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

GODOT FROM ZERO TO PROFICIENCY

(INTERMEDIATE)

PATRICK FELICIA

Godot From Zero to proficiency (Intermediate)

First Edition

A step-by-step guide to programming your first FPS with GDScript.

Table of Contents

Godot From Zero to Proficiency (Intermediate)
First Edition

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

- Published by Patrick Felicia

Credits

Author: Patrick Felicia

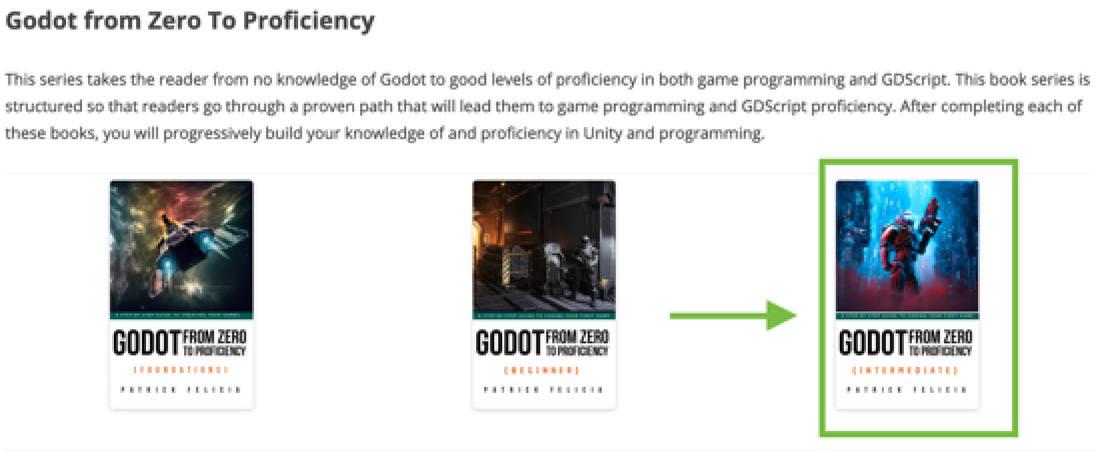
About the Author

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

Support and Resources for this Book

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

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

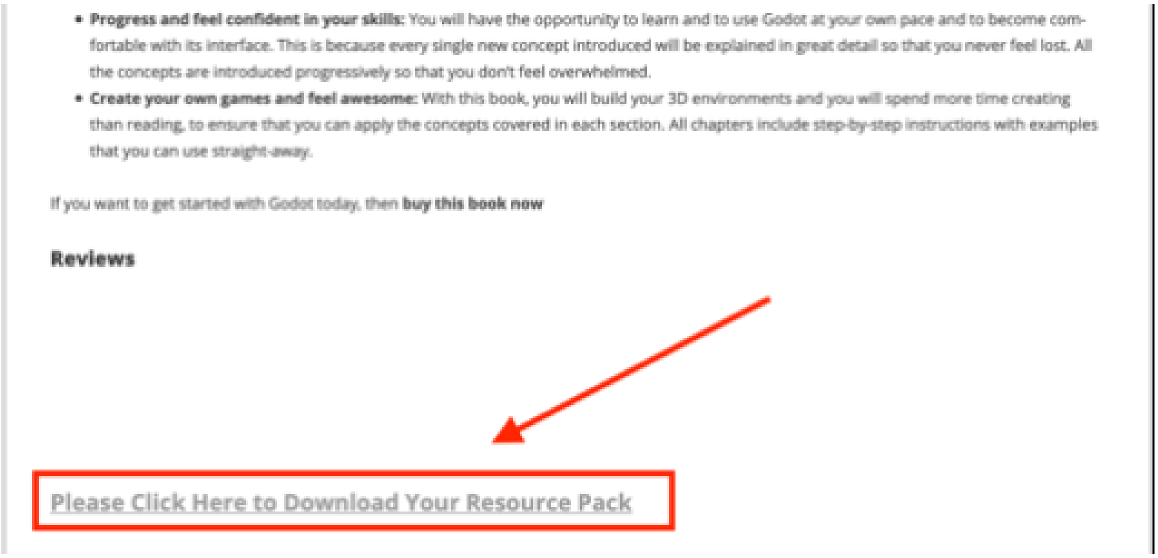


Godot from Zero To Proficiency

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

GODOT FROM ZERO TO PROFICIENCY
(FOUNDATIONS)
PATRICK FELICIA
GODOT FROM ZERO TO PROFICIENCY
(BEGINNER)
PATRICK FELICIA
GODOT FROM ZERO TO PROFICIENCY
(INTERMEDIATE)
PATRICK FELICIA

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



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

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

Reviews

Please Click Here to Download Your Resource Pack

This book is dedicated to Mathis

[ ]

Table of Contents

Preface

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

Although it may not be as powerful as Unity or Unreal yet, Godot offers a wide range of features for you to create your video games. More importantly, this game engine is both Open Source and lightweight which means that even if you have (or you are teaching with) computers with very low technical 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 a survival game using Godot's core features in terms of programming (i.e., GDScript), game design, and drag and drop features.

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

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

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

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

[ ]

Content Covered by this Book

Chapter 1, *Adding Simple Artificial Intelligence*, shows you how you can create robots that will propel projectiles towards the player; you will also create a mini-survival game where the player has to reach the end of the level while avoiding the projectiles.

Chapter 2, *Creating and Managing Weapons*, explains how you can create a weapon management system, collect ammunition, switch between weapons (i.e., a gun, an automatic rifle, and grenades), and hit targets using ray-casting.

Chapter 3, *Using Finite State Machines*, gets you to create a system that will enable your NPCs to have a good level intelligence to detect the player, chase the player, patrol around the level, and inflict damage to the player through punches or gunshots.

Chapter 4, *More Artificial Intelligence*, explains how you can improve the NPCs' intelligence so that they can follow a given path, and avoid obstacles along the way. Finally, you will learn to create a screen flash, and to also spawn new NPCs automatically at run time and at frequent intervals.

Chapter 5 provides answers to Frequently Asked Questions (FAQs) related to the topics covered in this book (e.g., scripting, audio, interaction, AI, or user interface).

Chapter 6 explains how you can create, animate and export your animations using Mixamo and Blender so that they can be used and imported in Godot as a single file.

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

What you Need to Use this Book

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

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

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

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

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

Who this book is not for

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

1. Can you already code with GDScript to implement navmesh navigation, character animation, collision detection, ray-casting, or to update the user interface?
2. Can you already easily code a 3D game with Godot with built-in objects, controllers, cameras, lights, and terrains?
3. Are you looking for a reference book on GDScript?
4. Are you an experienced (or at least advanced) Godot user?

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

How you will Learn from this Book

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

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

Format of Each Chapter and Writing Conventions

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

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

Special Notes

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

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

The author's notes appear as described below:

The author's suggestions appear in this box.

Code appears as described below:

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

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

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

How can You Learn Best from this Book

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

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

- **Do the exercises.**

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

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

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

- **Export your games early.**

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

- **Learn in chunks.**

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

  keep your motivation levels high.

Feedback

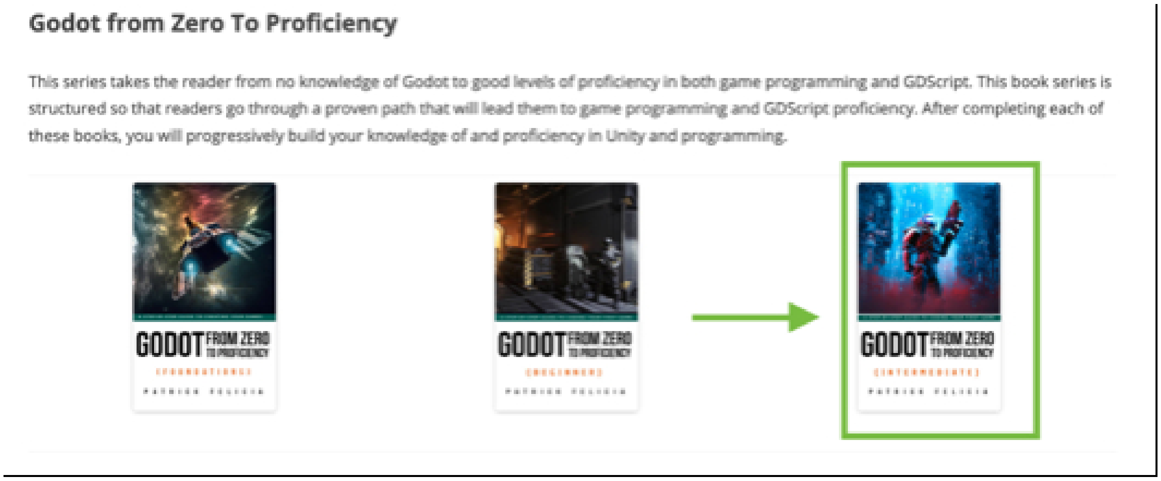
While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback, you can email me at **learntocreategames@gmail.com**.

Downloading the Solutions for the Book

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

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

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

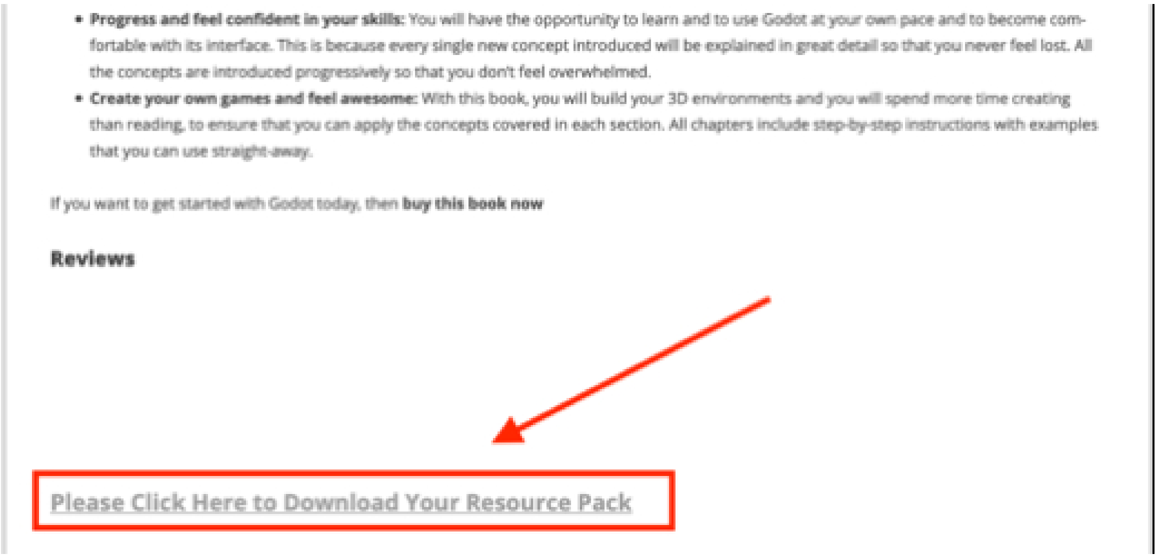


Godot from Zero To Proficiency

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

GODOT FROM ZERO TO PROFICIENCY
(FOUNDATIONS)
PATRICK FELICIA
GODOT FROM ZERO TO PROFICIENCY
(BEGINNER)
PATRICK FELICIA
GODOT FROM ZERO TO PROFICIENCY
(INTERMEDIATE)
PATRICK FELICIA

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



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

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

Reviews

Please Click Here to Download Your Resource Pack

Improving the Book

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

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

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

Supporting the Author

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

This means that if a friend wants a copy of this book, s/he will have to buy it through the official channels or the book's official website:

http://www.learntocreategames.com/books).

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

[ ]

Chapter 1: Adding Simple Artificial Intelligence

In this section we will discover how you can create very simple Artificial Intelligence (AI) to implement a cannon that fires projectiles towards your character.

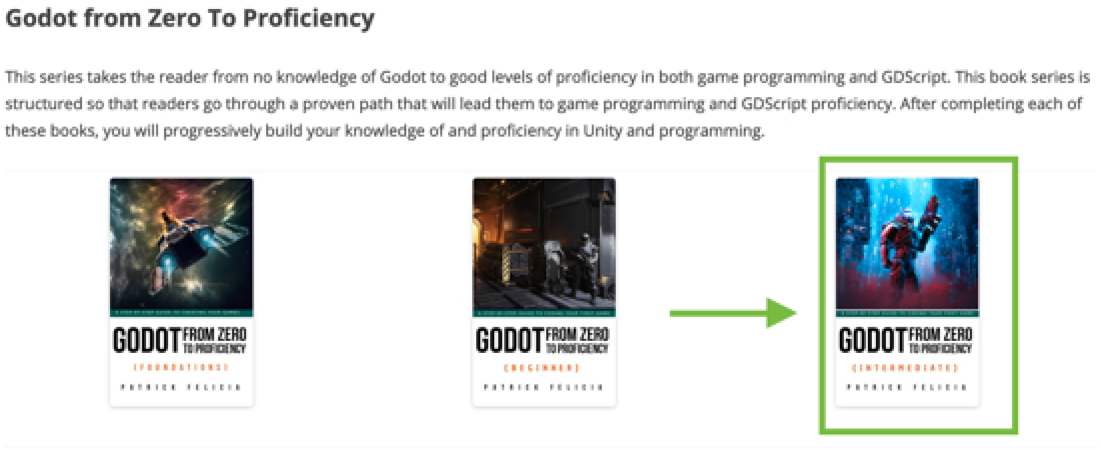
After completing this chapter, you will be able to:

- Instantiate objects.
- Make it possible for a cannon to follow and shoot at a particular object.
- Modify the firing rate of the cannon.
- Detect collision between the cannon balls and the player.

RESOURCES NECESSARY FOR THIS CHAPTER

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

- Open the page **http://www.learntocreategames.com/books**.
- Click on your book (**Godot From Zero to Proficiency (Intermediate)**)
- In the new page, please click the link that says "**Please Click Here to Download Your Resource Pack**"



Godot from Zero To Proficiency

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

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

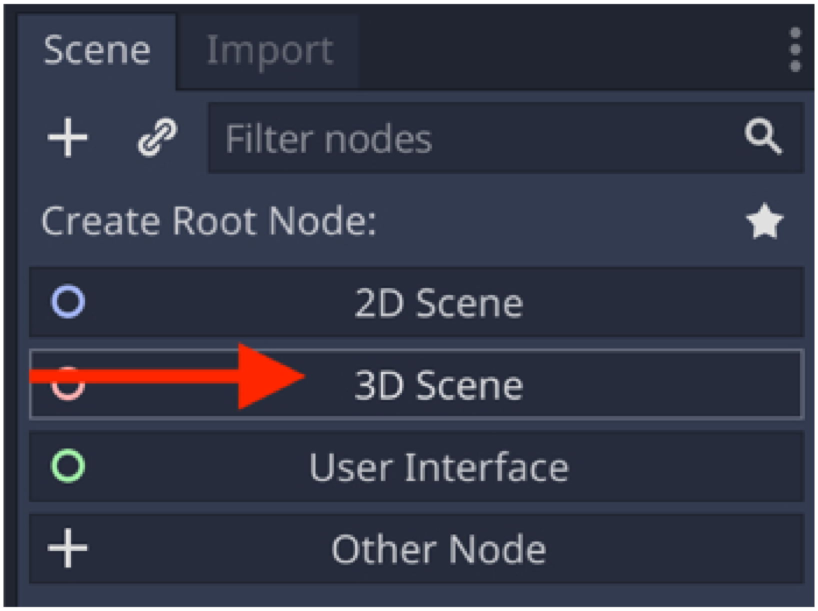
INSTANTIATING PROJECTILES

In this section we will get to create a very simple AI object that will target the player and shoot cannon balls in its direction. The workflow will be as follows; we will:

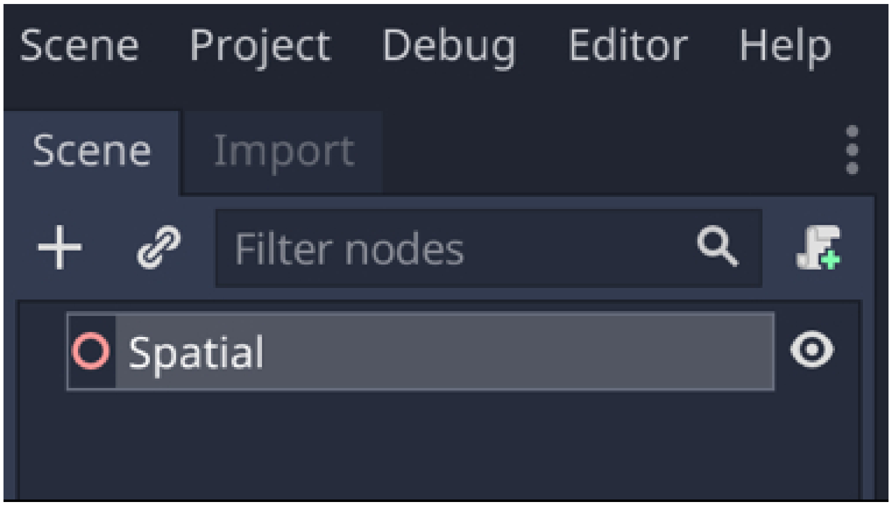
- Create a simple environment.
- Add a first-person controller.
- Add a cannon.
- Instantiate cannon balls frequently.
- Ensure that the cannon is always facing its target (i.e., the player) and shoot in its direction.
- Detect when the cannon balls have hit the player.
- Modify the firing rate of the cannon.

So, let's get started:

- Please open Godot.
- Open the project included in the resource pack called **start-up project**: from the **Project** window, select **Import**, navigate to the folder where you saved the resources, and select the **project** (or /folder) called "**startup_project**".
- In the new window, select the option **3D Scene** so that we can create a 3D scene, as illustrated in the next figure.



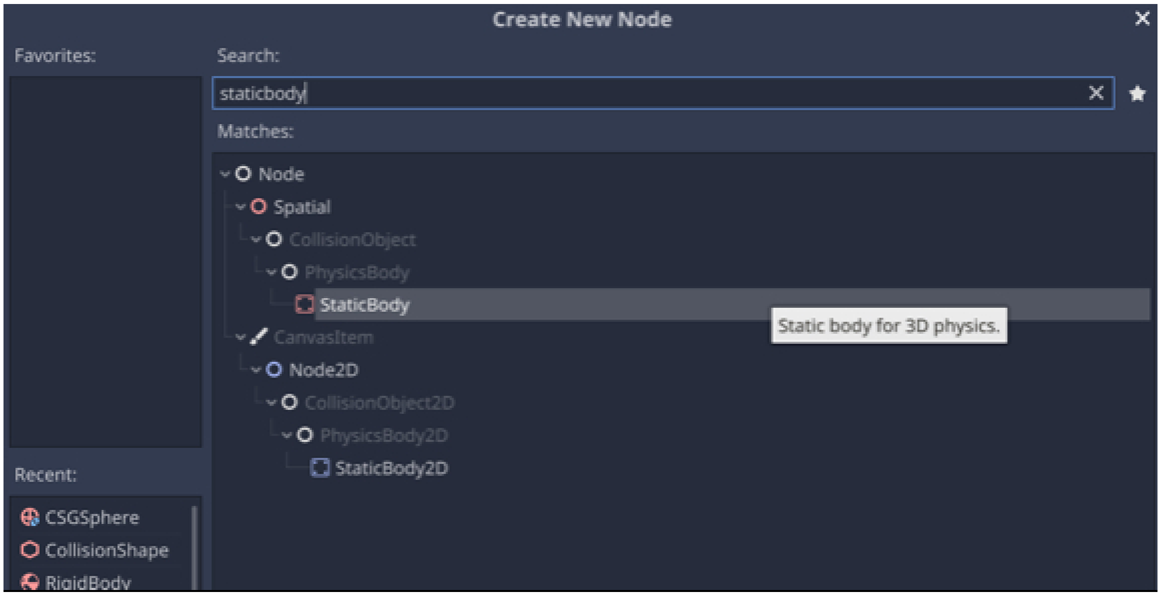
- This will create a new scene with a default node called **Spatial**.



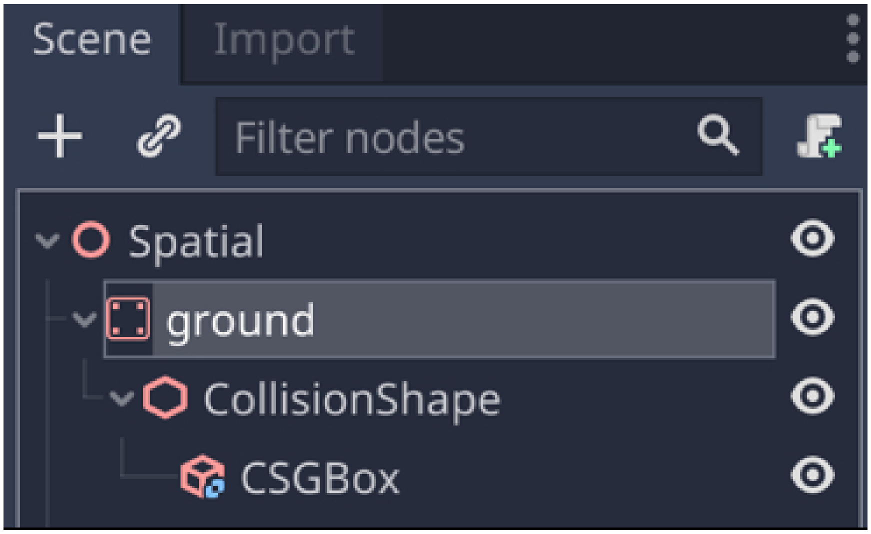
- Please save your scene as **level1** (**Scene | Save Scene As...**).

Once the new scene is saved we will add a ground and a First-Person Controller to it as follows:

- Add a new **StaticBody** node as a child of the existing **Spatial node**: right-click on the **Spatial** node, select the option **Add Child Node** from the contextual menu, enter the text "**staticbody**" in the search field, select **StaticBody** from the list, and click on the button labelled **Create**, as per the next figure.



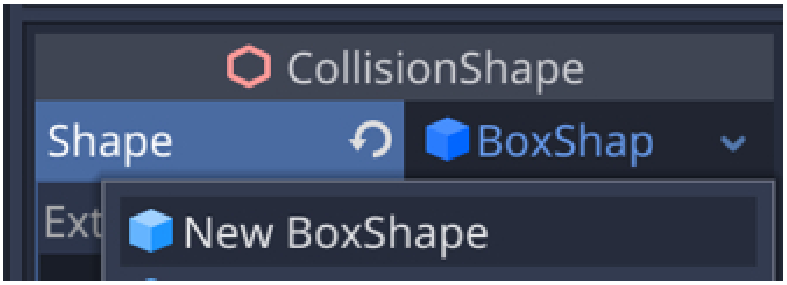
- This will create a new node called **SaticBody**.
- Using the **Scene Tree**, please rename this node **ground**.
- Once this is done, please add a node of type **CollisionShape** to the node **ground**. This will create a new node called **CollisionShape.**
- Add a new node of type **CSGBox** as a child of the node **CollisionShape**.



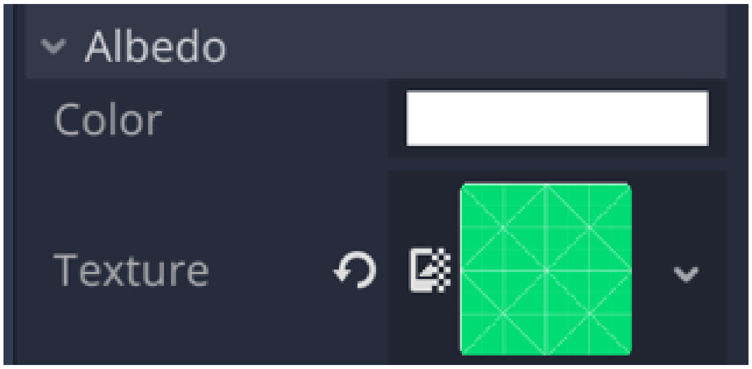
At this stage, we just need to resize the **ground** node, specify a collision shape for the node **CollisionShape** and also provide some texture and color to the ground. So please do the following:

- Select the node **CollisionShape**.

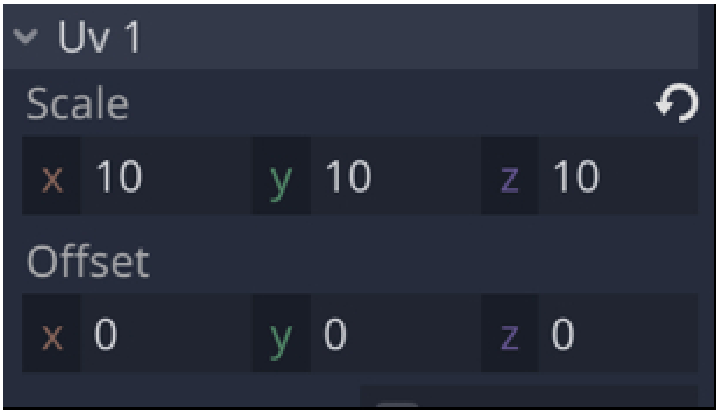
- Using the **Inspector**, click on the downward-facing arrow to the right of the label **Shape**, and select the option **New Box Shape**, as per the next figure



- Select the node **CSGBox**, and using the **Inspector**, click on the downward-facing arrow to the right of the label **Material**.
- From the drop-down menu, select the option "**New SpatialMaterial**", click on the white sphere, expand the section called **Albedo**, and drag and drop the texture **texture05.png** from the **FileSystem** window to the field **Texture** located in the **Albedo** section, as per the next figure.



- Using the section **UV1**, change the **scale** property of this node to (**10, 10, 10**).



Uv 1

Scale

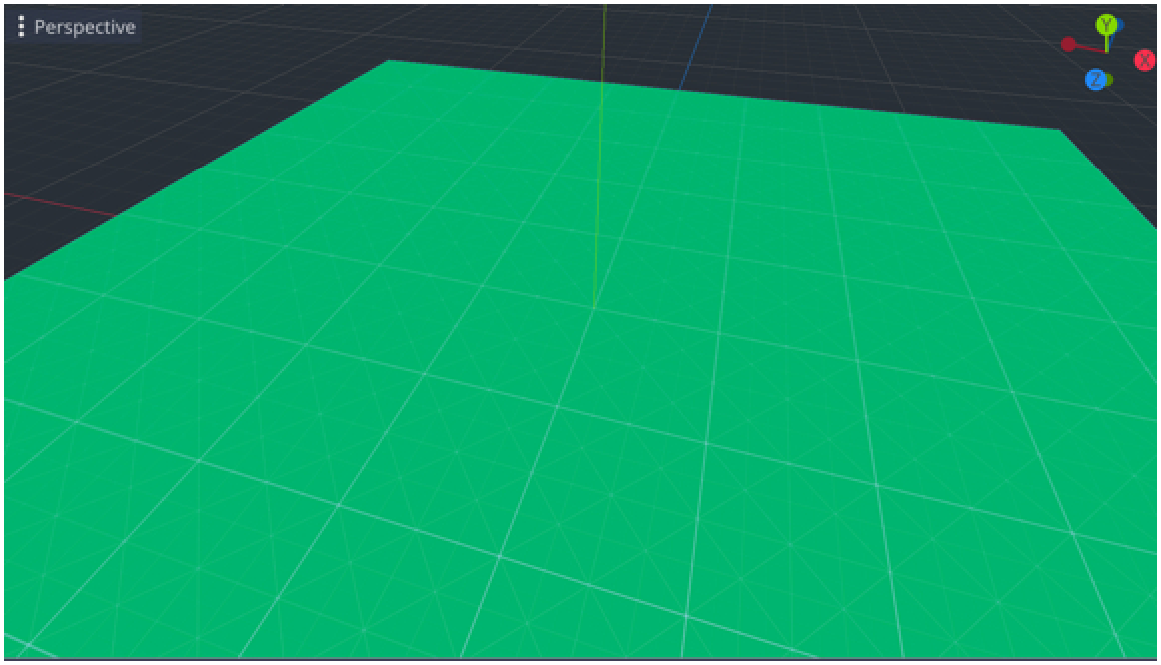
x 10 y 10 z 10

Offset

x 0 y 0 z 0

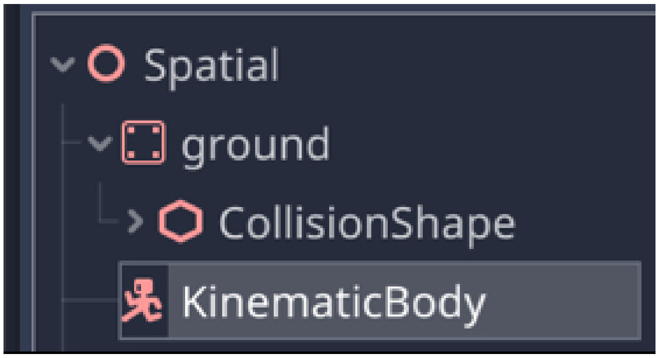
- Finally, select the node **ground**, and, using the **Inspector**, in the section called **Transform**, change the scale property to **(100, 1, 100)**.

So at this stage, the scene should look like the following figure:



Next, we will add a **First-Person Controller**:

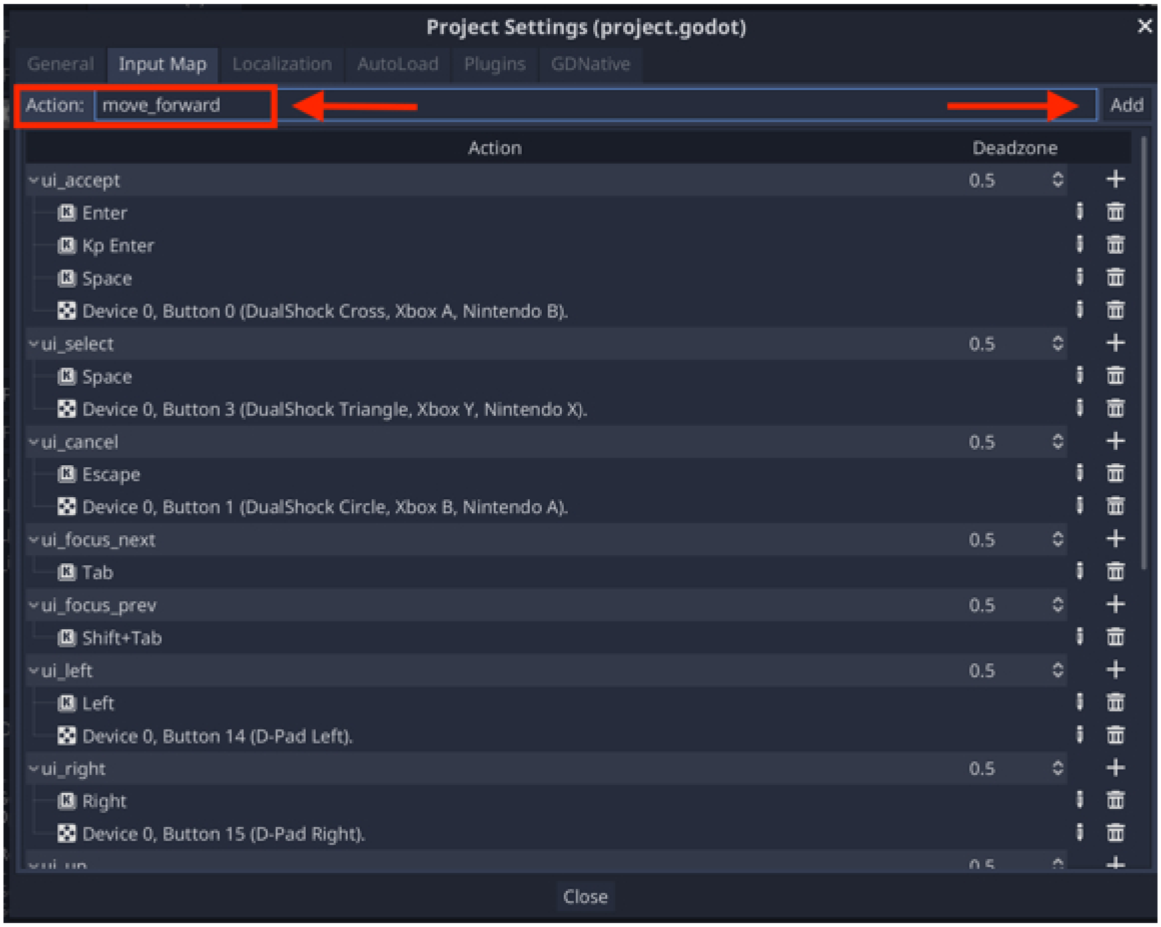
- Please right-click on the node called **Spatial**.
- From the contextual menu, select the option **Instance Child Scene**.
- In the new window, select the scene **player.tscn** from the new window, and click on the button labeled "**Open**".
- This should add a node called **KinematicBody** as a child of the node **Spatial**.



- Please rename this new node **player**.

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

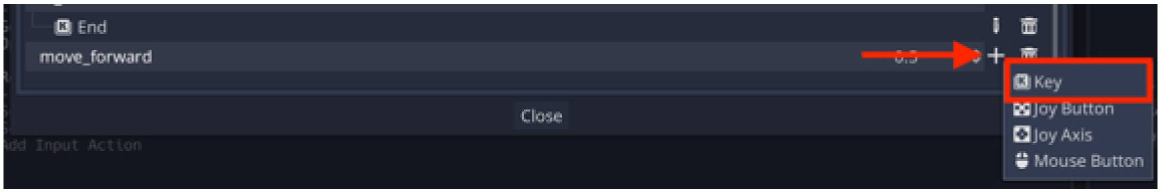
- From the top menu, please select: **Project | Project Settings | Input Map**.
- In the new window, enter the text "**move_forward**" in the top field (i.e., the field labelled "**Action**") and then press the button labelled **Add** (to the right of the field), as per the next figure.



Action: move_forward

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

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



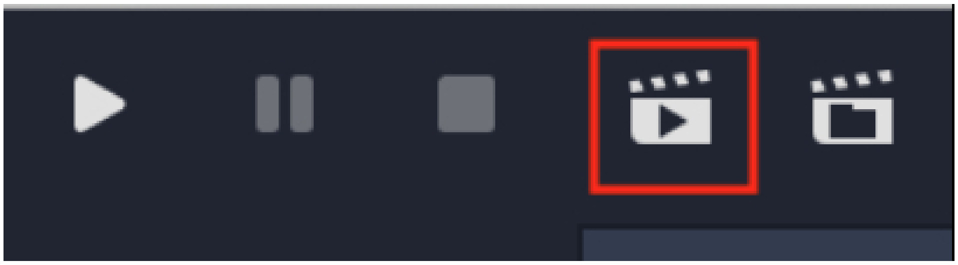
- Press the **Up Arrow** on your keyboard and then press **OK**. This means that the up arrow key is now associated with an input that is referred to as **move_-forward** in any script used in this project, and especially the script that is used to move the FPS controller.

Please repeat the previous steps to add the following settings:

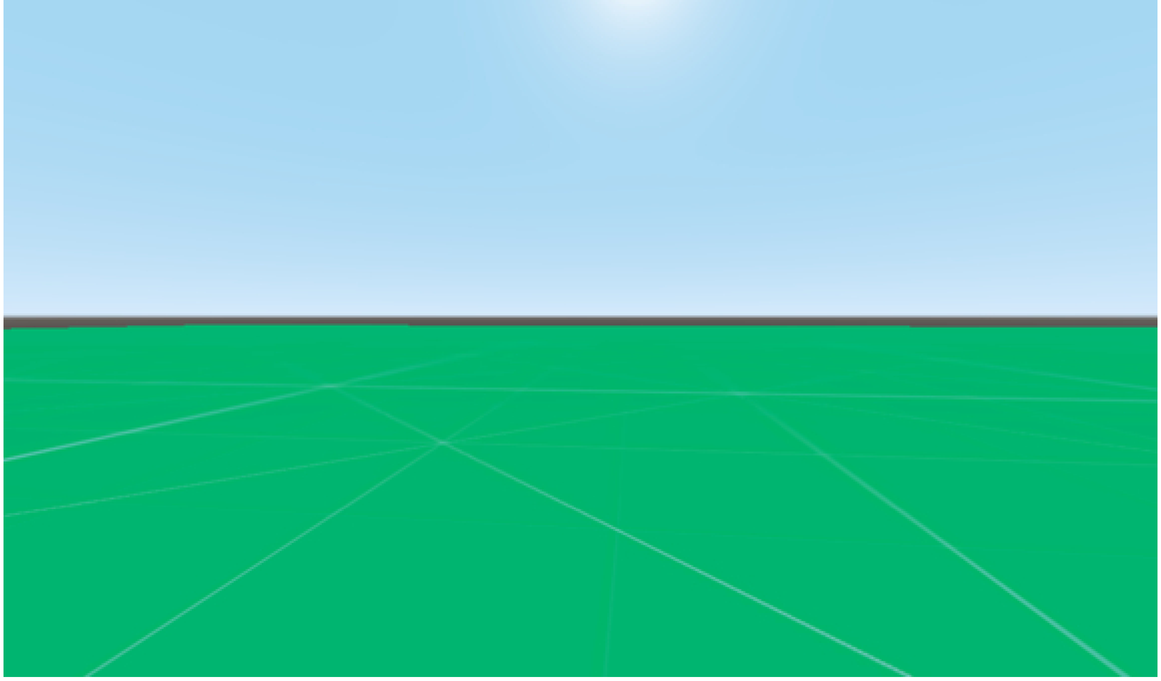
- **move_back** using the **Down Arrow**.
- **move_left** using the **Left Arrow**.
- **move_right** using the **Right Arrow**.
- **jump** using the **Spacebar**.

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

At this stage, you can play the scene by pressing **F6** (if you are using a Windows operating system) or **CMD + R** (if you are using a MacOS operating system). You can also press the corresponding button in the top-right corner - the fourth from the left) to play the scene.



- As you play the scene, you should be able to navigate the scene without going through the walls by pressing the arrow keys on your keyboard and the mouse to rotate the view.



You can now stop the scene.

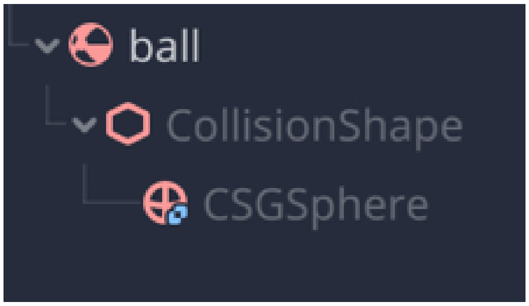
We will now create a launcher that will be used to fire projectiles at the character.

- Please create a new node of type **CSGBox** as a child of the node **Spatial** and rename it **launcher**. Change its position so that it is above the ground and away from the player, for example **(15, 7, 0)**.

Next, we will create the actual projectile that will consist of a simple sphere. This sphere will be used as a template for any projectile fired from the launcher.

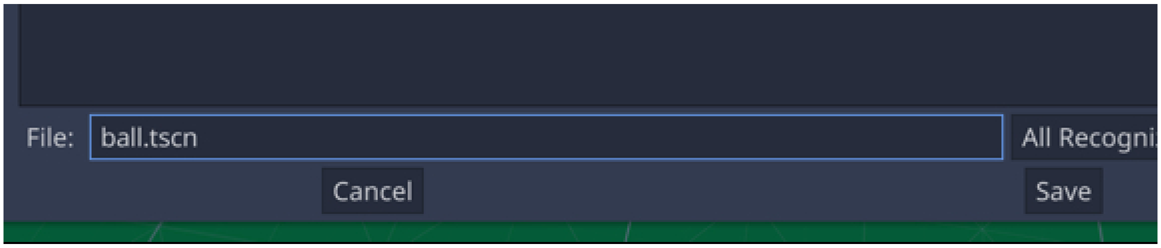
Because this ball will be used as an airborne projectile, it will need to behave as if it was subject to gravity and emulate other physics-based behaviors. For this, we will create a **RigidBody** node that effectively has physics properties:

- Please create a new **RigidBody** node as a child of the **Spatial** node and rename it **ball**.
- Make sure that its position is **(0, 0, 0)**.
- Add a **CollisionShape** node as a child of this node.
- Change its scale settings to **(0.2, 0.2, 0.2)** so that its radius is **0.2**.
- Using the **Inspector**, set the **Collision Shape** to a **Sphere** by clicking on the downwards-facing arrow to the right of the field **Shape**, and by selecting the option **New SphereShape**.
- Finally, add a **CSGSphere** node as a child of the node **CollisionShape**.



Once this is done we can save this node as a new scene, so that it can be reused (and instantiated) several times throughout the game to create several projec tiles, so we will create a new scene from the node called **ball**.

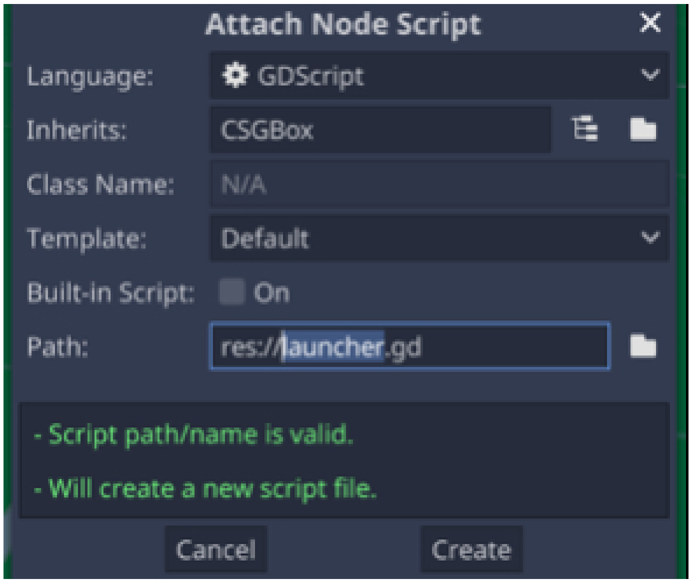
- Please right-click on the node called **ball**.
- From the contextual menu, please select "**Save Branch As Scene**".
- In the new window, type **ball.tscn** in the field called **File**, and press the button labelled "**Save**". This means that now the node called ball will be available throughout your project.



File: ball.tscn

Once the scene called **ball.tscn** has been created, it is time to create a script that will control the launcher and instantiate projectiles.

- Using the **Scene Tree** window, please right-click on the node called **launcher**, and select the option **Attach Script** from the contextual menu.
- In the new window, leave all the options as default and click on the button called "**Create**". This will create a new script called **launcher.gd** that will be attached to the node **launcher**.



Attach Node Script

Language: GDScript

Inherits: CSGBox

Class Name: N/A

Template: Default

Built-in Script: □ On

Path: res://launcher.gd

- Script path/name is valid.

- Will create a new script file.

The first thing we will do in this script is to create a member variable that will act as a placeholder and a template for all projectiles. So we will create a variable for the projectile's node (i.e., ball) that is currently saved as a scene, so that, when the script has been attached to the node, we can drag and drop the scene ball to this empty field.

- Please add the following code to the script, within the class definition (new code in bold).

```
extends CSGBox
export (PackedScene) var ball
```

Then, we can try to detect when the player has pressed a key on the keyboard so that an object (i.e., a projectile) can be instantiated in this case. Please enter the following code outside of any other function; for example, at the end of the script.

```
func _input(event):
if (Input.is_key_pressed(KEY_O)):
var new_ball = ball.instance()
add_child(new_ball)
```

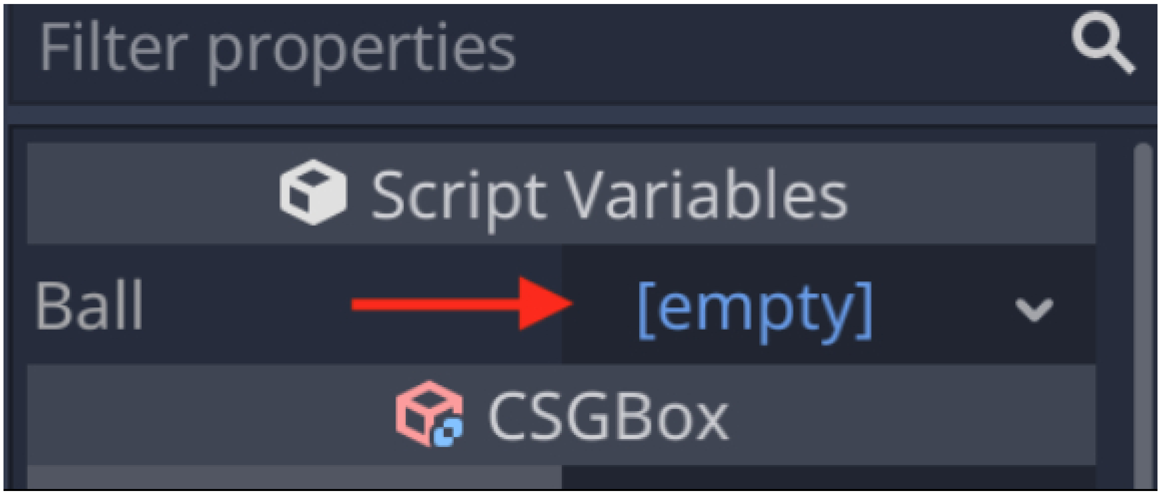
In the previous code:

- We use the function **_input** that is called whenever, amongst other things, the user types on the keyboard.
- We check whether the key **O** has been pressed.
- We then create a new instance of the object **ball** using the keyword **instance**. The new object is instantiated from the object called **ball**; its position will be the same as the node linked to this script (i.e., in our case, this will be the node called **launcher**).
- We can now save our script and check that there are no errors.

There is now one last thing we need to do, that is, dragging and dropping the scene called **ball** that we have created initially to the field called **ball** in the script that is attached to the launcher.

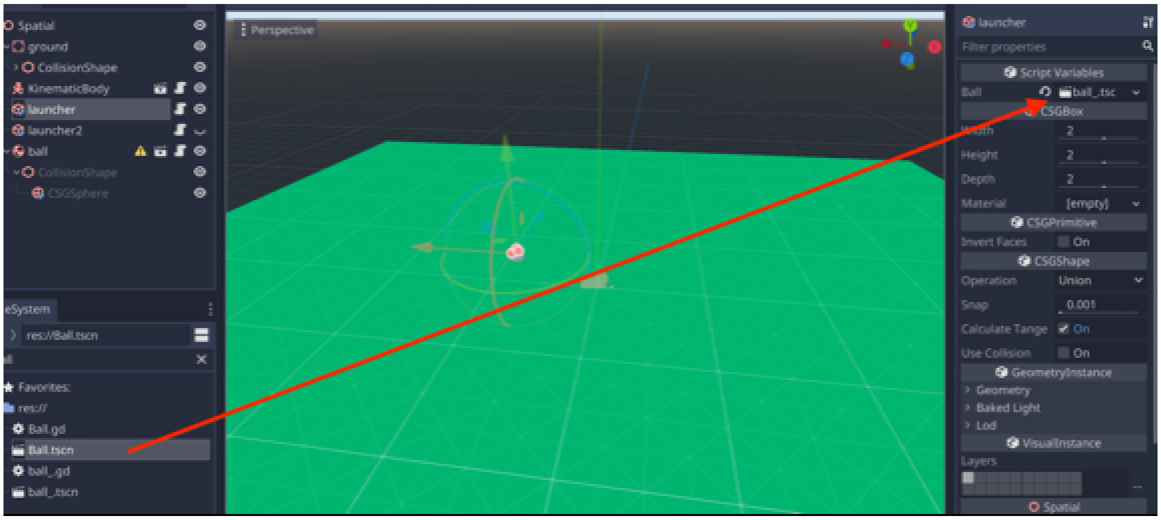
- Please select the object **launcher** in the **Scene Tree** window.
- In the **Inspector** window, you should see that there is an empty field, in the

section **Script Variables**.



Ball [empty]

- Please drag and drop the scene **ball.tscn** from the **FileSystem** window to this field; this will effectively set the value of the variable **ball** in the script to the scene **ball.tscn**, as described in the next figure.



Ball ball_.tsc
Width 2
Height 2
Depth 2
Material [empty]
Invert Faces On
Operation Union
Snap 0.001
Calculate Tange ☑ On
Use Collision On

Once this is done, let's play the scene and test our script.

- Please play the scene.
- Move the character near the launcher.
- Press the **O** key, and you should notice that balls are dropped from the launcher.

You can stop playing the scene now.

So at this stage, the instantiation is working properly; however, we would like to add a force to these spheres, so that they are effectively propelled in the air, rather than being dropped on the ground. Before that, there is also another aspect that we need to look at: as we create these projectiles, they may start to overcrowd the scene and we could decide to destroy them after a few seconds, to keep the performance of our game high. For this purpose, we will use the method **queue_free**.

- Please add the following code, within the code that tests for the user input (just after the instantiation).

```
var timer = Timer.new()
new_ball.add_child(timer)
timer.connect("timeout", new_ball, "queue_free")
timer.set_wait_time(2)
timer.start()
```

In the previous code:

- We create a new timer.
- We then add the timer to the current node (i.e., the ball).
- We connect the event **timeout** to the function **queue_free** for the node **new_ball**; in other words, when the timer times out, the function **queue_free** will be called in relation to the node **new_ball**, which will remove this node from the scene.
- Finally, we set the duration of this timer to 2 seconds and we then start the timer.

That's it; we are now ready to play and test the scene.

- Please save your code.
- Play the scene.
- Move your character near the launcher.
- Press the **O** key several times.
- Check that the projectiles (i.e., the balls) have been created and that they

disappear after a few seconds.

- You can playing the scene now.

We can now start to think about adding a force to the projectiles. To do so, we will be modifying the velocity of each ball.

- Please add the following code just after the previous code.

```
new_ball.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

In the previous code:

- We modify the linear velocity of the ball; the direction of the velocity is **forward**.
- This being said, we need to obtain the local forward for the ball; this is achieved by using the function **xform**; so in other words, we use the global location of the node **transform.basis** and then use the **xform** function to extract the local transform.
- We then add a forward force to the ball with an intensity of **20** (i.e., 20meteres per second).

Now, the last modification we will make will be to ensure that the launcher is always facing the player, so that the projectile is launched towards the player. Thankfully, Godot provides a method that makes it possible to look at (or rotate a node to face) a specific object; this method usually needs a target to look at; in our case this will be the player. So let's create this feature.

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

look_at(get_node("../player").global_transform.origin,Vector3.UP)

```
newBall.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

In the previous code:

- We use the function **look_a**t that takes 2 parameters: the position of the object to look at, and an axis of rotation.

- So, we look at the node called player using its global position.
- We rotate around the y axis (i.e., **Vector3.UP**) to do so.

Once this is done we can test the scene, and as you play it, you will see that when you move the character around and press the *O* key, the projectiles will be fired in the direction of the player.

Now that we have managed to test whether the launcher could fire projectiles towards the player, we can now make it possible for this launcher to automatically fire projectiles without the need for you to press a key. For this purpose, we just need to:

- Create a new function.
- Add all the code that was previously in the **_input** function in a new function.
- Create a timer that will call this new function at regular intervals.

So let's go ahead:

- Please create a new function called **throw_projectile** and copy all the code that was previously in the function **_input** in that new function as per the next code snippet:

```
func throw_projectile():
var new_ball = ball.instance()
add_child(new_ball)
var timer = Timer.new()
new_ball.add_child(timer)
timer.connect("timeout", new_ball, "queue_free")
timer.set_wait_time(2)
timer.start()
look_at(get_node("../player").global_transform.origin,Vector3.UP)
new_ball.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

- Next, please comment the whole content of the function **_input**, as we won't be using it for now (select the code and press **CTRL/APPLE + K**).

Finally, now that the function **throw_projectile** includes all the code that we need to throw projectiles, we just need to create a timer that will call this function every 2 seconds.

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

```
var shoot_timer:Timer
```

In the previous code, we create a new timer; a timer node works like an alarm clock: you specify when it should go off and what should be done when this happens. This is usually done by setting its attribute **wait_time**, and by wiring its timeout event (i.e., when it goes off) to a specific function.

- Next, add the following code to the function **_ready** (new code in bold).

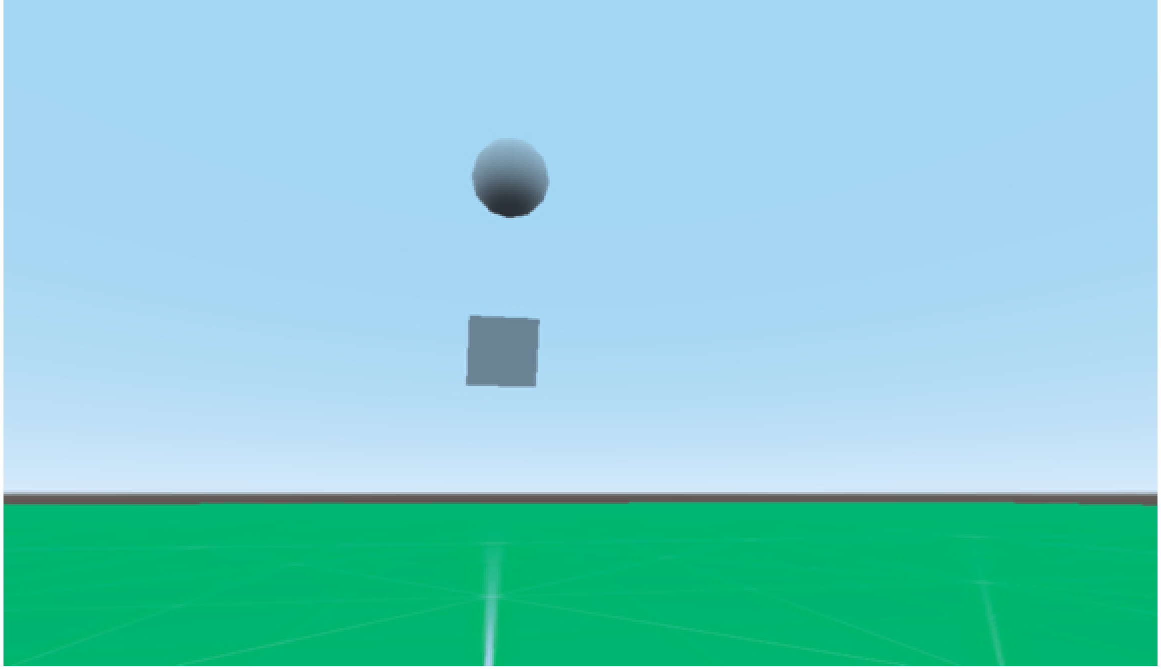
```
func _ready():
shoot_timer = Timer.new()
add_child(shoot_timer)
shoot_timer.wait_time = 2
shoot_timer.connect("timeout",self,"throw_projectile")
shoot_timer.start()
```

In the previous code:

- We instantiate the new timer.
- This timer is added to the current node (i.e., the node **launcher**).
- We set the **wait_time** attribute of the timer to **2** (i.e., 2 seconds)
- We connect the event **timeout** of this timer and the function **throw_projectile** which is defined in this script; in other words, we say that whenever the timer reaches 2 seconds (when it **times out**), the function called **throw_projectile** that is in this script (hence the use of the keyword **self**) will be called.

Note that most nodes have specific associated events; for the timers, timeout is one of these events.

You can now play the scene and see how the launcher shoots balls every 2 seconds towards the player.



MANAGING COLLISION

So at this stage, we have managed to implement a relatively simple, yet effective, artificial intelligence, that follows the player and shoots projectiles in his/her direction. We could now modify the code that we have to detect when the projectile hits the player.

So let's get started:

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

```
var timer = Timer.new()
new_ball.add_child(timer)
new_ball.set_contact_monitor( true )
new_ball.set_max_contacts_reported( 5 )
new_ball.connect("body_entered",self,"ball_collision")
```

In the previous code:

- We use the function **set_contact** so that collisions are monitored for the balls.
- We also set a maximum number of contacts to be reported to **5**; this was chosen arbitrarily, but as long as the value is greater than one, you should be able to detect collisions.
- Finally, we connect the event **body_entered** and the script called **ball_collision** (that we have yet to create) that is defined on this script (hence the use of the keyword **self**).
- The event **body_entered** is called whenever a **KineticBody** node collides with another node; so in our case, whenever the ball collides with another object, the function **ball_collision** will be called.

Finally, we just need to define the function **ball_collision**; please add the following code at the end of the script:

```
func ball_collision (body):
```

```
if (body.name == "player"):
print("hit player")
```

You can now save your code and play the scene; as the player is hit by a ball, you should see the message “**hit player**” in the **Output** window.

FINISHING OUR FIRST GAME

So the game is working fairly well at this stage, as the launcher manages to track (and shoot at) the player.

There are a few last things that we could do to both improve the game and to be able to reuse the code:

- To be able to reuse the objects that we have created, especially the launcher, it would be great to create a template from them (i.e., a new scene based on the node).
- Since the user will lose a life when hit by a projectile, it would be good to keep track of his/her number of lives.
- Finally, it would be nice to create a game with a goal, a scoring system, and the ability to display information onscreen about the status of the game.

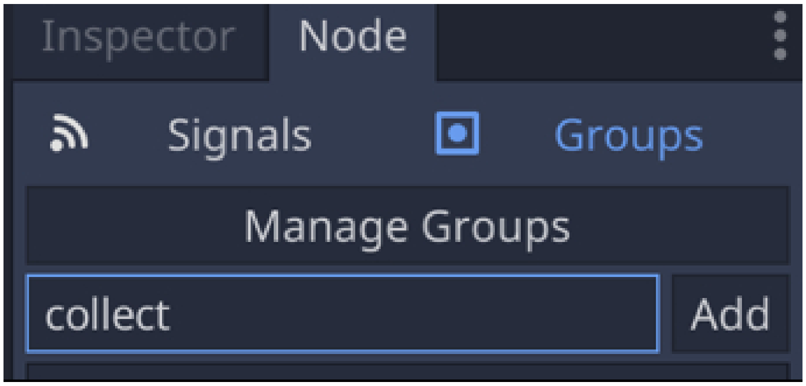
So let's make these modifications. First, we will modify the game so that it uses the following gameplay:

- The player needs to collect five objects and reach a platform that symbolizes the end of the level.
- Intelligent robot launchers will track and shoot at the player.
- There will be four launchers in total.
- The player, if hit by a projectile, has to restart the level from the starting point.

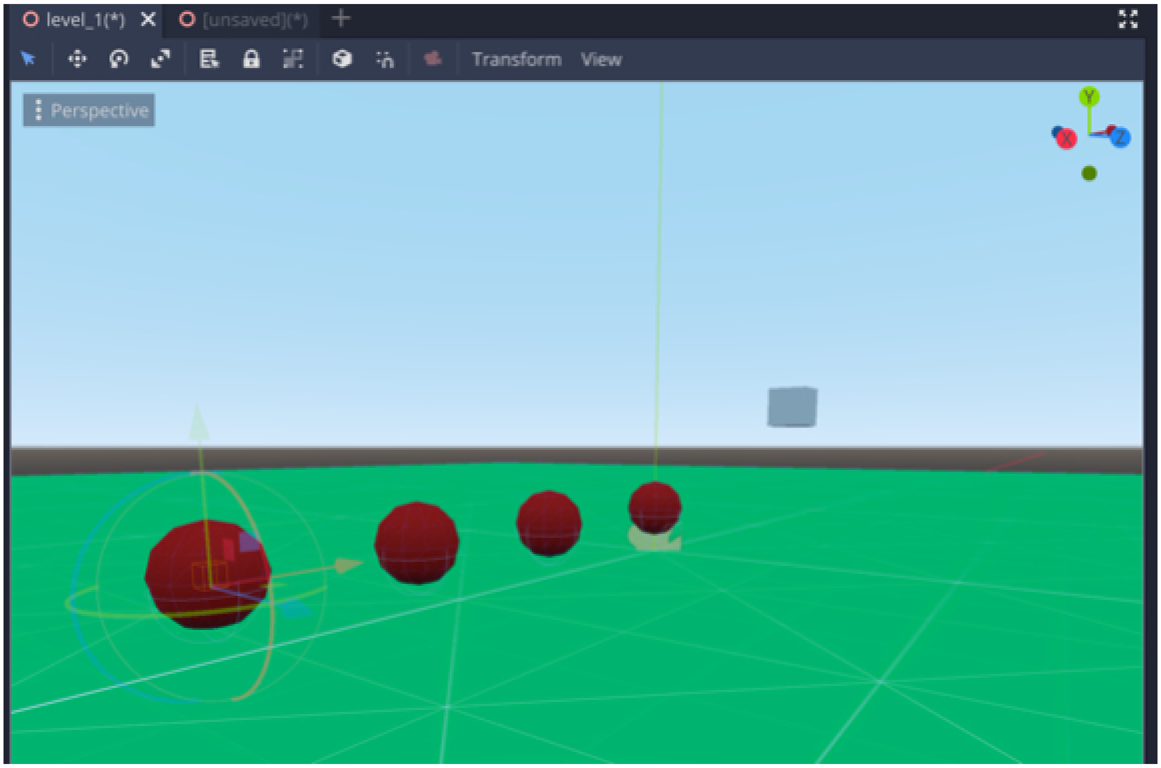
Please do the following:

- Create a node of type **StaticBody**, as a child of the **Spatial** node; rename this node **sphere1**, add a **CollisionShape** node as a child of the node **sphere1** and set its shape to a **Sphere**. Finally, add a node of type **CSGSphere** as a child of the node **CollisionShape** and change its color to **red**, by creating a new **Spatial Material** and by setting its color using its **Albedo** attribute.
- Please set a group for this node: select the object **sphere1**, and using the **Node | Group** tab, set a group called **collect** for this node, as per the next figure. This is to ensure that the player can detect them and collect them

accordingly.



- Change the y-coordinate of the node **sphere1** to **1.5**.
- Duplicate this node three times and rename the duplicates **sphere2**, **sphere3**, and **sphere4**.
- Move these nodes a few meters apart, as per the next figure.



Once this is done, we will create the code that will detect when the player collides with the objects to be picked-up.

- Please open the script called **Player.gd** that is attached to the **player** node and add the following code before the **_ready** function.

```
var score = 0
```

In the previous code, we create a new variable **score** that will be used to keep track of the score. We then initialize the **score** to 0.

- Please add the following code at the end of the function **_physics_process** as follows (new code in bold):

```
for index in get_slide_count():
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("collect")):
print("Collision with " + collision.collider.name)
score += 1
print(“score”+str(score))
collision.collider.queue_free()
```

In the previous code:

- We detect collisions with the player and we loop through all the nodes that the player is colliding with using the for loop.
- For each of the nodes colliding with the player, we check if they belong to the group called **collect**.
- If this is the case, we destroy this node and we also increase and display the score.

Please save your code and ensure that there are no errors. Please play the scene and make sure that you can collect the different spheres.

Once this is done, we can then create two platforms: a platform that symbolizes the start of the level, and a platform that symbolizes the end of the level.

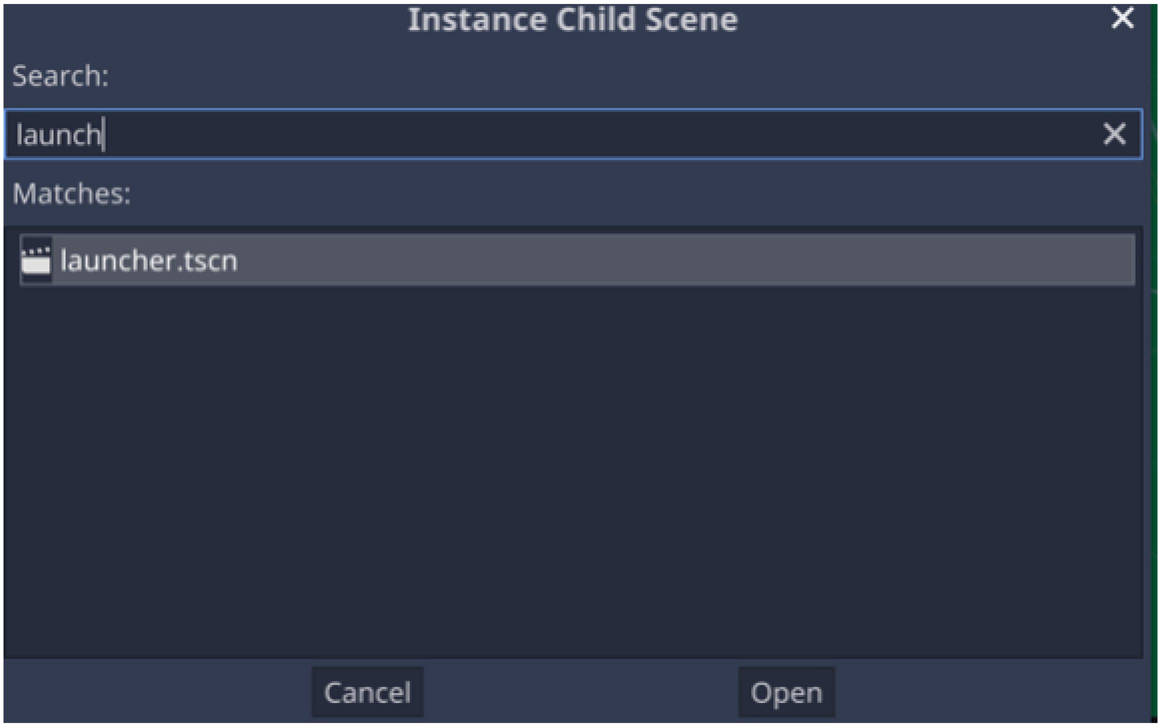
Please do the following:

- Create a node of type **StaticBody**, as a child of the **Spatial** node; rename this node **start**; add a **CollisionShape** node as a child of the node **start** and set its shape to a **Cylinder**.

- Add a node of type **CSGCylinder** as a child of the node **CollisionShape** and change its color to green after creating a new **Spatial Material** and setting its color using its **Albedo** attribute.
- Select the node **start** and change its scale attribute to **(2, 0.2, 2)** and its **y** position coordinate to **1.2**. This is the initial point from where the player will start.
- Please repeat the last steps to create a similar **red** cylinder, but with the name **end** (to speed-up the process, you can duplicate the previous node but you will need to create a new **SpatialMaterial** for its **CSGCylinder** node).
- Please move these two cylinders at least 50 meters (i.e., 50 Godot units) apart.

For now, we just need to add three more launchers; for this purpose, we will create a template scene for the launcher and reuse this scene to create the three additional launchers.

- Please right-click on the node **launcher**, and select the option **Save Branch As Scene**. In the new window, save the node as **launcher.tscn**.
- Right-click on the **Spatial** node, select the option **Instance Child Scene**.
- In the new window select the scene **launcher** and press open.



Instance Child Scene

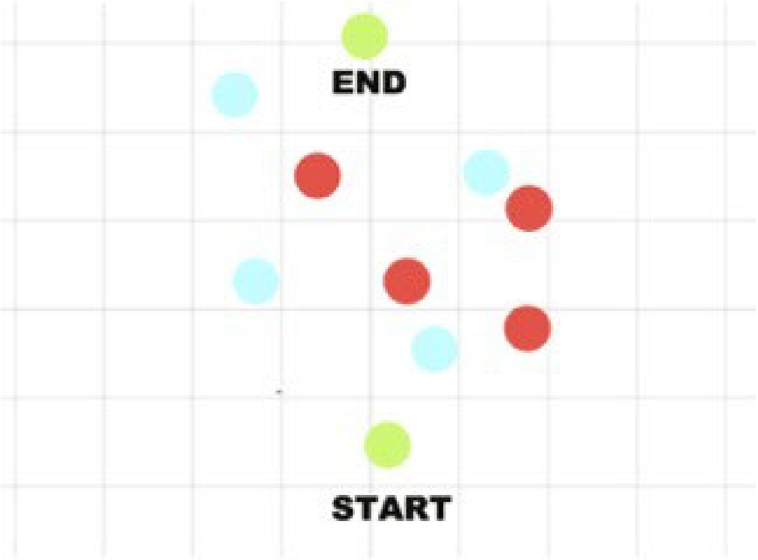
Search:

launch

Matches:

launcher.tscn

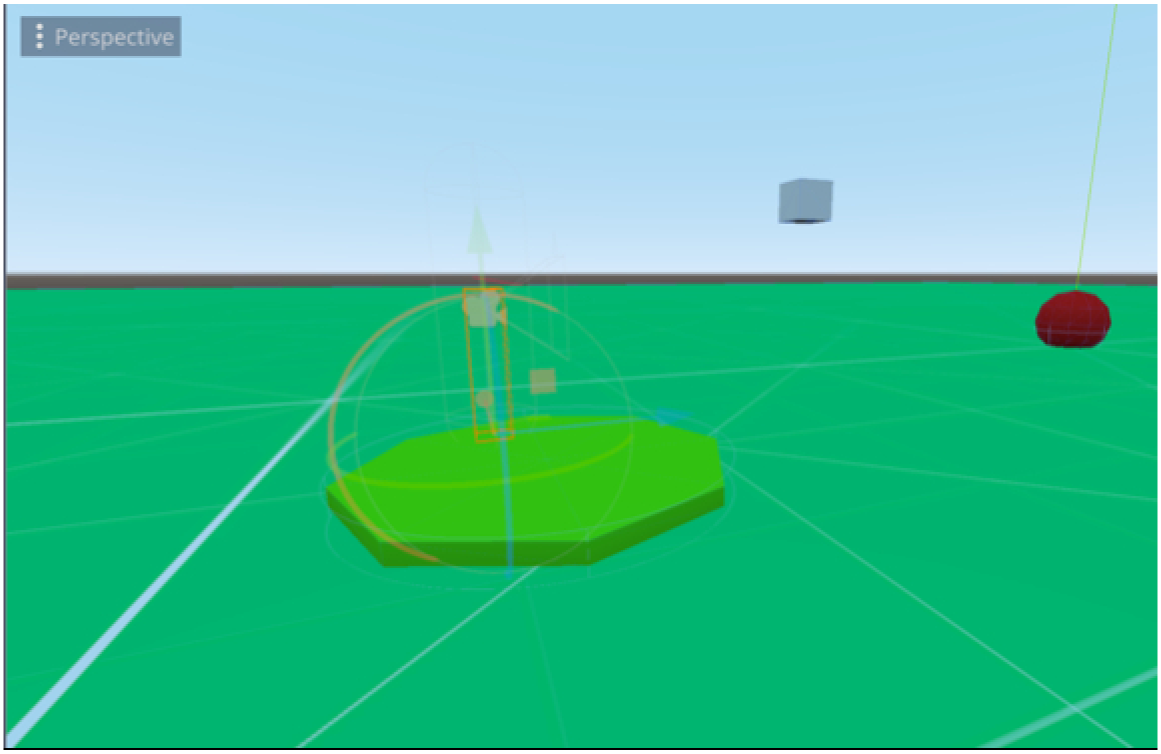
- This will create a new node called **launcher2**.
- Please repeat the last steps twice to create two additional launchers.
- So that it is easier to locate the different objects that make up this scene from the y-axis (i.e., from above), you can use different colors for the launchers, the objects to collect, the start of the level point, and the end of the level, as illustrated on the next figure.



At this stage, all the elements are ready for our game, and we just need to detect when the player has reached the end of the game, and place the player on the starting platform at the beginning of the game or after it has been hit.

Let's do the following:

- Please move the player node just 1 meter above the **start** node.



- Open the script **launcher.gd** and modify the function **ball_collision** so that, after it has been hit by a projectile, the player is moved back to the start, as follows (new code highlighted in bold).

```
func ball_collision (body):
print("collision")
if (body.name == "player"):
get_node("../player").transform.origin = get_node("../start").transform.origin +Vector3.UP
print("hit player")
```

In the previous code, we move the player back to the **start** of the game whenever it has been hit by a ball.

You can now play the scene and check that the player is taken back to the starting point after being hit by a ball.

DETECTING WHEN THE PLAYER HAS REACHED THE END OF THE GAME

At this stage, we need to detect whether the player has reached the end of the level (i.e., the **end** node) after collecting four objects. So we will need to check for both conditions and then display a message (e.g., "Congratulations") when both conditions have been fulfilled. Since we are already counting the number of objects collected, all we need to do is to access (i.e., read) the score when we have reached the **end** platform.

First, let's detect when we have reached this platform.

- Please open the script **Player.gd** and modify the function **physics_process** as illustrated in the next code snippet (new code highlighted in bold).

```
for index in get_slide_count():
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("collect")):
#print("Collision with " + collision.collider.name)
score += 1
print("score"+str(score))
collision.collider.queue_free()
elif (collision.collider.name == "end" && score == 4):
print("Congratulations!")
```

In the previous script:

- We check the name of the object colliding with the player.
- If this is the **end** platform and if the score is **4**, then we print the message **"Congratulations!"** in the **Output** window.

Please save your code, play the scene, collect four objects, and reach the **end** platform. A message saying **"Congratulations!"** should be displayed in the **Output** window.

Lastly, we will display this message onscreen using a **Label**.

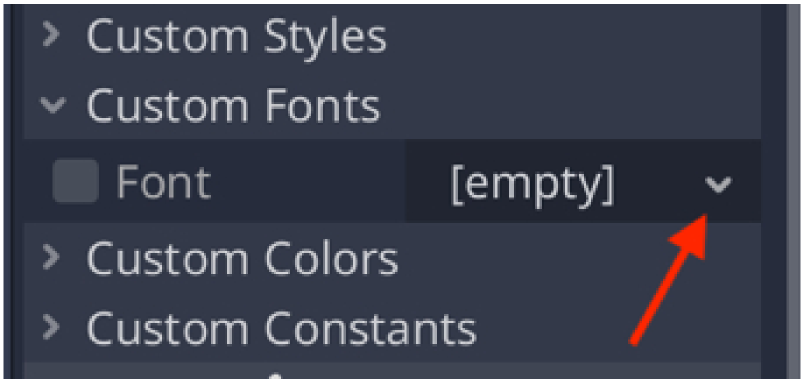
- Please add a **Label** node as a child of the **player** node.
- Rename this node **message**.
- Select this node.

Using the Inspector, modify its properties as follows:

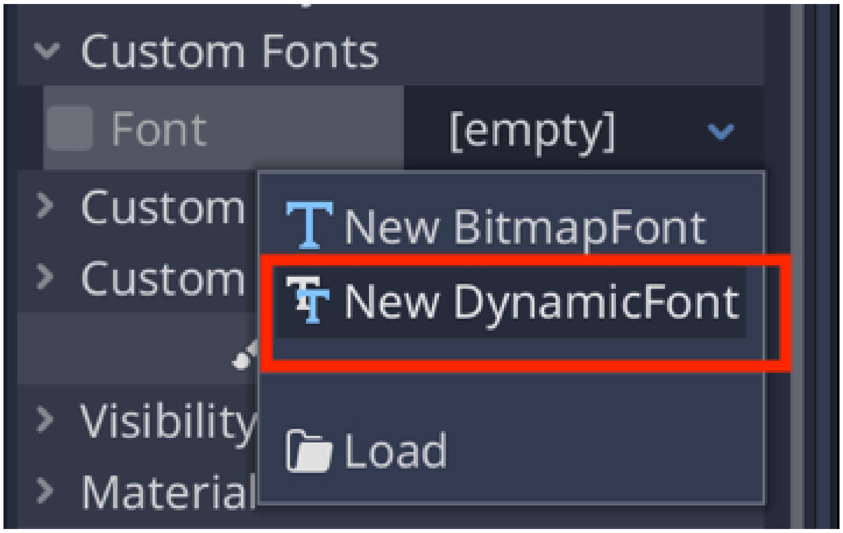
- Rect: **Size.x** = **500**.
- Rect: **Size.y** = **100**.
- **Align**: **Center**.
- **VAlign**: **Center**.

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

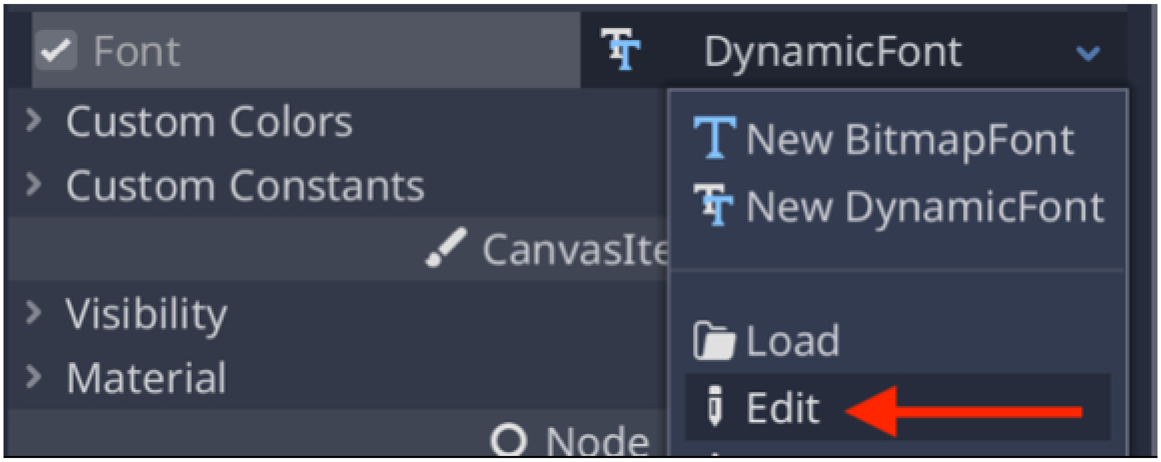
- Please select the node called **message** in the **Scene Tree**.
- Using the **Inspector**, scroll down to the section entitled **Custom Fonts** and click on the downward-facing arrow to the right of the label **Font**, as per the next figure.



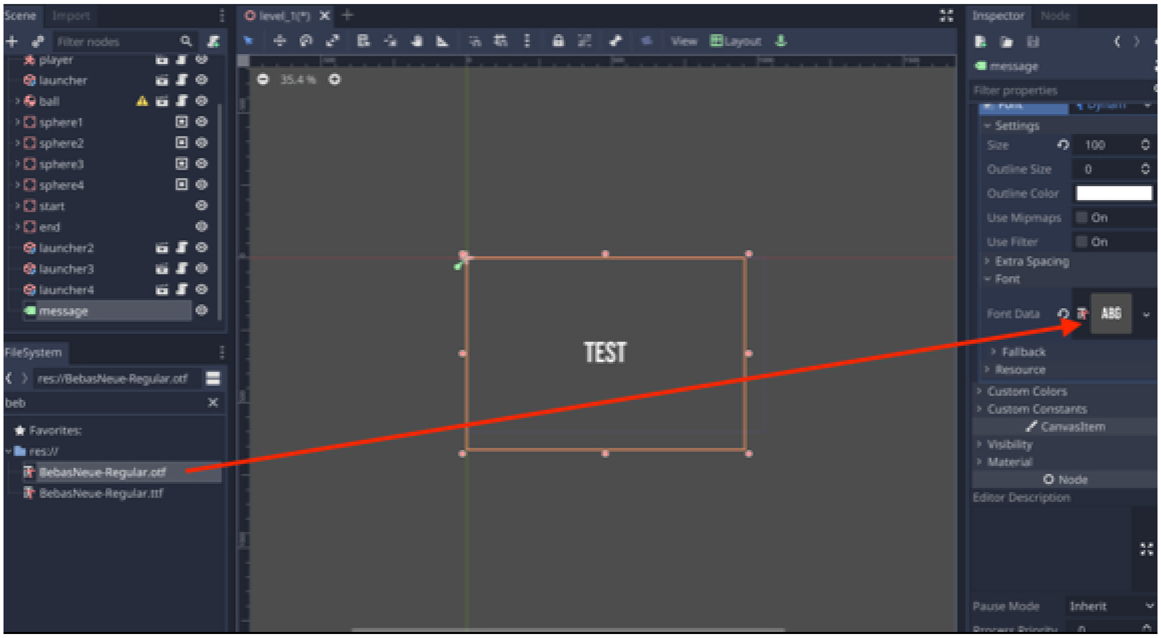
- From the contextual menu, please select the option **New DynamicFont**.



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

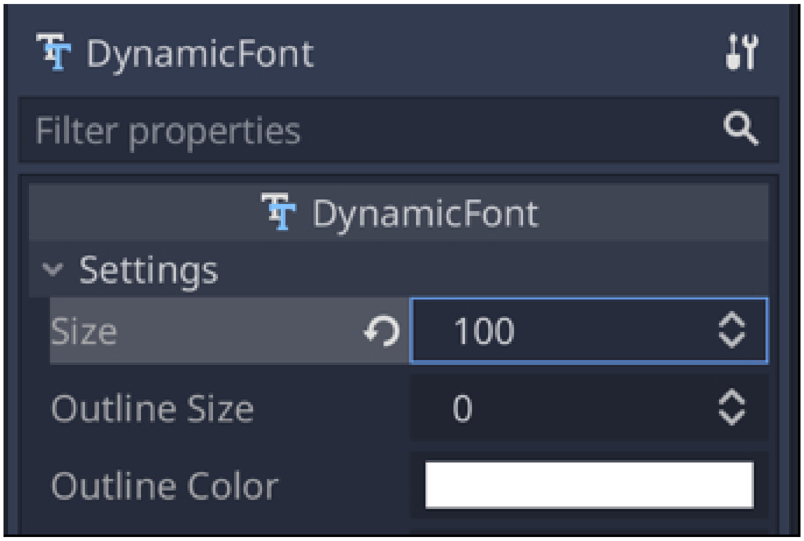


- In the new window, expand the section called **Font** and drag and drop the file called **BebasNeue-Regular.ttf** to the section called **Font Data**, as illustrated in the next figure.



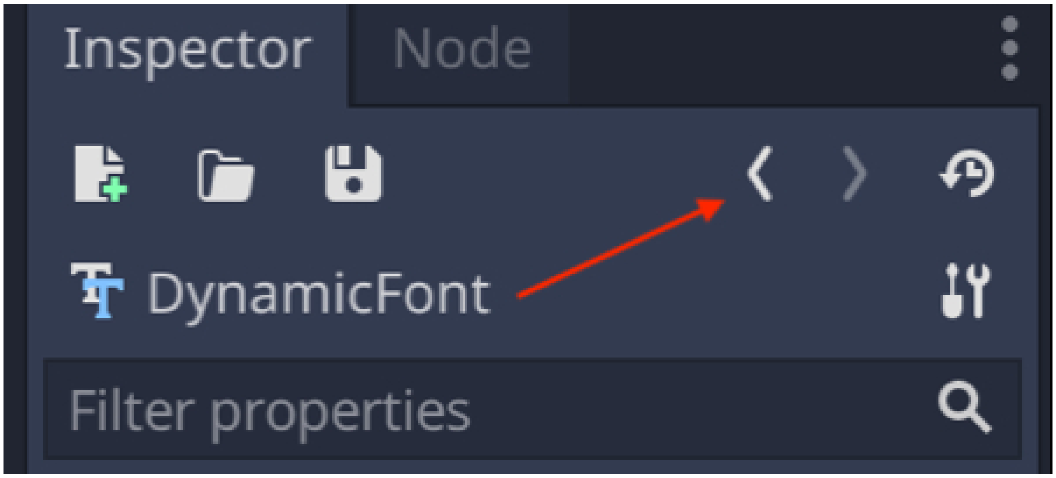
Size 100
Outline Size 0
Outline Color
Use Mipmaps On
Use Filter On
Font Data ABG

- You can now expand the section called **Settings**, and set the font size to **100**.



Size 100
Outline Size 0
Outline Color

- Click on the left arrow at the top of the **Inspector** to go back to the description of the node, as per the next figure.



- Once this is done, you should be able to see the section called **Text** for the node **message**.
- Please enter the text **Test** in the **Text** section and you should see it appearing on the interface.

We can then modify the script **Player.gd** so that it displays a message when all spheres have been collected.

- Please modify the script **Player.gd** and add this code just before the function **_ready**:

```
onready var user_message:Label = get_node("../message")
```

In the previous code we create a new variable called **user_mesage** that is linked to the **Label** that we have just created, so that we can modify its text content.

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

```
user_message.set_text("")
```

In the previous code, we just empty the text from the label that we have previously created.

- Finally, add this code to the function **physics_process** (new code in bold).

```
elif (collision.collider.name == "end" && score == 4):


print("Congratulations")
user_message.set_text("CONGRATULATIONS")
```

In the previous code, we display the "**Congratulations**" message in the label that we have previously created.

LEVEL ROUNDUP

Well this is it!

In this chapter, we have learned about instantiating objects in GDScript. We have created a very simple AI-driven launcher that follows the player and shoots projectiles in his/her direction, and at a controlled rate. We also managed to create explosions upon collision. Finally, we created a scoring system and a mini-game whereby the player has to collect four objects, and also avoid the projectiles, before reaching the end of the level. So yes, we have made some considerable progress, and we have by now looked at some simple ways to implement artificial intelligence in our games.

Checklist

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

- Create scenes from existing nodes.
- Instantiate nodes based on scenes.
- Shoot projectiles.
- Display messages onscreen.

Quiz

It’s now time to check your knowledge with a quiz. So please try to answer the following questions. The solutions are included in your resource pack. Good luck!

1. The event **body_entered** is called whenever a collision occurs between a **RigidBody** node and another node.
2. To be able to access a variable from a script through the **Inspector**, this variable can be declared preceded by the keyword **export**.
3. Write the missing line in this code to be able to destroy the object we have collided with, assuming that this object is referred to as **body**.

```
func ball_collision (body):
<MISSING CODE>
```

1. A node can be saved as a separate scene and re-used in any other scene.
2. A collision shape can be used to detect collision with a node.
3. Find one error in the following code.

```
_ready ()
{
var score_int = 0;
get_the_node(“../Label").set_text(“”);
}
```

1. Any object selected in the **Hierarchy** window can be duplicated using the shortcut *CTRL + D*.
2. If the node attached to the next script is a **Rigidbody**, the following code will access this node’s velocity.

```
linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

1. The following code will cause the current node to look towards the node called player.

```
look_at_the_node(get_node("../player").global_transform.origin,Vector3.UP)
```

1. The method **is_in_group** can be used to determine if a node is part of a specific group.

Solutions to the Quiz

Quiz

It’s now time to check your knowledge with a quiz. So please try to answer the following questions. The solutions are included in your resource pack. Good luck!

1. TRUE.

2. TRUE.

```
func ball_collision (body):
body.queue_free()
```

1. TRUE.

2. TRUE.

3. Find one error in the following code.

```
func _ready ()
{
var score_int = 0;
get_the_node(“../Label").set_text(“”);#it should be get_node
}
```

1. TRUE.

2. FALSE (the second parameter is missing); it could be instead.

```
new_ball.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

1. FALSE. It should be **look_at_node** instead of **look_at_the_node**.

```
look_at_node(get_node("../player").global_transform.origin,Vector3.UP)
```

1. TRUE.

Challenge

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

- Change the onscreen message to "**You have collected an object**" every time the user has collected an object.
- Display the score onscreen (i.e., create a new **Label** node and access it whenever the score is updated).
- Create a timer so that this message is displayed only for 5 seconds.

Chapter 2: Creating and Managing Weapons

In this section we will discover how to create and manage weapons using a simple inventory system.

After completing this chapter, you will be able to:

- Create different weapons including a gun, an automatic gun, and a grenade launcher.
- Collect and manage ammunition.
- Switch between weapons.
- Aim at targets using ray-casting and a crosshair.
- Detect objects in the distance based on the crosshair.

In this section, we will be creating a training camp with the following features:

- The player will avail of three different weapons.
- The player will need to hit specific targets.
- The player will be able to collect ammunition when needed.

Setting-up the environment

In this section we will create a very simple environment for this training camp.

So let's get started:

- Please save the current scene (**Scene | Save Scene**).
- Using the **FileSystem** window, duplicate (i.e., select the scene and then press **CTRL + D**) the scene that you have just saved (e.g., **level1.tscn**) and rename the duplicate **training_camp**.
- Open the scene **training_camp**.
- Delete the following nodes: all the launchers (i.e., **launcher1**, **launcher2**, **launcher3**), all the spheres to be collected (i.e., **sphere1**, **sphere2**, **sphere3** and **sphere4**), the nodes **start** and **end**, and the node **ball**.
- So after removing these nodes, you should only have the following nodes in your scene: the root node (i.e., **Spatial**), and the nodes **player**, **ground**, and **message**.
- After deleting these nodes, please play the scene and check that you can still navigate around the scene.
- You may also create a new folder to store assets created in this scene: in the **FileSystem** window, right-click on the folder labelled "**res://**", select the option "**New Folder**" from the contextual menu, and give a name to your new folder; for example, "**training_camp**".

We will now create three targets that will be used for the training:

- Please create a new **StaticBody** node as a child of the **Spatial** node, rename it **target1**, add a child of type **CollisionShape** to that node (i.e., **target1**), set its collision shape to a **box** shape, and finally, add a child node of type **CSGBox** to this **CollisionShape** node.
- Select the node **target1** and change its position to **(0, 5, 0)** and its scale to **(1, 5, 0)**.
- You can also paint the node **CSGBox** in red by applying a new (or existing) **Spatial Material**: using the **Inspector**, click on the downward-facing arrow to

the right of the label “**Material**”, select the option “**New Spatial Material**”, click on the white sphere, expand the **Albedo** section, and choose a color.

- You can now duplicate this target three times and rename these targets **target2**, **target3**, and **target4**.
- Move these targets apart, so that they are aligned and about six meters apart; for example, they could be at the positions **(6, 5, 0)**, **(12, 54, 0)** and **(18, 5, 0)**.

Detecting objects ahead using raycasting

At this stage, we would like to detect what is in front of the player so that when we use our weapon (e.g., gun), we know whether an object is in the line of fire before shooting. To do so, we will use ray-casting. So we will create some code to be able to detect what is in front of the player using a ray.

A ray is a bit like a laser beam; it is casted over a distance and usually employed to detect if an object is in the line of fire. In our case, we will cast a ray from the player (forward) and detect if it "collides" with another object. Rays can be used for many applications, from weapons to controlling objects (e.g., opening a door only if you are facing it rather than using collision).

- Please open the script called **Player.gd**
- Add this code just before the function **_ready**.

```
onready var camera :Camera = get_node("Camera")
var ray:RayCast
```

In the previous code, we create a new ray along with the variable camera that is linked to the camera included in the First-Person Controller; this is so that the ray-casting follows where the player is looking.

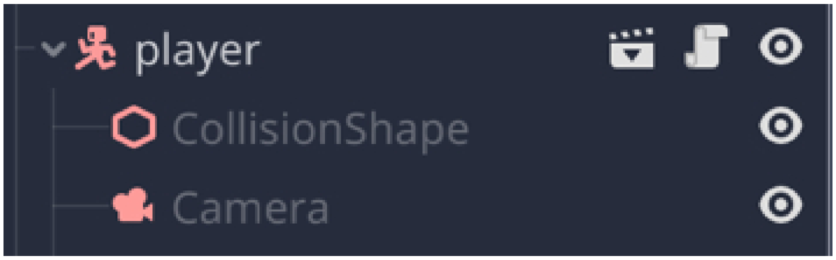
- Add this code to the function **_ready**.

```
ray = RayCast.new()
ray.enabled = true
camera.add_child(ray)
ray.cast_to = Vector3(0,0,-100)
```

In the previous code:

- We instantiate a new ray.
- We then enable this ray.
- The ray becomes a child of the **camera** node that is part of the First-Person Controller.
- Finally, the ray will be cast 100 meter forward.

- You may just check that there is a camera attached to the **First-Person Controller** using the **Inspector** window: if you right-click on the node **player**, and select the option “**Editable Children**” you will be able to expand the **player** node and see the children nodes within, including the **camera** node, as per the next figure.



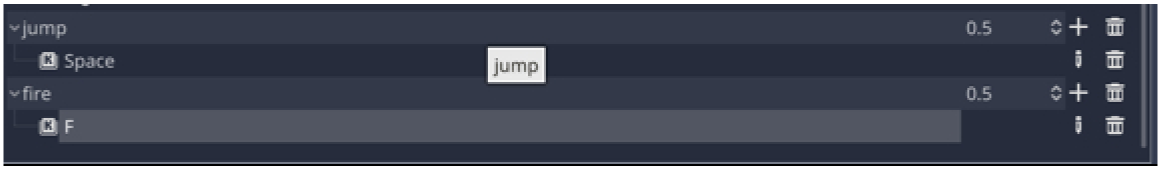
- Please add the following code to the function **physics_process** (in the script **Player.gd**) as follows:

```
if (Input.is_action_just_pressed("fire") ):
if ray.is_colliding():
var obj = ray.get_collider()
print("the object " +obj.get_name()+" is in front of the player")
```

In the previous code, we do the following:

- We check that the “**fire**” key has been pressed. For this to happen, please make sure that the key “**F**” is mapped to the action “**fire**” in the **Project Settings** (see next instructions on how this could be done).
- If that’s the case, we check if the ray that we have created earlier is colliding with another object
- If there is a collision, then we obtain the name of the object colliding with the ray and print a message onscreen with the name of that object.

Before we can test this feature, we just need to map the “**fire**” key to the key **F**; so please open the **Project Settings**, as you have done previously (**Project |Project Settings | Input Map**), create a new action called “**fire**” and link it to the key **F**.



jump 0.5
Space
fire 0.5
F

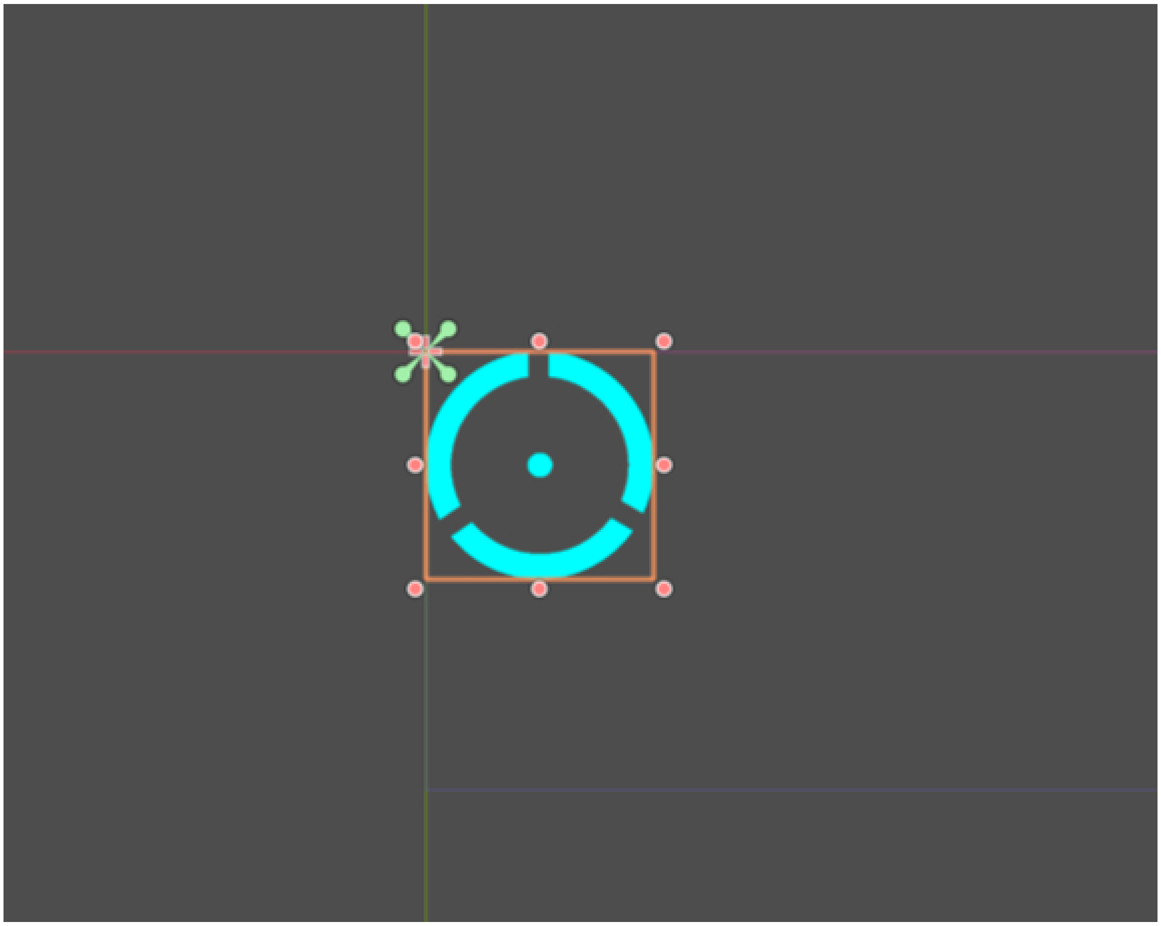
Please play the scene, and as you walk towards one of the targets (for example **target1**) and press the **F** key, the message **“target1 is in front of the player”** should be displayed in the **Output** window.

Creating a weapon

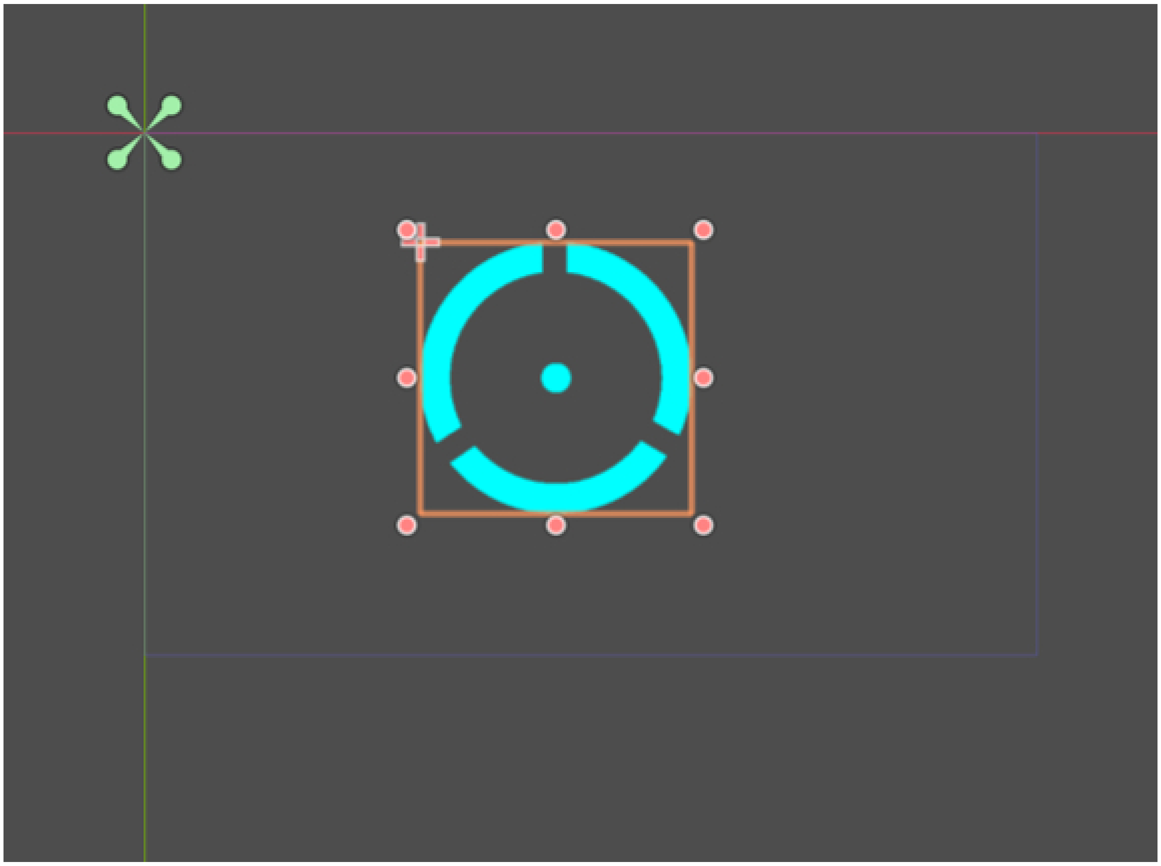
So, well done: at this stage we can cast rays and detect objects in front of the player. So the next step for us is to create and fire a weapon.

The first step will be to make shooting more accurate by adding a crosshair.

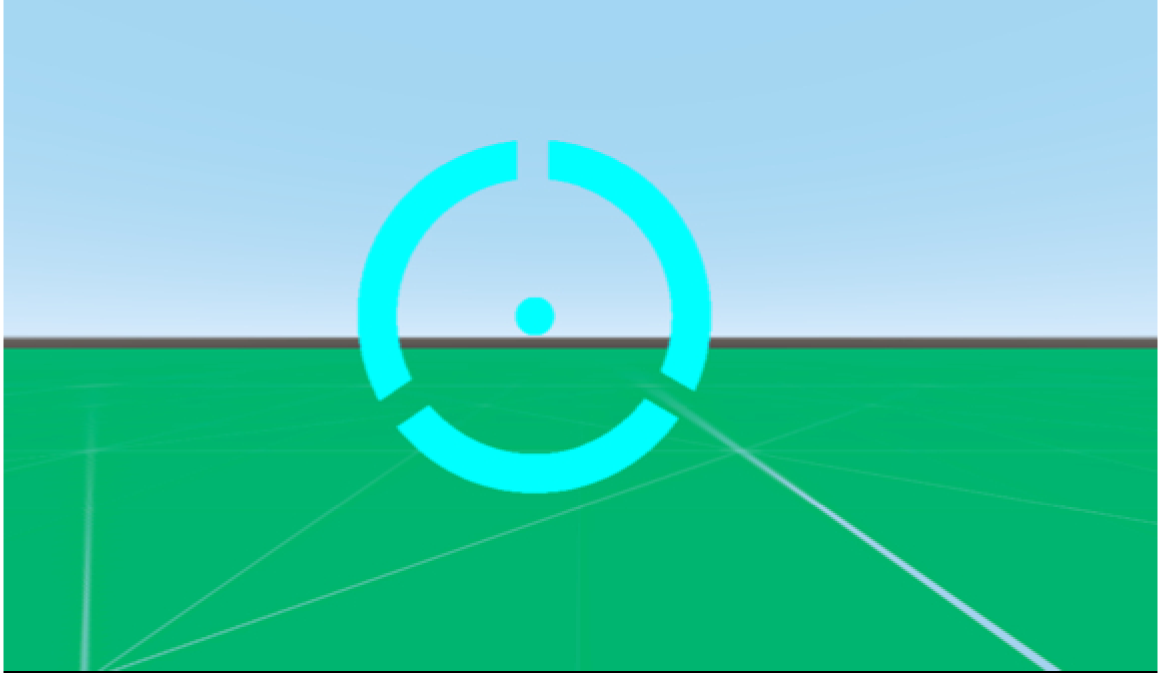
- Please locate the file **crosshair.png** in the **FileSystem** window, as we will be using it for the interface.
- Create a new node of type **TextureRect** as a child of the node **Spatial**. This will create a new node called **TextureRect**. Please rename this node **crosshair**.
- Drag and drop the texture **crosshair** from the **FileSystem** window to the **Texture** attribute in the **Inspector** for the node **TextureRect**. This should display the crosshair onscreen.



- Move the image to the middle of the screen using the move tool.



- Once this is done, if you play the scene, you should now see the crosshair onscreen, as per the next figure.



One more thing; while you probably centered the crosshair manually, you can ensure that it is done perfectly through code as follows:

- Please select the node **crosshair.**
- Add a new script to this node, and save the script as **crosshair.gd**.
- Add the following code to the function **_ready**.

```
var viewport_size = get_viewport().size
self.rect_position = viewport_size/2 - self.rect_size/2
```

In the previous code, we obtain the size of the viewport (the window used to display the game), and we then set the position of the rectangle defining the crosshair based on its own size and the size of the viewport.

Last but not least, we will try to manage ammunition for this particular gun. You see, at present, the player can shoot indefinitely; so we could just give the player an initial amount of bullets, and make it possible to fire the gun only if there are bullets left.

- Please add the following code just before the function **_ready** in the script **Player.gd**.

```
var gun_ammo = 3
```

In the previous code, we declare a new variable **gun_ammo** that will be used to store the number of ammunition left.

- Then, we can modify the code to manage these ammunition, as follows (new code in bold) in the function **_physics_process**:

```
if (Input.is_action_just_pressed("fire") && gun_ammo > 0):
if ray.is_colliding():
gun_ammo -= 1
var obj = ray.get_collider()
print("the object " +obj.get_name()+" is in front og the player")
print("you have "+ str(gun_ammo)+" ammunition left")
```

In the previous code:

- We check that we have enough ammunition before firing the gun.
- If this is the case, we decrease the number of bullets left.
- We then display the number of bullets left.

Please play the scene, and check that after shooting three times, you can no longer fire the gun.

Managing Damage

So at this stage, we have managed to create a weapon and fire bullets precisely using ray-casting and a crosshair. This being said, it would be great to be able to manage the targets (that we will refer to as NPCs in this section) by knowing how many times they have been hit and when they should be destroyed (e.g., after being hit five times). So for this purpose, we will create a script that will store the NPC's health, count how many times it was hit, decrease its health whenever it has been hit, and destroy it after its health has reached 0.

So let's create our script.

- Please select the node **target1**.
- Right-click on this node and select the option "**Attach Script**".
- In the new window, modify the path to "**res://ManageNPC.gd**", leave all the other options as default, and click "**Create**".
- This should open the script **ManageNPC**.
- Add the following code (new code in bold).

```
var health:int
func _ready():
health = 100
func got_hit():
health -= 20
print("Health"+str(health))
if (health <=0): destroy_target()
func destroy_target():
queue_free()
```

In the previous code:

- We declare the variable **health** which is used to track the NPCs' health.
- We then initialize the health to 100 in the **_ready** function.
- We also create a method **got_hit**. This method will be called whenever the object has been hit; when this happens, the health is decreased by **20**.

- Finally, we create a function called **destroy_target** that is called whenever the health of the NPC is zero.

Once these changes have been made, we just need to modify the script **Player.gd** so that we can modify the health of each NPC when it has been hit:

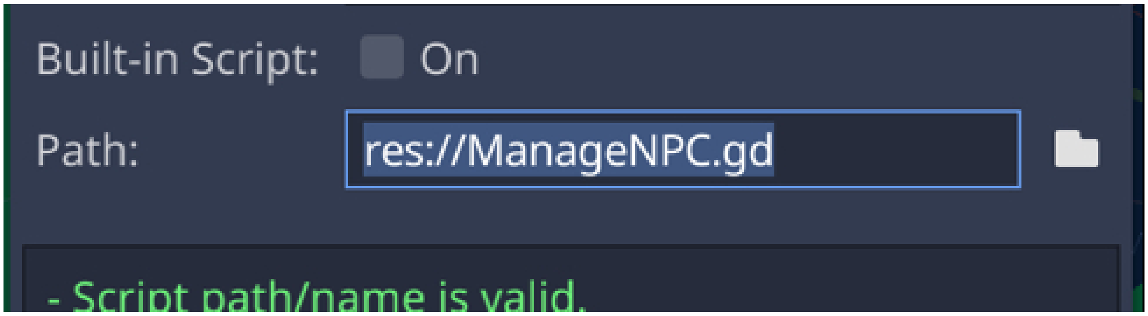
- Please open the script **Player.gd**.
- Modify the code as follows (new code in bold).

```
if (Input.is_action_just_pressed("fire") && gun_ammo > 0):
if ray.is_colliding():
gun_ammo -= 1
var obj = ray.get_collider()
var sphere=CSGSphere.new()
print("the object " +obj.get_name()+" is in front og the player")
print("you have "+ str(gun_ammo)+" ammunition left")
if (obj.is_in_group("target")):
obj.got_hit()
```

In the previous code, if the tag of the object detected by the raycast is **target** we will access its script and call the function **got_hit**.

Next, please attach the script **ManageNPC** to the other targets.

- Right-click on the target.
- Select the option “**Attach Script**” from the contextual menu.
- Enter the path “**res://ManageNPC.gd**” in the path section.



Built-in Script: □ On

Path: res://ManageNPC.gd

- Script path/name is valid.

- Press the button labelled **Load**.
- Repeat these steps for each target.

The last thing we need to do is to create a tag called **target** (using the **Inspector**, as we have done before) and apply it to all the targets. We can also set the initial number of bullets, **gun_ammo**, to **10** (instead of three, in the script **Player.gd**) so that we can test the game properly.

- Please make these changes (i.e., add the group **target** to all targets and set the initial number of ammos to **10**).
- Please play the scene.
- Shoot at each target five times and check that they disappear.

Collecting and managing ammunition

At this stage, the game level is working well, however our player may run out of ammunition. So it would be good to create ammunition that can be collected by the player. To do so, we will create and texture boxes that will be used as ammunition; we will also give them a group, and detect whenever the player collides with them. We will also get to create templates (i.e., new scenes based on these nodes) with these so that they can be reused later (i.e., in different levels).

So let's create these ammunition boxes:

- Please duplicate the node called **ground** and rename the duplicate **ammo**.
- Select the node called **ammo**.
- Change its scale property to **(1, 1, 1)**.
- Move it slightly away from the targets and above the ground, for example at the position **(-10.0, 2, 4)**.
- Use a texture of your choice or import a texture from the resource pack and apply it to the box.
- Create a new tag called **ammo_gun** and apply it to this cube.

Once this is done, we just need to detect collisions between this cube and the player:

- Please open the script **Player.gd**.
- Add the following code to this script in the function **physics_process** (new code in bold):

```
elif (collision.collider.name == "end"):# && score == 4):
print("Congratulations")
user_message.set_text("CONGRATULATIONS")
elif (collision.collider.is_in_group("ammo_gun")):
gun_ammo += 5
if(gun_ammo == 10): gun_ammo = 10
collision.collider.queue_free()
```

In the previous code:

- We check if we have collided with gun ammunition.
- If this is the case, we increase the ammunition for the player by **5**.
- We then check if we have reached the maximum number of ammunition that we can carry.
- If this is the case, we cap the number of ammunition to this maximum.
- We then destroy the ammo pack.

Please save your code, check for any errors and play the scene. As you play the scene, please check that you can shoot at the targets several times, that you can collect ammunition, and also that your ammunition levels have been updated accordingly.

Building a weapon management system with classes

At present, we have a relatively simple weapon management system that works with one weapon. We can collect ammunition and also shoot targets. However, in the next sections, we will be adding more types of weapons (i.e., an automatic gun and grenades), so we need to find a way to manage these simply, using structures that make it easy to track the ammunition for each of them or the time it takes to reload a weapon. So, before even creating new weapons, we will make sure that we have a structure in place that will make it possible to track the following information, for each of them:

- The reload time for this weapon.
- The name of the weapon.
- The ammunition that the player is carrying (or currently has) for this weapon.
- The maximum number of ammunition that the player can carry for this weapon.

To do so, we will be using a combination of two classes and the process will be as follows.

- We will create a class for the weapons that will include information on each weapon's type, name, reload time, ammunition and maximum ammunition.
- We will then create a class for the weapon inventory that will be used to manage the weapons that the player has (e.g., change weapon, change ammunition, etc.) .
- Whenever the player presses the tab key, we will switch between weapons.
- When the player presses the **fire** key, we check that there is enough ammunition for the current weapon, and then, if this is the case, we will fire this weapon.
- As the player tries to shoot another time, we check the reload time for this particular weapon (e.g., 2 for a normal gun, .5 for an automatic gun, etc.).
- When we collect an ammo pack, we check its type, and also update the ammunition levels for the corresponding weapon.

So, this is the general idea of how things will work; now that it is clearer, let's implement the corresponding code.

First let's create the class **Weapon**:

- Please create a new script and rename it **Weapon.gd**: switch to the **Script** workspace, then select **File | New Script**.
- Once the script is open, please remove all of its content and add the following code instead:

```
class_name Weapon
const TYPE_GUN:int = 0
const TYPE_AUTO_GUN:int = 1
const TYPE_GRENADE:int = 2
var reload_time:int
var name:String
var ammos:int
var max_ammos:int
```

In the previous code:

- We declare the name of the class.
- We create three constants that will correspond to the three types of weapons available to the player.
- We then declare four variables that will be used to store information related to a weapon's reload time, name, number of ammos, and maximum number of ammos.

Now that we have defined member variables for this class, it is time to define a constructor that will be used to instantiate new weapons; so please add the following code just after the previous code:

```
func _init(weapon_type:int=TYPE_GUN):
match weapon_type:
TYPE_GUN:
```

```
name= "GUN"
reload_time=2
ammos=10
max_ammos=20
TYPE_AUTO_GUN:
name= "AUTOMATIC GUN"
reload_time=1
ammos=20
max_ammos=20
TYPE_GRENADE:
name= "GRENADE"
reload_time=3
ammos=10
max_ammos=5
print("Created Weapon Type: "+str(weapon_type))
```

In the previous code:

- We create the function **_init** which is the constructor for the class **Weapon**.
- This constructor takes the weapon's type as a parameter, and if none is specified, the new weapon will be a gun.
- We then use a match structure to set-up the new instance of the class **Weapon** based on the type of weapon to be created and we initialize its name, reload time, initial ammo and maximum ammo.

Finally, we just need to add member functions that can be used to decrease or increase the number of ammunition for a given weapon; so please add the following code to the class.

```
func increase_ammo(ammo_increase:int = 1):
if (ammos + ammo_increase <= max_ammos):
ammos+=ammo_increase
func decrease_ammo(ammo_decrease:int = 1):
ammos -= ammo_decrease
if ammos < 0: ammos=0
```

In the previous code:

- We declare two functions: **increase_ammo** and **decrease_ammo**.
- The first function takes one parameter which corresponds to the increase to be applied for the ammunition; if none is specified then the increase in ammunition will be **1** by default; however, the increase is applied only if it results in a number of ammunition that is less than the maximum allowed.
- The second function takes one parameter which corresponds to the decrease to be applied for the ammunition; if none is specified then the decrease in ammunition will be **1** by default; we also ensure that the minimum number of ammunition is 0.

Now that the class **Weapon** has been defined, we will define a new class called **WeaponInventory**.

- Please create a new script and rename it **WeaponInventory.gd**.
- Once the script is open, please remove all of its content and add the following code instead:

```
class_name WeaponInventory
    var Weapon = load("res://Weapon.gd")
    var weapons = []
    var weapon_index:int = 0
    func _init():
    var weapon1 = Weapon.new(Weapon.TYPE_GUN)
    var weapon2 = Weapon.new(Weapon.TYPE_AUTO_GUN)
    var weapon3 = Weapon.new(Weapon.TYPE_GRENADE)
    weapons=[weapon1,weapon2,weapon3]
```

In the previous code:

- We state that the name of this class is **WeaponInventory**.
- We load the class **Weapon** so that instances of this class can be created.
- We declare an array called **weapons** along with a variable **weapon_index** that will be used to access a specific weapon from this array.
- Finally, we create three instances of the class **Weapon** (i.e., **weapon1**, **weapon2**, and **weapon3**) and add them to the array weapons. These will be used for the three different types of weapons that will be available to the player.

Please save this file.

Now that the different types of weapons have been created, we will create functions that will help with the management of these weapons; so please add the following code to the class **WeaponInventory**.

```
func change_weapon():
weapon_index += 1
if weapon_index>2:weapon_index=0
func has_ammo_for_current():
return(weapons[weapon_index].ammos>0)
func decrease_curr_ammo(amount:int=1):
weapons[weapon_index].decrease_ammo(amount)
func get_curr_reload_time():
return weapons[weapon_index].reload_time
func get_curr_weapon_name():
return weapons[weapon_index].name
func get_curr_weapon_ammos():
return weapons[weapon_index].ammos
```

In the previous code, we create several functions:

- **change_weapon**: this function modifies the index for the current weapon.
- **has_ammo_for_current**: this function returns whether the player has ammunition left for the weapon currently selected.

- **decrease_curr_ammo**: this function decreases the ammunition for the weapon currently selected.
- **get_curr_reload_time**: this function returns the reload time for the weapon currently selected.
- **get_weapon_name**: this function returns the name for the weapon currently selected.
- **get_curr_weapon_ammo**: this function returns the number of ammunition for the weapon currently selected.

Please save this file.

Now that we have defined these two classes, it is time to use them in the script **Player.gd** so that the player can switch between and use weapons.

- Please open the script **Player.gd**.
- Add the following code before the function **_ready**.

```
var inventory : WeaponInventory
var reload_timer :Timer
var can_shoot = true
```

In the previous code: we declare three variables **inventory**, **reload_timer**, and **can_shoot**; these will be used to respectively create a weapon inventory, set a delay (reload time) between shots, and determine whether the player can shoot (based on whether the reload time has elapsed).

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

```
reload_timer = Timer.new()
add_child(reload_timer)
reload_timer.connect("timeout",self,"reload_timer_timeout")
inventory=WeaponInventory.new()
```

In the previous code:

- We initialize the variable **reload_timer**.

- This timer is then added as a child of the current node (i.e., **player**).
- We specify, using the built-in function **connect**, that when the timer times out, the function **reload_timer_timeout** (located in this script) should be called.
- Finally, we instantiate a new instance of the class **WeaponInventory**.

Now that the inventory has been created, we can start to write some code to manage this inventory and when the player uses weapons within the inventory.

Please add the following code at the end of the function **_physics_process**:

```
if (Input.is_action_just_pressed("change_weapon")) :
inventory.change_weapon()
var message = inventory.get_curr_weapon_name()
message += "(" + str(inventory.get_curr_weapon_ammos()) + ")"
print(message)
reload_timer.wait_time=inventory.get_curr_reload_time()
```

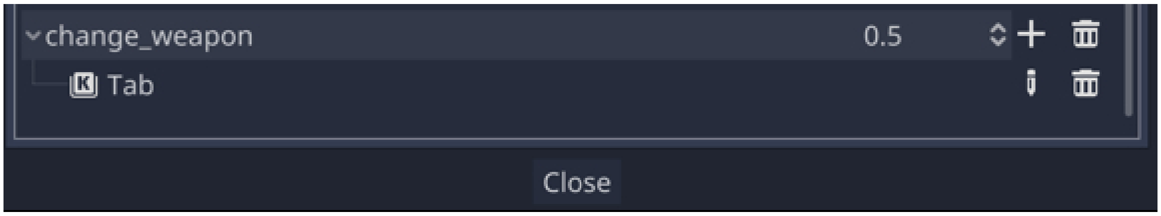
In the previous code:

- We check that the button mapped as "**change_weapon**" is pressed.
- If that's the case, we call the method **change_weapon** and we display the name of the current weapon in the **Output** window.
- We display a message that includes the name of the current weapon along with corresponding ammunition.
- Finally, we set the **wait_time** attribute of the timer according to the reload time for the current weapon.

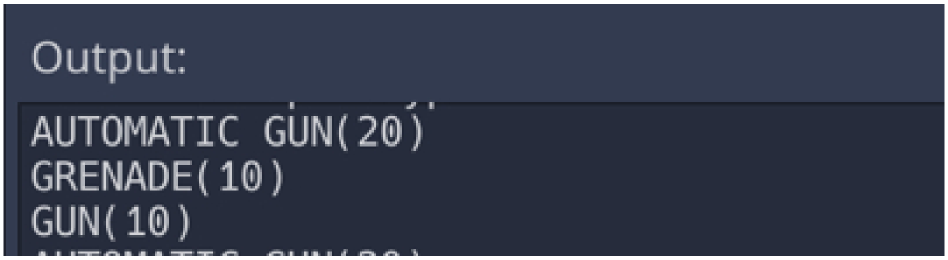
Before we can test this code, we need to map the action "**change_weapon**" to the **TAB** key, so please do the following:

- Select: **Project | Project Settings | Input Map**.
- Type **change_weapon** in the action field and click "**Add**".
- Scroll down to the item that you have created in the list "**change_weapon**", click on the + button to the right of the window, select "**Key**" from the

drop-down list, and press **Tab**.



Once this is done, please save your code, and test the scene. As you press the *Tab* key, you should see the message “**AUTOMATIC GUN (20)**” and “**GUN (10)**” in the **Output** window.



Output:

```
AUTOMATIC GUN(20)
GRENADE(10)
GUN(10)
```

Well, our system is working properly; now we just need to link it to the firing system, so that we can shoot depending on the current weapon and ammunition available for this weapon.

First we will define the function **reload_timer_timeout**; so please add the following code to the **Player.gd** script outside of any function:

```
func reload_timer_timeout():
can_shoot = true
reload_timer.stop()
```

- Please modify the **function _physics_process** and replace this code...

```
if (Input.is_action_just_pressed("fire") && gun_ammo > 0):
if ray.is_colliding():
gun_ammo -= 1
var obj = ray.get_collider()
print("the object " +obj.get_name()+" is in front og the player")
print("you have "+ str(gun_ammo)+" ammunition left")
if (obj.is_in_group("target")):
```

```
obj.got_hit()
```

... with this code...

```
if (Input.is_action_just_pressed("fire")):
var condition1 = (inventory.weapon_index == Weapon.TYPE_GUN)
var condition2 = (inventory.weapon_index == Weapon.TYPE_AUTO_GUN)
var condition3 = inventory.has_ammo_for_current()
var condition4 = can_shoot
if (( condition1|| condition2) && condition3 && condition4):
inventory.decrease_curr_ammo()
can_shoot=false
reload_timer.wait_time=inventory.get_curr_reload_time()
reload_timer.start()
```

In the previous code:

- If the player presses the “**fire**” key, we check if s/he is using a gun or an automatic gun.
- We also check that there is enough ammunition for the weapon selected, and that the reload time has elapsed.
- If these conditions are fulfilled, we then decrease the ammunition for the active weapon and we set the variable **can_shoot** to false
- We also set the **wait_time** attribute of the time based on the reload time for the current weapon.
- Finally, we restart the timer so that a new delay (reload time) can be applied.

At this stage, we just need to detect the object in front of the player; so please add the following code after the previous code that you have just typed (new code in bold):

```
reload_timer.start()
if ray.is_colliding():
var obj = ray.get_collider()
if (obj.is_in_group("target")):
```

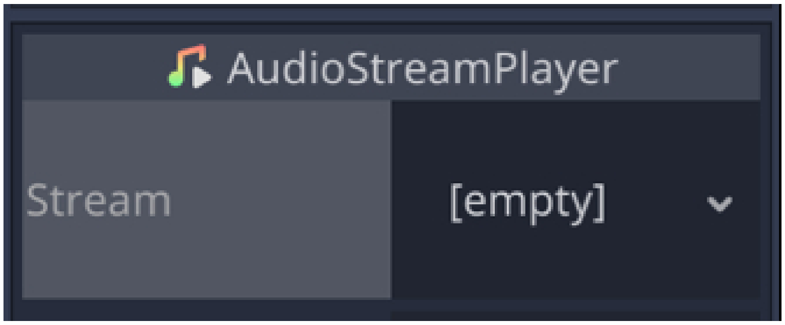
obj.got_hit()

In the previous code, we detect whether the raycast has collided with a target. If this is the case, we access the function **got_hit** that is in the script attached to the target.

Adding sound effects

To make sure that we hear when the gun is shot, and to tell the difference between the two guns, we will also add a sound when one of these is fired.

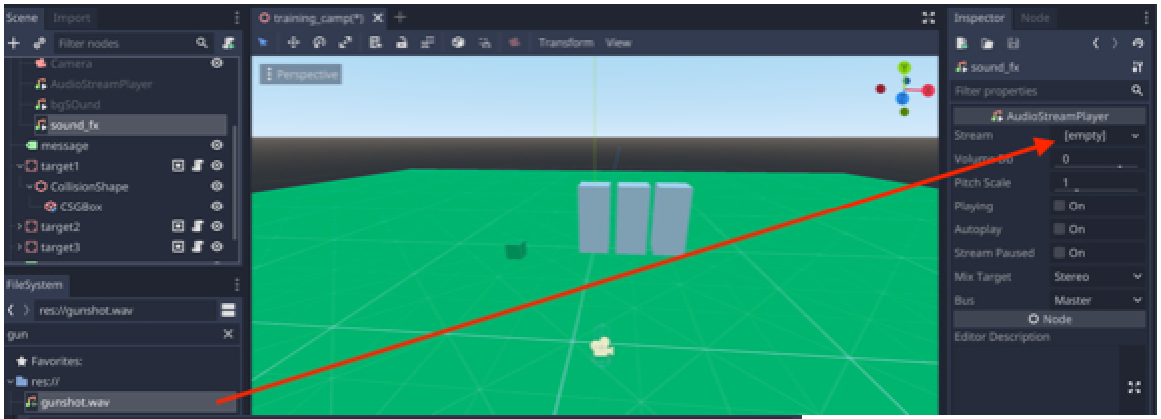
- Please locate the file **gunshot.wav** in the **FileSystem** window.
- Select the node **player** in the **Scene Tree**.
- Add a node of type Audio **AudioStreamPlayer** as a child of the node **player**.
- Once this is done, a new **AudioStreamPlayer2** node should be added as a child of the node player. Please rename it **sound_fx**.
- You can look at its properties in the **Inspector** window.
- As you look at the **Inspector** window, you will notice a section called **Stream** as per the next figure.



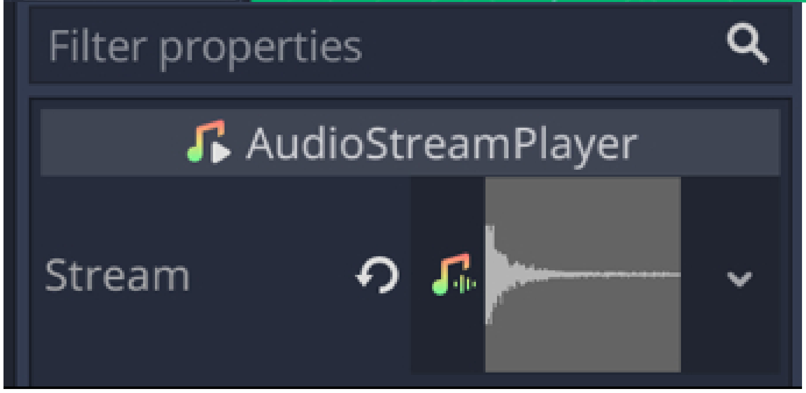
AudioStreamPlayer

Stream [empty]

- Please locate the file **gunshot** in the **FileSystem** window and drag and drop it to the “**empty**” slot to the right of the label **Stream** in the **Inspector**, as illustrated in the next figure.



Stream [empty]
Volume dB 0
Pitch Scale 1
Playing On
Autoplay On
Stream Paused On
Mix Target Stereo
Bus Master



Once this is done, we can trigger this sound through our script.

- Please open the script **Player.gd**.
- Add this code just before the function **_ready**.

```
onready var  sound_fx:AudioStreamPlayer = get_node("sound_fx")
```

In the previous code, we create a variable that is linked to the node “**sound_fx**” so that we can access this node and play the corresponding sound effect.

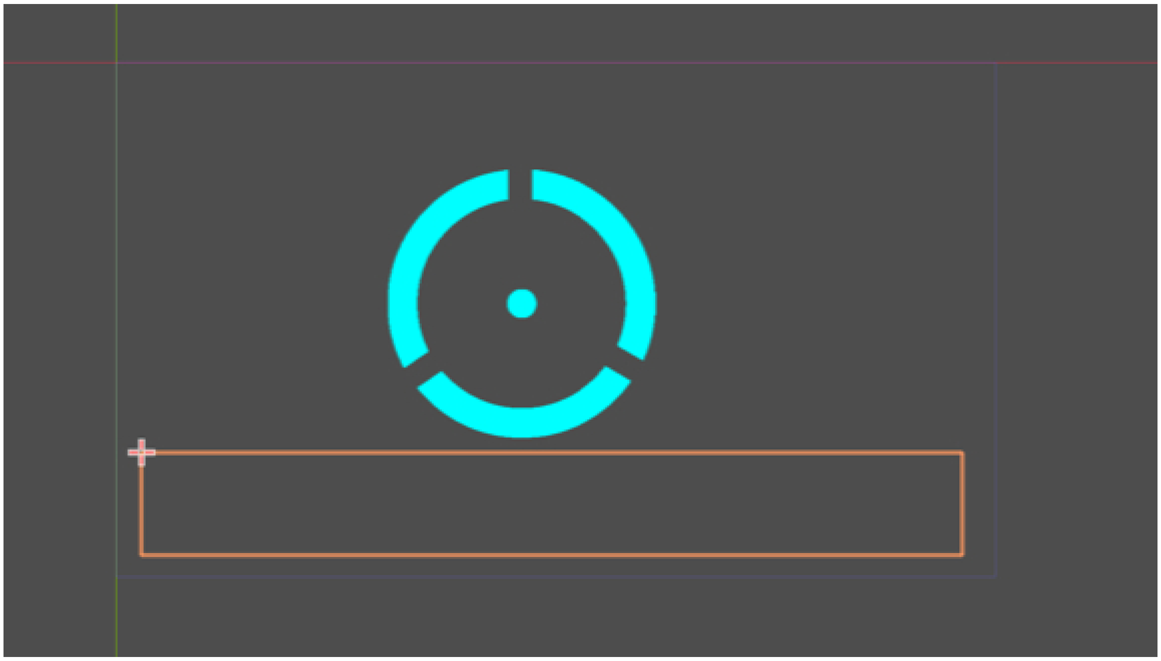
- Please modify the function **_physics_process** as follows (new code in bold):

```
reload_timer.wait_time = inventory.get_curr_reload_time()
reload_timer.start()
sound_fx.play()
```

In the previous code, we just play the audio file that has been associated with the node **sound_fx**.

The last change that we will make will be to display the current weapon on-screen. For this purpose, we will re-use the node called **message** that we have created previously to display onscreen messages.

- Please select the node called **message** and move it so that it appears at the bottom of the area that delimitates the size of the screen. You can also change its width so that it uses most of the screen's width.



Next, we will modify the text for this object from the script to display the name of the current weapon:

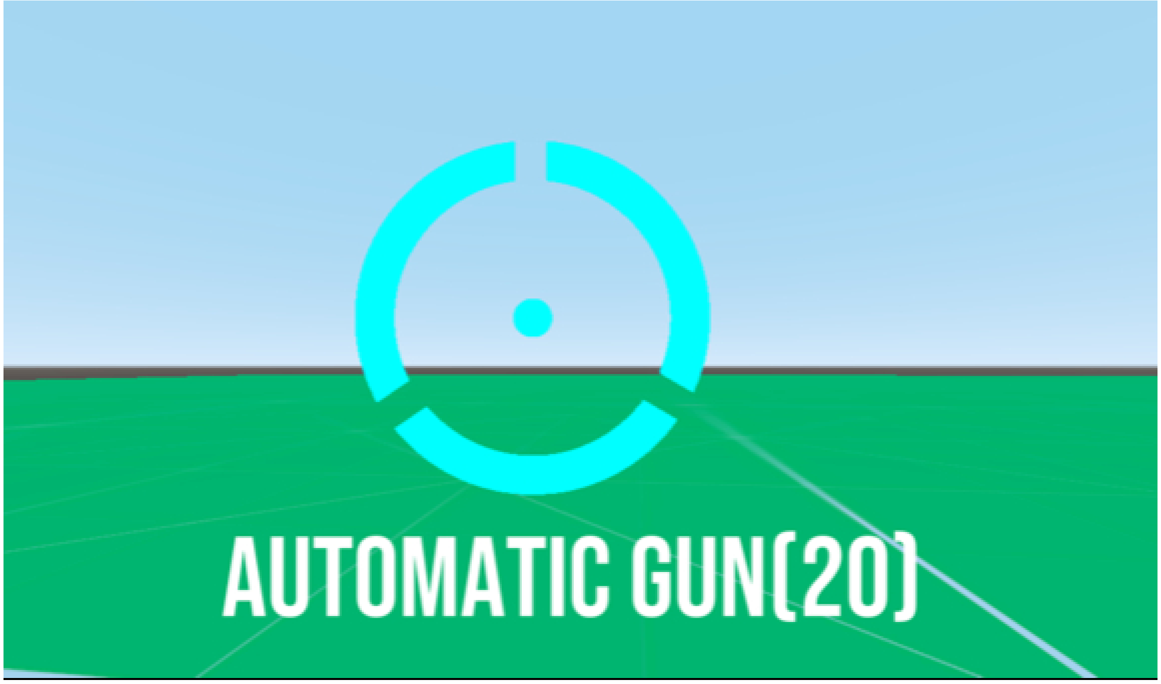
- Please add the following code in the script **Player.gd** for the function **_physics_process** (new code in bold):

```
if (Input.is_action_just_pressed("change_weapon")) :
inventory.change_weapon()
var message = inventory.get_curr_weapon_name()
message += "(" + str(inventory.get_curr_weapon_ammos()) + ")"
print(message)
user_message.set_text(message)
```

In the previous code, we set the text for the node **message** with the information about the current weapon and the corresponding ammunition.

We are now ready to go, so:

- Please save your code.
- Check for any errors.
- Play the scene.
- As you play the scene, try to switch between the two guns, see the information displayed at the bottom of the screen, and how the reload delay varies as you try to press the **F** key several times consecutively.



Now, at this stage, all works well; this being said we could just make a small change; that is, making it possible for the player to shoot repeatedly but without having to press the **F** key again; in other words, the weapon should fire, as long as the **F** key is kept pressed (or is down) and that we have sufficient ammos. For this, we just need to change the type of event detected. Instead of using the built-in function **is_action_just_pressed**, we will use the built-in function **is_action_pressed**.

While the function (**is_action_pressed**) is triggered only when the key has been pressed, the second one (**is_action_just_pressed**) is triggered as long as the key is being pressed.

Please modify the function **_physics_process** for the conditional statement that checks whether the player has pressed the **F** key as follows (new code in bold):

```
if (Input.is_action_pressed("fire")):
var condition1 = (inventory.weapon_index == Weapon.TYPE_GUN)
```

Play the scene and check that you can now fire consecutive shots by just keeping the **F** key pressed.

Managing the collection of ammunition

Well, so far we have managed to define three different weapons (but use only two for the time being) and to fire them based on the corresponding ammunition. What we need to do now is to make it possible, as we have done before, for the player to collect ammunition, and to then update the game information accordingly. For this, we will need to do the following:

- Detect collision with ammo packs.
- Increase the number of ammunition for a given weapon.
- Destroy the ammo pack.

So let's modify the script **Player.gd** by editing the code for the method **_physics_process** as follows.

- Please comment the following lines:

```
#elif (collision.collider.is_in_group("ammo_gun")):
#gun_ammo += 5
#if(gun_ammo == 10): gun_ammo = 10
#collision.collider.queue_free()
```

- Please add the following code just after the previous one:

```
elif (collision.collider.is_in_group("ammo_gun")):
inventory.weapons[Weapon.TYPE_GUN].increase_ammo(10)        collision.col-
collid-er.queue_free()
elif (collision.collider.is_in_group("ammo_auto_gun")):
Inventory.weapons[Weapon.TYPE_AUTO_GUN].increase_ammo(10)
collision.collider.queue_free()
elif (collision.collider.is_in_group("ammo_grenade")):
inventory.weapons[Weapon.TYPE_GRENADE].increase_ammo(10)
collision.collider.queue_free()
```

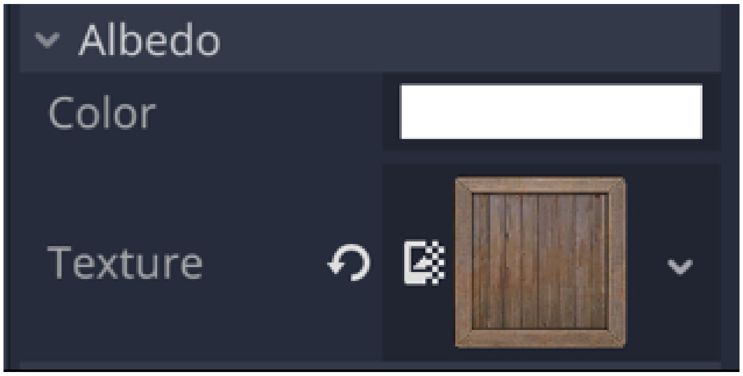
In the previous code:

- We comment the code that we used beforehand to collect ammunition since we are now using an inventory system.
- We then detect the type (i.e., group) of the ammunition box that we have just collided with.
- Depending on the type of ammunition pack, we increase the corresponding ammunition in our inventory and then destroy the ammunition pack.

Please note that we are detecting ammunition for grenades although we have not yet implemented the grenade launcher, but this will be done in the next sections

At this stage, you should have an ammo pack with a label (or group) called **ammo_gun** in your scene.

Before you play the scene, you can, if you wish, use the texture called **box_texture.jpg** located in the **FileSystem** and apply it to the node **CSGBox** that is a child of the node **ammo**. Before using this texture, please create a new **SpatialMaterial** for the **CSGBox** node, so that this texture is only applied to the box and not the ground (since the box was initially a duplicate of the **ground** node).



Before we play the scene, we will ensure that the weapon information is always up to date:

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

```
func update_UI():
var message = inventory.get_curr_weapon_name()
message += "(" + str(inventory.get_curr_weapon_ammos()) + ")"
```

```
user_message.set_text(message)
```

- Modify the code in the **_physics_process** as follows:

```
if (Input.is_action_just_pressed("change_weapon")) :
inventory.change_weapon()
update_UI()
#var message = inventory.get_curr_weapon_name()
#message += "(" + str(inventory.get_curr_weapon_ammos()) + ")"
#print(message)
#user_message.set_text(message)
reload_timer.wait_time=inventory.get_curr_reload_time()
```

- Modify this code (new code in bold)

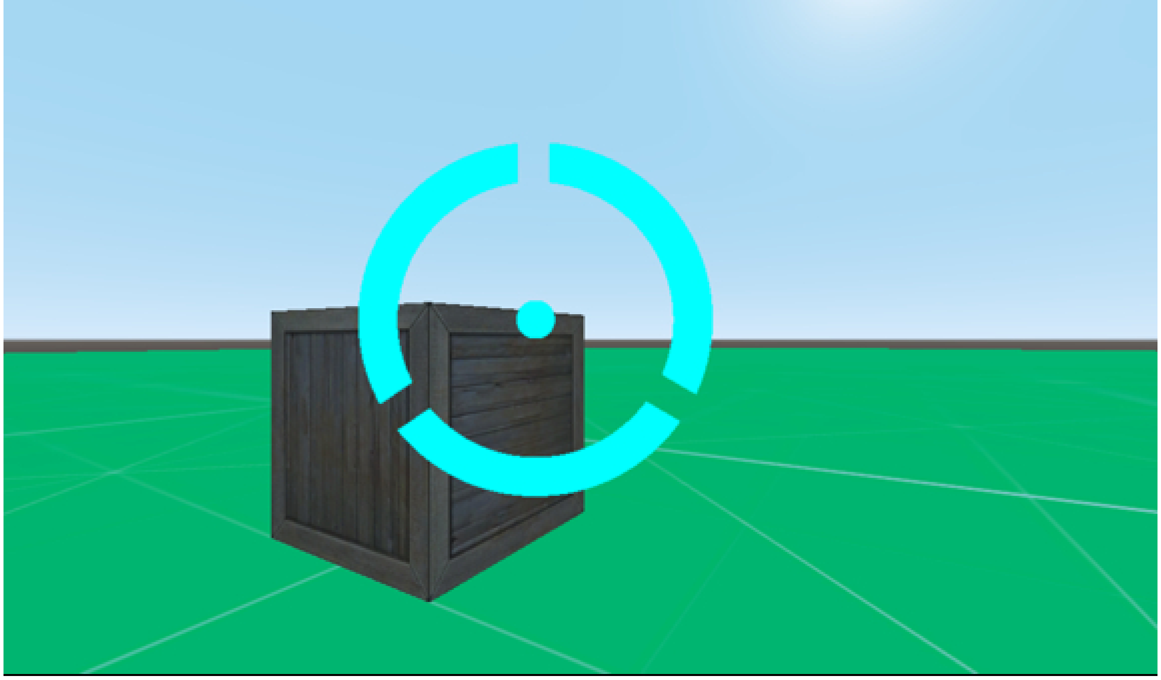
```
if (( condition1|| condition2) && condition3 && condition4):
inventory.decrease_curr_ammo()
```

update_UI()

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

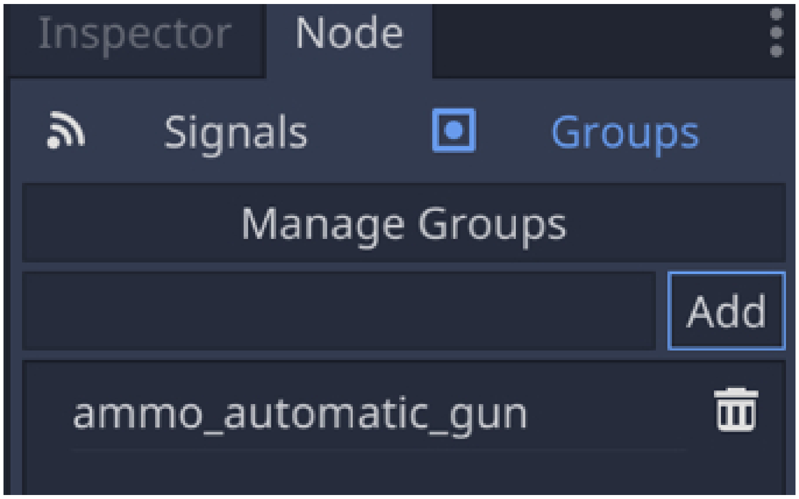
```
update_UI()
```

So, with this in mind, please play the scene, collect this ammo pack and check that the number of ammos for your gun has increased.



We could now duplicate the object used for the ammunition object, and change the duplicate's group to **ammo_automatic_gun**.

- Please rename the node **ammo** to **ammo_gun**.
- Select this node (i.e., **ammo_gun**).
- Duplicate this node (***CTRL* + *D***).
- Rename the duplicate **ammo_auto**.
- Using the **Node | Group** tab, remove the group (i.e., label) **ammo_gun** for this node, then create (and apply) a new group called **ammo_automatic_gun** to this node.

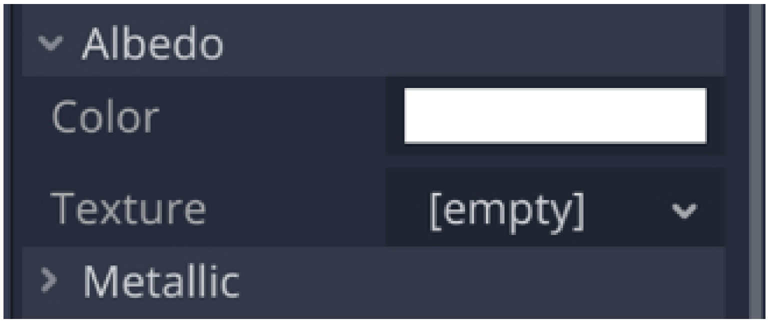


ammo_automatic_gun

- Using the **Scene** view, move the object **ammo_auto** apart from the other ammo pack.

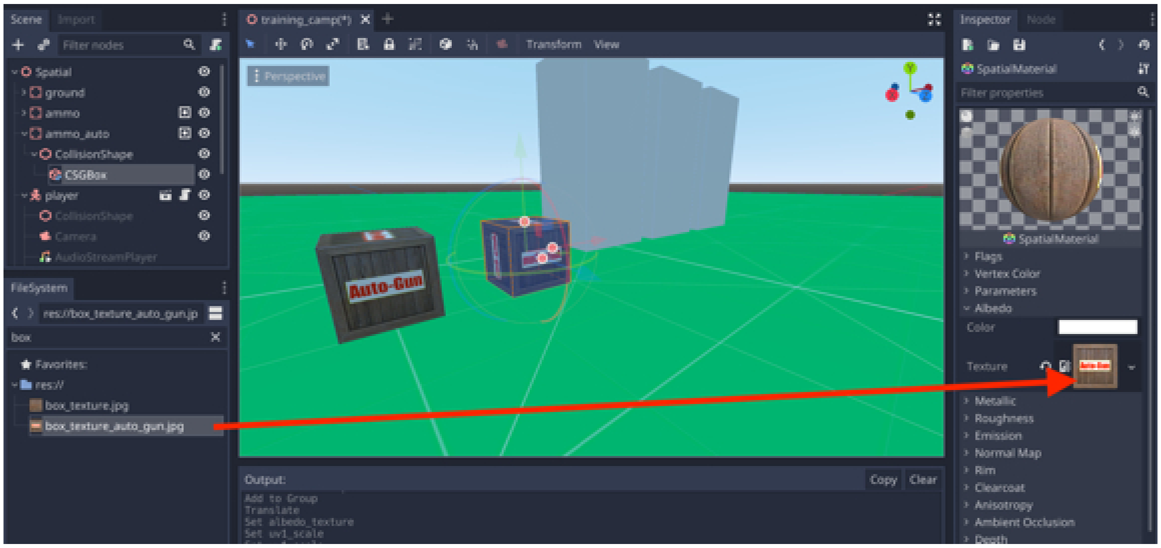
Finally, so that this pack is easily recognizable, we could apply a different texture to it:

- Please select the node **CSGBox** that is a child of the node **ammo_auto.**
- Using the **Inspector**, click on drop-down arrow to the right of the label "**Material**" and select the option "**New Spatial Material**".
- Click on the white sphere.
- Expand the section called **Albedo**.



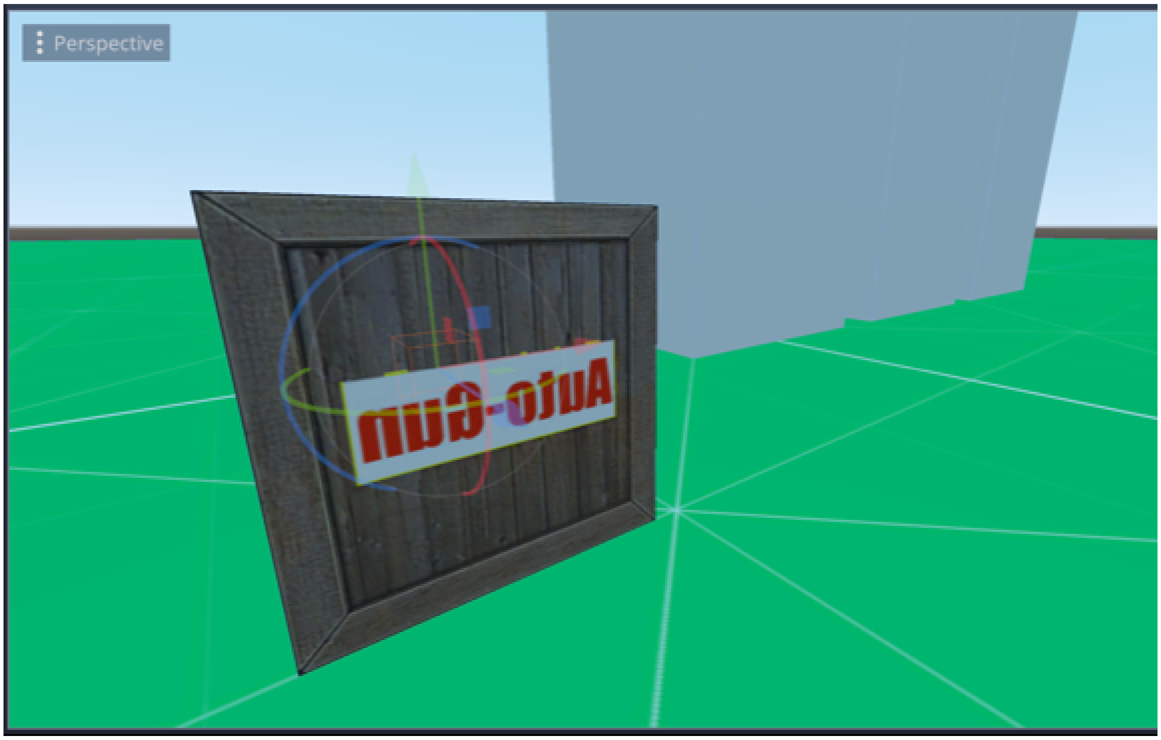
Color
Texture [empty]

- Locate the file **box_texture_auto_gun.jpg** in the **FileSystem** window and drag and drop it to the right of the label **Texture** in the **Inspector** window.

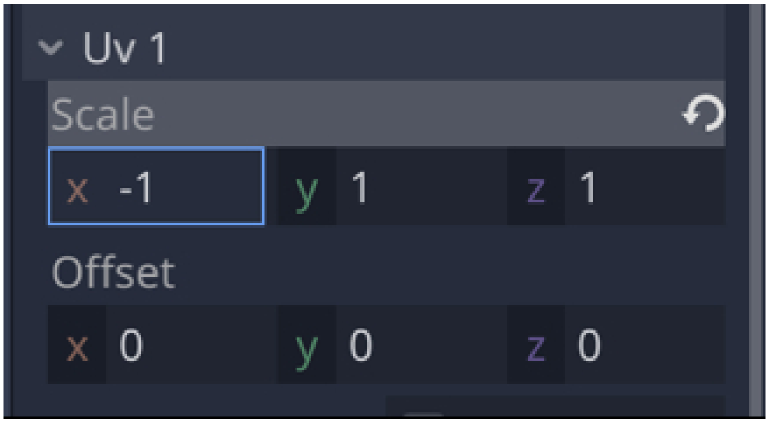


```
Add to Group
Translate
Set albedo_texture
Set uv1_scale
```

- As you apply this texture you may notice that the label "**Auto Gun**" on the box is written backwards.



- To solve this, please scroll down to the section called **UV1** in the **Inspector** and change the **x scale** attribute to **-1.**



Uv 1

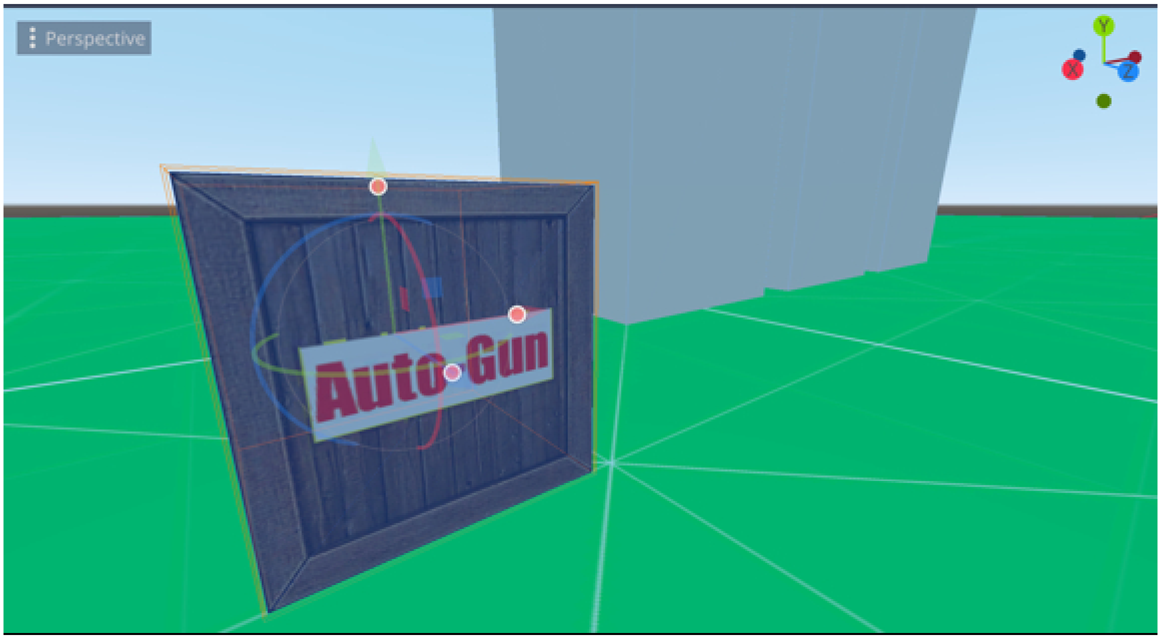
Scale

x -1 y 1 z 1

Offset

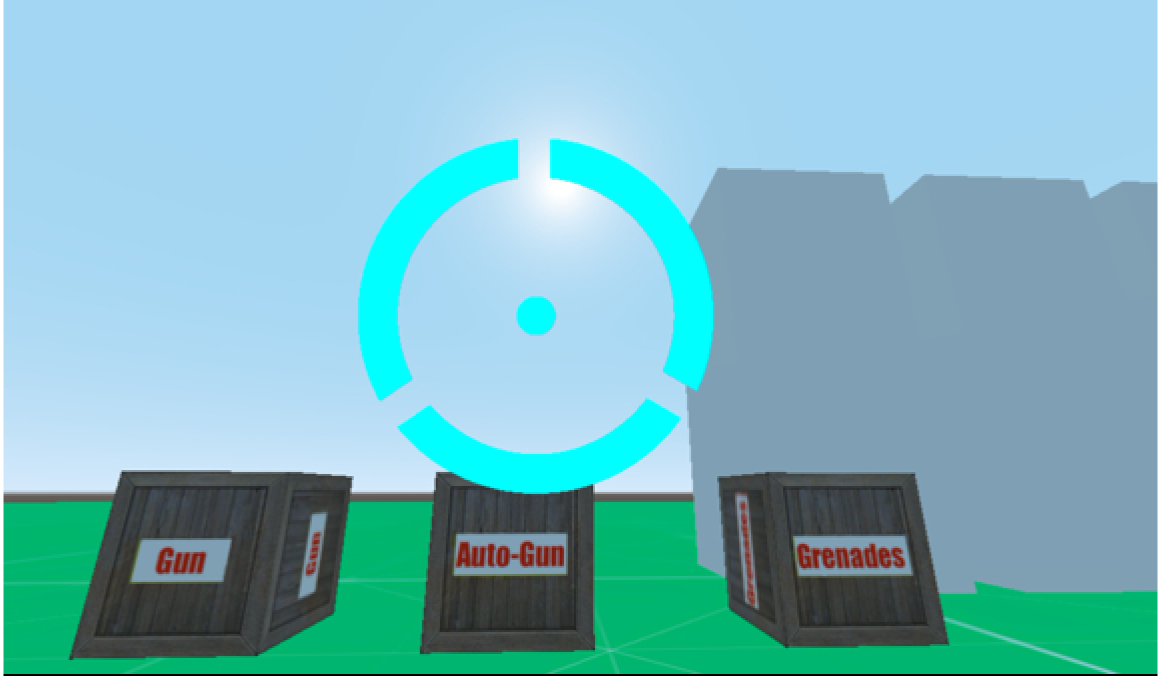
x 0 y 0 z 0

- This should flip the texture and ensure that it is displayed properly as illustrated in the next figure.



Once this is working, you can repeat the previous steps to create a new ammunition pack named **ammo_grenade**, with a group called "**ammo_grenade**" and using the texture **box_texture_grenade.jpg**. You may also modify the texture used for the **CSGSphere** node that is a child of the node **ammo_gun** and use the one called **box_texture_gun** that is located in the **FileSystem**.

Once this is done, you can play the scene to see the three ammo packs.



The last interesting detail we could add here, is to create prefabs from these packs, so that they can be instantiated or modified later.

- Please, right-click on the node **ammo_gun**, select the option **Save Branch As Scene**, and save this node as **ammo_gun.tscn**.
- This will create a scene called **ammo_gun.tscn** in your project.
- Repeat these steps for the nodes **ammo_auto** and **ammo_grenade**.

By creating scenes for these ammo packs, we have created templates that can be reused (or instantiated) either from the **Scene Tree** (i.e., by dragging and dropping the scene to the view port), or from the code, by instantiating these nodes while the game is playing. This is interesting, for example when you would like to balance the game difficulty and spawn some ammos when the player is in trouble and needs them.

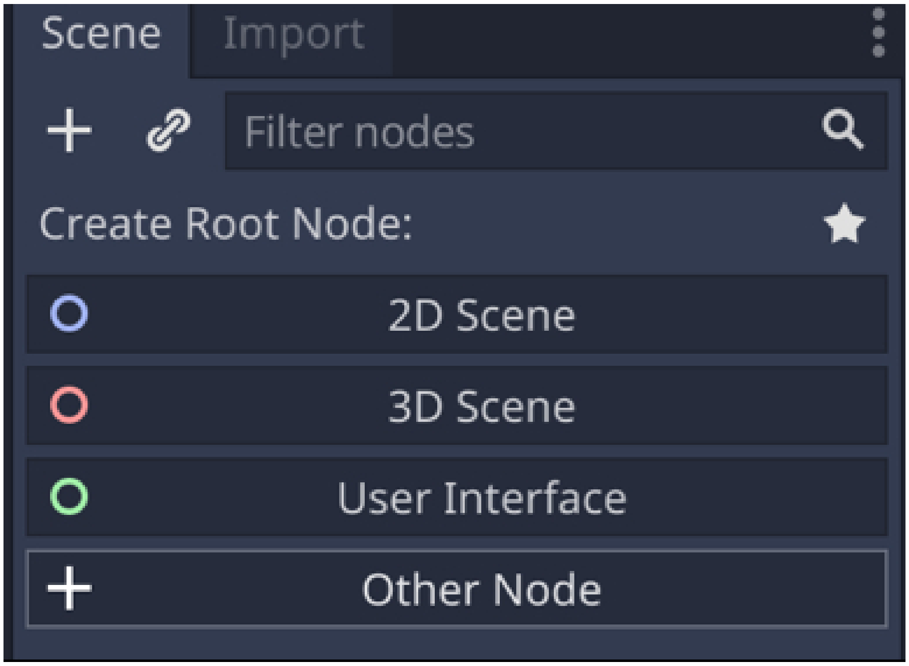
Creating a grenade launcher

At this stage our weapon management system works well; however, we just need to add the ability to throw grenades (and to also pick-up corresponding ammos). For this purpose, we will use rigid body physics, as we have done in the last chapter, to propel the grenade and also apply damage, where applicable. For this purpose, we will:

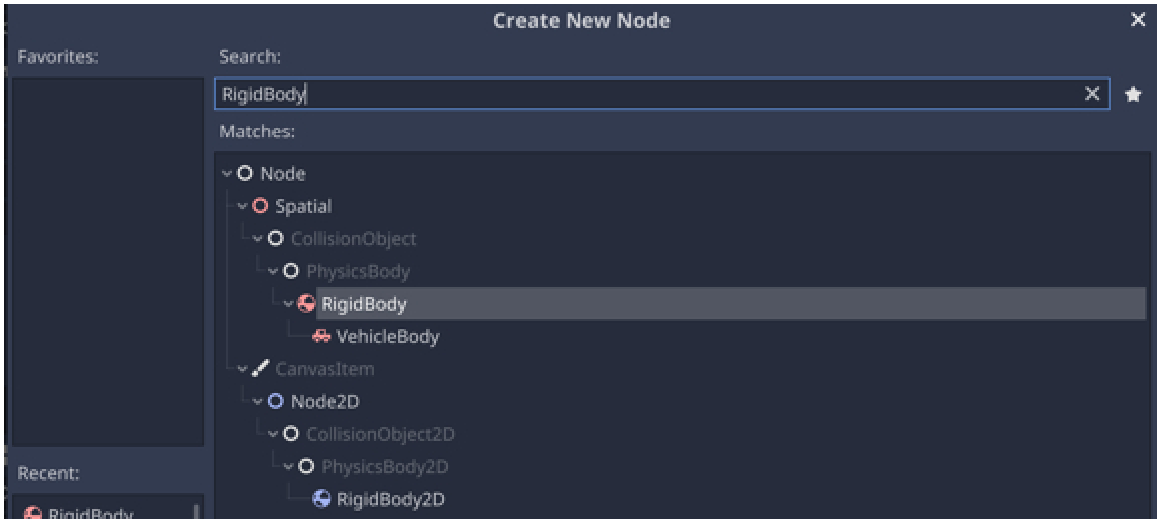
- Create a grenade.
- When the **F** key is pressed and the grenade launcher is selected, we will propel a grenade forward, provided that we have enough ammunition.
- The grenade will explode after a few seconds.
- Upon explosion, all targets within a specific radius of the grenade will be destroyed, including the grenade itself.

So let's get started; first, we will create a grenade

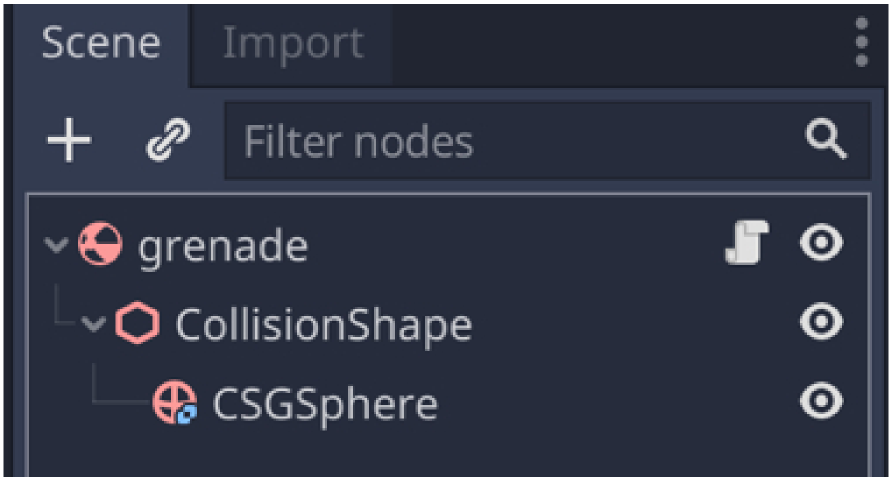
- Please create a new scene (**Scene | New Scene**).
- In the new window, select **Other Node**.



- Select the node type **RigidBody**, and press **Create**.



- This will create a new scene with a node called **RigidBody**.
- Please rename this node **grenade**.
- Add a **CollisionShape** node as a child of this node.
- Using the **Inspector**, set the collision **Shape** to a **CapsuleShape** by clicking on the downward-facing arrow to the right of the field **Shape**, and by selecting the option **New CapsuleShape**.
- Modify the **scale** property of this **CollisionShape** node to **(0.2, 0.3, 0.2)**.
- Finally, add a **CSGSphere** node as a child of the **CollisionShape** node.



- Please save the current scene (**CTRL + S**) as **grenade.tscn**.

Once this is done, we can start to create and attach a script to the node grenade that will serve to detonate the grenade after 2 seconds and to destroy any target within a specific radius.

So let’s get started:

- Please open the scene called **grenade.tscn**.
- Select the node called **grenade**.
- Attach a new scrip to it: right-click on the node **grenade**, select **Attach Script** from the drop-down menu, and call this script **grenade.gd**.
- Once the script is open, add the following code to it (new code in bold):

```
var timer : Timer;
func _ready():
timer = Timer.new()
add_child(timer)
timer.wait_time = 2
timer.start()
timer.connect("timeout",self,"destroy")
```

In the previous code:

- We create a variable called **timer** of type **Timer** that will be used to detonate the grenade after 2 seconds.
- In the function **_ready**, we initialize the timer, add it as a child of the **grenade** node, set its wait time to 2 seconds, and start the timer.
- Finally, we connect the event **timeout** for the timer to the function destroy that will be included in the same script.

Next, please add the following code to the script:

```
func destroy():
var targets = get_tree().get_nodes_in_group("target")
var size = targets.size()
if (targets != null):
for index in targets:
```

```
var other_node = index;
var distance_to_target = global_transform.origin.distance_to(other_node.global_transform.origin)
if (distance_to_target <5):
other_node.queue_free()
queue_free()
```

In the previous code:

- We create the function **destroy**.
- We then initialize the variable **targets** which will hold all the nodes that belong to the group **target**; so the variable called **targets** is effectively an array of nodes.
- If this variable is not empty, we loop through the array **targets** and, for each node within, we calculate the distance between this node and the grenade; if the distance is less than 1 meter, we then destroy the corresponding node.
- In all cases, we will also destroy the grenade.

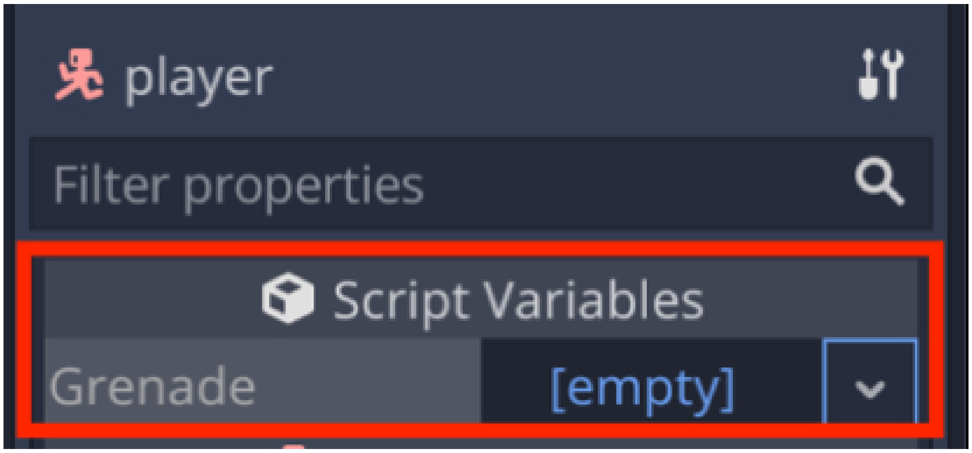
So at this stage, we know that if and when a grenade is instantiated, it will destroy any target that is within a 5-meters radius after 2 seconds. So, now, we just need to be able to throw this grenade from the player.

- Please add the following code at the beginning of the script **Player.gd** just after the first line, as per the next snippet (new code in bold):

```
extends KinematicBody
export (PackedScene) var grenade
```

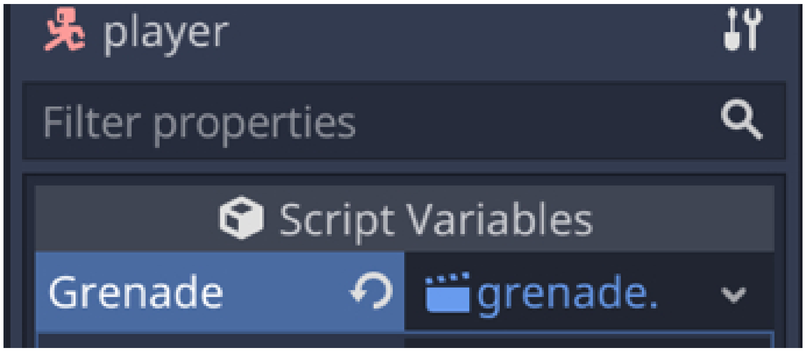
By using the **export** keyword, we create a placeholder in the **Inspector** where we can drag and drop any scene, including the scene **grenade** as we want to use it as a template for the grenades to be thrown.

If you open the scene **training_camp**, switch to the 3D workspace, select the node **player** in the Scene Tree and, and look at the **Inspector**, you should now see a placeholder called **grenade**, as per the next figure.



Grenade [empty]

Please drag and drop the scene called **grenade.tscn** from the **FileSystem** window to the empty slot to the right of the label **grenade** in the **Inspector**.



Grenade grenade.

- Once this is done, please add the following code (new code in bold) to the script **Player.gd**.

```
var condition3 = inventory.has_ammo_for_current()
var condition4 = can_shoot
var condition5 = (inventory.weapon_index==Weapon.TYPE_GRENADE)
```

- Add the following code to the function **_physics_process** just after the previous conditional statement that dealt with the guns, ensuring that the **elif** keyword is at the same level as the one used for the guns:

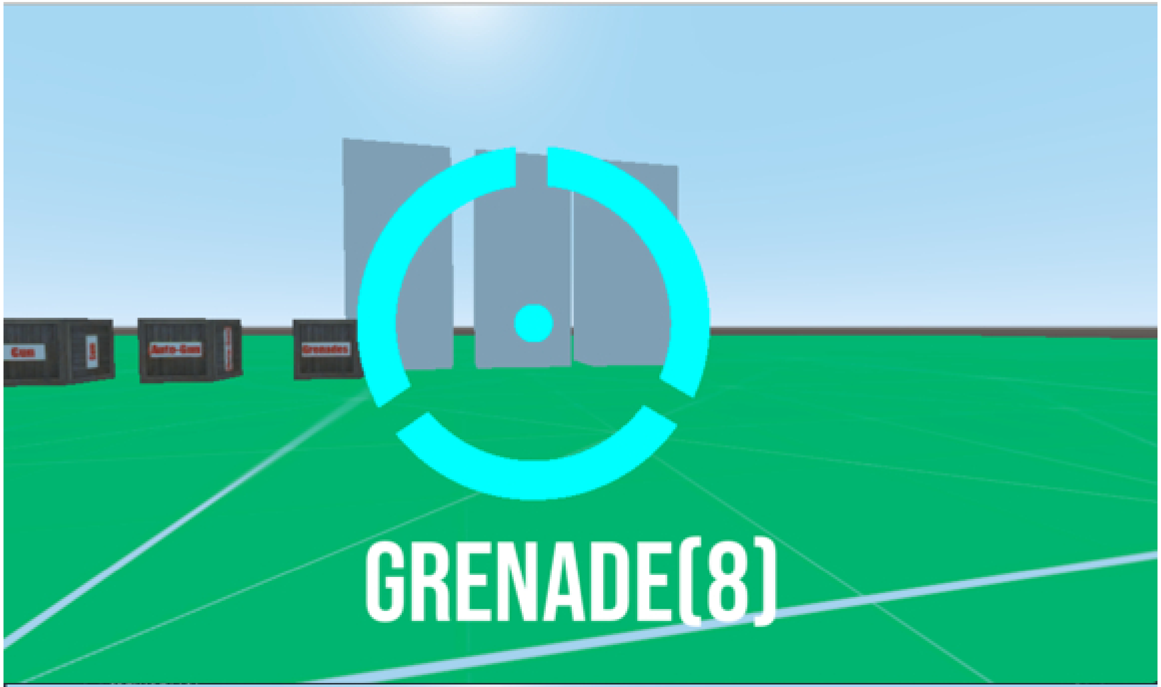
```
elif (condition5 && condition3 && condition4):
inventory.decrease_curr_ammo()
var new_grenade = grenade.instance()
can_shoot=false
```

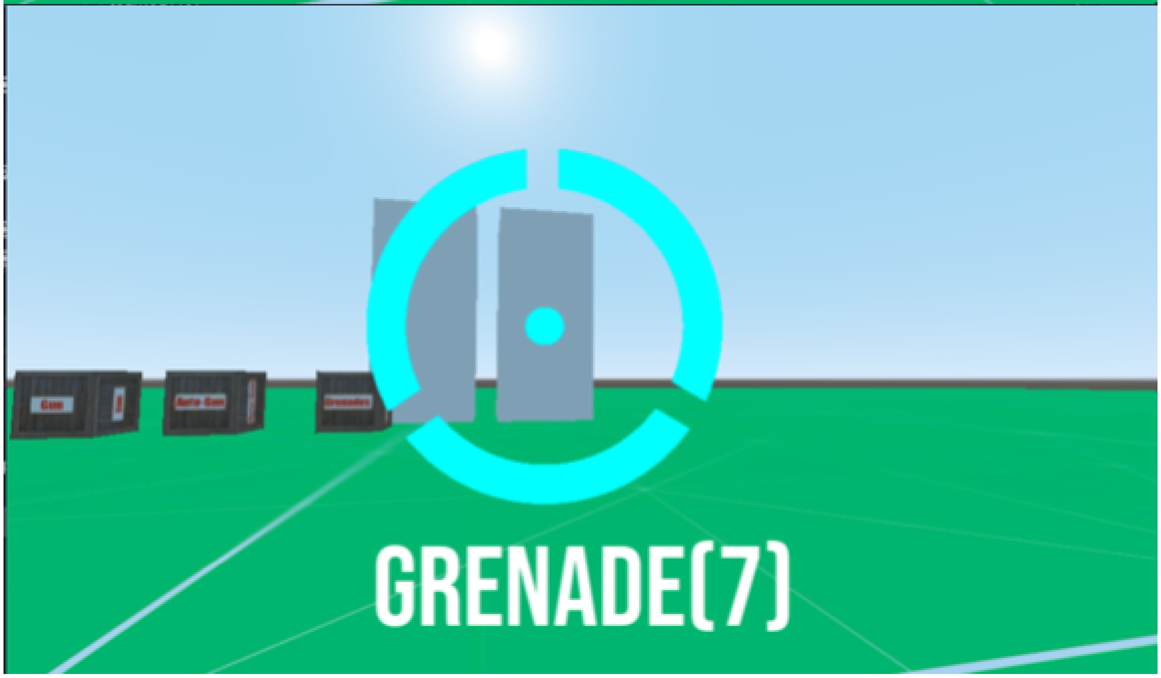
```
reload_timer.wait_time=inventory.get_curr_reload_time()
reload_timer.start()
get_node ("../../Spatial").add_child(new_grenade)
var zPos = global_transform.basis.z
var xPos = global_transform.basis.x
new_grenade.global_transform.origin += get_global_transform().origin
new_grenade.global_transform.origin += transform.basis.xform(Vector3(0,2,1)
new_grenade.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(-20)
```

In the previous code:

- We check whether the grenade launcher is selected, that we have enough grenades, and that we can shoot.
- If these conditions are fulfilled, we decrease our ammunition.
- We set the variable **can_shoot** to false, so that the player cannot use the grenade launcher before the reload time has elapsed.
- We initialize the timer with the reload time related to the grenade launcher and we start the timer.
- We then add the grenade to the 3D world as a child of the node **Spatial**.
- We place the grenade 1 meter ahead of the player, and 2 meters above the ground.
- We then add velocity to the grenade so that it is propelled forward.

You can now save your code, and test the scene. As you play the scene, switch to your grenade launcher and throw some grenades. You should see that the targets within range disappear.





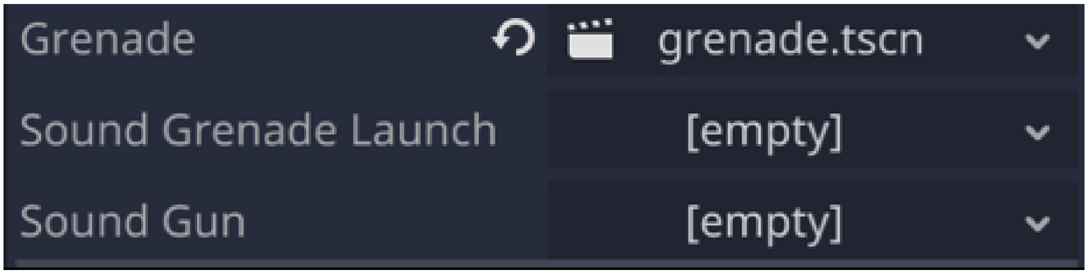
So, the grenades work quite well; however, it would be great to add a sound effect for the grenade, when they are thrown and when they explode, and we will do that just now:

- Please open the script **Player.gd**.
- Add the following lines at the beginning of the script (new code in bold).

```
extends KinematicBody
export (PackedScene) var grenade
export (AudioStream) var sound_grenade_launch:AudioStream
export (AudioStream) var sound_gun :AudioStream
```

In the previous code we create two variables **sound_grenade_launch** and **sound_gun** that will be accessible through the **Inspector.**

- You can save your script, switch to the 3D workspace, select the node **player** in the **Scene Tree**, and look at the **Inspector**, you should now see two new placeholders, as per the next figure.

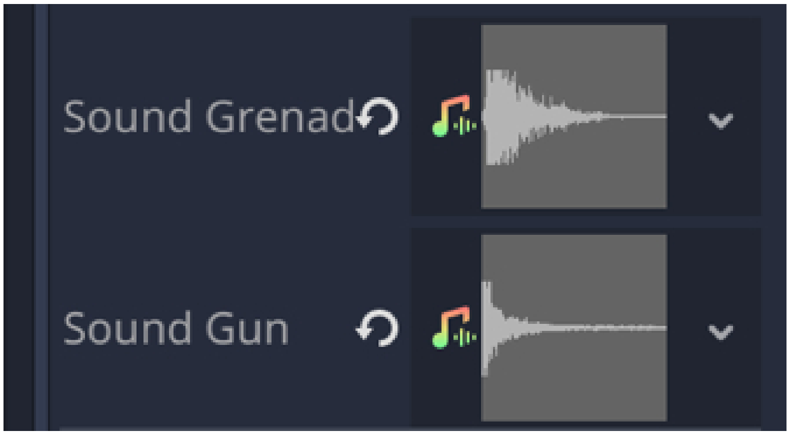


Grenade grenade.tscn

Sound Grenade Launch [empty]

Sound Gun [empty]

- Please locate the file **grenade_launch.wav** in the **FileSystem** window, and drag and drop it to the empty field to the right of the label **Sound Grenade**; do the same for the file **gunshot** for the field to the right of the label **Sound Gun**, as per the next figure.



Sound Grenad

Sound Gun

Now that we have initialized the variables **sound_gun** and **sound_grenade_launch**, let’s go back to the script **Player.gd** to ensure that these sounds are played whenever we use the gun or the grenade launcher, respectively.

- Please open the script **Player.gd**.
- Modify the code that plays the gunshot sound as follows (new code in bold).

```
if (( condition1|| condition2) && condition3 and condition4):
inventory.decrease_curr_ammo()
can_shoot=false
reload_timer.wait_time=inventory.get_curr_reload_time()
reload_timer.start()
sound_fx.stream = sound_gun
sound_fx.play()
```

In the previous code, we just specify that the sound to be played is the one for the gun.

- Please modify the code created to launch the grenade so that a sound is played (new code in bold).

```
new_grenade.global_transform.origin += get_global_transform().origin
new_grenade.global_transform.origin += transform.basis.xform(Vector3(0,2,1))
new_grenade.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(-20)
sound_fx.stream = sound_grenade_launch
sound_fx.play()
```

Finally, we just need to play an explosion sound whenever the grenade explodes; for this purpose, we will need to modify the script called **grenade**.

Please open the script called **grenade**.

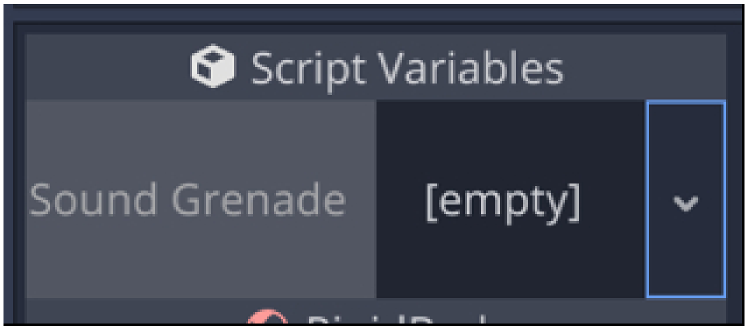
- Add the following code at the beginning of the script (new cold in bold).

```
extends RigidBody
export (AudioStream) var sound_grenade_explode:AudioStream
onready var player_sound_fx = get_node("/root/Spatial/player/sound_fx")
```

- Add the following code to the function **destroy** (new code in bold).

```
if (distance_to_target <5):
other_node.queue_free()
player_sound_fx.stream = sound_grenade_explode
player_sound_fx.play()
queue_free()
```

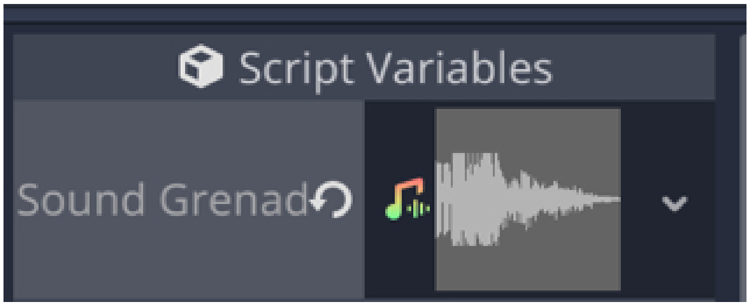
Last but not least, please save your script, switch to the 3D workspace, open the scene called **grenade**, select the node **grenade** in that scene, look at the **Inspector**, and notice the empty field to the right of the label **Sound Grenade**.



Script Variables

Sound Grenade [empty]

- Please drag and drop the file **grenade_explode.wav** from the **FileSystem** window to the empty field.



Script Variables

Sound Grenad

You can now save your script, open the scene **training_camp**, play that scene, and check that a sound effect is played whenever a grenade is launched or explodes.

LEVEL ROUNDUP

In this chapter, we have further improved our skills to learn about how to create a complete weapon management system. We became more comfortable with rays, arrays, and particles. We managed to create scripts to detect objects in the distance, fire a weapon, or propel a projectile in the air. We also optimized our game and code by creating classes, templates, functions, and arrays. So, again, we have made considerable progress since the last chapter. Well done!

Checklist

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

- Create a **ray**.
- Create and instantiate scenes.
- Detect objects ahead with a ray.
- Detect keystrokes.
- Manage and account for ammunition.

Quiz

It’s time to check your knowledge. Please answer the following questions or specify whether a statement is TRUE or FALSE.

1. It is possible to create a template for (or a scene based on) a node by right-clicking on that node and selecting “**Save Branch As Scene**”.
2. The following code will empty the text for the node named **user_message** of type **Label**.

```
get_node(../user_message).set_text(“”)
```

1. To be able to instantiate a scene that was saved as **grenade.tscn**, the following code could be used at the beginning of the script, provided that the file **greanade.tscn** has been dragged and dropped in the corresponding empty field in the **Inspector**:

```
export (PackedScene) var grenade
..
var new_grenade = grenade.instance()
```

1. Find one error in the following code.

```
if (Input.was_action_pressed("fire")):
print(“Fire was pressed”)
```

1. Any scene can be duplicated using the shortcut *CTRL + D*.
2. It is possible to define when a timer times out using the attribute **wait_time**.
3. It is possible to change the sound played by an **AudioStreamPlayer** node by accessing its attribute called **stream**.
4. Provided that the current node is colliding with a node that belongs to the group “**target**”, the following code will display the message “**Collided with target**”.

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

```
var collision = get_slide_collision(index)
if (collision.collider.is_in_group("collect")):
print("Collided with target")
```

1. The only way to create a timer in a scene is by adding a **Timer** node.
2. It is possible to create a script that does not inherit from any of Godot's classes.

Answers to the Quiz

1. **TRUE**.
2. **TRUE**.
3. **TRUE**
4. Find one error in the following code.

```
if (Input.was_action_pressed("fire")):#that should read Input.is_action_presed
print("Fire was pressed")
```

1. **TRUE**.
2. **TRUE**.
3. **TRUE**.
4. **TRUE**
5. **FALSE** (Timers can also be created through code).
6. **TRUE**.

Challenge 1

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

- Add more ammos to the scene, based on the templates that you have created earlier (i.e., drag and drop the scenes for the different types of ammunition to the current scene).
- Test the scene and check that you can collect them and increase your ammunition.

Challenge 2

- Use the **message** node to display the message "**You have just collected ammos**" every time you pick up an ammo pack.
- Using one of the sounds located in the **FileSystem** window, play a sound whenever an ammo pack has been collected.

Chapter 3: Using Finite State Machines

In this section, we will start to work with Finite State Machines (FSM) to be able to manage NPCs and how they behave depending on the environment. We will also get to design some basic, intermediate and advanced artificial intelligence, and animate 3D characters through scripting.

After completing this chapter, you will be able to:

- Create and manage a Finite State Machine.
- Associate character animations to different states.
- Use the FSM to implement basic and intermediate types of artificial intelligence.
- Simulate vision detection for the NPCs.
- Get the NPCs to behave realistically.

Introduction to finite state machines

So, you have probably heard about state machines in the past, but you may not know exactly what it means. In a nutshell, when you create a game, you will most probably use Non-Player Characters (NPCs). These characters will probably have some levels of artificial intelligence.

When applying these different levels of intelligence, we usually want to mimic how people would behave in real life. This means that based on specific factors (e.g., low ammos or enemy in sight), the NPC will follow a specific behavior.

This behavior is often broken down into states. That is, we consider that at any time during the game, the NPC is in only one state. So the NPC will be either idle, following the player, shooting at the player, or looking for health packs.

Now, when we consider states, we also need to consider how (and why) the NPC will enter or exit a state. For example, at the start of the game, our NPC could be idle (state = **IDLE**), and then, if the NPC sees the player, it will transition to the **CHASE** state. While following the player, the NPC may lose sight of the player, and then decide to go back to its initial position (state = **Go Back To Initial Position**), and once it has reached its initial position, it will be **IDLE** again.

So what we can see here, is that we have different states, and there are triggers and/or conditions to enter or exit a state.

Now, this is of course just one possible behavior, and we could create several different behaviors to implement different types of NPCs; but in all cases, this behavior will be determined by states, transitions, and conditions that will need to be fulfilled to transition between states.

Getting started with finite-state machines in Godot

In this section, we will become familiar with creating a simple FSM and applying it to an NPC.

- Please save your current scene (i.e., **training_camp**).
- Using the **Scene Tree**, please duplicate the scene **training_camp.tscn** and rename the duplicate **level2.tscn**.
- Open the scene **level2.tscn**.

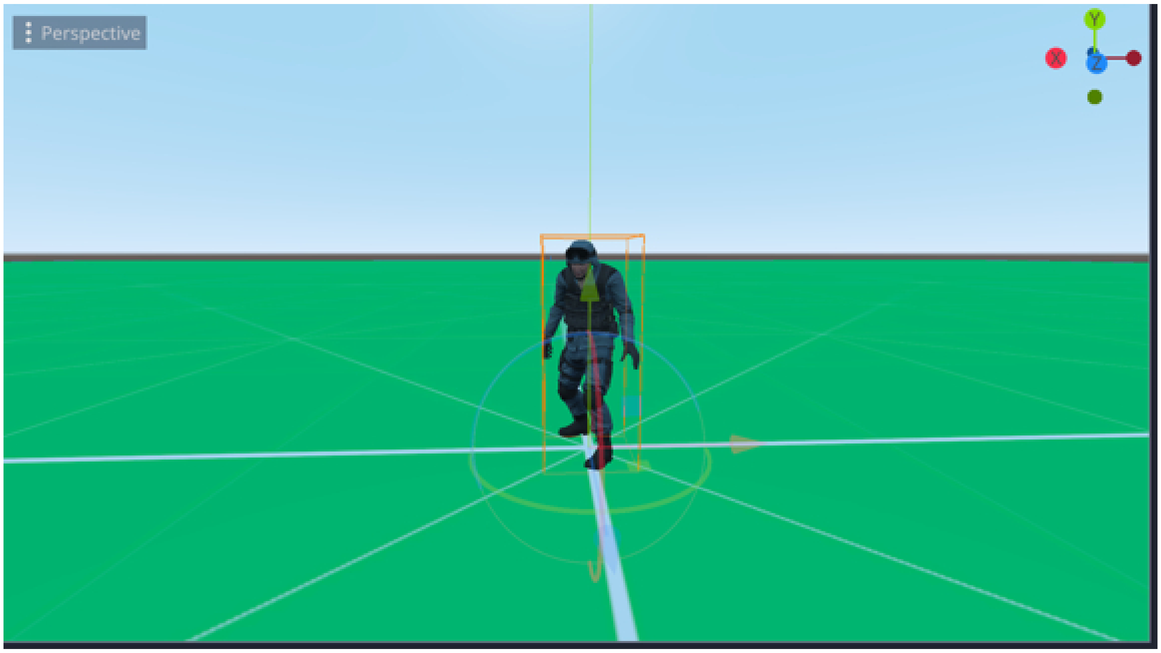
Now that the scene is open, we will start by deleting objects that we no longer need, so please remove the following nodes: the targets (i.e., **target1**, **target2**, **target3**, and **target4**), and the ammo nodes (i.e., **ammo_gun**, **ammo_auto**, **ammo_grenade**).

So the only nodes left now should be the nodes: **Spatial**, **ground**, **player**, **message** and **crosshair**.

When this is done, please play the scene and check that you can still navigate the scene and switch between weapons.

Once this is done, we will now use a file for the animations related to our NPC.

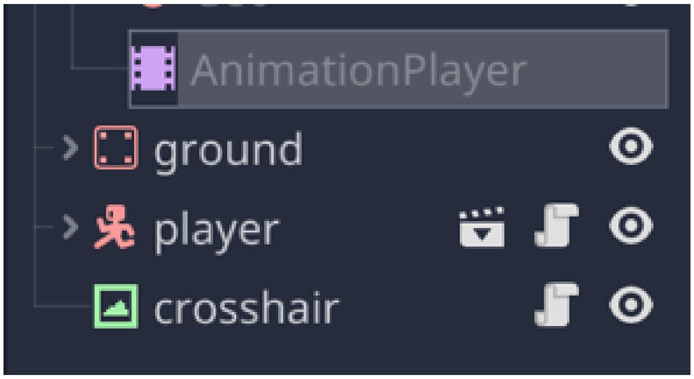
- Please locate the file **soldier6.gltf** in the **FileSystem** window and drag and drop it in the **Scene Tree** atop the node **Spatial**, and this will create a node called **soldier6** that is a child of the node **Spatial**.
- Please rename this node (i.e., the node **soldier6**) **NPC** and change its position to **(0, 1, 0)**.
- If you look at the **ViewPort**, you should see the following 3D character.



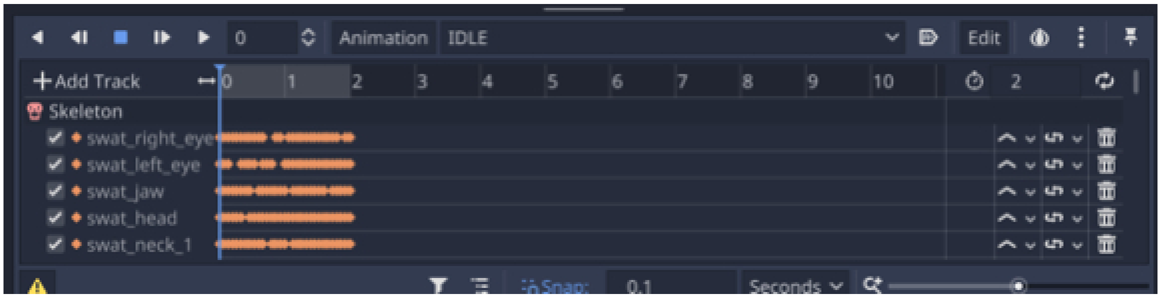
This is the 3D character that we will be using for our NPC; as you can see it is in an **IDLE** pose and it is not animated yet. This being said, as we will discover in the next section, this node includes several animations within that we will be able to play, depending on the state of the NPC (e.g., idle, walking, or running).

So let's discover these animations:

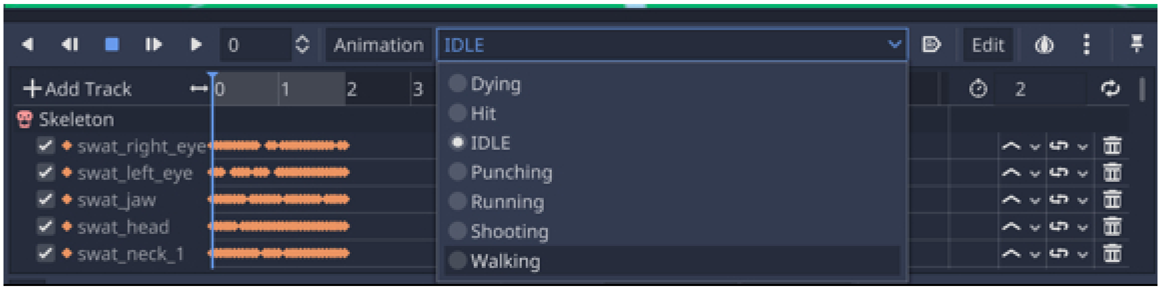
- Please right-click on the node **NPC**, and select the option **Editable Children** from the contextual menu.
- This will expand the node **NPC** and show its children nodes.
- Please scroll down, through the children nodes, then locate and select the node called **AnimationPlayer** as illustrated in the next figure.



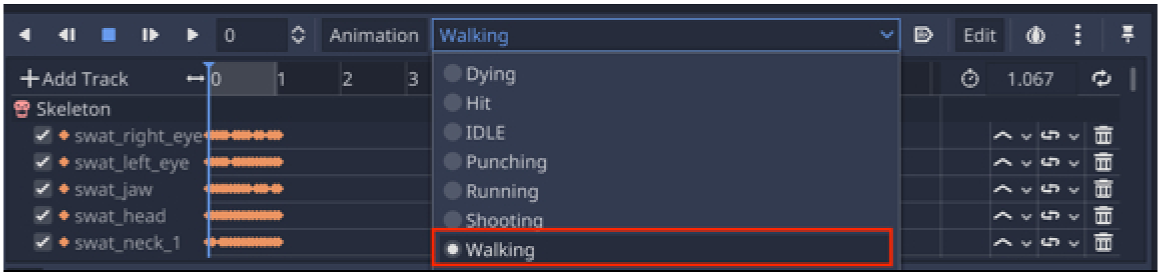
- As you click on the node **AnimationPlayer**, please look at the **Animation** window located at the bottom of the screen.



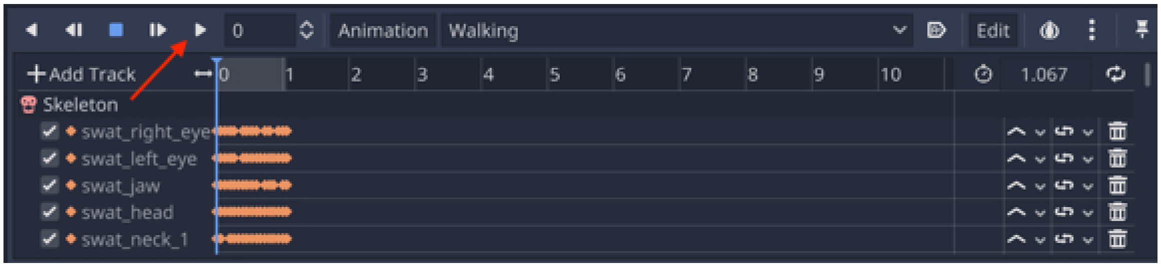
- Click one the downwards facing arrow to the right of the label **Animation**, this will display a list of all the animations within, as per the next figure.



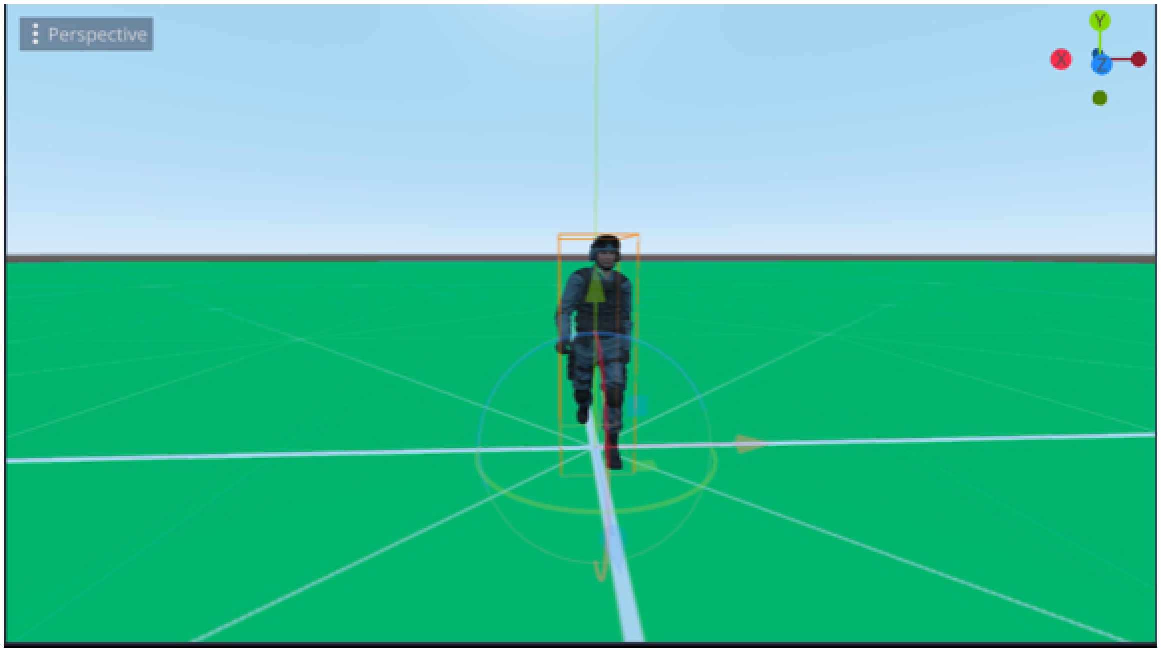
- Please select the animation called **Walking**.



- Then press the **Play** button.



- You should see that the soldier is now walking.



You can select and play any of the other animations if you wish.

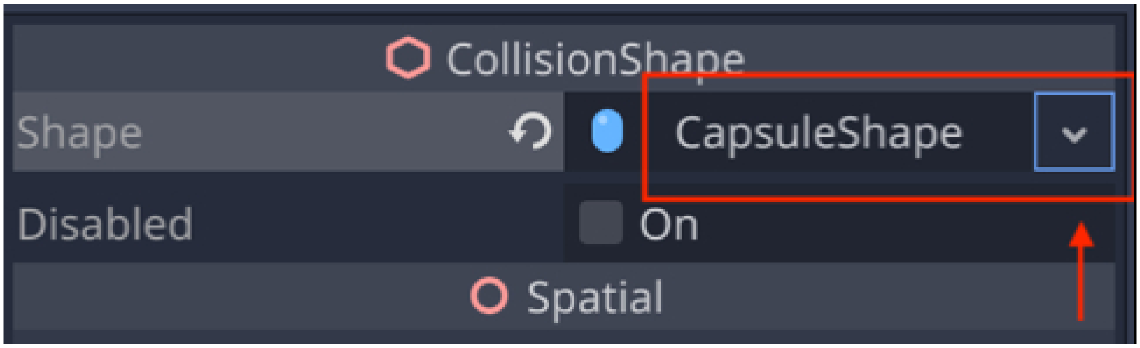
As you can see, the node **AnimationPlayer** includes several animations that we will be able to use including: **IDLE**, **Punching**, **Walking**, **Running**, **Shooting**.

Please note that in order to create a file that includes several animations, as the one used in this chapter, you will need to use a combination of Blender and **Mixamo**; if you would like to use and combine your own animations so that you can re-use them in your game, please check the document included in the resource pack called "**Creating and Combining Your Animations in Once file**"

Now that you have seen the animations that we can use, we will create what is called a Finite-State Machine, through a piece of code that will determine what state our NPC is in, how it can transition between states, and what animation should be played in a given state.

- Please open the scene **level2** (if it is not already open).
- Please create a new node of type **KinematicBody** as a child of the node **Spatial**, and add the node **NPC** (i.e., drag and drop) as a child of that node (**KinematicBody**).
- Change the y coordinate of the **KinematicBody** node to **1** and the **y** coordinate of the **NPC** node to **0**.

- Please also add a node of type **CollisionShape** as a child of the node **KinematicBody** and set its shape to a **CapsuleShape** using the **Inspector**.
- Change the **y** coordinate of this **CollisionShape** node to **1**.
- Click on the downward facing arrow to the right of the label “**CapsuleShape**” and select the option “**Edit**” from the contextual menu.

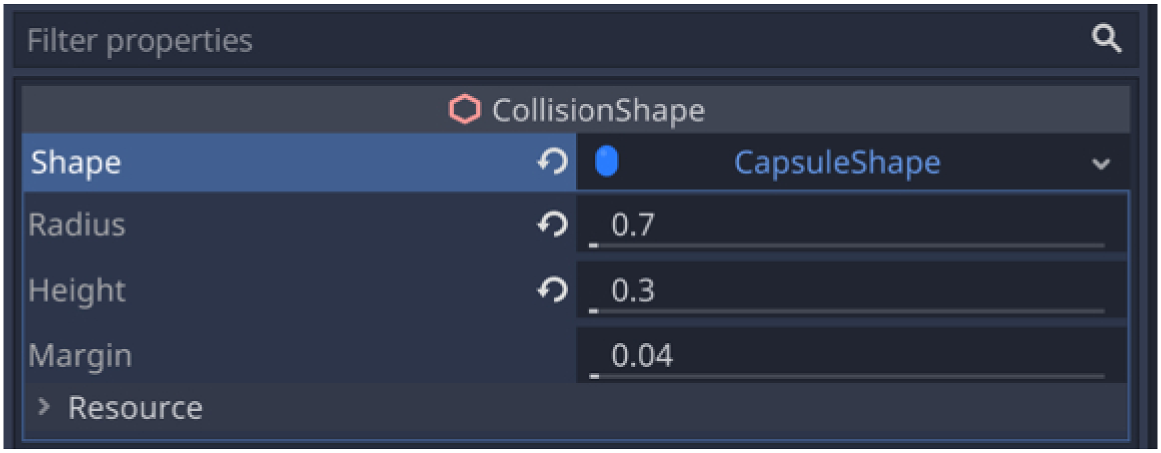


CollisionShape

Shape CapsuleShape

Disabled On

- Change its settings with: **radius** = **0.7**, **height** = **0.3**, and **margin** = **0.04**.



CollisionShape

Shape CapsuleShape

Radius 0.7

Height 0.3

Margin 0.04

- Next, using the **FileSystem** window, please duplicate the script **ManageNPC** and rename the duplicate **ManageNPC2**.
- Next, please attach the script **ManageNPC2** to the node **KinematicBody**.

Now that the nodes are set-up, we will start to define our Finite State Machine. It will consist of a script that will define the different possible states for our NPC, as well as what the NPC should do while in these states, and the corresponding animation that should be played.

- Open the script **ManageNPC2**.
- Change this code...

```
extends StaticBody
```

...to this code...

```
extends KinematicBody
```

- Please add the following code to the script (new code in bold) **ManageNPC2**.

```
extends KinematicBody
var health:int
enum {HIT, IDLE, DYING}
```

In the previous code we define the different states that will be used for our NPC using an **enum**, which is basically a list of constants.

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

```
var current_state = IDLE
```

In the previous code, we define the variable **current_state** that will be used to define the current state for the NPC.

- Next, please add the following code to the function **_ready**.

```
get_node("NPC/AnimationPlayer").get_animation("IDLE").loop = true
get_node("NPC/AnimationPlayer").get_animation("Dying").loop = false
get_node("NPC/AnimationPlayer").get_animation("Hit").loop = false
```

In the previous code we ensure that the animation **IDLE** is looping and that the animations **Dying** and **Hit** will stop after being played once. Note that these animations are part of the node **AnimationPlayer** and are accessed using the function **get_animation**.

- Please add this new function (outside of any other function).

```
func _process(delta):
```

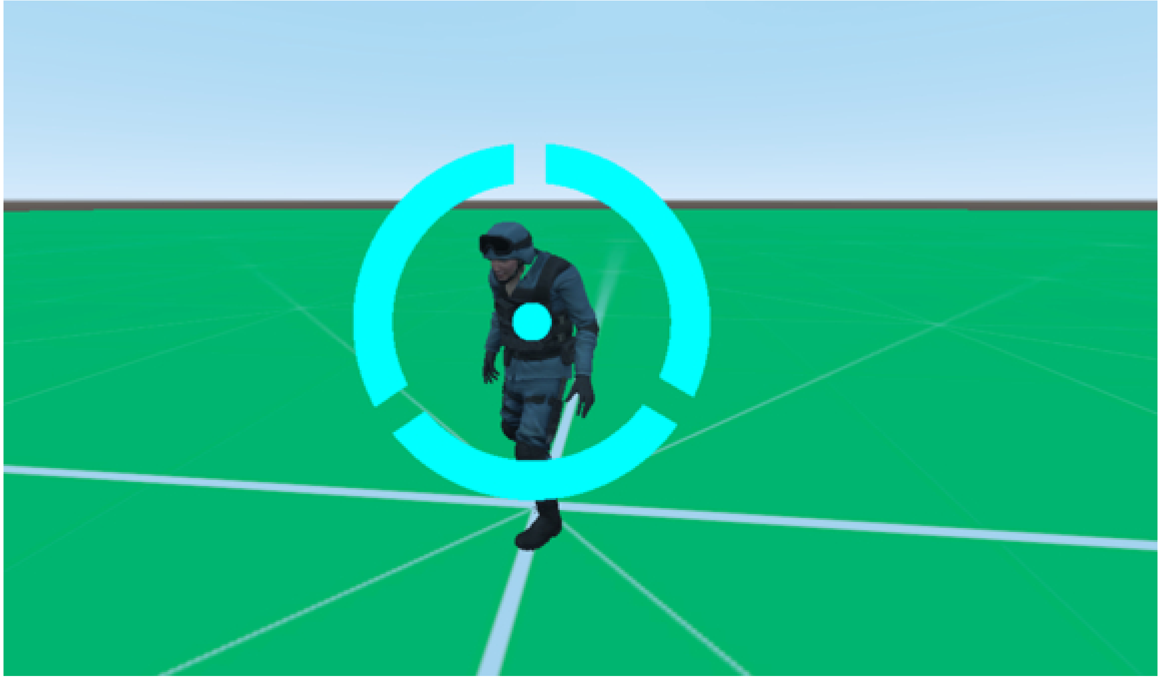
```
match current_state:
IDLE:
print("I am in the state IDLE")
get_node("NPC/AnimationPlayer").play("IDLE")
```

In the previous code:

- We use the function **_process** that is called every frame.
- In this function we use a **match** structure and check the current state.
- If the current state is **IDLE**, then the **IDLE** animation is played.

Finally, open the scene **player.tscn**, select the node called **Camera** in this scene, and change its **y** coordinate to **1**.

You can save your code and play your scene, and as you approach the NPC you should see that it is in an IDLE state and animated.



Next, we will simulate the NPC being hit:

- Please add the following code to the **_process** function (new code in bold) in the **Player.gd** script.

```
match current_state:
```

```
IDLE:
print("I am in the state IDLE")
get_node("NPC/AnimationPlayer").play("IDLE")
HIT:
print("I am in the state HIT")
get_node("NPC/AnimationPlayer").play("Hit")
```

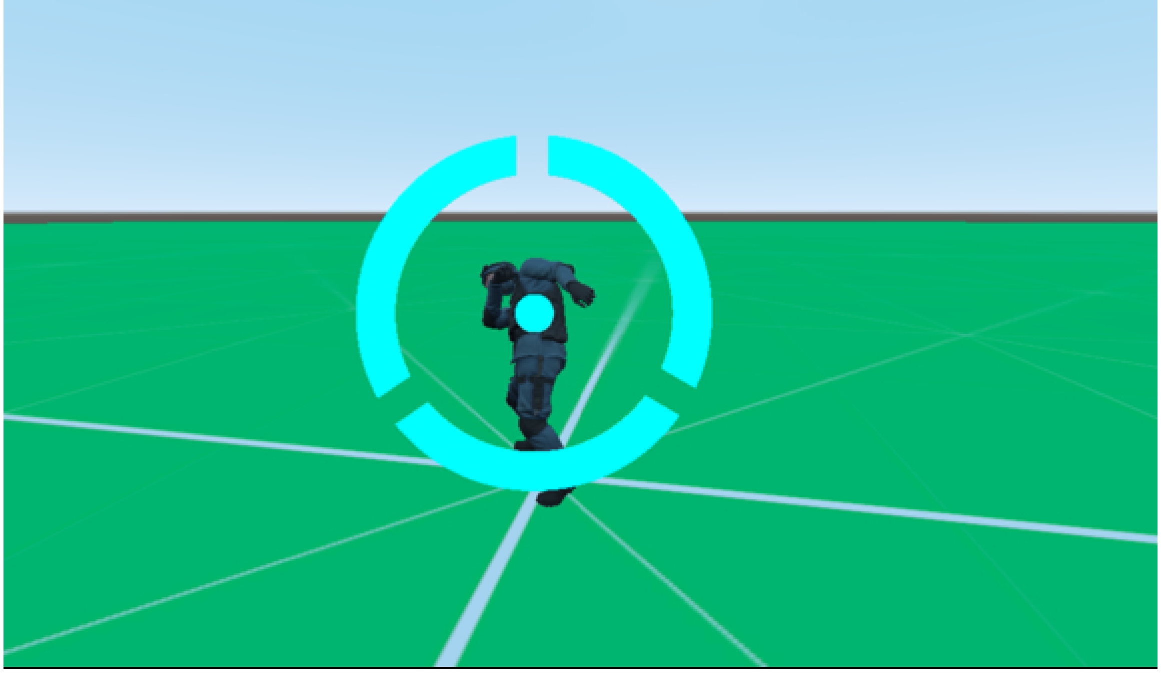
In the previous code: if the current state is **HIT**, we play the animation called **Hit**.

- Please add the following function:

```
func _input(event):
if (Input.is_key_pressed(KEY_P)) :
current_state = HIT
```

In the previous code, we use the built-in function **_input** to detect whether the key **P** has been pressed; if that is the case, we then set the current state to **HIT**.

You can now save your code, open the scene **level2.tscn** and play the scene. As you get closer to the NPC and press the **P** key, the NPC should switch to a **HIT** animation, as per the next figure.



This being said, you may have noticed that this animation is looping; this is because the function **_process** is called every frame, which means that the code within the section for the **HIT** state is executed every frame, hence playing the animation over and over. To solve this issue, we just need to make sure that the animation just plays once and that the NPC switches back to the **IDLE** state afterwards.

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

```
HIT:
print("I am in the state HIT")
get_node("NPC/AnimationPlayer").play("Hit")
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
current_state = IDLE
```

In the previous code we wait until the current animation has finished, and we then set the state to **IDLE**.

You can now save your code and play the scene; as you get closer to the NPC and press the **P** key, the NPC switches to a **HIT** animation and then back to the **IDLE** animation.

Now that this is working, it would be great to interact with this NPC and modify the code so that if the player shoots the NPC, the latter loses energy and plays a **HIT** animation.

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

```
func got_hit():
health -=20
print("Health"+str(health))
if (health <=0): destroy_target()
current_state = HIT
```

In the previous code, if the health is **0** or less, we call the function **destroy_target** and we then set the current state to **HIT** so that the **HIT** animation can be played.

Finally, please select the node **KinematicBody** and set its group to **target**.

You can now save your code, play the scene and shoot the soldier. As you hit the soldier, the **HIT** animation should be played and after hitting the soldiers 5 times, it should disappear.

While this works well, ideally, we would like the dying animation to be played when the soldier dies, so we will need to modify the script.

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

```
HIT:
print("I am in the state HIT")
get_node("NPC/AnimationPlayer").play("Hit")
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
current_state = IDLE
DYING:
print("I am in the state DYING")
get_node("NPC/AnimationPlayer").play("Dying")
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
destroy_target()
```

In the previous code:

- We define the state called **DYING**.
- In that state, we play the animation called **Dying**.
- We also wait until the animation is finished, using the **yield** statement, and then call the function **destroy_target**.

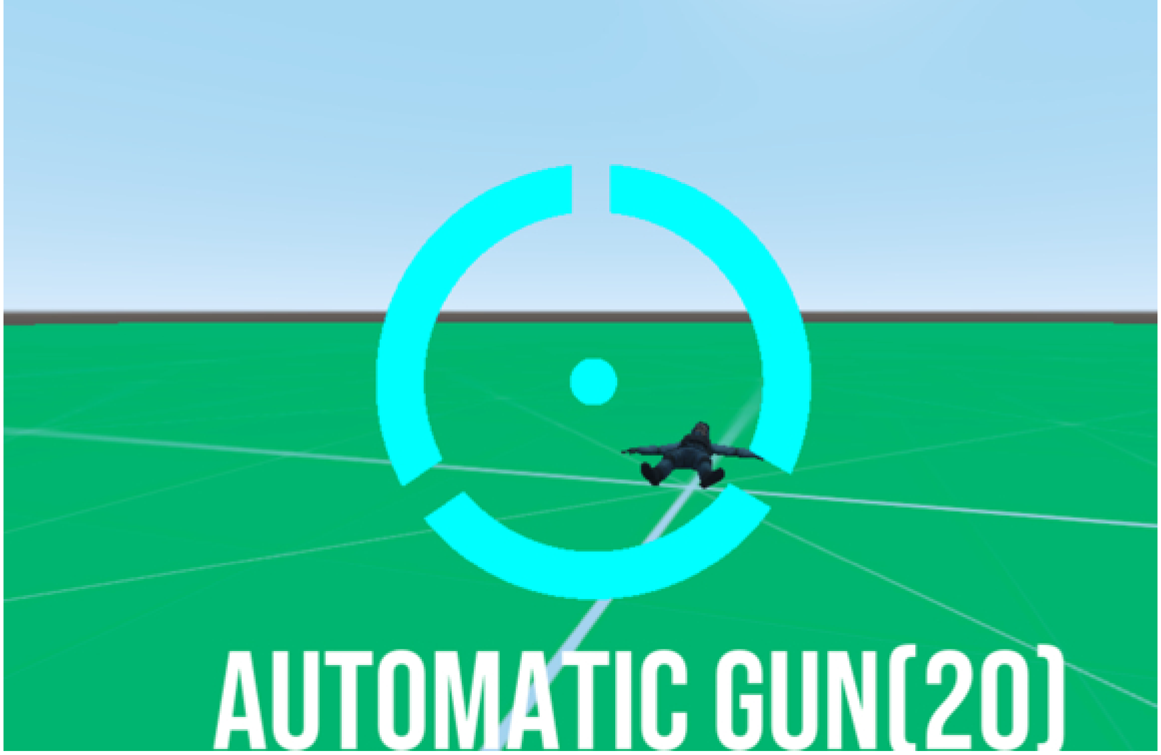
Finally, please modify the function **got_hit** as follows:

```
func got_hit():
health -=20
print("Health"+str(health))
#if (health <=0): destroy_target()
current_state = HIT
```

if (health <=0): current_state = DYIng

In the previous code, if the health of the NPC is **0** the current state becomes **DYING**.

You can now play the scene, after being hit 5 times, the NPC should fall to the ground and disappear.



So at this stage, we have managed to create an animated NPC that also features a Finite State Machine that defines its current state and the corresponding animations to be played. This NPC is initially IDLE, and can be shot by the player; as the NPC is hit, the corresponding animation is played and after 5 consecutive hits, the NPC falls to the ground and disappears.

Making it possible for the npc to attack or fight back

At this stage, while the NPC can be hit by the player, it would only be fair (and also more challenging) to make it possible for this NPC to either attack the player when the latter is in the line of sight or after being attacked. For this purpose, we will do the following:

- Create a state called **Shoot** that will be used when the NPC is firing its weapons at the player.
- In that state, the NPC will have a gun in its hand and look towards the player.
- The NPC will enter that state if the player is in the line of sight or just after being hit/shot by the player.

First, we will create a raycast, a ray that will originate from the NPC and detect if the player is ahead.

- Please modify the script **ManageNPC2** as follows (new code in bold)

```
var health:int
enum {HIT, IDLE, DYING, SHOOT}
```

In the previous code we add a new state called **SHOOT** that will be used when the NPC is shooting.

- Please add the following code just before the **_ready** function:

```
var ray:RayCast
var param_can_see_player = false
```

In the previous code, we declare two variables: the variable **ray** which is a ray-cast and that will be used to detect the player, and the variable **param_can_see_player**, a boolean variable that will let us know whether the player can be seen by the NPC.

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

```
health=100
```

```
get_node("NPC/AnimationPlayer").get_animation("IDLE").loop = true
get_node("NPC/AnimationPlayer").get_animation("Dying").loop = false
get_node("NPC/AnimationPlayer").get_animation("Hit").loop = false
get_node("NPC/AnimationPlayer").get_animation("Shooting").loop = false
```

In the previous code, we make sure that the **Shooting** animation will not loop.

- Please add the following code at the end of 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 and enable the raycast
- We then add the raycast as a child of the current node
- We raise the raycast 1 meter above the ground, to ensure that it will detect the player.
- Finally, we indicate the raycast should point forward.

Next, we need to check when the raycast is colliding, and to then set the variable **param_can_see_player** accordingly, and this will be done in the function **_physics_process.**

- Please add the following function to the script **ManageNPC2**.

```
func _physics_process(delta):
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 use the built-in function **_physics_process**.
- We check whether the ray has collided with an object (or a collider).
- If this is the case, and if this is the node **player**, we set the variable **param_can_see_player** to true
- Otherwise the variable **param_can_see_player** is set to false.

At this stage we can detect whether the player is in front of the NPC, and the variable **param_can_see_player** can be set accordingly. The last thing we need to do is to change the current state based on this variable.

- Please add the following code in the function **_process** (new code in bold) for the state called IDLE.

```
IDLE:
print("I am in the state IDLE")
get_node("NPC/AnimationPlayer").play("IDLE")
if (param_can_see_player): current_state = SHOOT
```

In the previous code, we specify that if the variable **param_can_see_player** is true then the current state should be **SHOOT**.

- Lastly, we will define what should be done in the state called **SHOOT**, so please add the following code to the function **_process** (new code in bold).

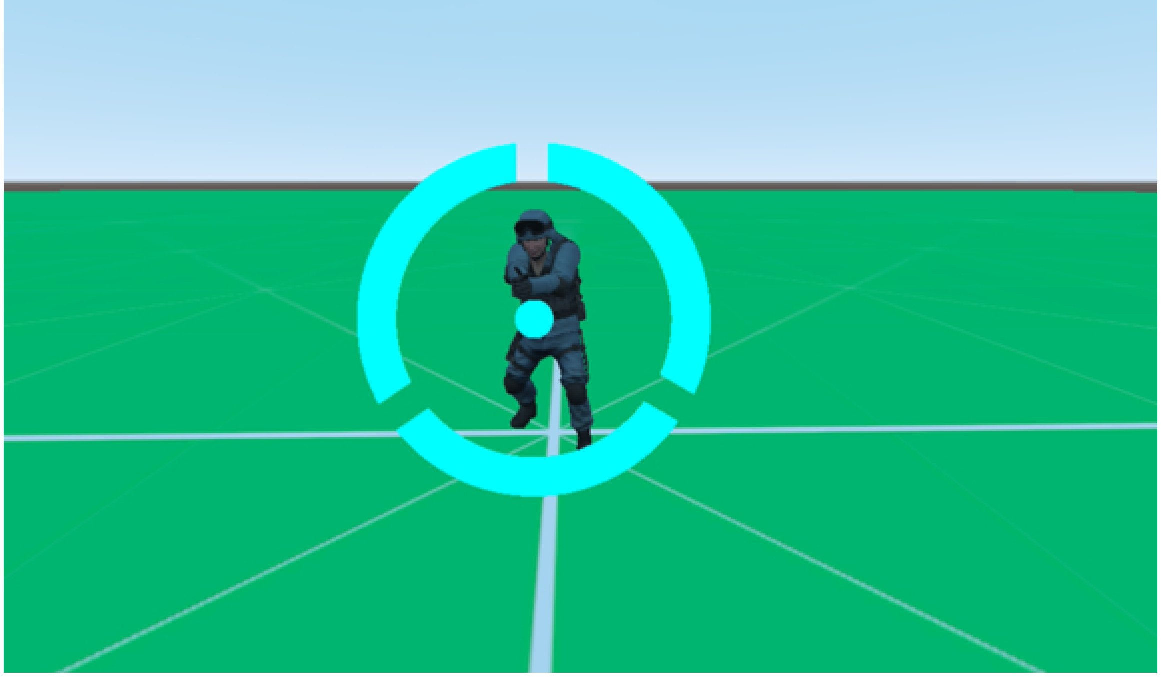
```
HIT:
#print("I am in the state HIT")
get_node("NPC/AnimationPlayer").play("Hit")
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
current_state = IDLE
SHOOT:
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
```

```
get_node("NPC/AnimationPlayer").play("Shooting")
```

In the previous code:

- We define a new match for the state **SHOOT**.
- We ensure that the NPC is facing the player.
- We then play the shooting animation.
- If this is the case, we play the corresponding animation, and we rotate the NPC towards the player.

That's it. Please save your code, and play the scene. The NPC should be IDLE until you walk in front of him which should trigger the NPC to turn towards you and to fire its weapon.



One of the last things we need to implement is the ability for the NPC to shoot back at the player after an attack; to do, so we will need to detect when the player has hit the NPC and change the state of the NPC to **SHOOT**. To do so, please change the following code in the script **ManageNPC2** (new code in bold).

```
HIT:
get_node("NPC/AnimationPlayer").play("Hit")
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
```

```
#current_state = IDLE
current_state = SHOOT
```

In the previous code, we specify that we can see the player and that the current state should now be **SHOOT**.

- Next, please modify the function **_process** in the script **ManageNPC2** (new code in bold):

```
SHOOT:
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
get_node("NPC/AnimationPlayer").play("Shooting")
get_node("../player").got_hit()
```

Finally, we just need to modify the script that is attached to the player to create a variable that represents its health, along with a function **got_hit**.

- Please open the script **Player.gd** and add the following code before the **_ready** function.

```
var health = 100
func got_hit():
health -= 10
print("Player's Health"+str(health))
if (health <= 0):
health = 0
```

In the previous code:

- We create a variable called **health** and we set it to **100**.
- We then create the function **got_hit**, that is called when an NPC hits the player with a bullet.

- In this function we decrease the player's **health** by **10**.
- We also check whether the player's health has reached **0**.

So, based on this code, the NPC once hit will retaliate and shoot towards the player; this being said, because this code, along with the code for the state **SHOOT**, is executed every frame, we will end-up inflicting damage to the player every frame, hence decreasing its health too fast. So we need to ensure that there is a delay between each shot. This can be achieved by doing the following:

- Creating a variable that will determine whether the NPC can inflict damage.
- Enable the NPC to shoot.
- Start the timer, and make it impossible for the NPC to inflict damage until the timer has timed-out.

So please, do the following:

- Add the following code before the function **_ready** in the script **ManageNPC2**.

```
var shoot_timer:Timer
var can_inflict_damage = true
```

- Add this code to the function **_ready** in the script **ManageNPC2**.

```
shoot_timer=Timer.new()
add_child(shoot_timer)
shoot_timer.wait_time = 1
shoot_timer.connect(("timeout"),self,"enable_shooting")
```

In the previous code, we instantiate and initialize the timer called **shoot_timer**, so that it calls the function **enable_shooting** after 1 second has elapsed.

- Add the function **enable_shooting**.

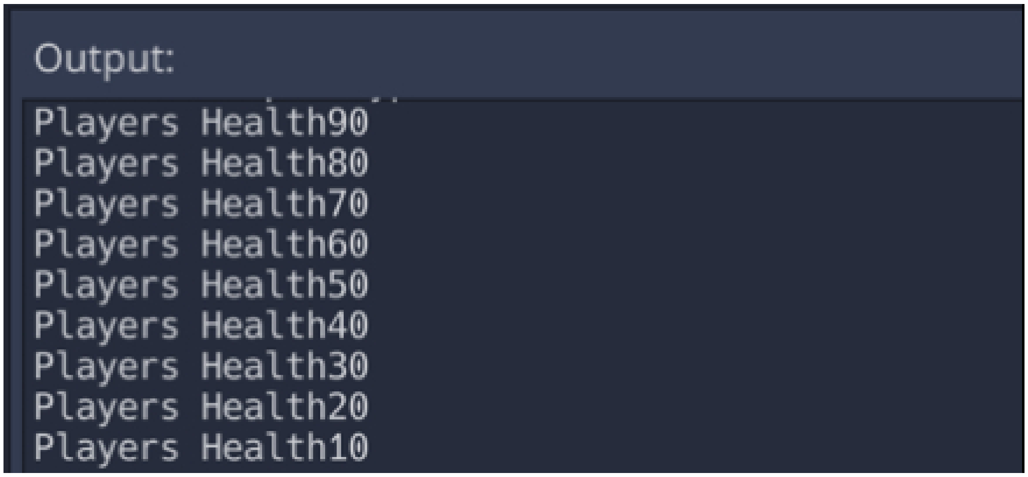
```
func enable_shooting():
can_inflict_damage = true
```

- Modify the function **_process** in the script **ManageNPC2** (new code in bold):

```
shoot:
gun.show()
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
get_node("NPC/AnimationPlayer").play("Shooting")
#get_node("../player").got_hit()
if (can_inflict_damage):
get_node("../player").got_hit()
shoot_timer.start()
can_inflict_damage = false
```

In the previous code, we ensure that damage is applied to the player only every second. This is achieved by starting the timer and keeping the variable **can_inflict_damage** to false for 1 second, until the timer times out.

You can now play your scene and either walk in front of the NPC or shoot in its direction; the NPC should start to shoot back and you should see that the player's health is decreasing and displayed in the **Output** window.



```
Players Health90
Players Health80
Players Health70
Players Health60
Players Health50
Players Health40
Players Health30
Players Health20
Players Health10
```

Adding a weapon to the NPC

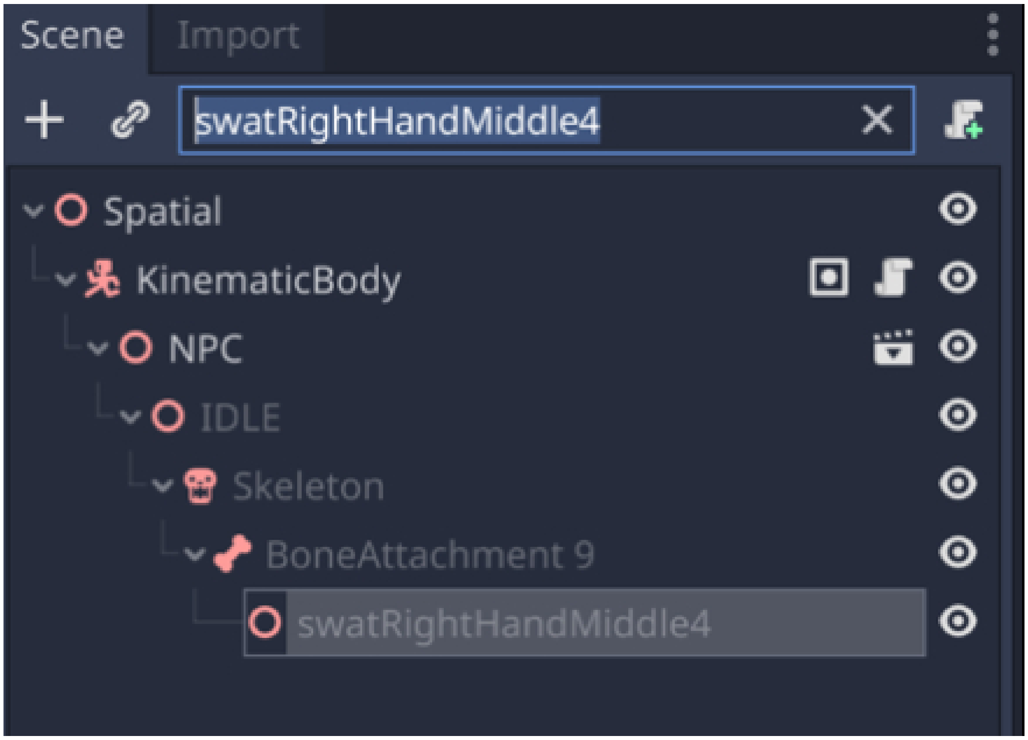
As you check the scene, you may notice that when the NPC is shooting, it has no gun yet in its hand; this is because the animation created did not include the weapon, so we need to add a gun object to this NPC when it is shooting.

So let's add this object:

- Please locate the asset called **Handgun_Game_Blender Gamer Engine.fbx** in the **FileSystem**.

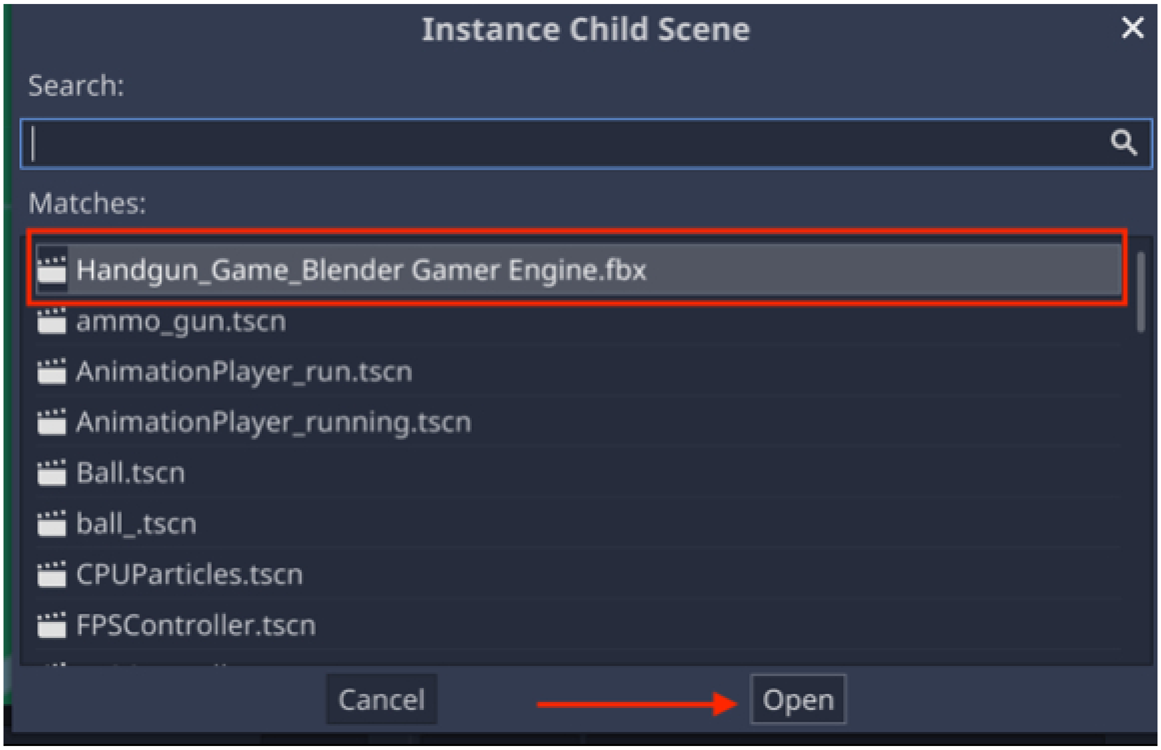
Please note that this model was created by Dennis Haupt and can be downloaded at this url: http://tf3dm.com/3d-model/45-
acp- smith- and- wes- 13999.html 13999.html

- Using the **Scene Tree**, please search the node called **swatRightHandMiddle4**.



- Right-click on it.
- From the contextual menu select **Instance Child Scene**.
- In the new window, select the scene **Handgun_Game_Blender Gamer**

Engine.fbx and click on **Open**.



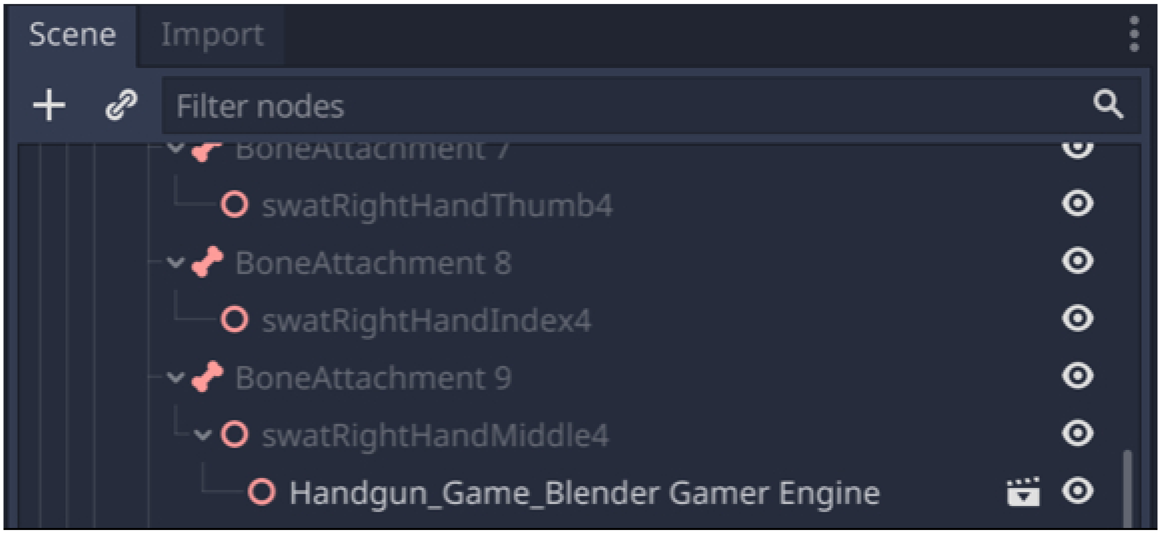
Instance Child Scene

Search:

Matches:

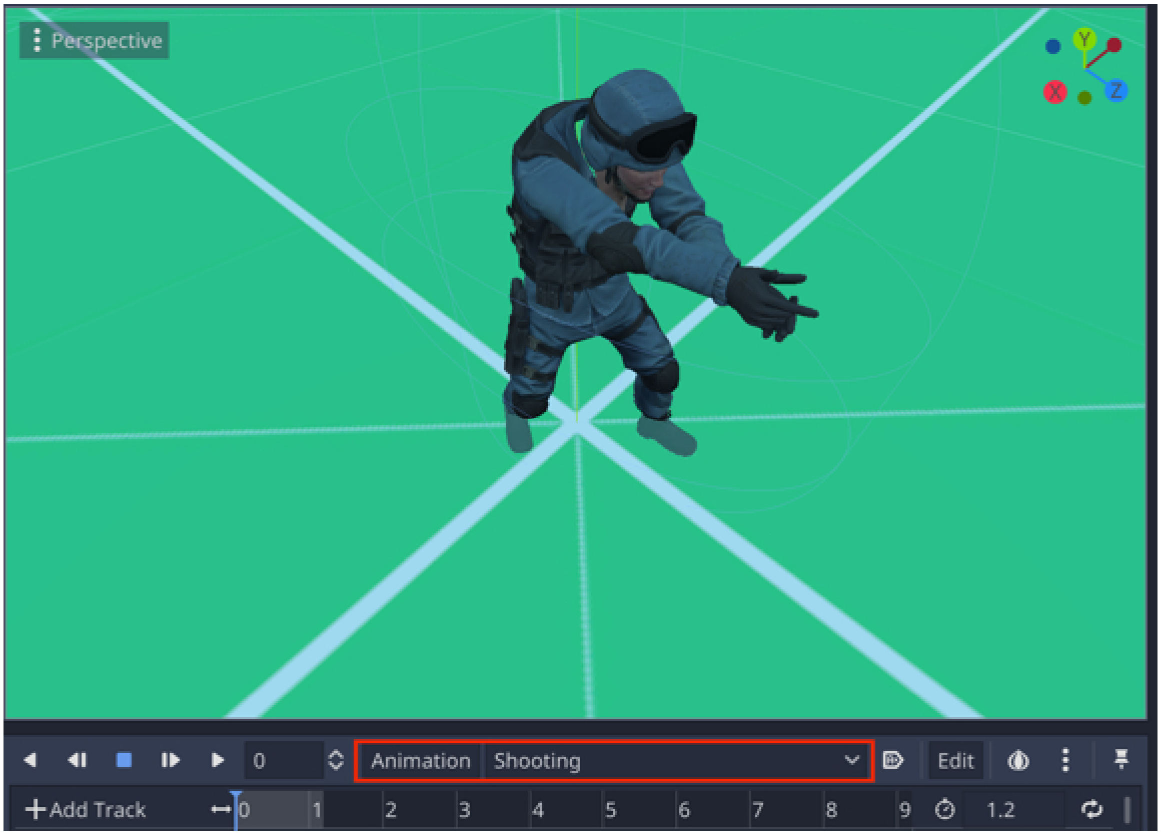
Handgun_Game_Blender Gamer Engine.fbx
ammo_gun.tscn
AnimationPlayer_run.tscn
AnimationPlayer_running.tscn
Ball.tscn
ball_.tscn
CPUParticles.tscn
FPSController.tscn

- This will create a node called **Handgun_Game_Blender Gamer Engine** as a child of the node **NPC**. You will need to clear the search field to be able to display this node in the **SceneTree**.
- You can rename this node **gun**.

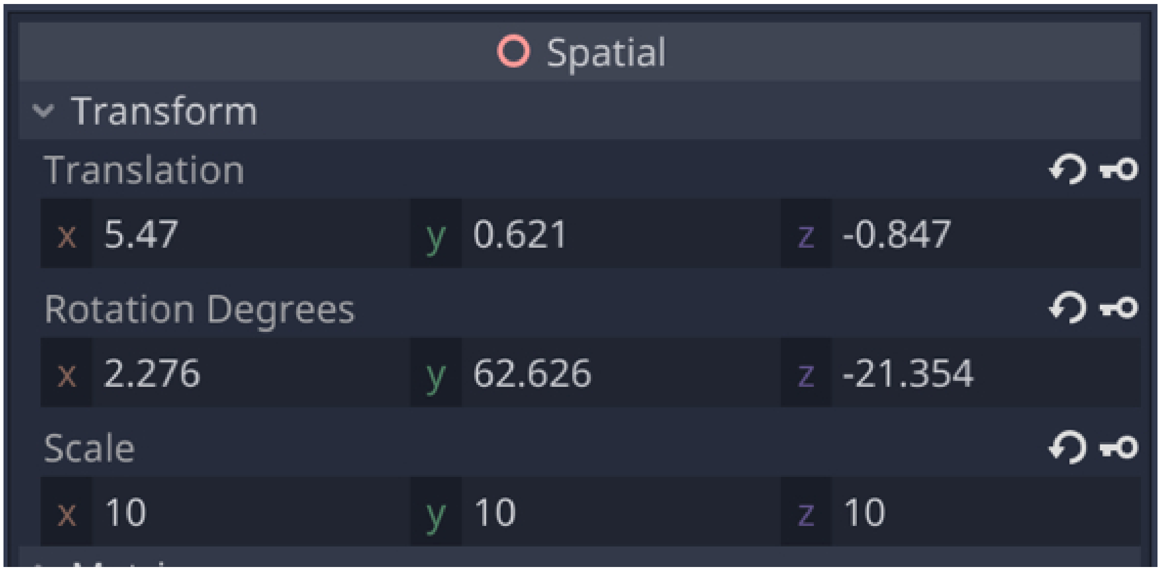


BoneAttachment 7
swatRightHandThumb4
BoneAttachment 8
swatRightHandIndex4
BoneAttachment 9
swatRightHandMiddle4
Handgun_Game_Blender Gamer Engine

- Please click on the node called **AnimationPlayer** that is a child of the node **NPC**.
- In the bottom window, please select the **Shooting** animation, so that the NPC is in the shooting position



- Next, please select the node **gun** and change the following settings in the **Inspector** window: **Translation** = **(5.47, 0.621, -0.847)**, **Rotation** = **(2.276, 62.626, -21.354)** and **Scale** = **(10, 10, 10)**.

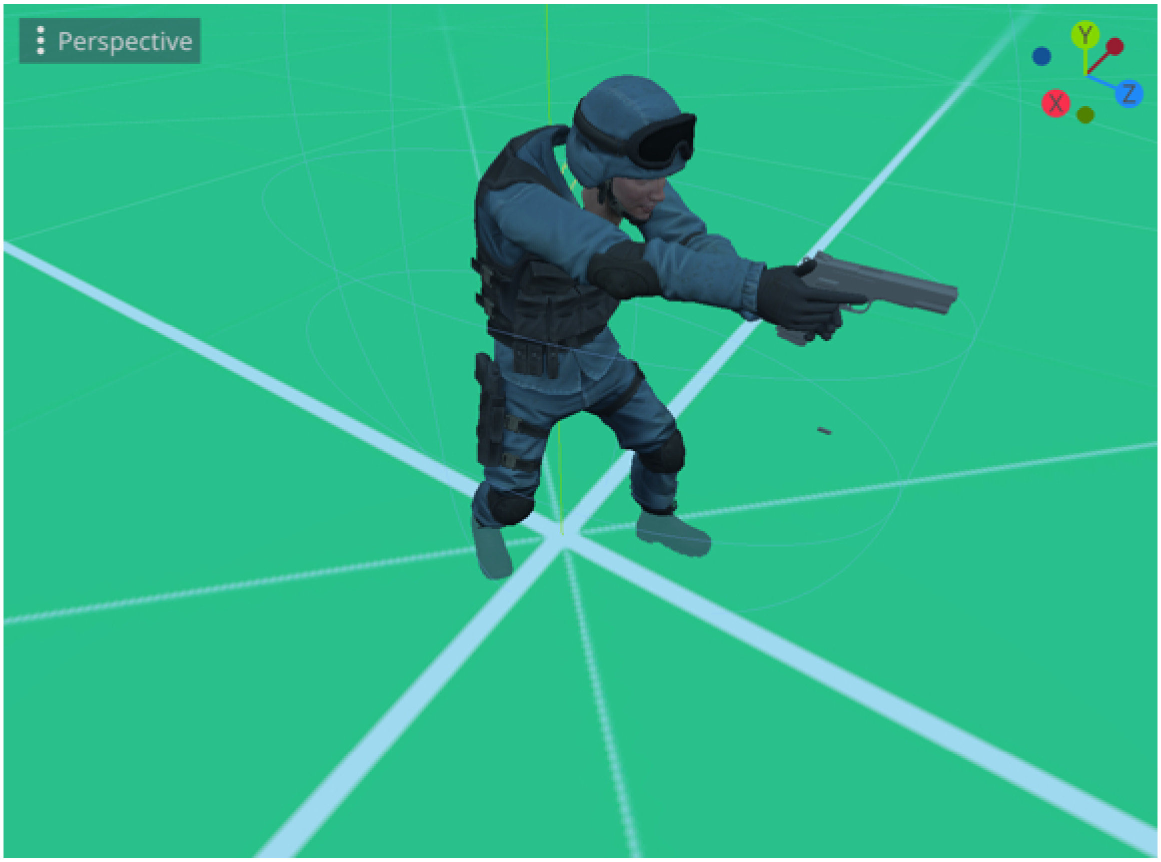


Spatial

Transform

	x	y	z
Translation	5.47	0.621	-0.847
Rotation Degrees	2.276	62.626	-21.354
Scale	10	10	10

- After making these changes, the NPC should look as per the next figure, with the gun appearing in his hands:



You can now test your scene, walk in front of the player, and as the NPC shoots towards you, you can shoot back and see the **HIT** animation being played.

The last things we need to do are to:

- Display the player’s health and the number of lives.
- Reload the level when the health levels are running low.
- Increase the player's health levels after collecting health packs.
- Decrease the number of lives by one after the player’s health has reached 0.

So, first let’s modify the script **Player.gd** accordingly.

- Please open the script **Player.gd**.
- Add this code before the function **got_hit**.

```
var nb_lives = 3
var initial_position
```

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

```
initial_position = global_transform.origin
```

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

```
func got_hit():
health -= 10
print("Player's Health"+str(health))
if (health <= 0):
restart_level()
```

In the previous code we call the function **restart_level** whenever the player’s health has reached 0.

Finally, please add the following function:

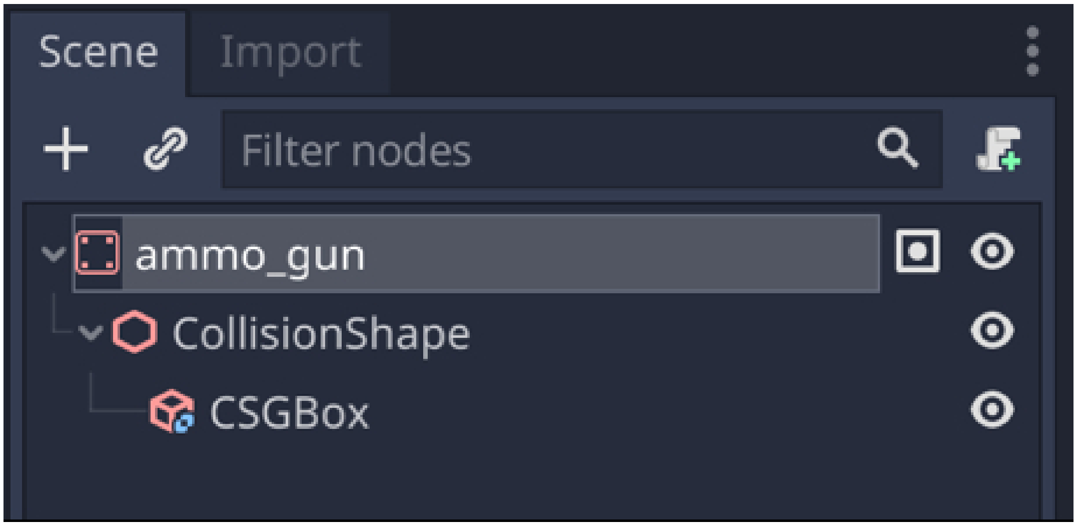
```
func restart_level():
nb_lives -= 1
health = 100
global_transform.origin = initial_position
```

In the previous code:

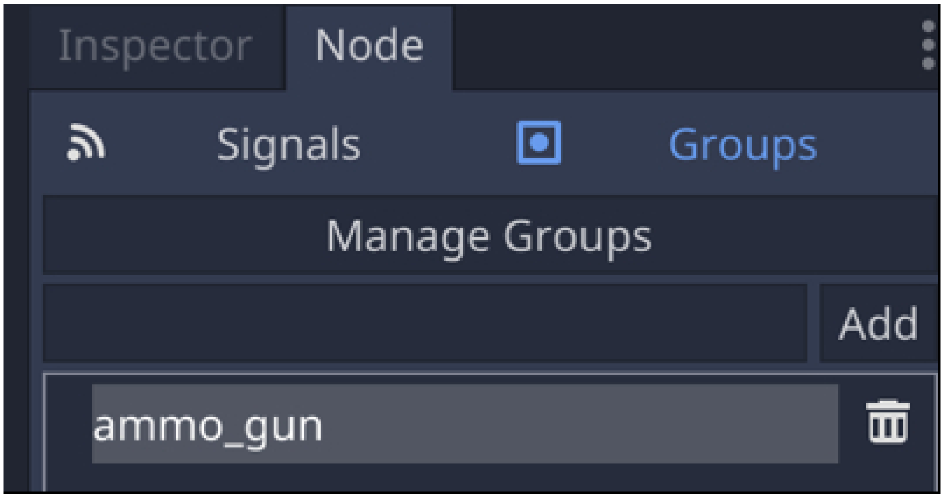
- We define the function **restart_level**.
- In this function, we decrease the number of lives, we set the player's **health** to **100**.
- We also reposition the player at its initial location.

After this, we just need to create and add health packs:

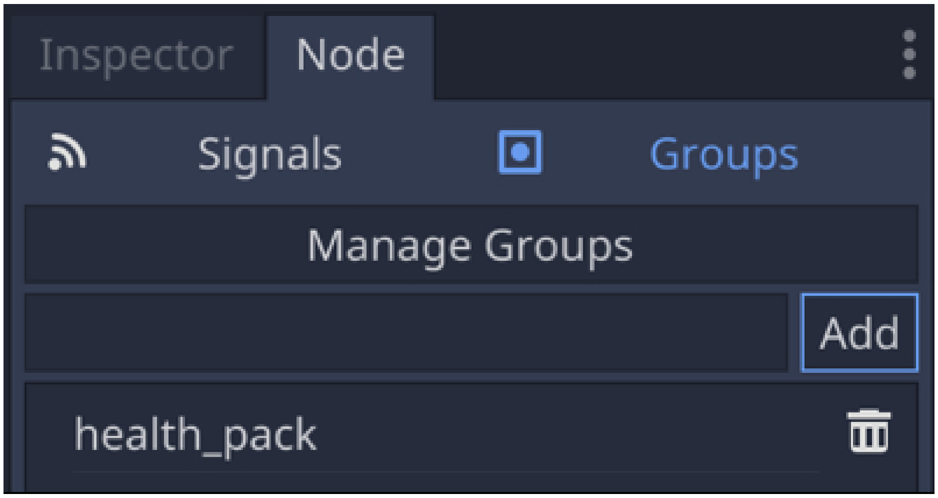
- Please duplicate the scene **ammo_gun**.
- Rename the duplicate scene **health_pack**.
- Open that scene.
- In the new scene, select the node **ammo_gun** and rename it **heath_pack**.



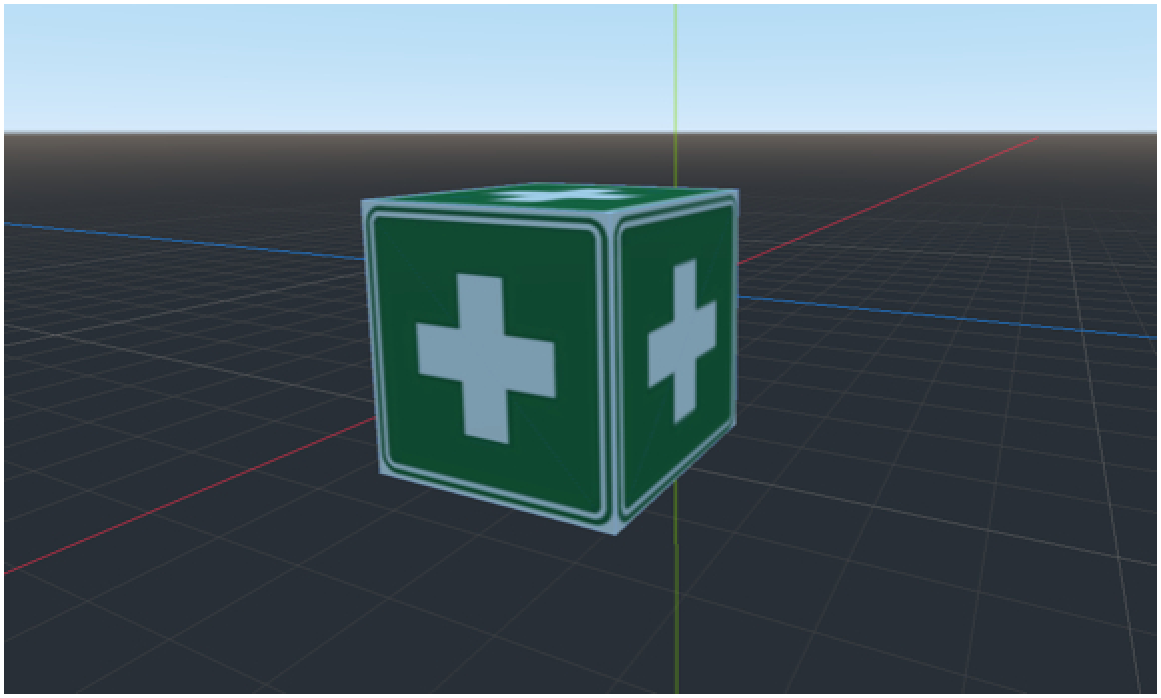
- Using the **Inspector**'s tab called **Node**, delete the current group **ammo_gun**.



- Create a group called **health_pack** and apply it to this node.



- Lastly, modify the material used for the node **CSGBox** by creating a new **Spatial Material** and applying the texture **health_pack** located in the **File System**.



We will now modify the script **Player.gd** so that, upon collision with health packs, health levels are increased:

- Please open the script **Player.gd**.
- Add the following code in the function **_physics_process** (new code in bold).

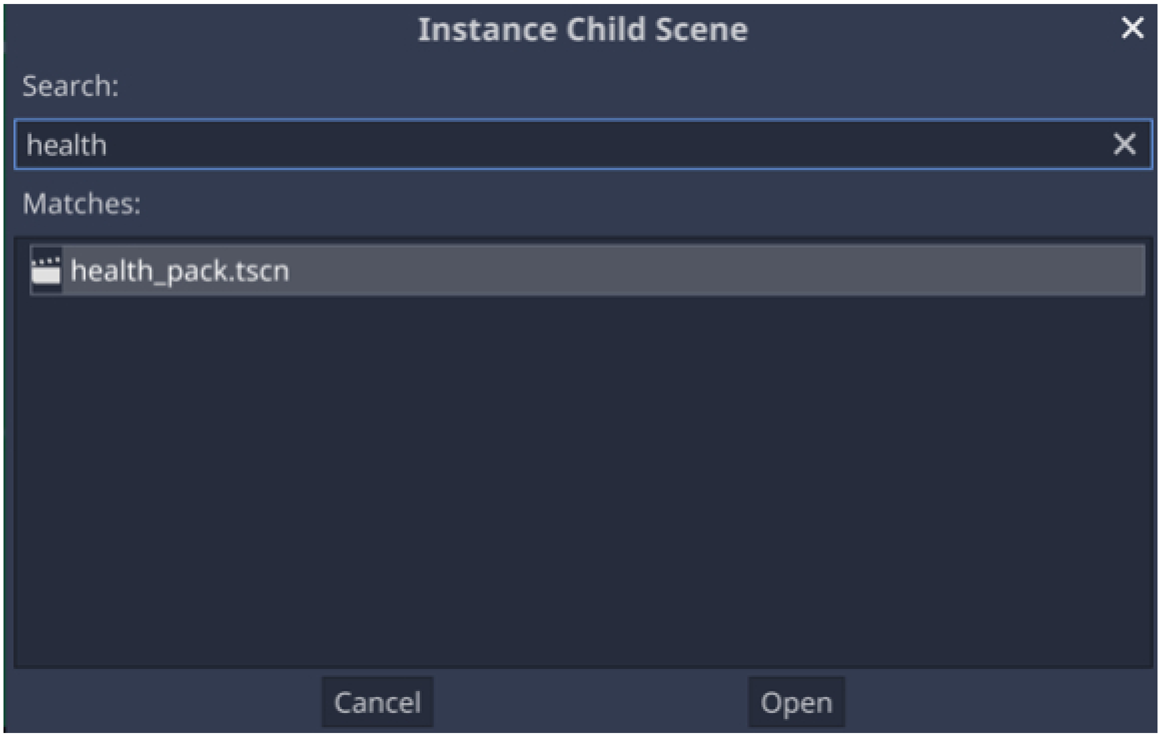
```
elif (collision.collider.is_in_group("ammo_grenade")):
inventory.weapons[Weapon.TYPE_GRENADE].increase_ammo(10)
collision.collider.queue_free()
elif (collision.collider.is_in_group("health_pack")):
health = 100
print("Ammo Collected; health is "+str(health))
collision.collider.queue_free()
```

In the previous code:

- We detect whether we have collided with a health pack.
- We then set the health to **100**.
- We then remove the health pack.

Before we can test the scene, we just need to add a heath pack:

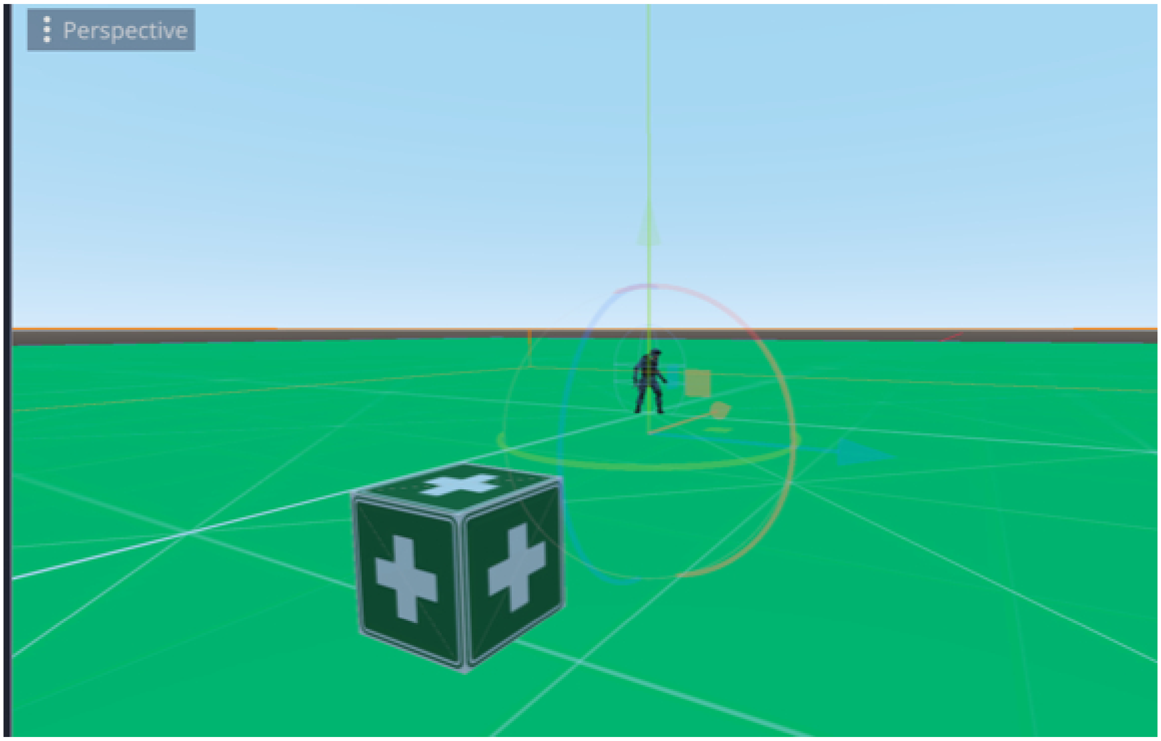
- Please save and close the scene **health_pack**.
- Open the scene **level2.tscn**
- Right-click on the node **Spatial** and select **Instance Child Scene**.
- In the new window, look for the scene **health_pack.tscn**.
- Select this scene.



Search: health

Matches: health_pack.tscn

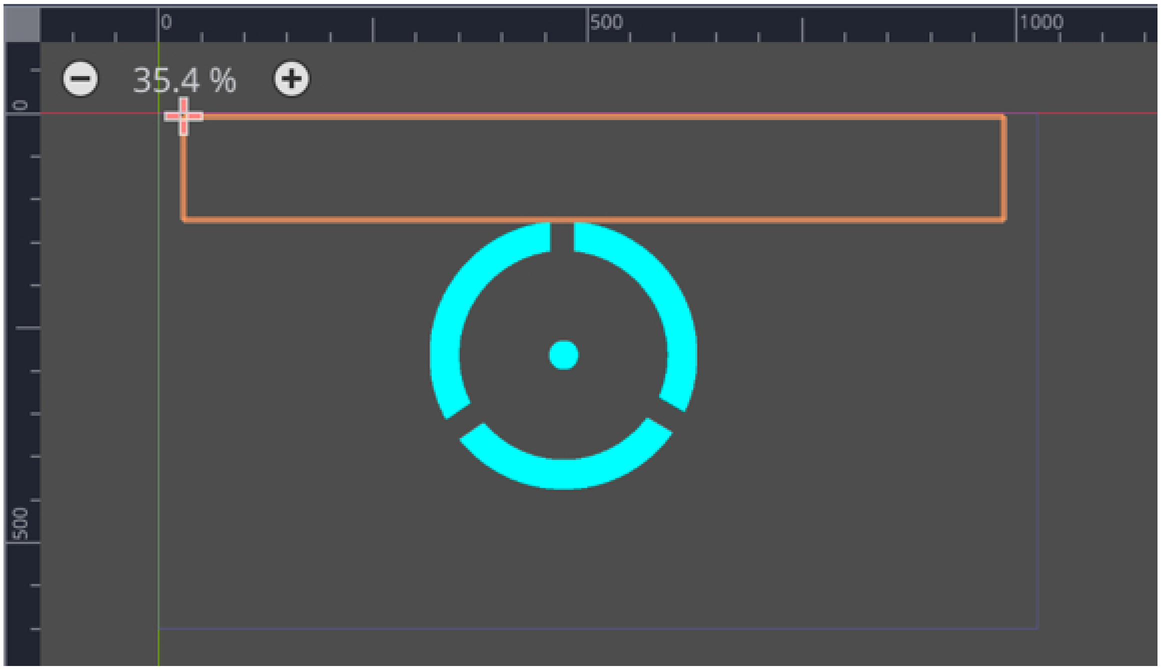
- Press **Open**.
- This will add a node called **health_pack** to the scene.
- Select this node, and resize it (e.g., **.2, .2, .2**) if need be.



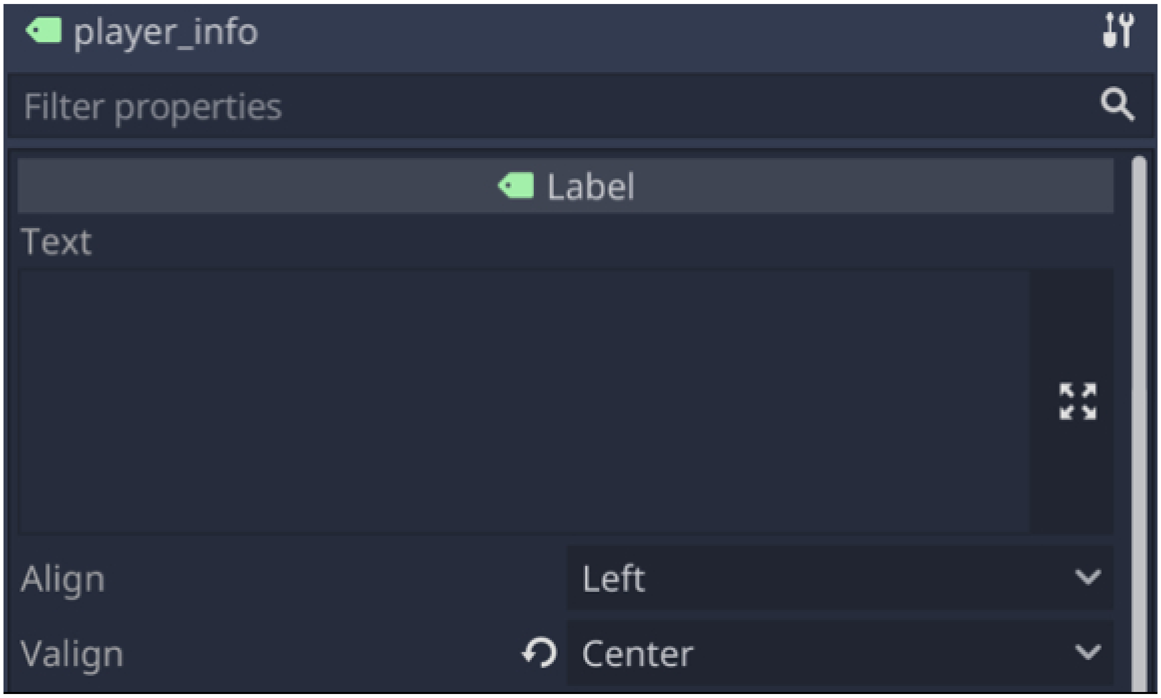
Please test your scene, and check that, after being hit by a bullet and collecting a health pack, your health is increased (i.e., you can check the **Output** window).

Last but not list, we will display the health and life information onscreen:

- Using the **Scene Tree**, please duplicate the node called **message** (**CTRL/Apple + D**) and rename the duplicate **player_info**.
- Using the 2D workspace, move this label to the top of the screen.



- Using the **Inspector** tab, modify the horizontal alignment of this node to **left**.



player_info

Filter properties

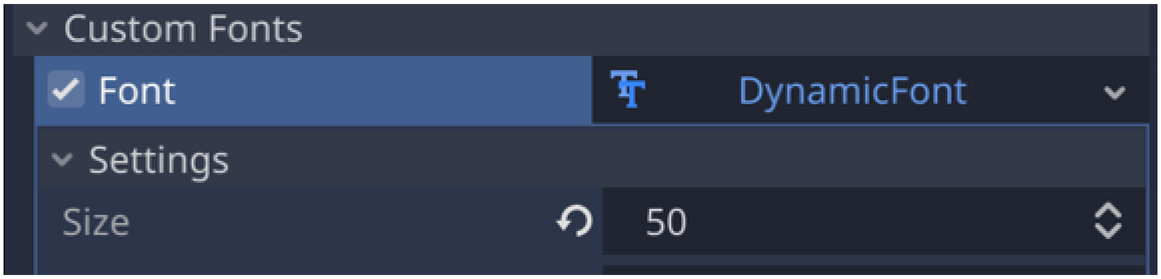
Label

Text

Align: Left

Valign: Center

- Using the **Font** section, modify the **font size** to **50**.



Custom Fonts

☑ Font: DynamicFont

Settings

Size: 50

Then, we just need to modify the script **Player.gd** so that the information about the health and the number of lives for the player are updated.

- Please open the script **Player.gd**.
- Add the following code before any function is declared:

```
onready var player_info:Label = get_node("player_info")
```

- Add this function.

```
func update_player_info():
var message = "Lives:"+str(nb_lives);
message += "\nHealth: "+str(health)
```

```
player_info.set_text(message)
```

In the previous code:

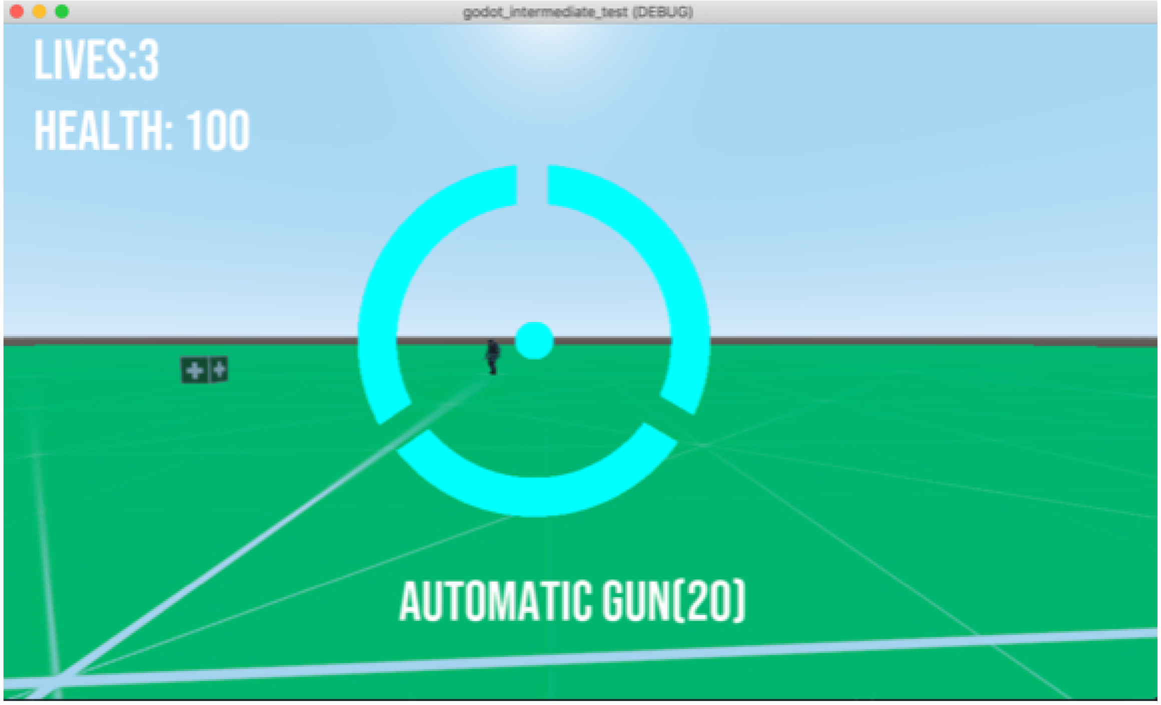
- We create a function called **update_player_info**.
- We then create the message to be displayed onscreen
- This message includes 2 lines, each providing information on the number of lives and the health levels.
- We use **"\n"** to go to the next line.
- This information is concatenated and then displayed by accessing the node **player_info**.

Finally, we just need to call this function when the scene is loaded, when the player gets hit, and when the level is restarted after.

- Please add the following code to the function **_ready**, as well as the functions **restart_level** and **got_hit**:

```
update_player_info()
```

- You can now play the scene and check that the number of lives and health levels are displayed onscreen.

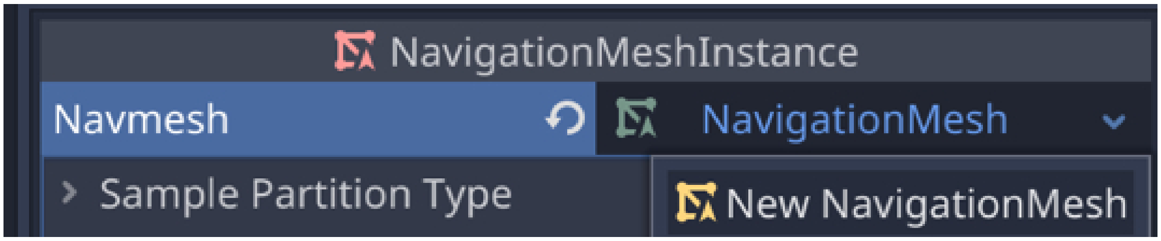


Getting the npc to navigate the scene

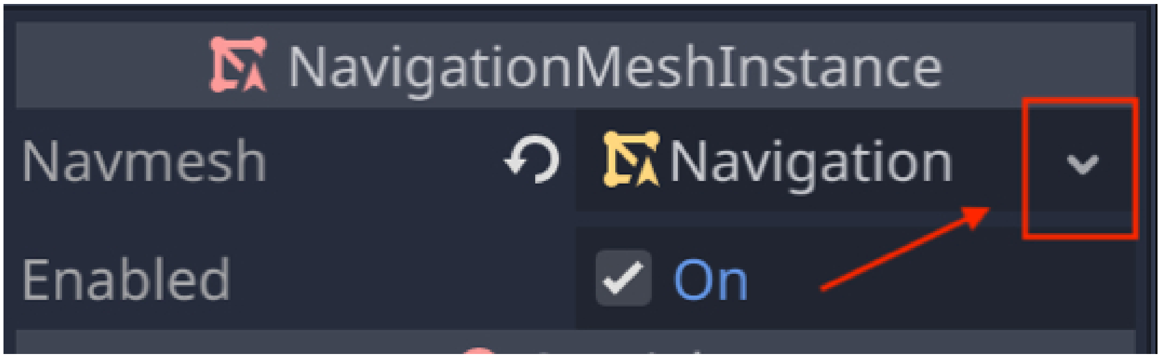
At this stage, we have managed to implement an NPC who can detect the player, shoot in its direction and inflict damage to the player as well. However, as would be expected in most FPS games, it would be good for our NPC to be able to move around, to either follow the player or to follow a specific path. So in this section, we will start to implement the navigation for the NPC using what is usually referred as pathfinding and navmesh.

Please do the following:

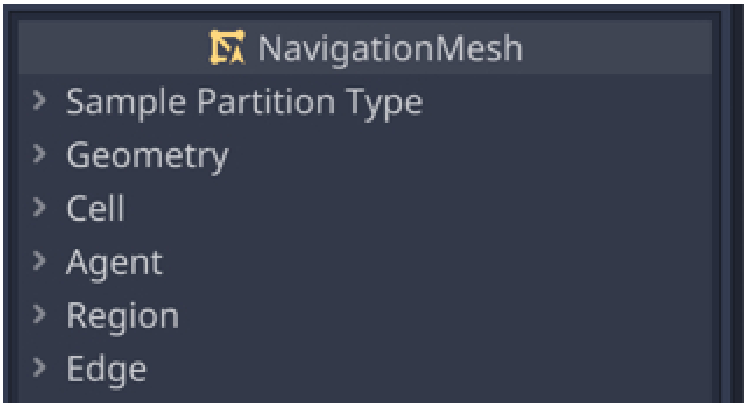
- Create a new node of type **Navigation** as a child of the node **Spatial**.
- Set its **y** coordinate to **.47**.
- Create a new node of type **NavigationMeshIntance** as a child of the node **Navigation**.
- Rename this node **ground_mesh**.
- Select this **ground_mesh** node, and, using the **Inspector** window, click on the downward-facing arrow to the right of the label **Navmesh**, and select "**New NavigationMesh**".



- Once this is done, please click on the arrow to the right of the label **NavigationMesh** and select the option called **Edit** from the contextual menu.

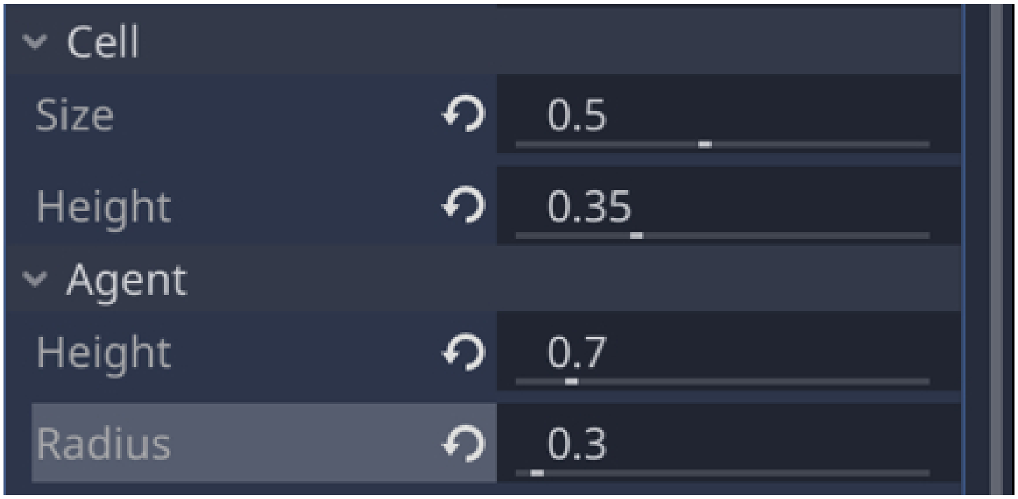


- This will display more settings.



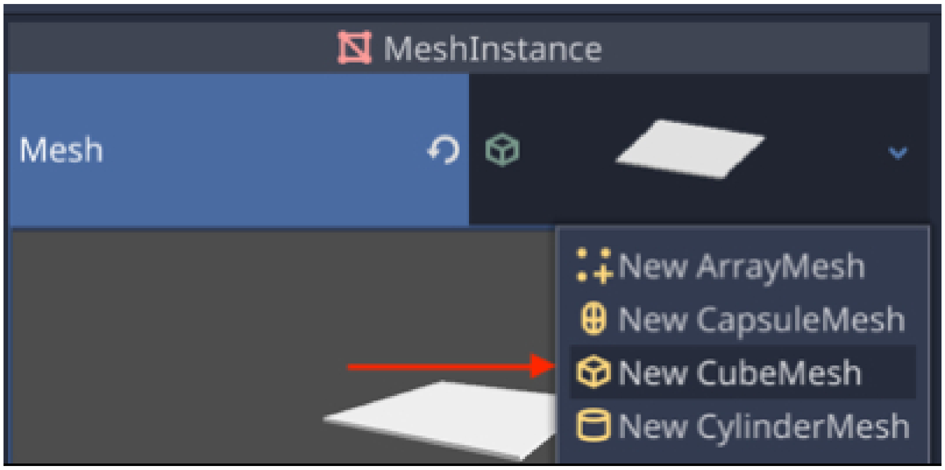
Modify the following settings:

- In the section called **Cell**, set the size to **0.5** and the height to **0.35**.
- In the section called **Agent**, set the height to **0.7** and the radius to 0.**3**.
- You can leave the other default parameters as they are.

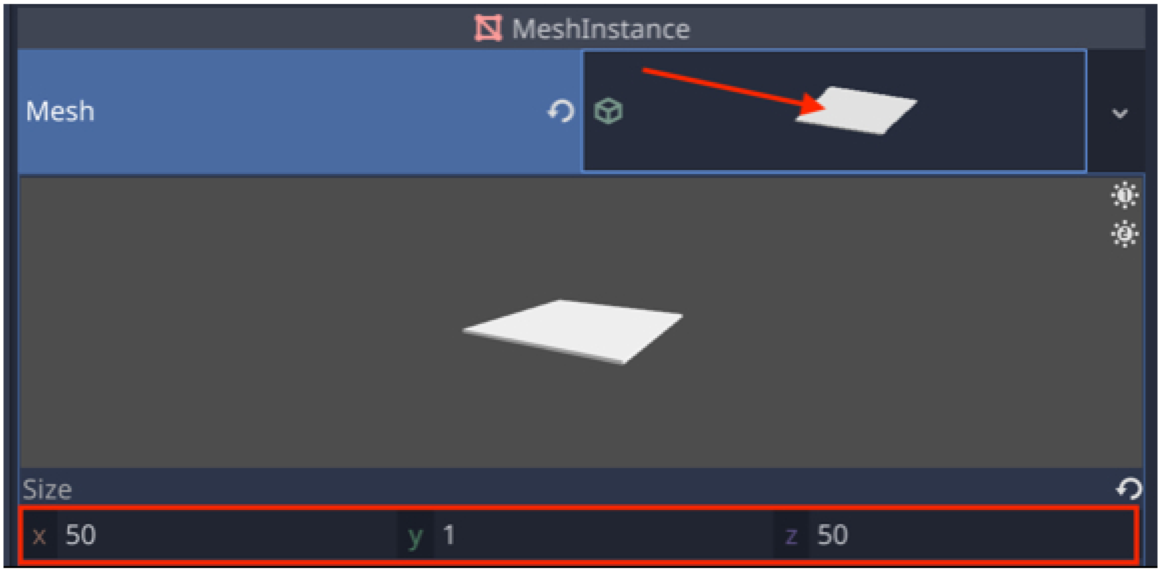


Cell
Size 0.5
Height 0.35
Agent
Height 0.7
Radius 0.3

- Once this is done, please add a new node of type **MeshInstance** as a child of the node **ground_mesh**.
- Select the node **MeshInstance** and, using the **Inspector**, click on the downward facing arrow to the right of the label **Mesh** and select "**New Cube Mesh**".



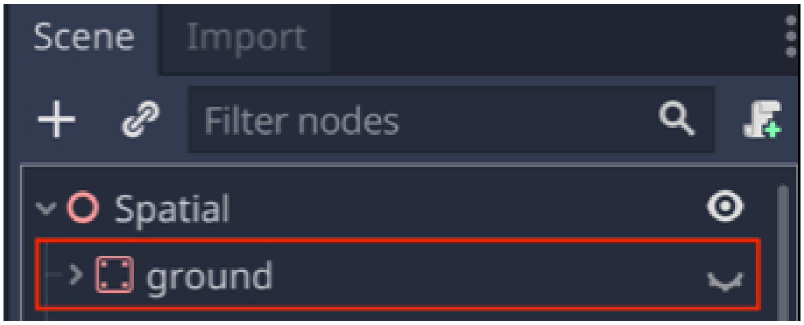
- Once this is done, you can click on the white cube that appears to the left of the downward facing arrow, this should display additional properties, including the **size** property.



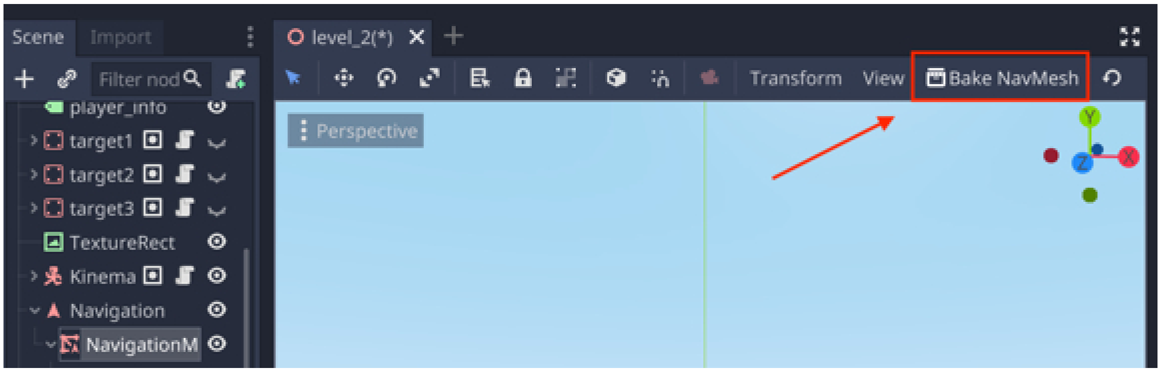
Size

x 50 y 1 z 50

- Please set the size to **(50, 1, 50)** so that this mesh covers the same area as the ground that we have previously created.
- You can also apply a texture to this node by modifying its Material attribute.
- Finally, you can now hide the node called **ground**, since the current node will be used as a ground for navigation.



Finally, please click on the node called **ground_mesh**, then click on the button labelled "**Bake NavMesh**".



At this stage, we have created the meshes that we need to make it possible for the NPC to navigate the scene.

Next, we need to create the code that will move the NPC towards the player. This code will consist in:

- Creating a state called **CHASE** in which the **NPC** will be chasing the player.
- Switch to this state when the player presses a specific key, for testing purposes, and then based on a specific event.
- Generating a path between the player and the NPC.
- Creating intermediary waypoints on this path that the NPC can follow to reach the player.
- Store these waypoints as part of an array.
- Move the NPC along this path by moving from one waypoint to the next, and by moving to the next waypoint when close enough to the current target waypoint.
- Transition to the state **IDLE** when close to the player.

So let’s get started:

- Please open the script **ManageNPC2**.
- Modify this code (new code in bold).

enum {HIT, IDLE, DYING, **CHASE**, SHOOT}

In the previous code, we create a new state called **CHASE** that will be used when the NPC is chasing the player.

- Please add the following code at the beginning of the script **ManageNPC2**.

```
var path = []
var path_node = 0
var speed = 7
onready var nav = get_node("/root/Spatial/Navigation")
onready var player = get_node("/root/Spatial/player")
```

- Please add the following function to the script.

```
func calc_path:
if (current_state == CHASE):
path = nav.get_simple_path(global_transform.origin, player.global_transform.origin, true)
path_node = 0
```

In the previous code:

- We create a new function called **calc_path** that will generate a path between the NPC and the player.
- We then generate the way points that make up the path from the NPC to the player. These waypoints are part of an array called path. This array is generated using the built-in function **get_simple_path** which takes two parameters: the starting position (i.e., the position of the NPC) and the destination (i.e., the position of the player).

- Finally, we set the variable **path_node** to 0 so that the first waypoint is used as the first intermediary destination.

Now that we have created a function that generates a path between the NPC and the player, we need to use this path in the state called **CHASE**, so that the NPC moves towards the player when it is in that state.

- Please add the following code at the start of the function _process:

```
func _process(delta):
var distance_to_player = (player.global_transform.origin - global_transform.origin)
```

- Add this code to the function **_physics_process** within the **match** structure that we have already created.

```
CHASE:
get_node("NPC/AnimationPlayer").play("Walking")
if (path_node < path.size()):
var direction = (path[path_node] - global_transform.origin)
if (direction.length() < 1) :
path_node+=1
else:
move_and_slide(direction.normalized()*speed, Vector3.UP)
look_at(global_transform.origin - direction, Vector3.UP)
if (distance_to_player.length() < 1.5) : current_state=IDLE
```

In the previous code:

- We play the walking animation.
- We then check whether the current path node is not the last one (i.e., that we haven't reached our destination.
- If that is not the case we then check whether we are less than a meter away from the next waypoint; if we are close to the next waypoint, we then increase

the index of the current waypoint to reach, otherwise, we move towards the next waypoint and also look towards it.

- If we are close to the destination, we then set the state to **IDLE**.

That's it; the last thing we need to do now is to trigger the chasing mode manually for now by pressing a key on the keyboard. So please add the following code (new code in bold).

```
func _input(event):
if (Input.is_key_pressed(KEY_P)) :
#current_state = HIT
calc_path()
```

In the previous code, if the key "P" is pressed, we call the function **calc_path**.

Please save your code.

Before we perform our test, we can temporarily comment this code, so that the NPC walks all the way to the player:

```
if ray.is_colliding():
pass
# var obj = ray.get_collider()
# if (obj.name == "player"):
#  param_can_see_player = true
# else: param_can_see_player = false
```

Finally, for testing purposes, please modify the following code:

```
var current_state = IDLE
```

to this code

```
var current_state = CHASE
```

You can now save your code, move the player away from the **NPC** and press the key "**P**", you should see that the **NPC** walks towards the player and then stops when it is about **1.5** meter away from the player.

LEVEL ROUNDUP

In this chapter, we have learned how to create and manage a finite state machine. We became more comfortable with creating states, transitions, and parameters. We managed to create NPCs that behave relatively realistically and to control their behaviors through our scripts. So, again, we have covered considerable ground to produce a relatively interesting scenario. In the next section, we will create a last level where we put all these skills together.

Checklist

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

- Create a Finite State Machine.
- Switch between states.
- Add animations to a state and configure this animation.
- Start an animation from your script.

Quiz

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

1. It is possible to import animated characters in Godot.
2. An **AnimationPlayer** node can contain 3D animations.
3. Using an **AnimationPlayer** node, it is possible to preview each animation within.
4. The following code will ensure that the animation contained in an **Animation Player** node is not looping

```
get_node("NPC/AnimationPlayer").get_animation("IDLE").loop = false
```

1. The function **_process** is called every frame.
2. **Enums** can be used to list different states that belong to a state machine.
3. The function **_input** can be used to detect keyboard events.
4. The following code can be used to wait until a given animation has been played fully.

```
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
```

1. If an NPC is using a finite state machine, it can only be in one state at any given time.
2. Provided that a Raycast node called **ray** has been properly configured, the following code will be called if it collides with an object:

```
func _physics_process(delta):
if ray.is_colliding():
var obj = ray.get_collider()
```

Answers to the Quiz

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

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, you could use these to improve the flow of your game. So for this challenge, you will be creating a new type of NPC that will behave as follows:

- Follow a random path.
- Follow the player when it sees the player or after being hit.
- Shoot the player when the player is in sight.

Chapter 4: More Artificial Intelligence

In this section, we start to work with Finite State Machines (FSM) to be able to manage NPCs and how they behave depending on the environment. We will also get to design some basic, intermediate and advanced artificial intelligence along with animated characters.

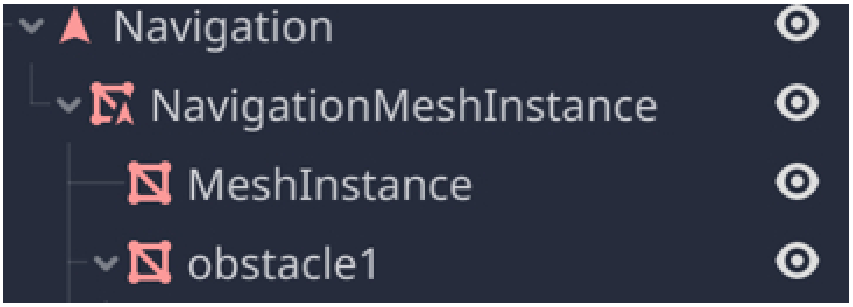
After completing this chapter, you will be able to:

- Create and manage a Finite State Machine.
- Associate character animations to different states.
- Use the FSM to implement basic and intermediate types of artificial intelligence.
- Simulate vision detection for the NPCs.
- Get the NPCs to behave realistically.

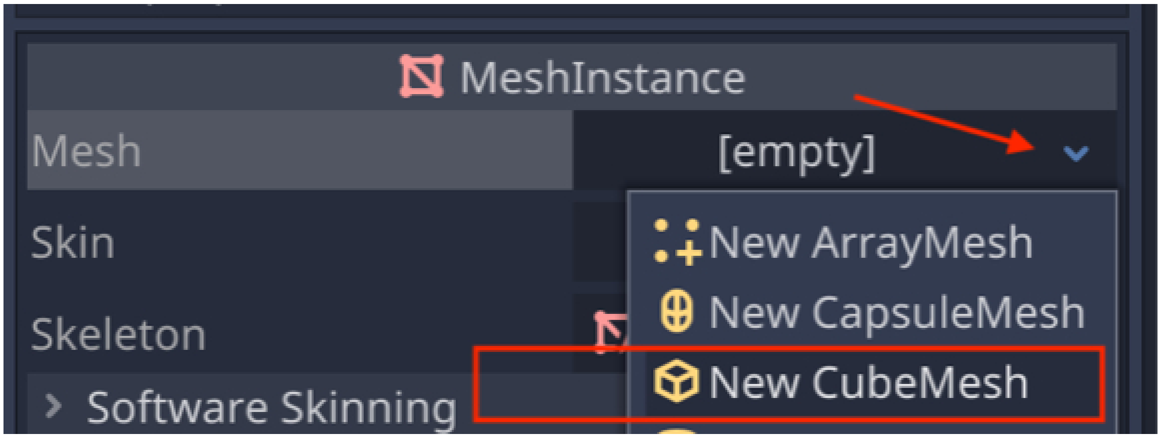
ADDING OBSTACLES FOR THE NPC TO AVOID

Once this is working, we can create obstacles and try to see if the NPC can avoid these in order to reach the player.

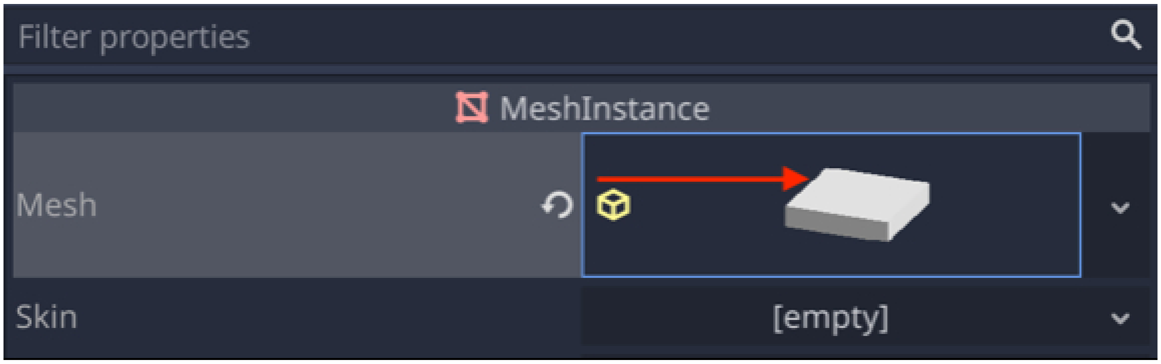
- Please create a new **MeshInstance** node as a child of the node **Navigation-MeshInstance (**or **ground_mesh** depending on how you called it**)** and rename it **obstacle1**.



- Select this node (**obstacle1**).
- In the **Inspector**, click on the downward facing arrow to the right of the label **Mesh**, and select the option **New CubeMesh**.



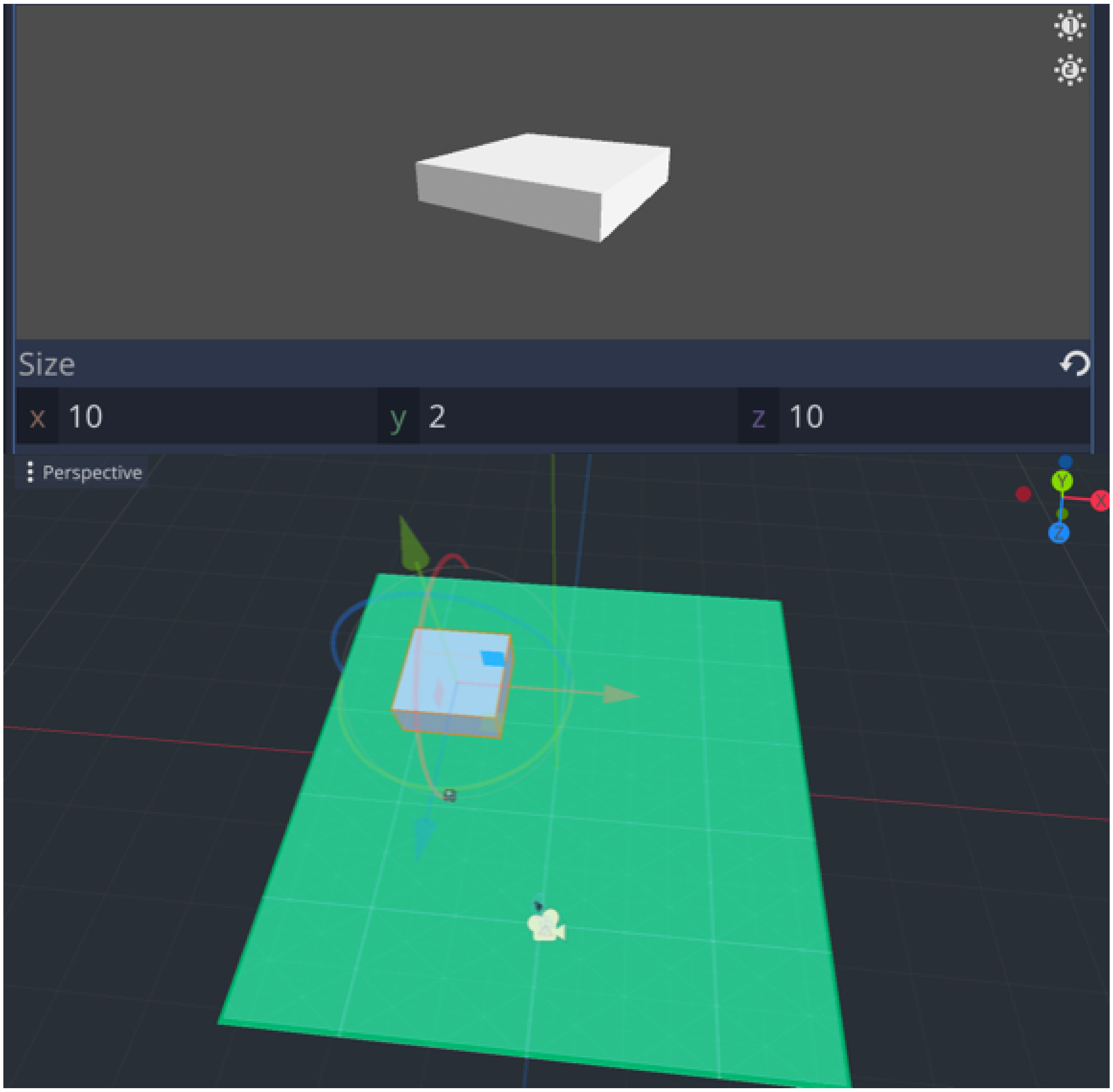
- In the **Inspector** window, click on the white cube.



Mesh

Skin [empty]

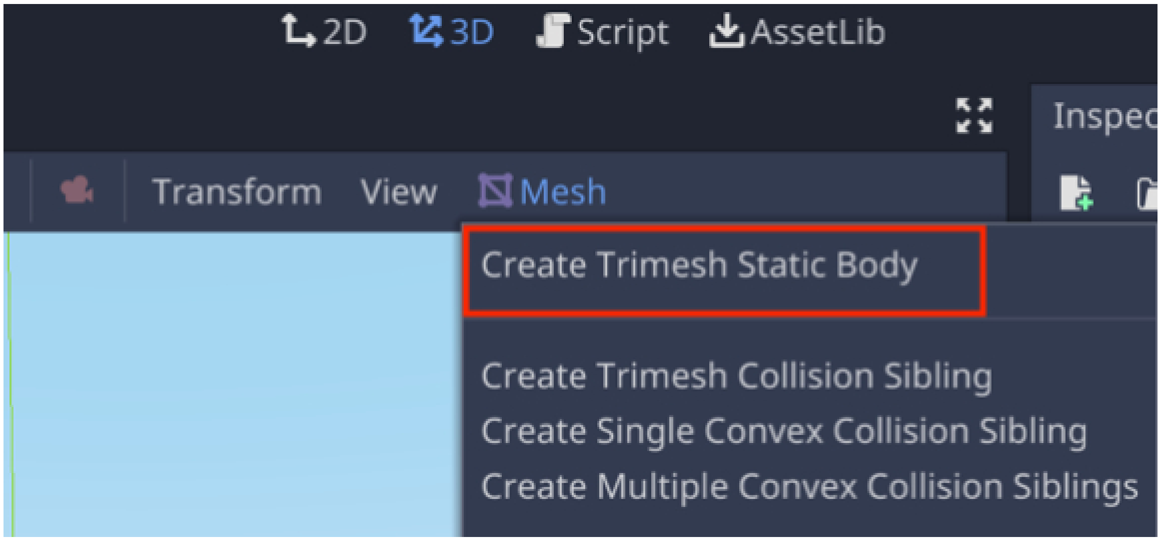
- Change the size to **(10, 2, 10)**.
- Using the **Transform** attribute, set its y coordinate to **2**.



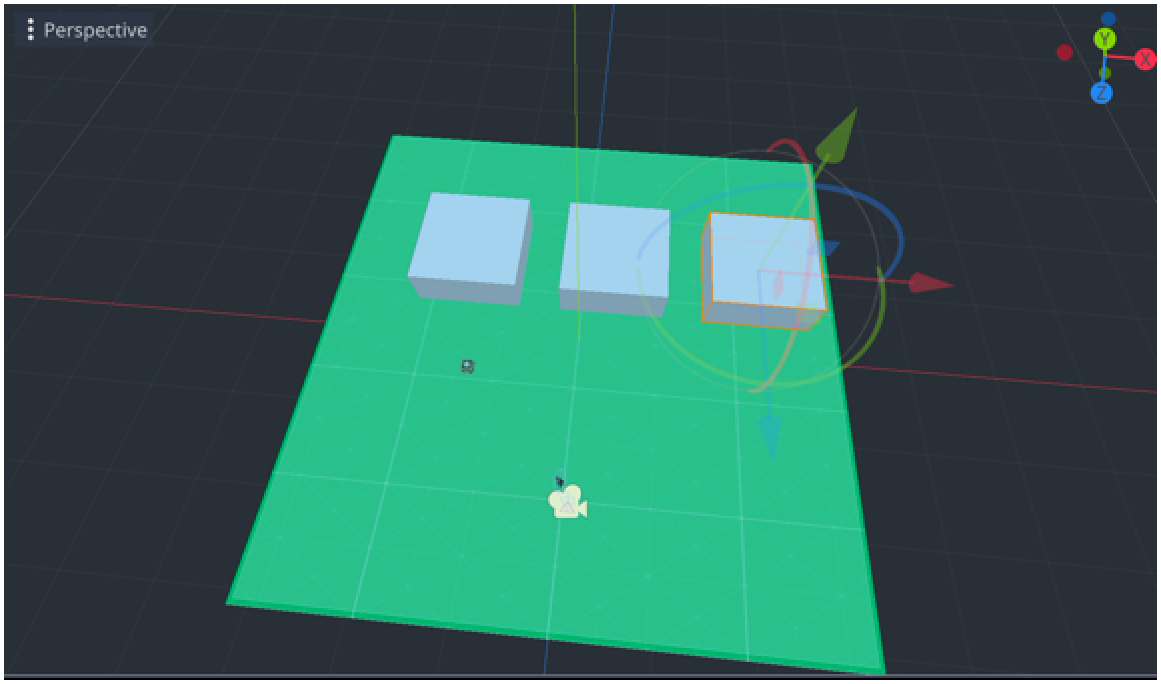
Size

x 10 y 2 z 10

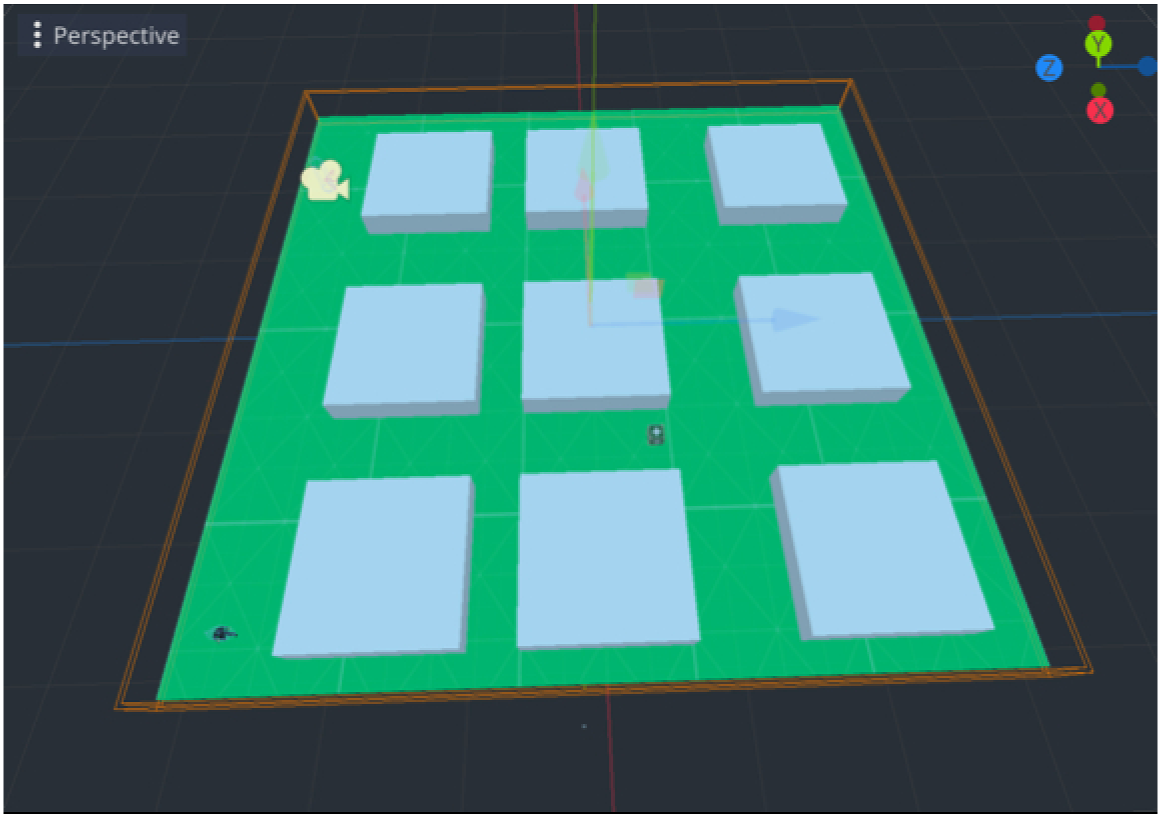
- In the top menu select **Mesh | Create Trimesh Static Body**.


2D
3D
Script
AssetLib
Inspec
Transform
View
Mesh
Create Trimesh Static Body
Create Trimesh Collision Sibling
Create Single Convex Collision Sibling
Create Multiple Convex Collision Siblings

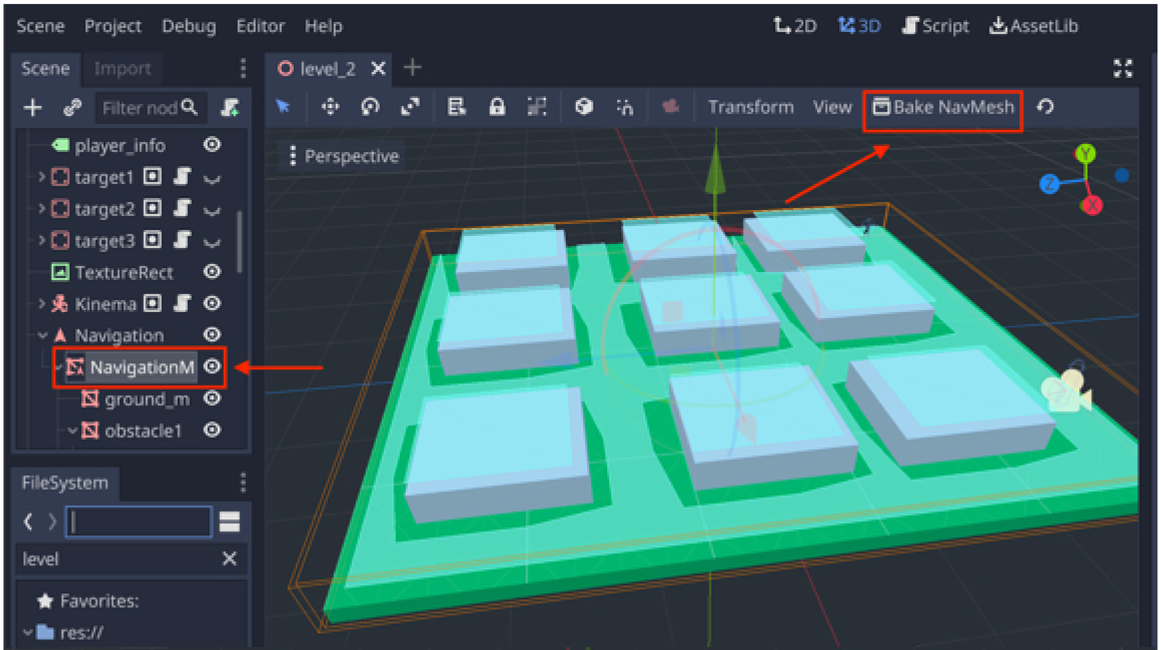
- This will add collision to your wall.
- Duplicate this mesh twice (i.e., **obstacle1**) and move the duplicates apart, as per the next figure.

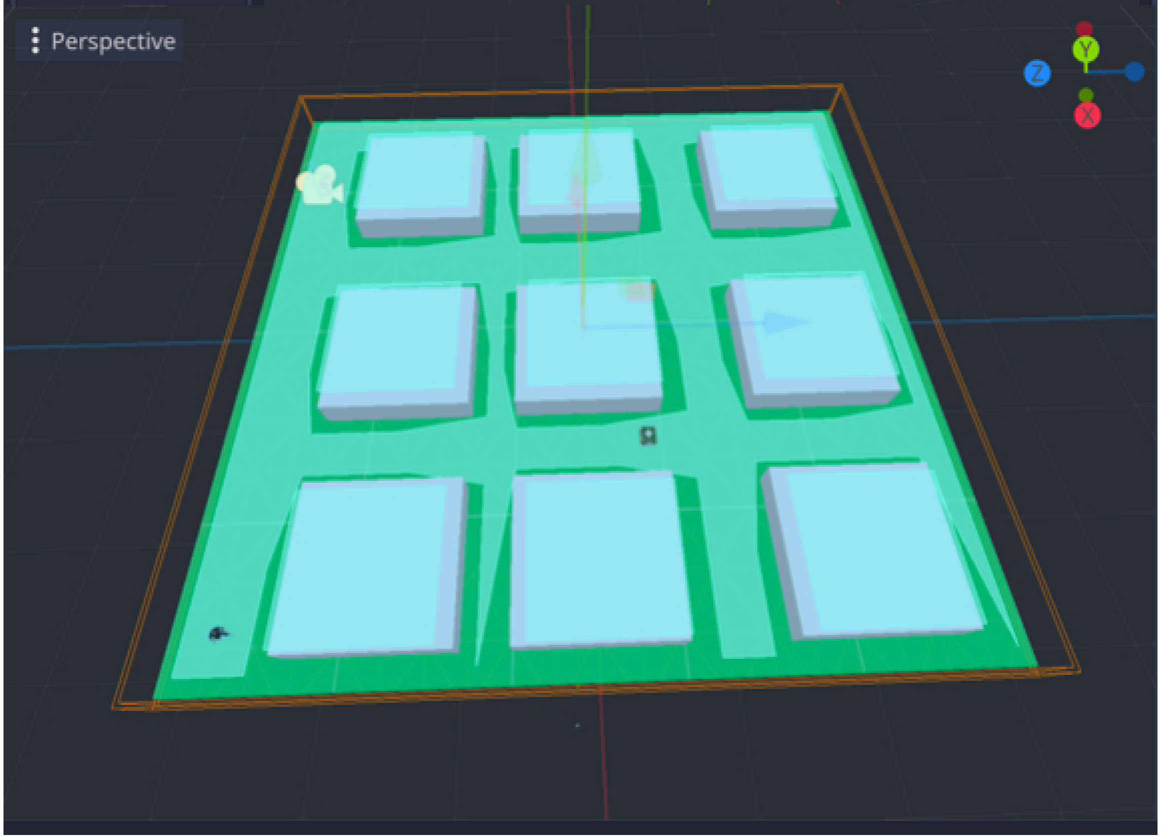


- Repeat this to create nine boxes in total, as per the next figure.



- Move the player and the **NPC** so that they are in different locations, and so that the health pack is not hidden by the obstacles
- You can now bake the scene by selecting the node **NavigationMesh** and then clicking on the button labelled "**Bake NavMesh**".





You can now play the scene, move around the maze and play the **P** key, and the NPC should move to your current location

GETTING THE NPC TO FOLLOW YOU CONSTANTLY IN THE CHASE MODE

At this stage, we have managed to get the NPC to follow the player after pressing a specific key; however, a more common behaviour is for the NPC to either follow the player after detecting him/her or to deliberately look for the player. So in this section, we will implement a behaviour whereby:

- The NPC is initially in the **IDLE** state.
- If the NPC sees the player or is being hit, it will shoot at the player and then chase the player.
- At a very close range, the NPC will punch the player and inflict damage.

So let's get started:

- Please open the script **ManageNPC2** and uncomment this code (previously commented for testing purposes):

```
if ray.is_colliding():
pass
var obj = ray.get_collider()
if (obj.name == "player"):
param_can_see_player = true
else: param_can_see_player = false
```

You may remember that, previously, we called the function **calc_path** manually to test whether the path between the NPC and its target was created and followed properly by the NPC; this being said, in our game, if the NPC is chasing the player, we need this path to be updated frequently. For this reason, we need to regularly call the function **calc_path**, and this can be achieved with a timer.

- Please add this code before the first function.

```
var path_timer:Timer
```

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

```
path_timer = Timer.new();
add_child(path_timer)
path_timer.wait_time=1
path_timer.connect("timeout",self,"calc_path")
path_timer.start()
```

In the previous code:

- We instantiate the timer.
- We add the timer to the current node.
- We set the attribute **wait_time** to **1**.
- We connect the **timeout** event to the function **calc_path**.
- We start the timer.
- In other words, we state that every second, the function **calc_path** will be called.

You can now save your code and test the scene; the NPC should automatically chase you and then shoot towards you at close range. You could also, for example, change the following code at the beginning of the script so that the NPC starts in the **IDLE** mode and not the **CHASE** state (new code in bold).

```
enum {HIT, IDLE, DYING, CHASE, SHOOT}
var current_state = IDLE
```

- Please also add the next code (new code in bold):

```
HOOT:
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
get_node("NPC/AnimationPlayer").play("Shooting")
if (can_inflict_damage):
get_node("../player").got_hit()
shoot_timer.start()
```

```
can_inflict_damage = false
if (!param_can_see_player): current_state = CHASE
```

In the previous code, we ensure that if the NPC is shooting and the player is no longer in the NPC's line of sight, then the NPC should chase the player.

Please save your code and play the scene. If you walk in front of the NPC, it should fire its weapon, and then chase you if you run away.

Once this is done, we now need to ensure that the NPC punches the player at close range (instead of shooting), this will mean creating a new state called **Punching**, using its corresponding animation, and triggering this state when the NPC is close to the player.

- Please add this code before the first function in the script **ManageNPC2**.

```
var punch_timer:Timer
var can_inflict_punch_damage = true
```

In the previous code, we create two variables that will be used to ensure that the players' energy does not deplete too fast while being punched. Every time we are in the state called **PUNCH** we will ensure that the player's health is decreased only once during the duration of the punching animation (and not every frame). For this purpose, we will ensure that the energy is depleted only when the variable **can_inflict_punch_damage** is true, and this variable will be set to true only after 2.33 second (the duration of the punching animation) using the timer **punch_timer**.

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

```
get_node("NPC/AnimationPlayer").get_animation("Punching").loop = false
punch_timer = Timer.new()
add_child(punch_timer)
punch_timer.wait_time = 2.23
punch_timer.connect(("timeout"),self,"enable_punching")
```

In the previous code:

- We make sure that the punching animation is not looping.

- We instantiate the timer and add it to the current node.
- We set the waiting time to 2.23 seconds (the duration of the punching animation).
- We then connect the event timeout for this timer to the function **enable_punching**, so that every 2.33 seconds, we ensure that the NPC can punch again.

We can now specify what should be done in the state **PUNCH**.

- Please modify this code (new code in bold).

```
enum {HIT, IDLE, DYING, CHASE, SHOOT, PUNCH}
```

- Add the following code before the first function:

```
onready var gun = get_node("NPC/IDLE/Skeleton/BoneAttachment 9/swatRightHandMiddle4/gun")
```

In the previous code we create a variable called **gun** that is linked to the node **gun**.

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

```
PUNCH:
gun.hide()
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
if (can_inflict_punch_damage && distance_to_player.length() <= 1.5):
punch_timer.start()
can_inflict_punch_damage = false
get_node("NPC/AnimationPlayer").play("Punching")
yield(get_tree().create_timer(1.0), "timeout")
get_node("../player").got_hit()
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
```

```
elif (distance_to_player.length() > 1.5):
calc_path()
current_state = CHASE
```

In the previous code:

- We create a new state called **PUNCH**.
- In this state, we hide the gun, since the **NPC** will not be punching with the gun.
- We then use the built-in function **look_at** so that the NPC rotates to face the player.
- We then check that the NPC can inflict damage to the player (i.e., once during the punching animation) and that the NPC is close enough to the player.
- Once this is done, we start the timer **punch_timer** which will ensure that the damage to the player is applied only once during the animation.
- We set the variable **can_inflict_damage** to false.
- We then play the punching animation, wait for one second, and then call the function **got_hit** to apply damage to the player.
- We then wait until the animation has completed.
- Finally, if the NPC is too far away from the player, it will switch to the state **CHASE**.

Next, we just need to create the function **enable_punching**, so please add this function:

```
func enable_punching():
can_inflict_punch_damage = true
```

In the previous code, we create the function **enable_punching** that sets the variable **can_inflict_punch_damage** to true.

Once the **PUNCH** state is complete, we just need to ensure that if the NPC is chasing the player and gets very close, that it switches to the state **PUNCH**.

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

```
CHASE:
get_node("NPC/AnimationPlayer").play("Walking")
if (path_node < path.size()):
var direction = (path[path_node] - global_transform.origin)
if (direction.length() < 1) :
path_node+=1
else:
move_and_slide(direction.normalized()*speed, Vector3.UP)
look_at(global_transform.origin - direction, Vector3.UP)
if (distance_to_player.length() < 1.5) : current_state=PUNCH
```

In the previous code, we just ensure that the NPC switches to the state **PUNCH** if it is less than **1.5** meters away from the player.

Finally, please modify this code (new code in bold):

```
SHOOT:
gun.show()
look_at(get_node("../player").global_transform.origin, Vector3.UP)
rotate(Vector3.UP, 3.18)
```

At this stage, the NPC should follow the player after seeing him/her, and punch the player at close range to inflict damage.

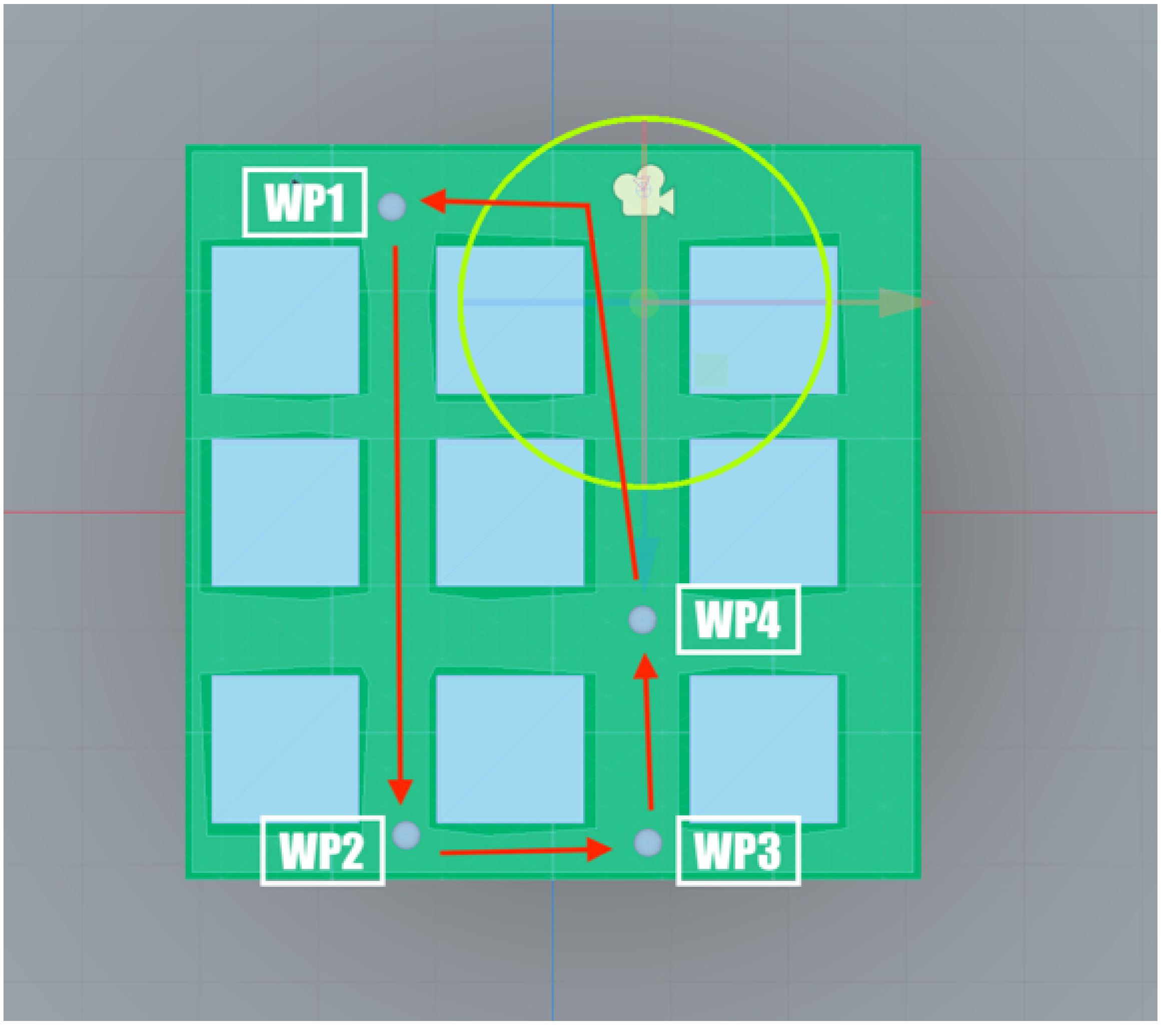
MAKING IT POSSIBLE FOR THE NPC TO FOLLOW A PATH

So at this stage, the NPC is able to detect the player, and start a chase. In this section, we will just create a new state whereby the NPC is walking on a predefined path. This will consist in:

- Creating waypoints that will be used as intermediary destinations along the path.
- Creating a state called **PATROL**.
- Checking that while in that state, the NPC will navigate to subsequent waypoints, changing its destination to the next waypoint when the current waypoint is close.
- Ensuring that if the NPC has reached the last waypoint, its next destination should then be the first waypoint.

First, let's create the waypoints that will be used to define the path to be followed by the NPC:

- Please create four **CSGSphere** nodes as children of the node **Spatial**.
- Rename them **WP1**, **WP2**, **WP3** and **WP4**.
- Ensure that their **y** coordinate is **2**.
- Move them to different locations of your choice.



Once they have been created, it is time to change our code and ensure that the NPC is following them.

- Please open the script **ManageNPC2**.
- Modify the following code (new code in bold).

```
enum {HIT, IDLE, DYING, CHASE, SHOOT, PUNCH, PATROL}
```

- Add the following code before the first function:

```
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_path_node = 0
```

In the previous code:

- We create the variables **WP1**, **WP2**, **WP3** and **WP4** that are linked to the waypoints that we have just created.
- We create a variable called **WP_index** that will be used to know the index of the current waypoint to reach while patrolling on the path.
- We create a variable called **current_WP** that relates to the current waypoint being followed by the NPC.
- Finally, we create a variable called **patrol_path_node** that will be used to calculate the path to the current waypoint.

Now we can create a new state called **PATROL** that will be used when the NPC is following the path that we have just defined.

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

```
current_WP = WP1
calc_patrol_path()
```

- Modify the **match** structure by adding this code to it:

```
PATROL:
var distance_to_WP = (current_WP.global_transform.origin - global_transform.origin)
get_node("NPC/AnimationPlayer").play("Walking")
if (path_node < path.size()):
var direction = (path[path_node] - global_transform.origin)
if (direction.length() < 1) :
path_node+=1
else:
```

```
move_and_slide(direction.normalized()*speed, Vector3.UP)
look_at(global_transform.origin - direction, Vector3.UP)
if (distance_to_WP.length() < 2) :
WP_index += 1
if (WP_index > 3): WP_index = 0
calc_patrol_path()
```

In the previous code:

- We create a state called **PATROL**.
- In that state, we calculate the distance between the NPC and the current waypoint (i.e., the one the NPC is trying to reach).
- We then play the walking animation.
- We then use the exact same code as we have done in the **CHASE** state; this code will move the NPC towards the current waypoint, unless the NPC has reached this location already.
- We also check whether the NPC is 2 meter away from the current waypoint it is trying to reach; if that's the case, then the NPC will try to reach the next waypoint.
- If the last waypoint of the path has been reached, the NPC will now try to reach the first one.

Finally, we just need to create the function **calc_patrol_path**, so please add the following function to the script:

```
func calc_patrol_path():
match (WP_index):
0:current_WP = WP1
1:current_WP = WP2
2:current_WP = WP3
3:current_WP = WP4
path = nav.get_simple_path(global_transform.origin, current_WP.global_transform.origin, true)
```

```
path_node = 0
```

In the previous code, we check the index of the current waypoint, we set the corresponding destination, and we calculate the path to that destination.

Last but not least, we just need to set the NPC to a **PATROL** state, so please modify the code at the start of the script:

From...

```
var current_state = IDLE
```

to

```
var current_state = PATROL
```

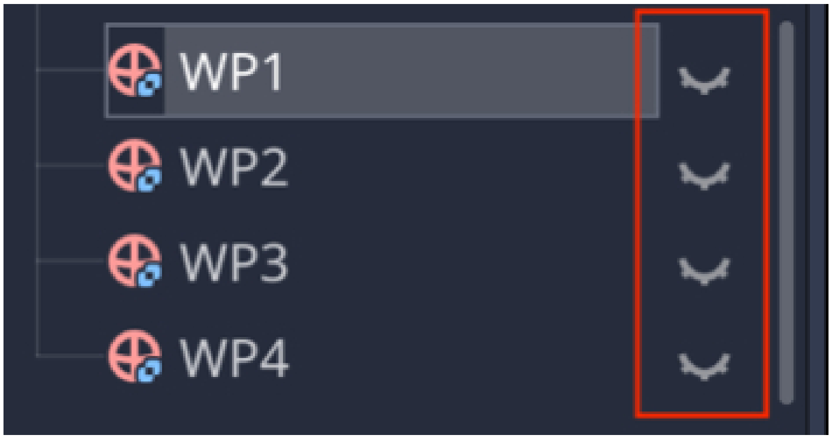
Last but not least, we need to ensure that if the NPC detects the player while patrolling, that it will shoot towards the player; so please add the following code (new code in bold).

```
PATROL:
if (param_can_see_player): current_state = SHOOT
var distance_to_WP = (current_WP.global_transform.origin - global_transform.origin)
get_node("NPC/AnimationPlayer").play("Walking")
```

In the previous code, we just ensure that if the NPC can see the player that its state switches to **SHOOT**.

You can now save your code and test the scene, and you should see that the NPC is following the waypoints that you have created earlier.

Once you have checked this, you can hide the waypoints using the **Inspector**.



Note that if we wanted to add the sense of hearing to the NPC, we could just check for the distance between the player and the NPC and activate the variable **param_can_see_player** if the distance is within a specific range.

ENSURING THAT THE NPC DOES NOT GET STUCK

At this stage, the NPC can chase the player but also follow a predefined path. This being said, it is possible that sometimes the NPC gets stuck and no longer progresses to the next destination. This could be due to several factors; however, there is a way to avoid this situation by checking frequently whether the NPC is progressing, and by changing its target if that is the case, and recalculating the path to the new destination accordingly.

- Please open the script **ManageNPC2** and add the following code before the first function.

```
var check_progress_timer:Timer
var patrol_previous_position
```

In the previous code we create a timer called **check_progress**, along with a variable called **patrol_previous_position**; this variable will be used to track the position of the NPC regularly and detect whether it is stuck (i.e., making no progress).

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

```
check_progress_timer = Timer.new()
add_child(check_progress_timer)
check_progress_timer.wait_time = 2
check_progress_timer.connect(("timeout"),self,"check_progress")
check_progress_timer.start()
patrol_previous_position = global_transform.origin
```

In the previous code we create a timer that will call the function **check_progress** every two seconds; we also store the initial position of the NPC.

- Please add the following function:

```
func check_progress():
if (current_state == PATROL ||  current_state == PATROL ):
var distance_to_previous_position = (patrol_previous_position -
global_transform.origin)
    if (distance_to_previous_position.length() <1):
    calc_path()
    WP_index +=1
    patrol_previous_position = global_transform.origin
```

In the previous code:

- We create a function called **check_progress**.
- In this function, we check that the NPC has moved at least one meter away from its previous position; if this is not the case, we select another waypoint to be reached and we calculate the path to reach it.
- Finally, we store the current position.

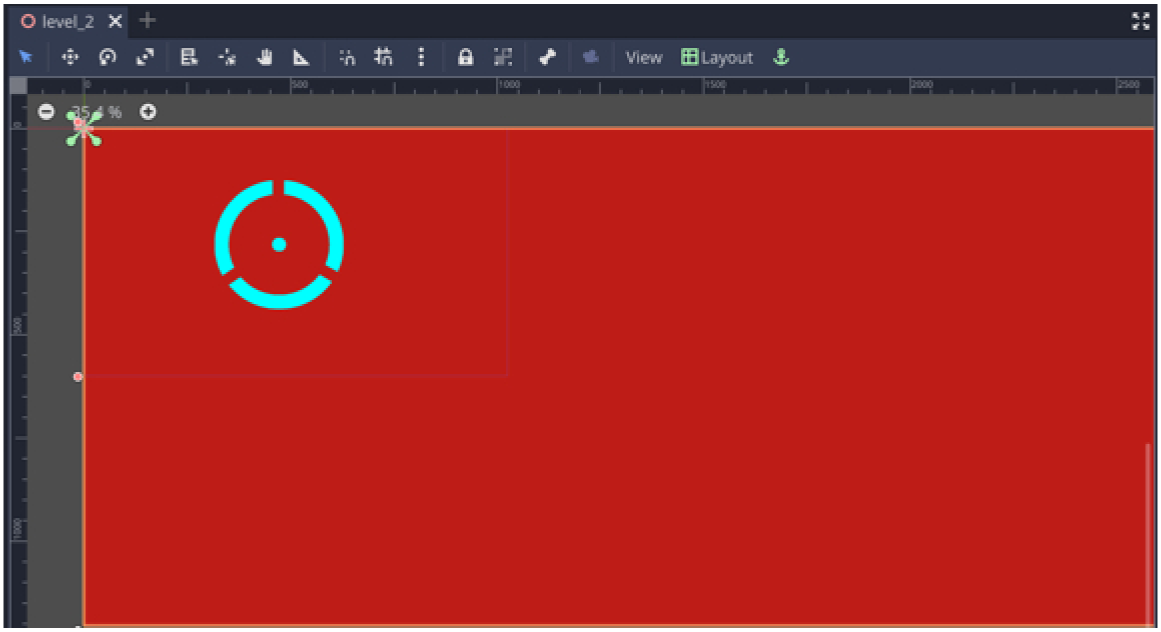
You can now save your code and see if the NPC follows its path properly.

ADDING A SCREEN FLASH WHEN THE PLAYER IS HIT

As the NPC can inflict damage to the player, it would be great to add visual feedback when the player has been hit by the NPC. One common way to do this is to add a screen flash; that is, a brief moment when the screen flashes to red. There are many ways to achieve this effect, and one of them is to create a red texture or color material, stretch it so that it occupies the entire screen, and quickly fade its alpha (i.e., transparency) value from opaque to fully transparent.

So here, we will achieve this effect using a **ColorRect** node.

- Please add a **ColorRect** node as a child of the **player** node and rename this node **screen_flash**.
- Modify its size so that it fills the screen area and change its color to **red**.



Once you have created the UI element for the screen flash, we will control its opacity through a script.

- Please add a new script to this node and call it **screen_flash**.
- Open the script.
- Add this code (new code in bold)

```
extends ColorRect
```

```
var alpha=0
var fade_timer:Timer
```

In the previous code we declare two variables that will be used to define the alpha value (i.e., the transparency) of the current node and to frequently decrease this value once the player has been hit.

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

```
func _ready():
color.a=alpha
fade_timer = Timer.new()
fade_timer.wait_time = 0.1
add_child(fade_timer)
fade_timer.connect("timeout",self,"decrease_alpha")
```

In the previous code we set the transparency of the node; we also set up the timer so that it calls the function **decrease_alpha** every **.1** seconds; since the alpha value will initially be **1**, this means that the alpha value will go from **1** to **0** in 1 second.

- Please add the following function.

```
func start_fade():
alpha = 1;
fade_timer.start()
```

In the previous code, we create the function **start_fade** which sets the node to opaque and then starts the timer **fade_timer** that will progressively decrease the opacity over time.

- Please add the following function:

```
func decrease_alpha():
alpha -= .1
if (alpha <= 0):
```

```
alpha = 0
fade_timer.stop()
color.a = alpha
```

In the previous code, we create the function **decrease_alpha**; whenever it is called (every .1 seconds) it will decrease the value of the variable alpha and apply this value to the current node.

We will now modify the script **Player.gd** so that this red screen flash appears briefly whenever the player is hit.

- Please open the script **Player.gd**
- Add the following code outside of any function

```
onready var screen_flash:ColorRect = get_node("screen_flash")
```

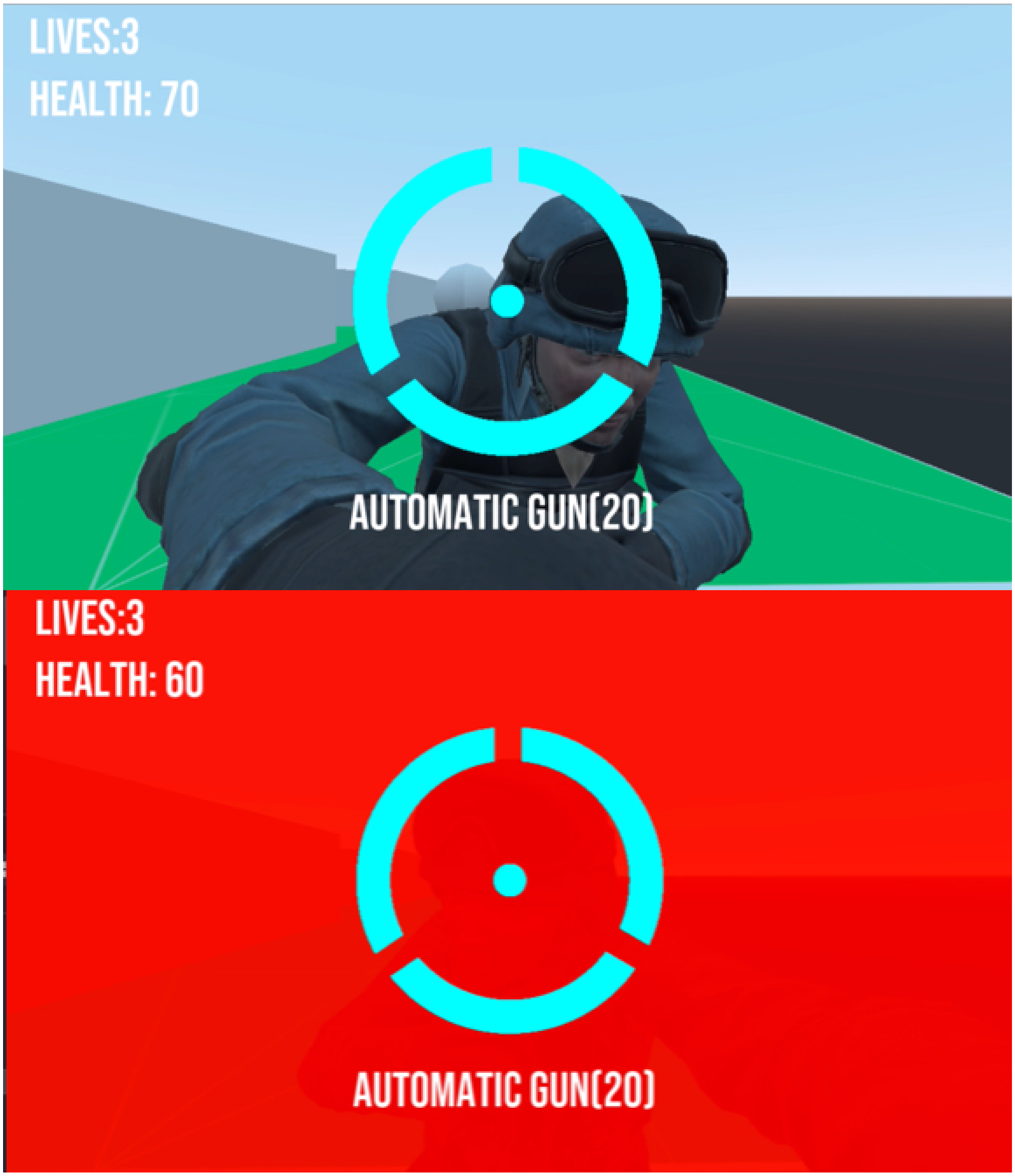
- Add this code to the function **got_hit** (new code in bold)

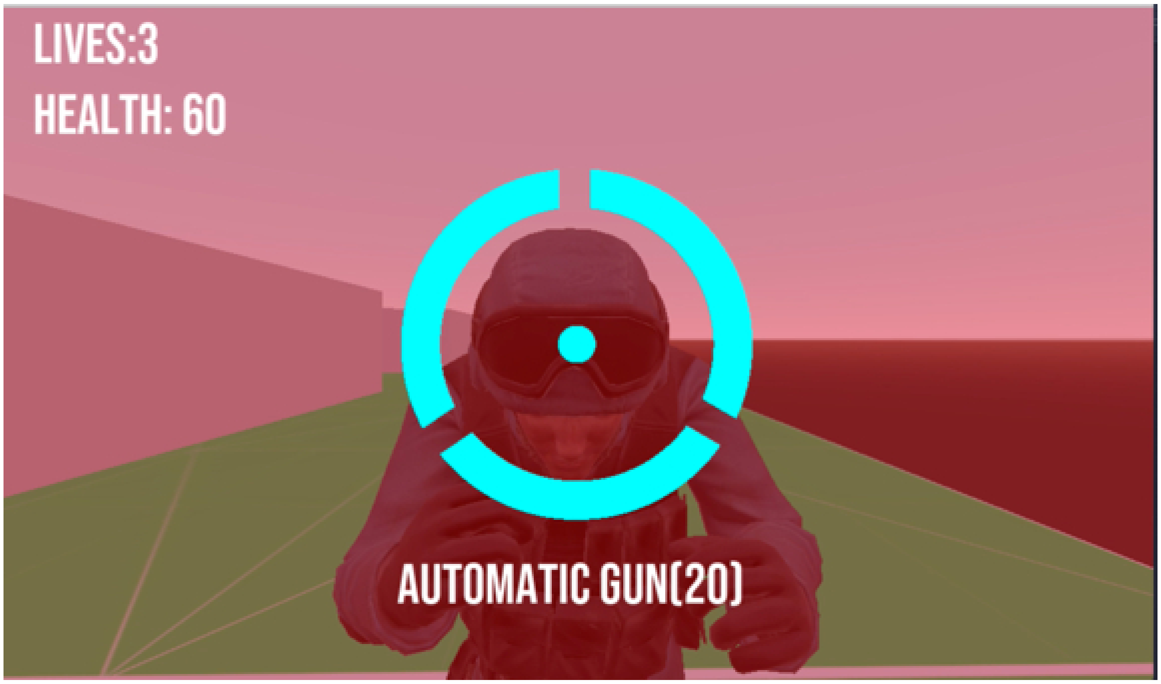
```
func got_hit():
health -= 10
print("Player's Health"+str(health))
screen_flash.start_fade()
```

In the previous code, we call the function **start_fade** whenever the player is hit.

Please save your code and check that the screen flash appears whenever the player has been hit.

The following screenshots show the screen flash effect.


LIVES:3
HEALTH: 70
AUTOMATIC GUN(20)
LIVES:3
HEALTH: 60
AUTOMATIC GUN(20)



So all seems to work well, we can now save our NPC and create a template with it, so that it can be reused at a later stage.

- Please select the node **KinematicBody** used for the NPC.
- Right-click on that node and select the option **Save Branch As Scene**.
- In the new window save as **NPC.tscn**.

SETTING-UP THE ENVIRONMENT

In the next sections we will put all the skills that we have learned to create a fully functional level with the following features:

- The player has three lives.
- The player will then need to collect four objects in the scene to win.
- The player will need to avoid or neutralize NPCs.
- The level will include a combination of safe and dangerous areas.
- Ammunition will be present in places and possibly re-spawn when the player runs low on ammunition.
- NPCs will be present at the start of the game and more will spawn at regular intervals.
- The player wins if he/she has collected all four objects.
- The player will lose if s/he has no lives left.

First, we will create an NPC spawner, a node that will instantiate NPCs at frequent intervals; this will consist in creating an empty node and adding a script to it; this script will be using a timer so that we can instantiate new NPCs every 5 seconds.

- Please create a new node of type **Spatial** as a child of the node **Spatial**.
- Rename this new node **NPC_spawner**.
- Make sure that its **y** coordinate is **1**.
- Add a new script to this node and rename the script **spawn_NPC**.
- Add the following code to it (new code in bold):

```
extends Spatial
export (PackedScene) var NPC
var spawning_timer:Timer
```

In the previous code, we declare two variables: NPC and **spawning_timer**; the former will be used as a placeholder for the NPC node, while the latter will be a timer used to spawn NPCs at regular intervals.

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

```
func _ready():
spawning_timer = Timer.new()
add_child(spawning_timer)
spawning_timer.wait_time = 5
spawning_timer.connect("timeout",self,"instantiate_npc")
spawning_timer.start()
```

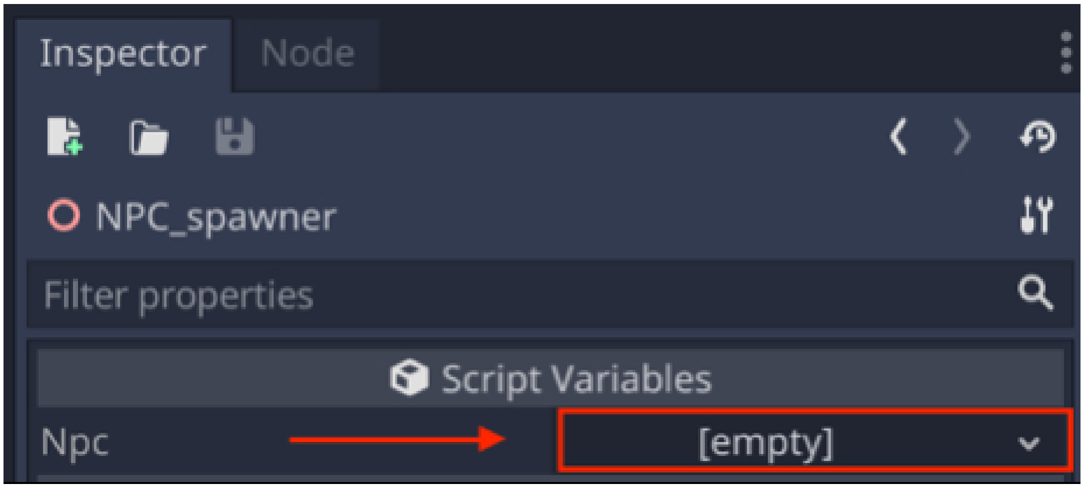
In the previous code, we initiate, set-up, and start the timer **spawning_timer**, so that the function **instantiate_npc** is called every 5 seconds.

- Please add the following function:

```
func instantiate_npc():
var new_scene = load("res://NPC.tscn").instance()
get_parent().add_child(new_scene)
```

In the previous code, we add a new NPC based on the scene **NPC.tscn**.

Please save your code, select the node **NPC_spawner**, switch to the 3D workspace, and you should see an empty field called NPC in the **Inspector** (you may need to click on another node before that to force the **Inspector** to refresh its information).

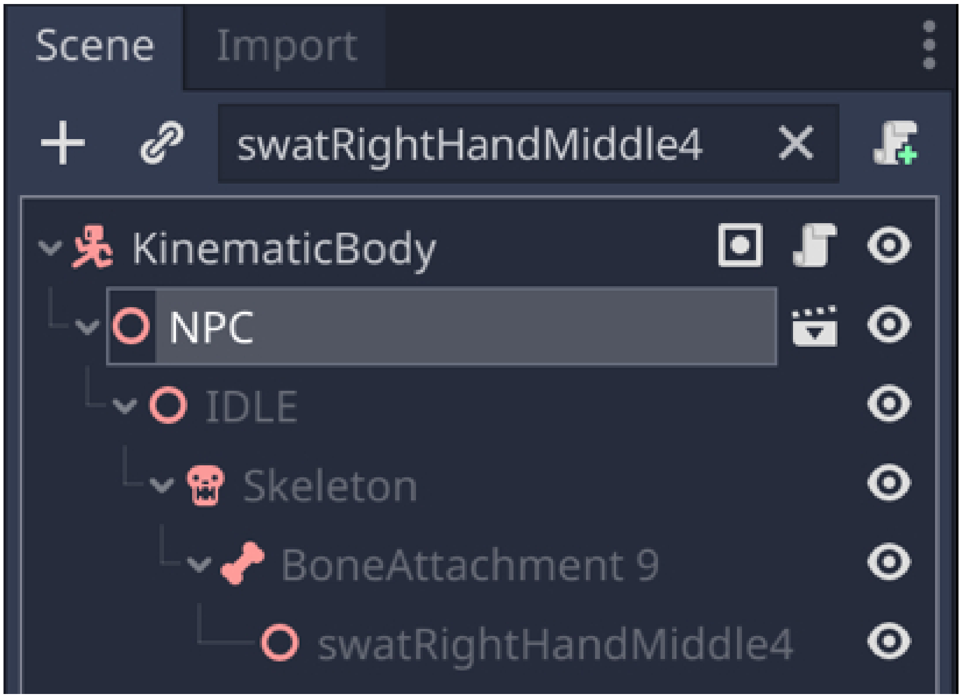


Npc [empty]

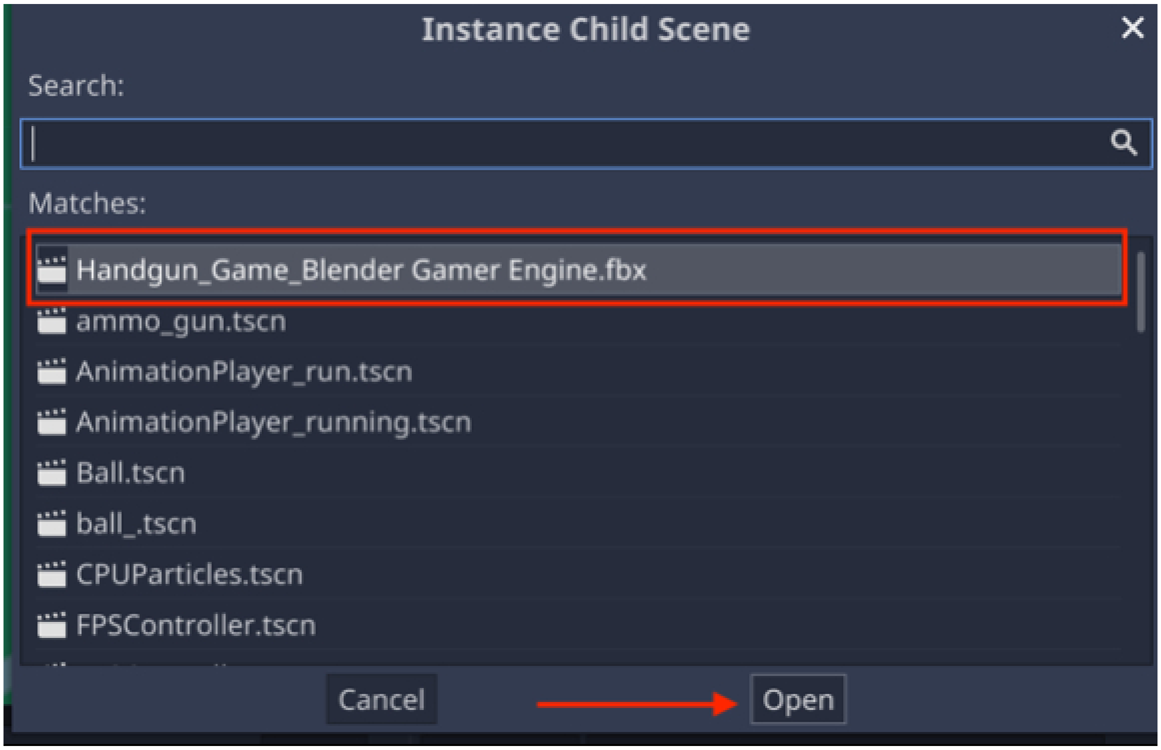
- Please drag and drop the file **NPC.tscn** from the **FileSystem** window to that slot.

You can play the scene and check that a new NPC is instantiated every 5 seconds. As you do, you may notice that if you shoot the NPC, an error related to the presence of the gun appears. If it does, you may need to re-include the gun as part of the NPC node; to do so, you can follow these steps:

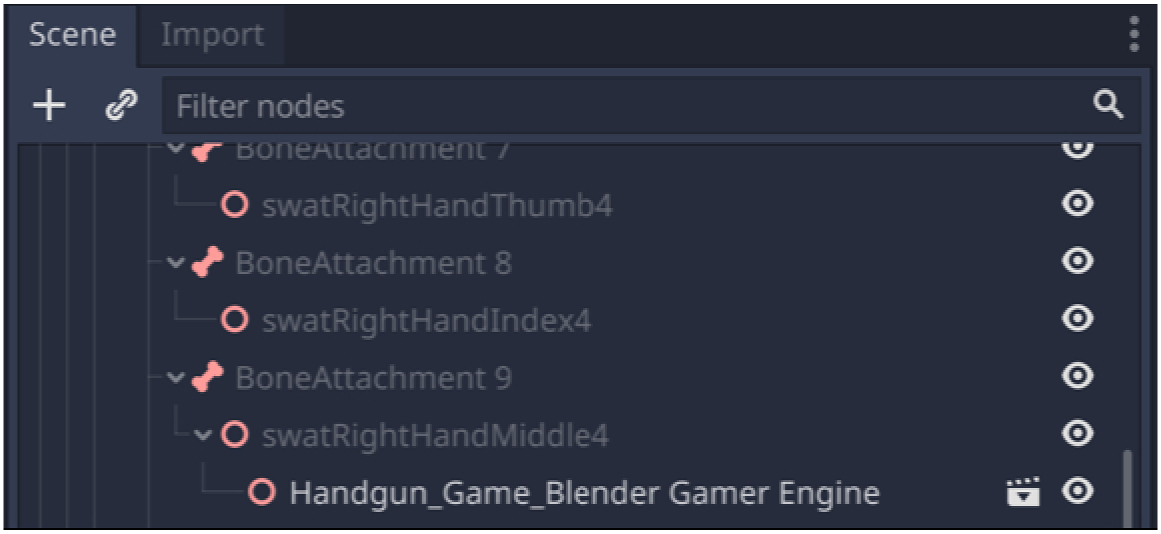
- Please open the scene **NPC.tscn**.
- Right-click on the **NPC** node and select the option **Editable Children**.
- Please locate the asset called **Handgun_Game_Blender Gamer Engine.fbx** in the **File System**.
- Using the **Scene Tree**, please search the node called **swatRightHandMiddle4**.



- Right-click on it.
- From the contextual menu select **Instance Child Scene**.
- In the new window, select the scene **Handgun_Game_Blender Gamer Engine.fbx** and click on **Open**.

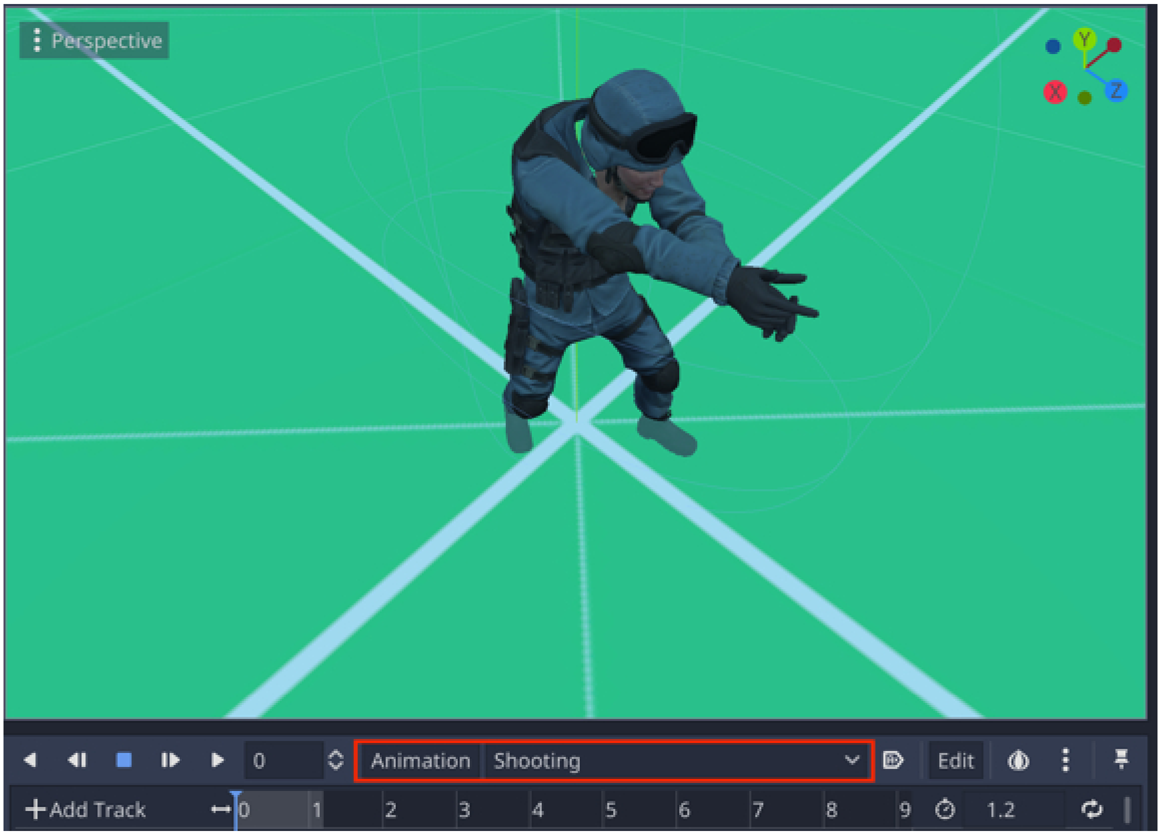


- This will create a node called **Handgun_Game_Blender Gamer Engine** as a child of the node **NPC**. You will need to clear the search field to be able to display this node in the **Scene Tree**.
- You can rename this node **gun**.

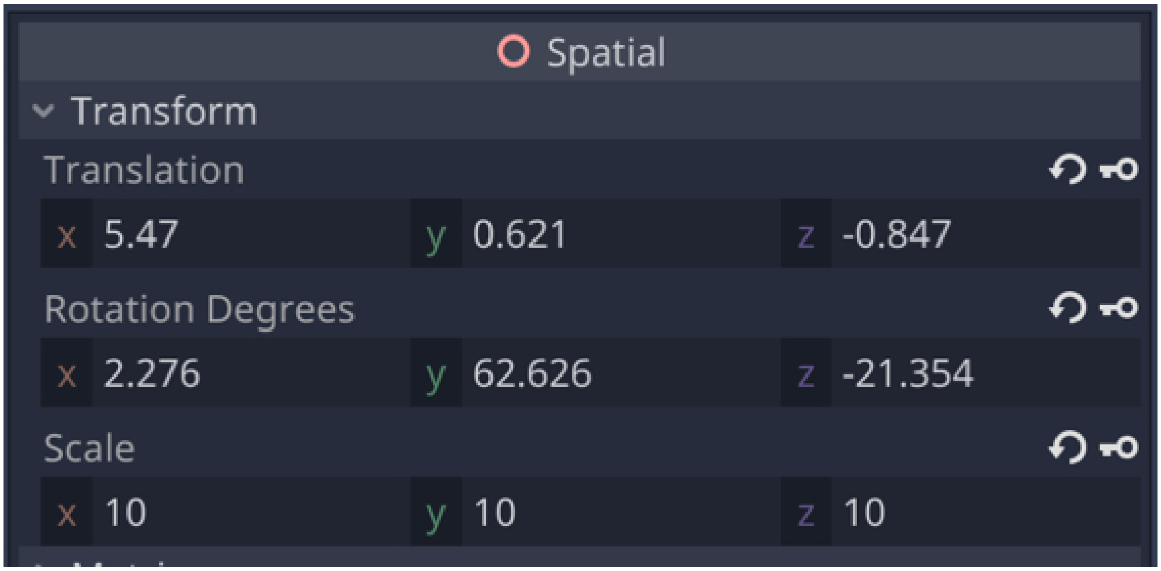


- Please click on the node called **AnimationPlayer** that is a child of the node **NPC**.

- In the bottom window, please select the **Shooting** animation, so that the NPC is in the shooting position



- Next, please select the node **gun** and change the following settings in the **Inspector** window: **Translation** = **(5.47, 0.621, -0.847)**, **Rotation** = **(2.27, 62.6, -21.3)** and **Scale** = **(10, 10, 10)**.



Spatial

Transform

Translation

x 5.47 y 0.621 z -0.847

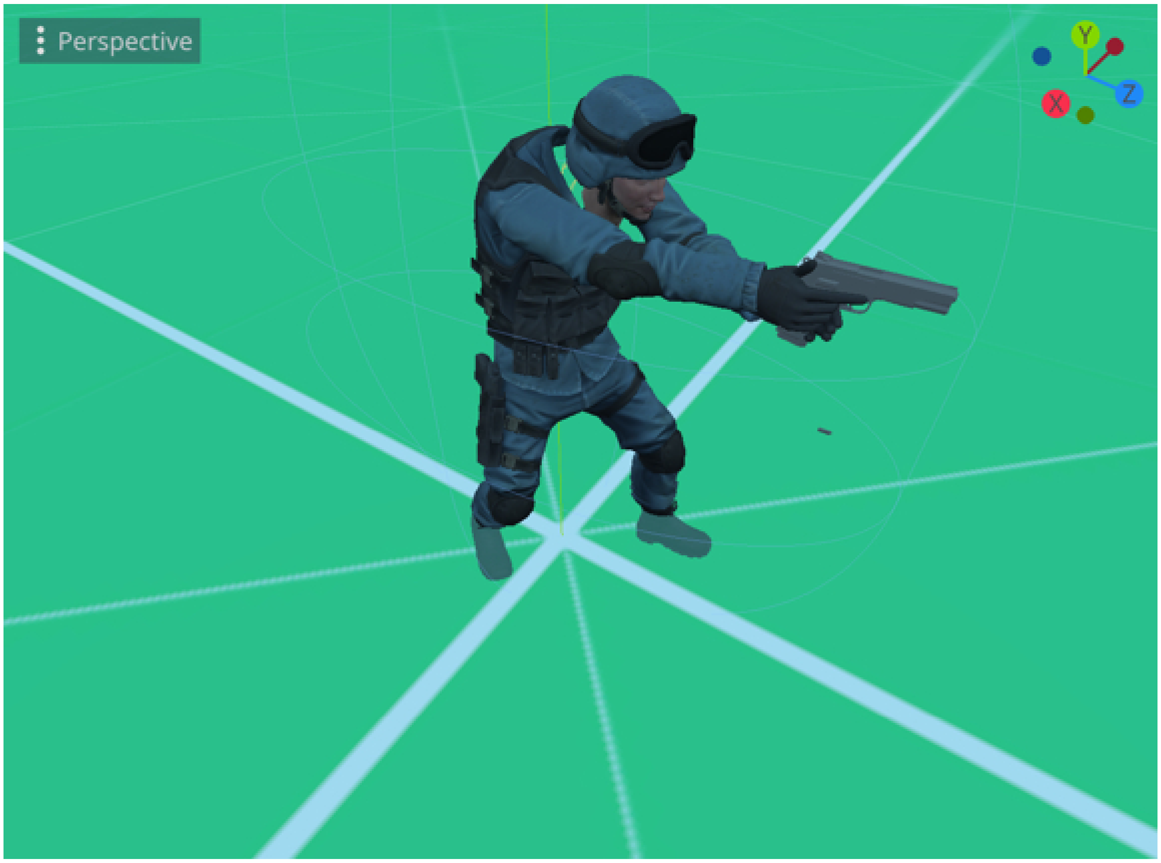
Rotation Degrees

x 2.276 y 62.626 z -21.354

Scale

x 10 y 10 z 10

- After making these changes, the NPC should look like the next figure.



- Please save the scene **NPC.tscn**.

Next you can add ammunition by adding new nodes based on the scenes

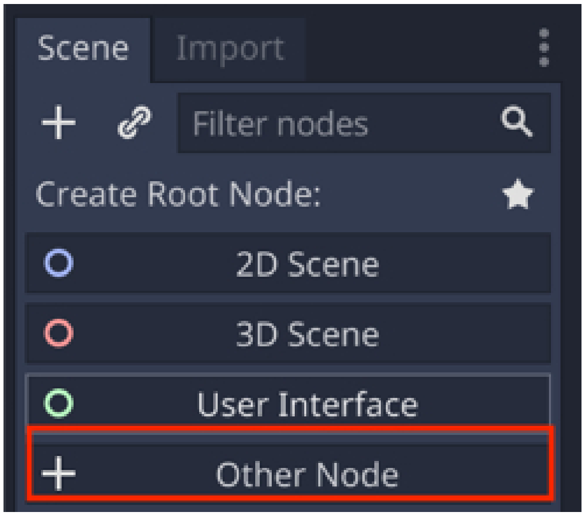
ammo_gun, **ammo_grenade**, or **ammo_auto** and moving them around the maze. To do so, just right-click on the **Spatial** node, select the option **Instance Child Scene**, select the scene that you need to instantiate (i.e., **ammo_gun**, **ammo_grenade**, or **ammo_auto**), and re-scale the new node if necessary, for example by setting its scale to (**0.2**, **0.2**, **0.2**).

So now, we just need to:

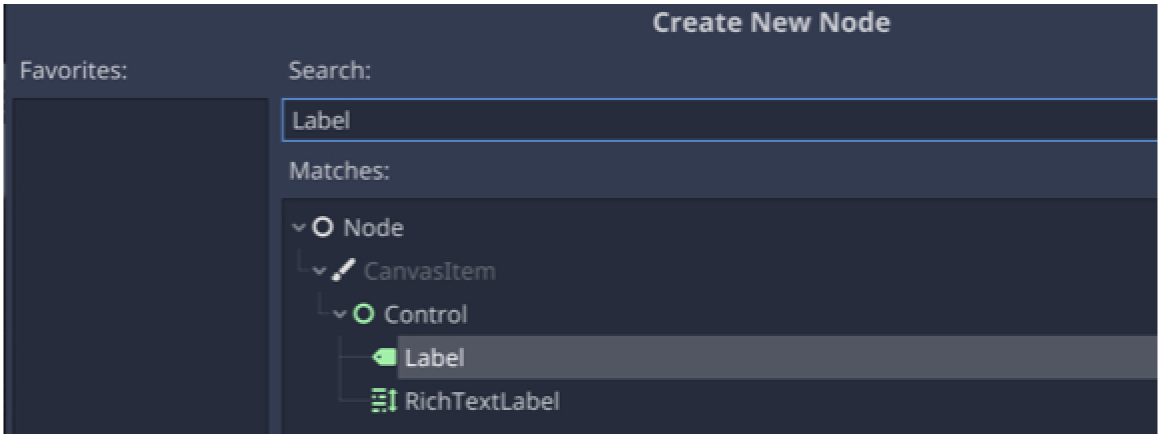
- Add objects to collect.
- Count them as we collect them.
- Load the **Win** scene when the player has collected all of them.
- Load the **Lose** scene when the player has been killed (i.e., **nb_lives= 0**).

First let's create two scenes: one for when the player wins and the other one for when the player loses.

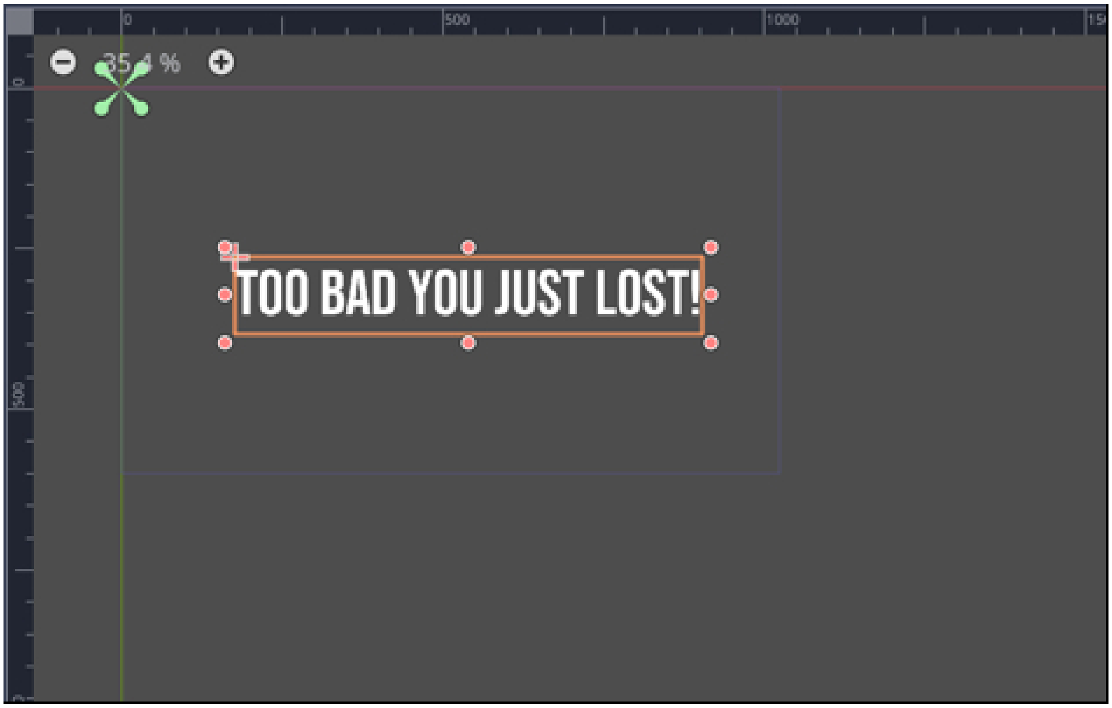
- Please save the current scene.
- Create a new scene.
- In the new window, select the option "**Other Node**".



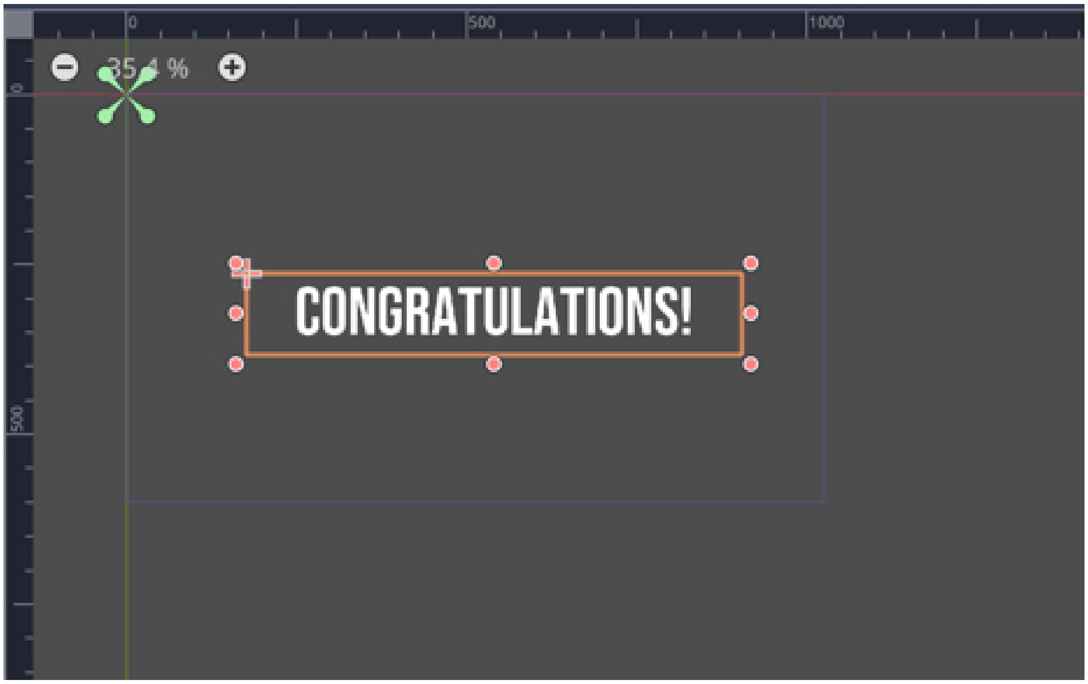
- In the new window, select the node type **Label** and press **Create**.



- Modify the text of the label so that it says “**Too Bad, you’ve just lost**”.
- As we have done before, using the section **Custom Fonts**, create a new **Dynamic Font**.
- Adjust the size and position of this object so that it is displayed onscreen.



- Save the scene as **lost.tscn**.
- Copy this scene (using the **File System** window), rename the duplicate **win.tscn**, and change the **Label** node so that it displays **“Congratulations”**.



- Once this is done, save the scene and go back to the main scene **level_2**.

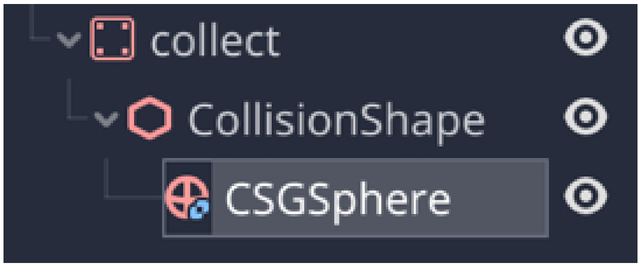
Once this is done, we just need to gain access to these scenes from our scripts.

- Please open the script **Player.gd**.
- Modify the function **got_hit** as follows (new code in bold).

```
func got_hit():
health -= 10
print("Player's Health"+str(health))
screen_flash.start_fade()
if (health <= 0):
restart_level()
else: get_tree().change_scene("res://lose_scene.tscn")
update_player_info()
```

Next, we will work on adding objects to collect and count them.

- Please create a sphere that will need to be collected by the player: create a **StaticBody** node as a child of the **Spatial** node, and rename this node **collect**, add a **CollisionShape** node to this node, set the **collision shape** to a **SphereShape**, and add a **CSGSphere** node to that **CollisionShape** node.



- Add a color to this sphere if you wish.
- Add the group "**collect**" to the node **collect** using the **Inspector**.
- Make sure that the **y** coordinate of the node **collect** is **2** so that the sphere can be above the ground.
- Duplicate the node "**collect" four** times and move the duplicates in different locations (i.e., far apart from each other)

We will now modify the script **Player.gd**

- Please open this script.
- Add the following code (new code in bold).

```
if (collision.collider.is_in_group("collect")):
score += 1
print("score"+str(score))
collision.collider.queue_free()
update_player_info()
if (score >= 4):
get_tree().change_scene("res://win.tscn")
```

- Modify the following code (new code in bold):

```
func update_player_info():
var message = "Lives:"+str(nb_lives);
message += "\nHealth: "+str(health)


message += "\nScore: "+str(score)
player_info.set_text(message)
```

In the previous code, we just ensure that the score is displayed on screen.

Quiz

Now, let’s check your knowledge! Please answer the following questions (the answers are included one the next page).

1. A **MeshInstance** node can be used when adding an obstacle that the NPC needs to avoid while navigating.
2. The option to “**Create Trimesh Static Body**” can be used to add colliders to a**MesInstance** node.
3. It is possible to bake a scene after selecting the **NavigationMesh** node.
4. This code will calculate the distance between the current node and the player:

```
var dist = (player.global_transform.origin -global_transform.origin).length()
```

1. The function **_process** is called every frame.
2. **Enums** can be used to list different states that belong to a state machine.
3. The following code will pause the game for **1** second

```
yield(get_tree().create_timer(1.0), "timeout")
```

1. The following code can be used to wait until a given animation has been played fully.

```
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
```

1. The following code will rotate the current node.

```
rotate(Vector3.UP, 3.18)
```

1. Provided that a Raycast node called ray has been properly configured, the following code will be called if it collides with an object

```
func _physics_process(delta):
if ray.is_colliding():
var obj = ray.get_collider()
```

Answers to the Quiz

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

Chapter 5: Frequently Asked Questions

This chapter provides answers to the most frequently asked questions about the features that we have covered in this book.

GDSCRIPT

How do I create a script?

Right-click on the node and select **Add Script**.

How can my script be executed?

Once the script is linked to a node, it will be executed at run-time.

How can I check that my script has no errors?

Open the **Output** window and any error should be displayed here.

What is object-oriented programming?

In object-oriented programming, your program is seen as a collection of objects that interact with each other using, for example, methods/functions.

What is the dot notation for?

The dot notation refers to **object-oriented programming**. Using dots, you can access properties and functions (or methods) related to a particular object. For example **timer.wait_time** gives you access to the **position** from the **wait time** attribute for the **timer** node. It is often useful to read it backward; in this case, the dot can be interpreted as **"of"**. So in our case, **timer.wait_time** can be translated as "the attribute **wait_time** of the node **timer**".

RIGID BODIES

What are rigid bodies?

Rigid bodies are components that make it possible for an object to be subject to the laws of physics, including gravity.

How can I add a rigid body?

You can create a **RigidBody** node and add it to the scene.

How can I add force to an object that includes a rigid body?

You can modify its velocity; for example:

```
new_ball.linear_velocity = transform.basis.xform(Vector3.FORWARD)*(20)
```

USING SCENES AS TEMPLATES

What is a template?

A template is usually based on a node and can be reused (and updated) indefinitely.

How can I create a template?

In Godot, you can save a node as a scene. This scene can then be re-used later-on. To save the node as a scene, right-click on the node and select **Save Branch as a Scene**.

How can I add a template/scene to an existing scene?

You can either drag and drop the scene (template) from the **File System** window to the **Scene Tree**.

Can I use templates across scenes?

Yes, since the template (i.e., a scene) scene is saved in your project, it can be accessed from any scene within this project.

FINITE STATE MACHINES

How can I create an FSM in Godot?

You will need to create states and a match structure that ensures that the NPC is in only one state at any given time.

How do I link my animations and the Finite State Machine?

If your animations were imported as a **.gltf** file, then all the animations will be included in a node of type **AnimationPlayer**. You can then access any of the animations within using a code similar to the following:

```
get_node("AnimationPlayer").play("IDLE")
```

How can I ensure that an animation will loop?

- Similar to the previous example, you can access the animation and set its looping attribute to true, as follows:

```
get_node("AnimationPlayer").get_animation("IDLE").loop = true
```

How can I wait for an animation to be completed before any further code is executed?

- After playing the animation, you can use some code similar to the following snippet, so that the code pauses until the animation is complete.

```
yield(get_node("NPC/AnimationPlayer"), "animation_finished")
```

NPC NAVIGATION

How does navigation work in Godot?

One of the ways you can implement navigation for NPCs is to create a **Navigation** mode, a **NavigationMeshInstance** node, along with additional **Mesh** nodes. Following this, you can use the function **get_simple_path** to calculate the path between two points (i.e., the start and the destination).

Why do I need to bake the NavigationMeshInstance node before the NPC can move?

By baking the **NavigationMeshInstance** node, Godot determines the area that is accessible to the NPC as part of its navigation; this is based on the different obstacles present in the scene that the NPC will need to avoid to reach its destination.

USER INTERACTION

How can I detect keystrokes?

You can detect keystrokes by using the function **Input.is_key_pressed**. For example, the following code detects when the key **E** is pressed; this code should be added to the **_input** function.

```
func _input(event):
if (Input.is_key_pressed(KEY_P)) :
```

How can I play a sound?

To play a sound, you need to create an **AudioStream** node; when this is done, you can either play its default audio clip, or select which audio clip should be played.

```
export (AudioStream) var sound_gun :AudioStream
...
onready var  sound_fx:AudioStreamPlayer = get_node("sound_fx")
...
sound_fx.stream = sound_gun
sound_fx.play()
```

How can I display text onscreen?

To display text onscreen you will need to create a **Label** node, and then access it through a script. For example:

```
onready var user_message:Label = get_node("../message")
...
user_message.set_text(“Hello”)
```

Chapter 6: [Bonus Chapter] Creating and Exporting Multiple Animations with Mixamo and Blender

In this section, we will go through some very simple steps to create the character that you will be using in your games, including:

- Using Mixamo, a free online software, to select a character for your game.
- Animating your character in Mixamo.
- Exporting the different animations for your character.
- Opening these animations in Blender, another free software, to combine them into one file so that they can be used in Godot.

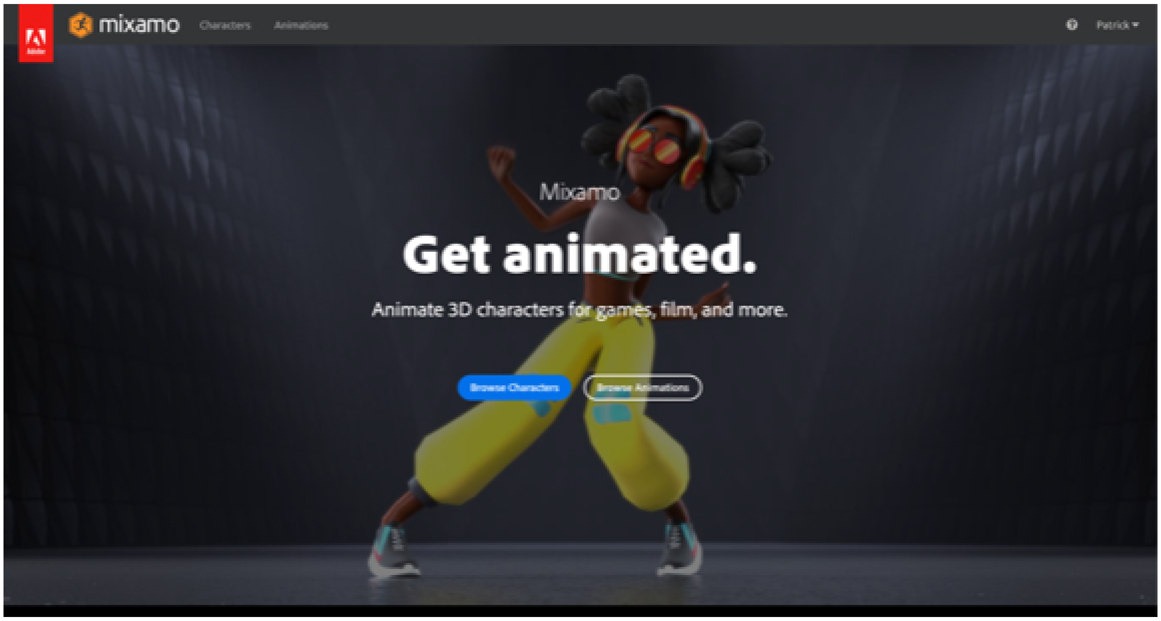
So, after completing this chapter, you will be able to:

- Select your own 3D character.
- Select and apply different types of animations to your character.
- Export your character and the associated animations.
- Combine these animations and generate a file that can be used in Godot.

CREATING YOUR CHARACTER

For this part of the book, we will have a look at Mixamo which is an online software that makes it possible to create and animate 3D characters. The idea, when using this software is to create and animate your characters, and then to import them so that they can be used in your game.

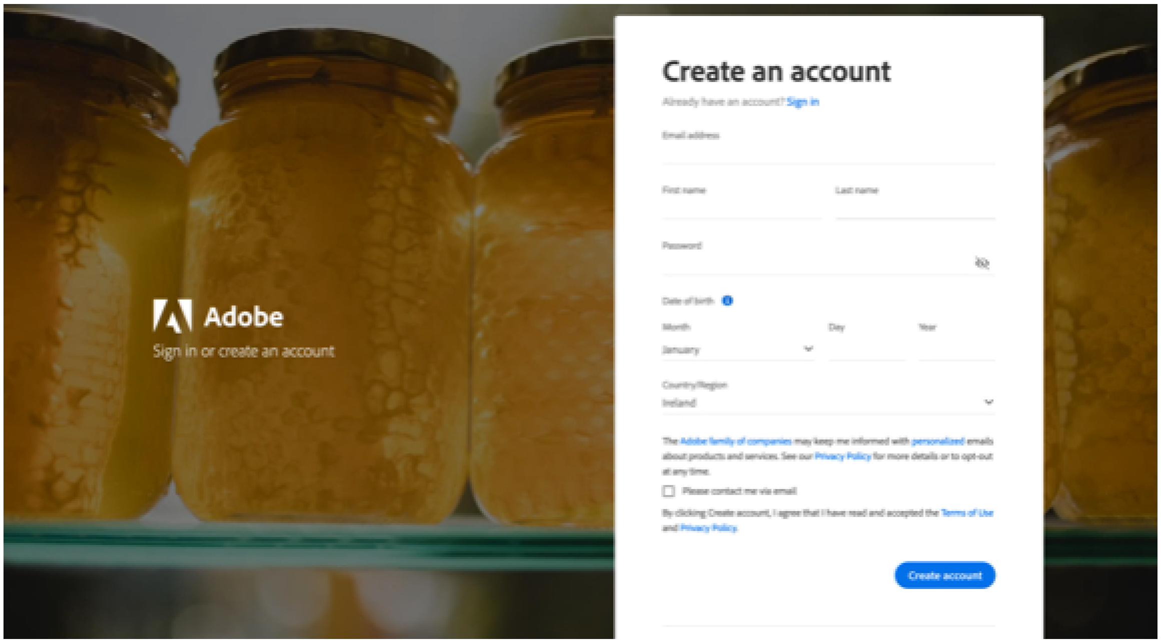
- Please open the following page in your browser: **http://mixamo.com**.
- Once the page is open, you can click on the button labelled **Login**, if you already have an account for this site.
- You can also click on the button **Sign-up for FREE**, to create an account otherwise.


Mixamo

Get animated.

Animate 3D characters for games, film, and more.

- If you chose to create an account, the following window should appear:



Adobe

Sign in or create an account

Create an account

Already have an account? Sign in

Email address

First name

Last name

Password

Date of birth

Month: January

Day

Year

Country/Region: Ireland

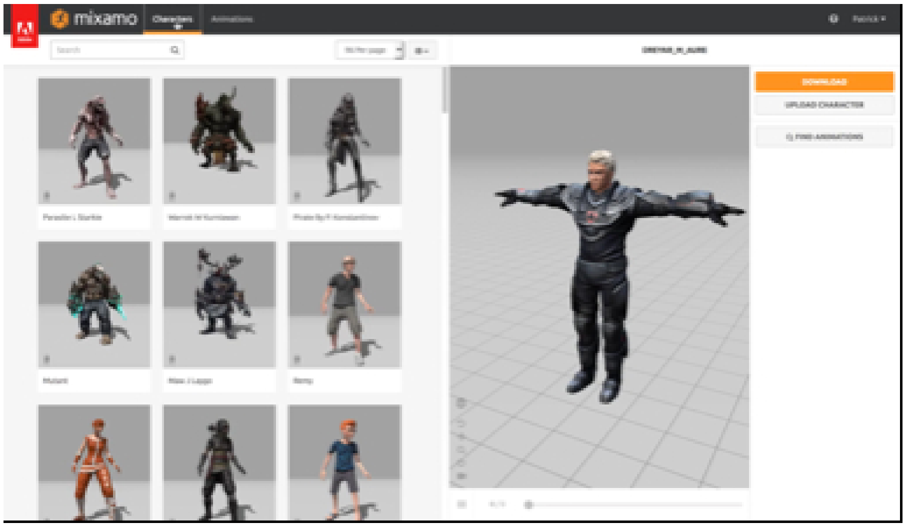
The Adobe family of companies may keep me informed with personalized emails about products and services. See our Privacy Policy for more details or to opt-out at any time.

□ Please contact me via email

By clicking Create account, I agree that I have read and accepted the Terms of Use and Privacy Policy.

Create account

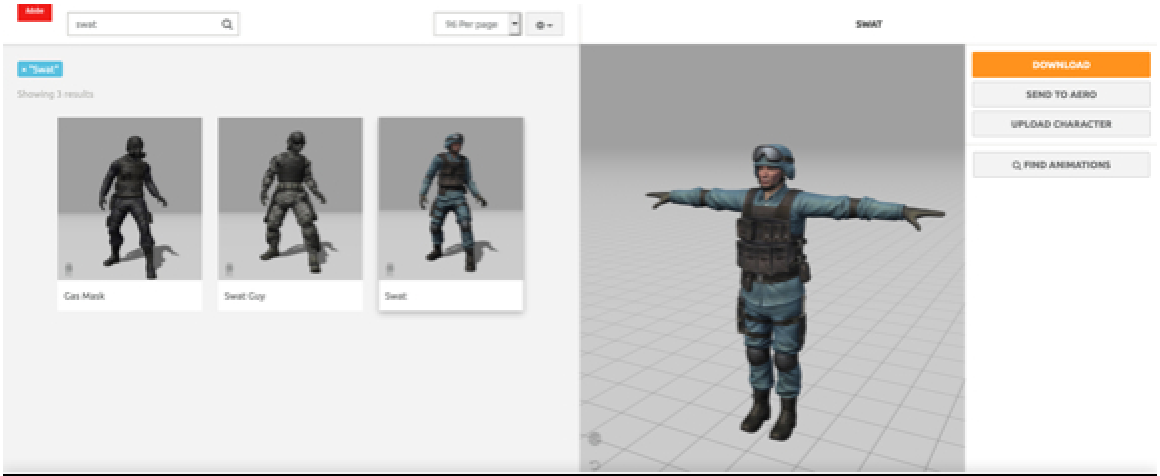
- After signing-up, and completing all the necessary steps, you should now be able to login.
- As soon as you are logged-in, you will have the choice to select your own character and animation.



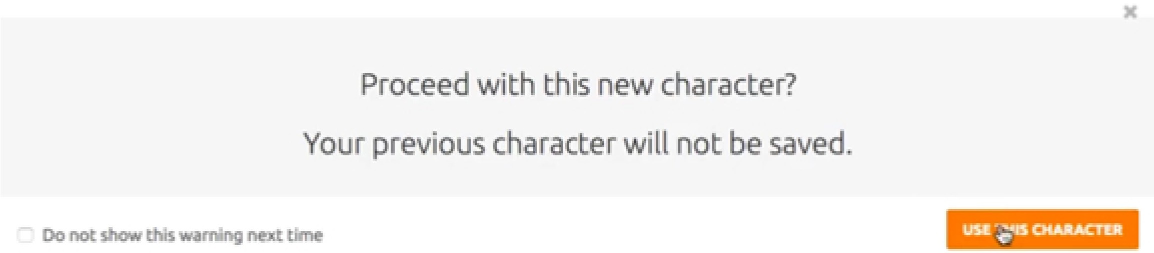
- To start with, we will select a new character; to do so, please press the tab called **Character** located in the top-left part of the window.
- Once this is done, a list of characters that you can use for your games will be displayed; these characters are already textured and rigged which means that

they include a skeleton that makes it possible for them to be animated later.

For this example, but again you can choose any character if you wish, we will select the character **Swat**, by clicking on the corresponding image on the left hand side of the screen, as per the next figure.



- If a window asking you to "**go ahead although the previous character will not be saved**" appears, please press the button labelled "**Use this character**", as per the next figure



Proceed with this new character?

Your previous character will not be saved.

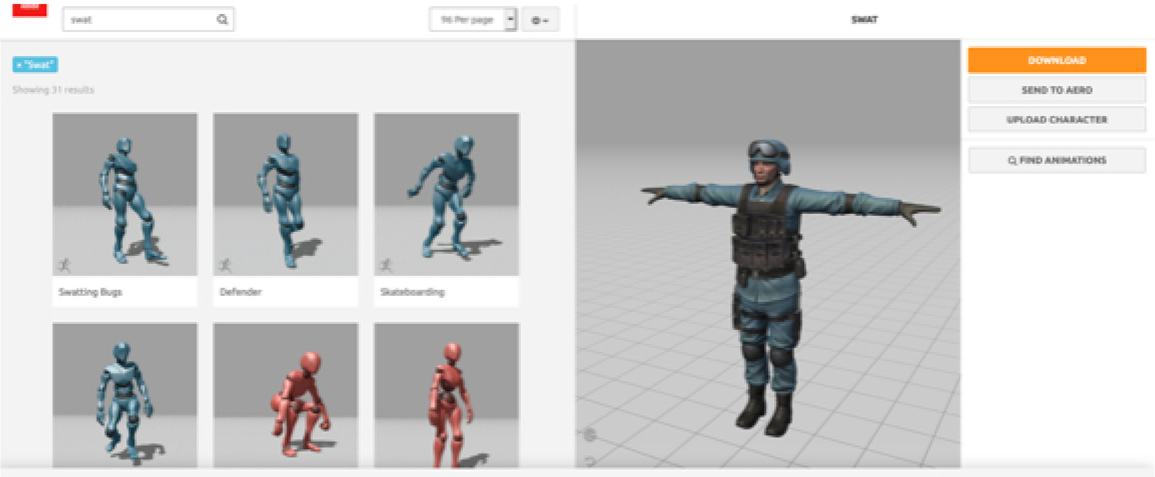
☐ Do not show this warning next time

Once this is done, you will be able to see the character that you have selected in the right-hand side of the screen, in what is often called a **T pose** (often called the reference pose), as per the next figure, which means that the character is not yet animated.

ANIMATING YOUR CHARACTER

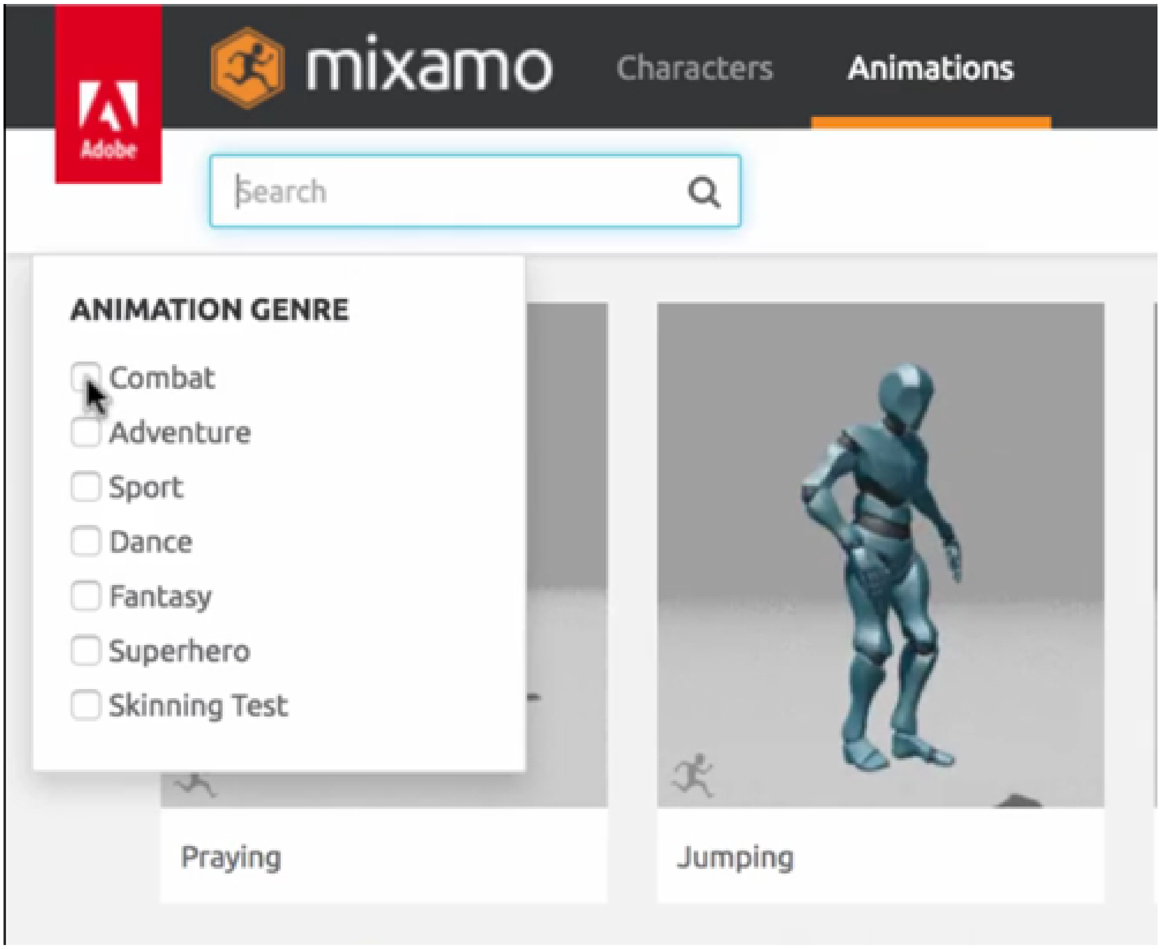
So at this stage you have selected a character, and the next step is to be able to add some animations to it.

- Please click on the tab called **Animation**, located in the top-left corner of the window.
- You should now see a list of animations made available to you by Mixamo.



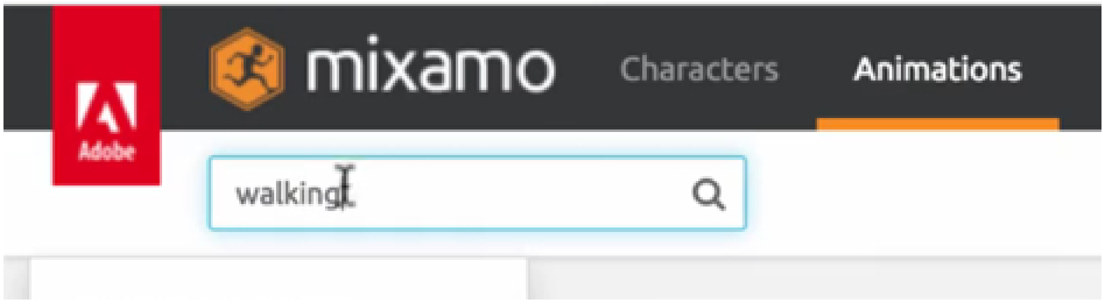
Note that the character that you see on the animation is not the final animated character, but just a representation of how the animation that you are about to apply to your character would look like.

To look for specific types of animation, you can use the menu located in the top-left corner of the window, as per the next figure.

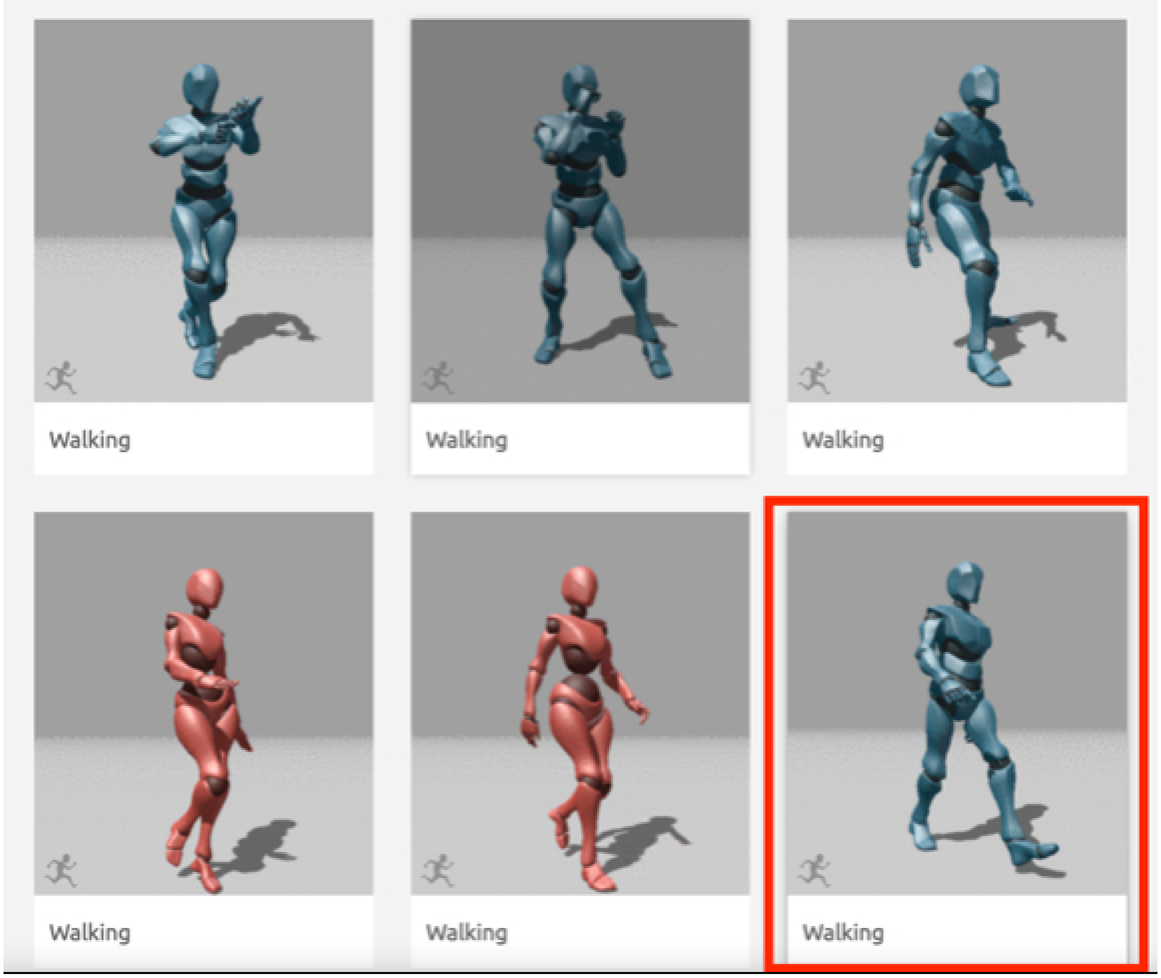


This menu makes it possible to filter through the animations by genre (e.g., Combat, Adventure, Sport, etc.)

You can also, if need be, perform a search, by using the search field available in the top left corner of the window. In our case, because we will be looking for a walking animation, we can type the text **walking** in the search field.



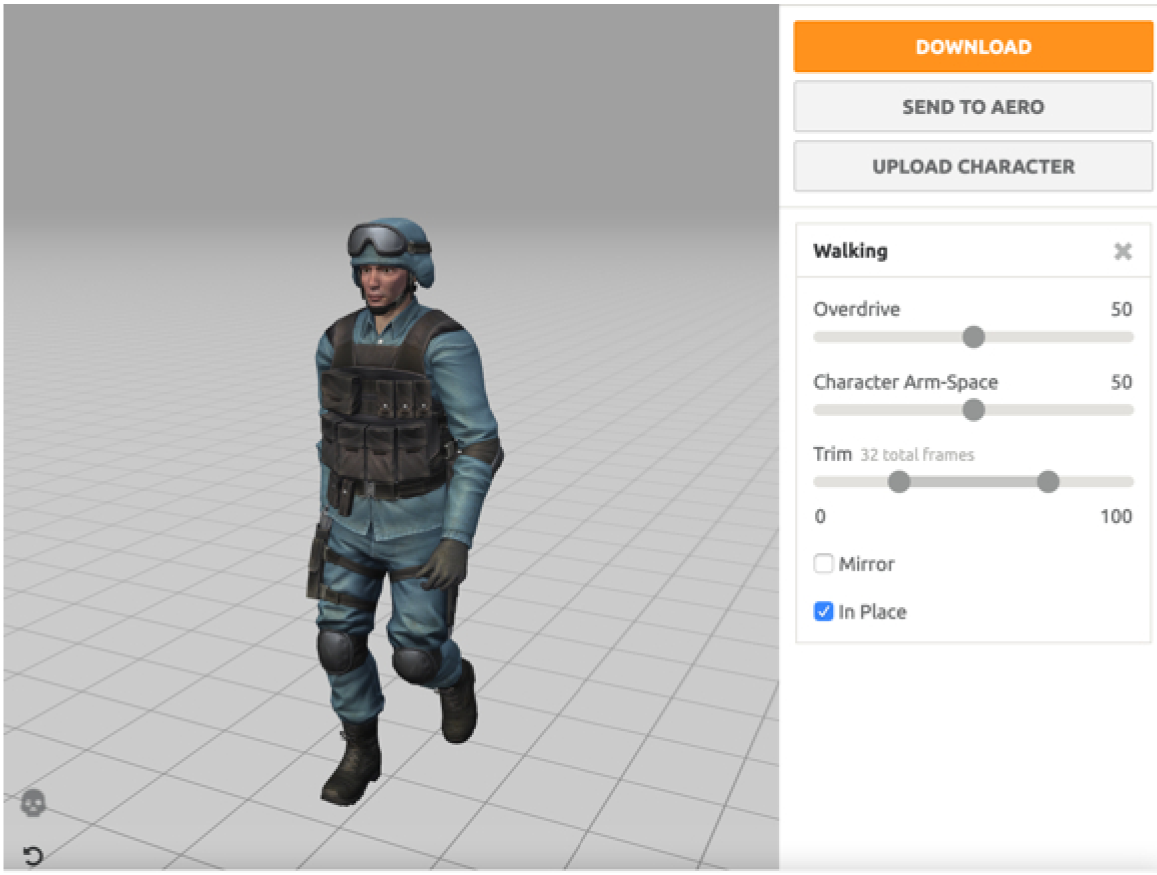
After pressing **Enter** on your keyboard, Maximo will display a list of animations for which the name includes the word **walking**.



In our case, we will select (i.e., click on) the third animation on the second row called **Walking**; this being said, you can choose any animation.

After a few seconds, you should see that you character located in the right part of the window is animated, using the animation that you have just picked as per the next figure. If you can't see any animation, please refresh the page.

As you will see, the character is animated; however, it doesn't stay on the spot, and it is difficult to see the full animation. To remedy this problem, you can tick the box for the option called "**In place**", located in the right part of the window.



Walking

Overdrive 50

Character Arm-Space 50

Trim 32 total frames

0 100

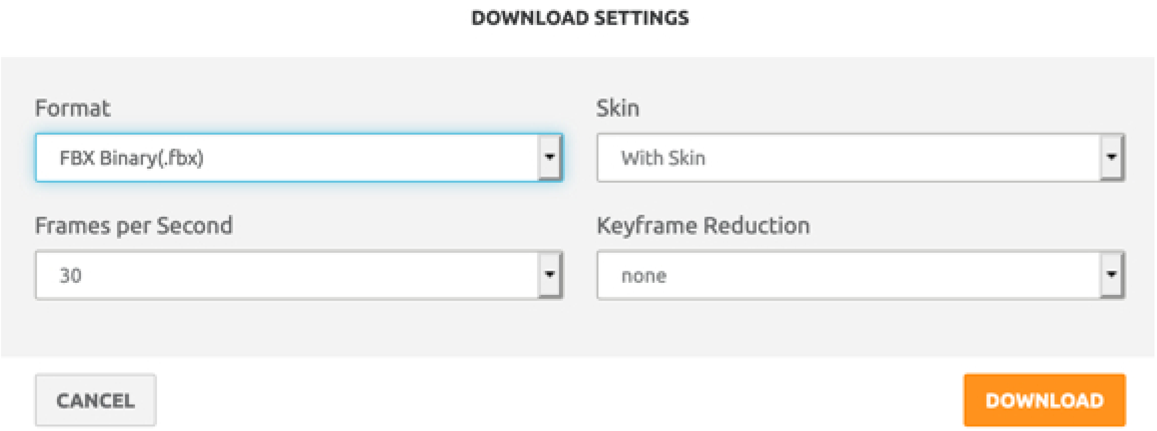
□ Mirror

☑ In Place

In fact, for your game, you only need this animated character to walk on the spot, as the overall movement (i.e., going left, right, forward and back) can be managed through Godot. Also note that you can modify some of the other parameters if you wish (e.g., speed, overdrive, etc.)

Once this animation has been applied, we can now save it.

- Please lick on the button labelled **Download**.
- In the new window, select the format FBX Binary (**.fbx**) and leave all the other options as default, and then click on the button **Download**.



Format: FBX Binary(.fbx)

Skin: With Skin

Frames per Second: 30

Keyframe Reduction: none

- The browser will ask you where you would like to download this file (i.e., **Walking.fbx**).
- Please save the file to a location of your choice.

Now that you have managed to save the walking animation, please repeat the last steps to apply and save an animation for the following:

- Shooting (using the animation called **Shooting**).
- Punching (using one of the animations called **Punching**).
- Dying (using one of the animations called **Punching**).

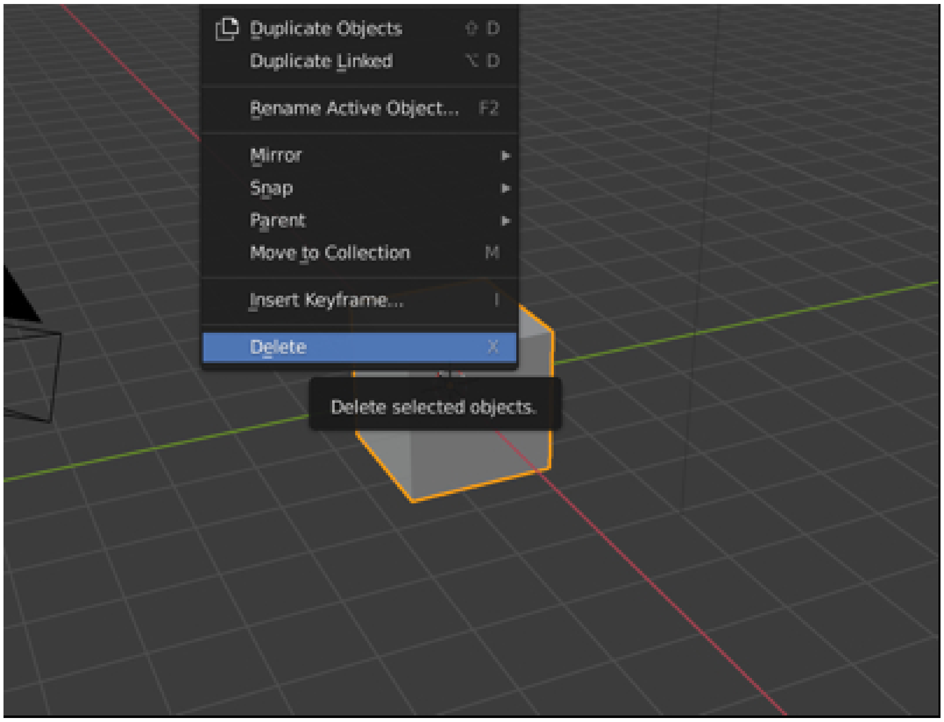
So at this stage you should have four different animations saved on your computer.

IMPORTING THE ANIMATIONS IN BLENDER

So, now, the goal is to import these animations, so that they can be combined into just one file and then exported for Godot.

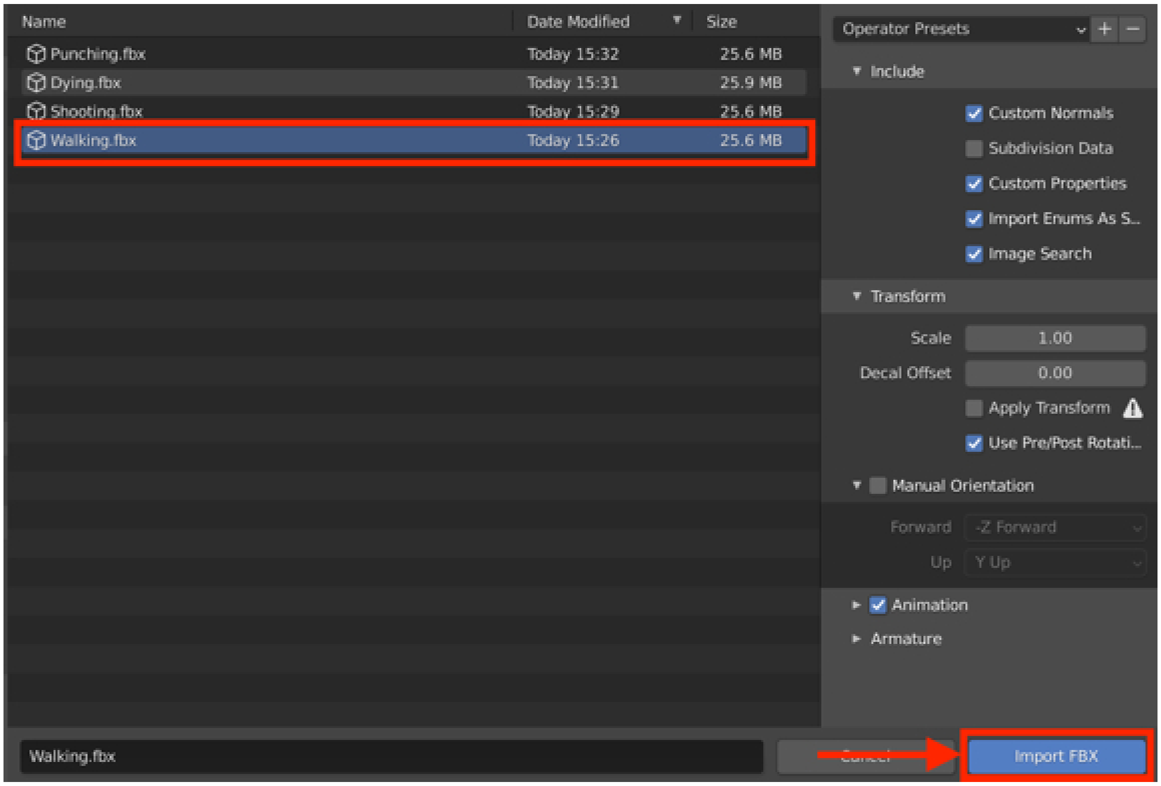
So let’s get started.

- If you don’t have the application called **Blender** already installed, you can find the latest version of this free software using this link: **http://www.blender.org/download**.
- Please install and open Blender.
- By default, when you open Blender there will be a cube already present in the scene, and we can delete it by right-clicking on it and by selecting the option “**Delete**”.



Once this is done, please do the following:

- Select: **File | Import | FBX**.
- Navigate to the folder were you have saved your animations, select the animation called **Walking** and press the button labelled **Import FBX**, as per the next figure.



Name	Date Modified	Size
Punching.fbx	Today 15:32	25.6 MB
Dying.fbx	Today 15:31	25.9 MB
Shooting.fbx	Today 15:29	25.6 MB
Walking.fbx	Today 15:26	25.6 MB

Operator Presets

▾ Include

☑ Custom Normals
☐ Subdivision Data
☑ Custom Properties
☑ Import Enums As S...
☑ Image Search

▾ Transform

Scale 1.00
Decal Offset 0.00
☐ Apply Transform
☑ Use Pre/Post Rotati...

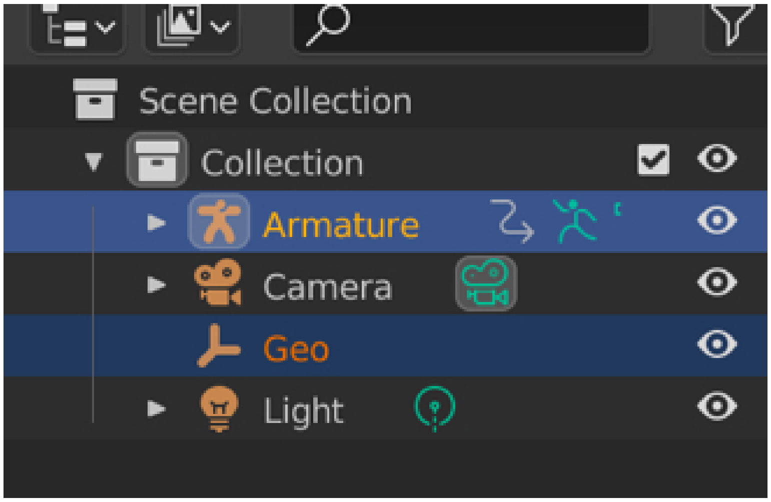
▾ ☐ Manual Orientation

Forward -Z Forward
Up Y Up

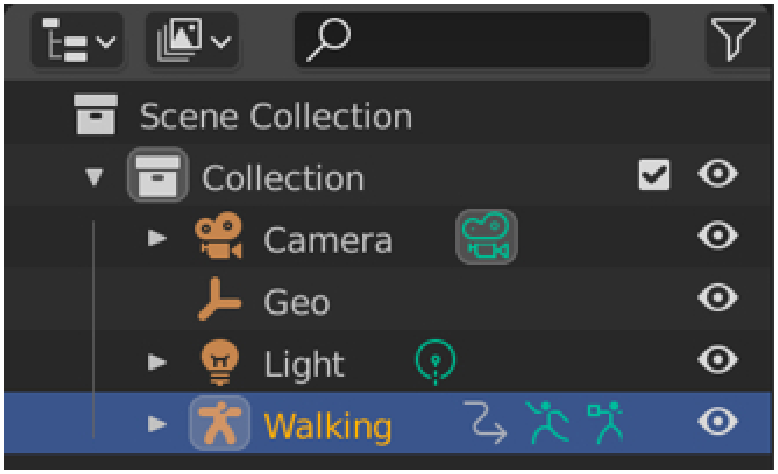
▸ ☑ Animation
▸ Armature

Walking.fbx

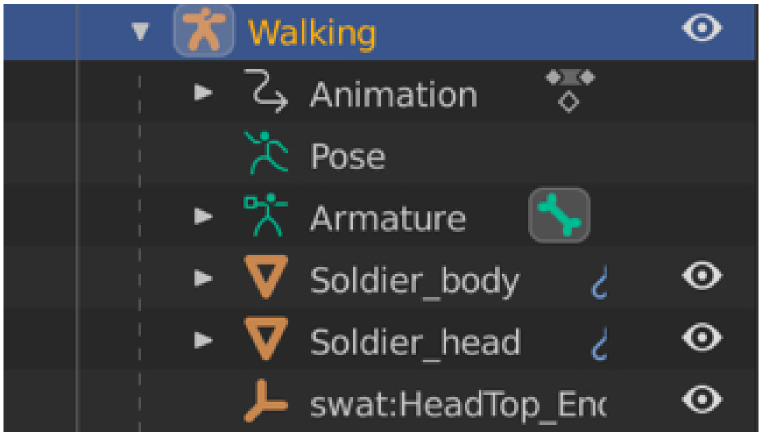
- Once this is done, the window located in the top-right corner will display the objects included in the current file, including the animated character that we have just imported.



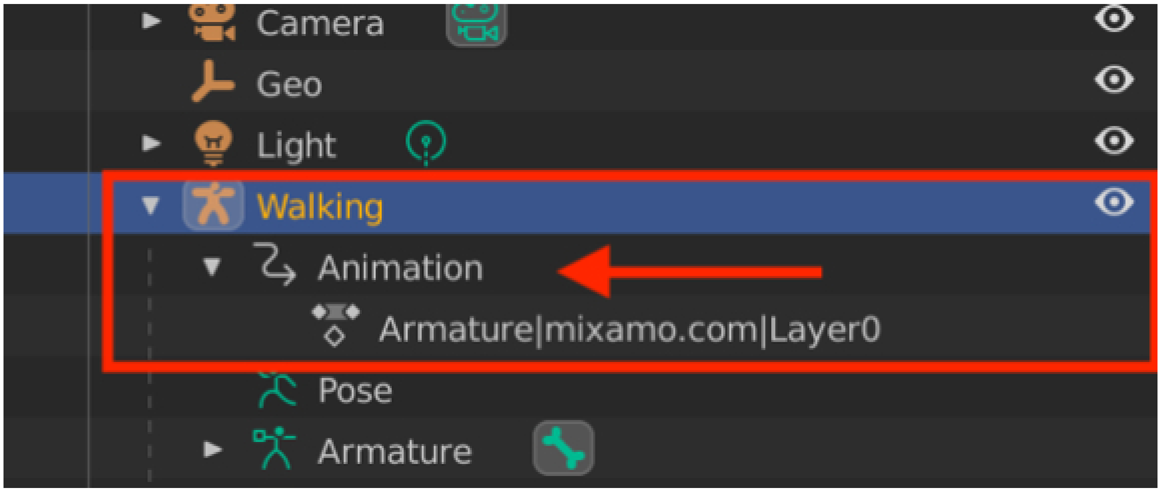
- In that window, please change the name **Armature** to **Walking**: double click on "**Armature**" and change the name.



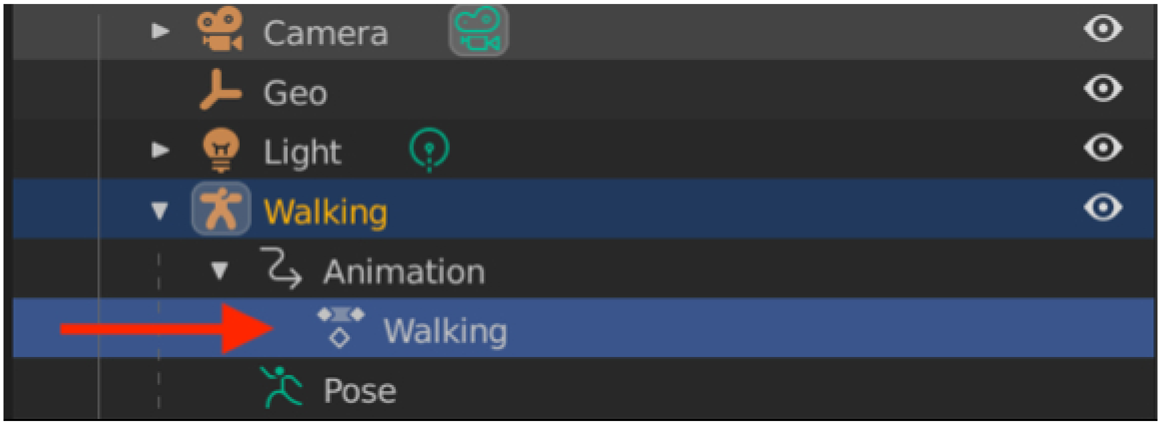
- Once this is done, expand the section that you have renamed **Walking** by clicking on the arrow to the left of the label "**Walking**"; this will show more properties.



- Please expand the section **Animatio**n by clicking on the arrow to the left of the label "**Animation**", this will display another node/property called "**Armature|mixamo.com|Layero**".

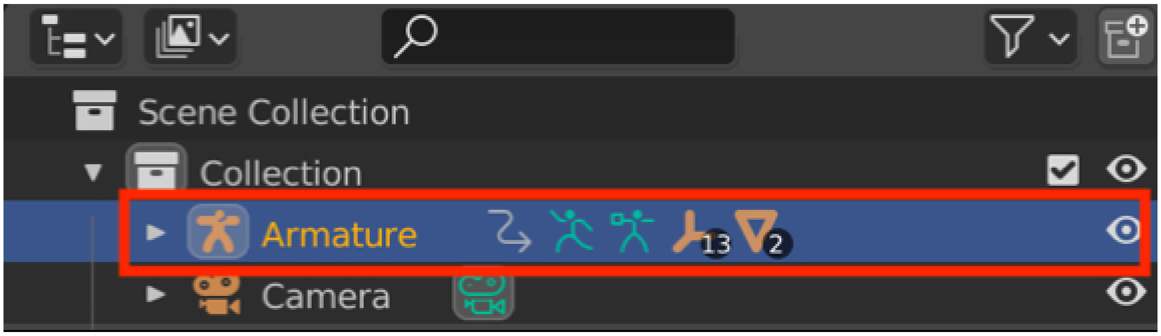


- Please rename the node "**Armature|mixamo.com|Layero**" to **Walking**.

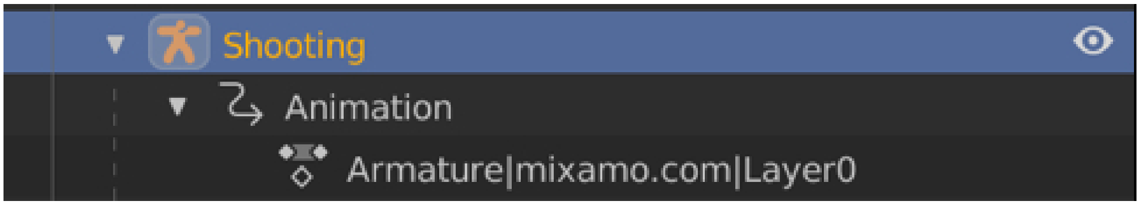


Please repeat the last steps with the file **Shooting**:

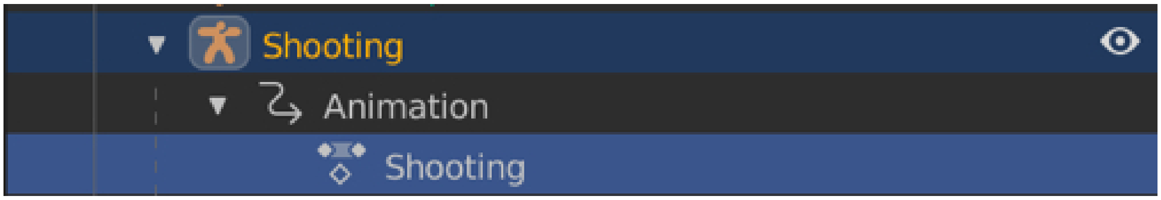
- Select **File | Import**, and select the file **Shooting.FBX** from your file system.
- This will create a new node called **Armature** in the top-right window.



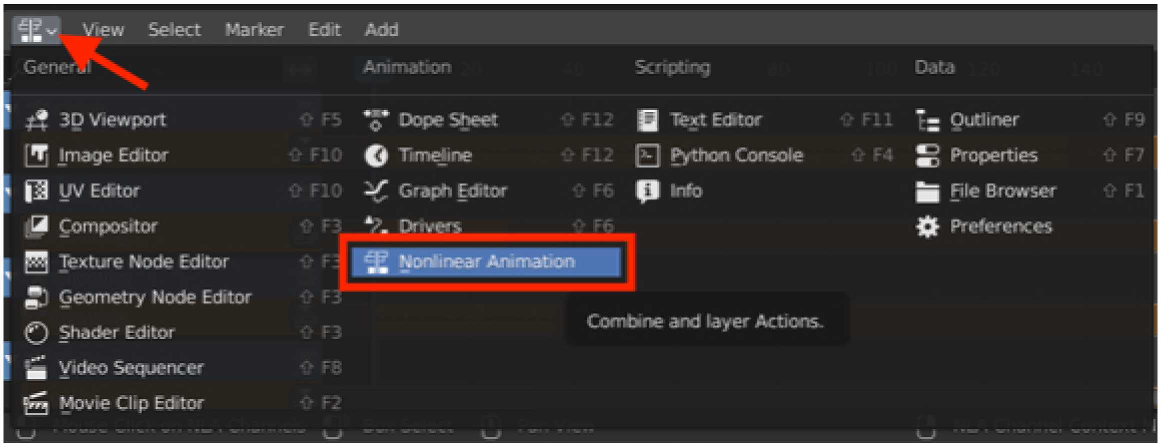
- Change the name of this node to **Shooting**.
- Expand this node, and the **Animation** node within, as per the next figure.



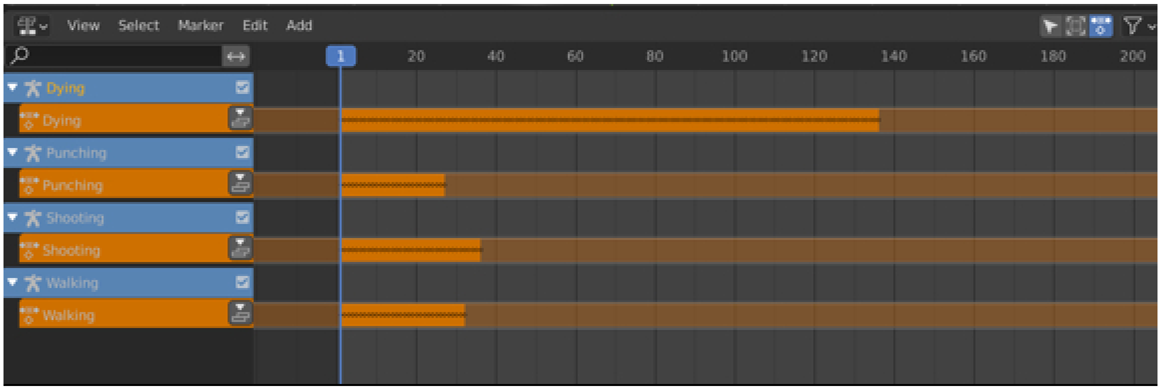
- Change the name of the node “**Armature|mixamo|Layero**” to **Shooting**.



- Please repeat the last steps for the animation **Punching.FBX** and **Dying.FBX**.
- Once this is done, please open the non-linear animation window, as illustrated in the next figure (you can also use the shortcut **CTRL + SHIFT + F12**).

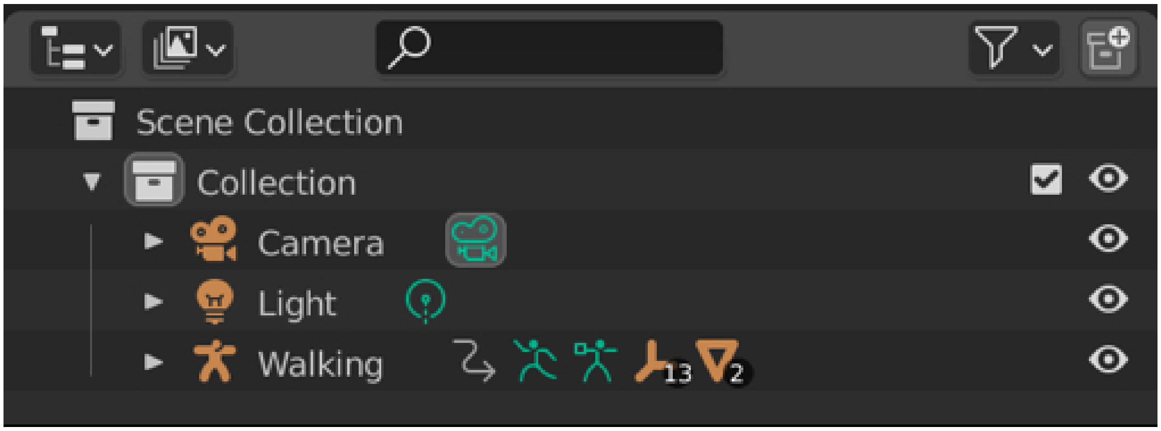


- You should now see all the animations that you have imported as new tracks.

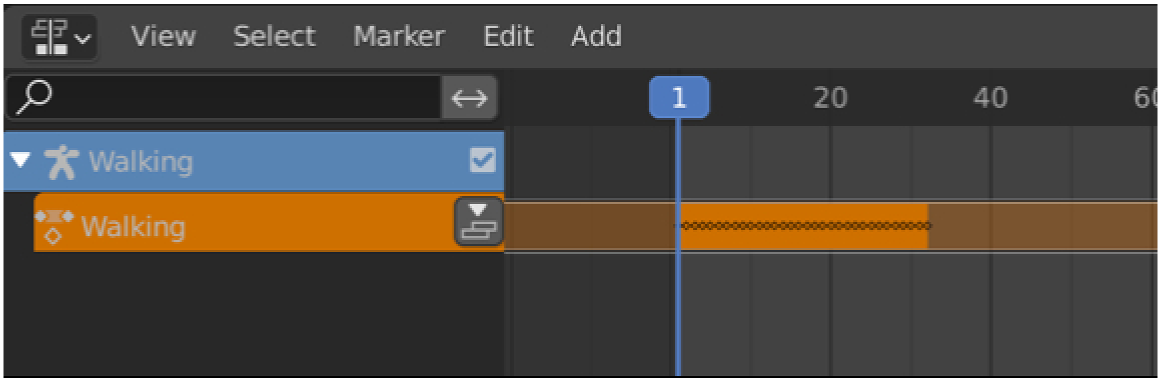


Since we are only interested in the animations, we will delete all the nodes that we don't need:

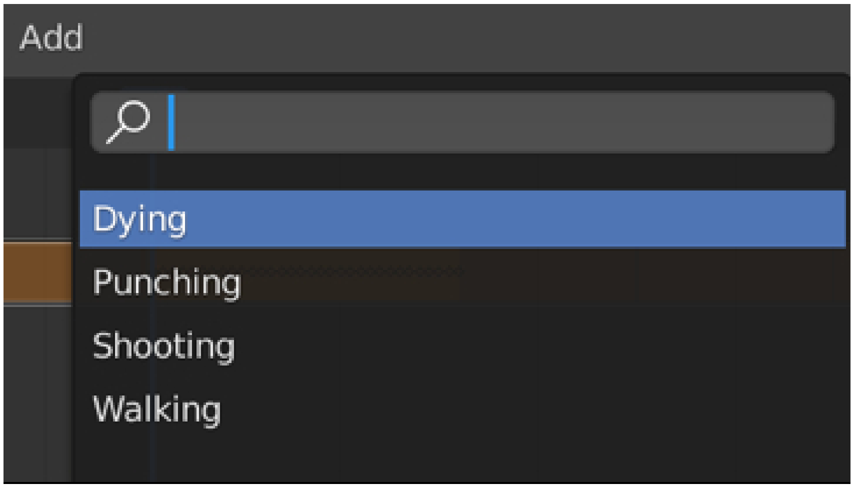
- Using the window in the top-right corner, please delete all the nodes except from **Camera**, **Light** and **Walking**.



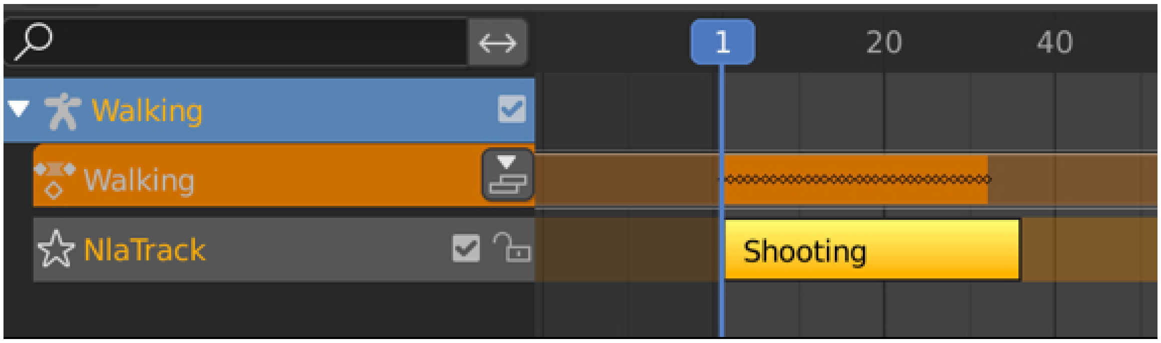
- Now your animation window should include only two strips.



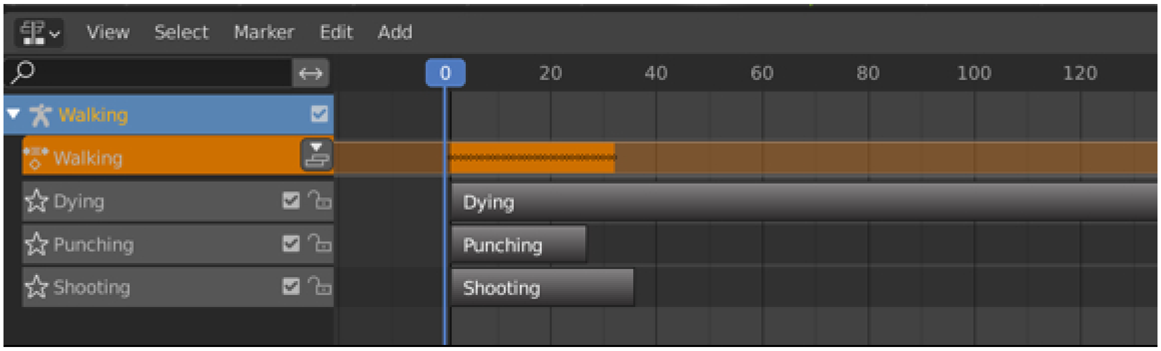
- Please click on the first strip (the blue strip labelled **Walking**).
- Then select **Add | Action Strip** from the menu above the strips.
- In the new window, please select the option **Shooting**.



- This will create a new track.



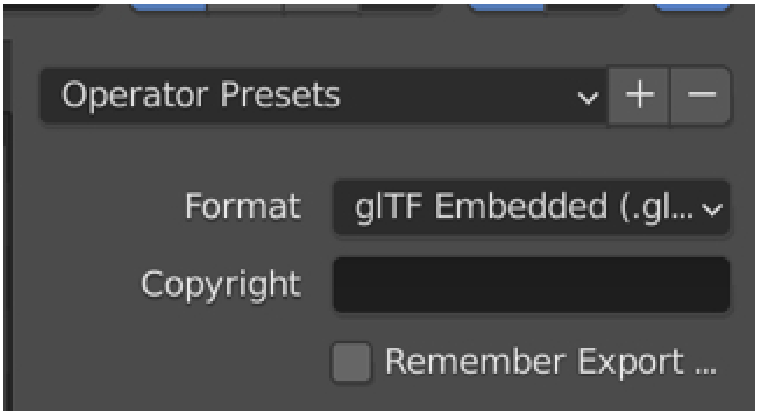
- Please rename this track "**Shooting**".
- Repeat the past steps to add the animations **Punching** and **Dying**.
- At this stage, you should have five layers (or strips) as follows:



At this stage, we are ready to export our animations for Godot:

- Please select: **File | Export | glTF 2.0.**
- Select a folder where you want the new file to be saved.

- Give a name to this file, for example **new_soldier**.
- Make sure that the format is **gltf embedded** using the option located int the top-right corner of the current window, as per the next figure.



Format: glTF Embedded (.gl…

Copyright:

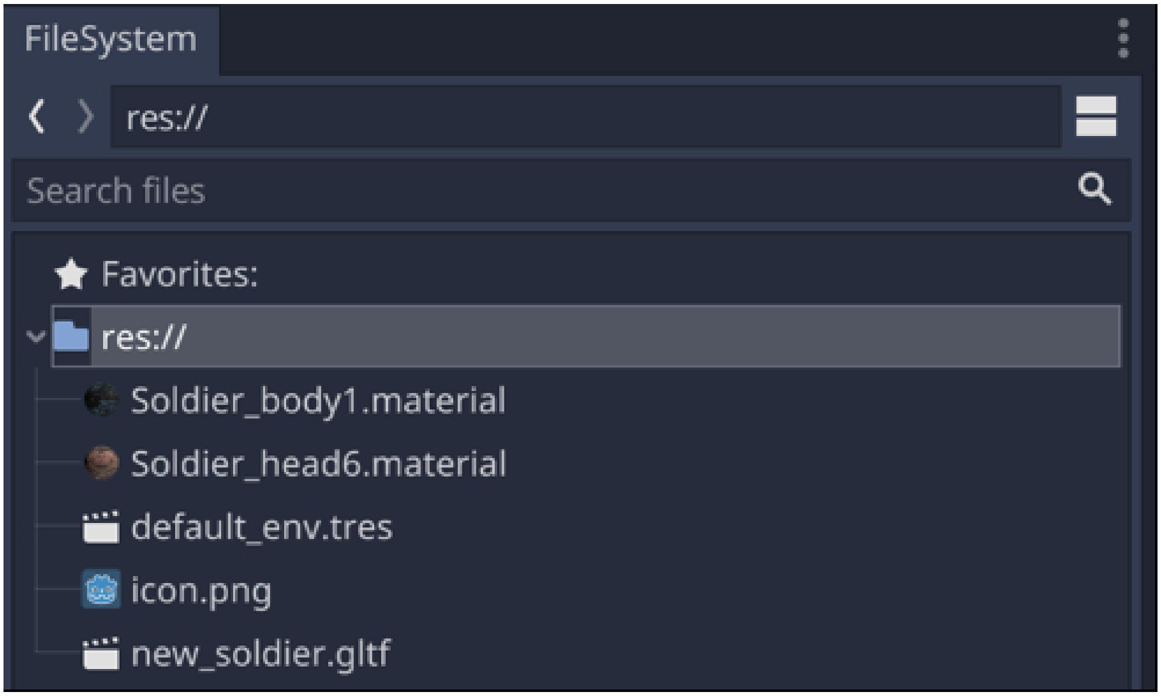
□ Remember Export ...

- Click on "**Export to GLTF2**".

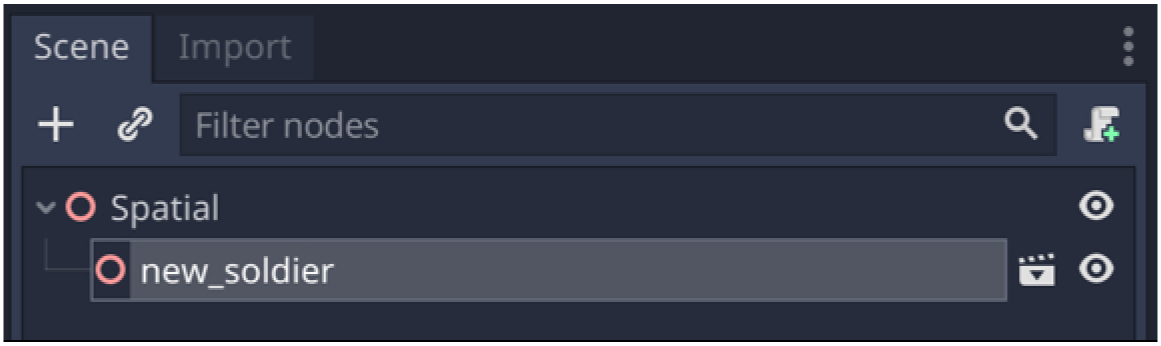
IMPORTING TO FILE IN GODOT

Now that the combined animations have been saved, we can import them in Godot.

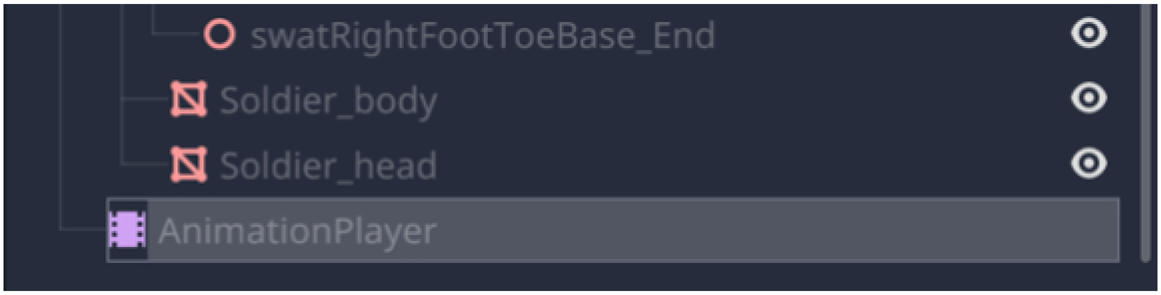
- Please open a new project in Godot.
- Create a new 3D scene.
- Drag and drop the file **new_soldier.gltf** to the **FileSystem** window in Godot.
- This will create a new file called **new_soldier.gltf** in the **File System** window.



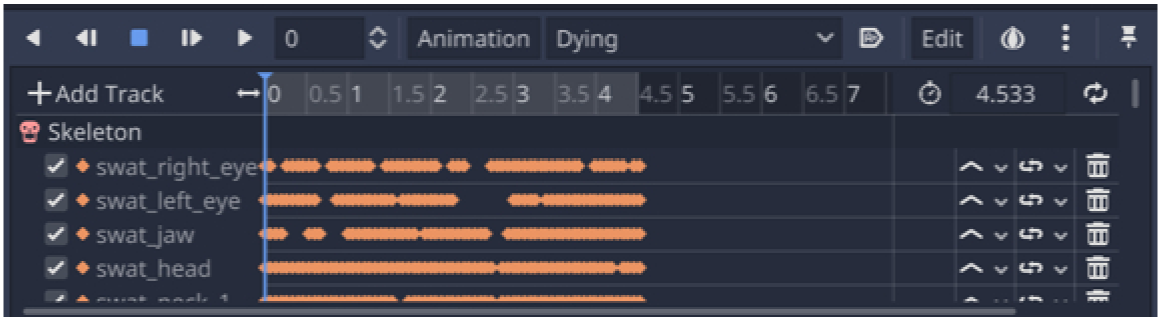
- Drag and drop this file atop (as a child of) the **Spatial** node already present in the scene.



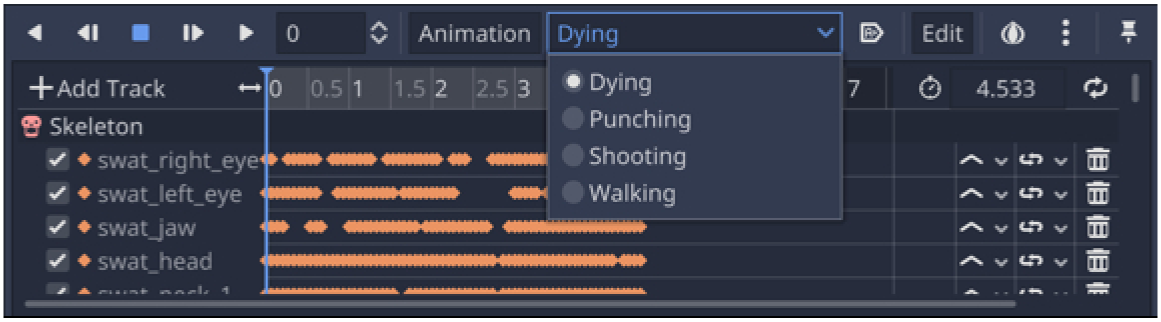
- Right-click on the new node and select the option **Editable Children**.
- Scroll down, locate and select the node called **Animation Player**.



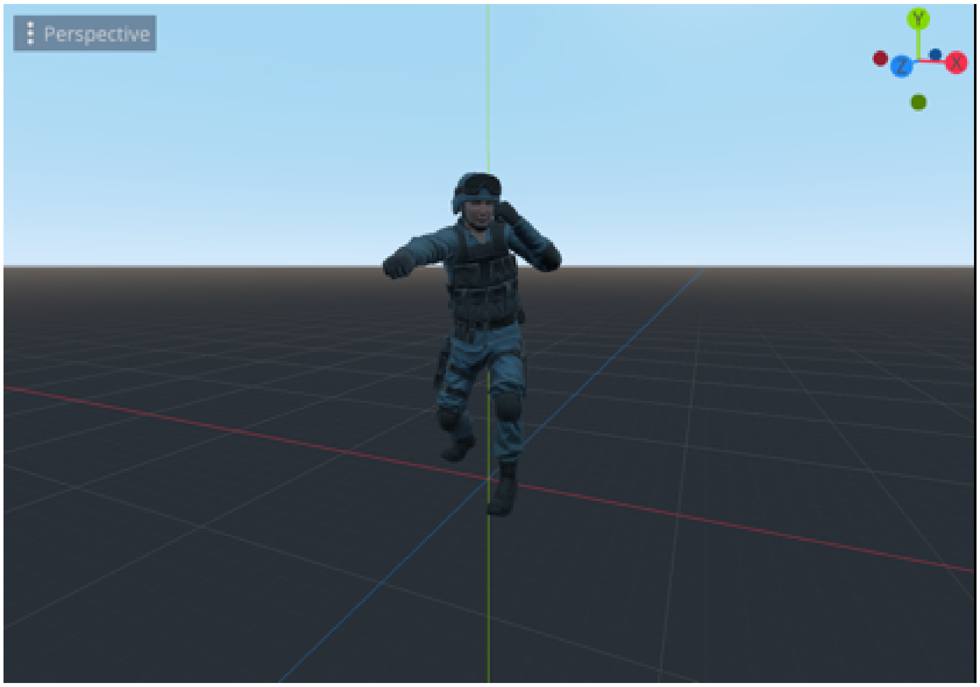
- In the **Animation** window, you should see that the **Dying** animation is selected by default.



- You can play this animation, and/or select other animations, using the drop-down menu.



As you are playing the animation, you should see the main character animated accordingly in the main window.



So your animations are now ready to be used.

Chapter 7: Thank you

I would like to thank you for completing this book; I trust that you are now comfortable with scripting in GDScript and that you can create interactive 3D game environments. This book is the third in a series of four books on Godot, so it may be time to move on to the next book for the advanced level where you will learn more advanced features, including networking, database access, and much more. You can find a description of this book on the official page **http://www.learntocreategames.com/books**.

So that the book can be constantly improved, I would really appreciate your feedback. So, please leave me a helpful review 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!!

[ ]

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.