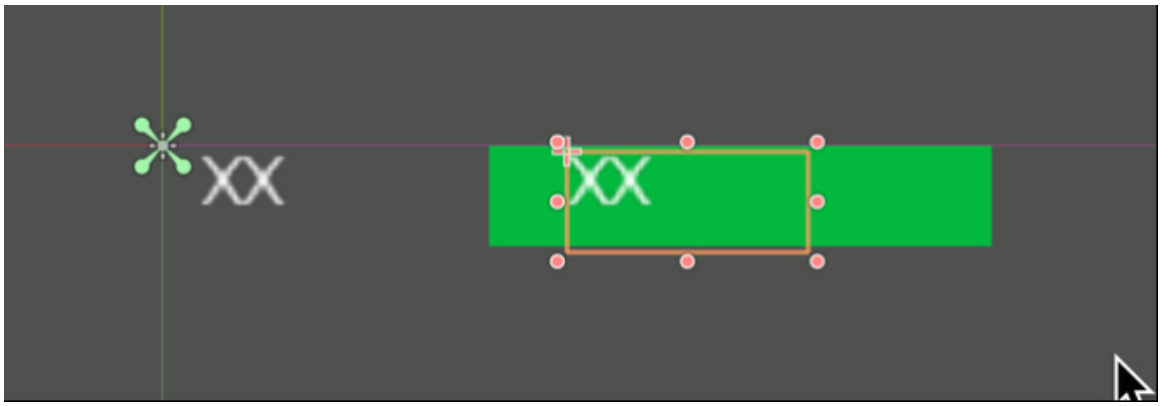
You should also see that both the apple and the meat have disappeared from your inventory after they have been used.

Creating a health bar

At this stage, we can see, in the **Console** window, that the player's health is increasing whenever s/he uses food from the inventory. The next step is to display the player's health on screen, with a health bar, rather than in the **Console** window. This health bar will consist of three parts: a black border, a black background, and a green color within, as per the next figure.

So, let's start creating this health bar:

- Please create a new node of the type **Node2D** as a child of the node **Spatial** in the main scene (i.e., **level1**) and rename this new node **health_bar**.
- Add a node of the type **ColorRect** as a child of the node **health_bar** and rename this new node **black_bg**.
- Select the node **black_bg**.
- Set its color to **black** using the section **ColorRect | Color** in the **Inspector**.
- Using the **Inspector**, expand the section **Rect**, and change the attributes **Position** to **(64.5, -0.2)** and **Size** to **(100, 20)**.
- Duplicate the node **black_bg**, and rename the duplicate **green_bar**, and change its color to **green** using the section **ColorRect | Color** in the **Inspector**.
- Create a new node of the type **Label** as a child of the node **health_bar**, rename this new node **left_label**, and, using the **Rect** section in the **Inspector**, change its position to **(7.5, 1.3)** and its size to **(45, 20)**. Set the **horizontal** and **vertical alignment** attributes to **left** and **top**, respectively and type **XX** in the **Text** attribute, so that you can see how the text will look.
- Duplicate the node **left_label**, rename the duplicate **value_label**, and using the **Rect** section in the **Inspector**, change its **position** to **(80.5, 1.3)** and its **size** to **(48, 20)**.



The next step will be for us to control the appearance of the health bar so that its length reflects the value of the variable **health** that we have defined for the player earlier.

- Please attach a new script to the node **health_bar** and save it as **manage_bar**.
- Add this code at the beginning of the script.

```
var value = 0
var label = "Health"
```

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

```
func _ready():
get_node("left_label").text = label
update_value()
```

- Add the following function to the script **manage_bar**.

```
func increase_value(amount:int):
value += amount
if (value >= 100): value = 100
update_value()
```

In the previous code, we increase the value represented by the bar and call the method **update_value**.

- Please add the following function.

```
func update_value():
get_node("value_label").text = str(value)
var scale_factor:float = value/100.0
get_node("green_bar").rect_scale = Vector2(scale_factor,1)
```

In the previous code, we rescale the node **green_bar** based on the value of the variable **value** and we also update the text displayed within the bar.

- Please add the following function:

```
func _process(delta):
if (Input.is_key_pressed(KEY_B)):increase_value(10)
```

In the previous code, that is used for testing purposes, we increase the health by 10 if the user presses the key **B**.

Please save your code. Once this is done, you can check that the health bar works by playing the scene, and pressing twice the letter **B** on your keyboard; you should see that the health increases, as illustrated on the next screenshot.

Once you have checked that this is working, you can comment out the code in the **_process** function in the class **manage_bar** as follows.

```
func _process(delta):
#if (Input.is_key_pressed(KEY_B)):increase_value(10)
pass
```

In the next section, we will link the health bar that we have just created to the inventory and the player so that the health is displayed at all times, and also so that any change in the player's health (e.g., after s/he eats food) is mirrored in the health bar.

- Please add this function to the script **manage_bar**.

```
func set_value(amount):
value = amount
update_value()
```

- Modify the function **increase_health** in the file **manage_player** as follows (new code in bold).

```
func increase_health(amount:int):
health += amount
if (health >= 100): health = 100
print("Health="+str(health))
get_node("../health_bar").set_value(health)
```

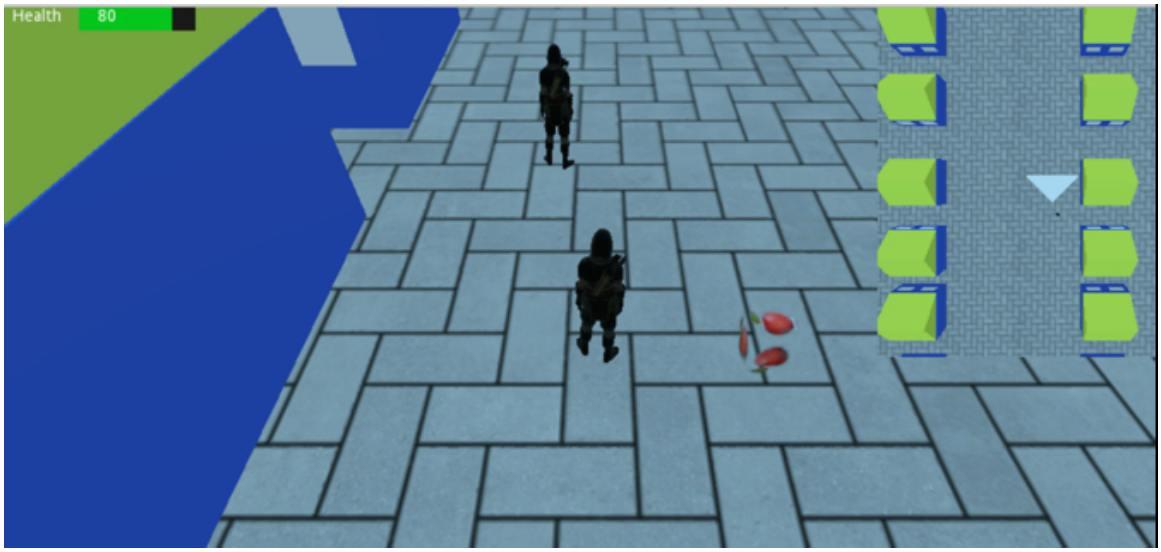
In the previous code, we just set the value of the **health** bar with the value of the variable **health**.

- Add this code to the **_ready** method in the script **manage_player**.

```
get_node("../health_bar").set_value(health)
```

Once this is done, you can test your code:

- Save your code.
- Play the scene.
- At the start your health should be **50%**.
- As you use the meat from the inventory, the health should increase to **80**.
- And then to **90** if you pick up and use the apple present in the scene.



You can also create a new scene with the object called **health_bar** (i.e.,

right-click on that node and select the option **Save Brach as a Scene**). This will be useful if you want to use a progress bar or a status bar for other elements of your game; as you may have noticed, the health bar was created so that it is independent from other parts of your code; in other words, it can be re-used to display the status of any other variable.

LEVEL ROUNDUP

Summary

In this chapter, we have created an inventory system whereby the player can collect objects and use them to increase its health if need be. Along the way, we have used a combination of **Lists**, **Triggers**, **UI** elements, and key detection to be able to list all the items owned by the player and to also be able to detect when a new object close to the player character can be collected.

So in the end, we built a system with the following characteristics:

- You (the designer) can easily create items and their associated properties by modifying a script.
- You can add these items to the scene and change their type, if need be, using the **Inspector** and the generic object **object_to_collect**.
- You can specify an image and a 3D representation for each of these items, and store them.
- The player will be able to pick up an item and use it to increase its health.
- You (the designer) can also reuse the health bar prefab to display other attributes, if you wish.

This system is configurable, which means that you can change the attributes of any of the items that are to be used in your game, in terms of appearance, health benefits, etc., very easily; this will make your RPG creation and testing much faster.

Quiz

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

1. In Godot it is possible to access files present in the folder **res://** from a script.
2. The following code will add an object of the type item to the array **player_inventory**.

```
var new_item = Item.new(Item.item_type.GOLD)
player_inventory.push_back(new_item)
```

1. It is impossible to add or remove elements from an array.
2. The following code will check whether the user pressed the key mapped to the action "**inventory**".

```
if (Input.is_action_just_pressed("inventory")):
```

1. The following code will return the size of the array **player_inventory**.

```
var size = player_inventory.size()
```

1. It is possible to change the type of a node in the **Hierarchy Tree** using the option "**Change Type**".
2. It is not possible to change the color of a node of the type **ColorRect**.
3. The following code will create a placeholder in the **Inspector** so that a node can be dragged and dropped to it.

```
export (PackedScene) var baton
```

1. The following code will create a drop-down list in the **Inspector** from which the user can select a value/type.

```
export(int, "APPLE", "MEAT","GOLD","RED_DIAMOND", "BLUE_DIAMOND", "YELLOW_DIAMOND","SWORD","BATON") var type
```

1. It is possible to instantiate a scene that has been previously created, using the option "**Instance Child Scene**".

Solutions to the Quiz

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

Checklist

You can move to the next chapter if you can do the following:

- Understand how to create an array
- Understand how to update an array.
- Create a panel using the nodes of the type **ColorRect**.
- Display or hide UI elements.

Challenge 1

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

- Create two other objects to be collected and added to the inventory: oranges and tomatoes.
- Create spheres that can be used to represent these items (i.e., orange and red spheres).
- Find and use images that can be used to represent these items.
- Modify the file called **Item** to create corresponding **enum** types and also modify the function called **init**.
- Add some apples and tomatoes to the scene and ensure that the player can add them to their inventory.

Chapter 5: Creating a Shop and a Buying System

This chapter helps you to create a shop system that you can use to implement any type of shop in your game; this, as for most of the material covered in this book, will allow you to create a flexible system that you can reuse in all of your games without having to re-invent the wheel every time.

After completing this section, you should be able to:

- Create a shopping system whereby players can buy items and add them to their inventory.
- Create a user interface that the player can use to add items to their shopping basket, calculate the corresponding total, and check that these can be added based on the amount of money that the player currently has.
- Create a reusable interface with images, buttons, and the associated logic to process the user's actions.

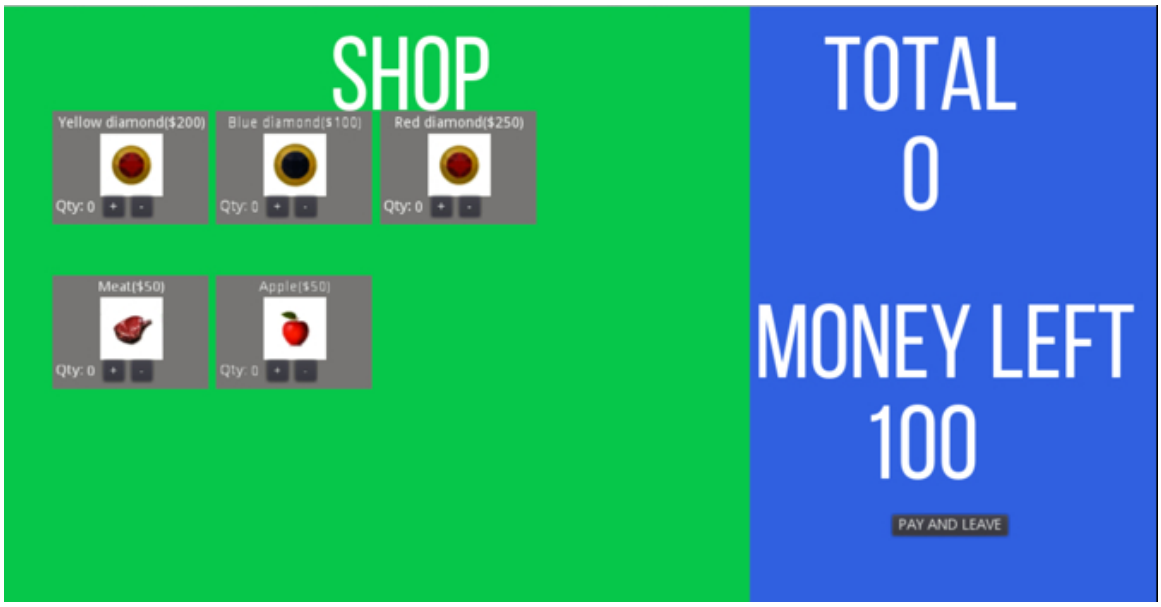
Creating a shop

In this and the next sections, we will start creating a shop whereby the player can enter a shop, buy items based on the number of gold coins that s/he owns, and add these items to his/her inventory after paying. The process will be as follows; we will:

- Create a user interface for the shop.
- Create a template to be used for each item in the shop.
- Add the ability to add or remove items to and from the shopping bag, accounting for the money that the player has and the total price.
- Add the items to the player's inventory once the purchase is complete.

Creating the interface

Before we start, let's look at the layout of the shop that we are going to create.

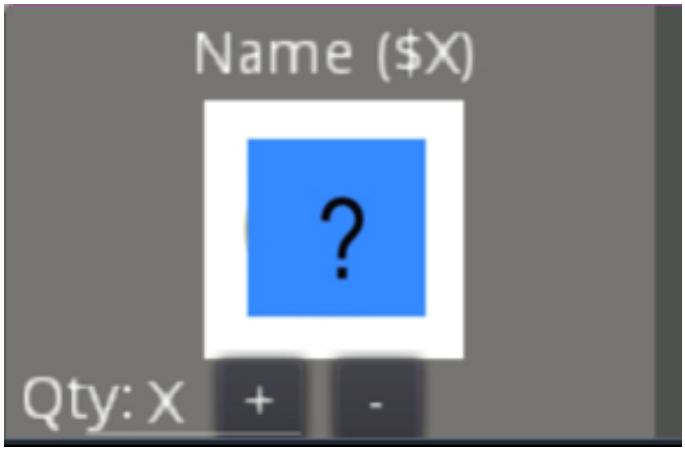


As you can see, the interface will consist of:

- A left panel where the items available in the shop are listed.
- A right panel with the total (i.e., updated after items have been selected), the amount of money left for the player, and a button to pay for the items selected and leave the shop.
- For each item available, there is information that consists of an image, a label, a price, a quantity, and a plus and minus button to add or remove this specific item to/from our shopping bag.
- You may notice that the layout of the information provided for each item is the same, as we will create a template that will be used for every type of item available.

Creating the user interface for each item available in the shop

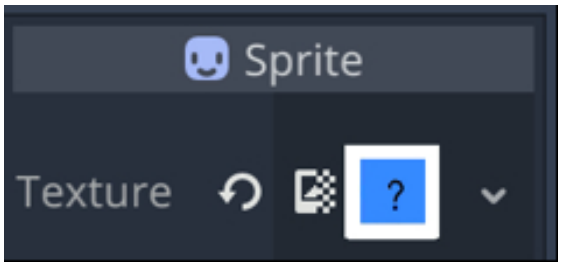
First, we will create the UI for each of the items available in the shop. This UI will look like the next figure.



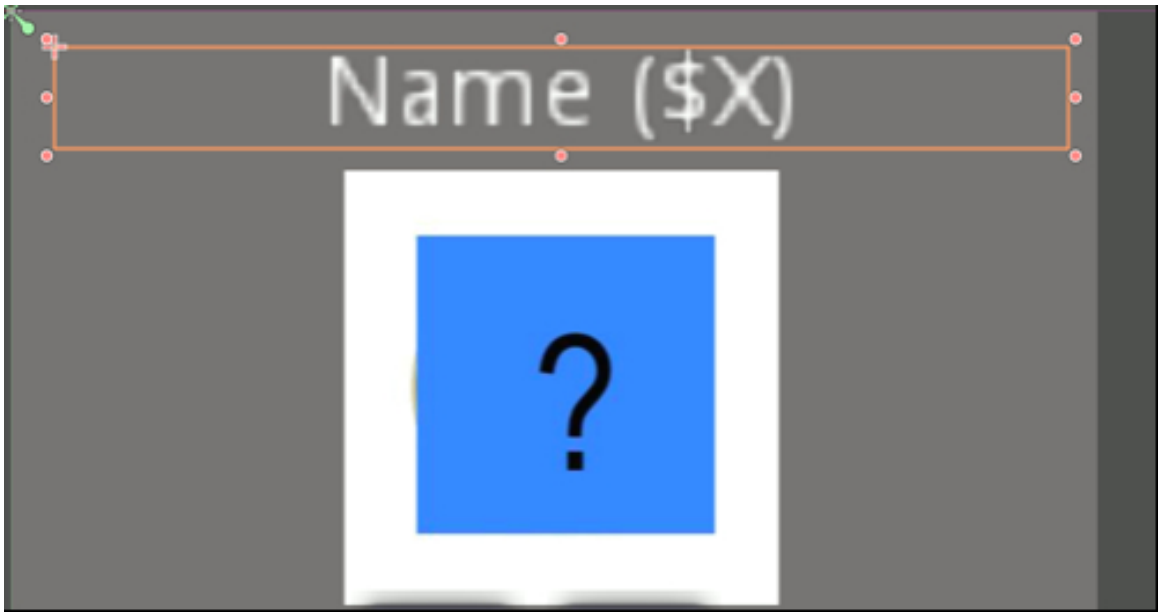
We will then create a scene from this UI and re-use it for any other items available in the shop.

Let's proceed:

- Please create a new **Node2D** scene.
- This will create a new scene with a default node called **Node2D**.
- Please rename this node **shop_item**.
- Add a new node of the type **ColorRect** as a child of the node **shop_item** and rename this node **item_bg**.
- Change its color to **grey** and, using the section **Rect**, change its position to **(0, 0)** and its scale to **(150, 109)**.
- Add a new node of the type **Sprite** as a child of the node **shop_item**, and rename the new node **item_image**.
- Import the texture **empty_shop_item.jpg** from the older **images** in the resource pack to your project.
- Using the section **Sprite | Texture** in the **Inspector**, drag and drop the file **empty_shop_item.jpg** to the right of the label **Texture**.



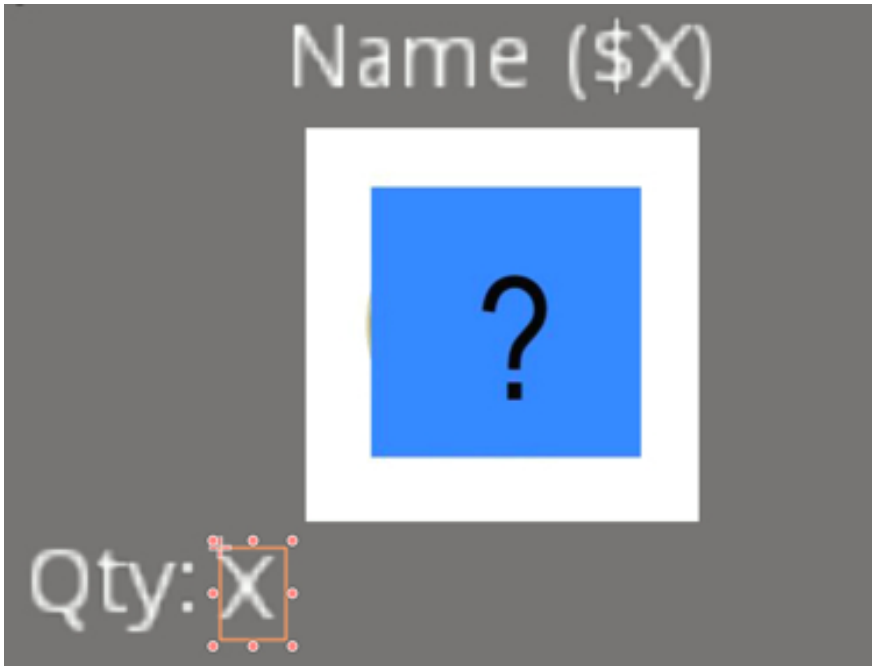
- Change the position (i.e., **Transform | Position**) of the node **item_image** to **(76, 52)** and its scale (i.e., **Transform | Scale**) to **(0.2, 0.2)**.
- Add a new node of the type **Label** as a child of the node **shop_item** and rename it **item_label**.
- Change its text attribute to **Name ($X)**, change its alignment to **Center/Center**, and, using the **Rect** section in the **Inspector**, change its **position** to **(6, 5)** and its **scale** to **(140, 14).**



- Add a new node of the type **Label** as a child of the node **shop_item** and rename it **item_qty_label**.
- Change its text attribute to **Qty:30**, its alignment to **Left/Center**, and, using the **Rect** section in the **Inspector**, change its **Position** to **(3, 85)** and its **Size** to **(30, 14).**
- Finally, add a new node of the type **Label** as a child of the node **shop_item**

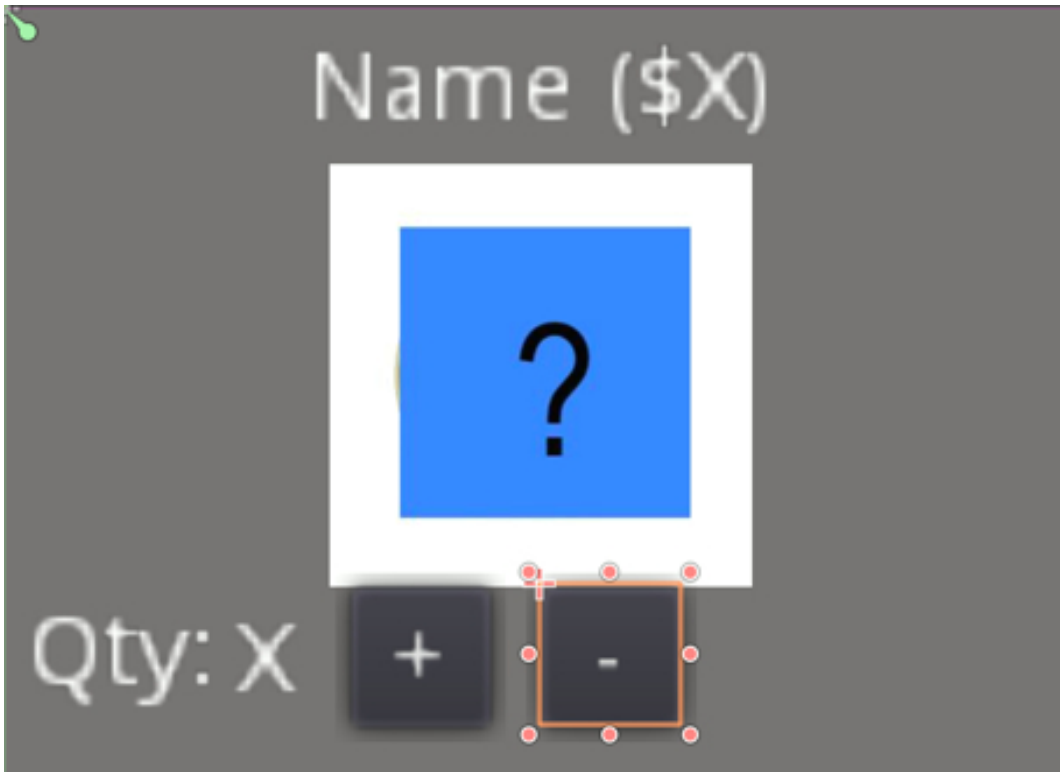
and rename it **item_qty**.

- Change its **Text** attribute to **X**, change its alignment to **Left/Center**, and, using the **Rect** section in the **Inspector**, change its **position** to **(33, 86)** and its **size** to **(10, 14).**



Once this is done, we can start to create the buttons that will be used to increase or decrease the quantity for the current item.

- Please add a new node of the type **Button** as a child of the node **shop_item** and rename it **plus_button**.
- Change its text attribute to **+**, and, using the **Rect** section in the **Inspector**, change its **Position** to **(48.8, 81.5)** and its **Size** to **(20, 20).**
- Duplicate the node **plus_button**, rename the duplicate **minus_button**, change its **Text** attribute to **-** and change its position to **(75.8, 81.5)**.



At this stage, the interface for the item is complete, and we just need to add a corresponding image that will replace the white square. We also need to create the corresponding logic so that pressing on the + or – button increases or decreases the quantity for this item accordingly.

- Please save this scene as **shop_item.tscn**.

Creating scripts to add or remove items

In this section, we will create the scripts that will be linked to the + and – buttons so that we can increase or decrease the quantity for a specific item.

- Please open the scene **shop_item.tscn**.
- Select the node **shop_item**.
- Attach a new script to it, and rename the script **shop_item.gd**.
- Open the script.
- Add this code at the beginning of the script.

```
var item_name
var item_price
var item_quantity = 0
var item_index = 0
```

- Add this code to the function **_ready**.

```
update_quantity_label()
```

- Add these new functions:

```
func increase_quantity():
if (!can_click()): return
item_quantity += 1
update_quantity_label()
func decrease_quantity():
item_quantity-=1
if (item_quantity < 0): item_quantity = 0
update_quantity_label()
func update_quantity_label():
get_node("item_qty").text = str(item_quantity)
func can_click()->bool:
return true
```

In the previous code:

- The function **increase_quantity** increases the quantity of the current item if the method **can_click** returns true; it then calls the method **update_quantity_label** that will update the text for the node **item_qty**.
- The function **decrease_quantity** decreases the quantity of the current item if **can_click** returns true; it then calls the method **update_quantity_label** that will update the text for the node called **item_qty**.
- Finally, the function **update_quantity_label** looks for the node **item_qty** and updates its text attribute.
- For the time being the method **can_click** is a stub, as it always returns true; later on, we will get it to communicate with a shop system that will check for several attributes, including whether the player has enough funds to purchase this item.
- Please save your code, and check that it is error-free.

In the next section, we will manage the user interaction with the buttons by detect ing when s/he presses the **+** or **–** button and updating the interface accordingly.

- Please select the node **plus_button** (in the scene **shop_item**) and, using the **Inspector**, open the tab **Node | Signals**, double-click on the event called **pressed**, and link this event to the function called **increase_quantity** that is in the script linked to the node **shop_item**.
- In the same way as in the previous step, please select the node **minus_button** (in the scene **shop_item**), and using the **Inspector**, open the tab **Node | Signals**, double-click on the event called **pressed**, and link this event to the function called **decrease_quantity** that is in the script linked to the node **shop_item**.

So at this stage, both buttons have been linked to the **pressed** events; so pressing the buttons **plus_button** or **minus_button** will either increase or decrease the quantity of the current item, respectively.

At this stage, you can play the scene **shop_item**: as you use the **plus** and **minus** buttons, you should see

how the quantity increases or decreases for this item.

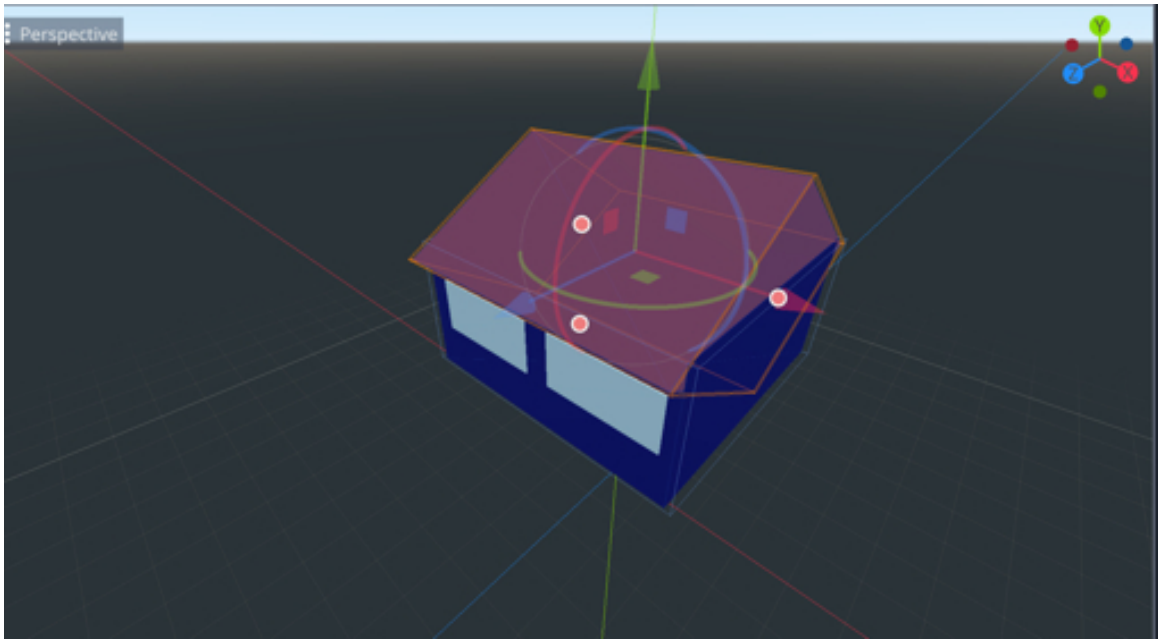
Adding items to the shop

In this section we will:

- Create an interface for the shop.
- Add items and the corresponding UIs elements to the shopping basket.
- Make it possible to calculate the total amount/cost for the shopping basket.

First, we will create a house that will be used for the shop:

- Please open (or switch to) the scene **level1**.
- Please duplicate the scene **house.tscn** and rename the duplicate **shop.tscn**.
- Open the scene **shop.tscn**.
- Rename the node **house** to **shop**.
- Select the node **roof** and change its color to **red**.



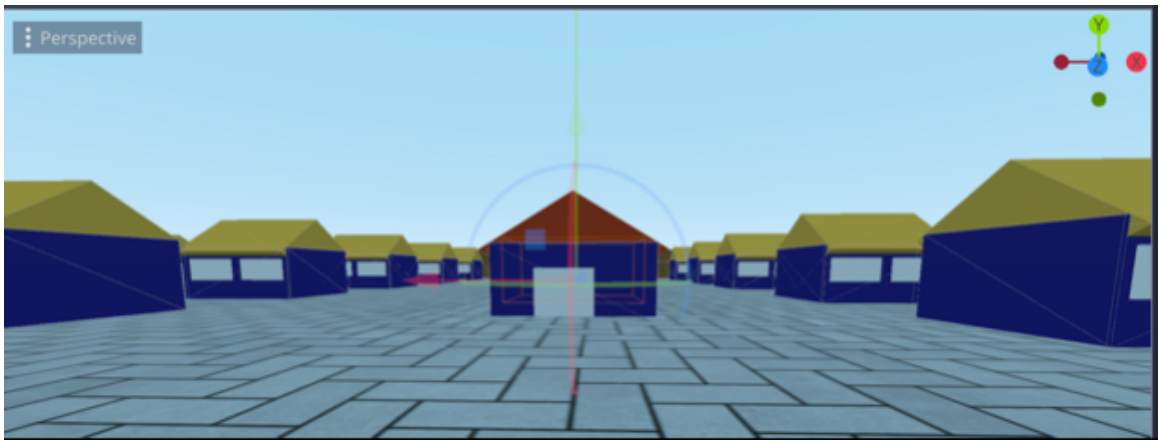
- You can save your scene.

We can now add this shop to the main scene:

- Please open (or switch to) the main scene **level1**.
- Create a new node of the type **Area** as a child of the node **Spatial**, and rename

this new node **shop**.

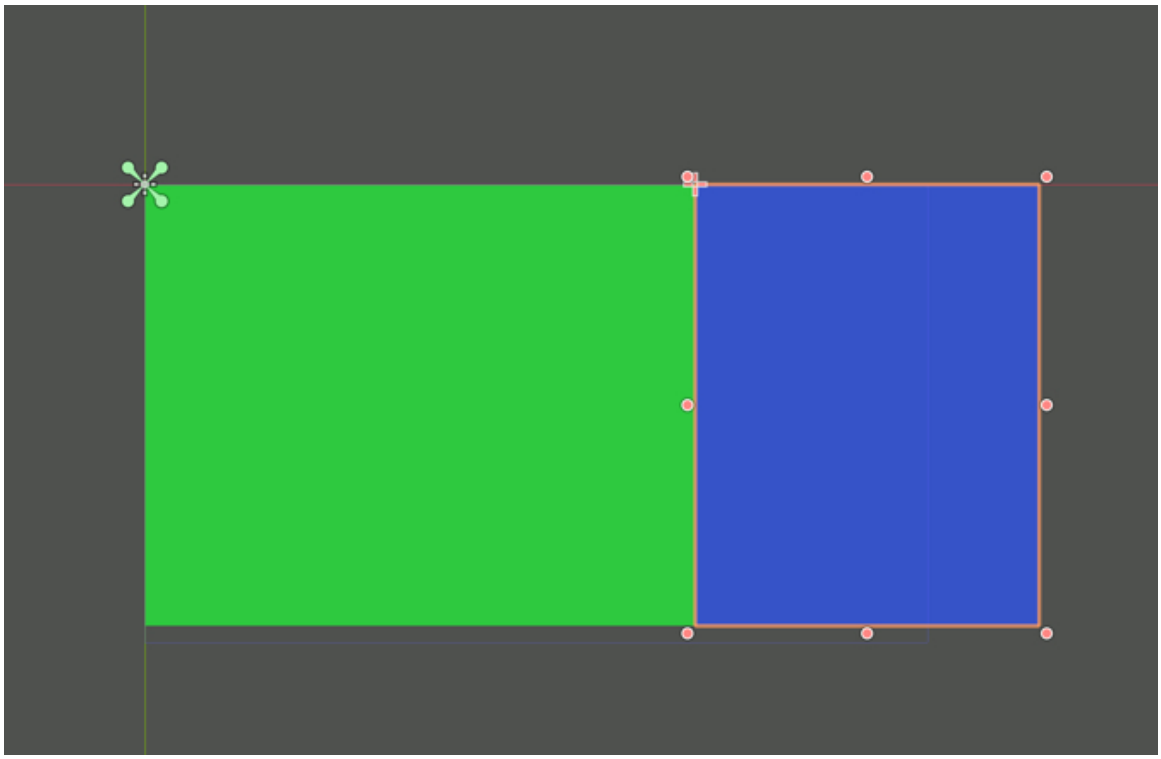
- Create a new node of the type **CollisionShape** as a child of the node **shop**, and set a **BoxShape** for this node: from the **Inspector**, click to the right of the section **CollisionShape | Shape**, and select the option **New Box Shape**.
- Create a new node of the type **Node** as a child of the node **shop**, and rename this new node **shop_system**.
- Right-click on the node **shop**, select the option **Instance Child Scene**, and select the scene **shop.tscn**.
- This will create a new node called **shop**: please rename it **shop_walls**.
- Change the rotation of this node to **(0, -90, 0), and** its **scale to (3, 3, 3)**.
- Finally, select the node **shop**, change its **y** coordinate to **2**, and move it along the **x** and **z** axis so that it appears in the middle of the village and so that the entrance is accessible.



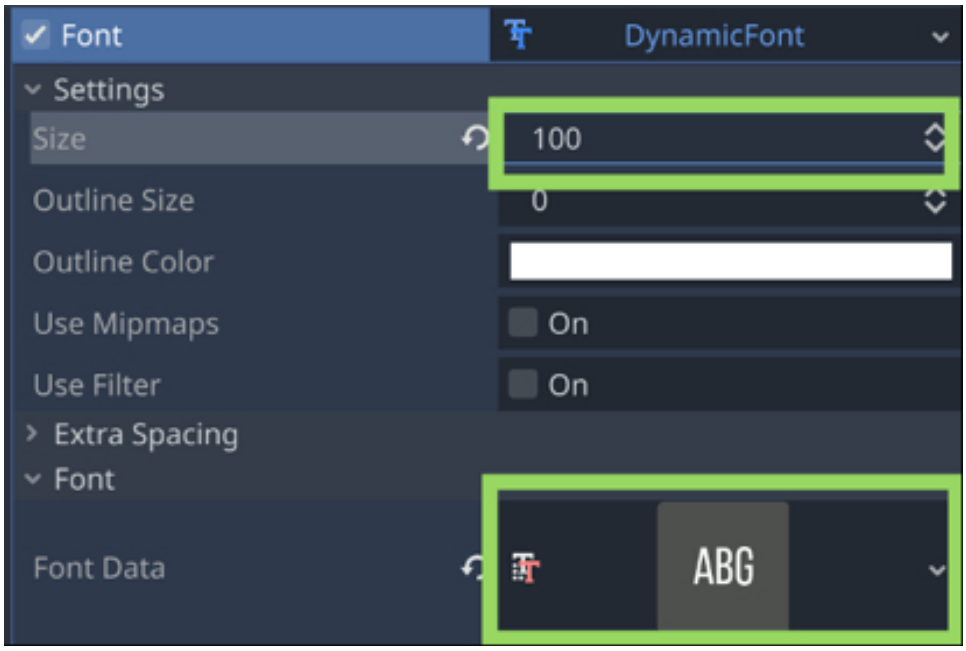
Now that we have created the 3-Dimensional aspect of the shop, and before we start to implement the logic to manage the shop and to display items, we will start to create the UI for the shop.

- Please open (or switch to) the main scene (i.e., **level1**), create a new node of the type **Node2D** as a child of the node **Spatial**, and rename it **shop_ui**.
- Create a new node of the type **ColorRect** as a child of the node **shop_ui**, rename it **shop_items**, set its color to **green**, its **position** to **(0, 0)**, and its **size** (using the section called **Rect**) to **(723, 578)**.
- Duplicate the node **shop_items**, rename the duplicate **bag_items**, set its color

to **blue**, its position to **(720, 0)**, and its size (using the section called **Rect**) to **(420, 578)**.



- Add a new node of the type **Label** as a child of the node **shop_ui,** rename it **shop_label** and change its text to **SHOP**; using the section **Custom Fonts**, create a new **Dynamic Font** (i.e., click to the right of the label **Font** and select the option **Dynamic Font**), set the font as **Bebas Neue** (i.e., import the font **Bebas Neue** from the folder **fonts** in the resource pack to Godot, and then drag and drop this font from the **FileSystem** tab in Godot to the section **Font | Font Data**), and set the **font-size** to **100**.



- Place this node at the top of the screen, as per the next figure (you can modify its **font-size** and **font-style** attributes as well, if you wish).

- Duplicate the node **shop_label**, rename the duplicate **shop_total_label**, change its text to **TOTAL**, and move it at the top of the **blue** panel.

- Copy the node **shop_total_label** three times and rename the duplicates **shop_total_value**, **shop_money_left_label**, and **shop_money_left_value,** change their text respectively to **XXXX**, **MONEY LEFT**, and **XXXX**, and arrange them as illustrated in the next figure.

Now that the interface is complete, we will start to add code so that several items can be added to the shop.

- Please select the node **shop_system**.
- Add a script to this node and rename the script **shop_system.gd**.
- Add this code at the beginning of the script.

```
var shop_items:Array;
export (PackedScene) var shop_item_component;
var shop_item_components:Array
var total_purchase = 0;
var initial_money;
var money_left;
var top_left_x;
var y_top_padding = 100;
```

In the previous code:

- We create a variable **shop_item_component** that will be accessible in the **Inspec tor** and we will be able to initialize it by dragging and dropping a scene to this empty slot in the **Inspector** (i.e., the **scene shop_item**).
- We also create an array **shop_item_components** that will be used to store all the items that will be available in the shop.
- The variable **total_purchase** will store the total amount of the purchase.
- The variable **initial_money** will be used to store how much money the payer has when entering the store.
- The variable **money_left** will track how much money the player has left after buying the items so that this money can be put back in its inventory.
- Finally, the variables **top_left_x** and and **y_top_padding** will be used to position the images used to represent each of the items available in the shop. This is so that all items are displayed properly on screen (i.e., so that they are visible).

Now that these variables have been created, we can initialize them:

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

```
initial_money = 100;
money_left = initial_money;
top_left_x = 50;
```

In the previous code, we set the initial money owned by the user to **100**, we specify that the player has not spent any money yet, and we also set the value for the variable **top_left_x**.

To set the values for the variable **top_left_x**, you can place a new node of the type **shop_item** (i.e., right-click on the **Spatial** node, select the option **Instance Child Scene**, and select the scene **shop_item**) in the top left corner of the scene and check its position; its **x** and **y** coordinates can be used for the variable **top_left_x**. Because each **shopItem** instance will be added as a child of **shopUI**, their coordinates will be calculated in relation to their parent; as a result, we need to know the position of the object **shopItem** relative to **shopUI**; therefore, before taking note of its position, we need to make it a child of **shopUI**. In my case, this value was **50**. If you can't see the object **shopItem**, please make sure that it is below the object **shopUI** in the **Hierarchy** window.

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

```
shop_items.push_back(Item.new(Item.item_type.YELLOW_DIAMOND))
shop_items.push_back(Item.new(Item.item_type.BLUE_DIAMOND))
shop_items.push_back(Item.new(Item.item_type.RED_DIAMOND))
shop_items.push_back(Item.new(Item.item_type.MEAT))
shop_items.push_back(Item.new(Item.item_type.APPLE))
```

In the previous code, we initialize the array called **shop_items**, and we add items to it. The function **push_back** adds each item at the back (i.e., the end) of the array.

- Please add this new function:

```
func init():
```

- Please add this code to the function **init**.

```
shop_item_components.push_back(shop_item_component.instance())
shop_item_components.push_back(shop_item_component.instance())
shop_item_components.push_back(shop_item_component.instance())
shop_item_components.push_back(shop_item_component.instance())
shop_item_components.push_back(shop_item_component.instance())
```

In the previous code, we create **5** instances of the class **shop_item_component** that will be use to represent the items available in the shop on screen.

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

```
get_node("../../shop_ui/shop_money_left_value").text = str(initial_money)
for i in range (0, shop_item_components.size()):
setup_shop_item_component(i);
```

In the previous code, we display the initial money available to the player, we initialize the items of the array called **shop_item_components** through the method **setup_shop_ component** that we need to create. The array **shop_item_components** includes a user interface for each item available in the shop.

In the next section, we will create the method **setup_shop_component**. This method will determine the position and content of the user interfaces for each items available in the shop.

- Please add the function **setup_shop_component:**

```
func setup_shop_item_component (index:int):
```

```
shop_items[index].nb = 0
shop_item_components[index].item_index = index
```

In the previous code, we create the new function **setup_shop_item_component**, we set the initial number of items to **0**, and we refresh the corresponding index for this item in the shopping system.

The items displayed on screen are part of the array **shop_item_components** and will effectively be instances of the scene **shop_item** and show information about each of the items available in the shop; so they will all have a label, a price, an image, along with two buttons **plus** and **minus**.

- Please add the following code to the same function (i.e., **setup_shop_item_component**):

```
var width = 150;
var border_around_each_item = 1.05;
shop_item_components[index].name = "shop_item_" + str(index) + shop_items[index].name + "($" + str(shop_items[index].price) + ")";
shop_item_components[index].find_node ("item_label").text = shop_items[index].name + "($" + str(shop_items[index].price) + ")";
get_node("../../shop_ui").add_child(shop_item_components[index])
shop_item_components[index].global_transform.origin = get_node("../../shop_ui").global_transform.origin
```

In the previous code:

- We set the **width** of the items to be displayed on screen to **150**.
- We set the space between each item to be displayed on screen to **1.05**.
- We then define the name of each item.
- We then select the item (which is a node) and its label that will consist of its name followed by its price.
- We then add the items that we have just configured as a child of the node **shop_ui**.
- Finally, we change the position of the item to be displayed.

Now that each item has been added, we will just specify their position based on their index and the border that should be added between them:

- Please add the following code to the one you have just typed:

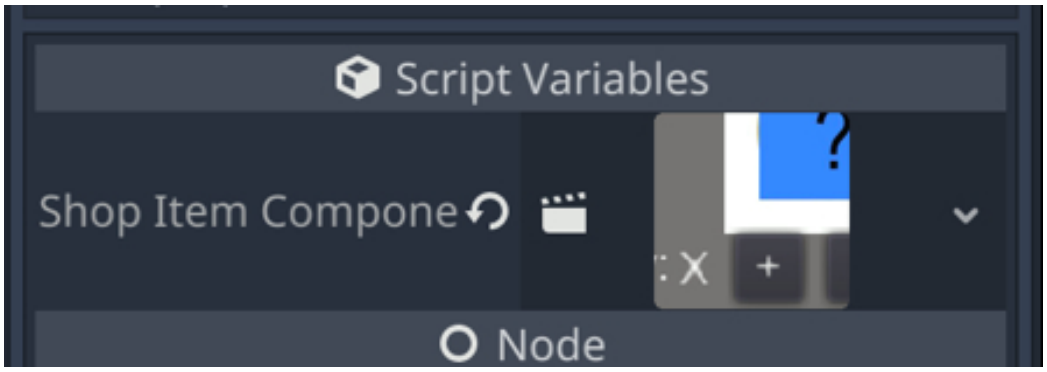
```
var add_on = shop_item_components[index].global_transform.origin
var x_pos = top_left_x + (index%3)* width*border_around_each_item
var y_pos = y_top_padding + (index/3) * width * border_around_each_item
shop_item_components[index].global_transform.origin = add_on + Vector2(x_pos, y_pos);
shop_item_components[index].find_node("item_image").texture = load(shop_items[index].get_texture_path())
```

In the previous code, we specify the position of each item, along with the path of the corresponding texture that should be displayed on screen.

- Finally, add this code to the function **_ready**.

```
init()
```

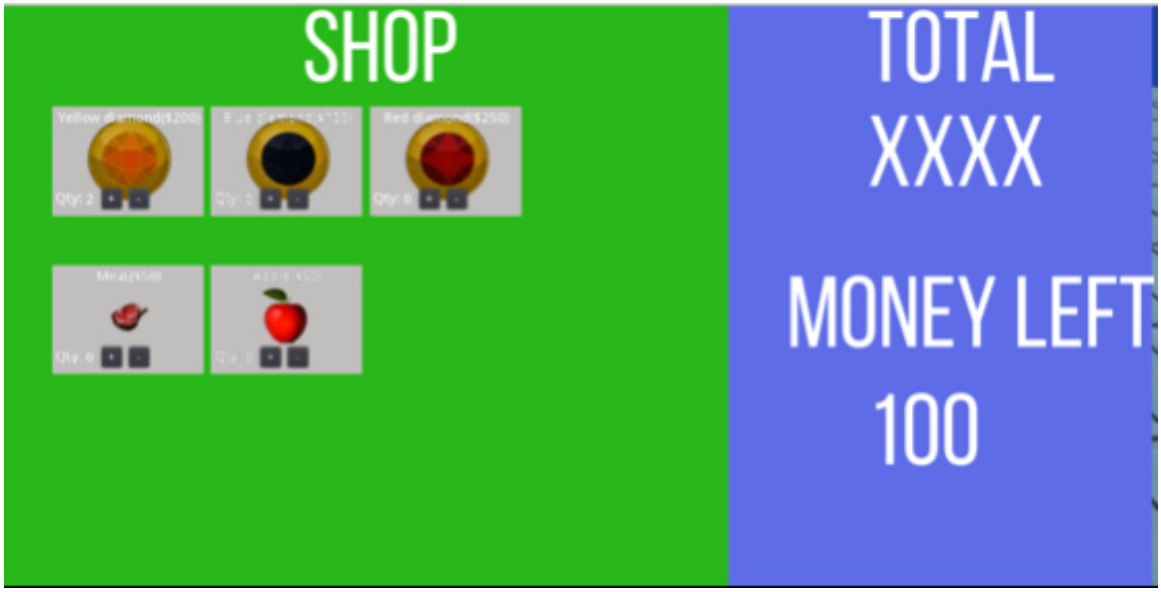
- Please save your code.
- Select the object **shop_system** (child of the object **shop**) in the scene **level1**.
- Drag and drop scene **shop_item.tscn** from the **FileSystem** tab to the empty slot to the left of the label **ShopItem Component** in the **Inspector**.



- You can hide or delete the node called **shop_item** that is already in the scene if it was added in the previous section to adjust the position of the other

items.

- Play the scene, and you should see the following interface.



You should be able to press the plus and minus buttons for each item and to update their quantity accordingly.

Updating the total and the amount of money left

At this stage, we just need to be able to update the shopping total and the money left for the player.

- Please add the following functions to the script **shop_system**.

```
func update_total(item_index:int, item_amount:int):
shop_items[item_index].nb = item_amount;
var temp_total:int;
temp_total = calculate_total();
print("Updating Total: index=" + str(item_index)+ "nb items"+str(shop_items[item_index].nb) + "; price:" + str(shop_items[item_index].price))
get_node("../../shop_ui/shop_total_value").text = str(temp_total)
total_purchase = temp_total
money_left = initial_money - temp_total
get_node("../../shop_ui/shop_money_left_value").text= "" + str(money_left)
```

In the previous code, we calculate the total amount for the shopping cart thanks to the method **calculate_total** (that we will create later); we also calculate how much money is left for the player, and both values are displayed on screen.

- Please add this function to the script **shop_system**:

```
func calculate_total()->int:
var temp:int = 0;
for i in range (0,shop_items.size()):
temp += shop_items[i].nb * shop_items[i].price;
return temp;
```

In the previous code, we go through all the items in the shop and we calculate the corresponding total based on the number of items selected and their price.

- Please open the script **shop_item** and add the following code to the function **update_quantity_label** (new code in bold).

```
func update_quantity_label():
get_node("item_qty").text = str(item_quantity)
get_tree().get_root().get_node("Spatial/
shop/shop_system").update_total(item_index, item_quantity);
```

In the previous code, we just update the total.

- Please add the following function to the script **shop_system**.

```
func can_add_items_to_cart(index:int):
if (money_left >= shop_items[index].price && shop_items[index].nb <
shop_items[index].max_nb):
return true
else: return false
```

In the previous code, before an item can be added to the cart, we check that we have enough money left to pay for it, and also that we can carry it without going over the limit specified for this item.

- Please add this code to the function **init** in the script **shop_system**.

```
initial_money = 1000
```

- Please update the function **can_click** in the script **shop_item** (new code in bold).

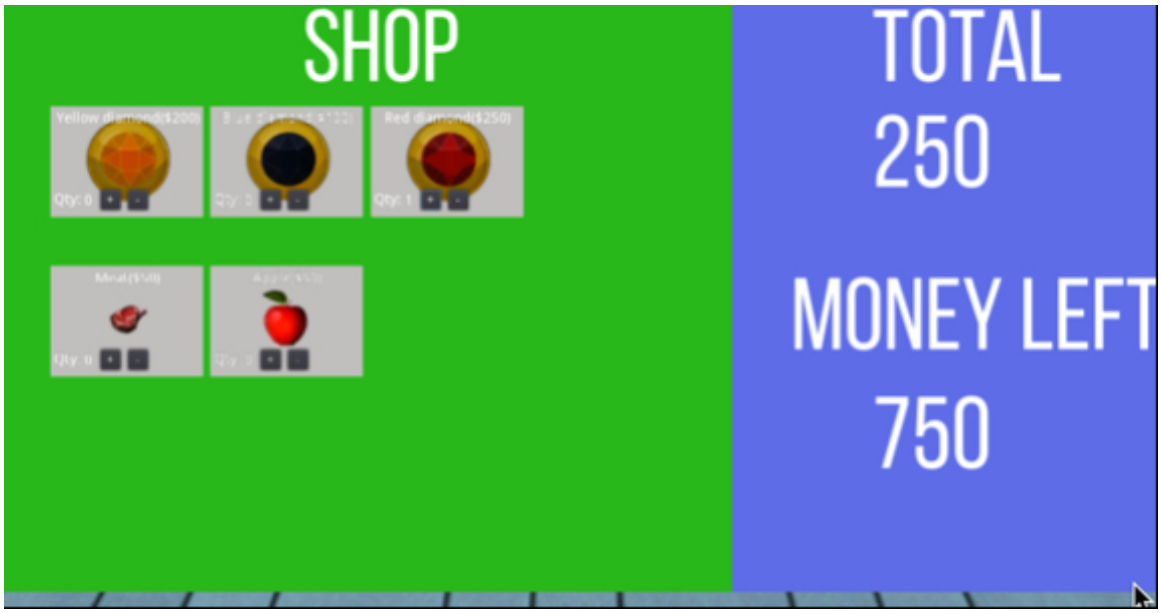
```
func can_click()->bool:
var  shop_sysem = get_tree().get_root().get_node("Spatial/shop/shop_system")
return shop_sysem.can_add_items_to_cart(item_index);
#return true
```

In the previous code, we comment the code that returns the value **true**, and we then return true only if the method **can_add_items_to_cart** returns **true** for this item.

You can now test your scene:

- Please save the code.
- Play the scene.
- Your initial amount of money will be 1000.

- As you add items to your cart, you will have less money left and the total will also increase.



Linking the shop to the players' gold coins

So while the shopping system works, you may notice that the initial money is set arbitrarily to 1000; as we have seen before the player will collect money throughout his/her journey, so we will, instead, use this amount whenever the player enters a shop.

- Please open the script **InventorySystem**.
- Add this function:

```
func get_money():
for i in range (0,player_inventory.size()):
if (player_inventory[i].type == Item.item_type.GOLD):
return (player_inventory[i].nb * player_inventory[i].price)
return 0
```

In the previous code, we identify how much money (gold) the player has in its inventory.

- Please open the script **shop_system**.
- Modify the function **init** as follows.

```
#initial_money = 1000
initial_money= get_tree().get_root().get_node ("Spatial/Player/
in-ventory_system").get_-money()
money_left = initial_money;
```

In the previous code, we ensure that the player will use its gold coins (stored in the inventory) to pay for items purchased in the shop.

- Please open the script **Item.gd** and ensure that the value of the attribute **nb** for the item **gold** is **10**, as per the next code snippet.

```
item_type.GOLD:
name = "Gold";
price = 100;
```

health_benefits = 0;

dammage = 0;

nb = 10;

In the next section, we will initialize and display the shop when the player presses a key, and then, once this is working, the shop will only be displayed when the player enters the shop in the 3D scene, using a trigger.

- Please open the file **shop_system**.
- Comment this code in the **_ready** function.

#init()

- Add this function to the script **shop_system**.

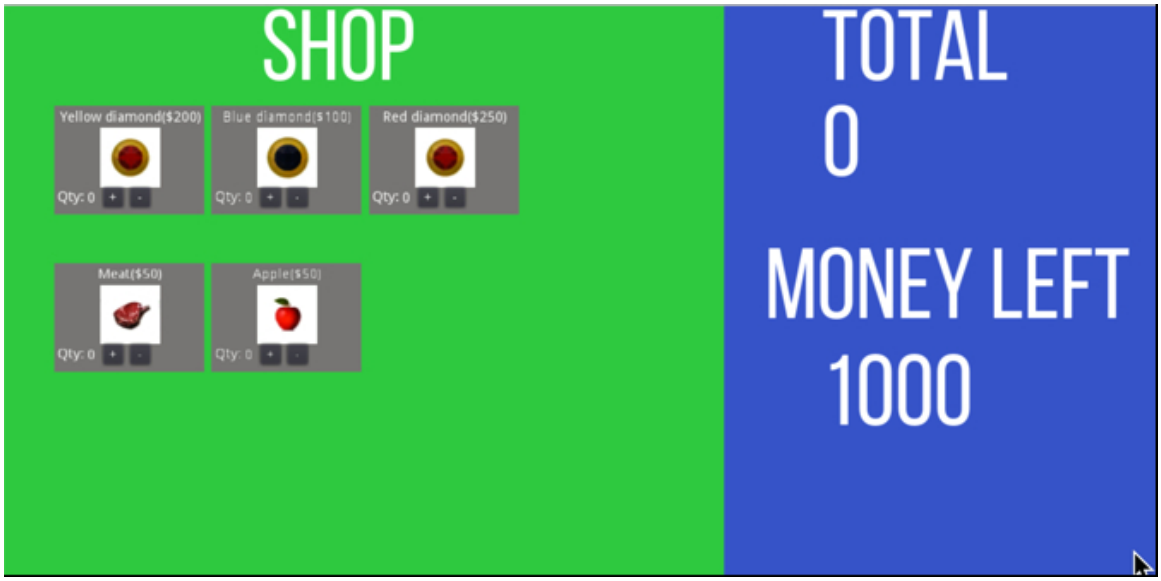
func _process(delta):

if (Input.is_action_just_pressed("fire1")):

init()

- Save your code.
- Map the **CTRL** key to the event "**fire1**" in the **Input Manager**.
- Play the scene.
- The shop panel should be empty.

- If you press the **CTRL** key on your keyboard then all the items should appear, as per the next figure.
- You should be able to add items using the plus buttons.



Once this is working, we could display the shop system only when the player collides with (i.e., enters) the shop and we will do that in the next section.

- Please open the script **manage_player**.
- Add this code at the beginning of the script.

```
var shop_is_displayed:bool;
onready var shop_ui = get_tree().get_root().get_node("Spatial/shop_ui")
```

- Add this code to the function **_ready**:

```
shop_ui.hide()
```

- Add this function to the script **manage_player**.

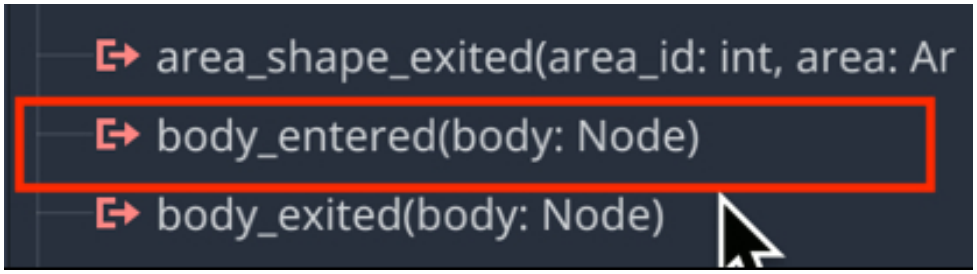
```
func display_shop_ui():
shop_ui.show()
```

- Add this new function to the script **manage_player**.

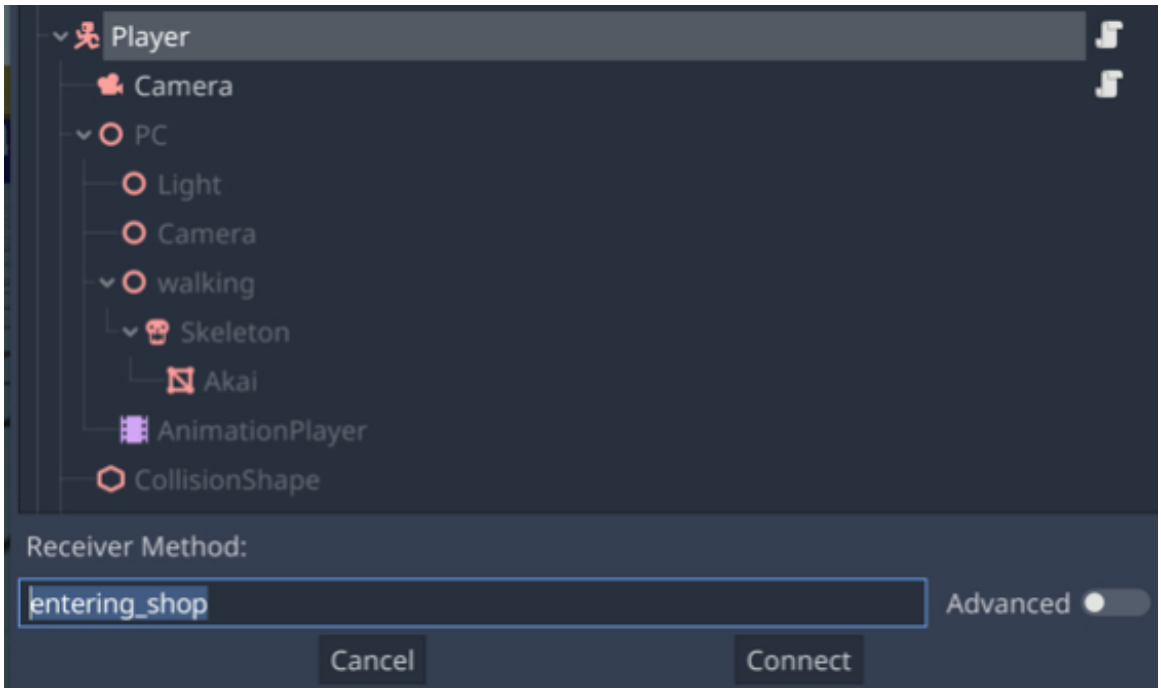
```
func entering_shop(body):
if (body.is_in_group("player")):
shop_is_displayed = true
display_shop_ui()
get_tree().get_root().get_node("Spatial/shop/shop_system").init()
speed = 0
```

In the previous code, we create a function that will be called when the player enters the shop; in this function, we check that the player is entering the shop, we display the interface for the shop and we then initialize its content.

- Please select the node **Player**, and using the **Inspector** and the tab **Node | Groups**, set its group to **player**.
- Select the node **shop**, select the tab **Node | Signals** in the **Inspector**, and double- click on the event **body_entered**.

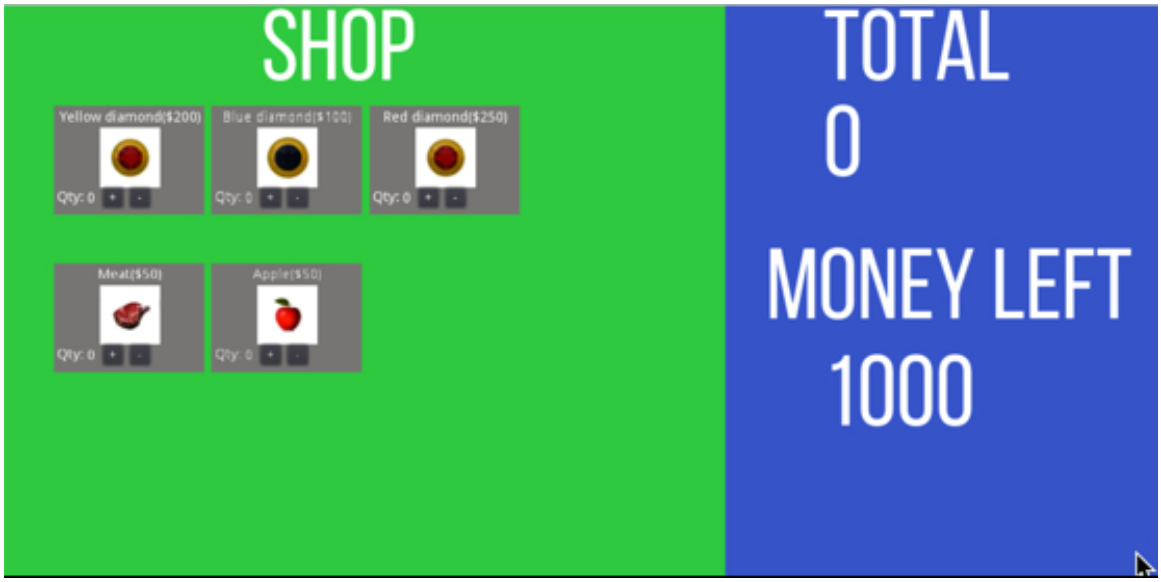


- In the new window, select the node **Player**, type the text **entering_shop** in the field labeled **Receiver Method**, and press **Connect**.



- Finally, please make sure that the attribute **Transform | Scale** for the node **CollisionShape** that is a child of the node **shop**, is **(4, 4, 3.5)** so that the collision shape is around the shop, and so that it is possible to detect collision between the player and the shop.

You can now play the scene and navigate to the shop, and as soon as you enter the shop, items should appear on screen.



Saving the items bought to the player's inventory

At this stage, we just need to be able to complete the transaction, get the player to pay for the items, and leave the shop. For this purpose, we will create a new button that will be pressed when the player wants to complete the purchase and leave the shop.

- Please create a new node of the type **Button** as a child of the node **shop_ui**, and rename it **pay_button**.
- Using the **Inspector**, change its text to "**Pay and Leave**".
- Using the section **Control | Rect**, set its size to **(120, 20)**.
- Move it to the bottom of the panel as per the next figure.

- Select the node **pay_button**, attach a new script to it and rename the script **pay_script**.
- Open the script **pay_script** and add this function to it.

```
func leave_shop():
var purchased_items = []
var value_of_gold_coins = Item.new(Item.item_type.GOLD).price
purchased_items = get_tree().get_root().get_node("Spatial/shop/shop_system").shop_items
```

```
var money_left = get_tree().get_root().get_node("Spatial/shop/shop_system").money_left;
get_tree().get_root().get_node("Spatial/Player/in-ventory_system").set_money(-ey_left/val-ue_of_gold_-coins);
```

In the previous code:

- We define the items that have been purchased.
- We define how much money the player has left.
- We set the "**wallet**" of the player with the money that's left.

Please add this code to the previous function:

```
get_tree().get_root().get_node("Spatial/Player/in-ventory_system").ad-d_purchased_item-s(purchased_items);
get_tree().get_root().get_node("Spatial/Player").shop_is_displayed = false;
get_tree().get_root().get_node("Spatial/Player").end_talking()
get_tree().get_root().get_node("Spatial/shop_ui").hide()
```

In the previous code:

- We add the items purchased to the player's inventory.
- We hide the UI for the shop.
- We call the function **end_talking** that moves the player away from the shop and rotates it also so that it is facing outwards.

We now just need to create the functions **set_money** and **add_purchased_item**.

- Please open the script **InventorySystem**.
- Add this function to the script.

```
func add_purchased_items( purchased_items):
var t:bool;
for i in range (0, purchased_items.size()):
if (purchased_items[i].nb > 0):
t = update_inventory(purchased_items[i].type, purchased_items[i].nb);
```

In the previous code, all the items that have been purchased are added to the player's inventory through the method **update_inventory**.

- Please add this function to the script:

```
func set_money(new_amount:int):
for i  in range (0, player_inventory.size()):
if (player_inventory[i].type == Item.item_type.GOLD):
player_inventory[i].nb = new_amount
```

In the previous function, we set the money owned by the player.

Next, we need to link the button "**Pay and Leave**" to the code that we have created.

- Please select the node **pay_button**.
- Using the **Inspector**, select the tab **Node | Signals**, double-click on the event **pressed**.
- In the new window, select the node **pay_button**, type **leave_shop** in the field **Receiver Method**, and press **Connect**.

You can now save your code, play the scene, check your inventory beforehand, go to the shop, buy a few items, leave the shop and check that you have fewer gold coins and more items added to your inventory.

If you notice that the text for some of the items (e.g., for the diamonds) is not displayed fully, you may need to adjust the size of the objects **inventory_panel**, **inventory_description**, or **inventory_text**.

LEVEL ROUNDUP

Summary

In this chapter, we managed to create a shop system whereby the user can buy items based on the gold that s/he owns, and add them to their inventory. Of course, this shopping system could be improved, but for now it works well and makes it possible to implement a simple system to your RPG.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are true or false. The solutions are on the next page.

1. It is possible to create labels using the nodes of the type **Label**.
2. It is possible to add an item to an array using the method **push_back**.
3. The following code will change the text of a node of the type **Label**.

```
text ="hello";
```

1. The following code will add the node "**new_node**" to the node "**parent_node**".

```
get_node("parent_the_node").add_child("new_node");
```

1. The following function should return an integer value.

```
func calculate_total()->int:
```

1. The event called "**pressed**" is called whenever a user clicks on a button in Godot.
2. It is not possible to create a scene and to then add this scene to another scene, as a child node.
3. The following code will create a loop that will iterate from 0 to the size of the array called **shop_items**.

```
for i in range (0,shop_items.size()):
```

1. The following code will access the root of the current scene.

```
get_tree().get_root()
```

1. The function **_process** is called every second.

Solutions to the Quiz

1. TRUE.
2. TRUE.
3. TRUE.
4. TRUE.
5. TRUE.
6. TRUE.
7. FALSE.
8. TRUE.
9. TRUE.
10. FALSE (every frame).

Checklist

You can move to the next chapter if you can do the followin

- Create an array.
- Create a panel with **ColorRect** nodes.
- Create and instantiate scenes.
- Process clicks.

Challenge 1

For this challenge, you will need to:

- Change the objects available in the shop.
- Add new objects to be sold in the shop.
- Create a field where the player can enter (and apply) a discount code.

Chapter 6: Adding Weapons and Protagonists

In this chapter, you will add the ability for the player to engage in battles against intelligent NPCs.

After completing this section, you should be able to:

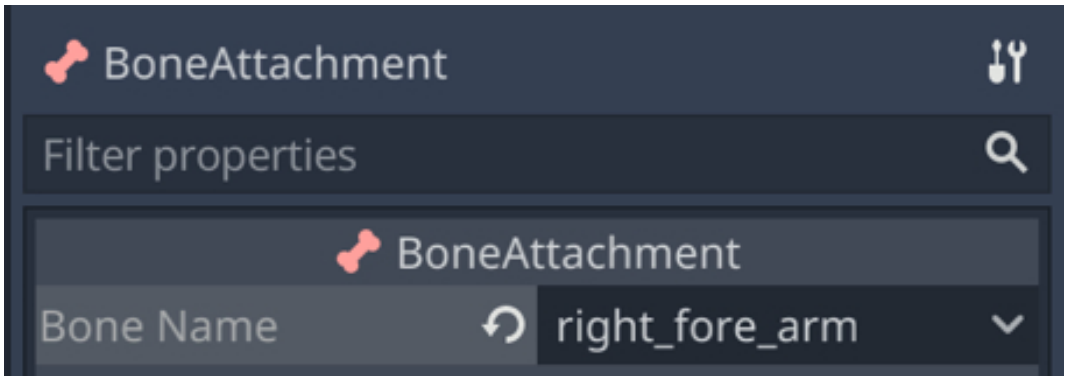
- Provide a sword to your main character.
- Add animations for attacks.
- Make it possible for the main character to inflict damage to NPCs.
- Create NPCs that can be idle, patrolling, or attacking the player.
- Create NPCs that follow a path, detect the player, and engage in a battle.
- Amend the health of the player based on the amount of damage inflicted by enemies.

Adding a weapon

In this section, we will add a sword to the right hand of the main character.

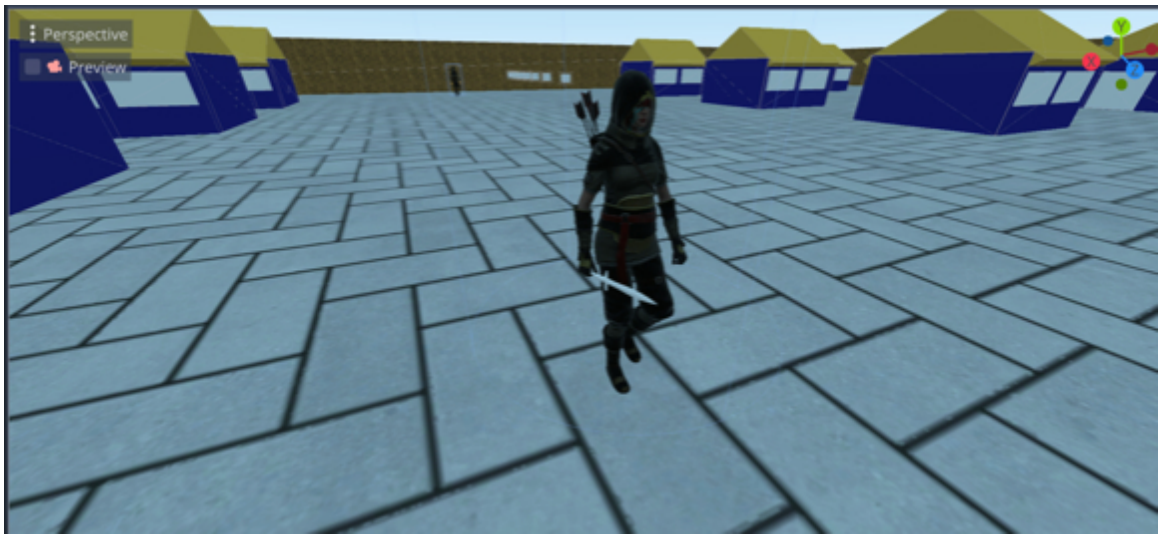
- Please open (or switch to) the scene **level1**.
- Select the node **Player**.
- Identify and click on the node **Player | PC | walking | Skeleton**.

- Add a node of the type **BoneAttachment** as a child of the node **Skeleton**
- Change its **position** to **(-25, -9, -128)**, its rotation to **(-76, 118, -33)**.
- Select the node **BoneAttachement**, and using the **Inspector**, click to the right of the label **Bone Name** (in the section **Bone Attachment**), and select the option **right_fore_arm** from the drop-down list as illustrated in the next figure.

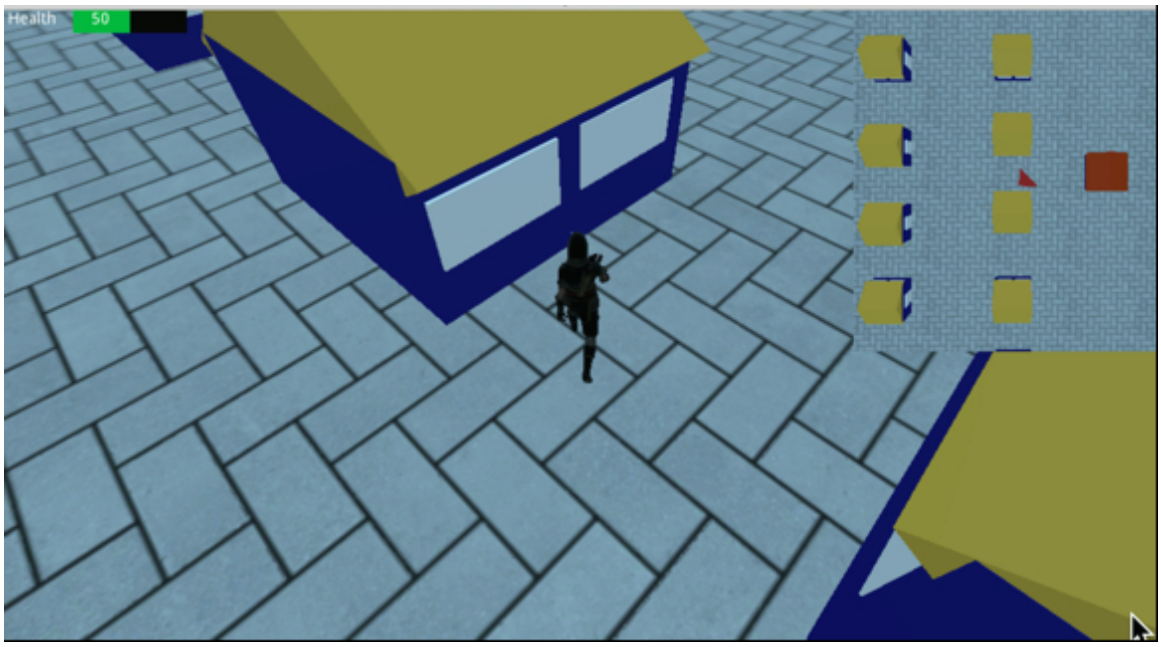


- Add a node of the type **Spatial** as child of the node **BoneAttachment** and rename the new node **player_weapon**.

- Import the object **sword.fbx** from the resource pack (i.e., drag and drop this file from the folder **3D-objects** in the resource pack to the folder **res://** in **Godot**).
- Drag the asset **sword.fbx** from the tab **FileSystem** atop the node **player_weapon** in the **Hierarchy Tree**; this will create a new node called **sword** as a child of the node **player_weapon**.
- Change its position to **(-42, -4, 31)**, its rotation to **(0, 180, -180)** and its scale to **(100, 100, 100)**.
- You should now see that the sword is held in the character's right hand, as per the next figure.



- As you play the scene, you should now see that the sword is displayed at all times.


Health
50

Performing an attack against a stationary target

In this section, we will make it possible for the player to attack a stationary target; this will involve the following:

- Creating targets with a corresponding groups.
- Detecting when the player's sword hits a target.
- Adding health to the target, decreasing this health when the target is hit, and finally destroying the target when its health has reached **0**.

So let's proceed:

- Please open the script **manage_player**.
- Add the following code at the beginning of the script (new code in bold):

```
enum {IDLE, WALK, WALK_REVERSE, RUN, ATTACK_WITH_SWORD}
```

- Add the following code to the **_ready** function:

```
pc_node.get_animation("sword-and-shield-slash").loop = false
```

- Add the following code to the function **_process** (within the **match** structure).

```
ATTACK_WITH_SWORD:
pc_node.play("sword-and-shield-slash")
yield(pc_node, "animation_finished")
current_state = IDLE
```

In the previous code, we play the attack animation, we wait until it is complete, and we then set the character to the **IDLE** state.

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

```
if (Input.is_action_just_released("fire1")):
current_state = ATTACK_WITH_SWORD
```

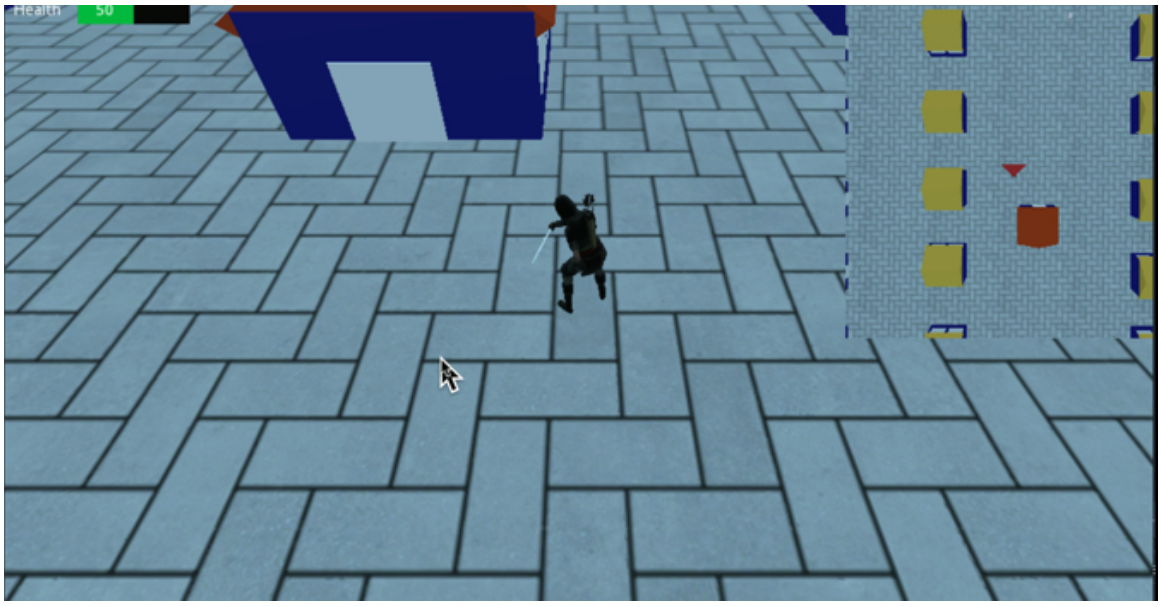
In the previous code, we check whether the player has just released the key mapped to the action "**fire1**", and in that case, we change the character's state to

ATTACK_WITH_SWORD.

- Before we can check this feature, please comment the following function in the script **shop_system**.

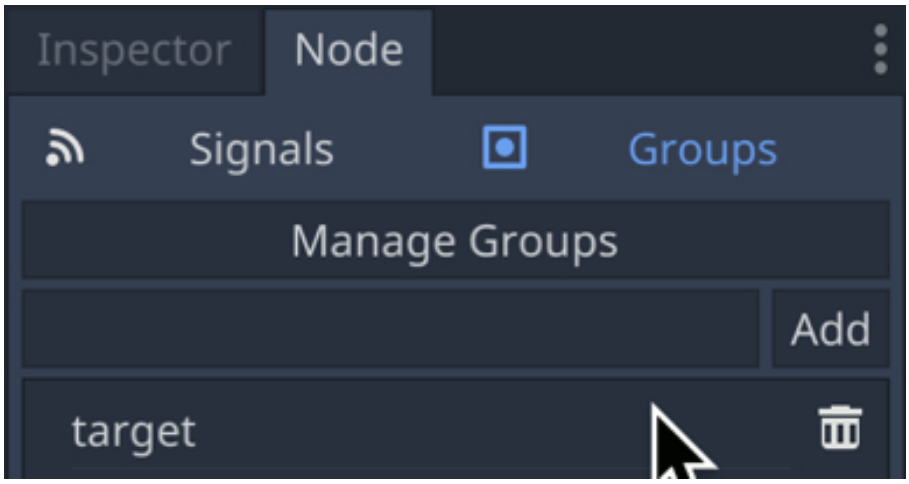
```
#func _process(delta):
# if (Input.is_action_just_pressed("fire1")):
#  init()
```

- Please save your code and play the scene. You should be able to press the left **CTRL** key to perform an attack.

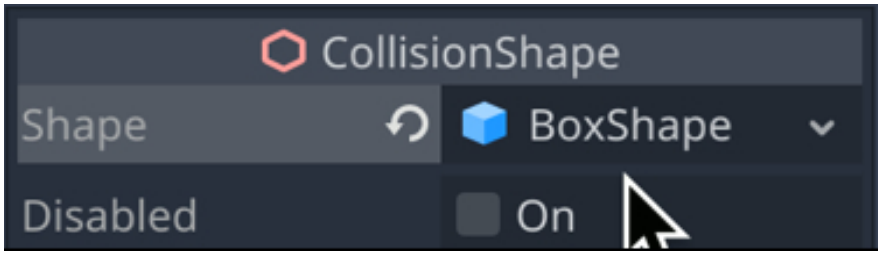


Now that we know that the attack animation can be triggered, we will create a target on which the player can train and perform attacks.

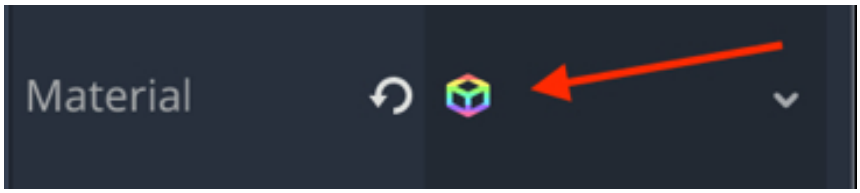
- Please create a node of the type **Spatial** as a child of the node **Spatial**, rename the new node **target**, move it a few meters from the player, and change its **y** coordinate to **1.5**.
- Add a node of the type **StaticBody** as a child of the node **target**.
- Select the node **StaticBody**, and, using the tab **Node | Groups**, set its group to **target**.



- Add a node of the type **CollisionShape** as a child of the node **StaticBody**.
- Select the node **CollisionShape**, and, using the **Inspector**, click to the right of the label **Shape** (**Collision | Shape**) and select the option **New BoxShape** from the drop-down menu.

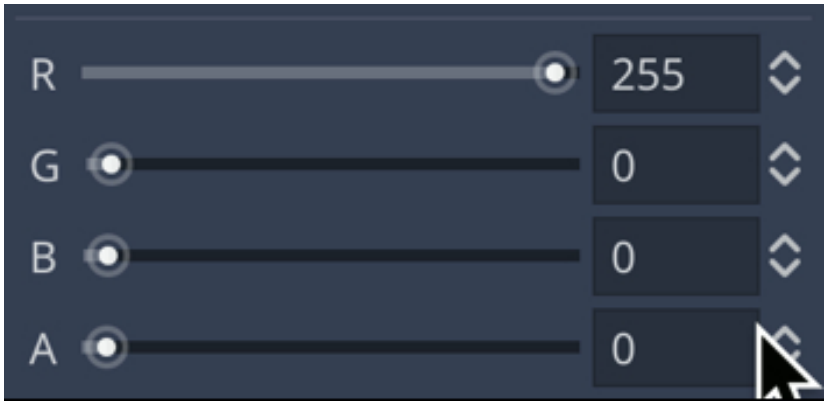


- Add a new node of the type **CSGBox** as a child of the node **StaticBody**; check that its size (i.e., **width**, **height**, and **depth**) is **(2, 2, 2)**, and apply a **red** material to it. This shape will be there to represent the target.
- Finally, add a new node of the type **CSGMesh** as a child of the node **StaticBody**, and check that its size (i.e., **width**, **height**, and **depth**) is **(2, 2, 2)**. This node will be used to display a flash whenever the target has been hit.
- Select the node **CSGMesh**, and using the **Inspector**, click to the right of the label **Mesh**, and select the option **New SphereMesh** from the drop-down menu.
- Click to the right of the label **Material** and select the option **New Spatial Material** from the drop-down menu.
- Once this is done, click on the **cube** logo to the right of the label **Material**.



- This will display additional attributes.
- Please expand the section called **Albedo**, and click to the right of the label **Color**.

- In the new window, set the **color** to **red** (i.e., **R=255**, **G=0**, **B=0**), and the **alpha** value (i.e., the attribute **A**) to **0**.



Now that the target node has been created, we will create a script that will manage the target's health.

- Please select the node **StaticBody** (that is a child of the node **target**).
- Attach a new script to this node and rename the script **manage_target_health.gd**.
- Add this code at the beginning of the script.

```
var health:int = 100
```

In this code, we set the initial health to 100 for the target.

In the next section, we will create several methods that will be used to decrease the health of a target or to destroy the target when its health has reached 0.

- Please add this function to the script **manage_target_health**.

```
func set_health(new_health:int):
health = new_health
if (health <=0):
health = 0;
destroy_target()
```

- Add this function.

```
func get_health():
return health;
```

- Add this function:

```
func destroy_target():
var name_of_parent = get_parent().name;
get_parent().queue_free()
```

- Add this function.

```
func decrease_health(increment:int):
set_health(health-increment)
```

Finally, we need to decrease the health of the target whenever it collides with the players' weapon.

- Please create a new node of the type **Area** as a child of the node **sword**.
- Add a new node of the type **CollisionShape** as a child of the node **Area**.
- Select this new node, and, using the **Inspector**, click to the right of the label **Shape** and select the option **New CapsuleShape**.

Once this is done, we will need to create a script that will be called whenever

the sword collides with an object:

- Please attach a new script to the node **Area** (that this a child of the node **sword**), and rename the script **detect_target.gd**.
- Please open the script **detect_target.gd**.
- Add the following code to it:

```
func detect_target(body):
if (body.is_in_group("target")):
if (get_tree().get_root().get_node("Spatial/Player").is_attacking()):
body.decrease_health(10)
```

In the previous code:

- We check whether the object that the sword is colliding with is part of the group **target**.
- If this is the case, we check that the player is actually attacking using the function **is_attacking** (that we yet have to create).
- Finally, we decrease the health of the target by 10 using the function **decrease_health** that is available in the script attached to the target.

We now need to link the collision event to the function d**etect_target**, and to also create the function **is_attacking**:

- Please add the following code to the script **manage_player**:

```
func is_attacking()->bool:
if (current_state == ATTACK_WITH_SWORD): return true;
else: return false;
```

In the previous code, we return the value **true** if the character is in the state **ATTACK _WITH_SWORD** and **true** otherwise.

Finally, please modify the following code in the script **manage_player**; you will need to indent the code that is after the third line (new code in bold):

```
for index in get_slide_count():
```

var collision = get_slide_collision(index)

if (collision.collider != null):

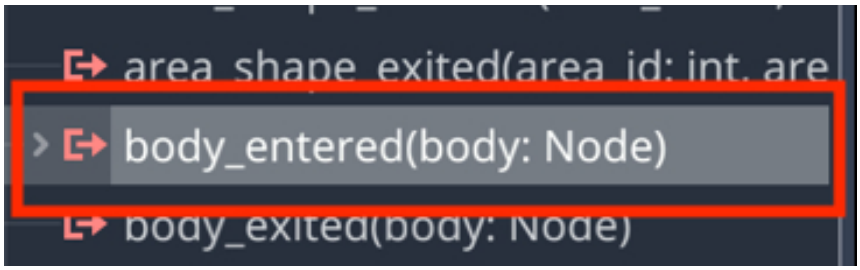
if (collision.collider.is_in_group("NPC_TALK")):

In the previous code, we ensure that the collision between the player and the target is not accounted for when the latter has been removed (i.e., is **null**).

The full function should then look as the following:

```
func _physics_process(delta):
for index in get_slide_count():
var collision = get_slide_collision(index)
if (collision.collider != null):
if (collision.collider.is_in_group("NPC_TALK")):
if (collision.collider.name == "Diana" && !is_talking):
dialogue_image.texture = load("res://Diana.jpg");
collision.collider.start_dialogue()
current_state = IDLE
is_talking = true;
```

- Please select the node **Area** (that is a child of the node **sword**), select the tab **Node | Signals**, and double-click on the event **body_entered**.



- In the new window, select the node **Area** (that is a child of the node **sword**), enter the text **detect_target** in the field **Receiver Method**, and press **Connect**.
- You can now test the scene and check that after hitting the target **10** times it disappears.

Adding a flash when the target is hit

Next, we will add some feedback for when the target is hit; this will involve a red flash around the target displayed for less than a second; for this purpose, we will proceed as follows; we will:

- Use the sphere that we have already created as a child of the object **target**.
- Make it invisible.
- Color this sphere in red whenever the target is hit.
- Progressively make this red sphere transparent again in less than 1 second.

Let's proceed, by controlling the sphere's appearance through a script:

- Please open the script **manage_target_health**.
- Add this code at the beginning of the script.

```
var hit_timer:float;
var hit_flash:bool;
var alpha:float;
```

- Add this code in the **_ready** function.

```
alpha = 0.0
var material;
var the_node = get_node("CSGMesh")
material = the_node.material
material.albedo_color = Color(1,0,0,alpha)
```

In the previous code, we set the sphere to red, but it is still transparent at this stage (i.e., alpha = 0);

- Add this code to the function **decrease_health**.

```
hit_flash = true;
alpha = 0.5
```

In the previous code, we specify that the sphere is now semi-transparent, and

that we can start the animation whereby the sphere will progressively disappear by becoming more and more transparent.

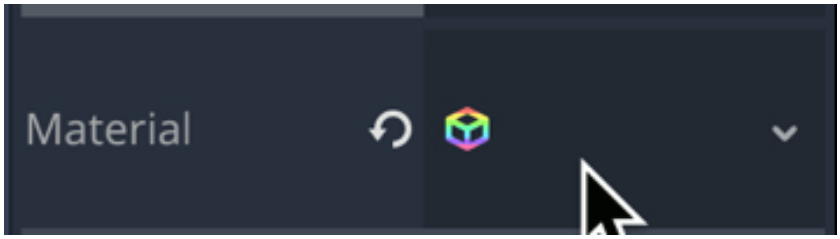
- Please add this function to the script **manage_target_health**:

```
func _process(delta):
if (hit_flash):
alpha -= delta
var material;
var the_node = get_node("CSGMesh")
material = the_node.material
material.albedo_color = Color(1,0,0,alpha)
if (alpha <= 0):
hit_flash = false
alpha = 0
```

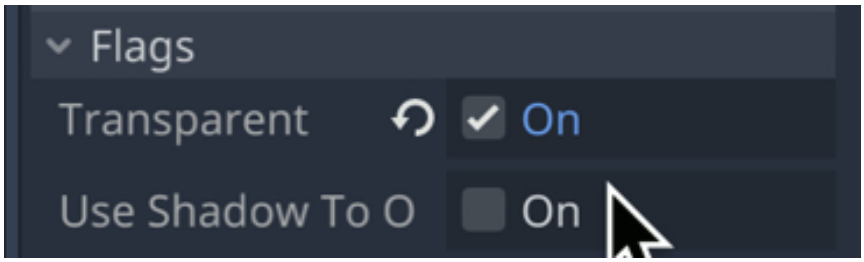
In the previous code, we create a timer, whereby the transparency value (i.e., alpha) for the red sphere will decrease by **1** every second, or **.1** every tenth of a second until it reaches 0.

Before you test the scene, please do the following:

- Using the **Inspector**, please modify (or check) that the scale attribute (i.e., **Transform | Scale**) of the node **CSGMesh** that is a child of the node **target** is **(2, 2, 2)**.
- Click on the cube to the right of the label **Material**.

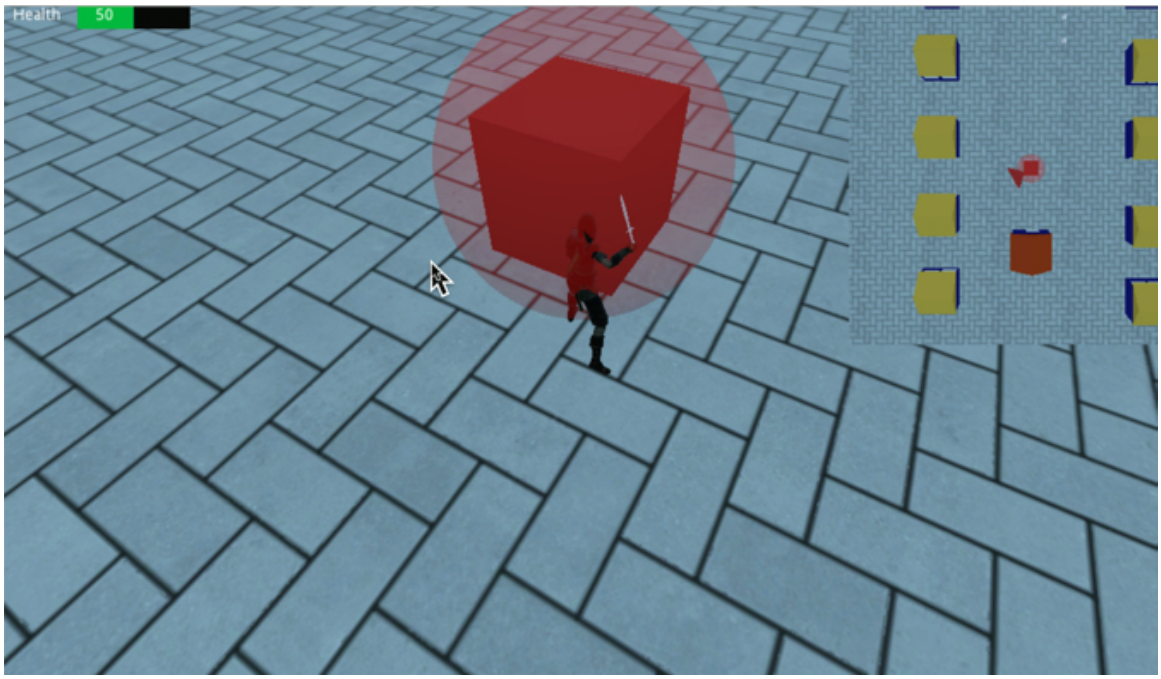


- In the new window, expand the section called **Flags** and set the attribute **Transparent** to **On**.



- Expand the section called **Vertex Color**, and set the attribute **Use As Albedo** to **On**.

You can save your code and test the scene by hitting the target several times and you should see a red flash around the target, as per the next figure.



Creating intelligent NPCs

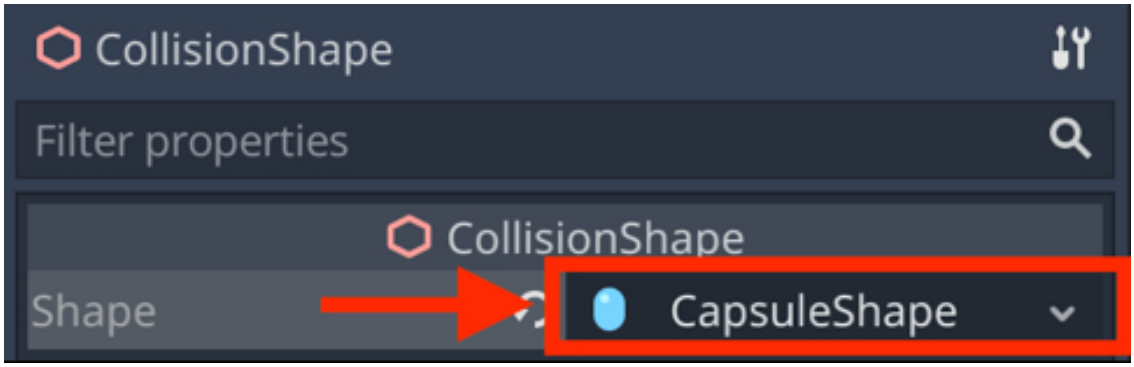
At this stage we can move our character around, pick up objects, and attack a target; so to increase the challenge for the player, we will start to create NPCs that will move along a path, detect and also attack the player; the player will also be able to attack them.

In this section we will proceed as follows; we will:

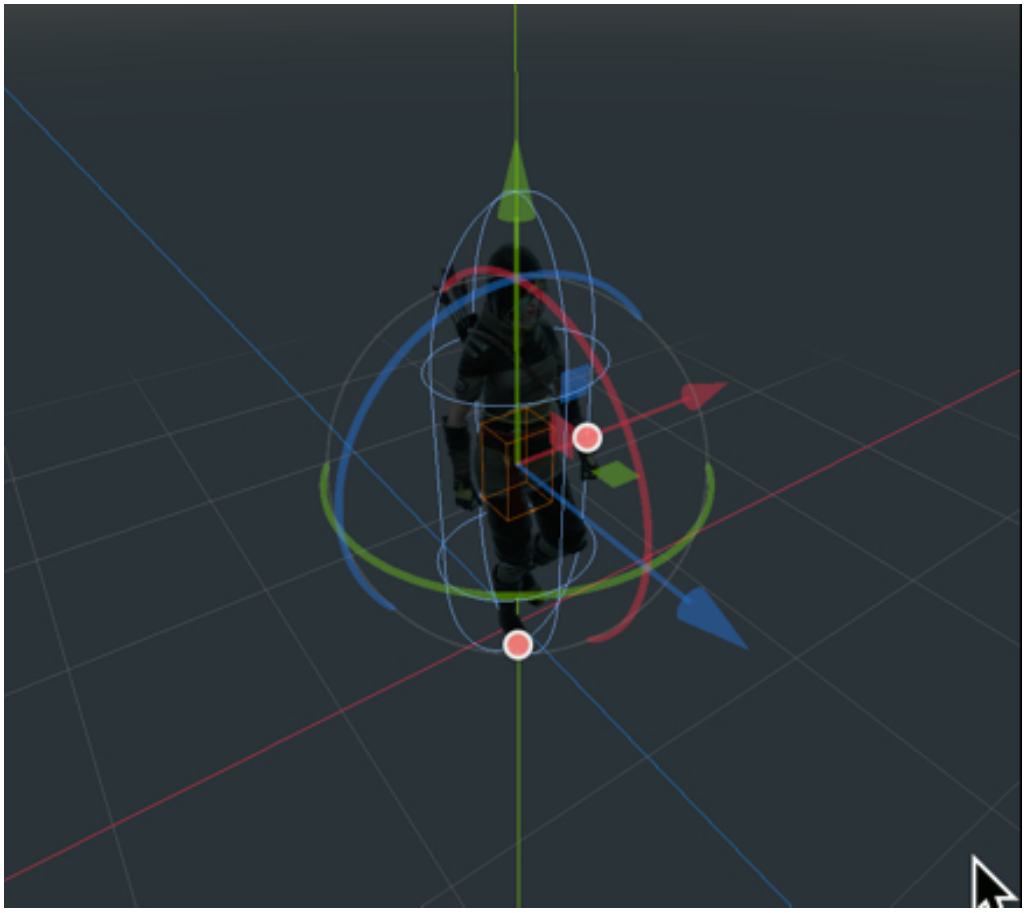
- Import a 3D character for the NPCs.
- Animate the NPCs.
- Create different states and transitions for the NPCs depending on their types.
- Make it possible for these NPCs to detect and attack the player.

Let's proceed:

- Please create a new scene of the type **KinematicBody** (**Scene | New Scene**).
- The new scene will have a default node called **KinematicBody**.
- Please rename this node **npc_guard**.
- Please drag and drop the item called **akai.gltf** from the **FileSystem** window atop the node **npc_guard**, so that this item becomes a child of the node **npc_guard**, and rename this new node **NPC**.
- Add a new node of the type **CollisionShape** as a child of the node **npc_guard**.
- Using the **Inspector**, click to the right of the label **Shape** and select the option **New CapsuleShape** from the drop-down menu.
- Click on the item **CapsuleShape** to the right of the label **Shape**.



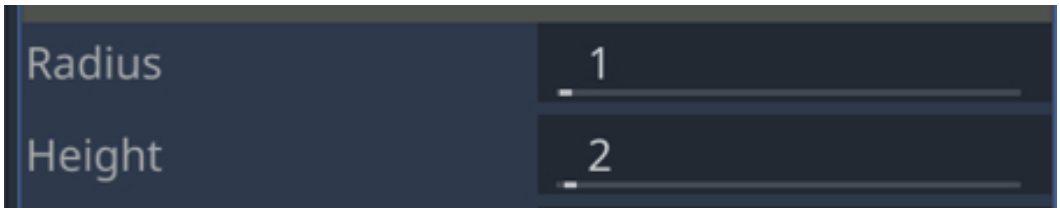
- This will display information about the C**apsuleShape** item.
- Please check that the radius is **.67** and that the height is **1** for this **CapsuleShape**.
- Expand the **Transform** section in the **Inspector** for the same node (i.e., **CollisionShape**).
- Modify the **position (**i.e., the attribute **Translation)** to **(0, 1, 0)**, the **rotation** to **(90, 0, 0)**, and the **scale** to **(0.6, 0.5, 1)**.



- Finally, add a node of the type **CSGMesh** as a child of the node **npc_guard**.
- Select this node (i.e., **CSGMesh**).
- Change its position (i.e., **Transform | Translation**) to **(0, 1, 0)**.
- Using the **Inspector**, right-click to the right of the label **Mesh**, and select the option **New SphereMesh** from the drop-down menu.
- Click on the sphere to the right of the label **Mesh**, this will display more

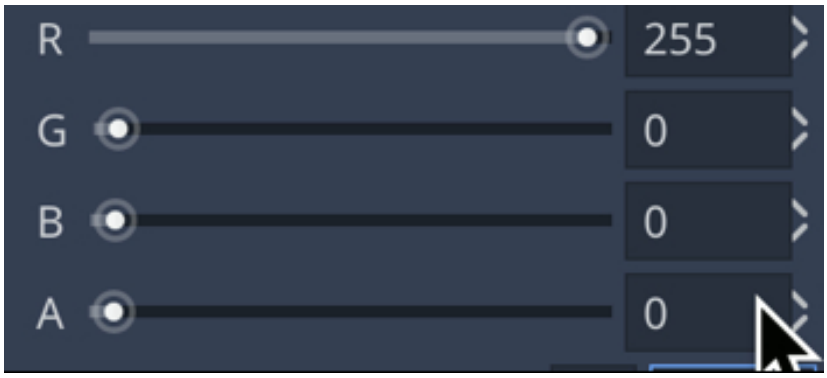
information about the new mesh.

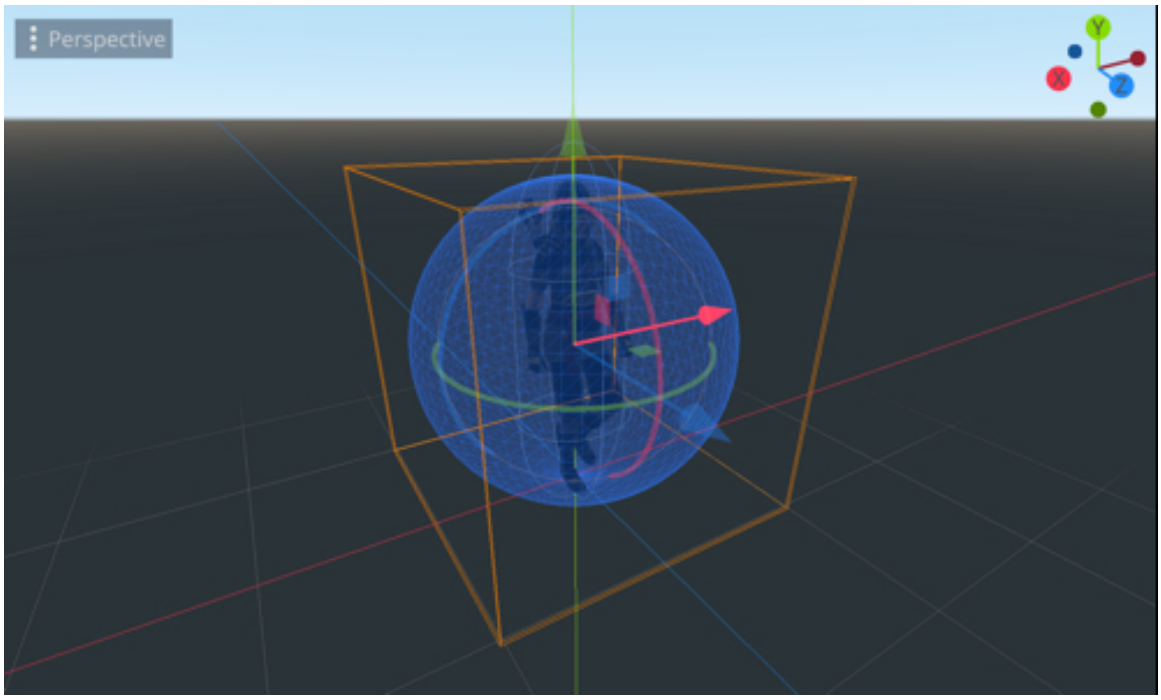
- Check that the **radius** is **1** and that the **height** is **2**.



- Right-click to the right of the label **Material**, and select the option **New Spatial Material** from the drop-down menu.
- Click on the cube to the right of the label **Material**.

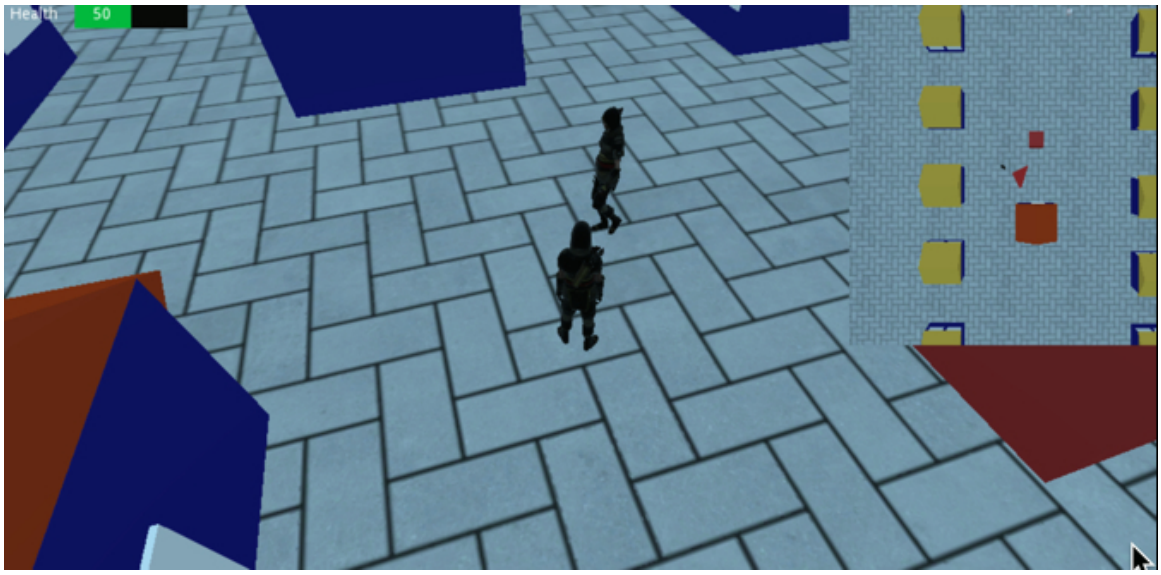
- This will display a list of properties.
- Expand the section called **Flags**, and set the attribute **Transparent** to **On**.
- Expand the section called **Vertex Color**, and set the attribute **Use As Albedo** to **On**.
- Expand the section called **Albedo**, and click on the white rectangle to the right of the label **Color**.
- In the new window, set the color to **red**, and the **Alpha** attribute to **0**.





We can now add our NPC to the main scene:

- Please save the current scene as **npc_guard.tscn**.
- Open (or switch to) the main scene (i.e., **level1**).
- Right-click on the node **Spatial**, select the option **Instance Child Scene**, and select the scene **npc_guard**, that we have just created.
- This will create a new node called **npc_guard**, and you may adjust its **y** coordinate to **.5**.



Now that we have set up the NPC, it is time to animate it; so in this section, we will create all the necessary states for the NPC to be able to be idle, to patrol, and to follow or to attack the player.

First let's use the **idle** animation:

- Please open the scene called **npc_guard**.
- Select the node **npc_guard**.
- Attach a new script to this node and name this script **manage_npc_guard**.
- Add is code at the beginning of the script.

```
enum {IDLE}
var current_state = IDLE
onready var player = get_node("/root/Spatial/Player")
```

In the previous code:

- We declare an **enum** that will be used to list the different states for our NPC.
- We set the current state to **IDLE**.
- We link the variable **player** to the **Player** node .

We can now set-up the **idle** animation.

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

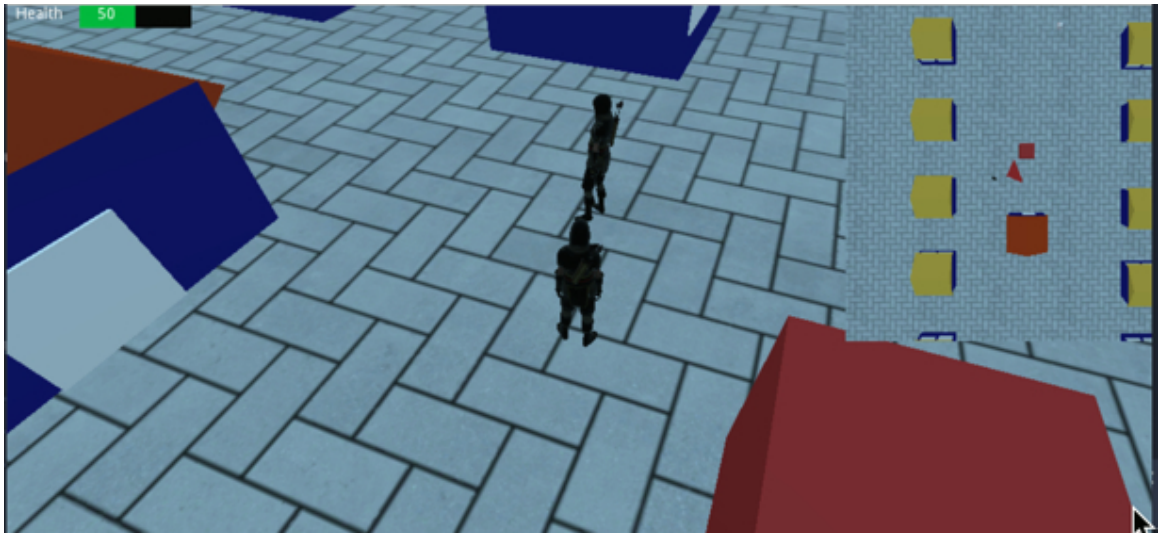
```
func _ready():
get_node("NPC/AnimationPlayer").get_animation("idle").loop = true
```

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

```
func _process(delta):
match current_state:
IDLE:
get_node("NPC/AnimationPlayer").play("idle")
```

- In the previous code, as we have done in earlier sections for the player character, we play the **idle** animation while in the state **IDLE**.

You can now save your code, switch to the scene **level1**, and play the scene: you should see the **idle** animation is playing while the guard is in the **IDLE** state.



Once you have checked that this is working, we can configure the guard so that it can patrol.

- Please open the script **manage_npc_guard**.
- Modify this line (new code in bold):

```
enum {IDLE, PATROL}
```

- Add this code before the function **_ready**.

```
onready var WP1 = get_node("/root/Spatial//WP1")
onready var WP2 = get_node("/root/Spatial/WP2")
onready var WP3 = get_node("/root/Spatial/WP3")
onready var WP4 = get_node("/root/Spatial/WP4")
var WP_index = 0
var current_WP
var patrol_previous_position
var speed = 1
```

In the previous code:

- We create the variables **WP1**, **WP2**, **WP3** and **WP4** that will be linked to the waypoints **WP1**, **WP2**, **WP3** and **WP4** and that we yet have to create.
- We create the variables **WP_index** and **current_WP** that will be used to keep track of the current way point followed by the NPC.
- We also create the variable **patrol_previous_position** to store the initial position of the NPC, along with the variable **speed** to set the speed of the NPC.

Now that these variables have been created, we can start to set up and initialize the animations and the state called patrol state.

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

```
get_node("NPC/AnimationPlayer").get_animation("walking").loop = true
current_state = PATROL
current_WP = WP1
patrol_previous_position = global_transform.origin
```

In the previous code:

- We ensure that the **walking** animation will be looping.
- We set the NPC's state to **PATROL**.
- We set the first waypoint to be followed by the NPC.
- We then save the current position.

Now that this has been set-up we can manage the state called **PATROL**.

- Please add the following code to the function **_process** in the **match** section (the keyword **PATROL** will need to be as the same level of indentation as the keyword **IDLE**)

```
PATROL:
match (WP_index):
0:current_WP = WP1
1:current_WP = WP2
2:current_WP = WP3
3:current_WP = WP4
get_node("NPC/AnimationPlayer").play("walking")
```

In the previous code:

- We create a new state called **PATROL**.
- In that state, we set the current waypoint to be followed based on the value of the variable **current_WP**.
- We then play the animation called **walking.**

Please add the following code just after the previous code (i.e., within the code for the state **PATROL** in the function **_process**):

```
var distance_to_WP = (current_WP.global_transform.origin - global_transform.origin)
if (distance_to_WP.length() < .5) :
WP_index += 1
if (WP_index > 3): WP_index = 0
var direction = (current_WP.transform.origin - transform.origin).normalized();
move_and_slide(direction.normalized()*speed, Vector3.UP)
look_at(global_transform.origin - direction, Vector3.UP)
```

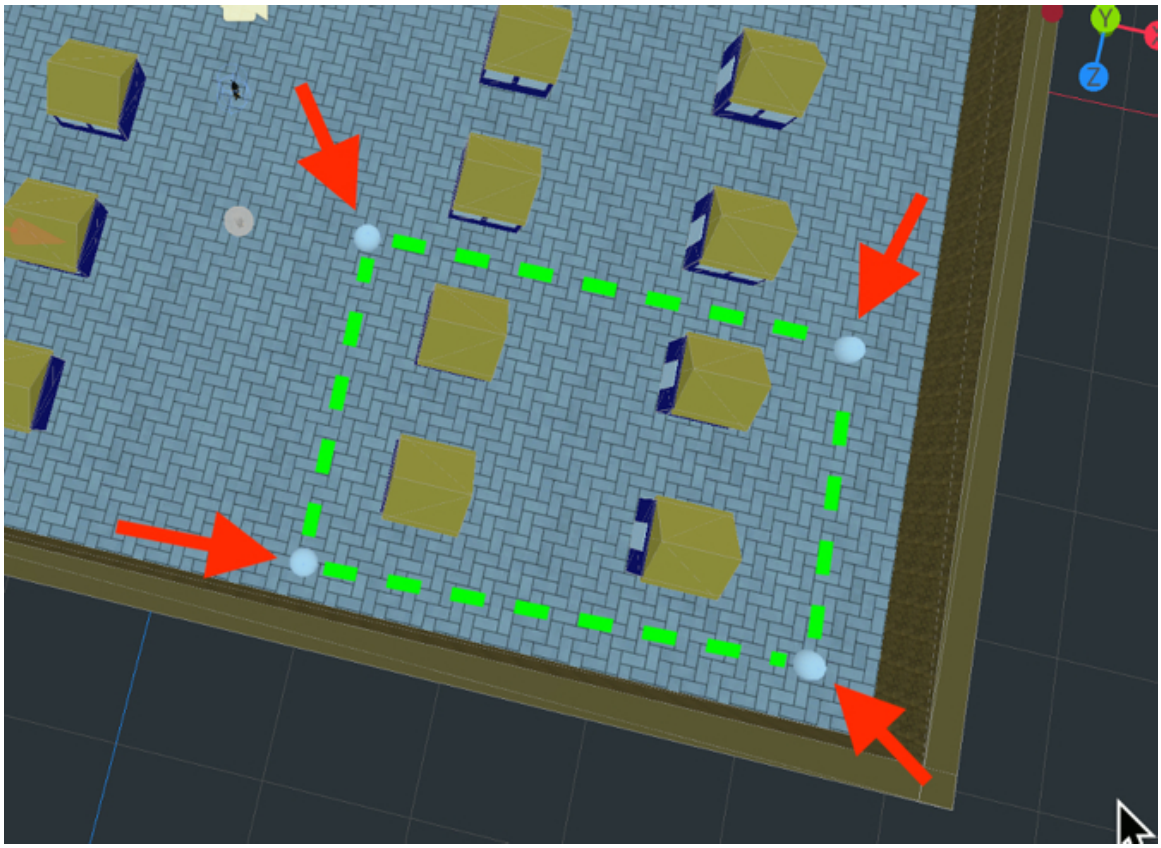
In the previous code:

- We calculate the distance between the NPC and the current waypoint.

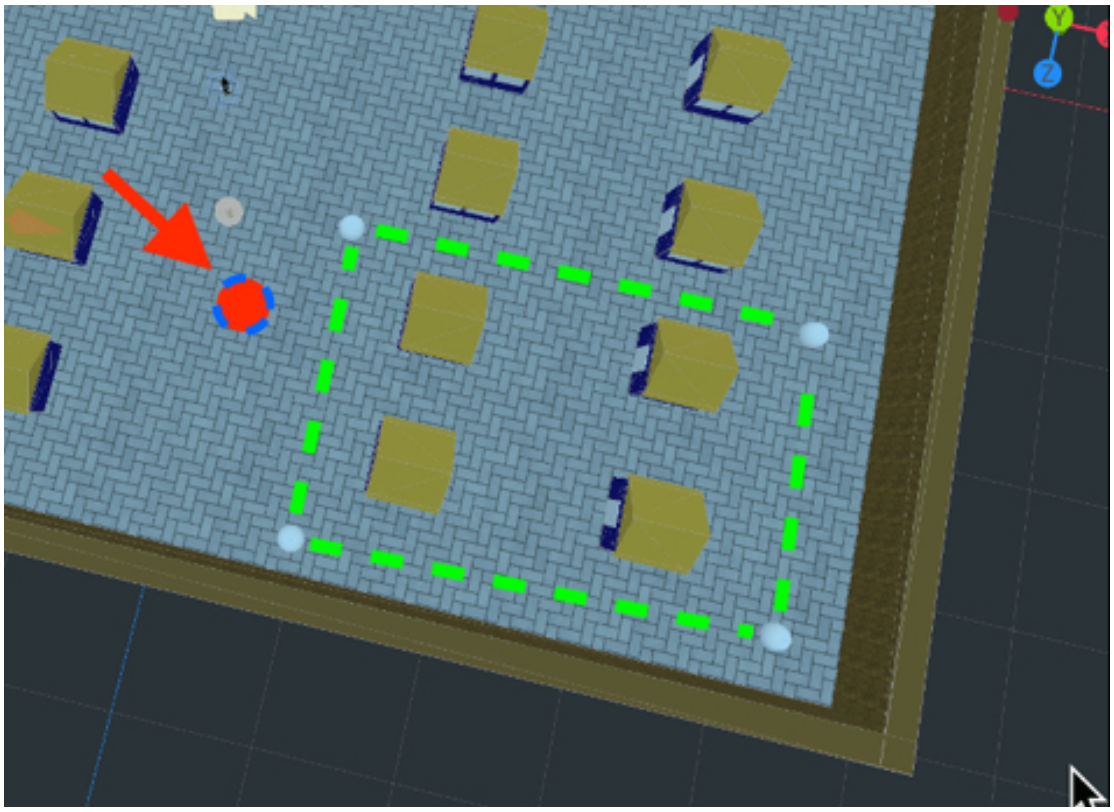
- If this distance is less than **.5** meters we ensure that the NPC starts to follow the next waypoint.
- The new direction for the NPC is calculated by combining (i.e., subtracting) the position of the NPC and the position of the waypoint.
- The function **move_and_slide** is used to move the NPC.
- Finally, we ensure that the NPC is looking forward by using the function **look_at**.

Now that the code has been created for the patrolling state, we just need to create the waypoints:

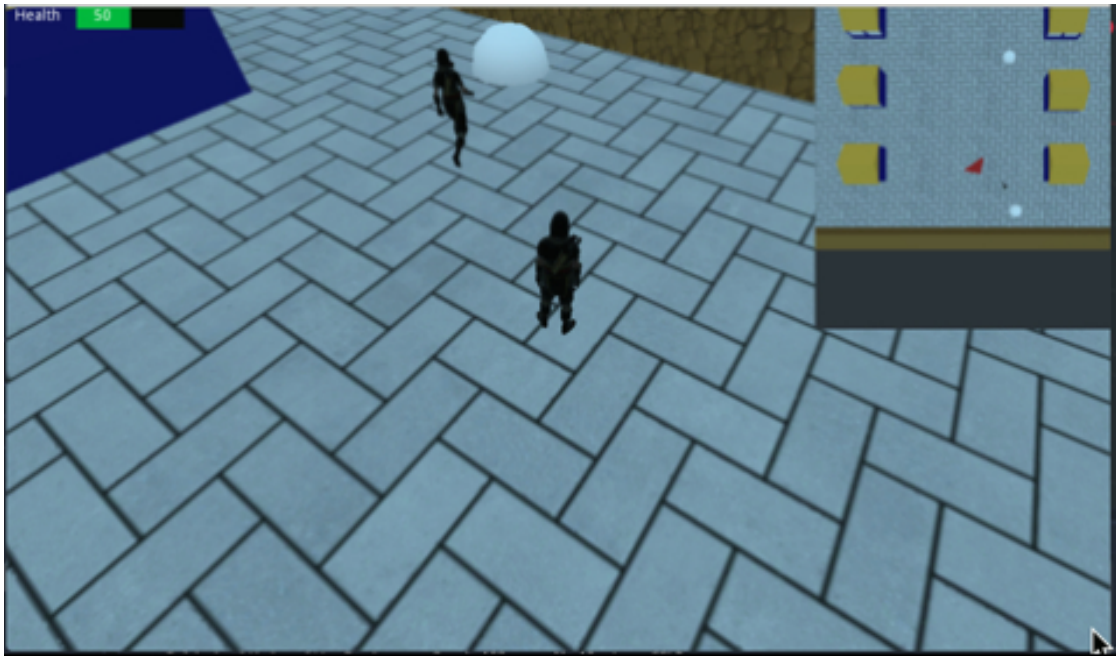
- Please save your code.
- Open (or switch to) the main scene (i.e., **level1**).
- Create a new node of the type **CSGSphere** as a child of the node **Spatial**.
- Rename this node **WP1**, and change its **y** coordinate to **.5**.
- Duplicate this node **3** times (**CTRL/CMD + D**), this will create **3** nodes named **WP2**, **WP3**, and **WP4**.
- Move each of the waypoints so that they form a path as illustrated in the next figure



- Make sure that each waypoint is just above the ground.
- Move the NPC close to the first waypoint as per the next figure.



- Play the scene, you should see that the NPC navigates to each waypoint point after avoiding the houses and the shop.



If you want to hide the waypoints while the game is running, you can also do

the following:

- Select the node **WP1**.
- Attach a new script called **hide_waypoint** to this node.
- Add this code to the script (new code in bold):

```
func _ready():
hide()
```

- Save the script and attach it to the other waypoints (i.e., **WP2**, **WP3**, and **WP4**).

So at this stage, the NPC can navigate to the different waypoints that we have defined; the next phase will consist in enabling the NPC to detect, to chase and to attack the player. This will be done through the creation of two additional states (**CHASE_PLAYER** and **ATTACK**) along with the ability for the NPC to detect the player by seeing or hearing him/her.

So let's proceed:

- Please open the script **manage_npc_guard**.
- Add this code at the beginning of the script (new code in bold).

```
enum {IDLE, PATROL, CHASE_PLAYER, ATTACK}
```

- Add this code just before the function **_ready**.

```
var ray:RayCast
var param_can_see_player = false
var param_can_hear_player = false
```

In the previous code:

- We create a variable of the type **RayCast** that will be used to detect the object in front of the NPC, including the player.
- We also create two variables **param_can_see_player** and

param_can_hear_player.

- These variables are used to specify whether the **NPC** can see or hear the player.

Now that we have declared these variables, we can initialize the ray.

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

```
ray = RayCast.new()
ray.enabled = true
add_child(ray)
ray.global_transform.origin += Vector3.UP
ray.cast_to = Vector3(0,0,100)
```

In the previous code:

- We instantiate a new ray.
- We enable/activate the ray.
- We add the ray to the node attached to the script (i.e., the NPC).
- We make sure that the ray is 1 meter above the ground.
- The ray is pointing **100** meters forward (i.e., along the z-axis).

Now that the ray has been set up, we can use it to detect the player.

- Please add the following code at the beginning of the function **_process**.

```
if (global_transform.origin.distance_to(player.global_transform.origin) < 4):
param_can_hear_player = true
else:
param_can_hear_player = false
```

In the previous code:

- We check whether the distance between the player and the NPC is less than **4**.
- If that's the case the variable **param_can_hear_player** is set to **true**.

- Otherwise, the variable **param_can_hear_player** is set to false.

We can now detect whether the NPC can see the player:

- Please add the following code just after the previous code (i.e., in the function **_process**):

```
if ray.is_colliding():
var obj = ray.get_collider()
if (obj.name == "player"):
param_can_see_player = true
else:
param_can_see_player = false
```

In the previous code:

- We check whether the **ray** is colliding with another object.
- If the player is colliding with the ray then the variable **param_can_see_player** is set to **true**.
- Otherwise, this variable is set to **false**.

Now that the detection has been implemented, we just need to make sure that the player starts chasing the player whenever it has detected the latter; we also need to make sure that he NPC, when in the state **CHASE_PLAYER**, follows the player.

- Please add the following code at the end of the function **_process**, within the code for the state **PATROL** (new code in bold).

```
move_and_slide(direction.normalized()*speed, Vector3.UP)
look_at(global_transform.origin - direction, Vector3.UP)
if (param_can_see_player or param_can_hear_player): current_state = CHASE_PLAYER
```

- Please add this code at the end of the function **_process**; the keyword **CHASE_PLAYER** will need to be at the same level of indentation as the

keyword **PATROL** (new code in bold).

if (param_can_see_player or param_can_hear_player): current_state = CHASE_-PLAYER

CHASE_PLAYER:

var direction = (player.transform.origin - transform.origin).normalized() * 3;

move_and_slide(direction.normalized()*speed, Vector3.UP)

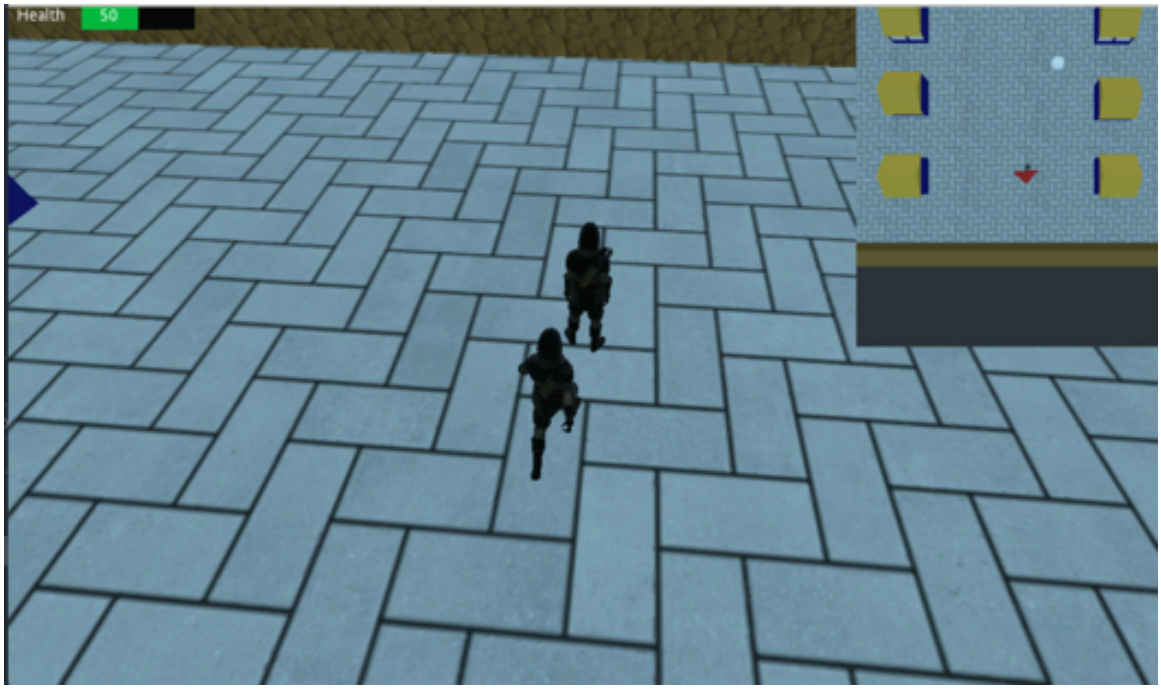
look_at(global_transform.origin - direction, Vector3.UP)

get_node("NPC/AnimationPlayer").play("walking")

In the previous code:

- We create a new state called **CHASE_PLAYER**.
- The directions of the NPC is calculated based on the position of the player.
- We move the **NPC** forward (towards the player) ensuring that the **NPC** is always facing the player.

You can now save your code and play the scene, and as you get closer to the **NPC**, it will start to follow you.



Now that we have ensured that the NPC can detect and chase the player, we just need to make sure that the NPC attacks the player when its near enough.

- Please add this code before the function **_ready** in the script **manage_npc_guard**.

```
var param_is_close_to_player = false
```

- Add this code to the function **_ready**.

```
get_node("NPC/AnimationPlayer").get_animation("sword-and-shield-slash").loop = false
```

- Add this code at the beginning of the function **_process**.

```
if (global_transform.origin.distance_to(player.global_transform.origin) < 2):
param_is_close_to_player = true
else :
param_is_close_to_player = false
```

- Add this code to the function **_process**, in the section that corresponds to the state **CHASE_PLAYER** (new code in bold):

```
get_node("NPC/AnimationPlayer").play("walking")
if (param_is_close_to_player):
current_state = ATTACK
```

Finally, we just need to add the code that defines what the NPC should in the state called **ATTACK**.

- Please add this code to the function **_process**, within the **match** structure, ensuring that the keyword **ATTACK** is aligned with (or at the same level of indentation as) the keyword **CHASE_PLAYER**:

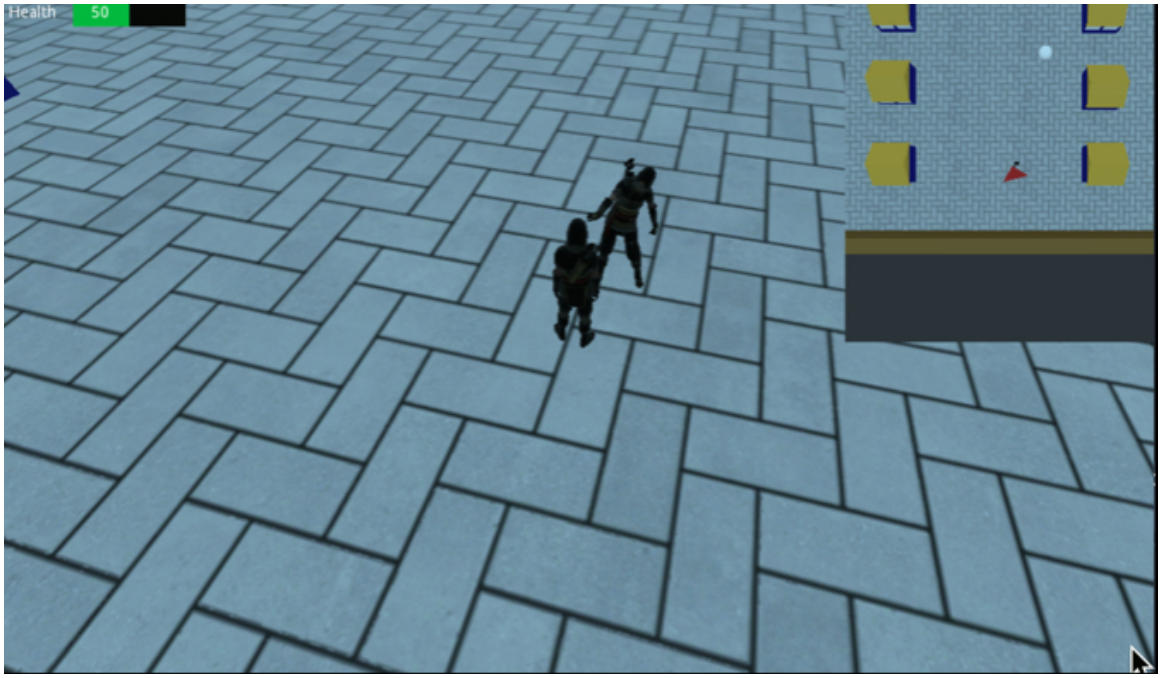
```
ATTACK:
if (!param_is_close_to_player):
current_state = CHASE_PLAYER
look_at(get_node("../Player").global_transform.origin, Vector3.UP)
```

rotate(Vector3.UP, 3.18)

get_node("NPC/AnimationPlayer").play("sword-and-shield-slash")

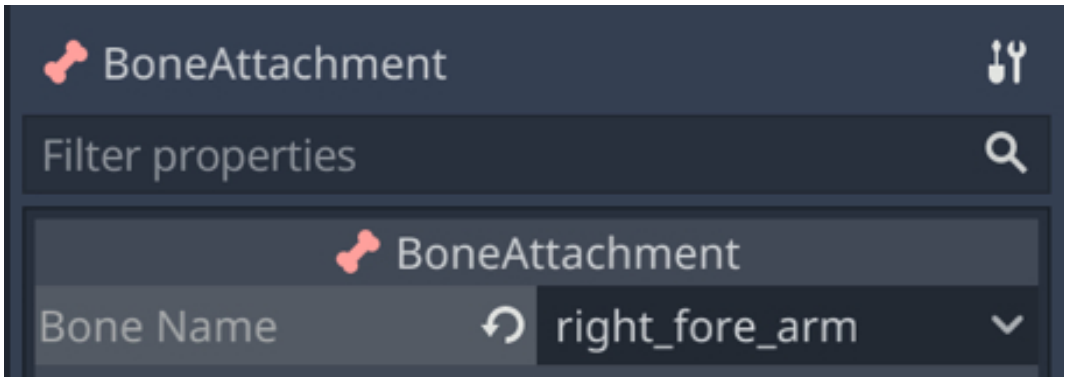
In the previous code, we check that the player is still close to the **NPC** and we play the **attack** animation; otherwise, the current state is changed to **CHASE_PLAYER**.

You can test the scene and you should see that the NPC starts to attack the player when the player is within **2** meters.



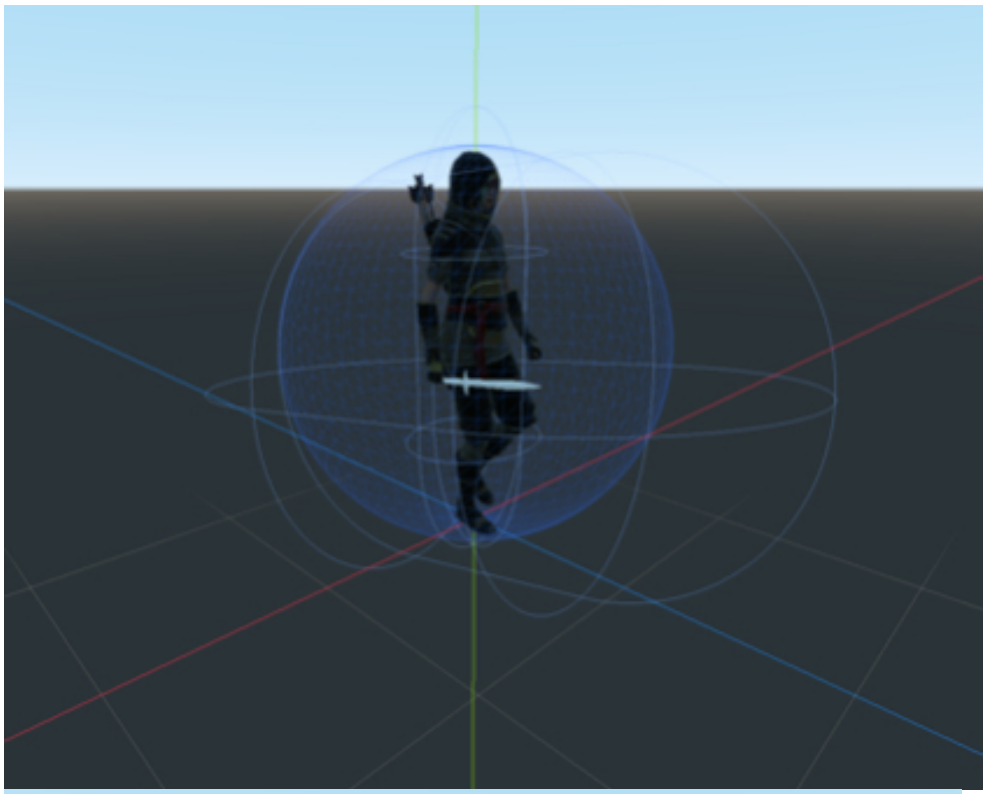
Note that the NPC has no weapon yet, so we could fix this as follows:

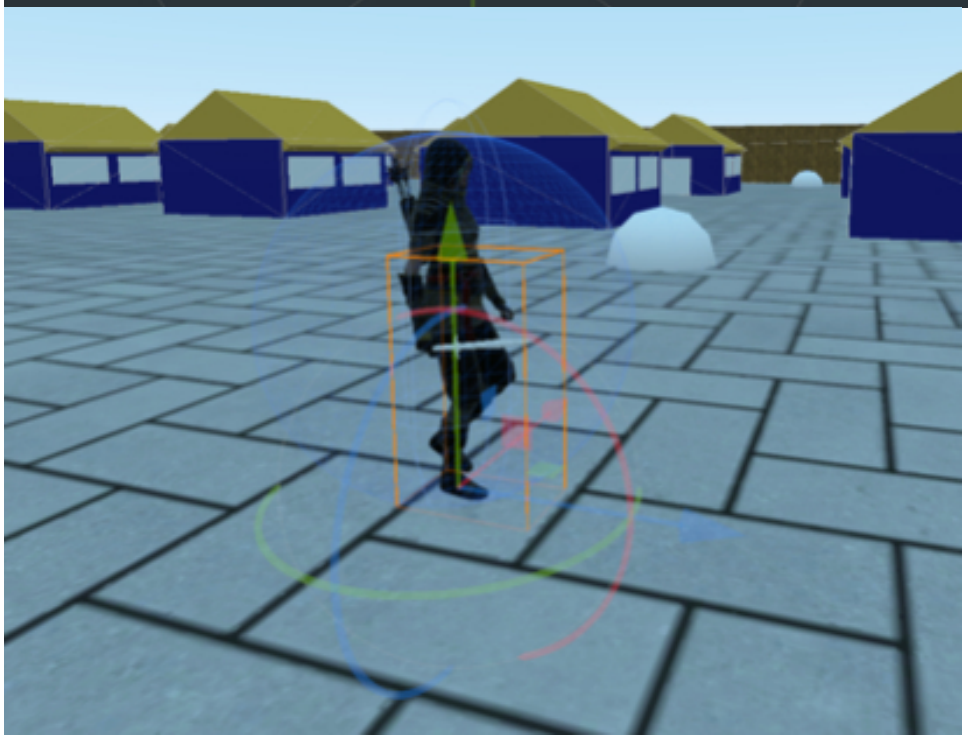
- Please open the scene **npc_guard** that you have created previously.
- Right-click on the node **NPC** and select the option **Editable Children**.
- Select the node **Skeleton** that is a child of the node **NPC**.
- Adda a node of the type **BoneAttachment** as a child of the node **Skeleton**
- Change its **position** to **(-25, -9, -128)**, and its **rotation** to **(-76, 118, -33)**.
- Select the node **BoneAttachement**, and using the **Inspector**, click to the right of the label **Bone Name** (in the section **Bone Attachment**), and select the option **right_fore_arm** from the drop-down list.



- Add a node of the type **Spatial** as child of the node **BoneAttachment** and rename the new node **npc_weapon**.
- If you have not done it already, please import the object **sword.fbx** from the resource pack (i.e., drag and drop from the resource pack to the folder **FileSystem**).
- Drag the asset **sword.fbx** from the tab **FileSystem** atop the node **npc_weapon**; this will create a new node called **sword** as a child of the node **player_weapon**.
- Change its **position** to **(-42, -4, 31)**, its **rotation** to **(0, 180, -180)** and its **scale** to **(100, 100, 100)**.

You can now save the scene **npc_guard**, and you should now see that the sword has been added to the NPC's right hand in both the scene **npc_guard** and the main scene (i.e., **level1**).





Now that the attacking animation is working, we will ensure that the NPC can inflict damage to the player every time it's attacking.

- Please open (or switch to) the script **manage_npc_guard**.

- Add this code before the function **_ready**.

```
var attack_timer:Timer
var can_inflict_damage = true
```

- Add this code to the function **_ready**:

```
attack_timer=Timer.new()
add_child(attack_timer)
attack_timer.wait_time = 1.7
attack_timer.connect(("timeout"),self,"enable_attacking")
```

In the previous code:

- We create a new timer that will be used to make sure that the NPC can inflict damage only once per attack.
- The time is added to the current node.
- Its duration will be **1.7** second (the duration of the attack animation).
- Once this time has elapsed, the function **enable_attacking** will be called.

We can now add the code that will make it possible to inflict damage.

- Please add the following code to the function **_process** in the section related to the state **CHASE_PLAYER** (new code in bold).

```
get_node("NPC/AnimationPlayer").play("walking")
if (param_is_close_to_player):
current_state = ATTACK
can_inflict_damage = true
```

- Please add the following code to the function **_process** in the section related to the state **ATTACK** (new code in bold).

```
get_node("NPC/AnimationPlayer").play("sword-and-shield-slash")
if (can_inflict_damage):
```

get_node("../Player").got_hit()

attack_timer.start()

can_inflict_damage = false

if (!param_is_close_to_player):

current_state = CHASE_PLAYER

In the previous code:

- We check whether the NPC can inflict damage.
- If that is the case, we call the function **got_hit** that is part of the script attached to the **Player** node.
- We then start the timer and set the variable **can_inflict_damage**; this is so that the NPC inflicts damage only once during the attack.
- If the NPC is no longer close to the player, then its state is changed to **CHASE_PLAYER**.

So now we just need to create the function that enables attacking and another one to decrease the player's energy.

- Please add this function at the end of the script **manage_npc_guard**.

func enable_attacking():

can_inflict_damage = true

- Open (or switch to) the script **manage_player**.
- Add this code at the beginning of the script.

var nb_lives = 3

var initial_position:Vector3;

- Add this code to the function **_ready** in the same script.

initial_position = transform.origin

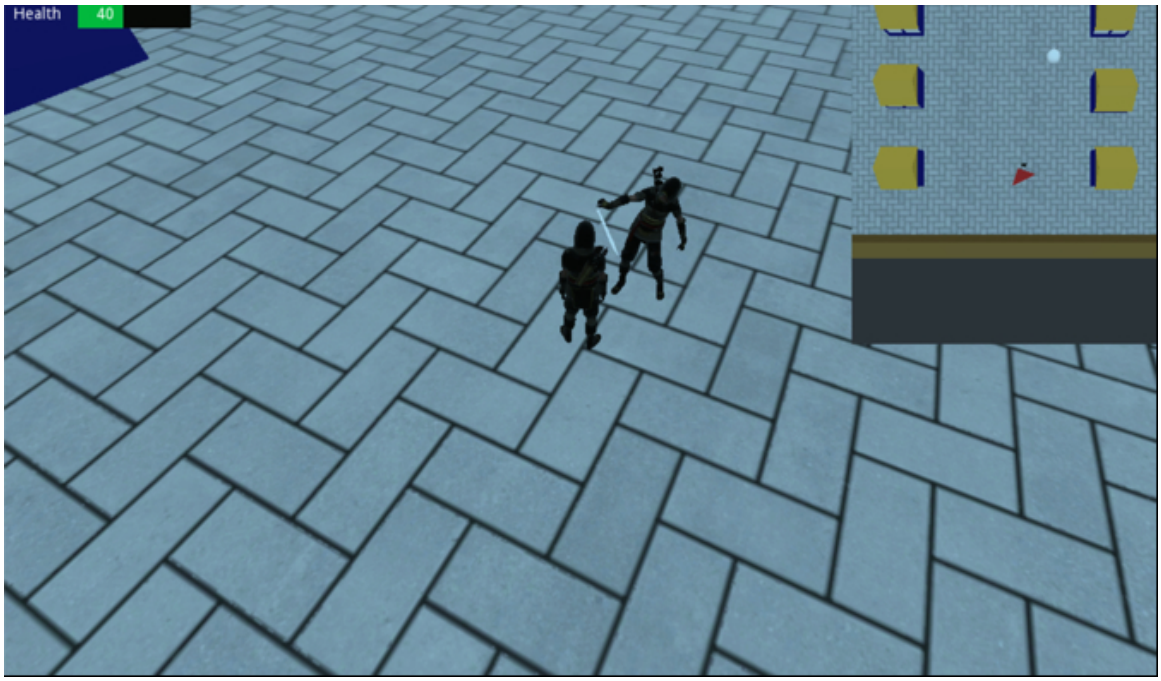
- Add this function to the script:

```
func got_hit():
health -= 5
if (health <= 0):
health = 100
nb_lives -=1
if (nb_lives == 0): print ("Game Over");
transform.origin = initial_position
get_node("../health_bar").set_value(health)
```

In the previous code:

- We create the function **got_hit**.
- In that function we decrease the health of the player by **5**.
- If the variable health is lesser than or equal to **0**, we set tit to **100**, we decrease the number of lives by **1**, and we move the player to its original position.
- For the time-being, if the player has no lives left, we print a message in the **Console** window; however, this can be (and will) be changed so that a game-over screen is displayed instead.

You can now test your scene, and check that upon being attacked by the NPC, the player's health decreases, and that it is ultimately moved to its original position after its health has reached **0**.



Now that the NPC's attack is working, we can look into inflicting damage to the NPC.

- Please open the scene **npc_guard**, select the node **npc_guard**, and, using the tab **Node | Groups**, set its group to **target**.
- Save the scene **npc_guard**.
- Open the script **manage_npc_guard**.
- Add this code before the function **_ready**.

```
var health:int = 100
```

- Add the following functions at the end of the script.

```
func decrease_health(increment:int):
set_health(health-increment)
pass
func set_health(new_health:int):
health = new_health
if (health <=0):
health = 0;
```

```
destroy_target()
func destroy_target():
queue_free()
```

In the previous code:

- We create three functions **set_health**, **decrease_health** and **destroy_target**.
- The function **decrease_health** is called every time the player's sword collides with a node that belongs to the group **target**; when this happens this function calls the function **set_health** to decrease the health of the NPC.
- In the function **set_health**, we set the NPC's health, and if it has reached **0**, we call the function **destroy_target** to eliminate this NPC.

For testing purposes, you can change the health of the NPC to **30** by modifying the following code in the function **manage_npc_guard** (new code in bold).

```
var health:int = 30
```

You can save your code, switch to the main scene (i.e., **level1**) and play the scene; as you attack the NPC several times, it will eventually disappear.

The next step is to add a flash whenever the NPC has been hit.

- Please switch to the script **manage_npc_guard**.
- Add this code before the function **_ready**.

```
var hitTimer:float;
var hit_flash:bool;
var alpha:float;
```

- Add this code to the function **_ready**.

```
alpha = 0.0
var material;
var the_node = get_node("CSGMesh")
material = the_node.material
material.albedo_color = Color(1,0,0,alpha)
```

In the previous code, we ensure that the sphere used for the flash is red and transparent.

- Please modify this code in the function **decrease_health** (new code in bold):

```
func decrease_health(increment:int):
set_health(health-increment)
hit_flash = true;
alpha = 0.5
```

In the previous code, whenever the NPC is hit, we set the variable **hit_flash** to true (so that the fading animation can start for the red sphere) and we also set the alpha value for the sphere to **.5**; so, effectively, at the start of the animation, the red sphere will be semi-opaque and will progressively become fully transparent over time.

- Please add this code at the end of the function **_process** (this code needs to be indented just once so that it is **not** part of the code related to the state **ATTACK**).

```
if (hit_flash):
alpha -= delta
var material;
var the_node = get_node("CSGMesh")
material = the_node.material
material.albedo_color = Color(1,0,0,alpha)
if (alpha <= 0):
hit_flash = false
alpha = 0
```

In the previous code, we decrease the transparency of the sphere until it becomes fully transparent.

You can now save your scene and check that the red flash appears whenever the NPC is hit.

LEVEL ROUNDUP

Summary

In this chapter, we have added a weapon to the player and we made it possible for the player to engage in battles against relatively intelligent NPCs.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are true or false. The solutions are on the next page.

1. It is possible to add objects to an animated character by using a node of the type **BoneAttachement**.
2. The following code will ensure that the animation called **walk** is looping.

```
get_animation("animation").loop = true
```

1. The following code will play the animation called **walk**.

```
play_anim("walk")
```

1. The following code checks whether the key mapped to the action "**walk**" has been pressed.

```
if (Input.is_action_just_released("walk")):
```

1. It is possible to add a group to a node using the tab **Node | Groups**.
2. The following function should always return a Boolean value

```
func is_attacking()->bool:
```

1. It is possible to change the alpha value and color of a node through GDScript.
2. It is possible to define different states for an NPC in GDScript using **enums**.
3. Waypoints can be used to set temporary destinations for an NPC.
4. The function **move_and_slide** can be used to move a node of the type **KinematicBody**.

Solutions to the Quiz

1. TRUE.
2. TRUE.
3. TRUE.
4. FALSE (it checks whether it has been released)
5. TRUE.
6. TRUE
7. TRUE.
8. TRUE.
9. TRUE.
10. TRUE.

Checklist

You can move to the next chapter if you can do the following

- Create states and switch between these.
- Apply an animation to a state.
- Calculate the distance between objects.
- Add an object to an animated character's hand.

Challenge 1

For this challenge, you will need to create another type of NPC that can see the player using **raycasts** and attack the player then.

Chapter 7: Adding a Quest System

So far, we have managed to create all the components of a quest including a player character, objects to collect, as well as NPCs that can engage in a battle with the character. Generally, you would use a combination of these items to create your quests; however, as you create multiple levels, designing and coordinating the different components of your quest might become tedious; sometimes you may want to configure the quest without needing to recode or re-invent the wheel. This is where you will need to use a quest system, a system whereby you can simply configure the player's goals for each level and check that they have been achieved to move to the next level and attribute corresponding XPs to the player.

This chapter offers exactly this, a simple quest system that makes it possible to design and monitor the player's achievements based on pre-defined goals.

After completing this chapter, you will be able to:

- Create a quest system.
- Create and modify a text file that stores all the goals for the player for each level.
- Display this information to the player at the start of each level.
- Automatically notify a game manager when one of these achievements has been reached.
- Automatically attribute corresponding XPs to the player.
- Load the next level when all goals have been achieved.
- Configure each level easily.

Creating a quest system

In this section, we will create a quest system; so far, we have managed to create a level where the player can achieve some significant actions including picking items and adding them to their inventory, defeating enemies (i.e., targets), or talking to an NPC. In most RPGs, the player will also need to complete quests to gain experience; a quest is usually comprised of several tasks, and creating a quest manager will make it possible to:

- Set objectives for each level.
- Set several XPs (e**X**perience **P**oints) to be gained after completing these tasks.
- Check that the tasks have been achieved and reward the player accordingly.

To achieve this purpose, we will be doing as follows:

- We will create a JSON file that will store, for each level, the different tasks to be completed, along with the number of corresponding XPs.
- We will then create a game manager and a quest manager that will read this file and initialize the requirements for the current quest. These tasks will range from picking-up objects to destroying an enemy, entering an area, or talking to a specific NPC.
- We will add the different items that the player need to interact with to complete his/her quest (e.g., an NPC that the player can talk to, an area that s/he can enter, a target that can be destroyed, or an object to be collected).
- Whenever one of the tasks has been completed, a notification will be sent to the game manager, corresponding XPs will be given to the player, and the game manager will check how many of these goals have been completed and whether they correspond to the initial objectives.
- Finally, once all the objectives have been met, the game manager will load a scene where the player can use his/her XPs to increase its attributes such as power, accuracy, or communication.

Saving quest information in an json file

In this section, we will start to create a file that will be used to store information about each of the quests for your game.

- Please create a new folder called **quests**, in your project.
- Please import the file called **quests.json** from the folder **quests** in the resource pack to the folder **quests** in your project.
- If you open this file from the resource pack, you will see the following code:

```
{
"Quest1":
{
"name":"XXX",
"stages":
[
{
"id" : "0",
"name": "Training Camp",
"description":"This is a training camp where you will master your skills",
"results":
[
{"action":"Talk to","target":"Diana","xp" : "100"},
{"action":"Acquire a","target":"Apple","xp" : "100"},
{"action":"Enter place called","target":"yellow_house","xp" : "100"},
{"action":"Destroy one","target":"NPC_Guard","xp" : "100"},
]
}
]
}
}
```

In this file:

- We define the quest called **Quest1**.
- We define a first stage named "**Training Camp**" along with its description.
- We then define the different types of actions that the player will need to complete that stage, along with the associated XPs.

You can of course modify this file later.

In the next section, we will create a script called **QuestSystem** that will be used to read this file and to create quests accordingly.

Creating the game manager

In this section, we will read the JSON file described previously. The goal will be to extract information for a specific stage, including its name, its description, as well as the objectives for the player.

Let's go ahead and create the game manager:

- Please switch to the main scene (i.e., **level1**).
- Create a new node of the type **Node** as a child of the node **Spatial** and rename the new node **game_manager**.
- Select the new node **game_manager** and attach a new script named **game_manager.gd** to it.
- Open the script **game_manager**.
- Add this code at the beginning of the script.

```
var current_stage:int;
var timer_started:bool;
var timer;
var actions;
var targets;
var xps;
var objective_achieved
var xp_achieved
```

In the previous code, we define the current stage (or level); we define the variables **timer_started** and **timer** that will be used to display a message on screen, as well as four arrays: an array for the different actions performed by the user (**actions**), an array for the objectives of each stage (**targets**), an array for the xps associated with each objective (**xps**), along with a list of Boolean values that correspond to whether each objective has been reached (**objective_achieved**) and the number of xp achieved (**xp_achieved**).

- Please add the following code after the previous code (i.e., just before the function **_ready**).

```
var stage_title
var stage_description
var stage_objectives
var starting_point_for_player
var panel_displayed = true
var display_timer
var start_display_timer
```

In the previous code, we define the variables that will be used to display information about the current stage (i.e., **stage_title**, **stage_description**, **stage_objectives**, **starting_point_for_player**), a Boolean variable **panel_displayed** and a float variable called **display_timer** that will be used to time the information displayed on screen.

- Please add the following code after the code you typed:

```
var nb_objectives_achieved = 0;
var nb_objectives_to_achieve = 0
var xb_achieved
```

The previous variables will be used to store how many objectives were reached, how many were to be reached, as well las the number of xps gained for a specific stage.

- Add the following code just before the function **_ready**.

```
enum possible_actions { do_nothing=0, talk_to= 1, acquire_a = 2, destroy_one= 3,
enter_place_called = 4};
var actions_for_quest;
onready var stage_panel = get_node("../stage_panel")
onready var stage_title_text = get_node("../stage_panel/stage_title")
onready var stage_description_text = get_node("../stage_panel/stage_de-scrip-tion")
```

onready var stage_objectives_text = get_node("../stage_panel/stage_ob-jec-tives")

var quests:Dictionary

In the previous code, we create an **enum** variable called **possible_actions** which is a list of the actions that can be performed by the player; we then declare a list of actions called **actions_for_quest** that will include all the actions that the player will have to perform to complete a stage. Finally, we declare four nodes that will be used to display information about the current quest (and that we yet have to create). We then declare a variable called **quests** which is an array that will be used to read the quest(s) from the JSON file.

- Add this code to the function **_ready**:

```
nb_objectives_achieved = 0;
actions = []
targets = []
xps = []
objective_achieved = []
actions_for_quest = []
actions = []
xp_achieved = 0
load_quest();
```

In the previous code, we just initialized the variables that will be used to keep track of the player's achievements. We then call the method **load_quest** that we will define in the next section.

- Please add this function to the same script.

```
func load_quest():
var file = File.new()
file.open("res://quests/quests.json", file.READ)
var json_data = parse_json(file.get_as_text())
```

```
var json = to_json(json_data)
quests = JSON.parse(json).result
stage_objectives = "For this stage, you need to:\n";
```

In the previous code:

- We initialize the array that will be used to save information about the quest.
- We create a new file that will be used to read the content of the file **quests.json**.
- The content of the file **quests.json** is read and stored in the variable **quests**.
- We then print a message to introduce the stage's objectives.

Now that we have written the code to open the JSON file, we just need to read its content:

- Please add this code to the function **load_quest**.

```
for quest in quests:
stage_title = quests[quest].stages[current_stage].name
stage_description = quests[quest].stages[current_stage].description
print ("Stage: "+stage_title)
print ("Description: "+stage_description)
var counter = 0;
for result in quests[quest].stages[current_stage].results:
var action = quests[quest].stages[current_stage].results[counter].action
var target = quests[quest].stages[current_stage].results[counter].target
var xp = quests[quest].stages[current_stage].results[counter].xp
```

In the previous code, we gather information about the quest and the current stage, including the stage name and the stage description; we then create a loop that will go through all the objective for a specific stage including the action, the target, and the corresponding xp to be gained.

- Please add this code after the previous code (new code in bold):

```
var xp = quests[quest].stages[current_stage].results[counter].xp
actions.push_back(action)
targets.push_back(target)
xps.push_back(xp)
objective_achieved.push_back(false)
stage_objectives += "\n -> "+str(action) + " " + str(target) + " [ " + str(xp) + " XP]"
nb_objectives_to_achieve += 1
counter += 1;
print(stage_objectives)
```

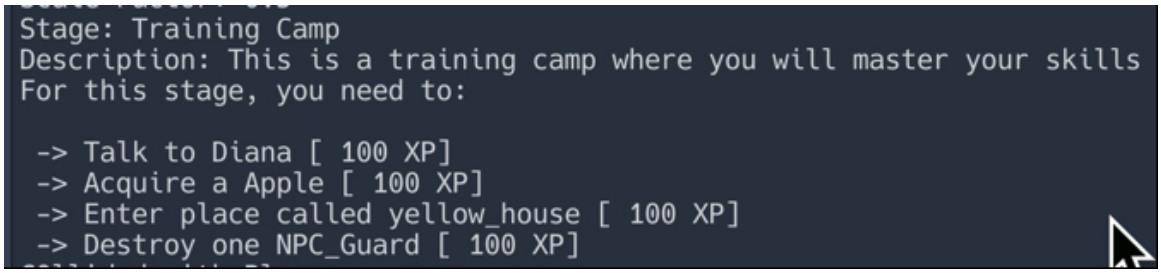
In the previous code, for each action found in the file we do the following:

- Each goal consists of an action and a goal (e.g., talk to X, collect Y, reach Z); so to record a goal, we will need to save both the action (i.e., talk, destroy, reach, collect, etc.) and a target (e.g., character, location or item).
- As a result, for each goal, we save the corresponding action, its target, and its associated number of xps in the corresponding list (e.g., the lists **actions**, **targets**, and **xps**)
- We also specify that, by default, none of the stage's goals are achieved at the start.
- We then construct the string that will be displayed on screen to the player and that lists the different objectives.
- Finally, we increase the number of objectives to achieve.

Before we can display this information in the **Console** window, we just need to call the function **load_quest** from the function **_ready**; so please add this code to the function **_ready**:

```
load_quest()
```

You can now save your code, and play the scene. As the scene is played, please look at the **Console**, and you should see the following message about your quest.



So we have managed to read the JSON file and translate it into a quest, with different required actions and associated xps.

The next step will be to display this information on screen and we will do that in the next section.

Displaying quest information to the user

At this stage, we can read the JSON file and set up a quest. While the information is displayed to the user in the **Console** window, we will, in this section, display the information read from the JSON file on screen.

- Please open the script **game_manager.gd**.
- Add this code to at the end of the script.

```
func init():
display_quest_info()
func display_quest_info():
stage_title_text.text = stage_title
stage_description_text.text = stage_description;
stage_objectives_text.text = stage_objectives + "\n Press H to Hide/Display this information";
```

In the previous code:

- We create the function **init**; this function calls, in turn, the function **display_quest_info**.
- We also create the function **display_quest_info**.
- This function display the stage's **title**, **description** and **objectives**.

Finally, add this code to the function **_ready**.

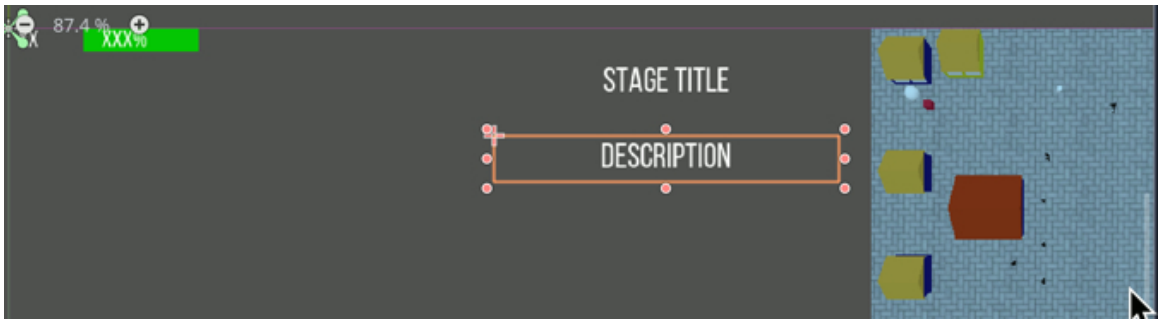
```
init()
```

Next, we will build the interface:

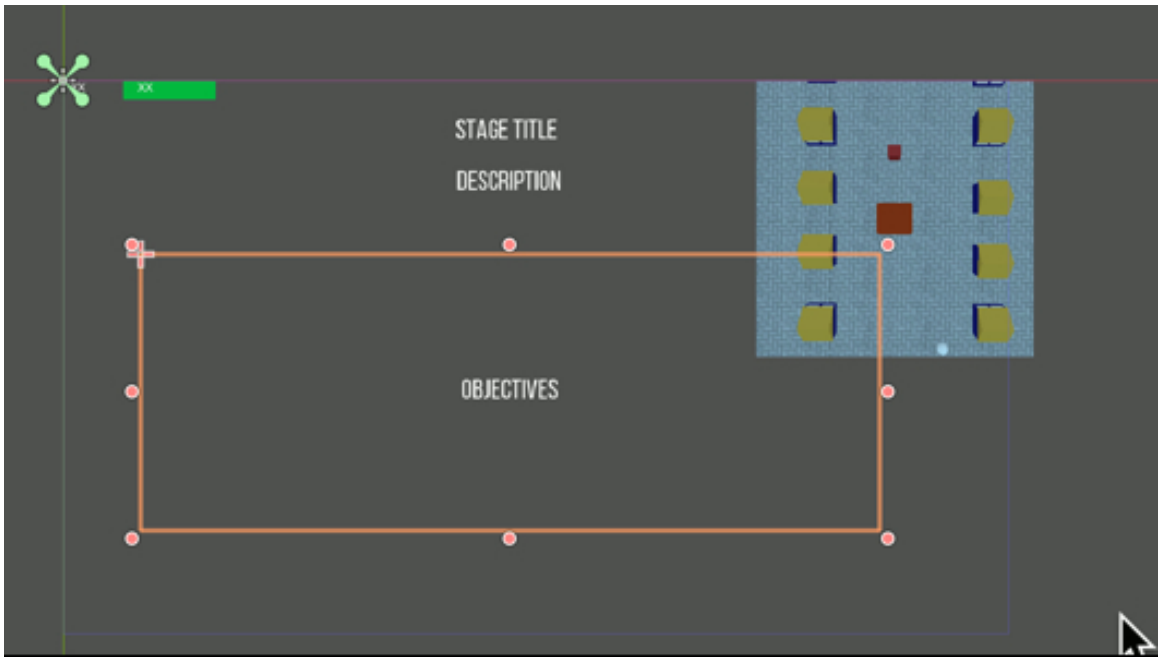
- You can temporarily deactivate/hide the nodes used to display the **shop**, the **inventory**, and the **dialogues** (i.e., deactivate/hide the nodes **shop_ui**, **dialogue_ panel** and **inventory**).
- Please switch to the main scene (i.e., **level1**).
- Create a new node of the type **Node2D** as a child of the node **Spatial** and rename the new node **stage_panel**.
- Create a new node of the type **Label** as a child of the node **stage_panel** and

rename it **stage_title**. Move it to the top of the screen, and, using the **Inspector**, change its **text** to **STAGE TITLE**, change its **vertical** and **horizontal** alignment to **Center**, and its size (i.e., **Rect | Size**) to **(300, 40)**.

- Finally, apply the font **BebasNeue** to this node (**Custom Font | New Dynamic Font**) and change the font size to **30**.
- Duplicate the node **stage_title**, rename the duplicate **stage_description**, change its text to **DESCRIPTION** and move the new node slightly below the first node, as per the next figure.



- Duplicate the node **stage_description**, rename it **stage_objectives**, change its text to **OBJECTIVES**, move it slightly below the node **stage_description**, and change its size (i.e., **Rect | Size**) to **(800, 300)**.



You can now reactivate the objects **shop_ui**, **inventory** and **dialogue_panel**, play the scene and the UI should look like the next figure.

Next, we will make it possible for the player to hide this information after pressing the **H** key.

- Please add this code to the function **Udpate** in the file **QuestSystem**.
- Add this function.

```
func _process(delta):
if (Input.is_action_just_pressed("hide_objectives")):
panel_displayed = ! panel_displayed
if (panel_displayed): stage_panel.show()
else:  stage_panel.hide()
```

In the previous code, we check whether the player has pressed the key mapped to the action "**hide_obectives**" and we then display or hide the panel accordingly.

Now that this is done, we just need to map the action "**hide_obectives**" to the key **H** (i.e., select **Project | Project Settings | Input Map**).

Checking the actions performed by the player

So far, we are only loading information for the current quest and displaying it on screen; however, for our game, we'd like to check if the goals have been achieved for each level; this means translating the required actions into a format/variable that can be measured in the code.

In other words, we need to translate the goals included in the JSON quest file so that they can be interpreted in a way that makes it possible to check, programmatically, whether they have been achieved. For this purpose, we will, for each objective listed in the JSON file, look for keywords and then match them with an item from a list of possible and predefined actions (e.g., reach, collect, destroy, etc.).

- Please open the script **game_manager**, copy and paste the method **load_quest** and rename the duplicate **load_quest2**.
- Remove the code that is after the following lines in the function **load_quest2**:

```
var action = quests[quest].stages[current_stage].results[counter].action
var target = quests[quest].stages[current_stage].results[counter].target
var xp = quests[quest].stages[current_stage].results[counter].xp
```

- Add the following code instead within the for loop (new code in bold):

```
var xp = quests[quest].stages[current_stage].results[counter].xp
var action_for_quest
if (action.find("Acquire",0) >=0):action_for_quest=possible_actions.acquire_a
if (action.find("Talk",0) >=0):action_for_quest=possible_actions.talk_to
if (action.find("Destroy",0) >=0) && action.find("one",0) >=0:
action_for_quest=possible_actions.destroy_one
if (action.find("Enter",0 && action.find("place",0) >=0) >=0):
action_for_quest=possible_actions.enter_place_called
```

In the previous code, we check if the string that describes the action in the JSON file includes the word **Acquire**, **Talk**, **Destroy**, or **Enter** and we set the action

accordingly. This is so that even if you made a typographical error when writing the goals in the JSON file, it is still possible to identify the action based on a keyword present in the string. The new actions are based on the **enum** variable that we have created earlier on.

- Please add the following code after the previous code:

```
actions_for_quest.push_back(action_for_quest)
targets.push_back(target)
xps.push_back(xp)
objective_achieved.push_back(false)
print(str(action) + " " + str(target) + "[" + str(xp) + " XP]");
stage_objectives += "\n -> " + str(action) + " " + str(target) + " [ " + str(xp) + " XP]";
nb_objectives_to_achieve +=1
counter+=1
```

In the previous code, we add a new action, a target, and the corresponding XPs, and we specify that this goal has not been achieved yet. The rest of the code is the same as the previous function.

- In the **_ready** method, replace this code:

load_quest()

...with this code...

load_quest2()

Adding a notification system to check achievements

At this stage, while we can now quantify the objectives that need to be achieved by the player, we need to account for the achievements from the player based on the objectives for the current level, so we will create a system whereby:

- When an objective has been reached, the game manager will be notified.
- XPs will be allocated accordingly to the player.
- When all objectives have been reached, the player will end the level and be redirected to a menu where the XPs gained in the level can be used to improve the characters' attributes such as accuracy, power or communication.

Let's create the notification system:

- Please add the following function to the script **game_manager**.

```
func notify(var action, var target:String):
print("Notified: Action=" + str(action) + " Target:" + str(target));
for i in range (0, actions_for_quest.size()):
if (action == actions_for_quest[i] && target == targets[i] && !objective_achieved[i]):
nb_objectives_achieved += 1
xp_achieved += int(xps[i])
objective_achieved[i] = true;
break
```

In the previous code:

- We create a funtion called **notify**; it takes two parameters: an action and a target; we need these to check whether a goal has been reached, since any goal for a level is identified by an action and a target.
- We then loop through the required actions for a specific level.
- If the parameters passed match a specific goal (i.e., same action and same target), and that this specific goal has not been achieved yet, we consider that this objective or goal is now achieved.

- We then add corresponding XPs and specify that this objective no longer needs to be achieved.

Next, we will need to call this function whenever the player has performed an action. These actions can include: collecting items, reaching an area, talking to a character or destroying an object or an NPC. So first, let's detect when the player collects an object of interest (i.e., an object that should be collected in order to complete the level).

- Please open the script **object_to_be_collected**.
- Add this code before the function **_ready**.

onready var game_manager = get_tree().get_root().get_node("Spatial/game_manager")

- Add the following code (new code in bold) to the method **pick_up_object1**.

func pick_up_object1():
get_node("../Player").item_to_pickup_nearby = true
var can_pick_up = get_node("../Player/in-ventory_system").update_in-tory(type,1)
if (can_pick_up):
get_node("../Player").item_to_pickup_nearby = false
get_node("../user_message").hide()
get_node("../user_message").text = ""
game_manager.notify(game_manager.possible_actions.acquire_a, str(item.-name));
queue_free()

In the previous code: we notify the **game manager** whenever an object has been collected. For this purpose, we call the method **notify** and pass the name of the object collected, along with the type of action just performed by the user (in our case it is **acquire_a**).

- Please save your code, play the scene, collect an apple, and the following message should appear in the **Console** window.

Notified: Action=2 Target:Apple

If you can see this message, then it means that the notification system works.

Next, we will notify the quest system when the player talks to an NPC.

- Please open the script **DialogueSystem**.
- Add this code before the function **_ready**.

onready var game_manager = get_tree().get_root().get_node("Spatial/game_manager")

- Add this code to it to the function **_process** (new code in bold).

else:

dialogue_is_active = false;

dialogue_panel.hide(); #new code

waiting_for_user_input = true;

current_dialogue_index = 0

player.end_talking();

game_manager.notify(game_manager.possible_actions.talk_to, name);

In the previous code, similarly to the item collection notification, we notify the quest system when we talk to a character by passing the name of the character along with the type of action performed (i.e., **talk_to**)

Please save your code and play the scene, after talking to the character called **Diana** you should see the following message in the **Console** window:

Notified: Action=1 Target:Diana

Thirdly, we will notify the quest system when the player has defeated or "destroyed" an NPC called **npc_guard**.

- Please open the script called **manage_npc_guard** that is attached to the NPC called **npc_guard**.

- Add this code before the function **_ready**.

onready var game_manager = get_tree().get_root().get_node("Spatial/game_manager")

- Modify the method **destroy_target** as follows (new code in bold):

func destroy_target():

game_manager.notify(game_manager.possible_actions.destroy_one, name);

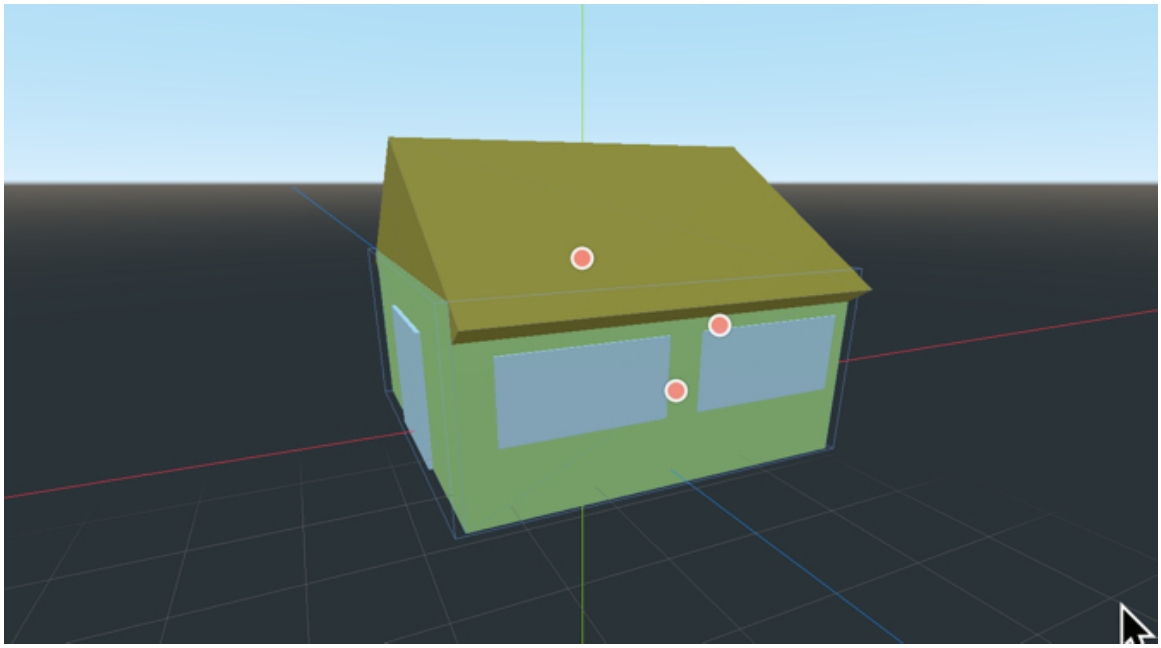
queue_free()

Save your code, play the scene, attack the guard several times; after defeating the guard, the following message should appear in the **Console** window.

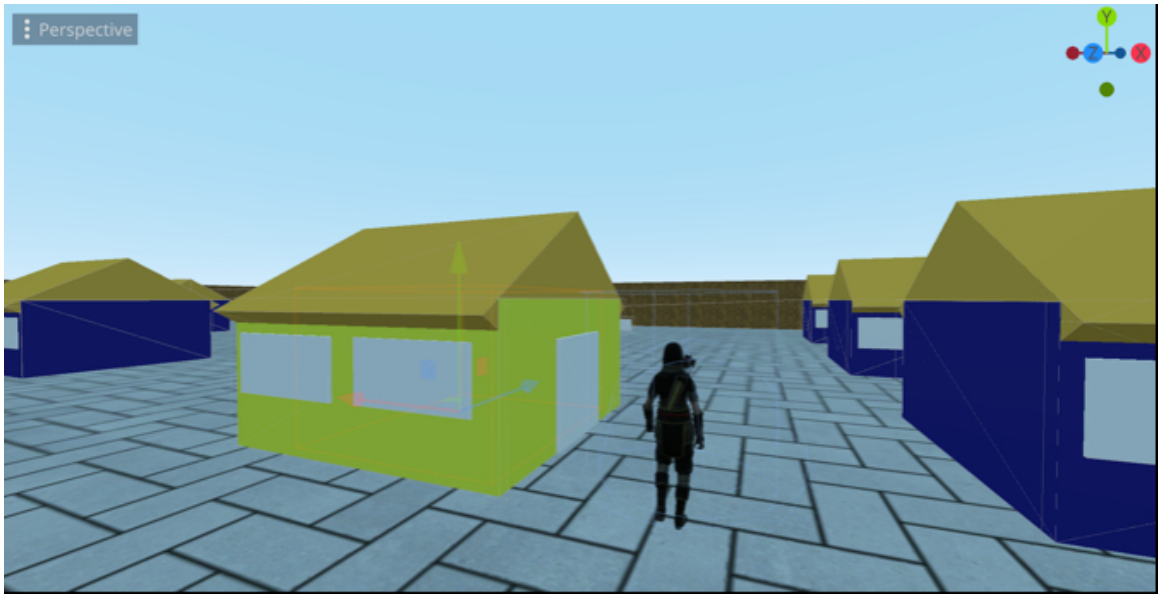
Notified: Action=3 Target:npc_guard

Finally, we will set up the last type of objective which consists in entering a place called **yellow_house**.

- Please duplicate the scene **house.tscn** and rename the duplicate **yellow_house.tscn**.
- Open the scene **yellow_house.tscn**.
- Select the node called "**walls**" and change its color to **yellow**.



- Save the scene.
- Once this is done, switch to the main scene (**level1**).
- Create a new node of the type **Area** as a child of the node **Spatial** and call this node **yellow_house**.
- Create a new node of the type **CollisionShape** as a child of the node **yellow_house**.
- Select this node and, using the **Inspector**, change its scale to **(4.5, 4, 4.5)**, create a new **BoxShape CollisionShape** (i.e., click to the right of the label **Shape**, and select the option **New BoxShape**).
- Right-click on the node called **yellow_house**, choose the option **Instance Child Scene**, select the scene **yellow_house**, and press **Open**. This will create a new node called **house**.
- Please select this node (i.e., **house**), change its scale to **(4, 4, 4)**.
- Select the node **yellow_house**, move it so that it doesn't overlap with other houses, and change its **y** coordinate to **2.5**.



Now that the yellow house has been created and added we need to create a script that will be used to detect and process the event whereby the player enters (or collides with) the yellow house.

- Please select the node **yellow_house** in the main scene.

- Attach a new script called **detect_location** to it.
- Add the following code at the beginning of the script.

onready var game_manager = get_tree().get_root().get_node("Spatial/game_manager")

- Add the following function to the script.

func player_entered(body):
if (body.name == "Player"):
game_manager.notify(game_manager.possible_actions.enter_place_called, name);

In the previous code, we check that the player is colliding with the yellow house, and, in that case, we notify the game manager.

- Please select the node **yellow_house**, and, using the **Inspector**, select the tab **Node | Signals**, and double-click on the event **body_entered**.
- In the new window, select the node **yellow_house** from the list of nodes, type the text **player_entered** in the field **Receiver Method**, and press **Connect**.
- Please save your code, play the scene, and enter the yellow house; you should see the following message in the **Console** window.

Notified: Action=4 Target:yelllow_house

- So that the number of xp gained is also known to the player, you can add this code to the function **notify** in the script **game_manager** and play the scene again (new code in bold).

for i in range (0, actions_for_quest.size()):
if (action == actions_for_quest[i] && target == targets[i] && !objective_achieved[i]):
print("+" + str(xps[i]) + " XP");
nb_objectives_achieved += 1

Save your code, and play the scene; after colliding with the yellow house the following message should be displayed in the **Console** window.

Notified: Action=4 Target:yellow_house

+10 XP

Displaying the XPs on screen

So far, the player's achievements are displayed in the **Console** window, and it would be better to display them on screen, so we will create a system whereby, whenever the player reaches a goal, the number of corresponding XPs will be displayed on screen for a few seconds before it disappears.

- Please add these functions to the script **game_manager**

```
func display_message(var the_message):
get_node("../user_message").show()
get_node("../user_message").text = the_message
var display_timer = Timer.new()
add_child(display_timer)
display_timer.set_wait_time(4)
display_timer.start()
display_timer.connect("timeout",self,"clear_message")
func clear_message():
get_node("../user_message").text = ""
```

In the previous code:

- We create a new function called **display_messge** that takes one parameter (i.e., the text to be displayed on screen).
- We display the node **user_message** that we ye need to create.
- We display the message.
- We then create a timer that will last for 4 seconds; after that time, the function **clear_message** is called.
- The function **clear_message** just clears the message previously displayed on screen.

Finally, change this code:

```
print("+" + str(xps[i]) + " XP");
```

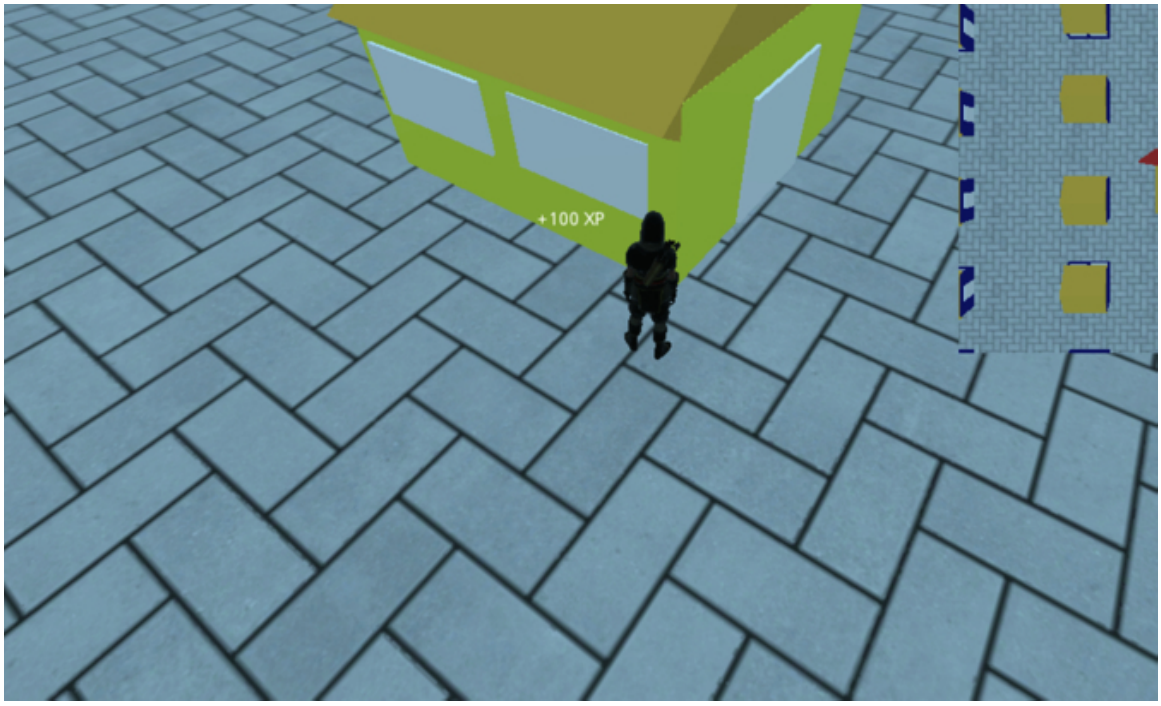
to

```
display_message("+" + str(xps[i]) + " XP");
```

You can now save your code.

Before you can test your code with the guard, please select the node **npc_guard** and change its name to **NPC_Guard** so that it is in line with the name expected as per the **JSON** file.

You can now play the scene, enter the yellow house, and you should see the message **+10 XP** on screen as per the next figure.



If you decide to also defeat the guard, the same message will be displayed. The same will happen if you talk to the character **Dian** or if you collect an apple.

Checking that the player has completed all the tasks and saving data to the game manager

At this stage, we can check whether each of the objectives was achieved and also set the corresponding number of xps for the player. We just need to check whether all the tasks have been completed and then redirect the player to a menu where all the XPs can be used to increase the player character's attributes.

- Please create a new script **PlayerInfo**.
- Remove all the code from the script.
- Add the following code to the script.

```
class_name PlayerInfo
var akai;
var  dreyar;
var name;
var inventory;
var health = 0
var power = 0
var accuracy = 0
var communication = 0
var xp = 0;
enum player_type { Akai = 0, Dreyar = 1 }
```

In the previous code:

- We declare a class called **PlayerInfo** that we will be able to instantiate later.
- We create two variables **akai** and **dreyar** that will be linked to a specific type of NPC, along with a variable called **name** that will be used for the name of the **character**.
- We also define attributes such as health, power, accuracy, communication and XPs.
- We create an **enum** that will list the different types of player characters.

Now that these have been created, we can define a function that will be used as a constructor.

- Please add the following function:

```
func _init(var the_new_type):
if (the_new_type == player_type.Akai):
health = 100;
power = 40;
accuracy = 40;
communication = 40;
xp = 0;
```

In the previous code:

- We create a constructor that takes a parameter which will be the type of Player Character.
- We then create the code to be used in case we want to create an NPC of the type **akai**, and we set its attributes accordingly.
- The idea behind this script (or class) is that, as a designer, you may let the player choose their character for the quest, and also set different attributes for each character, if need be.

Now that the class **PlayerInfo** has been created, we will be able to use it in the game manager, and throughout the game.

- Please add this code in the script **game_manager**, at the beginning, before the function **_ready**.

```
var player_info = PlayerInfo.new(PlayerInfo.player_type.Akai)
```

- Add this code at the end of the function **notify** in the script **game_manager** (new code in bold).

```
func notify(var action, var target:String):
```

```
print("Notified: Action=" + str(action) + " Target:" + str(target));
for i in range (0, actions_for_quest.size()):
if (action == actions_for_quest[i] && target == targets[i] && !objective_achieved[i]):
display_message("+" + str(xps[i]) + " XP");
nb_objectives_achieved += 1
xp_achieved += int(xps[i])
objective_achieved[i] = true;
break
if (nb_objectives_achieved == nb_objectives_to_achieve):
display_message("Stage Complete");
player_info.xp = calculate_total_xp_for_level();
print("XPS for this level"+str(player_info.xp))
```

In the previous code, we display the message "**Stage Complete**", and we calculate the XPs using the function **calculate_total_xp_for_level** (that we yet have to define).

- Please add this function to the script **game_manager**.

```
func calculate_total_xp_for_level():
var total_xp = 0
for i in range (0,actions_for_quest.size()):
total_xp += int(xps[i])
return total_xp
```

In the previous code, we check the XPs related to each of the goals and add them up; the total is then returned.

LEVEL ROUNDUP

Summary

In this chapter, we have created an interesting quest system whereby the objectives for a given level can be written in a JSON file, by the designer or anyone else. We also created a system through which the objectives of each level are displayed at the start of the level, along with a notification system that will check if and when the player has reached all the objectives for the current level.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are true or false. The solutions are on the next page.

1. It is not possible to read a JSON file from Godot.
2. A quest system can be used to identify objectives for a level and check that they have been achieved.
3. The following code will create an array of enums that can be used to identify actions.

enum possible_actions { do_nothing=0, talk_to= 1, acquire_a = 2, destroy_one= 3,

1. The function **file.open** can be used to open and read **JSON** files stored in the project.
2. The function **push_back** can be used to add items to an array.
3. The following node will show the node linked to the variable **panel1**.

stage_panel.show()

1. Messages to the player can be displayed through a **ColorRect** node.
2. The function **get_tree().get_root()** can be used to access the root of the current scene.
3. It is possible to duplicate a scene in Godot
4. The following code, will print the name of the node that has just entered the area linked to this script.

```
func player_entered(body):
print(body.name)
```

Solutions to the Quiz

1. FALSE.
2. TRUE.
3. TRUE
4. TRUE.
5. TRUE.
6. TRUE.
7. FALSE (a node of the type **Label** would be more suitable).
8. TRUE.
9. TRUE.
10. TRUE

Checklist

You can move to the next chapter if you can do the following:

- Create enums.
- Display information on screen.

Challenge 1

For this challenge, you will need to improve the quest system by doing the following:

- Add more objectives to the first level.
- Add the corresponding objects, locations, or NPCs in the scene.
- Check that after completing all the required goals the new scene is loaded.

Chapter 8: Creating an XP Attribution System

This chapter helps you to create an XP attribution system. This means that whenever the player has completed a level and won some experience points, s/he will be able to use them to increase their attributes, in terms of power, accuracy, communication, or any other attributes that you have defined for the player character.

After completing this section, you should be able to:

- Create a user interface.
- Create the logic behind the user interface.
- Control sliders and update data based on the sliders.

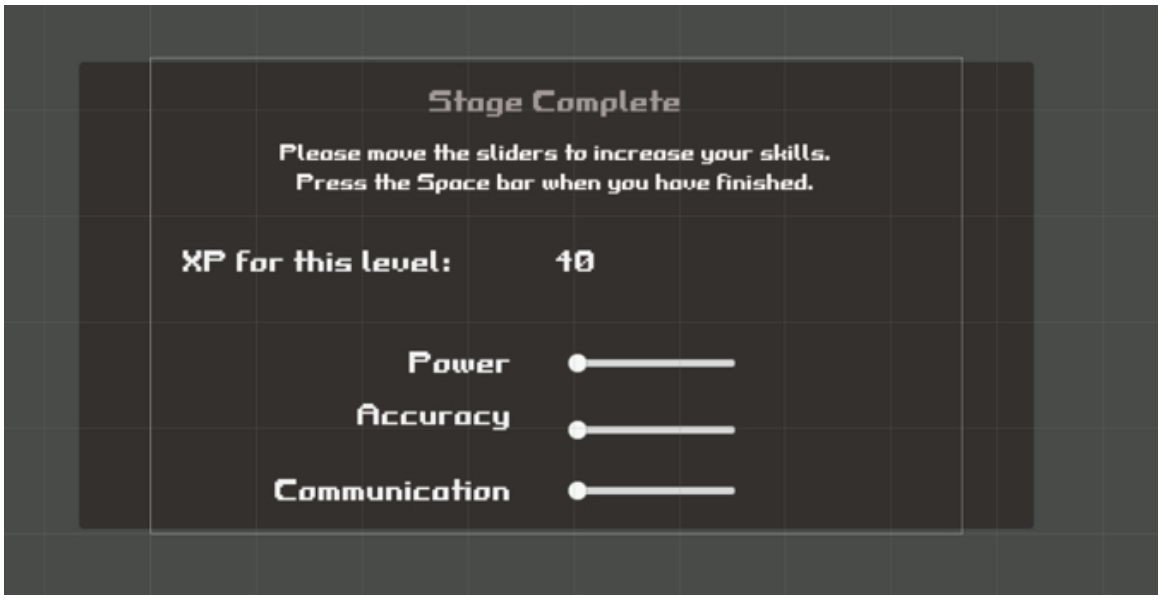
Introduction

In this section, we will start to design a system through which the player can use the xps gained during a specific quest to increase their skills, including power, accuracy, and communication. This is a very common feature in RPG games, as it helps to customize the character and help him/her to grow throughout their quest.

Creating the interface for the XP attribution system

In this section, we will create the interface for the XP attribution system; it will consist of a title, a description, as well as labels and sliders for each of the skills.

- Please create a new scene of the type **Node2D** called **attribute_xp**.
- Open this scene.
- This is the layout that we will be creating along with the logic to increase values.



The players will be able to increase some of their features such as power, accuracy, or communication, using the associated sliders, given the number of XPs gained in the previous level. In any case, if all the XPs gained have been used it will be impossible for the players to increase any of their skills.

- Change the name of the default node **Node2D** to **manage_xp**.
- Add a node of the type **ColorRect** as a child of the node **manage_xp**.
- Using the **Inspector**, change its color to **black**, and its size (i.e., **Rect | Size**) to **(1063, 617)** so that it fills up the screen.

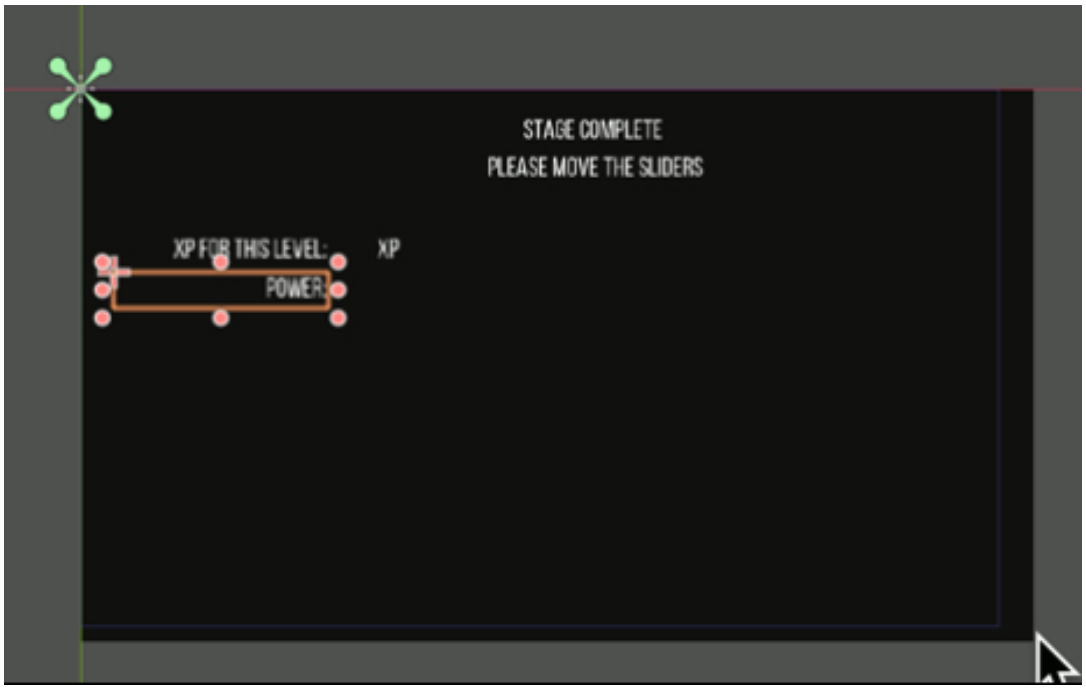
- Add a new node of the type **Label** as a child of the node **manage_xp** and rename this new node **stage_title**.
- Using the **Inspector**, change its text to "**STAGE COMPLETE**", its **vertical** and **horizontal** alignment to **Center**, and its size (i.e., **Rect | Size**) to **(300, 40)**.
- Use the font **BebasNeue** to set a new dynamic font (i.e., **Custom Fonts | New Dynamic Font**) for this node (as we have already done for other **Label** nodes) and set the font size to **30** (i.e., **Custom Fonts | Font | Settings | Size**).
- Move this node (i.e., **stage_title**) at the very top of the window.

- Duplicate the node **stage_title** and rename the duplicate **stage_description**.
- Change the text of the node **stage_description** to "**PLEASE MOVE THE SLIDERS**".
- Move this node below the node **stage_title**.

- Duplicate the node **stage_description**, rename the duplicate **stage_description2**, change its **horizontal alignment** to "**Right**", change its text to **XP FOR THIS LEVEL**, and move this node to the left of the screen, as per the next figure.

- Duplicate the node **stage_description2**, rename the duplicate **text_xp**, change its horizontal alignment to "**Left**", change its text to **XP**, and move this node to the right of the node **stage_description2**.

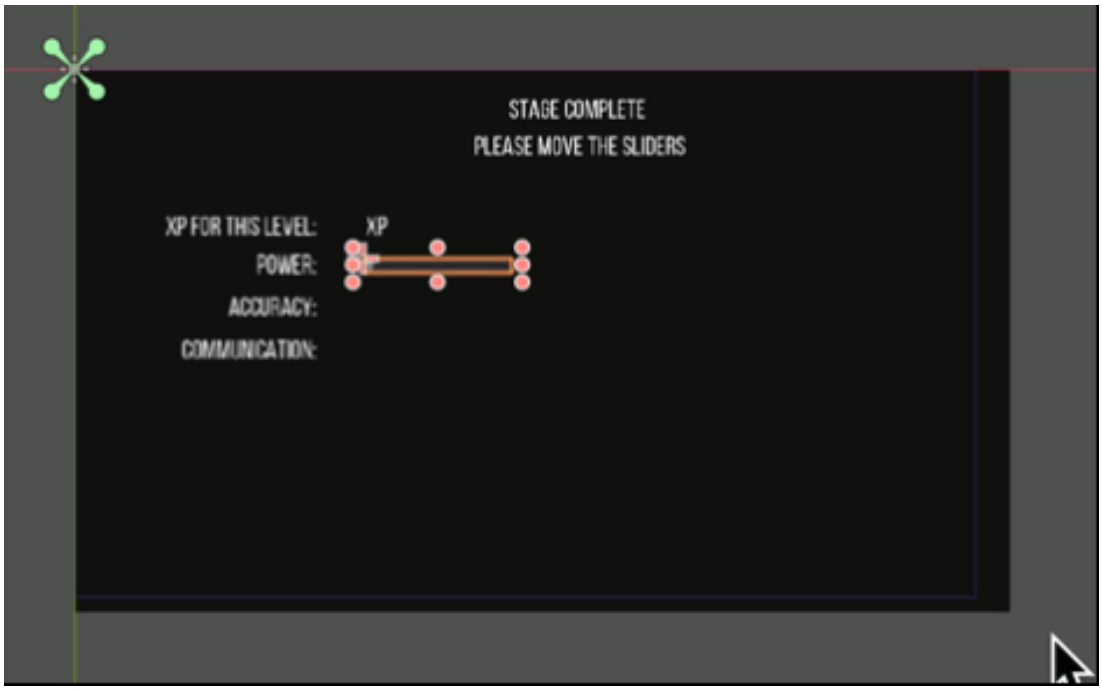
- Duplicate the node **text_description2**, rename the duplicate **power_label**, change its text to "**Power:"**, and move this node below the node **stage_description2**.



- Duplicate the node **power_label** twice, rename the duplicates **accuracy_label** and **communication_label**, change their text respectively to **ACCURACY** and

COMMUNICATION, and move them below the node **power_label**.

- Create a new node of the type **HSlider** as a child of the node **manage_xp**, rename it **power_slider**, change its size (i.e., **Rect | Size**) to **(169, 16)**, and move it to the right of the node **power_label**.



- Duplicate the node **power_slider** twice, rename the duplicates **accuracy_slider** and **communication_slider**, and move them to the right of the nodes **accuracy_label** and **communicaton_label**, respectively.

- The **Hierarchy** view should look like the following figure:

Now that we have designed the interface, we will create the logic to control the sliders:

- Please select the node **manage_xp**, attach a new script to it, and name this script **manage_xp.gd**.
- Open the script **manage_xp.gd**.
- Add the following code at the beginning of the script.

```
var initial_power = 0
var initial_accuracy = 0
var initial_communication = 0
var initial_xp = 0;
var delta_power
var delta_accuracy
var delta_communication
var delta_xp;
```

In the previous code, we initialize the variables that will be used to save the initial, current, and previous (i.e., before a change) values for the attributes **power**,

accuracy, **communication**, and **XPs**. We declare the nodes that will be linked to the sliders, the initial XP, and the game manager; finally, we declare the variables that will be used to measure the change (i.e., delta) in power, accuracy, communication, or XPs after moving a slider.

- Please add this code after the previous code:

```
var current_power
var current_accuracy
var current_communication
var current_xp;
var previous_power
var previous_accuracy
var previous_communication
var previous_xp
```

In the previous code, we initialize the current and previous values for each of the sliders.

- Please add this code after the previous code:

```
onready  var power_slider = get_node("power_slider")
onready  var accuracy_slider = get_node("accuracy_slider")
onready  var communication_slider = get_node("communication_slider")
onready  var power_label = get_node("power_label")
onready  var accuracy_label = get_node("accuracy_label")
onready  var communication_label = get_node("communication_label")
onready var game_manager = get_node("../game_manager")
onready var initial_xp_text_ui = get_node("text_xp")
```

In the previous code, we link the variables **power_slider**, **accuracy_slider**, **communication_slider, power_label**, **accuracy_ label**, and **communication_ label** to the corresponding nodes. We also link the variable **game_manager** to the corresponding node.

- Please add this function:

```
func init():
initial_power = 40
initial_accuracy = 40
initial_communication = 40
initial_xp = 50
```

- Add the following code to the function **init**.

```
current_xp = initial_xp
current_power = initial_power
current_communication = initial_communication
current_accuracy = initial_accuracy
initial_xp_text_ui.text = "" + str(initial_xp)
```

- Add the foloowing code to the fuction **init**:

```
power_slider.set_min(initial_power);
power_slider.set_max(initial_power + initial_xp);
accuracy_slider.set_min(initial_accuracy);
accuracy_slider.set_max(initial_accuracy + initial_xp)
communication_slider.set_min(initial_accuracy)
communication_slider.set_max(initial_communication + initial_xp)
power_label.text = "Power(" + str(current_power) +")";
accuracy_label.text = "Accuracy(" + str(current_accuracy) + ")";
communication_label.text = "Communication(" + str(current_communication)
+ ")"
```

In the previous code:

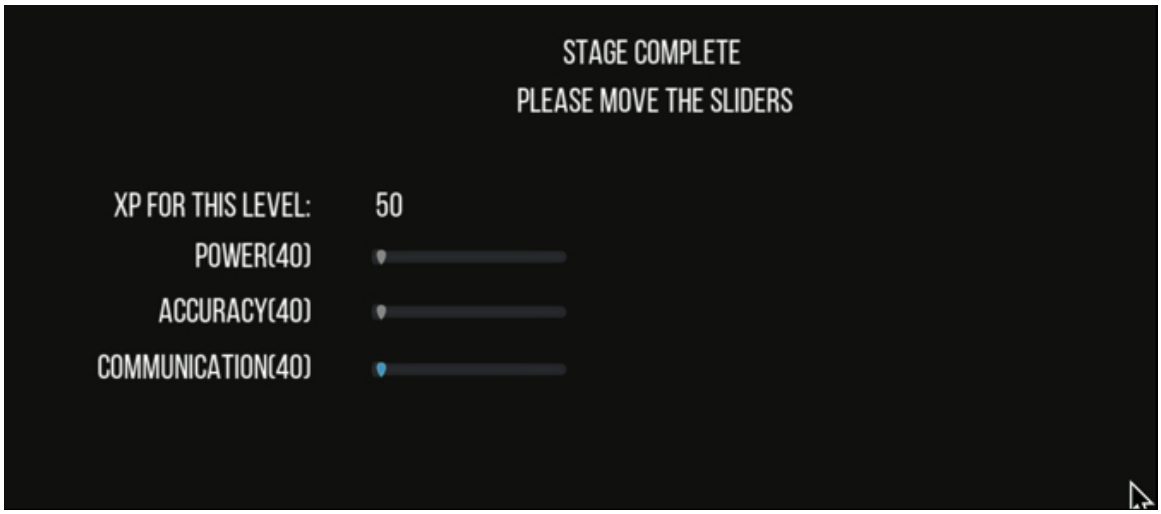
- We set the minimum and maximum values for the sliders associated with the attributes for **power**, **accuracy**, or **communication**.
- In this case, it will only be possible to specify values ranging from the initial

power/accuracy/communication to these values plus the initial XP. In other words, any of the attributes can only be increased by a maximum defined by the initial number of XPs.

Finally, please add this code to the function **_ready** (new code in bold).

```
func _ready():
init()
```

You can now save your code and play the scene **attribute_xp**. You should see that the initial number of XPs is 50.



Next, we will need to check whether the sliders have been moved and to update the interface accordingly. For this purpose, we will need to create several methods that do the following:

- Save the current values (before a change) for each attribute.
- Save the values from the sliders.
- Calculate the difference between the new and old values for each attribute.
- Update the value for XPs.

So let's proceed:

- Please open the script **manage_xp** and add this function.

```
func save_current_values():
```

```
previous_power = current_power
previous_accuracy = current_accuracy
previous_communication = current_communication
```

In the previous code, we save the current values (before the slider is moved) for each attribute.

- Please add the next function.

```
func get_new_values():
current_power = power_slider.get_value()
current_accuracy = accuracy_slider.get_value()
current_communication = communication_slider.get_value()
```

In the previous code, we obtain and store the values from the sliders related to the attributes for accuracy, power, and communication.

- Please add the next function.

```
func calculate_deltas():
delta_power = current_power - initial_power;
delta_accuracy = current_accuracy - initial_accuracy;
delta_communication = current_communication - initial_communication;
delta_xp = delta_accuracy + delta_power + delta_communication;
initial_xp_text_ui.text = "" + str(initial_xp- delta_xp);
```

In the previous code, we calculate the difference between the old and new values for each of the attributes, as well as the number of XPs used in this case. We then update the user interface to display the number of XPs left to be used.

- Please add the next function.

```
func set_xp(value):
print("Set XP" + str(value))
var current_xp_as_text = initial_xp_text_ui.text;
current_xp = int(current_xp_as_text);
```

```
    save_current_values()
    get_new_values()
    if (current_xp == 0):
    if (current_power > previous_power): current_power = previous_power;
    if (current_accuracy > previous_accuracy): current_accuracy = previous_accuracy;
    if (current_communication > previous_communication): current_communication = previous_communication
    power_slider.set_value(current_power);
    accuracy_slider.set_value(current_accuracy);
    communication_slider.set_value(current_communication)
    calculate_deltas();
    power_label.text = "Power(" + str(current_power) +")";
    accuracy_label.text = "Accuracy(" + str(current_accuracy) + ")";
    communication_label.text = "Communication(" + str(current_communication) + ")";
```

In the previous code:

- We read the value in the text field from the current XP and parse (i.e., convert) it to a number.
- We call the method **save_curent_values** and we also read the values from the sliders by calling the method **get_new_values**.
- We check whether there are some XPs left to be able to increase the value of the attributes. If this is not the case (i.e., **currentXP == 0**), and if the user is trying to increase one of its attributes, we ensure that the value for each attribute remains unchanged.
- We call the method **calculate_delta** that will calculate the new values for each attribute. We then update the corresponding labels.

Once this is done, we just need to make sure that the functions that we have created are called whenever a slider is moved.

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

```
power_slider.connect("value_changed",self,"set_xp")
accuracy_slider.connect("value_changed",self,"set_xp")
communication_slider.connect("value_changed",self,"set_xp")
```

In the previous code, we connect the events triggered when a slider is moved (i.e., **value_changed**), to the corresponding function (i.e., **set_xp**).

- Please play the scene and move the sliders, you should see that the values for each attribute are updated accordingly in the user interface.

Once you have checked that this is working, we need to save the information when the player presses the space bar, and also link-up with the actual XP gained by the player after completing a level (at present the XPs are temporarily set to 50).

- Please add this function to the script **manage_xp**.

```
func _process(delta):
if (Input.is_action_just_pressed("validate_space")):
game_manager.player_info.power = current_power
game_manager.player_info.accuracy = current_accuracy;
game_manager.player_info.communication = current_communication
game_manager.player_info.xp = current_xp
```

```
hide()
game_manager.increase_stage(1)
game_manager.load_new_scene()
```

In the previous code, we check whether the key mapped to the action "**validate_space**" was pressed, we save the new values for the player's attributes to the game manager, we increase the index of the current stage (this is so that the quest manager can load the next level), we hide this XP panel, and we call the function **load_scene**.

- Please map the action **validate_space** to the space bar.
- In the **init** method, please replace this code:

```
initial_power = 40
initial_accuracy = 40
initial_communication = 40
initial_xp = 50
```

- with this code...

```
initial_power = game_manager.player_info.power
initial_accuracy = game_manager.player_info.accuracy
initial_communication = game_manager.player_info.communication
initial_xp = game_manager.player_info.xp
```

- Next, please open the script **game_manager.**
- Add the following two functions.

```
func increase_stage(var increment):
current_stage += 1
func load_new_scene():
var all_current_level_nodes = get_tree().get_nodes_in_group("level0")
for  item in all_current_level_nodes:
item.free()
```

In the previous code, we load the next scene based on the current stage (the last level completed before accessing the XP attribution system).

- Please open the script **game_manager**.
- Change the following code in the function **notify**:

```
if (nb_objectives_achieved == nb_objectives_to_achieve):
display_message("Stage Complete");
player_info.xp = calculate_total_xp_for_level();
stage_complete()
```

- Add this function to the script **game_manager**.

```
func stage_complete():
get_node("../manage_xp").init()
get_node("../manage_xp").show()
print("Stage Complete")
print("XPs for Player="+ str(player_info.xp))
print("Loading stage complete scene")
```

- Add this code to the function **_ready**.

```
get_node("../manage_xp").hide()
```

Once this is done, you can do the following:

- Save the scene **attribute_xp**.
- Open the main scene (i.e., **level1**), right-click on the node **Spatial** (i.e., the top-most node), select the option **Instance Child Scene**.
- Select the scene **attribute_xp** and press **Open**, this will create a new node named **manage_xp** as a child of the node **Spatial**.

Last but not least, when loading the new scene, we will just remove the node that belongs to the first scene. This is partly done in the function **load_new_scene**; however, we yet have to add a group to the nodes that we don't want to keep for

the next stage. So please, apply the group **level0** to the nodes **shop target**, and **yellow_ house** using the **Inspector** (i.e., **Nodes | Groups**). We can leave the nodes **WP1**, **WP2**, **WP3**, and **WP4** because they will be used for the next level as well.

- You can now save the scene.
- Play the first scene (i.e., **level1**) and reach all the objectives. You should see the message "**Stage Complete**" displayed on screen.
- Then use the XPs to increase your skills, press the **SPACE** bar, and the scene should be similar except that the shop and the yellow house should have been removed.

LEVEL ROUNDUP

Summary

In this chapter, we have created an XP attribution system whereby the players can gather XPs during their quest and then use them to improve their skills.

Quiz

It is now time to test your knowledge. Please specify whether the following statements are true or false. The solutions are on the next page.

1. It is not possible to obtain the value of a slider from GDScript.
2. In an RPG it is often possible to use XPs to increase the player character's skills.
3. It is possible to set the minimum and maximum values for a slider from GDScript.
4. The event "**value_changed** " is used to detect whenever the user has moved a slider.
5. It is possible to change the text (i.e., value) of a node of the type **Label** from GDScript.
6. The function **set_min** can be used to set the minimum value for a slider.
7. The function **set_max** can be used to set the maximum value for a slider.
8. When using the function **print**, any variable within the round brackets needs to be converted to a string/text value.
9. The following code can check whether the key mapped to the action "**validate_space**" has been pressed.

```
if (Input.is_action_just_pressed("validate_space")):
```

1. The following code will connect the event **value_changed** for the node **power_s lider** to the function **set_xp**.

```
power_slider.connect("value_changed",self,"set_xp")
```

Solutions to the Quiz

1. TRUE.
2. TRUE: it is possible to set the minimum and maximum values for a slider from GDScript.
3. TRUE.
4. TRUE.
5. TRUE
6. TRUE.
7. TRUE.
8. TRUE.
9. TRUE.
10. TRUE.

Chapter 9: Creating The Final Level

In this chapter, we will create the final level for the game where the player will need to collect three different diamonds and avoid guards.

Setting-up the level

In this section we will set up the level by doing the following:

- Keep the current scene.
- Remove the elements from this scene that should not be in the second stage of the mission.
- Add elements that need to present in the second stage of the mission (i.e., guards and diamonds).

First, we will tag the objects that only belong to the first level, by adding them to the group "**levelo**".

- If you haven't already done so, please select the nodes **shop**, **target**, **NPC_Guard**, and **yellow_house**, and using the **Inspector**, add the group "**levelo**" to these nodes.

Next, we need to modify the file **quests.json** so that we can specify the name and objectives of the next level.

Please do the following:

- Using your file system (i.e., outside of Godot), navigate to the folder where the file **quests.json** is located. To know its location, you can right-click on the folder **res://** in Godot, then select the option "**Open in File Manager**", then open the folder **quests**, and the file **quests.json** should be located there.
- Duplicate the file **quests.json** and rename the duplicate **quests_backup.json**.
- Open the file **quests.json**.
- Change this code (i.e., line **10** to line **21**):

```
{
"id" : "0",
"name": "Training Camp",
"description":"This is a training camp where you will master your skills",
"results":
```

```
[
{"action":"Talk to","target":"Diana","xp" : "100"},
{"action":"Acquire a","target":"Apple","xp" : "100"},
{"action":"Enter place called","target":"yellow_house","xp" : "100"},
{"action":"Destroy one","target":"NPC_Guard","xp" : "100"},
]
}
```

... to this code (new code in bold)

```
{
"id" : "0",
"name": "Training Camp",
"description":"This is a training camp where you will master your skills",
"results":
[
{"action":"Talk to","target":"Diana","xp" : "100"},
]
}
```

In the previous code, as we just want to test that the second scene is located after completing all the objectives from the first level, we just use one objective, for testing purposes, to speed up the testing process; so effectively, after talking to the character **Diana**, the XP panel will be displayed, and after allocating XPs to improve different skills, the player should be taken to the second mission.

Next, we need to specify the objectives for the second mission.

- Please add the following code to the file **quests.json** (new code in bold) after the third last closing curly bracket.

```
}
,
{
"id" : "0",
```

"name": "The Thief",

"description":"Avoid the guards and Collect 3 Diamonds",

"results":

[

{"action":"Acquire a","target":"Yellow diamond","xp" : "100"},

{"action":"Acquire a","target":"Red diamond","xp" : "100"},

{"action":"Acquire a","target":"Blue diamond","xp" : "100"},

]

}

]

In the previous code, we specify that for the second mission, the player will need to collect three diamonds (i.e., a yellow diamond, a red diamond, and a blue diamond).

- Please save the file.

At this stage, we have specified the elements that need to be removed from the current scene, along with the objectives for the first and second mission.

Next, we need to make sure that once the player has leveled up its XPs, the new mission is loaded; to do so, we will just reload the objectives from the JSON file and display them as follows:

- Please open the script **game_manager** and add this code to the function **load-_new_scene** (new code in bold).

for item in all_current_level_nodes:

item.free()

load_quest2()

display_quest_info()

panel_displayed = true

stage_panel.show()

In the previous code, we call the function **load_quest2** to load the mission's

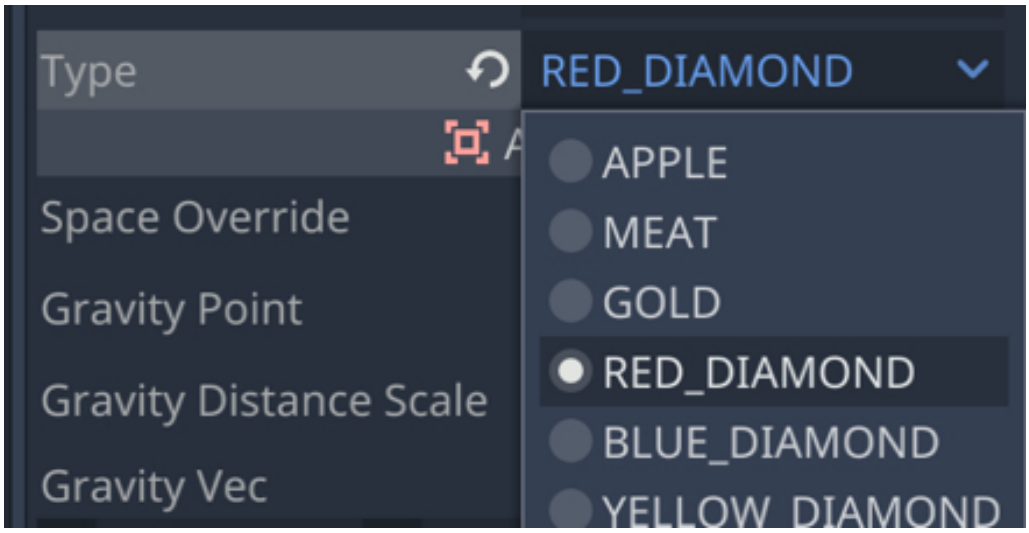
objectives, and we also display the quest's information.

- Please save your code. You can test the scene; after talking to the character **Diana**, the XP attribution panel system should be displayed, and after allocating XPs and pressing the **space bar**, you should see the level again, but this time with new instructions, and without the yellow house, or the initial NPC.

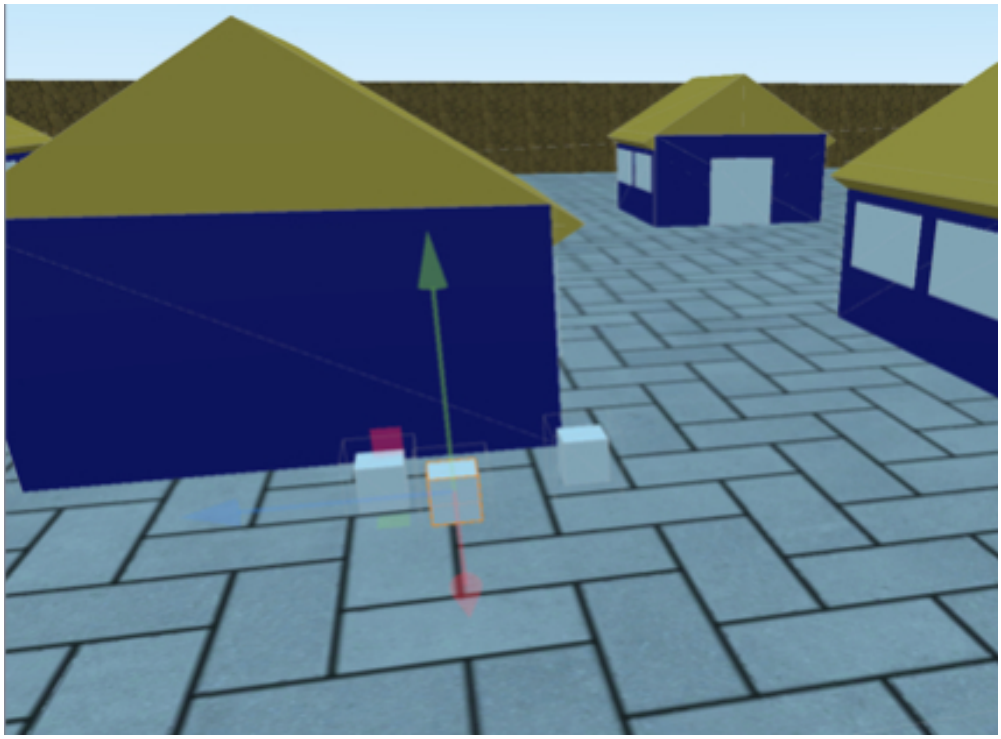
The next steps is to add the elements that will be part of the new mission, namely: the three diamonds and the guards that will be patrolling.

First, let's add the diamonds:

- Please duplicate the node **object_to_collect** (that you have created earlier) three times, rename them **red_diamond**, **yellow_diamond**, and **blue_diamond**, and using the **Inspector**, change their attribute **Type** (in the section **Script Variables**) to **RED_DIAMOND**, **YELLOW_DIAMOND**, and **BLUE_DIAMOND**, respectively.



- Once this is done, move these three nodes apart and make sure that they are above the ground as per the next figure.



- Apply the group **level1** to all these nodes using the **Inspector**.
- Hide these nodes by clicking on the eye item to the right of their names in the **Hierarchy** tree.

Next, we will display these items only when we have reached the mission where they should appear.

- Please open the script **game_manager**.
- Add this code to the function **load_new_scene**.

```
var all_nodes_to_display = get_tree().get_nodes_in_group("level" + str(current_stage))
for item in all_nodes_to_display:
item.show()
```

In the previous code, we display the nodes that should appear in the scene for the current stage.

- Using the **Inspector**, please add the group **level0** to the node **object_to_be_collected**.
- Open the script **object_to_be_collected**.
- Change the following code (new code in bold)

```
func body_entered(body):
# pick_up_object1()
get_node("../Player").item_to_pickup_nearby = true
if (is_in_group("level"+str(game_manager.current_stage)) or is_in_group("level0")):
pick_up_object2()
```

In the previous code, we ensure that the options to collect an item only appears if that item belongs to the current level (i.e., level 0 for the apple, and level 1 for the diamonds).

Next, we will make sure that an end-of-game screen is displayed when the player has completed all the missions defined in the JSON file.

- Please open the script **game_manager.gd**.
- Add the following code at the beginning of the script.

```
var nb_quests = 1
```

- Add the following function to the script **game_manager.gd**.

```
func count_nb_quests():
var file = File.new()
file.open("res://quests/quests.json", file.READ)
var json_data = parse_json(file.get_as_text())
var json = to_json(json_data)
quests = JSON.parse(json).result
for quest in quests:
nb_quests += 1
```

In the previous code, we go through the JSON file and count the number of missions within.

- Please add the following function to the script **game_manager**:

```
func load_end_screen():
get_tree().change_scene("res://end_of_game.tscn")
```

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

```
count_nb_quests()
```

- Open the script **manage_xp.**
- In the function **_process** remove the code after the call to the function **hide**:

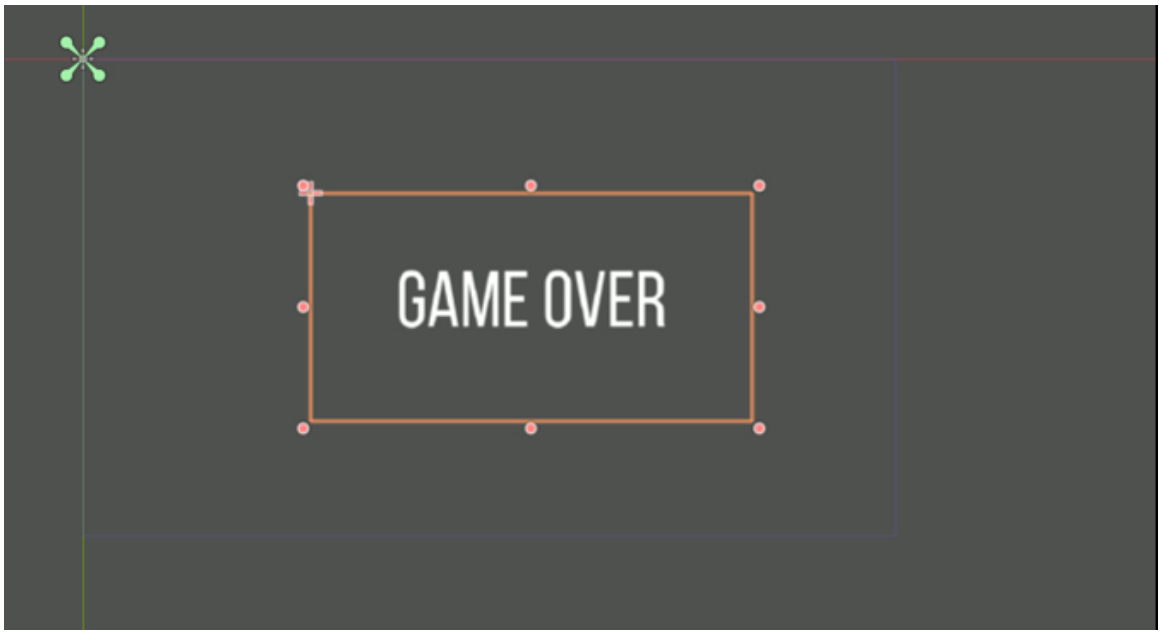
```
#game_manager.increase_stage(1)
#game_manager.load_new_scene()
```

- Add this code instead (new code in bold):

```
hide()
if (game_manager.current_stage == game_manager.nb_quests-1):
game_manager.load_end_screen()
else:
game_manager.increase_stage(1)
game_manager.load_new_scene()
```

In the previous code, once the player has saved its xps, s/he will be taken to the next mission or to the screen for the end of the game, if all missions have been completed.

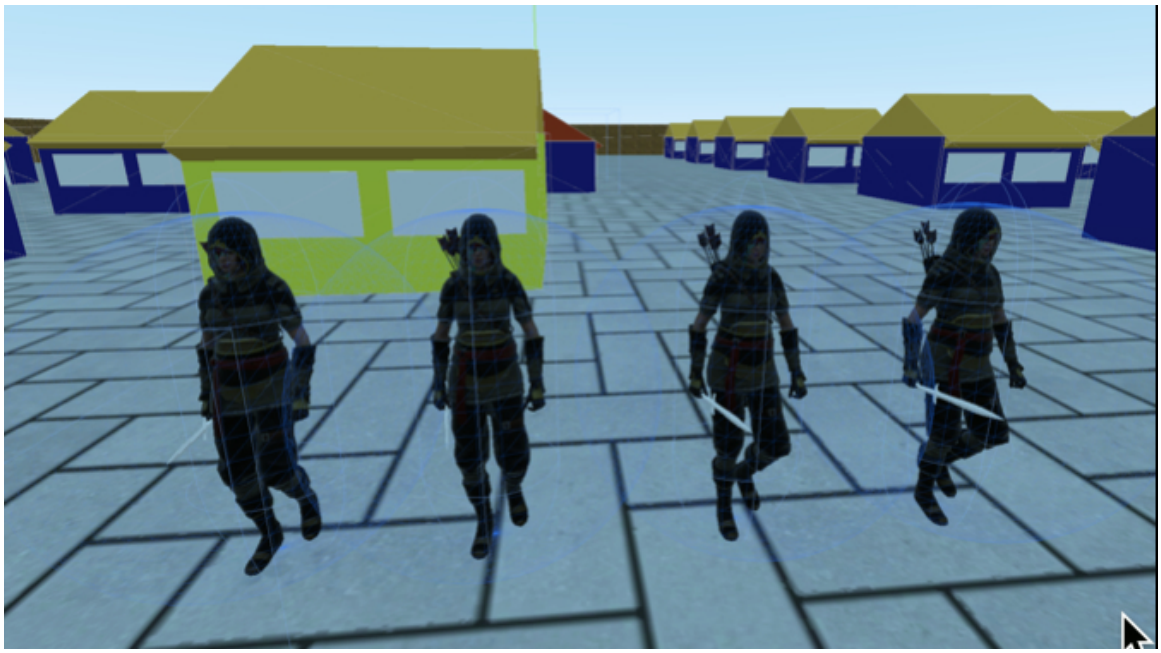
- Finally, please create a new **2D scene** called **end_of_game.tscn**, which includes a label centered on the middle of the screen that says "**Game Over**".



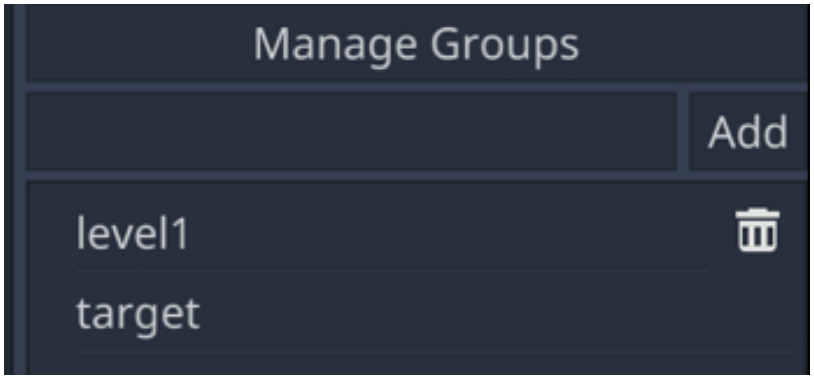
You can now save your changes, switch to the main scene, play the scene, and check that after completing all the missions, the **Game Over** screen is displayed.

Last but not least, we will add several NPCs that will patrol, and chase the player.

- Please duplicate the node **NPC_Guard** three times.
- Move the duplicates apart so that, for example, they form a row or a line.



- Select each duplicate, and using the **Inspector** and the tab **Node | Group**, delete (i.e., select **level0** and press the **bin** icon) the group **level0** and add the group **level1** to these nodes.



- Add this code in the script **manage_npc_guard**, at the beginning of the function **_ready**.

```
var current_stage = "level"+str(game_manager.current_stage);
if (!is_in_group(current_stage)): hide()
```

In the previous code, we check that the NPCs are displayed only for the level they belong to.

- Finally, please add this code at the beginning of the function **_process** in the script **manage_npc_guard**.

```
if (!is_visible()): return
```

In the previous code, if the NPC is not visible (i.e., not to be included in the current scene) we ensure that is not moving.

That's it; you can now save your scene and check that, in the second level, the three NPCs patrol, and chase the player whenever they detect him/her.

LEVEL ROUNDUP

Summary

In this chapter, we completed our last level and we also included additional features that should make the game more challenging.

Checklist

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

- Show or Hide nodes based on their group.
- Modify the file **quests.json** to create new levels for the quest.
- Detect when the player has managed to complete all the quests.

Challenge 1

For this challenge, you will need to:

- Create a new level.
- Modify the file **quests.json** to include new objectives.
- Add the necessary objects to the level.
- Test the level.

Chapter 10: Your Next Steps

This chapter will show you how you can expand the code that you have created to improve your RPG.

CREATE OBJECTS

Create more objects to be used and collected

- Open the file **Item.gd** and add your own objects.
- Set a family for this object (or create a new object family).
- Add the corresponding image to the folders **food, loot**, or **weapons**.
- Add a corresponding scene to the folder **res://**.

Add the objects to your scene

- Duplicate the node **object_to_collect**.
- Set its type using the **Inspector**.

Add characters

- Duplicate the node **NPC_Guard** and set the scene where it should appear by changing its group.

Add a shop

- Instantiate the shop in the main scene.
- Modify the method **init** in the file **shop_system** to add different types of items to the shop.

CREATE NEW QUESTS

Update the file quest.json

- Open the file called **quest.json**.
- Copy/paste the information from an existing quest.
- Modify the attribute **stage_id**.
- Modify the section **results** to specify the goals for a specific level along with the corresponding **XP**s.
- For the quest goals, make sure that you use the right keywords for each action (e.g., "acquire", "talk", "destroy", or "enter").

Add objects for each scene

- Add nodes to the main scene.
- Set the group for these nodes accordingly.

CREATE NEW DIALOGUES

Update the file dialogues.json

- Copy paste the dialogue for an existing character in this file.
- Change the attribute "character name"
- For each dialogue, modify the question and the possible answers.
- Add an image for this NPC (with the same name as the one in the JSON file) in the folder **res://**.

Create and add the corresponding 3D character

- Create a 3D character and rename it using the same name as the one provided in the file **dialogues.json**
- Add an animation to this character if need be.
- Add the script **dialogue_system** to it.

USE NEW SKILLS

Update the skills attribution system

- Create labels and sliders for new skills (e.g., speed).
- Modify the file **manage_xp** to add the corresponding variables.
- Add similar variables to the **game_manager** and the file **PlayerInfo**.
- Use this skill for the player (e.g., to increase its speed).

Add status bars for the player's skills

- Duplicate the node **health_bar**.
- Add it as a child of the node **Spatial**.
- Rename this node and associate it to a different attribute (e.g., power, speed, etc.).
- Access and modify the aspect of the bar using a script; for example:

```
get_node("../health_bar").set_value(health)
```

ADD MORE CHALLENGE

Make the NPCs more challenging

- Increase the number of NPCs.
- Increase the health of each NPC using the **Inspector**.
- Add vision to the NPCs using ray-casting.

Thank you

I would like to thank you for completing this book. I trust that you are now comfortable with RPG creation and that you can create your very own RPG efficiently. If you have enjoyed the book, please leave an honest review on your favorite e-store, this would mean the world to me.

So that the book can be constantly improved, I would really appreciate your feedback and hearing what you have to say. So, please leave me a helpful review in the e-store where you bought the book letting me know what you thought of the book, and also send me an email (**learntocreategames@gmail.com**) with any suggestions you may have. I read and reply to every email.

Thanks so much!!

- Pat

Also by Patrick Felicia

Beginners' Guides

A Beginner's Guide to 2D Platform Games with Unity

A Beginner's Guide to 2D Shooter Games

A Beginner's Guide to Puzzle Games

C# from Zero to Proficiency

C# Programming from Zero to Proficiency (Introduction)

C# Programming from Zero to Proficiency (Beginner)

Getting Started

Getting Started with 3D Animation in Unity

Godot from Zero to Proficiency

Godot from Zero to Proficiency (Foundations)

Godot from Zero to Proficiency (Advanced)

Godot from Zero to Proficiency (Beginner)

Godot from Zero to Proficiency (Intermediate)

Godot from Zero to Proficiency (Proficient)

JavaScript from Zero to Proficiency

JavaScript from Zero to Proficiency (Beginner)

Quick Guides

A Quick Guide to c# with Unity

A Quick Guide to Procedural Levels with Unity

A Quick Guide to 2d Infinite Runners with Unity

A Quick Guide to Artificial Intelligence with Unity

A Quick Guide to Card Games with Unity

Ultimate Guides

The Ultimate Guide to 2D games with Unity

Unity 5 from Proficiency to Mastery

Unity from Proficiency to Mastery (C# Programming)

Unity from Proficiency to Mastery

Unity from Proficiency to Mastery (Artificial Intelligence)

Unity from Zero to Proficiency

Unity from Zero to Proficiency (Foundations): a Step-by-step Guide to Creating your First Game

Unity from Zero to Proficiency (Beginner)

Unity from Zero to Proficiency (Intermediate)

Unity from Zero to Proficiency (Advanced)

Unity from Zero to Proficiency (Proficient)

Standalone

Becoming Comfortable with Unity

Watch for more at Patrick Felicia's site.

About the Author

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

Read more at Patrick Felicia's site.