Community Experience Distilled

Unity 3.x Scripting

Write efficient, reusable scripts to build custom characters, game environments, and control enemy AI in your Unity game

Volodymyr Gerasimov Devon Kraczla

[PACKT] PUBLISHING

Unity 3.x Scripting

Write efficient, reusable scripts to build custom characters, game environments, and control enemy AI in your Unity game

Volodymyr Gerasimov

Devon Kraczla

BIRMINGHAM - MUMBAI

Unity 3.x Scripting

First published: June 2012

Production Reference: 1140612

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-84969-230-4

www.packtpub.com

Cover Image by Karl Moore (karl.moore@ukonline.co.uk)

Credits

Authors
Volodymyr Gerasimov
Devon Kraczla

Reviewers
Peter Chan
Jeff Mundee

Acquisition Editor
Rashmi Phadnis

Lead Technical Editor
Hithesh Uchil

Technical Editor
Devdutt Kulkarni

Project Coordinator
Alka Nayak

Proofreader
Bernadette Watkins

Indexer
Monica Ajmera

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Authors

Volodymyr Gerasimov is a level designer and scripter. His major passion is creating modifications for popular games, and developing small, indie projects, with scripting as a main tool. He learned various scripting and programming languages at The Art Institute of Vancouver. Introduced to Unity in 2010, he created and worked on a number of projects, indie games, and prototypes. He has worked as Lead Level Designer and Scripter, on the hack-and-slash action game, *Splik and Blitz: Baked in Blood*, and has also worked on a couple of indie projects for iOS and PC. His latest, finished project is the puzzle platformer game, *Red Rolling Hood*. Currently, he is working at Best Way, as Producer of an action role-playing game.

I would like to thank all my friends and teachers who shared their experience with me. They surrounded me with an aura of creativity and art, which kept my passion burning, and my work going. I would also like to thank all who will open this book, and be able to learn something, create, and share.

Devon Kraczla is an independent game developer. Having an artistic background, Devon came to the gaming industry to explore new ways to surprise people with his creations. Over the last couple of years, having graduated from The Art Institute of Vancouver, Devon has developed multiple, independent projects, both solo and with other enthusiasts, and has worked on the award-winning *Battlefield 3*, as a member of the motion capture team at EA Canada. In his games, Devon focuses on simple and engaging game mechanics, covered with a unique art style that makes his games appealing for hardcore and casual audiences alike. Currently, Devon is working on a new project along with a large group of passionate developers.

I would like to thank my teachers and peers of The Art Institute of Vancouver, for helping me pursue the endeavors that I sought after. I would also like to thank my friends and family, outside of my school life, who helped keep me sane, well, as sane as I can be, and for being there when it mattered most. Prost!

About the Reviewer

Jeff Mundee is a game designer and instructor from New Brunswick, USA, who moved to Vancouver, Canada, a decade ago to produce video games. Since then he has worked on many game projects in various roles, from Motion Capture Specialist at Electronic Arts, to Game Designer for Activision, and all sorts of independent productions in between. He is currently working on a Unity-based game with Holy Mountain Games. He also teaches classes at The Art Institute of Vancouver, about game production using Unity, among other subjects.

I would like to thank Vlad and Devon for being leaders in a strong graduating class, by taking the initiative to master Unity. I know they will both go on to make great games.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

If you are an enthusiastic gamer who is ready to seriously get into game development, this book will give you a great head start for your journey. We will guide you through the step-by-step process of creating your first playable game prototype, which you will be able to further extend into a full-scale game. This book contains examples of the most important features that can be found in games, and much more; it will help you to understand Unity better, and increase your programming skills.

What this book covers

Chapter 1, *Diving into Scripting*, will teach you how to set up the project and take advantage of built-in character controllers. We will talk about dynamic objects and their collision, as well as investigate creating a moving platform and explosions.

Chapter 2, *Custom Character Controller*, will show you how to create your own character controllers, camera rigs, and animation systems.

Chapter 3, *Action Game Essentials*, will introduce programming of basic gameplay features, such as shooting, picking up items, and opening treasure boxes, as well as soft bodies and tethering.

Chapter 4, *Drag-and-Drop Inventory*, will give you an example on how to create your own inventory and character customization with the help of Unity GUI.

Chapter 5, *Dynamic GUI*, will take you step by step, through the creation of the HUD and targeting system.

Chapter 6, *Game Master Controller*, will teach you how to design and program systems to run and manage your game.

Chapter 7, *Introduction to AI Pathfinding and Behaviors*, will give you a sneak peek of AI programming, and talk about the basic theory behind it.

Appendix, *Object-oriented Programming in Unity*, will cover some basics of programming that will help you to continue learning.

What you need for this book

You need to be comfortable in an editor's environment, and have a very basic knowledge of Unity's JavaScripts, or any other object-oriented programming language.

Who this book is for

This book is for passionate game developers, students who are preparing to make their first project, or people who think they are ready to learn something new.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "After the `Start` function, we will create the `MoveButton` function."

A block of code is set as follows:

```
function Update(){
if( tnt != null ){
    If(trigObj.getComponent("Button").ReturnButtonStatus()){
         BOOM();
     }
    }
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To gain access to the package data, open **Unity** and go to **Assets** | **Import Package** | **Custom Package...**, as shown in the following screenshot".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Diving into Scripting

Welcome to advanced Unity scripting! In this book, we will cover interesting information about scripting in Unity's built-in scripting language—JavaScript for Unity. We believe that this book, and included material, has the fundamentals needed to create a game that you always dreamed of creating.

In order to start working with this book, you need to have a basic understanding of what Unity3D is; navigate freely inside Unity, and have basic knowledge of JavaScript and **object-oriented programming** (**OOP**) in general.

In this chapter, we will:

- Set up a project and a third-person Character Controller
- Talk about dynamic objects and collision detection
- Create moving platform and explosion box

Downloading and installing assets for this book

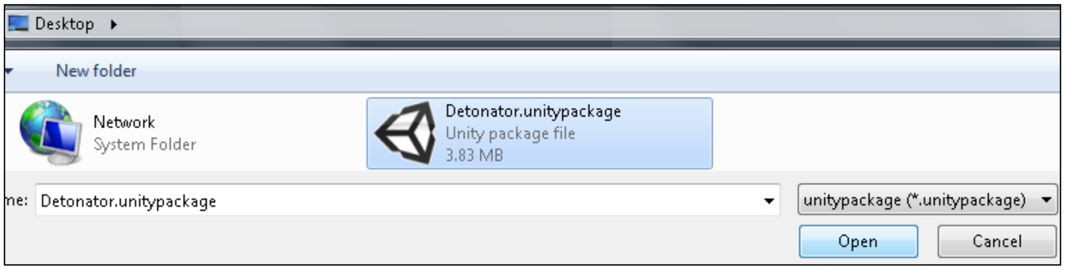
In Unity3D, there is the ability to download pre-made packages or import assets. These packages/assets can be of 3D models in the form of raw art assets, game objects, prefabs, particles, scripts, animations, sounds, and so on. Packages are identified by having a `.package` extension.

In order for the reader to be able to follow along with the examples in the book, get the greatest amount of experience, and practice out scripting in Unity, pre-made packages have been made available for the reader's convenience.

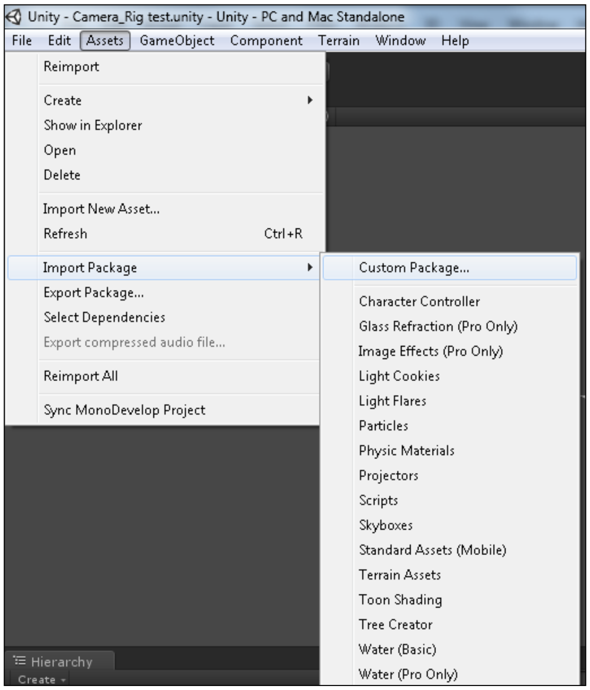
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

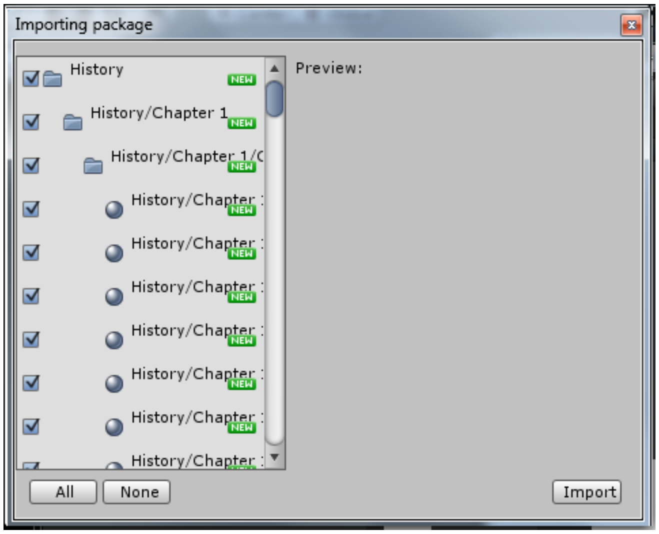
These packages are available for download on the book's website underneath the **Packages** heading. There is only one package here and it is called **Unity_Scripting.unitypackage**. The downloaded file will be a ZIP file.



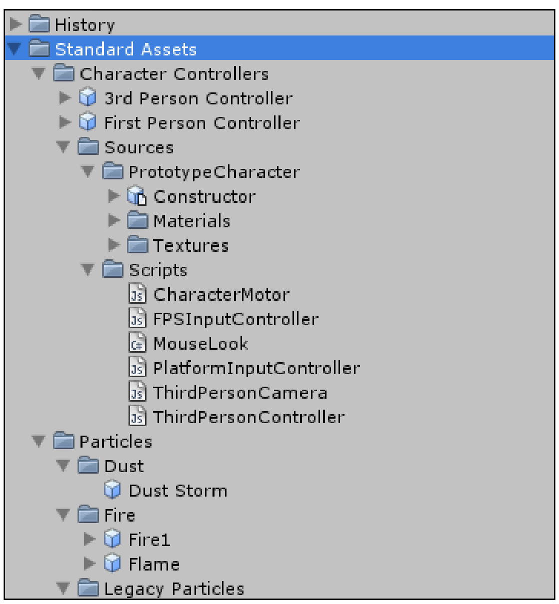
Extract the data and put the package where you would like it to be in your Unity project. To gain access to the package data, open **Unity** and go to **Assets** | **Import Package** | **Custom Package...**, as shown in the following screenshot:



Search for the location of your project and open your package. A small interface comes up showing a list of all the assets on the left-hand side and a prompt asking if you would like to install all assets. Click on **All**, as shown in the following screenshot:



This will open up the **Unity_Scripting** package. The default path for the downloaded assets is **Standard Assets** in the Unity project. If a `Standard Assets` folder does not exist, it will create one and download your package into it.



Congratulations, you have now downloaded and successfully installed the assets required for this book. Now, let's start building!

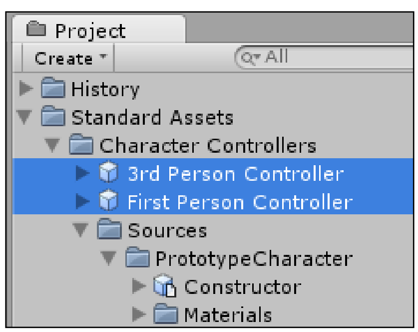
Getting started with the game

From now on, we will start to script our own game and dive into uncharted depths of JavaScript. The first chapter is dedicated to creating a simple platform game. We will learn to use the built-in functionality of Unity to set up our character, and use the Character Controller component to make that character move and be controlled with our commands. Later in the chapter, we will get into creating a playground for our character. We will also get into teaching him to move boxes around, script moving platforms, create custom triggers, and make huge explosions.

Available Character Controllers

Now, let's get into the fun part and set up a controllable character. Let's open the project that comes with the book and start coding.

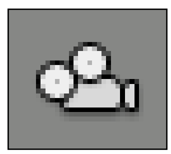
There are two kinds of Character Controllers that are available with a Pro version of Unity3D—**3rd Person Controller** and **First Person Controller**. Default Character Controllers can be found in **Project** view | **Standard Assets** | **Character Controllers**, as shown in the following screenshot. To use any of those Character Controllers, just drag-and-drop them on a scene using the left mouse button. Now, we can click **Play** and start the game, and see our character following orders when we press control buttons.



Now, let's take a look at what these Character Controllers consist of.

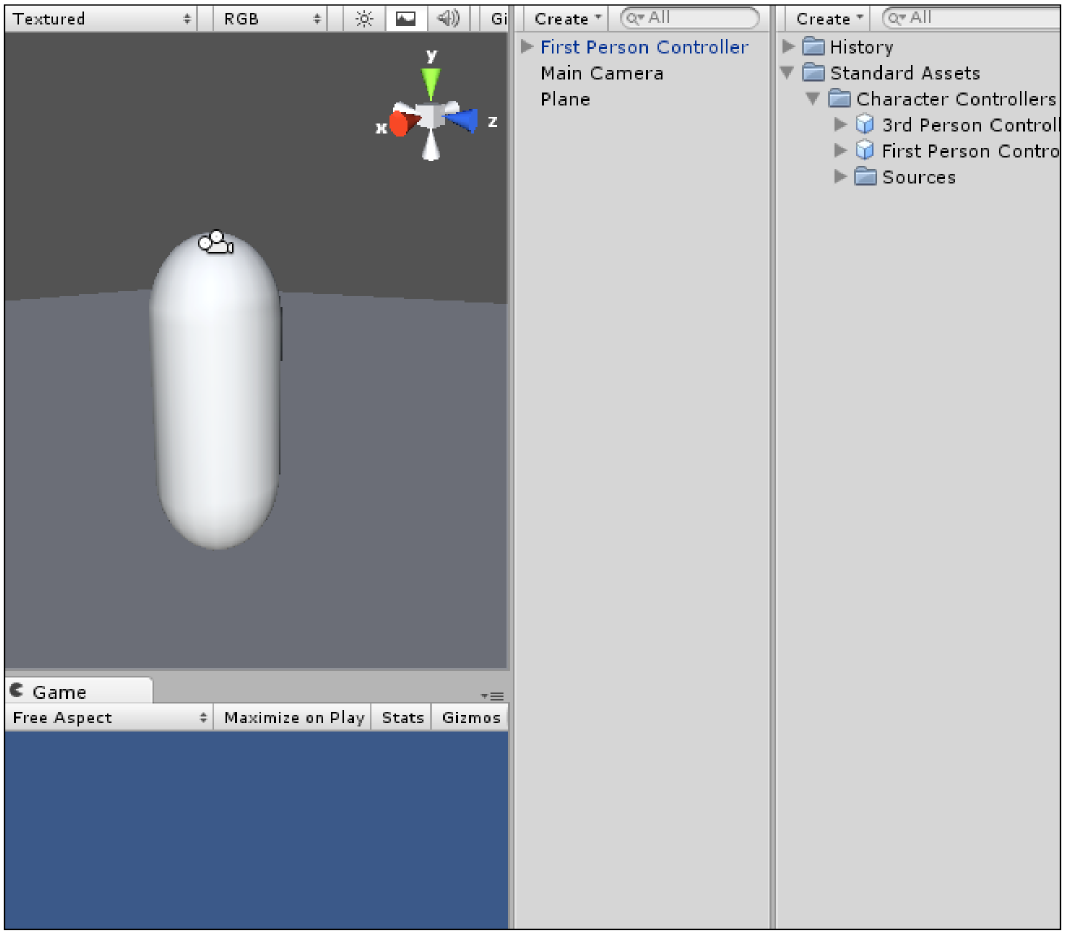
Character Controller is a default physics component that does all the necessary collision calculations for us but, at the same time, doesn't follow rules of physics and isn't affected by external forces. However, that doesn't mean that it can't push Rigidbodies if scripted. In general, if we are trying to create a controllable humanoid and don't wish bothering with tons of code, Character Controller will be our best choice. If we are planning to create a character that is being influenced by external forces (like physics) or interacting with objects that are influenced by physics, we will see Character Controller becoming our worst enemy that will break game functionality for no reason. Supplementary to Character Controller are pure physics objects—Rigidbodies. They allow us to create almost anything that is physics related and consist of many hard edges that we will go around in future chapters.

From now on, we will look into both Character Controllers separately and start with First Person Controller. By dragging **First Person Controller** prefab on the screen, we will see a simple cylinder with a camera icon above it. Let's take a look at what's inside:



- **Character Controller**: This is attached to the cylinder with the camera icon above it, at the very top of the list. To attach the Character Controller to the object, select the object, go to **Component** at the top of the screen, and click on **Physics** | **Character Controller**.
- **Mouse Look (Script)**: This handles the camera rotation based on mouse manipulations. This script is written in C# and is beyond this book's scope, but it has a fair amount of description inside, which can be used to tweak mouse controls. To attach a script, go to **Component** | **Camera Control** | **Mouse Look**.
- **Character Motor (Script)**: This is a script that is responsible for registering all the inputs and controlling **Movement**, **Jumping**, **Sliding**, and so on. It is available at **Component** | **Character** | **Character Motor**. Some of the functionality can be tweaked from the **Inspector** view, but most of it has been purposely hidden and is accessible only through scripts.

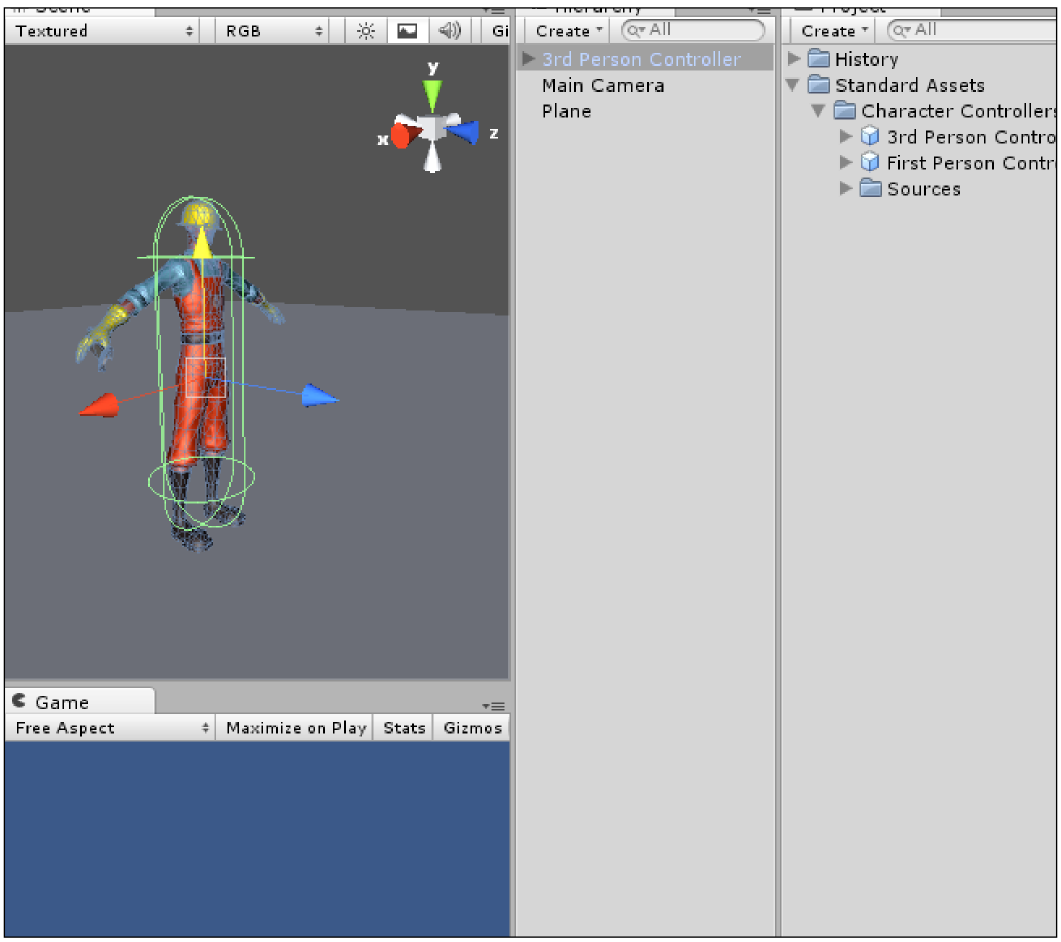
- **FPSInput Controller (Script)**: This works together with **Character Motor (Script)**. Its main purpose is to control the functionality of previous scripts (**Component** | **Character** | **FPSInput Controller**).



Now that we are done with the **First Person Controller**, lets switch to **3rd Person Controller**. There are few things that make it stand apart. They are as follows:

- **Animation**: Unlike First Person Camera, we are expecting to visually observe our character and watch it playing various types of animations. This is what **Animation** does; we simply attach it to the object (**Component** | **Miscellaneous** | **Animation**) and add baked animations to the animation array. The rest is done through code and will be covered in future chapters.

- **Third Person Controller (Script)** and **Third Person Camera (Script)**: They are self-explanatory. The first one controls character, registers inputs from the keyboard, handles animation synchronization, and so on. The latter one adjusts the camera according to character position and actions. Both scripts can be found in **Component** | **Scripts**.
- **Character Motor (Script)**: This is a script that is responsible for registering all the inputs and controlling **Movement**, **Jumping**, **Sliding**, and so on. It is available at **Component** | **Character** | **Character Motor**. Some of the functionality can be tweaked from the **Inspector** view, but most of it has been purposely hidden and is accessible only through scripts.
- **FPSInput Controller (Script)**: It works together with **Character Motor (Script)**. Its main purpose is to control the functionality of previous scripts (**Component** | **Character** | **FPSInput Controller**).



Interactive objects

So, you want to interact with objects in the environment now? Interactive objects are usually the objects, which the player has to interact with in order to continue their progression through a level and/or environment. In deciding which interactive items to include as examples, we have chosen to pick objects that show a variety of player interactions. The following is an overview of the type of interactive objects, which will be covered in this chapter:

- Buttons/plunger
- Explosion box
- Moving boxes
- Platform

The list of interactive items can be quite extensive but luckily, once you have thought of the logic behind one, scripting another becomes easier. For a better understanding of the preceding interactive objects, we can split them into two categories—**Triggers** and **Triggered Objects**. TNT plunger, targets, buttons, levers, and volumes fall under the **Triggers** category, whereas TNT box, triggered door, item required/event door, breakable door, and raft fall under **Triggered Objects**. For more information on other interactive items such as pickups, treasure chests, and weapons, see *Chapter 3*, *Action Game Essentials*. All assets for this chapter can be found in the **History | Resources | Chapter 1** folder.

Triggers

As stated previously, these objects are used to trigger events in the environment. Through interacting with them, doors can be opened, non-interactive events can be triggered, and enemies can be spawned. These are only a couple of examples of the infinite number of tasks that can be done by interacting with a trigger. Here is a breakdown of the mentioned triggers. Due to the limited number of pages, we will dive right into the description and breakdown of code for each project.

Buttons

In our case, a button will be described as an object, which the character directly has to interact with in order for it to be triggered. What we will write is a base script, which when used triggers an event. This script, once written, will be used to open a door and explode a box of TNT.

Base button script

So let's script a button. Go grab the **Button** prefab from the **Chapter 1 prefabs** folder and drag it into the **Hierarchy** view. Once that is done, there will be two game objects in the prefab asset. In **buttonTrigger**, there is a default script on the asset called **Button**.

In the `Start` function of this script, we want to get the initial position of the button.

Declare a variable for initial position, make its type a `Vector3` and default it to `Vector3.zero`. To get the position, have the variable equal to `transform.localPosition` in the `Start` function:

```
var initPos : Vector3;
function Start(){
     initPos = transform.localPosition;
}
```

After the `Start` function, we will create the `MoveButton` function.

Activating platform status

This next function will move the button to the move position and set the activated status for the platform.

Create a `private` variable for the button pressed, set its type as `Boolean` and default it to `false`. Inside the `MoveButton` function, create an `if` statement. Have the `if` statement check to see if the button pressed variable is equal to `false`. Inside the `if` statement, we want to send the activation information to triggered object.

To send the information to the appropriate platform in the level, we will have to create a new variable called `Platform`, or something along those lines, with the type of `gameObject` and defaulted to `null`. In the `MoveButton` function, we need to call the `Activated` function in the `platform` script (this script will be created later in this chapter). The following is an example of what it could look like:

```
Platform.GetComponent( platform ).Activated();
```

Now, we need to move the button to give visual indication to the player that the button has been pressed. To get the move position, create another variable for move position, set its type as a `float` and default its value to `0.1` (this value can be adjusted later in the inspector).

```
var movePos : float = 0.1;
```

To move the button from its current position to the new position, we will take the local Z position of the button, subtract the move position value and apply it to the current local position of the button (we will use the **Z** axis for the example due to the world having Z as depth and the button being mounted on a wall).

The last thing to add to this `if` statement before we close it is to turn the button pressed variable `true`. That's it for this script. We just need to add the collision check function to the built-in `Character Controller` script and we will have functionality.

Inside of this function, we will do a name check to identify what object the character has collided with. In order to get the name information from the collided object, we have to access the name component, which is a property of `gameObject`. We will then compare this to one that we want, which in this case is `Button`:

```
function OnControllerColliderHit ( Hit : ControllerColliderHit){
     if ( Hit.gameObject.name == "Button" ){
     }
}
```

If the name matches what we want, we need to access the `MoveButton` function in the `Button` script. To do this, use `GetComponent` to grab the `Button` script and access the desired function. The following statement shows roughly what it should look like:

```
Hit.gameObject.GetComponent("Button").MoveButton();
```

Then in the `if` statement for the detonator plunger, we want to access the `GetPressed` function in the `Button` script.

You have finished writing the base `Button` script. The following is a sample of what that script could look like:

```
var initPos : Vector3;
var movePos : float = 0.1;
var Platform : Transform = null;
var isPressed : Boolean = false;
function Start(){
     initPos = transform.position;
}
function MoveButton(){
    if(!isPressed){
     Platform.GetComponent( platform ).Activated();
     transform.position.z = transform.position.z - movePos;
    isPressed = true;
    }
}
```

Remember that this is a base script and much, much more functionality can be scripted into it.

Explosion box

It's time to make things explode. Let's script a little bit of explosion box. When the player applies pressure to a detonator box, it triggers the explosion box, and the explosion box explodes! There are just six steps to achieve that, as follows:

1. Prepare objects.
2. Write `Update` function.
3. Write `BOOM` function.
4. Download and install **Detonator** package.
5. Write functionality for button pressing.
6. Preparation.

In this section, we will handle the entire preparation of available resources.

> Grab the **Detonator_Box** prefab out of the `Chapter 1 prefabs` folder and drag it into the **Hierarchy** view.

If you open the `gameObject` of the detonator box, you will see that it is made up of two pieces—**Detonator_Box** and **Explosion Box**. We want to drag the **Button** script, made in the last example, to the inspector of the **Detonator_Plunger** asset located underneath the **Detonator_Box** group. As the plunger is essentially a button, and the base `Button` script is generic, it can be used for many purposes, such as triggering the explosion box to explode. This script will be the master control for the explosion box as well as the detonator box. It will determine the explosion created when the explosion box explodes, what object is used as the trigger, and the object triggered. You will notice that the prefab parent of **Detonator_Box** has the **TNT** script in its **Inspector** menu.

The Update function

The next function that we will write is the `Update` function. In this function, we will do a check for getting the trigger object's pressed status.

First, we have to create a couple of variables—the first one for a trigger and the second one for the explosion box. We want the trigger variable to be of `Transform` type and defaulted to `null` and the `tnt` variable to be of a `Transform` type as well and defaulted to `null`. Create an `Update` function. We will have an `if` statement to make sure that the `tnt` variable has an object associated with it.

To do the trigger check, we will have to write an `if` statement that gets the `Button` script component from the trigger object. To do this, we will have to declare a new variable, make it `public` and call it something along the lines of `trigObj`. We should declare its type as `gameObject`, and default it to `null`.

The value we need for this statement is the return function located in the `Button` script. To access this, we get the script component of the trigger object using `GetComponent`. We then declare the script by the name that we wish to access and then the name of the function which has the value to check. The following is an example.

```
function Update(){
if( tnt != null ){
    If(trigObj.getComponent("Button").ReturnButtonStatus()){
         BOOM();
     }
    }
}
```

As you can see, we have added the `BOOM` function in the name of the next function, which we will be writing.

The BOOM function

The `BOOM` function will create an explosion at the location of the explosion box and destroy the explosion box game object from the **Hierarchy** view. Before we do anything, let's declare two more variables. The first variable is `explosion` and the second one is `collidedObj`. Make sure that explosion is `public`, its type declaration is `Transform`, and it is defaulted to `null`. The `collidedObj` variable should be `private`, and the type declaration should be as a `Collider` array.

In the `BOOM` function, we want to create a collision sphere that will detect all colliders within a given area from a given point. To accomplish this, we will use the `Physics.OverlapSphere` function. Have the `collidedObj` variable equal to the `Physics` function with the parameters of the `tnt` variables—`position` for position and the size of the collision sphere set to `1`. The following is an example of how it should look:

```
collidedObj = Physics.OverlapSphere(tnt.transform.position, 1);
```

After this, we need to go through the `collidedObj` array and for each object in that array, create an explosion at its position and then destroy the object. To do this use a `for` loop to loop through the array. Call Unity's built-in creation function—`Instantiate` inside of the loop.

The `Instantiate` parameters are the explosion variables, `Obj` in the `collidedObj` array position and then a rotation. The rotation of the current `gameObject` will perform `transform.rotation`. The following is a sample:

```
for (var obj in collidedObj)    {
    Instantiate(explosion, obj.transform.position, transform.
rotation);
}
```

Lastly, we will destroy the `gameObject` in the array. To do that, after the instantiation code, type the following line:

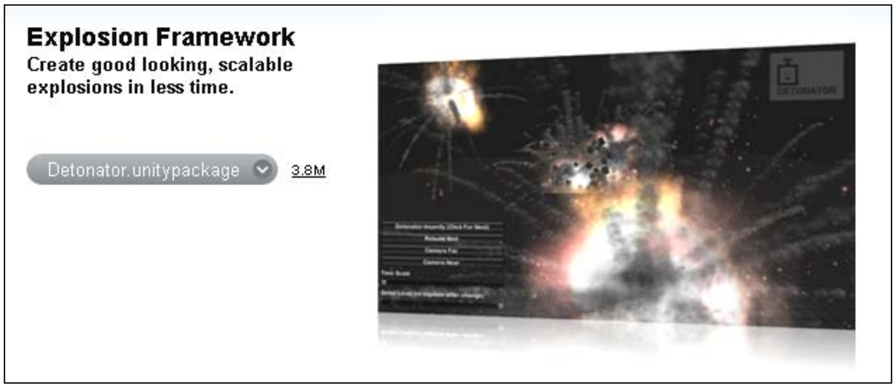
```
destroy(obj.gameObject);
```

Downloading the Detonator package

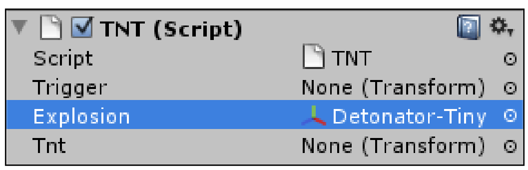
Now, return to the **Inspector** of **Detonator_Box**. Under the **TNT** script, you will see the variables that were `public`. These variables are, for example, trigger, explosion, and TNT.

In the trigger variable, drag your detonator trigger into it. For the explosion variable, we are going to do something different. For the explosion, we will utilize the **Detonator** package that can be downloaded off of Unity's website. You can find it in the **Support | Resources** section at `http://unity3d.com/support/resources/`.

You will find it in the **Resources** section, at the very bottom on `http://unity3d.com/support/resources/`, as shown in the following screenshot:



This package is downloaded and imported into Unity the same way as the assets for the book. It will also appear in the `Resources` folder. Below it, you will see **Prefab Examples**. In here, you will find a variety of pre-made explosion prefabs. For our purposes, just go ahead and drag the **Detonator-Tiny** into your explosion variable, as shown in the following screenshot:



The TNT variable—`obj` is the Explosion_Box object in the Detonator_Box prefab in the Hierarchy.

Remember, you can move the detonator box and the Explosion_Box box anywhere you want. Just make sure that they stay within the parental hierarchy in the **Hierarchy** view.

The following is an example of a complete `TNT` script:

```
var trigObj: Transform = null;
var explosion : Transform = null;
var tnt : Transform = null;
private var collidedObj : Collider[];
function Boom(){
```

```
        collidedObj = Physics.OverlapSphere(tnt.transform.position, 1);
     for (var obj in collidedObj){
            Instantiate(explosion, obj.transform.position,
             transform.rotation );
    Destroy(obj.gameObject);
    }
    }
    function Update(){
        if( tnt != null ){
        if(trigObj.getComponent(Button).ReturnButtonStatus();){
                Boom();
        }
        }
    }
```

Pressing the button

Now we need to add an `if` statement to the **ThirdPersonController** script inside of the `OnControllerColliderHit` function.

We need to check if the character is interacting with a detonator plunger. Duplicate the **Button**, check the `if` statement, and paste it below that `if` statement. This time, in place of the name `Button`, put `Detonator_Plunger` instead. The following statement shows an example of what it should look like:

```
if(objCollided.gameObject.name == "Detonator_Plunger"){
     }
```

Inside this `if` statement, access the `GetPressed` function inside of the **Button** script. Again, copy the `GetComponent` statement in the preceding `if` statement. Change the `MoveButton` function to `GetPressed`.

Now, we must get back to the **Button** script. We need to write the `GetPressed` function at the end of this script. In this function, check if the button/plunger has been pressed. If not, move the button. Move the button down so that it is on the **Y** axis, and by the `movePos` value multiplied by the time variable—`Time.deltaTime`:

```
transform.localPosition.y -=  movePos * Time.deltaTime;
```

Next, add another `if` statement inside of the previous one. This `if` statement is going to check if the current position is greater than the initial position subtracted by the `movePos` variable's value. Inside this `if`, we can have `isPressed` equal to `true`.

Lastly for this script, we will create a simple return function. This return function—`ReturnButtonStatus`—is going to return the `isPressed` variable's value:

```
function ReturnButtonStatus(): boolean{
    return isPressed;
}
```

Congratulations, you can now have fun blowing stuff up. Remember that these are very basic scripts and much more functionality can be added to them to make them a much more complex mechanism. Add the following code snippet at the end of the `Button` script:

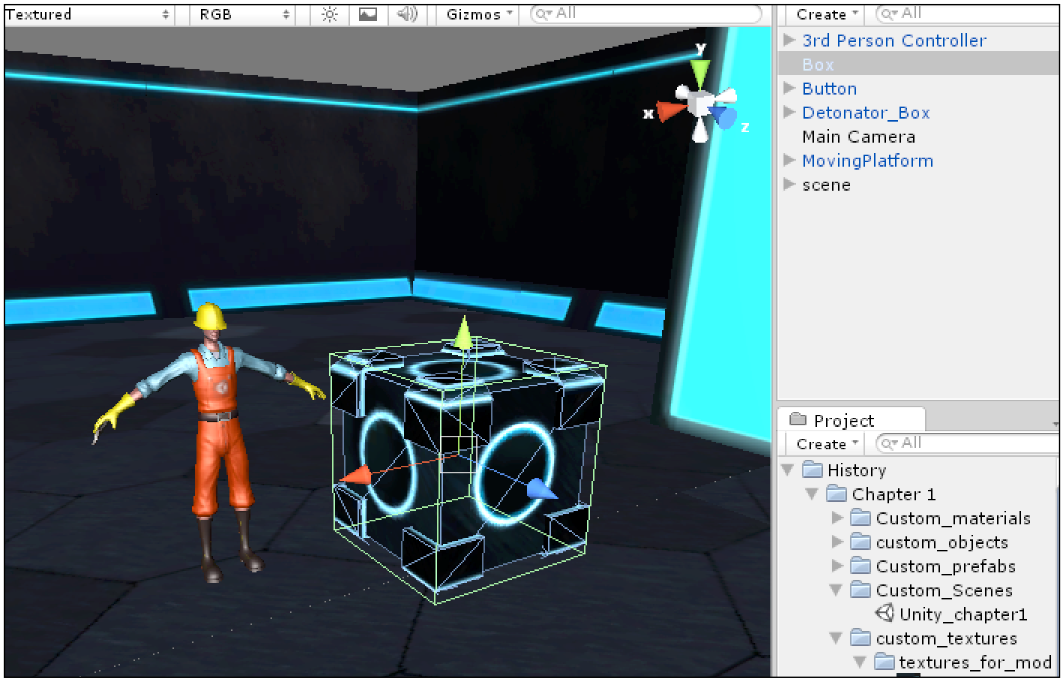
```
function GetPressed(){
    if( !isPressed ){
        transform.localPosition.y -=  movePos * Time.deltaTime;
        if(transform.localPosition.y > (initPos.y - movePos)){
            isPressed = true;
        }
    }
}
function ReturnButtonStatus(): boolean{
    return isPressed;
}
```

Dynamic objects

Now that you can interact with objects that don't move, why don't we interact with something more dynamic? Dynamic interactive objects are the ones that are, well, moving. They usually involve adding their velocity to the character. For example, this could be done when we want our character to travel through the moving train level. The train's velocity is being added to the Character Controller to keep the character's position with that of the train's position. Once the player moves the joystick, moving the character, the additional velocity from the input value is added to the current velocity of the Character Controller creating a clean, smooth walk/run/jump while on the moving train. We will cover this transfer of velocity of one object to another.

Moving boxes

Now let's teach our construction worker (character) to move boxes. Select the **Box** prefab from the **Chapter 1 prefabs** folder in the **Project** view and drag it to the scene. Having an instance of **Box** selected, go to **Component** | **Physics** | **Rigidbody** to make the box follow the laws of physics.



At the beginning of the chapter, we talked about Character Controller and its problems with Rigidbodies. For example, if we press **Play** and try to push the box with our character, we'll see that no matter what we do, there is no way our character can make this box move, not even an inch. That happens because the default **3rd Person Character Controller** doesn't know what to do when interacting with objects. Neither will the Character Controller component help us because it's not programmed to affect physics objects.

To solve our problem, we will have to modify the original script and add additional functionality to be able to interact with dynamic objects:

1. Select the character and open the attached **ThirdPersonController** script. Scroll down and find the **OnControllerColliderHit** function.
2. Declare the local variable—`pushForce` of a `float` type that will control the force that will be applied to the box when we push it. To make it more interesting, let's assign it a current speed that the character is moving at. For that, we will use the default function—`GetSpeed` that will return the speed of our character. (This way we can push the object even further if we run into it.)
3. Declare the `body` variable of `Rigidbody` type that will contain information of **Rigidbody** from the object we collided with.
4. Just to make sure that this script works only with objects that have **Rigidbody** attached, we'll add the following check:

   If the object that we collided with doesn't have **Rigidbody** or is checked as kinematic, then stop here and get out from the function.
5. Check if the distance from the center of the capsule collider to the place we are touching is less than -0.3 (simply if the object is below us and we don't want to push it).
6. Declare a `dirVector` variable of a `Vector3` type that will be used to prevent the box from moving up or down when we're pushing it. `dirVector` should store the current location of the box on the **X** and **Z** axes.
7. Apply force to the box's **Rigidbody** by modifying its velocity and multiplying its force by a vector.
8. In this case, we want the box to slide on the surface rather than spin over. To make that possible, select **Box** object and in the **Rigidbody** component, under **Constraints** check, select all boxes for **Freeze Rotation**.

The following code snippet shows how the `OnControllerColliderHit` function should look at this point:

```
function OnControllerColliderHit ( hit : ControllerColliderHit ){
if ( hit.gameObject.name == "Button" ){
        hit.gameObject.GetComponent("Button").MoveButton();
    }
var pushForce : float = GetSpeed();
var body : Rigidbody = hit.collider.attachedRigidbody;
if (body == null || body.isKinematic)
return;
if (hit.moveDirection.y < -0.3)
```

```
return;
var dirVector : Vector3 = Vector3(hit.moveDirection.x, 0,
                                  hit.moveDirection.z);
body.velocity = dirVector * pushForce;
}
```

Done. If we test the game now, we will see that when our character runs over the box, it is being pushed in the direction of the impact.

Triggered object

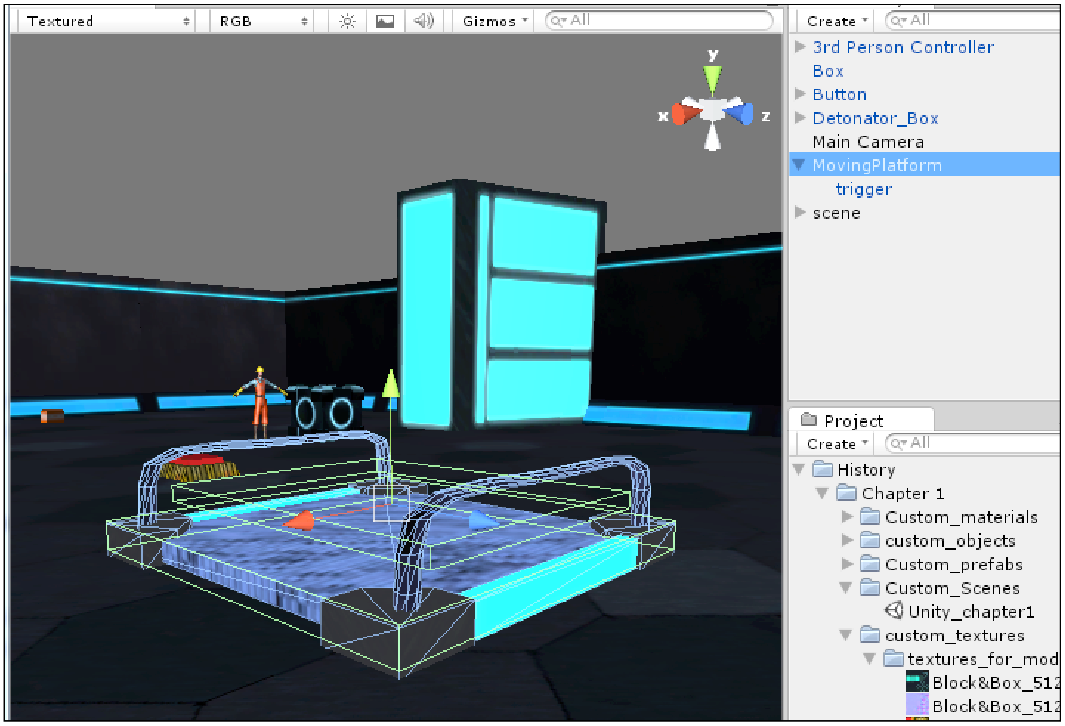
Next in line are triggered objects. These objects are classified by needing an outside interaction to make them active and usable. In games as well as in real life, this device that can activate them is usually found to be a switch, a lever, or a button. Unique triggers can be used as well, for example, when a player enters an area and triggers a cinematic sequence.

Moving platform

In this section, we will learn how to create button-triggered platforms and teach our character to move along with them. Let's get started. Perform the following steps:

1. Create new script and call it **platform** (this name will be used to reference it in the future).

2. Attach it to the **platform** prefab that can be found in a scene.



3. Declare two `public` variables of `GameObject` type. Call them PointA and PointB. We will use position information of these objects to navigate movement of our platform.
4. Declare another variable of a `Vector3` type, `private` this time, and call it `Target`. This variable will tell the platform where to move at this moment in time.
5. Create `Awake` function and assign position of PointA to the `Target` vector.
6. We need a function that will be changing targets for platform when it reached its destination to reverse direction. Create a `Toggle` function that will check current target and change it to opposite.
7. Then next thing we need is to control the platform to stop it from moving. Declare a `public` variable—`AllowMove` of a `Boolean` type and set its default value to `false` (we don't want to start moving the platform at the game start).
8. Write the `Activated` function that will activate the platform to move if it's not moving.

9. Finally, we need an `Update` function that will handle the platform moving and stop movement if the platform reached the goal.

The completed **platform** script is as follows:

```
public var AllowMove : boolean = false;
public var PointA : GameObject;
public var PointB : GameObject;
private var Target : Vector3;
function Awake(){
    Target = PointA.transform.position;
}
function Update(){
    if (AllowMove == true){
    this.transform.position =
    Vector3.MoveTowards(this.transform.position, Target, Time.
deltaTime);
    }
    if (this.transform.position == Target){
        AllowMove = false;
        Toggle();
    }
}
function Toggle(){
    if (Target == PointA.transform.position)
    Target = PointB.transform.position;
    else
    Target = PointA.transform.position;
}
function Activated(){
        if(AllowMove == false )
            AllowMove = true;
}
```

Moving the character with the platform

Now, we are done with the **platform** script and have a tough platform to move upon our command. However, if we jump at the platform when it moves, we will see that it simply moves away while we are standing still. Our new objective is to make the character move with the platform while standing on it. Let's create a new script and call it **moveAlong**. Perform the following steps:

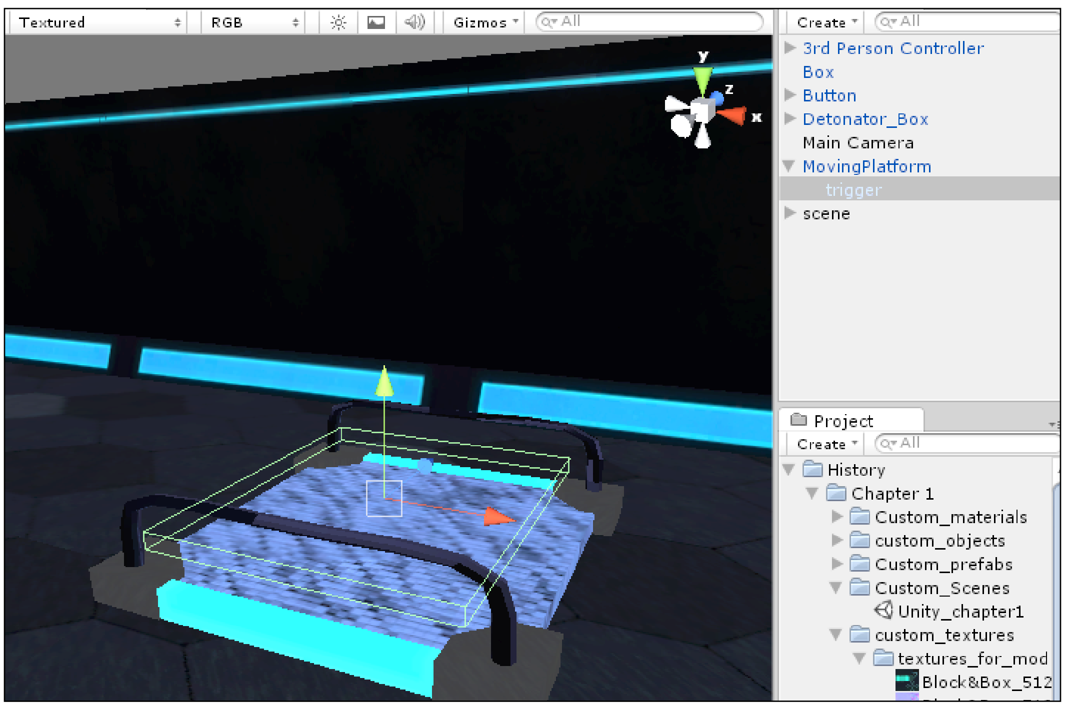
1. We will need one variable of `Vector3` type to guide in which direction to move the player.

2. Last, but not least, we will declare an `OnTriggerStay` function that will be triggered when a player is standing on it. We will also check the `AllowMove` variable that will tell us if the platform is moving or not.
3. Inside the `if` statement, we will get the destination of the platform and apply movement to the player.

The following is an example of the complete **moveAlong** script:

```
public var MoveTo:Vector3;
function OnTriggerStay ( other : Collider ) {
if( other.gameObject.tag == "Player" && transform.root.
GetComponent("platform").AllowMove ){
MoveTo = transform.root.GetComponent("platform").Target;
MoveTo.y = other.gameObject.transform.position.y;
MoveTo.z = other.gameObject.transform.position.z;
other.gameObject.transform.position = Vector3.MoveTowards( other.
gameObject.transform.position, MoveTo, Time.deltaTime );
}
}
```

Attach this script to the child of our **platform**, which is called **trigger** and we are done. Now we have a fully functional moving platform that will carry our character with it:



Summary

We have covered the basics of scripting basic triggers and the activation of objects based upon those triggers and the use of Unity's Character Controller component. We hope that you have come away with at least a better understanding of how to go about tackling the preparation and implementation of these game components.

In the next chapter, we will cover creating a custom Character Controller with an explanation for implementing animations and a camera rig system, which will allow you to change between a first-person view, a shoulder view, and a bird's eye/third-person view at the press of a button. Please continue to enjoy the book and we await you in *Chapter 2*, *Custom Character Controller*, to learn about animations and camera rigs.

2

Custom Character Controller

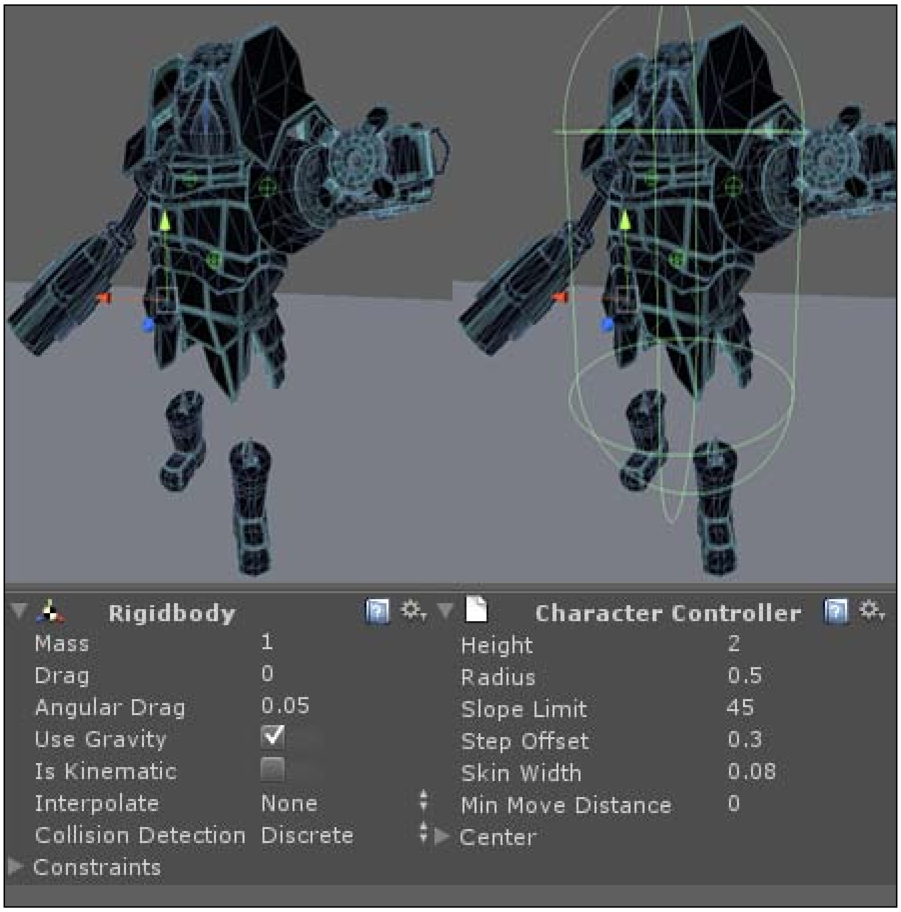
Welcome to creating custom Character Controller in Unity. In this chapter, we will go through scripting our own fully controllable character without Character Controller component, with attached Rigidbody. We will cover the barebones of movement such as walking, running, and jumping. We will talk about the ways to create fully player-controlled first-person, third-person, and shoulder view camera with ability to switch in between them at any time, and finally learn to attach and control animations through code. In this long awaited chapter, we will learn the following topics:

- Character Controller versus Rigidbody – pros and cons
- Player-controlled character walk, run, jump, and shoot
- Program camera controls and switching between different camera types with a press of a single button
- Script animations to follow characters actions

Creating a controllable character

As described in the previous chapter, there are two ways to create a controllable character in Unity, by using the Character Controller component or physical Rigidbody. Both of them have their pros and cons, and the choice to use one or the other is usually based on the needs of the project. For instance, if we want to create a basic role playing game, where a character is expected to be able to walk, fight, run, and interact with treasure chests, we would recommend using the Character Controller component. The character is not going to be affected by physical forces, and the Character Controller component gives us the ability to go up slopes and stairs without the need to add extra code. Sounds amazing, doesn't it? There is one caveat. The Character Controller component becomes useless if we decide to make our character non-humanoid. If our character is a dragon, spaceship, ball, or a piece of gum, the Character Controller component won't know what to do with it.

It's not programmed for those entities and their behavior. So, if we want our character to swing across the pit with his whip and dodge traps by rolling over his shoulder, the Character Controller component will cause us many problems.



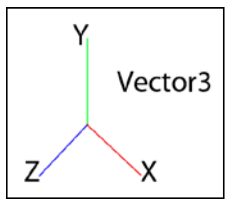
In this chapter, we will look into the creation of a character that is greatly affected by physical forces, therefore, we will look into the creation of a custom **Character Controller** with **Rigidbody**, as shown in the preceding screenshot.

Custom Character Controller

In this section, we will write a script that will take control of basic character manipulations. It will register a player's input and translate it into movement. We will talk about **vectors** and **vector arithmetic**, try out **raycasting**, make a character obey our controls and see different ways to register input, describe the purpose of the `FixedUpdate` function, and learn to control **Rigidbody**.

We shall start with teaching our character to walk in all directions, but before we start coding, there is a bit of theory that we need to know behind character movement.

Most game engines, if not all, use vectors to control the movement of objects. Vectors simply represent direction and magnitude, and they are usually used to define an object's position (specifically its **pivot point**) in a 3D space. Vector is a structure that consists of three variables—X, Y, and Z. In Unity, this structure is called **Vector3**, but we have encountered this variable type before in a previous chapter:



To make the object move, knowing its vector is not enough.

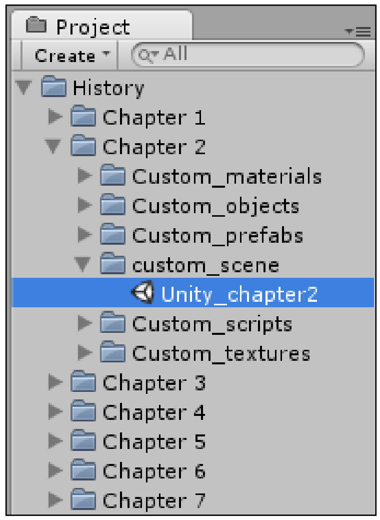
Length of vectors is known as **magnitude**. In physics, speed is a pure scalar, or something with a magnitude but no direction. To give an object a direction, we use vectors. Greater magnitude means greater speed. By controlling vectors and magnitude, we can easily change our direction or increase speed at any time we want.

Vectors are very important to understand if we want to create any movement in a game. Through the examples in this chapter, we will explain some basic vector manipulations and describe their influence on the character. It is recommended that you learn extra material about vectors to be able to perfect a Character Controller based on game needs.

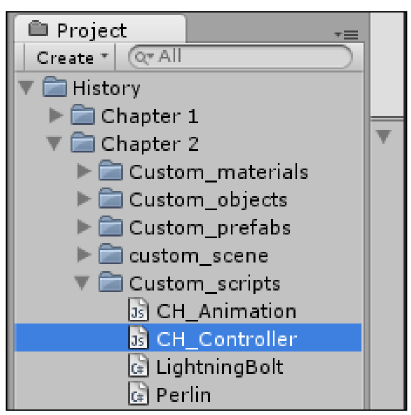
Setting up the project

To start this section, we need an example scene. Perform the following steps:

1. Select **Chapter 2** folder from book assets, and click on on the **Unity_chapter2** scene inside the **custom_scene** folder.



2. In the **Custom scripts** folder, create a new JavaScript file. Call it **CH_Controller** (we will reference this script in the future, so try to remember its name, if you choose a different one):



3. In a **Hierarchy** view, click on the object called **robot**. Translate the mouse to a **Scene** view and press *F*; the camera will focus on a funny looking character that we will teach to walk, run, jump, and behave as a character from a video game.

Creating movement

The following is the theory of what needs to be done to make a character move:

1. Register a player's input.
2. Store information into a vector variable.
3. Use it to move a character.

Sounds like a simple task, doesn't it? However, when it comes to moving a player-controlled character, there are a lot of things that we need to keep in mind, such as vector manipulation, registering input from the user, raycasting, Character Controller component manipulation, and so on. All these things are simple on their own, but when it comes to putting them all together, they might bring a few problems. To make sure that none of these problems will catch us by surprise, we will go through each of them step by step.

Manipulating character vector

By receiving input from the player, we will be able to manipulate character movement. The following is the list of actions that we need to perform in Unity:

1. Open the **CH_Character** script.
2. Declare `public` variables `Speed` and `MoveDirection` of types `float` and `Vector3` respectively. `Speed` is self-explanatory, it will determine at which speed our character will be moving. `MoveDirection` is a vector that will contain information about the direction in which our character will be moving.
3. Declare a new function called `Movement`. It will be checking horizontal and vertical inputs from the player.
4. Finally, we will use this information and apply movement to the character.

   An example of the code is as follows:

```
public var Speed : float = 5.0;
public var MoveDirection : Vector3 = Vector3.zero;
function Movement (){
if (Input.GetAxis("Horizontal") || Input.GetAxis("Vertical"))
MoveDirection = Vector3(Input.GetAxisRaw("Horizontal"),MoveDirecti
on.y, Input.GetAxisRaw("Vertical"));
this.transform.Translate(MoveDirection);
}
```

Register input from the user

In order to move the character, we need to register an input from the user. To do that, we will use the `Input.GetAxis` function. It registers input and returns values from -1 to 1 from the keyboard and joystick. `Input.GetAxis` can only register input that had been defined by passing a string parameter to it. To find out which options are available, we will go to **Edit** | **Projectsettings** | **Input**. In the **Inspector** view, we will see **Input Manager**.

Click on the **Axes** drop-down menu and you will be able to see all available input information that can be passed to the `Input.GetAxis` function. Alternatively, we can use `Input.GetAxisRaw`. The only difference is that we aren't using Unity's built-in smoothing and processing data as it is, which allows us to have greater control over character movement.

To create your own input axes, simply increase the size of the array by 1 and specify your preferences (later we will look into a better way of doing and registering input for different buttons).

`this.transform` is an access to transformation of this particular object. **transform** contains all the information about translation, rotation, scale, and children of this object (object parenting will be covered in later chapters of this book). `Translate` is a function inside Unity that translates **GameObject** to a specific direction based on a given vector.

If we simply leave it as it is, our character will move with the speed of light. That happens because translation is being applied on character every frame. Relying on frame rate when dealing with translation is very risky, and as each computer has different processing power, execution of our function will vary based on performance. To solve this problem, we will tell it to apply movement based on a common factor—time:

```
this.transform.Translate(MoveDirection * Time.deltaTime);
```

This will make our character move one Unity unit every second, which is still a bit too slow. Therefore, we will multiply our movement speed by the `Speed` variable:

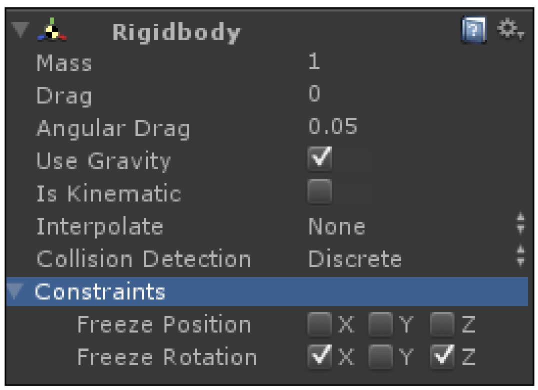
```
this.transform.Translate((MoveDirection * Speed) * Time.deltaTime);
```

Now, when the `Movement` function is written, we need to call it from `Update`. A word of warning though—controlling **GameObject** or **Rigidbody** from the usual `Update` function is not recommended since, as mentioned previously, that frame rate is unreliable. Thankfully, there is a `FixedUpdate` function that will help us by applying movement at every fixed frame. Simply change the `Update` function to `FixedUpdate` and call the `Movement` function from there:

```
function FixedUpdate (){
Movement();
}
```

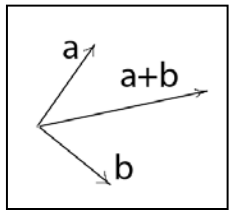
The Rigidbody component

Now, when our character is moving, take a closer look at the **Rigidbody** component that we have attached to it. Under the **Constraints** drop-down menu, we will notice that **Freeze Rotation** for **X** and **Z** axes is checked, as shown in the following screenshot:



If we uncheck those boxes and try to move our character, we will notice that it starts to fall in the direction of the movement. Why is this happening? Well, remember, we talked about **Rigidbody** being affected by physics laws in the engine? That applies to friction as well. To avoid force of friction affecting our character, we forced it to avoid rotation along all axes but **Y**. We will use the **Y** axis to rotate our character from left to right in the future.

Another problem that we will see when moving our character around is a significant increase in speed when walking in a diagonal direction. This is not an unusual bug, but an expected behavior of the `MoveDirection` vector. That happens because for directional movement we use vertical and horizontal vectors. As a result, we have a vector that inherits magnitude from both, in other words, its magnitude is equal to the sum of vertical and horizontal vectors.



To prevent that from happening, we need to set the magnitude of the new vector to 1. This operation is called **vector normalization**. With normalization and speed multiplier, we can always make sure to control our magnitude:

```
this.transform.Translate((MoveDirection.normalized * Speed) * Time.
deltaTime);
```

Jumping

Jumping is not as hard as it seems. Thanks to **Rigidbody**, our character is already affected by gravity, so the only thing we need to do is to send it up in the air. Jump force is different from the speed that we applied to movement. To make a decent jump, we need to set it to 500.0). For this specific example, we don't want our character to be controllable in the air (as in real life, that is physically impossible). Instead, we will make sure that he preserves transition velocity when jumping, to be able to jump in different directions. But, for now, let's limit our movement in air by declaring a separate vector for jumping.

User input verification

In order to make a jump, we need to be sure that we are on the ground and not floating in the air. To check that, we will declare three variables—`IsGrounded`, `Jumping`, and `inAir`—of a type `boolean`. `IsGrounded` will check if we are grounded. `Jumping` will determine if we pressed the jump button to perform a jump. `inAir` will help us to deal with a jump if we jumped off the platform without pressing the jump button. In this case, we don't want our character to fly with the same speed as he walks; we need to add an `airControl` variable that will smooth our fall.

Just as we did with movement, we need to register if the player pressed a jump button. To achieve this, we will perform a check right after registering `Vertical` and `Horizontal` inputs:

```
public var jumpSpeed : float = 500.0;
public var jumpDirection : Vector3 = Vector3.zero;
public var IsGrounded : boolean = false;
public var Jumping : boolean = false;
public var inAir : boolean = false;
public var airControl : float = 0.5;
function Movement(){
if (Input.GetAxis("Horizontal") || Input.GetAxis("Vertical")) {
MoveDirection = Vector3(Input.GetAxisRaw("Horizontal"),MoveDirection.y
,Input.GetAxisRaw("Vertical"));
}
if (Input.GetButtonDown("Jump") && isGrounded) {}
}
```

`GetButtonDown` determines if we pressed a specific button (in this case, *Space bar*), as specified in **Input Manager**. We also need to check if our character is grounded to make a jump.

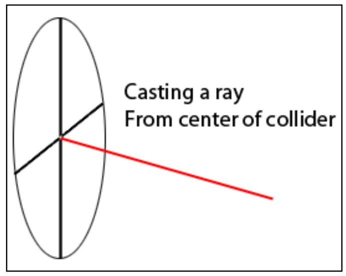
We will apply vertical force to a `rigidbody` by using the `AddForce` function that takes the vector as a parameter and pushes a `rigidbody` in the specified direction. We will also toggle `Jumping` `boolean` to `true`, as we pressed the jump button and preserve velocity with `JumpDirection`:

```
if (Input.GetButtonDown("Jump") &&isGrounded){
Jumping = true;
jumpDirection = MoveDirection;
rigidbody.AddForce((transform.up) * jumpSpeed);
}
if (isGrounded)
this.transform.Translate((MoveDirection.normalized * Speed) * Time.
deltaTime);
else if (Jumping || inAir)
this.transform.Translate((jumpDirection * Speed * airControl) * Time.
deltaTime);
```

To make sure that our character doesn't float in space, we need to restrict its movement and apply translation with `MoveDirection` only, when our character is on the ground, or else we will use `jumpDirection`.

Raycasting

The jumping functionality is almost written; we now need to determine whether our character is grounded. The easiest way to check that is to apply **raycasting**. Raycasting simply casts a ray in a specified direction and length, and returns if it hits any collider on its way (a collider of the object that the ray had been cast from is ignored):



To perform a raycast, we will need to specify a starting position, direction (vector), and length of the ray. In return, we will receive `true`, if the ray hits something, or `false`, if it doesn't:

```
function FixedUpdate ()
{
if (Physics.Raycast(transform.position, -transform.up, collider.
height/2 + 2)){
    isGrounded = true;
    Jumping = false;
    inAir = false;
}
else if (!inAir){
    inAir = true;
    JumpDirection = MoveDirection;
}
Movement();
}
```

As we have already mentioned, we used `transform.position` to specify the starting position of the ray as a center of our collider. `-transform.up` is a vector that is pointing downwards and `collider.height` is the height of the attached collider. We are using half of the height, as the starting position is located in the middle of the collider and extended ray for two units, to make sure that our ray will hit the ground. The rest of the code is simply toggling state booleans.

Improving efficiency in raycasting

But what if the ray didn't hit anything? That can happen in two cases—if we walk off the cliff or are performing a jump. In any case, we have to check for it.

If the ray didn't hit a collider, then obviously we are in the air and need to specify that. As this is our first check, we need to preserve our current velocity to ensure that our character doesn't drop down instantly.

Raycasting is a very handy thing and being used in many games. However, you should not rely on it too often. It is very expensive and can dramatically drop down your frame rate.

Right now, we are casting rays every frame, which is extremely inefficient. To improve our performance, we only need to cast rays when performing a jump, but never when grounded. To ensure this, we will put all our raycasting section in `FixedUpdate` to fire when the character is not grounded.

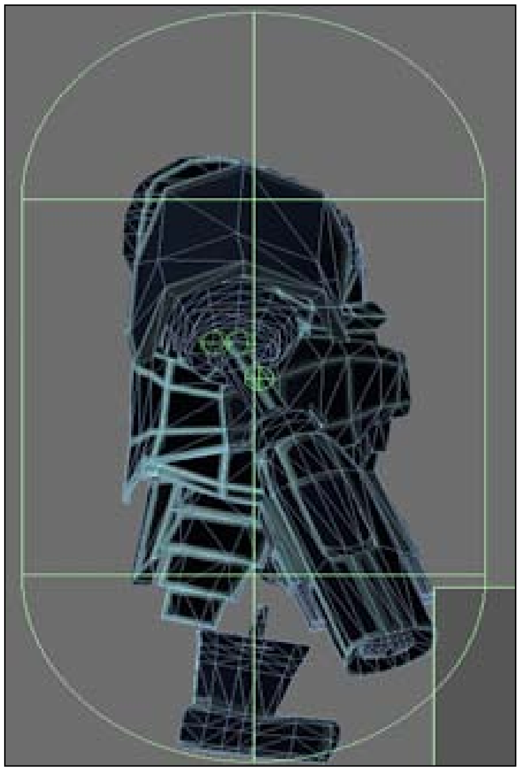
```
function FixedUpdate (){
    if (!isGrounded){
        if (Physics.Raycast(transform.position, -transform.up,
collider.height/2 + 0.2)){
            isGrounded = true;
            Jumping = false;
            inAir = false;
         }
        else if (!inAir){
            inAir = true;
            jumpDirection = MoveDirection;
         }
     }
Movement();
}
function OnCollisionExit(collisionInfo : Collision){
    isGrounded = false;
}
```

To determine if our character is not on the ground, we will use a default function—`OnCollisionExit()`. Unlike `OnControllerColliderHit()`, which had been used with Character Controller, this function is only for colliders and rigidbodies. So, whenever our character is not touching any collider or rigidbody, we will expect to be in the air, therefore, not grounded.

Let's hit **Play** and see our character jumping on our command.

Additional jump functionality

Now that we have our character jumping, there are a few issues that should be resolved. First of all, if we decide to jump on the sharp edge of the platform, we will see that our collider penetrates other colliders. Thus, our collider ends up being stuck in the wall without a chance of getting out:



A quick patch to this problem will be pushing the character away from the contact point while jumping. We will use the `OnCollisionStay()` function that's called at every frame when we are colliding with an object. This function receives collision contact information that can help us determine who we are colliding with, its velocity, name, if it has **Rigidbody**, and so on. In our case we are interested in contact points. Perform the following steps:

1. Declare a new `private` variable `contact` of a `ContactPoint` type that describes the collision point of colliding objects.
2. Declare the `OnCollisonStay` function.
3. Inside this function, we will take the first point of contact with the collider and assign it to our `private` variable.

4. Add force to the contact position to reverse the character's velocity, but only if the character is not on the ground.
5. Declare a new variable and call it `jumpClimax` of `boolean` type.

> **Contacts** is an array of all contact points.

Finally, we need to move away from that contact point by reversing our velocity. The `AddForceAtPosition` function will help us here. It is similar to the one that we used for jumping, however, this one applies force at a specified position (contact point):

```
public var jumpClimax :boolean = false;
...
function OnCollisionStay(collisionInfo : Collision){
contact = collisionInfo.contacts[0];
if (inAir || Jumping)
rigidbody.AddForceAtPosition(-rigidbody.velocity, contact.point);
}
```

The next patch will aid us in the future, when we will be adding animation to our character later in this chapter. To make sure that our jumping animation runs smoothly, we need to know when our character reaches jumping climax, in other words, when it stops going up and start a falling.

In the `FixedUpdate` function, right after the last `else if` statement, put the following code snippet:

```
else if (inAir&&rigidbody.velocity.y == 0.0) {
            jumpClimax = true;
         }
```

Nothing complex here. In theory, the moment we stop going up is a climax of our jump, that's why we check if we are in the air (obviously we can't reach jump climax when on the ground), and if vertical velocity of `rigidbody` is 0. The last part is to set our jumping climax to `false`. We'll do that at the moment when we touch the ground:

```
if (Physics.Raycast(transform.position, -transform.up, collider.
height/2 + 2)){
    isGrounded = true;
    Jumping = false;
    inAir = false;
    jumpClimax = false;
}
```

Running

We taught our character to walk, jump, and stand aimlessly on the same spot. The next logical step will be to teach him running. From a technical point of view, there is nothing too hard. Running is simply the same thing as walking, but with a greater speed. Perform the following steps:

1. Declare a new variable `IsRunning` of a type `boolean`, which will be used to determine whether our character has been told to run or not.
2. Inside the `Movement` function, at the very top, we will check if the player is pressing left or right, and shift and assign an appropriate value to `isRunning`:

```
public var isRunning : boolean = false;
...
function Movement()
{
if (Input.GetKey (KeyCode.LeftShift) || Input.GetKey (KeyCode.
RightShift))
    isRunning = true;
else
    isRunning = false;
...
}
```

> Another way to get input from the user is to use `KeyCode`. It is an enumeration for all physical keys on the keyboard. Look at the `KeyCode` script reference for a complete list of available keys, on the official website: `http://unity3d.com/support/documentation/ScriptReference/KeyCode`.

We will return to running later, in the animation section.

Cameras

They are important! It does not matter what discipline an individual is in. A camera and its uses are crucial to the development of the game and/or positions that the players will find themselves in. We will build a generic camera script that we will be able to configure for various positions.

Camera scripting

The camera script is quite heavy when it comes to scripting. So, we are going to first script functionality for the fps camera, which will form the basic structure for the other types of cameras.

It will come down to the following steps:

1. Create the camera script.
2. Write the camera switching functions.
3. Write the camera movement functionality.
4. Influence character movement through camera positioning.

Creating camera script

First, we will create a JavaScript named `CameraScr`. In that, we will have the functionalities for the reader to be able to manipulate various properties for the camera setup, such as the height that the camera will sit at and the distance from which the camera will be located from the target. In the script:

- We will set up some simple variables
- There will be a variable for the object to be tracked, another for distance, for height offset, side offset, smooth follow, and the current camera type

In the case of the object to be tracked, this will be the `GameObject` character. Make sure to set its type as `Transform` and make the variable `public`. We will use a list of variables of specified types to store multiple values of a specific type in a single variable. Be sure to use the square brackets after the following variable types and make them `public`, as we will be adding values to them in the **Inspector** menu:

- The second variable is the current camera state and will be of the type `int` and defaulted to `0`. It is okay if it is `private`.
- The third variable is for the distance, which we define as the camera distance and have its type defined as `float`.
- The fourth variable is for height and its type is again `float`. This variable will handle the height offset for the camera.

The following is an example of what these variables may look like:

```
public var charObj : Transform;
public var camNum: int = 0;
public var camDistance: float[];
public var heightOffset : float[];
```

At this point, we only care about the values that are in position `0` of the array variables.

Creating an enumeration list

Next, we will set up an enumeration that will deal with switching the values for the different camera types. An **enumeration** is a variable that can hold integer values in any form. The user just needs to keep in mind that whatever is put into an `enum`, enumeration for short, will be converted into an integer. The first value in an `enum` is considered in the first spot, the second in the second spot, and so on. Create an `enum` and call it `CamType`.

Remember that `enum` is like a class or list of variables (names in this case) and must use curly braces to begin and end its statement. Inside `enum`, create the camera types (`FP`, `SP`, `TP`). The camera types `FP`, `SP`, and `TP`, will have the variable integer values of 0, 1, and 2.

In order not to get an error at this point, you will have to create a variable, of type `enum`. Call it `CamType`. This variable allows the reader to change the `enum` type at will in the **Inspector** menu. The variable may be `private` but remember, if you wish to change the type in **Inspector**, it must be `public`. The variable and `enum` should resemble the following code snippet:

```
private var cameraType : CamType;
enum CamType {  FP, SP, TP }
```

Writing functions

For now, we have taken care of variables, and we have to begin to write the functions.

The Initialize function

We will start with giving values to our variables. Perform the following steps:

1. Right off the bat, we will need to write an `Initialize` function.
2. In this function, we want to change the camera type to the default camera type, which we want the character to start off with. In this case, it is camera 1. This is based upon the camera's value in `enum`.

3. After that, we will need to change the camera `enum` type to the first-person camera.
4. Set `charObj` with received `Player` value.

Right now, the `Initialize` function will look similar to the following code snippet:

```
function Initialize(Player : Transform){
    camNum = 1;
    cameraType = CamType.FP;
charObj = Player;
}
```

So, we have that function taken care of for the time being; next, we want another function to handle the switching of the camera.

In order to work, the `Initialize` function needs to be called. Perform the following steps:

1. Open the **CH_Controller** script.
2. Create a `public` variable `CPrefab` of a type `GameObject`.
3. Inside the `Start` function, check if this object exists.
4. If it does, set the `charObj` value to `transform`. Call the `Initialize` function with transform information about the character.
5. If it doesn't, try to find an object with a `MainCamera` tag, call the `Initialize` function and set the `charObj` value to `transform`.

This function needs to be called from `CH_Controller` just in case we forgot to set the camera. In the `CH_Controller` script, write the following code snippet:

```
public var CPrefab : GameObject;
function Start(){
...
if (CPrefab == null){
CPrefab = GameObject.FindGameObjectWithTag("MainCamera");CPr
efab.GetComponent("CameraScr").charObj = transform;CPrefab.
GetComponent("CameraScr").Initialize(transform);}
else{CPrefab.GetComponent("CameraScr").charObj = transform;CPrefab.
GetComponent("CameraScr").Initialize(transform);}
}
```

Changing camera function

The changing camera function will be called `ChangeCamType`. As its name implies, this function will change the camera from one type to another.

In this function, we need to check a couple of things, such as identify the current camera type and then change the camera to the next type. Perform the following steps:

1. First, create the `ChangeCamType` function.
2. Check for the camera number, inside of which we want to use the same camera switching line that we used in the `Initialize` function. After that line, we want to state that the camera number is now equal to the next camera type. This function should look similar to the following code snippet:

```
function ChangeCamType(){
            If(camNum == 1){
        cameraType = camType.SP;
        camNum = 2;
}
}
```

Now that we have the camera switching, we need to assign the list variable values to our equation variables. To do this, we will use a `switch case` statement to change the equation variables based upon the current camera type.

A `switch` statement is pretty much an `if` statement except that you can switch variable values without having to reassign them. This type of statement works great for **Artificial Intelligence** (**AI**) behaviors and will be used later on in the book for just that purpose.

Changing the camera values function

The changing camera values function will be called `SetCamValues`. Perform the following steps:

1. The first thing is to call the `ChangeCamType` function.
2. For the `switch` statement, the first thing that we have to check is that camera type is `true`. After that, we can use a `case` statement to switch variables based upon the current camera type. The first `case` statement will check for when the current camera is first person.

Now, we will create the equation variables. These variables will be used to hold the values from the selected array variables and in the final equations that will determine the final setup of the camera. This is done so that there can be one block of code for all of the camera types instead of a block of code for each of the camera types. Each of these variables will have the same type, minus the square brackets, of the values which they will be taking on but can be made `private` if preferred.

1. The variables to be created are as follows:
    - `camDist`: This variable will deal with camera distance list variable
    - `hOffset`: This variable will deal with height offset list variable
2. Inside the `case` statement, after the reader has matched all of the equation variables with the list variables, we need to make sure that the right list number has been assigned to the variable. For camera number 1, FP, the number is one but with lists, as in most scripting or programming languages, they start at 0. So, make sure that the list variables that the equation variables are equaling, have 0 in the brackets for FP, 1 for SP, and 2 for TP.
3. At the end of the `case` statement, we then want to put a `break` line in. This `break` line prevents the code from moving on to the next `case` statement.

In the `Initialize` function, at the end of it, we want to add this function in there as well. The following is an example of the code:

```
function SetCamValues(){
ChangeCamValues();
switch(cameraType ){
    case CamType.FP :
                camDist = camDistance[0];
                hOffset = heightOffset[0];
                break;
    }
    }
    function Initialize(Player : Transform){
    ...
    SetCameraValues();
    }
```

Writing camera switching controls

Now that the primary functionality is done, we need to write one more script that will deal with the player's input to toggle between camera types. Perform the following steps:

1. Create another JavaScript called `Player_Input`. Hook it up to camera.
2. In this script, we will need an `Update` function to always be checking for player's pressing of the toggle camera button, in this case *T*.
3. Create an `Update` function and put an `if` statement inside. This statement will check for the pressed status of *T* using Unity's built-in function—`Input.GetKeyDown(KeyCode.T)`.

4. Inside the `if` statement, we will call the `SetCameraValues` function in `CameraScr`.

   The script should resemble the following code snippet:

```
function Update(){
     If(Input.GetKeyDown(KeyCode.T){
     this.gameObject.GetComponent(CameraScr).SetCameraValues();
}
}
```

Make sure to hook up the `cameraObj` as your target camera.

Character movement and camera positioning

Now for secondary functionality of the camera, orbiting and character movement based upon camera positioning and the coding for the two other cameras. Third-person view camera is demonstrated in the following screenshot:

Updating camera type changing

First we will tackle the two other cameras as they are the easier of the two functionalities to implement. Let's venture back to the `ChangeCamType` function. In here, we only had a `change` statement for one camera. Now we need to add the other two in. Perform the following steps:

1. We just need to copy and paste the existing `if` statement two times.
2. Change the `if` statements to the `else if` statements. The camera numbers should be from 1 to 2 for the middle statement and 1 to 3 for the lower statement.
3. The `CampType` value for the middle statement should be changed from `SP` to `TP` as well and for the lower statement, `SP` to `FP`.
4. Lastly, the camera numbers found within the `if` statement blocks should be changed to `3` for the middle one and `1` for the lower one.

   These statements allow the camera to change its `enum` type whenever the *T* button is pressed. This function should now look like the following code snippet:

   ```
   function ChangeCamType(){
   if (camNum == 1){
           cameraType = camType.SP;
           camNum = 2;
   }
   if (camNum == 2){
           cameraType = camType.TP;
           camNum = 3;
   }
   if (camNum == 3){
           cameraType = camType.FP;
           camNum = 1;
   }
   }
   ```

5. Next, we will add in the `SetCameraValues` function to the `switch case` statement. All we have to do again is copy the `case` statement and change some values in the copies.
6. After copying and pasting the current `case` statement twice below the current one, we need to change `CamType` for the middle one to `SP` and the lower one to `TP`.

7. The bracket values need to change as well. For the middle statement, all bracket numbers need to be `1` and for the lower statement, all bracket numbers need to be `2`:

```
function SetCamValue(){
ChangeCamValues();
switch(cameraType ){
case CamType.FP :
            camDist = camDistance[0];
            hOffset = heightOffset[0];
            break;
case CamType.SP :
            camDist = camDistance[1];
            hOffset = heightOffset[1];
            break;
case CamType.TP :
            camDist = camDistance[2];
            hOffset = heightOffset[2];
            break;
}
}
```

Influencing camera with a mouse

The last two things to write now for the camera are mouse input for camera control and the `ClampAngle` function. First, we will add the mouse control to the `Apply` function. Perform the following steps:

1. At the bottom of the `Apply` function, we want to get the mouse X positioning and add it to the `x` angle of the camera and grab the mouse Y positioning and subtract it from the `y` angle of the camera.
2. As we want to limit the angle by which the camera can move on the Y axis, we will call the `ClampAngle` function.
3. As we wish to follow the rotation of the player, which is the Y axis, we grab the player's angle by using Unity's built-in euler angles functionality.
4. Then, we grab the camera's euler angle on Y.

> `eulerAngles` is a representation of a rotation around a specific axis. `eulerAngles.x` is, therefore, a rotation around the X axis and the same goes for the Y and Z axes.

5. After this, we want to create a variable for rotation and position.
6. The rotation variable will be equal to Quaternion Euler angles using the `x` value of the mouse, and the `y` value of the mouse.
7. Starting position of the character should be recorded when camera initializes.
8. For the positioning variable, we will take the rotation and multiply it by the camera distance and add the target object's position.
9. Lastly, we will have the camera's rotation equal to the variable rotation and the camera's position equal to the variable position.

The following code will go into the `Initialize` function and variable section of the script:

```
private var x : float = 0.0;
private var y : float = 0.0;
private var startRotation : int;
function Initialize(Player : Transform){
 camNum = 1;
 cameraType = CamType.FP;
 startRotation = charObj.transform.eulerAngles.y;
   x = transform.eulerAngles.y;
   y = transform.eulerAngles.x;
 charObj = Player;
 SetCameraValues();
}
```

The following code shows what should have been added to the bottom of the `Apply` function:

```
x += Input.GetAxis("Mouse X") * mouseSpeed[0] * Time.deltaTime;
y -= Input.GetAxis("Mouse Y") * mouseSpeed[1]  * Time.deltaTime;
y = ClampAngle(y, yLimit[0], yLimit[1]);
var targPos = Quaternion.Euler(y, x + startRotation, 0);
var position = rotation * Vector3(0.0, 0.0, camDist) + charObj.
position;

transform.rotation = rotation;
transform.position = targPos;
```

Clamping angles

The last function to write for this script is the `ClampAngle` function. It has the following characteristics:

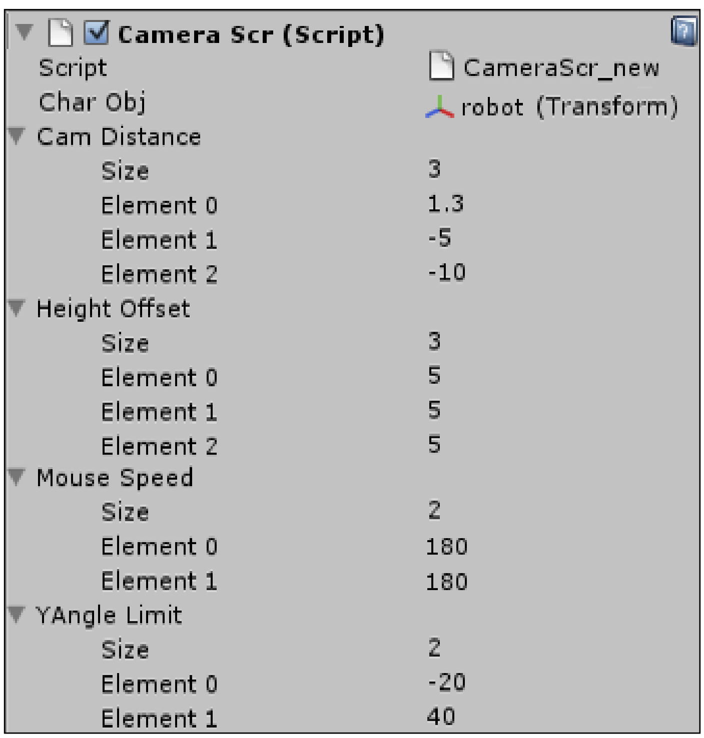
- The `ClampAngle` function is going to be taking three parameters.
- Those parameters are `angle`, `min`, and `max`.

- There will be two `if` statements in the function and then a `return` function.
- The parameters that are coming in are the angle, which we want to check and see if it is smaller or greater than 360 degrees. If greater, we subtract 360 so that the angle becomes within the acceptable range. If lower, we add 360 degrees. We return the result back to the function so that it makes sure that the angle never goes out of range.

The following is an example of the code:

```
function ClampAngle(angle:float, min:float, max:float){
    If(angle < -360)
        Angle += 360;
    If(angle > 360){
    Angle -=360;
}
return Mathf.Clamp(angle, min, max);
}
```

Now that everything is compiled, we need to go back to **Inspector** and add in the values for the list variables and the target object. These variables can be set to your own discretion but the following is a screenshot of our values:



Camera's late update

There is one more function to write for this stage of the camera and that is the `LateUpdate` function.

This function is used because during the `Update` function, the target object of the script might have moved beyond an area where the camera can see, that is, inside of a building. This function will handle the calling of the remaining functions. Perform the following steps:

1. Create the function and inside of it, do a simple check to make sure that a target exists (`charObj`).
2. Inside of this check, we want to call the `Apply` function.

   The following code snippet shows what it should look like:

   ```
   function LateUpdate(){
       If(charObj)
       Apply();
   }
   ```

Rotating character with a camera

One more function before we are done. In the `Character Controller` script, inside of the `FixedUpdate` function, right before the calling of the `Movement` function, we will add the following line of code:

```
transform.Rotate(Vector3(0, Input.GetAxis("Mouse X"), 0) * Time.
deltaTime * 250.0);
```

This line allows the character to rotate with the rotation of the camera. We grab the mouse `x` and rotate the character by it and we dampen it by the delta time to make sure that it becomes smooth gradually. The following code snippet, which shows the complete `CamerScr` script is the final code:

```
public var charObj: Transform;
public var camDistance: float[];
public var heightOffset: float[];
public var mouseSpeed : float[];
public var yAngleLimit : float[];
private var camNum : float = 0;
private var cameraType : CamType;
private var camDist : float;
private var hOffset : float;
private var x = 0.0;
private var y = 0.0;
enum CamType{FP,SP,TP}
```

```
function Initialize(Player : Transform){
    camNum = 1;
    cameraType = CamType.FP;
startRotation = charObj.transform.eulerAngles.y;
    x = transform.eulerAngles.y;
    y = transform.eulerAngles.x;
charObj = Player;
    SetCameraValues();
}
function ChangeCamType(){
        if(camNum == 1 ){
            cameraType = CamType.SP;
            camNum = 2;
        }
    else if( camNum == 2 ){
            cameraType = CamType.TP;
            camNum = 3;
        }
    else if( camNum == 3 ){
            cameraType = CamType.FP;
            camNum = 1;
        }
}
function SetCameraValues(){
    ChangeCamType();
    switch(cameraType ){
        caseCamType.FP :
            camDist = camDistance[0];
            hOffset = heightOffset[0];
            break;
        caseCamType.SP :
            camDist = camDistance[1];
            hOffset = heightOffset[1];
            break;
        case CamType.TP :
            camDist = camDistance[2];
            hOffset = heightOffset[2];
            break;
    }
}
function Apply(){
    x += Input.GetAxis("Mouse X") * mouseSpeed[0] * Time.deltaTime;
    y -= Input.GetAxis("Mouse Y") * mouseSpeed[1] * Time.deltaTime;
```

```
        y = ClampAngle(y, yAngleLimit[0], yAngleLimit[1]);\
        var rotation = Quaternion.Euler(y, x + startRotation, 0);
        var targPos = rotation * Vector3(0.0, 0.0, camDist) +
        charObj.position;
        targPos.y += hOffset;
        transform.rotation = rotation;
        transform.position = targPos;
    }
    function ClampAngle (angle : float, min : float, max : float) {
        if (angle < -360)
            angle += 360;
        if (angle > 360)
            angle -= 360;
        return Mathf.Clamp (angle, min, max);
    }
    function LateUpdate () {
            Apply();
    }
```

In the `Player_Input` script, add the following code snippet:

```
    function Update (){
        if(Input.GetKeyDown(KeyCode.T))
        this.gameObject.GetComponent(CameraScr).SetCameraValues();
    }
```

Congratulations! You can now have a camera rig that will give you a lot of functionality in a small limited package.

Animation controls

In the last part of this chapter, we will talk about what makes games look awesome—**animations**. We will learn how to control animations through code, learn the truth about the `Start` and `Awake` functions, and figure out how to make smooth transactions in between animations.

Playing simple animations

Time to add some visual indication to our movement and jump into the world of animations. Thankfully, we don't have to worry about animating our character, all animations are already done for us and are included with the model.

In this section, we will talk about basic animations and how to play them. As our game continues to grow, we will add more advanced techniques to handle various animations.

Let's create a new script whose main purpose will be to handle and control all animations for our character, such as their speed, play order, and modes. Perform the following steps:

1. Create a new script in the **Custom scripts** folder and call it **CH_Animation**.
2. Declare a `private` variable of a **CH_Controller** type (script that handles movement, if you name it differently, use your name to declare its type), call it **Controller**. This way we can reference any scripts, just by declaring them with a type of script's name.
3. Declare two functions—`Start` and `Awake`:

```
private var Controller : CH_Controller;
function Start (){}
function Awake (){}
```

Start function versus Awake function

Let's talk a bit about the difference between these two functions. At first glance, there is none, and many people make the same mistake by mismatching them. This is a mistake that can lead to problems.

The `Awake()` function is the first function that is called when you start a game. Right after you press the **Play** key, the engine goes through all scripts and executes the `Awake` function in each of them.

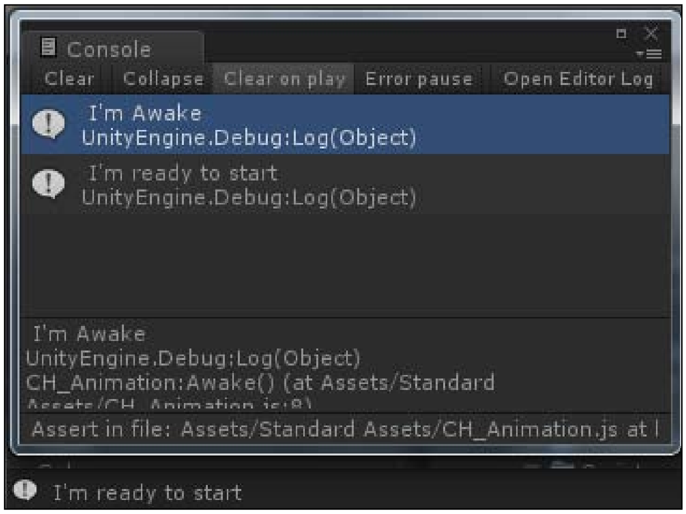
The `Start()` function is called right after all the `Awake()` functions on all objects are executed.

We can give both these functions a small test. Let's test this:

1. Add debug logs in both of these functions. In `Awake`, write something like `I'm awake` and `I'm ready to start` in the `Start` function:
2. Attach this script to our character and hit **Play**. Double-click at the debug line at the bottom and look at what we got—**I'm awake** printed before **I'm ready to start** as planned:

```
function Start (){
Debug.Log("I'm ready to start");
}
function Awake (){
Debug.Log("I'm awake");
}
```

Your console messages should be similar to those displayed on the following screenshot:



Remember, there is no order in which the engine calls the `awake` or `start` functions among the objects by default, it can randomly choose one or another and call it from there. Another interesting thing is that the `Start()` function won't be called if an object is disabled. In other words, if we disable an object in the `Awake()` function, we can save some performance for our game to run faster at start-up.

Using specifics of these functions we should be prepared to use `Awake()` for referencing objects, scripts, variables etc. Assigning default properties and start-up functionality is better in the `Start()` function.

Animation component and playing speed

We will use this script to control speed of animations and movement speed for the character; therefore, we need to declare the following variables to control them:

```
public var forwardSpeed : float = 5.0;
public var backwardSpeed : float = 3.0;
public var strafingSpeed : float = 4.0;
public var runningSpeed : float = 10.0;
public var idleAnimationSpeed : float = 1.0;
public var forwardAnimationSpeed : float = 6.0;
public var runningAnimationSpeed : float = 3.0;
public var backwardAnimationSpeed : float = 1.0;
public var strafingAnimationSpeed : float = 3.0;
public var jumpingAnimationSpeed : float = 1.5;
```

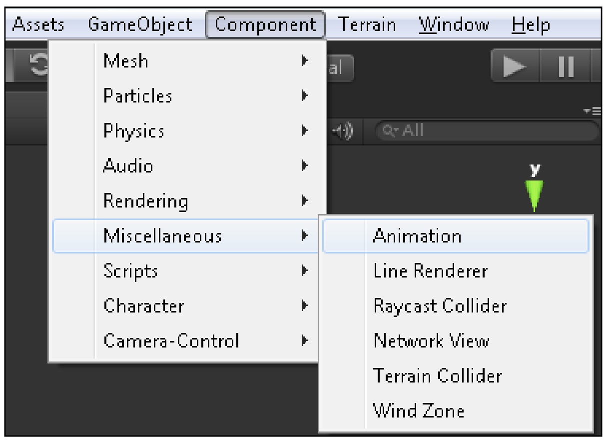
Variables in the preceding code snippet will control movement speed for our character based on direction, animation, and animation speed.

Let's get back to animations:

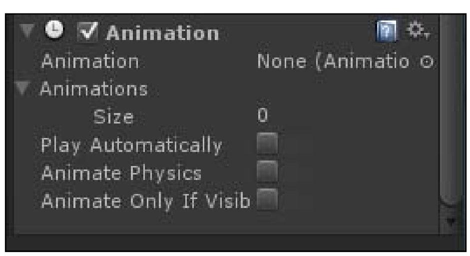
1. Remove the debug logs from this script and reference `CH_Controller` from this object in the `Awake` function:

   ```
   function Awake(){
       Controller = this.gameObject.GetComponent(CH_Controller);
   }
   ```

2. In order for the object to play animations, we need to attach an animation component to our character. Select character and go to **Component** | **Miscellaneous** | **Animation**, as shown in the following screenshot:



3. Inside **Animation Controller**, click on a small circle, it will lead you to the **Select AnimationClip** window. Click on any of the available animations.
4. Under the **Animations** drop-down menu, increase the size to **4** and assign a unique animation to each **Element**.
5. Uncheck the **Play Automatically** box. We don't want Unity to play random animation for us; we will take care of it through the code:



All the animation manipulations will be done through **Animation Controller**. The first thing that we need to learn about animations is **WrapMode**. `WrapMode` controls the play of animation—or repeating, to be more precise. There are a number of repeating modes available in Unity. They are as follows:

- **Once**: It plays the animation once and stops
- **Loop**: It plays the animation over and over again until told to stop
- **Ping-pong**: It plays the animation till the end, then reverses and plays it backwards
- **Default**: It reads a default repeat mode set higher up
- **ClampForever**: It will play the animation till the end and then continuously keeps playing its last frame

> We can specify `WrapMode` for all animations by referencing just an animation component or an individual animation, by specifying a name in square brackets:
>
> `animation.wrapMode = WrapMode.Loop;`
>
> or
>
> `animation["idle"].wrapMode = WrapMode.Loop;`

To play an animation, we simply call the **Play** function with name of the animation.

Animation scripting

In this section, we will put information learned in the preceding section into action. Perform the following steps:

1. When script initializes, we need to set **WrapMode** to **looping** by default.
2. Specify **ClampForever WrapMode** for `"jump"` animation.
3. Set speed for all known animations.
4. First animation to play should be `"idle"`.

Put the following code snippet inside the `Start` function:

```
function Start(){
animation.wrapMode = WrapMode.Loop;
animation["jump"].wrapMode = WrapMode.ClampForever;
animation["idle"].speed = idleAnimationSpeed;
animation["walk_forward"].speed = forwardAnimationSpeed;
animation["run"].speed = runningAnimationSpeed;
animation["walk_backward"].speed = backwardAnimationSpeed;
```

```
animation["walk_side"].speed = strafingAnimationSpeed;
animation["jump"].speed = jumpingAnimationSpeed;
animation.Play("idle");
}
```

Now that we have that, it's about time to add animation to our character's jump. Perform the following steps:

1. Create a new function and call it `DetermineDirection()`.
2. We will start with jumping animations; first, we need to determine if the character is in the air.
3. We will utilize `jumpClimax`, implemented earlier in this chapter, to check if the character reached a jump climax.
4. Call `DetermineDirection` function from `Update`:

```
function Update (){
DetermineDirection();
}
function DetermineDirection (){
if (Controller.inAir){
if(!Controller.jumpClimax) {}
}
}
```

Jump can be performed from any height, therefore, we have no idea how long animation should be played for. **ClampForever**, a loop playing the last frame of the animation, will help us here.

> `CrossFade` is used to blend in between animations. Blending is a very important aspect of animations, as it helps to create numerous transitions from one animation to another.

Imagine that there was no blending. Our character would be walking, then instantly changing animation to jumping, shooting, landing, and so on. That will look weird and hard-edged. If we want to make smooth transactions from one animation to another, from jumping to landing to walking, for instance, we will have to manually create numerous animations. Thankfully, Unity can blend in between animations for us, with the `CrossFade` function. `Crossfade` interpolates one basic animation into another, creating more complex and unique animations for our character to play. We can even specify a speed of fading by adding an extra float parameter, like the following one:

```
animation.CrossFade("jump", 0.3);
```

`0.3` seconds is a default value.

We will now add this functionality to our jump, right after we checked if our character didn't reach climax:

```
if(!Controller.jumpClimax){
animation.CrossFade("jump", 0.5,PlayMode.StopSameLayer);
}
```

But what if our character reached jump climax? To fix that, we need to do the same thing we did before climax, but reverse the animation with the `Rewind` function:

```
else{
animation.Rewind ("jump");
}
```

The only difference is that, once a character reaches climax, we want to reverse the animation. We will give its speed a negative value to make it play backwards; the rest of it is the same as before.

Walk, run, and idle animations

The rest of the animations are as simple as `jump` animation, so here we go.

If the character is not moving in any direction (stands on the same spot), he should be playing `idle` animation:

```
else if (Controller.MoveDirection == Vector3.zero){
        animation.CrossFade("idle");
    }
```

This script goes after the first `if` statement, at the very top. To determine whether the character is moving or not, we used the `MoveDirection` vector from `CH_Controller`.

Now we are left to deal with different movements. Realistically, we don't want our character to move with exactly the same speed in all directions. We will assign different values to the `Speed` variable in the `Controller` script based on the direction in which the character is moving:

```
else if (Controller.MoveDirection.z> 0){}
else if (Controller.MoveDirection.z< 0){}
else if (Controller.MoveDirection.x> 0 || Controller.MoveDirection.x<
0){}
```

We will use the `MoveDirection` vector to check the player's movement direction. Positive or negative **Z** axis will tell us if the character is moving forward or backwards; **X** axis controls side walk.

To play those animations we need to do three things. They are as follows:

1. Modify speed variable in `CH_Controller`.
2. Assign animation speed.
3. Crossfade the animation.

We can crossfade the animation as follows:

```
else if (Controller.MoveDirection.z> 0){
        Controller.Speed = forwardSpeed;
        animation.CrossFade("walk_forward",0.5, PlayMode.
StopSameLayer);
    }
else if (Controller.MoveDirection.z< 0){
        Controller.Speed = backwardSpeed;
        animation.CrossFade("walk_backward");
    }
else if (Controller.MoveDirection.x> 0 || Controller.MoveDirection.x<
0){
        Controller.Speed = strafingSpeed;
        animation.CrossFade("walk_side",0.5, PlayMode.StopSameLayer);
    }
```

We did exactly the same thing to every direction movement. The only exception should be forward movement. That's where we will implement running. In theory, we will check `isRunning` from `CH_Controller` and rewrite the function for moving forward as follows:

```
if (Controller.isRunning){
    Controller.Speed = runningSpeed;
    animation.CrossFade("run",0.5, PlayMode.StopSameLayer);
```

```
    }
   else{
       Controller.Speed = forwardSpeed;
       animation.CrossFade("walk_forward",0.5,PlayMode.StopSameLayer);
   }
```

The animation is now officially done.

Summary

In this chapter, we learned how to make **Rigidbody** act like a character and move around the world. We created different camera modes that a player can change by pressing a key, and touched upon animations. In the next chapter, we will teach our character to interact with objects in the world and talk about soft body projectiles that we will create for our bio gun.

3
Action Game Essentials

Welcome to the third, and probably, one of the most exiting chapters in this book! In this chapter, we will perform the following actions:

- We will cover the barebones of action game mechanics
- We will create a useable weapon that shoots soft bodies
- We will take a creative approach towards creating pickups
- We will dive deeper into the animation system and try out animation mixing techniques
- We will start creating a physical grappling hook that will make our character travel across dangerous obstacles

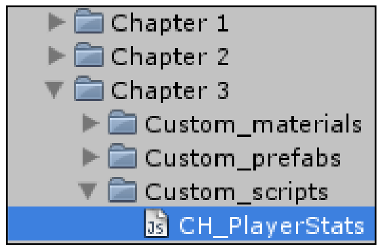
No more introductions, let's get to coding!

Programming weapons and pickables

Weapons are fun! Weapons can shoot! But there are limits to weapon functionality, aren't there? Sometimes, weapons have a cooldown between each shot, reloading when ammo in a clip is out, primary and secondary fire (a usual thing is videogames), and choice between spawning a physical bullet or casting rays in certain directions to save frame rate. Weapons can be tricky, and it's always recommended to plan ahead for the required functionality of the weapon. Pickables are easier, but can become a headache whenever we are dealing with particle effects and modifying stats.

Creating the base

Before we start programming weapons and pickables, we should create a base to store statistics and make them affect our character. Create a script called `CH_PlayerStats` and attach it to the character. Declare the following `private` variables of `int` type—`Health`, `AmmoPrime`, `AmmoAlt`, `Money`. Create enumeration called `TypeofAmmo`, as shown in the code snippet just after the following screenshot:




```
private var Health : int = 100;
private var AmmoPrime : int = 20;
private var AmmoAlt : int = 20;
private var Money : int = 0;
enum TypeofAmmo{
Prime,
Alt
};
```

Declare the `GetAmmo` and `AddAmmo` functions. To retrieve and set information we will be using enumerations:

```
function GetAmmo(Ammotype : int){
    switch (Ammotype){
        case TypeofAmmo.Prime:
            return AmmoPrime;
            break;
        case TypeofAmmo.Alt:
            return AmmoAlt;
            break;
        default:
            Debug.Log ("Wrong ammo type!");
    }
}
```

```
function AddAmmo(Ammotype : int , amount : int, modify : int){
    switch (Ammotype){
        case TypeofAmmo.Prime:
            if(modify)
            AmmoPrime += amount;
            else
            AmmoPrime = amount;
            break;
        case TypeofAmmo.Alt:
            if(modify)
            AmmoAlt += amount;
            else
            AmmoAlt = amount;
            break;
        default:
            Debug.Log ("wrong type");
    }
}
```

`AddAmmo` is asking for the type of ammo to change (`Ammotype : int`), the amount of ammo to add (`amount : int`), and if ammo needs to be modified or set (`modify : int`). We will go through the list of the `switch` statements and determine which type of ammo to add. If we send the wrong ammo type to function, the default case will tell us about it. `GetAmmo` is asking only for the ammo type that will be returned.

Next, we will declare `GetHealth` and `AddHealth`, that will work in a similar way. `AddMoney` and `GetMoney` are made in a similar way, too:

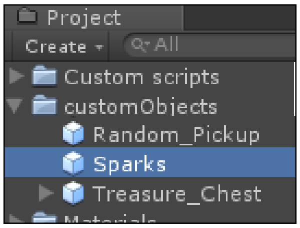
```
function GetHealth(){return Health;}
function AddHealth(amount : int, modify : int){
    if(modify)
    Health += amount;
    else
    Health = amount;
}
function GetMoney(){return Money;}
function AddMoney( amount : int ){Money += amount;}
```

Programming the weapon

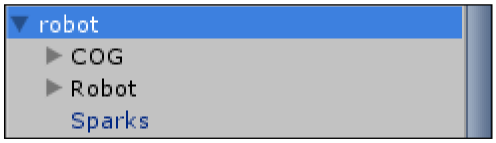
Now, we will start the interesting part—programming the weapon. Our weapon will be unusual. It will be able to shoot as an assault rifle, yes, it will also shoot exploding toxic goo that we will create with interactive cloth, but we will talk about this in future chapters. For now, let's focus on the weapon.

It so happens that a gun is already attached to a demonstrational model, so all we need to take care of is the proper particles to be emitted.

In the **customObjects** folder, you will find a prefab called **Sparks**. It will be used to emit muzzle fire for us:



Drag it to the scene and put it inside the **robot** prefab, as shown in the following screenshot:



Open the **CH_Controller** script. We have many interesting functionalities to add to it. Let's start with variables. We will need variables to store information about current state of weapon, reloading, and projectiles:

```
public var bIsShooting : boolean = false;
public var bIsShootingAlt : boolean = false;
public var Muzzle : GameObject;
public var MuzzleAlt1 : GameObject;
public var MuzzleAlt2 : GameObject;
public var Projectile : Cloth;
public var projectileSpeed : float = 20.0;
private var Stats : CH_PlayerStats;
private var counter : boolean = false;
private var countTime : float = 0;
private var canShootPrime : boolean = true;
private var canShootAlt : boolean = true;
public var flush: ParticleEmitter;
private var bWeaponEquiped : boolean = false;
```

The following list explains about the functions in the `CH_Controller` script and their uses:

`bIsShooting` and `bIsShootingAlt` will determine if the gun is currently in a shooting state. This will greatly aid us when we go into animations.

`Muzzle` will contain the location that our goo projectiles will shoot from.

`MuzzleAlt1` and `MuzzleAlt2` will contain the location that our usual bullets will shoot from.

`Projectile` is self explanatory; however, take a look at the variable type—`Cloth` that we gave to it. `InteractiveCloth` is a special type of object different from `GameObject`. We will talk more about this later in the chapter. The **Projectile** prefab can be found inside the **customObjects** folder.

`projectileSpeed` will control the speed of our projectile.

`Stats` is a reference to the `CH_PlayerStats` script that we just created to retrieve information from it.

`Counter` and `countTime` will control the reload counter. `Counter` will check if we are reloading or not and `countTime` will control the reloading time.

`canShootPrime` and `canShootAlt` will help us determine if we can shoot one type of fire or another. This is useful to be able to control animations and stop shooting when reloading.

`Flush` is the particle emitter that we just attached to the muzzle.

`bWeaponEquiped` will check if weapon is currently being equipped by character.

We will continue with the `CH_Controller` script. Declare a `Start` function. We will need to store a reference to the `CH_PlayerStats` script first. Now, we need to disable our particle emitter from emitting ahead of time. We will proceed with the `FixedUpdate` function and start creating weapon control by registering player input. Exactly the same thing will be done to register alternative fire:

```
...
function Start(){
Stats = this.gameObject.GetComponent(CH_PlayerStats);
 if (flush)
flush.emit = false;
}
function FixedUpdate(){
if (Input.GetKey (KeyCode.Mouse0) && bWeaponEquiped){

        if(canShootPrime && Stats.GetAmmo(0) > 0){
        Shooting();
        bIsShooting = true;
        }
    }
if(Input.GetKey (KeyCode.Mouse1) && bWeaponEquiped){

        if (canShootAlt && Stats.GetAmmo(1) > 0){
        bIsShootingAlt = true;
        AltShooting();
        }
if (!Input.GetKey (KeyCode.Mouse0))
    bIsShooting = false;
if (!Input.GetKey (KeyCode.Mouse1))
    bIsShootingAlt = false;
}
```

We registered the mouse button down and made sure that the weapon is currently being equipped. Toggle `bIsShooting` and `bIsShootingAlt` to `true` and go through another series of checks to determine if we can shoot with a prime fire and alternative fire (`canShootPrime` and `canShootAlt`) and have more than 0 ammo available.

> `KeyCode.Mouse0` is the left mouse button, and `KeyCode.Mouse1` is the right mouse button.

Declare the `Shooting` function; this will control everything that has to do with shooting the prime fire. Declare `AltShooting` to control alternative fire. We will also need to register that if a player is not pressing any button, then `bIsShooting` and `bIsShootingAlt` should be false.

The Shooting function

The next step will be to spawn the actual projectile that will kill enemies. To achieve this, we will instantiate a soft body projectile using the reference set by the `Projectile` variable and location specified in **Muzzle** and kick it hard so it can fly. To make sure that our `Projectile` fires when the robot points a gun at the target and not when the gun is looking down, we will use coroutines, one in particular—`WaitForSeconds`. This will allow us to postpone execution of the code for a specified number of seconds.

Coroutines are computer program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations.

We will deal with the `Shooting` function first. The first thing that needs to happen when this function is called is a start of our reload. Sounds strange indeed, but this is the way we need to do it to avoid problems with two projectiles spawning at the same time. It is better to eliminate the problem without even giving it a chance to appear.

As mentioned previously, we are creating a local variable of a `Cloth` type and instantiating it right at the muzzle position. Next, we are getting into a Rigidbody analogy in `Cloth` called `InteractiveCloth` and adding a force at a specified position in a positive **Z** direction with speed captured in the `projectileSpeed` variable.

Last, but not least, we will call the `AddAmmo` function from a `CH_PlayerStats` script and decrease the number of available ammo by one. All of it will happen in the `CH_Controller` script, after the last line of code:

```
...
function Shooting(){
canShootPrime = false;
counter = true;
```

```
yield WaitForSeconds (0.5);
var bullet : Cloth = Cloth.Instantiate(Projectile, Muzzle.transform.
position,Muzzle.transform.rotation);
bullet.transform.GetComponent(InteractiveCloth).
AddForceAtPosition(Muzzle.transform.TransformDirection(Vector3
(projectileSpeed* 10, 0, 0)), bullet.transform.position, 1.0,
ForceMode.Impulse);
Stats.AddAmmo (0, -1);
}
```

A word of warning

Do not use `Cloth` to create goo projectiles in a real project, as it will affect your performance dramatically. The example in this book is for demonstration purposes only.

Shooting cooldown

To prevent the character from shooting goo projectiles too often, we need to add a cooldown after every shot. Perform the following steps:

At the very beginning of the `FixedUpdate` function, we need to check if `counter` is true.

Increase the `countTime` variable with every second.

When `countTime` reaches `3` or more, we will set `counter` to `false`. Allow shooting with prime fire and reset `countTime`.

Add the following code snippet at the very beginning of the `FixedUpdate` function:

```
if (counter){
    countTime += Time.deltaTime;
    if (countTime >= 3.0){
        counter = false;
        canShootPrime = true;
        countTime = 0.0;
        }
    }
```

Now our gun can shoot only once in three seconds.

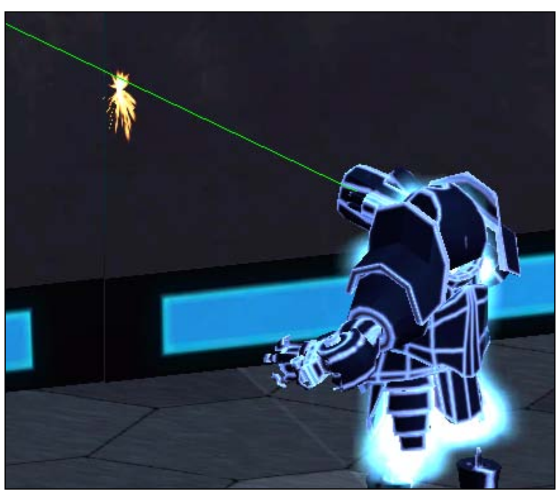
Alternative shooting function

Now that a prime shooting function is set up, we will move to alternative fire. In video games, instantiating projectiles when shooting a rocket launcher is totally fine and desirable, because we might want to show a flying rocket. But can you imagine what will happen if we decide to apply it to a machine gun?! This would be totally unacceptable, and would lead to frame rate killing. Instead, we will use raycasting, which is cheaper and faster than instantiation of a projectile. We will use raycasting for our alternative fire to fake an assault rifle.

Again, we are using a forward vector and shooting from the `MuzzleAlt1` and `MuzzleAlt2` positions with a created ray for 100 meters (it could be less if you want).

The reason to declare a new ray is to later retrieve information from colliding objects; in our case, we will need to get information from a point where the collision occurred to place sparks in that position and emit them, as shown in the screenshot just after the following code snippet. All of the following code will go after the `Shooting` function:

```
function AltShooting(){
var hit: RaycastHit;
Stats.AddAmmo(1, -1);
yield WaitForSeconds (0.5)
if (Physics.Raycast(MuzzleAlt1.transform.position, MuzzleAlt1.
transform.right, hit, 100) || Physics.Raycast(MuzzleAlt2.transform.
position, MuzzleAlt2.transform.right, hit, 100)){
flush.transform.position = hit.point;
flush.transform.rotation = Quaternion.FromToRotation(Vector3.up, hit.
normal);
flush.Emit();
}
}
```



And that's what we do in the preceding code. We place a particle in the position of a hit. Change its rotation based on the normal of the hit surface and activate it. Awesome, now we have a fully functional weapon, well… close to functional. If we try to shoot it now, we will find that we can't fully control it. To solve this problem, we will have to add a few more animations, to make our character move.

Advanced animation system

It is time for us to add additional animations to our character and teach it to hold a weapon and shoot. Open the **CH_Animation** script that takes care of all of our animations. Declare a new `public` variable called `ShootingAnimationSpeed`, which will take care of our animation speed. Next, we will go to the `DetermineDirection` function and, at the very top, check whether the player is shooting; if not we will make our character play animation, with a specified speed:

```
public var ShootingAnimationSpeed : float = 1.0;
...
function DetermineDirection(){
    if(Controller.bIsShooting ){
animation["shoot"].speed = ShootingAnimationSpeed;
animation.Play("shoot");
...
}
```

That could have been all; our character can shoot and play animation that ensures that the projectile will shoot in the right direction. There is only one small problem—if we try to walk and shoot, we will notice that our character will stop playing walking animation and will translate with shooting animation playing. That is an obvious flaw that we will fix with animation mixing.

But before we get to it, let's cover some basic theory to understand how animations work and how they affect our character.

Working of an animation

While we animate the character, we record all transforms and rotations of bones. Bones manipulate vertices to move them according to specified animation commands. But before we can transport our model to Unity, we have to perform animation baking.

> **Warning**
> Do not attempt to bake animations inside Unity. This could lead to various problems that will cause animations to break and deform.

Baking makes every bone remember the way it should be rotated and transformed over time (all transformations are done locally to the object). In other words, at every frame that animations are playing, they will have complete control over how bones transform and rotate, which is exactly what we want, if not trying to control every bone manually through code (this topic could use a book on its own, therefore will not be covered in here). Instead, we will use a trick mentioned previously—animation mixing.

Animation mixing

The theory behind animation mixing is simple and can be explained in a few sentences. Basically, we are creating new animations by slicing the original animation to be able to influence a part of a body that we need to animate. Clear? Not really? Practically, we will take an animation of shooting and transform all spine manipulations to the new animation. This way we can have animation that animates only the top part of the body, without influencing the bottom, which could be used for walking or running animation (lower part of the body – play running animation, upper part – shooting). Let's see this in action:

1. The code snippet given just after this list will go into the `Start` function, at the very top of the `CH_Animation` script.

2. Add mixing transformation to the spine bone of our character; it will now have a separate animation playing.
3. Put `idle`, `run`, `walk`, and `jump` animations at the lower layer in the `Start` function.
4. Set `WrapMode` for animations.
5. Set the playing speed for all animations.
6. Start playing the `idle` animation.

Here is the `Start` function in the `CH_Animation` script:

```
function Start (){
animation.AddClip(animation["shoot"].clip, "shootUpperBody");
animation.AddClip(animation["shoot2"].clip, "shootUpperBody2");
animation["shootUpperBody"].AddMixingTransform(transform.Find("COG/
Spine"));
animation["shootUpperBody2"].AddMixingTransform(transform.Find("COG/
Spine"));
animation["idle"].layer = -1;
animation["run"].layer = -1;
animation["jump"].layer = -1;
animation["walk_forward"].layer = -1;
animation["walk_backward"].layer = -1;
animation["walk_side"].layer = -1;
animation.wrapMode = WrapMode.Loop;
animation["jump"].wrapMode = WrapMode.ClampForever;
animation["shoot"].wrapMode = WrapMode.Once;
animation["shoot2"].wrapMode = WrapMode.Once;
animation["shootUpperBody"].wrapMode = WrapMode.Once;
animation["shootUpperBody2"].wrapMode = WrapMode.Once;
animation["idle"].speed = idleAnimationSpeed;
animation["walk_forward"].speed = forwardAnimationSpeed;
animation["run"].speed = runningAnimationSpeed;
animation["walk_backward"].speed = backwardAnimationSpeed;
animation["walk_side"].speed = strafingAnimationSpeed;
animation["jump"].speed = jumpingAnimationSpeed;
animation["shootUpperBody"].speed = ShootingAnimationSpeed;
animation["shootUpperBody2"].speed = ShootingAnimationSpeed
animation.Stop();
animation.Play("idle");
}
```

AddClip is a function within the animation component that we have attached to our character. It creates a new animation using animation["shoot"].clip as a reference, and we called it shootUpperBody. A new animation clip on its own doesn't do anything. To make it influence our character, we will add transforms to it at specific bones. Basically, we are manually specifying bones that will be animated while this clip will be playing, by using the transform.Find function that returns the object (bone in this case) from the hierarchy.

However, this is not the end of it. If we decide to play walking and upper body shooting animations at the same time, they will be in conflict, as both animations are playing and have the exact same priority. To fix this issue, Unity allows us to put animations at different layers and manually tweak the priority and influence of each layer.

By default, a higher animation layer has a higher priority of playing, and every single bone will be animated based on transforms that are specified at the highest levels. This way we can have walking and shooting animations playing at the same time without conflicting, as shooting animation doesn't affect the bottom part of the body.

As we've created animation just for shooting with upper body, we might replace all the Shoot animations by shootUpperBody; similarly, we can replace shoot2 by shootUpperBody2.

If character is shooting, we will continuously play shooting animation. If it is not shooting, we will play another animation. Change the code at the top of the DetermineDirection function in the CH_Animation script as follows:

```
if(Controller.bIsShooting ){
    if(!animation.IsPlaying("shootUpperBody"))
        animation.Play("shootUpperBody");
}
if(Controller.bIsShootingAlt ){
    if(!animation.IsPlaying("shootUpperBody2"))
        animation.Play("shootUpperBody2");
}
```

There is one more fix that we need to make in our animation script. In the DetermineDirection function, where we check if our character is moving forward, in the else statement:

```
else{
Controller.Speed = forwardSpeed;
if (!animation.IsPlaying("shootUpperBody") || !animation.
IsPlaying("shootUpperBody2"))
animation.CrossFade("walk_forward",0.5, PlayMode.StopSameLayer);
else{
 animation.CrossFade("walk_forward",0.5, PlayMode.StopSameLayer);
}
}
```

Same fix is required where we are playing the `idle` animation:

```
else if (Controller.MoveDirection == Vector3.zero &&
(animation.IsPlaying("shootUpperBody") || !animation.
IsPlaying("shootUpperBody2"))){
        animation.CrossFade("idle",0.5);
}
```

This way we are smoothly changing the animation if we stopped shooting.

Animation script overview

The following code snippet shows how the `CH_Animation` script should look by now:

```
private var Controller : CH_Controller;
public var forwardSpeed : float = 5.0;
public var backwardSpeed : float = 3.0;
public var strafingSpeed : float = 4.0;
public var runningSpeed : float = 10.0;
public var idleAnimationSpeed : float = 1.0;
public var forwardAnimationSpeed : float = 6.0;
public var runningAnimationSpeed : float = 3.0;
public var backwardAnimationSpeed : float = 1.0;
public var strafingAnimationSpeed : float = 1.0;
public var jumpingAnimationSpeed : float = 1.5;
public var ShootingAnimationSpeed : float = 5.0;
function Awake(){
Controller = this.gameObject.GetComponent("CH_Controller");
}
function Start(){
animation.AddClip(animation["shoot"].clip, "shootUpperBody");
animation.AddClip(animation["shoot2"].clip, "shootUpperBody2");
animation["shootUpperBody"].AddMixingTransform(transform.Find("COG/
Spine"));
animation["shootUpperBody2"].AddMixingTransform(transform.Find("COG/
Spine"));
animation.wrapMode = WrapMode.Loop;
animation["jump"].wrapMode = WrapMode.ClampForever;
animation["shoot"].wrapMode = WrapMode.Once;
animation["shoot2"].wrapMode = WrapMode.Once;
animation["shootUpperBody"].wrapMode = WrapMode.Once;
animation["shootUpperBody2"].wrapMode = WrapMode.Once;
animation["idle"].layer = -1;
animation["run"].layer = -1;
animation["jump"].layer = -1;
animation["walk_forward"].layer = -1;
animation["walk_backward"].layer = -1;
animation["walk_side"].layer = -1;
animation["idle"].speed = idleAnimationSpeed;
```

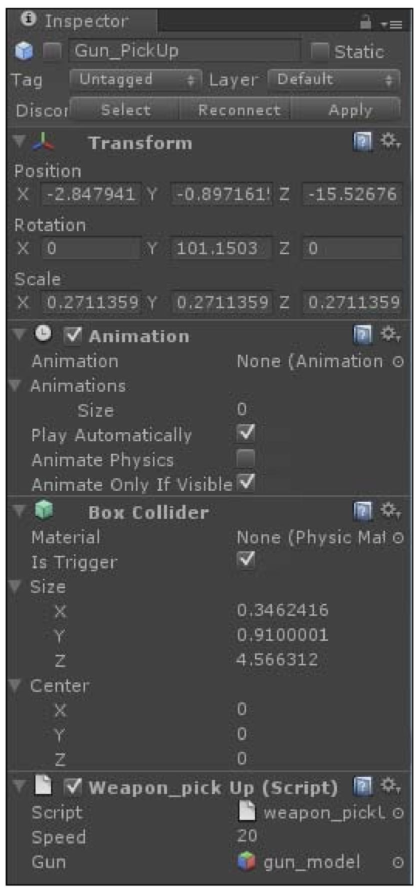
```
animation["walk_forward"].speed = forwardAnimationSpeed;
animation["run"].speed = runningAnimationSpeed;
animation["walk_backward"].speed = backwardAnimationSpeed;
animation["walk_side"].speed = strafingAnimationSpeed;
animation["jump"].speed = jumpingAnimationSpeed;
animation["shootUpperBody"].speed = ShootingAnimationSpeed;
animation["shootUpperBody2"].speed = ShootingAnimationSpeed;
animation.Stop();
animation.Play("idle");
}
function Update (){DetermineDirection();}
function DetermineDirection(){
    if(Controller.bIsShooting ){
    if(!animation.IsPlaying("shootUpperBody"))
        animation.Play("shootUpperBody");
    }
    if(Controller.bIsShootingAlt ){
    if(!animation.IsPlaying("shootUpperBody2"))
        animation.Play("shootUpperBody2");
    }
if (Controller.inAir){
        if(!Controller.jumpClimax) {
    animation.CrossFade("jump",0.5,PlayMode.StopSameLayer);
        }
        else {
        animation.Rewind("jump");
        }
    }
    else if (Controller.MoveDirection == Vector3.zero &&
    (!animation.IsPlaying("shootUpperBody") ||
    !animation.IsPlaying("shootUpperBody2"))){
        animation.CrossFade("idle",0.5);
    }
    else if (Controller.MoveDirection.z > 0){
        if (Controller.isRunning){
        Controller.Speed = runningSpeed;
        animation.CrossFade("run",0.5);
        }
        else{
        Controller.Speed = forwardSpeed;
        if (!animation.IsPlaying("shootUpperBody") ||
        !animation.IsPlaying("shootUpperBody2"))
        animation.CrossFade("walk_forward",0.5, PlayMode.
StopSameLayer);
        }
    }
    else if (Controller.MoveDirection.z < 0){
        Controller.Speed = backwardSpeed;
        animation.CrossFade("walk_backward",0.5, PlayMode.
StopSameLayer);
```

```
        }
        else if (Controller.MoveDirection.x > 0 || Controller.
    MoveDirection.x
        < 0){
            Controller.Speed = strafingSpeed;
            animation.CrossFade("walk_side",0.5, PlayMode.StopSameLayer);
        }
    }
```

We are done! Now, our character can run and shoot enemies at the same time. Next, we will talk about pickups creation.

Weapon pickup

Our next step will be programming of the weapon pickups. To begin with, create a new script called `Weapon_pickUp`. Attach it to the **Gun_PickUp** prefab and drag it to the scene.



As with most of the weapon pickups, we want it to rotate around its base and disappear when a player collides with it. Open the **Weapon_pick Up** script and declare a couple of variables:

- We need to declare a `public` variable called `Speed` of `float` type, which will be used to control weapon rotation speed around the base
- We need to make sure that we are not colliding with anything but `gameObject` with the `Player` tag
- Last is a placeholder for the function to notify the `CH_Controller` script that we have equipped the weapon and are now ready to use it

Add the following code snippet to the `Weapon_pickUp` script:

```
public var Speed : float = 20.0;
function OnTriggerEnter(other : Collider){
    if (other.gameObject.tag != "Player")
    return;
other.gameObject.GetComponent("CH_Controller").EquipWeapon();
```

Switch to the **CH_Controller** script and declare the `EquipWeapon()` function at the very bottom of the script.

Inside the `CH_Controller` script in the `EquipWeapon` function, we need to switch `boolean`, to set weapon to be equipped or not:

```
...
function EquipWeapon()
{    bWeaponEquiped = (bWeaponEquiped) ? false : true; }
...
```

> If `bWeaponEquiped = (bWeaponEquiped) ?false : true;` looks strange to you, this is how it can be interpreted:
>
> Variable = (condition) ? if true(first value) : else (second value).
>
> The other way to make that is: `bWeaponEquiped = !bWeaponEquiped`.

If `bWeaponEquiped` is `true`, then choose the first option and set it to `false`, or else set it to `true`. This new statement has to be added in several different places as follows:

- At the top of the `FixedUpdate` function, in the first `if` statement:

  ```
  if (counter && bWeaponEquiped){
  ```

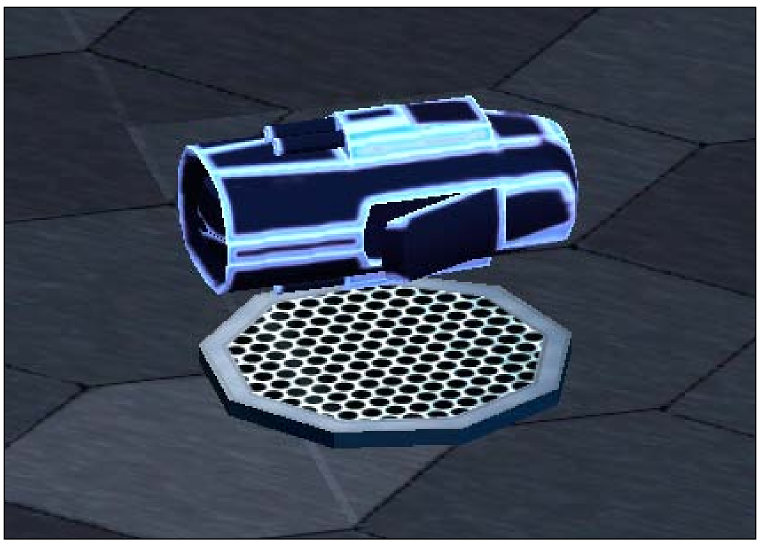
- At the end of the `FixedUpdate` function where we are registering input from the mouse:

```
if(Input.GetKey (KeyCode.Mouse0) && bWeaponEquiped){
...
if(Input.GetKey (KeyCode.Mouse1) && bWeaponEquiped){
```

- Last, but not least, we will add a rotation for our gun at the bottom of the `Weapon_pickUp` script:

```
function Update ()
{transform.Rotate(Vector3.up * Time.deltaTime * Speed); }
```

The following screenshot shows the gun pickup:

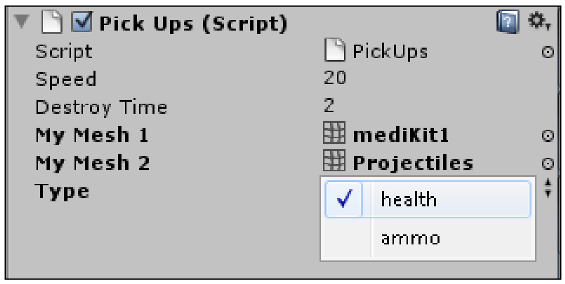


Adding ammo and health pickups

Apart from actual weapon pickup, we need to add ammo and health pickup for our character to replenish them. We don't want to create separate code for each type of pickup. That technically takes more copy-pasting skills than scripting, so we decided to show how to create a universal script to handle any type of pickups based on the string type specified for each individual instance. Go to the **custom meshes** folder and drag **Random_Pickup** prefab to the scene. Perform the following steps:

1. Create a script called `PickUps` and attach it to the prefab.

2. First, we need to specify which type of pickup this specific instance will represent; this will be done in the `Awake` function by checking the `Type` variable of a `MeshType` type. `MeshType` is enumeration; we will use it to switch pickup type in editor.



3. Next, we will detect if an object is colliding with a player or not, with the `OnTriggerEnter` function.
4. We need to make sure that the player will collide with pickup only once. Declare a new variable—`bCanCollide` of a `boolean` type.
5. If the collided object has the `Player` tag, then it must have a `CH_PlayerStats` script attached to it, which we will be referencing. Disable colliding with this object by setting `bCanCollide` to `false`.
6. Declare the `Speed` variable of `float` type that will be used to rotate pickup.
7. Now, based on the type of pickup, we want to do different things. If our pickup is `Type.health`, we will increase our character's health by 20 and increase rotating speed by 200 to make it spin very fast showing that this object had been picked up. On the other hand, if we have `Type.ammo`, we will add 10 ammo to our character and destroy pickup instantly.
8. We don't want our pickup to be there forever. Once it's picked up, it must disappear. Declare the `destroyTime` variable that will take countdown before destroying the object.
9. In the end, we will add another visual feature, which will make our pickup rotate and fly up when it's picked up.

The completed `PickUps` script should be as follows:

```
public var Speed : float = 20.0;
public var destroyTime : float = 2.0;
private var Stats : CH_PlayerStats;
private var bCanCollide : boolean = true;
```

```
public var myMesh1 : Mesh;
public var myMesh2 : Mesh;
public enum MeshType{
    health,
    ammo
};
public var Type : MeshType;
function Awake(){
    if (Type == Type.health)
    this.gameObject.GetComponent(MeshFilter).mesh = myMesh1;
    if(Type == Type.ammo)
    this.gameObject.GetComponent(MeshFilter).mesh = myMesh2;
}
function OnTriggerEnter(other:Collider){
if (other.gameObject.tag !="Player" || bCanCollide != true)
return;
bCanCollide = false;
Stats = other.gameObject.GetComponent(CH_PlayerStats);
switch (Type){
    case Type.health:
        Stats.AddHealth(20,1);
        Speed += 200;
        break;
    case Type.ammo:
        destroyTime = 0.0;
        Stats.AddAmmo(1,10,1);
    }
Destroy(this.gameObject, destroyTime);
}
function Update(){
    transform.Rotate(Vector3.up * Time.deltaTime * Speed);
    if(!bCanCollide)
    transform.Translate(Vector3.up * Time.deltaTime * Speed/100);
}
```

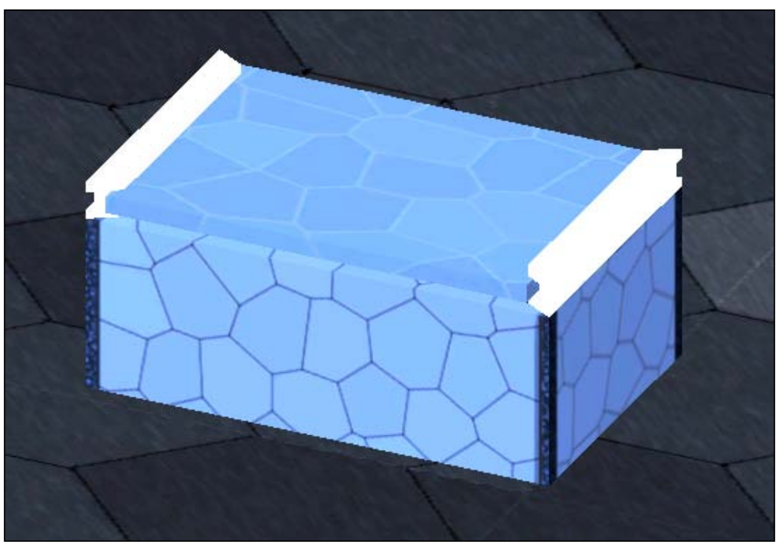
`Speed` and `destroyTime` are controlling rotation speed and the time it will take to destroy the object after it has been picked up. `Stats` is a reference to the character's `CH_PlayerStats` script. `bCanCollide` will control if we can or cannot collide with the pickup. `myMesh1` and `myMesh2` are mesh references to different mesh types that will be used to represent this pickup.

Our pickups are ready, and our character is happy! The following screenshot shows the ammo and health packages:

Creating a treasure chest

Treasure chests are the most interesting part of any game; everybody likes them. In this section, we will create a treasure chest that will be storing a specified reward for every instance.

Our treasure chest consists of three pieces—a stepping trigger, chest, and a top that will slide whenever the chest is to be opened by a player.



Go to the **custom objects** folder and drag the **treasure chest** prefab to the scene. To make our treasure chest work and generate rewards, we will have to define many variables:

```
private var bInRange : boolean = false;
private var bCanBeOpened : boolean = true;
private var bActivated : boolean = true;
private var OriginalPos : Vector3 = Vector3.zero;
private var DestinationPos :Vector3 = Vector3.zero;
public var LidSpeed : float = 3;
public var Top : GameObject;
private var Stats : CH_PlayerStats;
public enum TreasureType{
    Money,
    ammoPrime,
    ammoAlt,
    Health
};
public var treasure : TreasureType;
public var Constant : boolean = true;
public var Reward : int = 0;
public var MinRange : int = 0;
public var MaxRange : int = 0;
private var Bounty : int = 0;
public var Player : GameObject;
```

The top three variables in the preceding code snippet look similar, however, serve different purposes. The following list explains about variables declared in the preceding code:

`bInRange` checks if a player is standing in a using zone specified by our trigger. `bCanBeOpened` checks if this chest has already been used or not and `bActivated` will tell us if the chest is in the middle of sliding the top.

`OriginalPos` and `DestinationPos` are vectors that store information about the current position of a top, and destination to which it will be moved when the chest is opened. `LidSpeed` is a speed at which `Top` will be sliding and `Top` is a reference to the chest lid. `Stats`, as usual, represents a `CH_PlayerStats` script attached to the player. `treasure`, of a `TreasureType` type, will determine what type of treasure we will put inside each instance of a treasure chest.

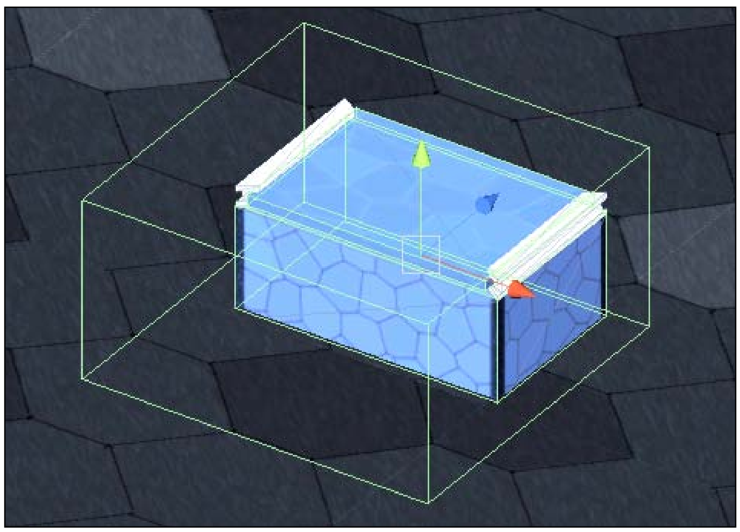
`Constant` checks if reward should be a constant number specified in the `Reward` variable. `MinRange` and `MaxRange` will be used to get a random amount of gold from a chest if we set `Constant` to `false`.

`Bounty` is a final reward that the character will receive after opening a chest.

`Player` is a reference to the controlled character.

Now that we are done with variables, let's get down to functions. They are as follows:

- In the `Start` function, we need to process all values that were given to our chest and set references.
- Then, we need the `OnTriggerEnter` and `OnTriggerExit` functions that will tell our chest if a player is standing in a zone where he can interact with it (this trigger is bigger than treasure chest itself and covers the area around it, far enough for the player to interact with it).



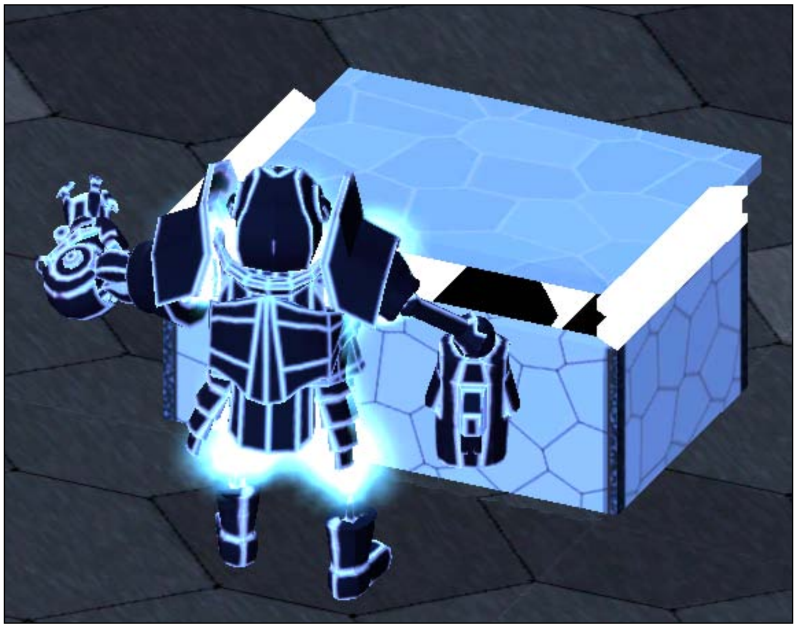
- Last is the `Update` function. The first thing that we will be checking is if a chest is active and can be used:
- Now is the best part; we will check if the player is within a chest reach and pressing an *E* button that will signify that they are opening the chest, then we will reward the player with a treasure specified in the `treasure` variable.

The completed `treasure` script is as follows:

```
function Start(){
    OriginalPos = Top.transform.position;
    DestinationPos = OriginalPos + Vector3(0,0,1);
    Stats = Player.gameObject.GetComponent(CH_PlayerStats);
    if (!Constant)
```

```
        Bounty = Random.Range (MinRange, MaxRange);
        else
        Bounty = Reward;
    }
    function OnTriggerEnter(other: Collider){
        if(other.gameObject.tag == "Player" && bCanBeOpened)
        bInRange = true;
    }
    function OnTriggerExit(other : Collider){
        if(other.gameObject.tag == "Player" && bCanBeOpened)
            bInRange = false;
    }
    function Update (){
    if (!bActivated)
    return;
    if (Top.transform.position == DestinationPos)
            bActivated = false;
    if(bInRange && Input.GetKeyDown(KeyCode.E) && bCanBeOpened){
            bCanBeOpened = false;
    switch(treasure){
                case TreasureType.Money:
                    Stats.AddMoney(Bounty);
                    break;
                case TreasureType.ammoPrime:
                    Stats.AddAmmo(0,Bounty,1);
                    break;
                case TreasureType.ammoAlt:
                    Stats.AddAmmo(1,Bounty,1);
                    break;
                case TreasureType.Health:
                    Stats.AddHealth(Bounty,1);
                    break;
                default:
                    Debug.Log("Unknown treasure is set");
            }
        }
    if(!bCanBeOpened && bActivated)
    Top.transform.position = Vector3.Lerp(Top.transform.position,
    DestinationPos, Time.deltaTime * LidSpeed);
    }
```

The final part is to make a top slide and stop when it reaches a destination. Now, we have a beautiful chest that gives us rewards for opening it.



Applying projectile fixes

Soft bodies are expensive to use, therefore, we should limit their use to a minimum, by having only two of them on the screen at a time. Perform the following steps:

1. Open a **CH_Controller** script and declare a couple of new variables:

   ```
   private var ProjectilesArray : Cloth[];
   private var aLength :int = 0;
   ```

   The first variable is an array of the **InteractiveCloth** objects and the second is a variable that controls the length of the array.
2. In the `Start` function, set `ProjectilesArray` size to `5`:

   ```
   ProjectilesArray = new Cloth[5];
   ```
3. Next, we will go to a `Shooting` function, add each created bullet to that array, and increase its length.

4. If our array grows to a limit of two, we need to delete the first member and shift the entire array to the left by one:

```
ProjectilesArray[aLength] = bullet;
aLength ++;
if (aLength> 2)
    {
    Destroy((ProjectilesArray[0]).gameObject);
        for (var i : int = 0; i<aLength; i++){
        ProjectilesArray[i] = ProjectilesArray[i+1];
        }
        ProjectilesArray[aLength] = null;
        aLength--;
    }
```

That's it! Now, the number of our projectiles will never go over two at a time.

Tethering and soft body

Finally, we get to see things interact with the world. We will see from a tether that will sway as you brush by or stick to **Rigidbody** and swing, to soft body projectiles that will deform as they are fired at a hard surface. Unity's joint systems and cloth simulations can be used for different tasks, for example, tethering, which we will talk about in this chapter.

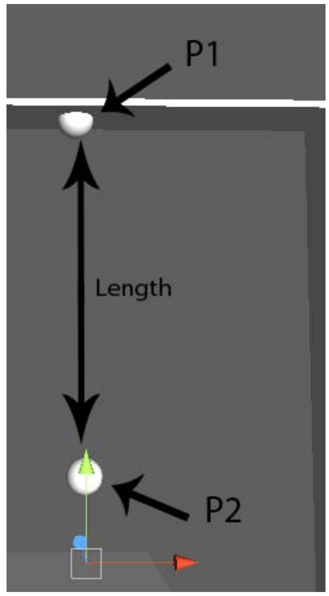
Tethering

This feature can be a main mechanic, a supporting mechanic, an environmental feature, or just plainly used as a source of entertainment for the player. With the tether that we will create, we will go see how it is created, some of the difficulties in creating one, and the end potential of having one. Tethers are really fun, so I hope you enjoy the following and are prepared to dwell into the next few pages.

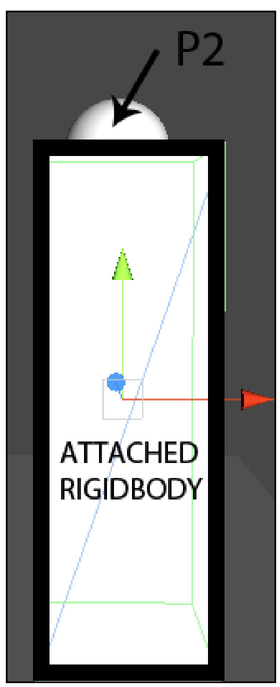
As stated in pretty much every one of the chapters, we are covering the basic fundamentals of the tools talked about in the book. Each can be explored beyond what is described and so the least we can do is point you in the right direction. For the tether, nothing else is different. We will have a tether with some basic functionality. It is by no means optimized, but will give a basis of comprehension when tackling the task of creating one. Without further deviation from the task at hand, let's script a tether.

Creating a tether

The functionality that our tether will have is the ability to create a series of **Rigidbody** links along a path determined by the placement of two points, begin and end. These points will also determine the length of the tether itself.



We will also add onto the last joint a sticky segment script which, when a Rigidbody comes into contact with it, will attach the body, allowing it to be manipulated by the tethers swaying.

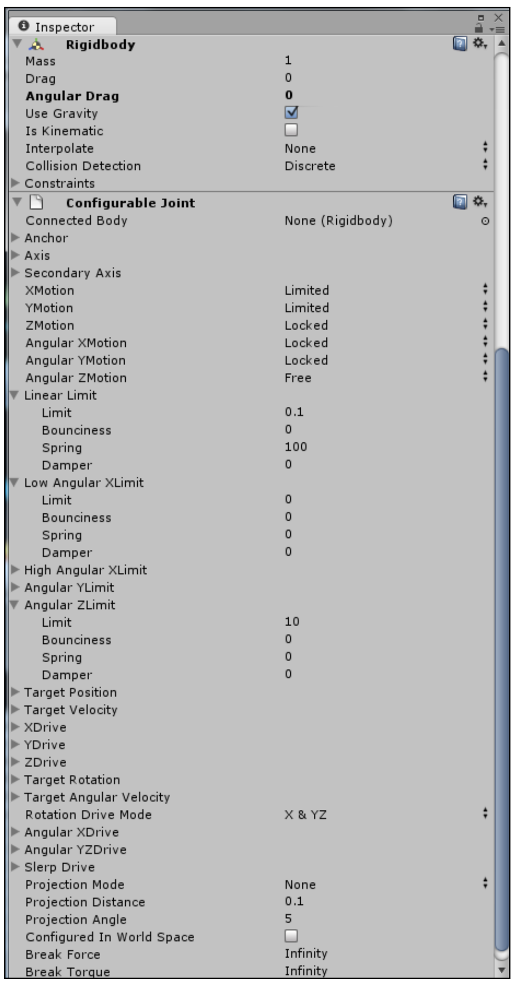


Creating assets

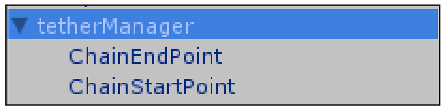
First off, we will create all the assets needed for this script to take place. We will need:

- Three spheres, of which two to represent the beginning point and endpoint of the tether and a third to represent what our tether segment will be. Call them **ChainStartPoint**, **ChainEndPoint**, and **tetherSegment**.
- One empty **gameObject** (call it **tetherManager**), which will house all aspects of the tether creation (**ChainStartPoint**, **ChainEndPoint**).

For the **ChainEndPoint** sphere, we can leave it as it is. However, for the **tetherSegment** and **ChainStartPoint** spheres, we will need to add the **Rigidbody** component and the **ConfigurableJoint** component. The following screenshot represents the values that we want for the **ConfigurableJoint** and **Rigidbody** components:



After you have created those three spheres and put the appropriate values in, we want to parent the **ChainStartPoint** and **ChainEndPoint** spheres to empty **tetherManager**. After doing so, create a prefab of it. As for the **tetherSegment** sphere, we will create a prefab of it and delete the original from the hierarchy:



Tether manager

We can start writing the tether script. This script will give us control over the tether, the mesh, which is used as the joint segment, and the creation of the tether itself. So, go ahead, create a JavaScript and call it `tetherManager`. Perform the following steps:

1. The first two variables that we will write will be `p0` to represent the beginning and `p1` to represent the end. These variables should be declared as `private` with a type of `Vector3` and set to `Vector3.zero`, as shown in the following code snippet:

```
private var p0 : Vector3 = Vector3.zero;
private var p1 : Vector3 = Vector3.zero;
```

2. In the `Awake` function, we want to set the `p0` and `p1` variables to the transform of their relative points in the world. So, for `p0`, we will find the transform of the parented begin point specified by its name and then grab its transform position. We will then do the same for `p1`. It should look similar to the following code snippet:

```
p0 = transform.Find("ChainStartPoint").transform.position;
p1 = transform.Find("ChainEndPoint").transform.position;
```

3. After these lines, we then want to find the `gameObject` instances of those points, again specified by name, and turn its renderer off using the `enabled` function and setting it to `false`. This is to make sure that at runtime, the endpoints are invisible and in the editor they can be seen:

```
gameObject.Find("ChainStartPoint").renderer.enabled = false;
gameObject.Find("ChainEndPoint").renderer.enabled = false;
```

4. After the `Awake` function, we will create the `Start` function. The `Start` function will house the creation of the tether itself. The first thing we want to check in the function is that `p0` and `p1` are in the scene and are placed. One way to do this is to check if each is not at `vector3.zero`.

5. Inside the `if` statement, the first step is to grab the tether length. This variable will be declared within the `if` statement and can be called `chainLength`. To get the length of the tether, we subtract the endpoint from the beginning point using the magnitude. After grabbing the length we want, we need to grab the number of segments. The number of segments for the tether is determined by the length of the tether divided by the distance between those segments.

We will then declare the `distanceBetweenSegments` variable at the top of the script as `public` and a `float`. It can be defaulted between 0.1 to 0.6. You can go larger but the best results come between these values.

Essentially, the greater the distance between the segments, the fewer the number of segments there will be in the tether:

```
var distanceBetweenSegments : float = 0.8;
function Start(){
    if(p0!=Vector3.zero && p1!=Vector3.zero){
var chainLength = (p1 – p0).magnitude;
var numberOfSegments = chainLength / distanceBetweenSegments;
}
}
```

First, we check to make sure that the number of segments is greater than 0 because, if it is not, then there is no need to create a tether. There are a few variables to declare inside of this `if` statement. Next, we will set the tether creation point variable to the location of the begin point. Then, we have a variable, which will hold the number of segments that will be added on to the creation segment until we reach our endpoint. The last variable to be created is our `counter`. The `counter` variable will be equal to the number of segments to be created:

```
if (numberOfSegments => 0){
var segmentCreatorPosition = p0;
var meshSegmented = (p1 - p0) / numberOfSegments;
var counter =  numberOfSegments;
}
```

Creation of tether

We have reached the tether creation. We will want to use a `while` loop here to make sure that it creates the entire tether before it goes and does anything else. The `while` loop will be controlled by the `counter` variable. As long as the `counter` is greater than `0`, we will loop through the `while` loop. Perform the following steps:

1. First off, inside the `while` loop, we want to create a new tether segment—`newSegment`. We will have a variable to hold the instantiation of the object.

2. Declare a new `meshSegment` variable of a `GameObject` type, in a variable section of the script , outside the `Start` function.
3. Instantiate `newSegment` at the tether creation point and use the default `transform.rotation` for the object's rotation, inside the `while` loop.
4. After this, we want to push the newly created segment (`newSegment`) into the array of segments. This array will be called `meshSegments` declared in a variable section as an `Array` type.
5. Back inside the `while` loop, an `if` statement here will check and see if `newSegment` has a collider and, if it does, it will check the collision variable to determine whether that collider is a trigger or not.
6. In the following `if` statement, we will set up the collision of the newly created segment. At the top, declare a variable called `useCollision` or something along those lines.
7. Next, we want the mass of the segment to be affected by the `chainMass` variable and the drag of the segment affected by the `chainDrag` variable. These two variables should be declared at the top as `public` and as `floats` so that you can change them in editor.

The following is an example of the code:

Created variables:

```
var meshSegment : GameObject = null;
var useCollision : boolean = true;
private var meshSegments : Array = new Array();
var chainMass : float = 2.0;
var chainDrag : float = 0.0;
```

`while` loop:

```
while ( counter > 0 ){
var newSegment = Instantiate( meshSegment, segmentCreatorPosition,
transform.rotation );
meshSegments.Push( newSegment );
if ( newSegment.collider ){
    if ( useCollision )
        newSegment.collider.isTrigger = false;
    else
        newSegment.collider.isTrigger = true;
    }
}
```

The following several lines will be affecting **ConfigurableJoint** on the tether.

As the beginning of each line starts the same, we will just state it now and avoid having to repeat it. We need to access **ConfigurableJoint** located on the new segments. This line will look like the following statement:

```
newSegment.GetComponent("ConfigurableJoint")
```

After `("ConfigurableJoint")`, we will be accessing different attributes of the joint. The first will be `linearLimit.spring`. This value will become equal to the `chainSpringiness` variable set up at the beginning. This variable controls how tight the tether is. The smaller this value, the less tight and more sway on the tether. This variable by default will be set to `420`.

The second attribute affected is `linearLimit.damper`. This value will be assigned to the `chainDamper` variable, which will be declared as a `float`, `public` and with a value of `0`. This variable controls how fast the tether moves. The last one will be `breakForce` and it will be equal to the `chainTolerance` variable. This variable will be defaulted to `Mathf.Infinity`. Make sure that it is `public` and has the type of `float`.

The following are the variables that need to be declared:

```
var chainSpringiness : float = 420.0;
var chainDamper : float = 0.0;
var chainTolerence : float = Mathf.Infinity;
```

So, let us take a look at what we have written inside of the `while` loop so far:

```
while( counter > 0 ){
    var newSegment = Instantiate(meshSegment, segmentCreatorPosition, 
transform.rotation);
meshSegments.Push( newSegment );
    if(newSegment.collider ){
    if(useCollision )
        newSegment.collider.isTrigger = false;
    else
        newSegment.collider.isTrigger = true;
}
newSegment.rigidbody.mass = chainMass;
newSegment.rigidbody.drag = chainDrag;
newSegment.GetComponent("ConfigurableJoint").linearLimit.spring = 
chainSpringiness;
newSegment.GetComponent("ConfigurableJoint").linearLimit.damper = 
chainDamper;
newSegment.GetComponent("ConfigurableJoint").breakForce = 
chainTolerence;
}
```

After assigning the attribute values of the `ConfigurableJoint`, we need to add a few more lines of code to the `while` loop.

The first will be another `if` statement. This statement is going to be checking to see if the first tether segment has been created. To do this, we use a variable to check if the last target has been created. The `lastTarget` variable is essentially going to hold the last segment created. If this variable is null, that means that this is indeed the first segment. Inside of this `if` statement, we will have this segment that becomes equal to another variable, which will hold the first created segment. Next, we will access the `ConfigurableJoint` attribute—`connectedBody` and connect the `rigidbody`:

```
private var firstSegment: GameObject = null;
private var lastTarget : GameObject = null;
...
while(counter > 0){
...
if ( lastTarget == null ){
firstSegment = newSegment;
newSegment.GetComponent(ConfigurableJoint).connectedBody = transform.
Find("ChainStartPoint").rigidbody;
}
```

Lastly, for this statement, we will check the `restrainStartingPoint` variable for the starting point. If it is true, we change the `isKinematic` property of the `rigidbody` to `true`. For the `else` statement, it will represent that this is not the first segment but has come afterwards. Inside of it, we will connect the new segment's `connectedBody` attribute to the last segment's `rigidbody`. This in the end creates a series of connected joints.

Following the `else` statement, have the last target variable equal to the new segment so that the last segment always equals the newly created segment. Next, have the segment creation position incremented by the mesh segmentation value. This is so that the distance between joints is in equal proportion. Lastly, decrement the counter by one.

The last line to add in this script will be to assign the stick segment script to the last joint created. As the mesh segments array is holding the created segments, we need the length of that array minus one to get the last segment in the array. Once done, we add the `StickySegment` component. If you wish to have this turned off, comment out this line of code.

```
while( counter > 0 ){
    var newSegment = Instantiate(meshSegment, segmentCreatorPosition,
transform.rotation);
meshSegments.Push( newSegment );
    if(newSegment.collider ){
```

```
        if(useCollision )
            newSegment.collider.isTrigger = false;
        else
            newSegment.collider.isTrigger = true;
    }
    newSegment.rigidbody.mass = chainMass;
    newSegment.rigidbody.drag = chainDrag;
    newSegment.GetComponent("ConfigurableJoint").linearLimit.spring =
    chainSpringiness;
    newSegment.GetComponent("ConfigurableJoint").linearLimit.damper =
    chainDamper;
    newSegment.GetComponent("ConfigurableJoint").breakForce =
    chainTolerence;
    if ( lastTarget == null ){
    firstSegment = newSegment;
    newSegment.GetComponent("ConfigurableJoint").connectedBody =
    transform.Find("ChainStartPoint").rigidbody;
        if( restrainStartingPoint)
            firstSegment.rigidbody.isKinematic = true;
    }
    else{
    newSegment.GetComponent(ConfigurableJoint).connectedBody = lastTarget.
    rigidbody;
    }

                    lastTarget = newSegment;
                    segmentCreatorPosition += meshSegmented;
                    counter--;
    }
    meshSegments[meshSegments.length - 1].AddComponent("StickySegment");
```

Now that this script is written, make sure that you have all the variables declared and, when ready, we will move on to the `StickySegment` script.

The StickySegment script

This script is very small and can be written quickly. There are two functions and no variables. The script itself will give the end joint the ability to stick to `rigidbody` that comes in contact with it. With that being said, the first function to write is `OnCollisionEnter()`. The parameter will be declared as `other` with the type of `Collision`. An `if` statement checks to make sure that the collided object has a `rigidbody` connect, and if not, then do nothing. A single line is present in the `if` statement. This line calls the `StickTo` function and has the parameter of the collided `rigidbody` objects:

```
function OnCollisionEnter( other : Collision ){
    if(other.gameObject.rigidbody)
    StickTo(other.gameObject.rigidbody );
}
```

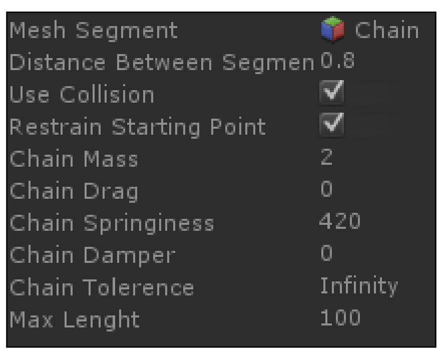
The following list explains about the functions and variables used in the `StickySegment` script:

- The `StickTo` function is next and carries the parameter of `other` as a `rigidbody` type. Inside of the function is an `if` statement checking to see if the end joint has `CharacterJoint` attached. If it does not, it proceeds. If it does, that means it is already attached to something and ends.
- Inside the `if` statement, a new variable is declared as `newStickyJoint` and it will have the `CharacterJoint` component added onto this joint.
- Afterwards, have `connectedBody` equal to that of the collided object `rigidbody`.
- Just in case the joint has been caused to become kinematic, we will set its kinematic property to off.

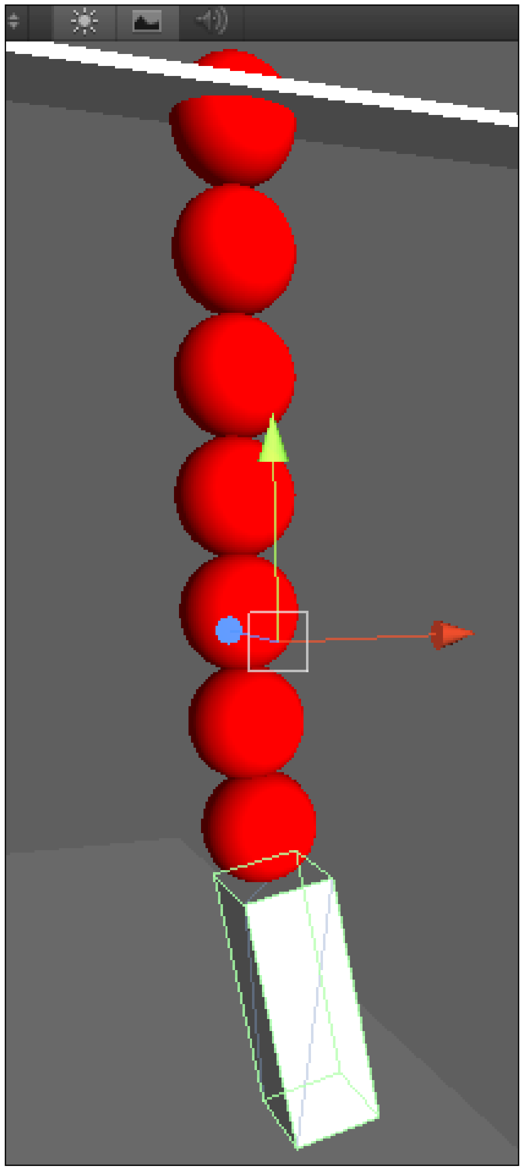
The following is the `StickTo` function:

```
function StickTo ( other : Rigidbody ){
    if(!gameObject.GetComponent(CharacterJoint))    {
        var newStickyJoint = gameObject.AddComponent(CharacterJoint);
        newStickyJoint.connectedBody = other;
    }
    if ( gameObject.rigidbody.isKinematic )
        gameObject.rigidbody.isKinematic = false;
}
```

After this script is done, go back to **Inspector** and add the **tetherManager** script to the **tetherManager** object of the tether points. Make sure that the values of the **tetherManager** are like those given in the following screenshot. Play around with them later, but for now, use these to make sure that it works. In the **Mesh Segment** variable, we will drag our mesh which will be instantiated. This is **tetherSegment**, which we had initially created at the beginning of the section.



Create any object, attach a **Rigidbody** component to it, and place it under the **ChainEndPoint** object so that they collide. Afterwards, it should be as simple as pressing **Play**. Between your beginning point and endpoint, you should see a series of segments created to represent your tether.



Congratulations! You have succeeded in the creation of the simple tether!

Tether scripts overview

The following code snippet shows how the `tetherManager` script should look by now:

```
var meshSegment :GameObject = null;
var distanceBetweenSegments : float = 0.5;
var useCollision : boolean = true;
var restrainStartingPoint : boolean = true;
var chainMass : float = 0.1;
var chainDrag : float = 0.1;
var chainSpringiness : float = 10.0;
var chainDamper : float = 1.0;
var chainTolerence : float = Mathf.Infinity;
private var meshSegments : Array = new Array();
private var firstSegment: GameObject = null;
private var lastTarget : GameObject = null;
private var p0 : Vector3 = Vector3.zero;
private var p1 : Vector3 = Vector3.zero;

function Awake(){
    p0 = transform.Find("ChainStartPoint").transform.position;
    p1 = transform.Find("ChainEndPoint").transform.position;
    gameObject.Find("ChainStartPoint").renderer.enabled = false;
    gameObject.Find("ChainEndPoint").renderer.enabled = false;
}
function Start(){
    if ( p0 != Vector3.zero && p1 !=Vector3.zero){
        var chainLength = ( p1 - p0 ).magnitude;
        var numberOfSegments = chainLength / distanceBetweenSegments;
        if ( numberOfSegments>= 0 ){
            var segmentCreatorPosition = p0;
            var meshSegmented = (p1 - p0) / numberOfSegments;
            var counter =  numberOfSegments;
            while ( counter > 0 ){
                var newSegment = Instantiate( meshSegment,
segmentCreatorPosition,
transform.rotation );
                meshSegments.Push(newSegment );
                if ( newSegment.collider ){
                    if ( useCollision )
                        newSegment.collider.isTrigger = false;
                    else
                        newSegment.collider.isTrigger = true;
                }
```

```
                newSegment.rigidbody.mass = chainMass;
                newSegment.rigidbody.drag = chainDrag;
                newSegment.GetComponent("ConfigurableJoint").
                linearLimit.spring = chainSpringiness;
                newSegment.GetComponent("ConfigurableJoint").
                linearLimit.damper = chainDamper;
                newSegment.GetComponent("ConfigurableJoint").
                breakForce =
                chainTolerence;
                if ( lastTarget == null ){
                    firstSegment = newSegment;
newSegment.GetComponent(ConfigurableJoint).connectedBody =
transform.Find(ChainStartPoint).rigidbody;
                    if(restrainStartingPoint)
                        firstSegment.rigidbody.isKinematic = true;
                }
                else{
                    newSegment.GetComponent(ConfigurableJoint).
                    connectedBody = lastTarget.rigidbody;
                }
                lastTarget = newSegment;
                segmentCreatorPosition += meshSegmented;
                counter--;
            }
            meshSegments[meshSegments.length -1].
AddComponent("StickySegment");
        }
    }
}
```

The following code snippet shows what the `StickSegment` script should look like:

```
function OnCollisionEnter( other : Collision ){
    if(other.gameObject.rigidbody)
    StickTo(other.gameObject.rigidbody );
}

function StickTo ( other : Rigidbody ){
    if(!gameObject.GetComponent(CharacterJoint))    {
        var newStickyJoint = gameObject.AddComponent(CharacterJoint);
        newStickyJoint.connectedBody = other;
    }
    if ( gameObject.rigidbody.isKinematic )
        gameObject.rigidbody.isKinematic = false;
}
```

Summary

In this chapter, we saw the breakdown of how you can create multiple-type pickups from the same `gameObject` and treasure chests that give you random amounts of the specified contents. We saw how animations can be brought into Unity and then, from there, their manipulation. We hope that you saw that animation manipulation in Unity is no small feat. Further on, tethering was explained in as simple a system that could be devised. Lastly, we have the soft body projectiles that are a lot of fun to play with. There is much functionality that can be added into each toolset and we hope that you will take what has been shown, and expand and explore it. In the next chapter, we will cover the basics of creating **role-playing games** (**RPGs**) inventory with **graphical user interface** (**GUI**).

4
Drag-and-Drop Inventory

It's time for your character to acquire its personal inventory. In this chapter, we will look into the creation of a drag-and-drop inventory that will allow us to customize our character, control his statistics and equipment, and show information about the current amount of money. In this example, we will utilize GUI to get the visuals on to the screen. However, due to multiple limitations of GUI, we can't rely on them to create drag-and-drop functionality. Therefore, we will have to recreate it ourselves. In this chapter, we will cover the following topics:

- GUI basics and the pros and cons of using them
- Draggable objects
- Working with windows and slots
- Basics of classes
- How to make inventory manipulations influence a character
- Creating a 3D character avatar
- Character customization

GUI basics

GUI stands for **Graphical User Interface** and is mainly used to get the user interface to the screen. When it comes to interactivity, GUI won't be a perfect choice, and it will give you lots of trouble if you decide to do something beyond its basic functionality. GUI elements are also located in the screen space. Let's talk about basic classes of GUI, functionality, and its uses in games.

GUI.Box

The following code snippet gives an example of the `GUI.Box` class:

```
static function Box (position : Rect, text : String) : void
static function Box (position : Rect, image : Texture) : void
static function Box (position : Rect, content : GUIContent) : void
static function Box (position : Rect, text : String, style : GUIStyle)
: void
static function Box (position : Rect, image : Texture, style :
GUIStyle) : void
static function Box (position : Rect, content : GUIContent, style :
GUIStyle) : void
```

`struct Rect` is a basic structure used by all GUI classes to mark its position on the screen. `Rect` is simple to create and use; for that we need four variables—x and y coordinates on the screen, as well as width and length of the rectangular box.

> `Rect` is just an abstract entity; it contains information about **GUI** position and doesn't render anything on the screen.

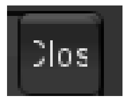
`GUI.Box` is a basic class of GUI that creates a non-interactive graphical box. We are asked to specify its position with `Rect` and can add any component such as texture or a string to be displayed in the box.

GUI.Button

`GUI.Button` is a first interactive GUI, and we will be using it in the following example.

```
static function Button (position : Rect, text : String) : boolean
static function Button (position : Rect, image : Texture) :
boolean
static function Button (position : Rect, content : GUIContent) :
boolean
static function Button (position : Rect, text : String, style :
GUIStyle) : boolean
static function Button (position : Rect, image : Texture, style :
GUIStyle) : boolean
static function Button (position : Rect, content : GUIContent, style :
GUIStyle) : boolean
```

Its creation is similar to `GUI.Box` with one major difference—the creation of `GUI.Button` is usually being put inside a `if` statement. That works because `GUI.Button` is a function with a `boolean` return value. It returns `true` if we have pressed the button and `false` if we haven't pressed it. Inside a `if` statement, we can add any functionality for this button.



`GUI.RepeatButton` is the same as `GUI.Button`, but will return `true` only if the user clicks and holds the left mouse button while hovering over the button.

GUI.Label

`GUI.Label` is a non-interactive label that can be used to display text or image at a specified position. The following code snippet gives an example of `GUI.Label`:

```
static function Label (position : Rect, text : String) : void
static function Label (position : Rect, image : Texture) : void
static function Label (position : Rect, content : GUIContent) : void
static function Label (position : Rect, text : String, style :
GUIStyle) : void
static function Label (position : Rect, image : Texture, style :
GUIStyle) : void
static function Label (position : Rect, content : GUIContent, style :
GUIStyle) : void
```

A `GUI.Label` may look as follows:

GUI.TextField

`GUI.TextField` creates areas where a user can type in one-line string. The following code snippet gives an example of a `GUI.TextField`:

```
static function TextField (position : Rect, text : String) : String
static function TextField (position : Rect, text : String, maxLength :
int) : String
```

```
static function TextField (position : Rect, text : String, style :
GUIStyle) : String
static function TextField (position : Rect, text : String, maxLength :
int, style : GUIStyle) : String
```

GUI.TextArea

`GUI.TextArea` creates areas where the user can type in multi-line string. The following code snippet gives an example of a `GUI.TextArea`:

```
static function TextArea (position : Rect, text : String) : String
static function TextArea (position : Rect, text : String, maxLength :
int) : String
static function TextArea (position : Rect, text : String, style :
GUIStyle) : String
static function TextArea (position : Rect, text : String, maxLength :
int, style : GUIStyle) : String
```

A `GUI.TextArea` may look as follows:

GUI.Toggle

`GUI.Toggle` makes a toggling button on/off. The following code snippet gives an example of `GUI.Toggle`:

```
static function Toggle (position : Rect, value : boolean, text :
String) : boolean
static function Toggle (position : Rect, value : boolean, image :
Texture) : boolean
static function Toggle (position : Rect, value : boolean, content :
GUIContent) : boolean
static function Toggle (position : Rect, value : boolean, text :
String, style : GUIStyle) : boolean
static function Toggle (position : Rect, value : boolean, image :
Texture, style : GUIStyle) : boolean
static function Toggle (position : Rect, value : boolean, content :
GUIContent, style : GUIStyle) : boolean
```

A `GUI.Toggle` may look as follows:

GUI.Toolbar and GUI.SelectionGrid

`GUI.Toolbar` creates a row of buttons and `GUI.SelectionGrid` creates rows and columns of buttons. The following code snippet gives an example of a `GUI.Toolbar`:

```
static function Toolbar (position : Rect, selected : int, texts :
string[]) : int
static function Toolbar (position : Rect, selected : int, images :
Texture[]) : int
static function Toolbar (position : Rect, selected : int, content :
GUIContent[]) : int
static function Toolbar (position : Rect, selected : int, texts :
string[], style : GUIStyle) : int
static function Toolbar (position : Rect, selected : int, images :
Texture[], style : GUIStyle) : int
static function Toolbar (position : Rect, selected : int, contents :
GUIContent[], style : GUIStyle) : int
```

The following code snippet gives an example of a `GUI.SelectionGrid`:

```
static function SelectionGrid (position : Rect, selected : int, texts
: string[], xCount : int) : int
static function SelectionGrid (position : Rect, selected : int, images
: Texture[], xCount : int) : int
static function SelectionGrid (position : Rect, selected : int,
content : GUIContent[], xCount : int) : int
static function SelectionGrid (position : Rect, selected : int, texts
: string[], xCount : int, style : GUIStyle) : int
static function SelectionGrid (position : Rect, selected : int, images
: Texture[], xCount : int, style : GUIStyle) : int
static function SelectionGrid (position : Rect, selected : int,
contents : GUIContent[], xCount : int, style : GUIStyle) : int
```

Only one of the buttons can be selected at a time and will be assigned a unique integer. This can be used as extended toggle button, where we can have more than on or off options.

GUI.HorizontalSlider and GUI.VerticalSlider

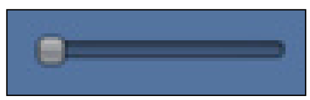
`GUI.HorizontalSlider` and `GUI.VerticalSlider` create interactive sliders. They are very useful as they allow us to withdraw integer values based on the current slider position. The following code snippet gives an example of a `GUI.HorizontalSlider`:

```
static function HorizontalSlider (position : Rect, value : float,
leftValue : float, rightValue : float) : float
static function HorizontalSlider (position : Rect, value : float,
leftValue : float, rightValue : float, slider : GUIStyle, thumb :
GUIStyle) : float
```

The following code snippet gives an example of a `GUI.VerticalSlider`:

```
static function VerticalSlider (position : Rect, value : float,
topValue : float, bottomValue : float) : float
static function VerticalSlider (position : Rect, value : float,
topValue : float, bottomValue : float, slider : GUIStyle, thumb :
GUIStyle) : float
```

A `GUI.HorizontalSlider` may look as follows:



GUI.HorizontalScrollBar and GUI.VerticalScrollBar

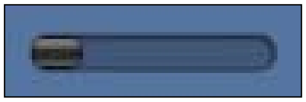
`GUI.HorizontalScrollBar` and `GUI.VerticalScrollBar` create scrollbars that can be used to scroll through documents or, in our case, inventory slots. The following code snippet gives an example of a `GUI.HorizontalScrollBar`:

```
static function HorizontalScrollbar (position : Rect, value : float,
size : float, leftValue : float, rightValue : float) : float
static function HorizontalScrollbar (position : Rect, value : float,
size : float, leftValue : float, rightValue : float, style : GUIStyle)
: float
```

The following code snippet gives an example of a `GUI.VerticalScrollBar`:

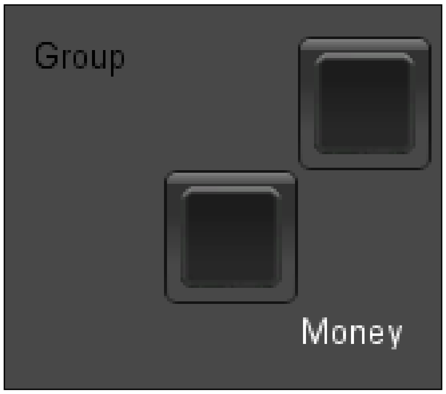
```
static function VerticalScrollbar (position : Rect, value : float,
size : float, topValue : float, bottomValue : float, style : GUIStyle)
: float
```

A GUI.HorizontalScrollBar may look as follows:



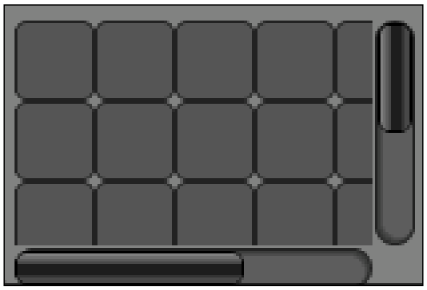
GUI.BeginGroup and GUI.EndGroup

GUI.BeginGroup and GUI.EndGroup will group together all GUIs that are put in between those two functions, and their positioning will be based on group origin and not screen origin.



GUI.BeginScrollView, GUI.EndScrollView, and ScrollTo

GUI.BeginScrollView, GUI.EndScrollView, and ScrollTo are used together to create a scrolling document with horizontal and vertical scrollbars. They will be used in our inventory example.



Other GUI classes

`GUI.Window` creates a window that calls its personal function and groups all GUIs that are being created from there. With windows, we can utilize the `GUI.DragWindow` function that will allow us to create an area where we can drag our window around.

`GUIContent` is a special class for contents of the GUI. It can contain text, image, or a tooltip, and the last one is a text that is displayed when the mouse is hovering over the GUI.

`GUILayout` is a special type of GUI that automatically lays out `GUIContent` for us. Sounds amazing, however, when it comes down to practical use, take great care, as any automatic layout can potentially be more headache than help and is therefore not recommended. To use it, simply substitute `GUI` with `GUILayout` and call any of the preceding functions.

`GUIStyle` is what makes GUI worth our time. `GUIStyle` allows us to change GUI texture based on various events such as click, button press, hover, and so on. We can directly control it through the code or to specify them in the **Inspector** view, we can even switch multiple `GUIStyle` classes for the same buttons.

There are some other GUI classes that we won't talk about here, simply because they have been created for very specific uses.

Before we start scripting, there is one thing that we should know and remember at all times when we deal with GUIs. Unlike screen origin point that is located at the bottom-left corner of the screen, GUIs are located in the top-left corner. This helps to visualize where GUIs will be created, however, it creates huge frustration when it comes to indicating GUI position based on mouse location. There are a couple of ways around that and we will look into them later in this chapter. Now let's start scripting.

Drag-and-drop inventory

Drag-and-drop inventory is a usual thing for **role playing games** (**RPGs**). In this section, we will look under the hood of the technology behind it. GUI is not the best technology for drag-and-drop functionality in Unity; it's hard to work with, and it slows down the performance. On the other hand, it is a great opportunity to showcase problems and limitations of GUI.

Basics

Perform the following steps:

1. Create a new script called `CH_Inventory` and assign it to the character.
2. Declare a `boolean` variable that will be controlling GUI rendering; call it `inventoryOpened`.
3. Declare variables that will be used to store textures, which will be used in the GUI manipulation as follows:

```
var inventoryOpened : boolean = false;
public var EmptySlotTexture : Texture;
public var ChestIcon : Texture;
public var LeftArmIcon : Texture;
public var RightWeaponIcon : Texture;
public var HeadIcon : Texture;
public var ShoulderIcon : Texture;
public var BootsIcon : Texture;
public var MedKitIcon : Texture;
public var AmmoIcon : Texture;
```

4. Declare a new function called `OnGUI()`.
5. Declare a new variable of the `Rect` type that will be used to reference and control the position of the window.
6. In the `OnGUI()` function, we will assign a value to it and create a `GUI.Window` at the same time.
7. Declare a `DoMyWindow()` function that will be called by our window.
8. Create a **GUI** button that is located at the top-right corner of the window and occupies 32 by 32 pixels.
9. Set the `inventoryOpened` variable to `false` inside that `if` statement to close the window.

In the `CH_Inventory` script, add the following code snippet:

```
var windowRect : Rect = Rect (20, 20, 200, 300);
function OnGUI () {
windowRect = GUI.Window (0, windowRect, DoMyWindow, "Inventory");
}
function DoMyWindow (windowID : int) {
if (GUI.Button(Rect(windowRect.width - 32, 0, 32, 32), "Close")){
inventoryOpened = false;
}
}
```

> `OnGUI` is a function where all GUIs should be called from (you cannot make GUI calls from anywhere else); this function is being executed every frame. It creates and checks GUI status every frame that allows creating instant response to all GUI changes.

The first thing that we are creating in our example is a window that will be used as the base for our inventory. The first argument in the window constructor is a personal ID of the window; it will be used to reference this specific window by window controlling functions. The second is a rectangular box that we declared at the top. The third is a function called by a window every time it's being rendered; we will use this function to create other GUIs that will be grouped by it. The last argument is a string that we are passing to the window to make it display that string's name.

Now that we have a window, we are creating a button to close it.

As we are calling this GUI from the window function, it will be grouped inside the window, and all coordinates that we set are based on window position. We used the width of the window and subtracted the width of the button from it to allow our button to be located in the top-right corner. When it's rendered, the word **Close** will be displayed on the button.

Inventory slots and draggable objects

Now, we will jump slightly ahead and create inventory slots and draggable objects that will be the basis of future code. We will start by creating our new classes for inventory slots and draggable objects.

> **Classes** are structures of variables and functions that were put together for convenience. We will go through classes in depth in the appendix.

We will check if the current location of the mouse is within the slot location. Remember that mouse position is different from GUI location, therefore, we need to subtract current location from overall screen height, so that they could be pointing at the same position. To make this happen, we will create classes for inventory slots and draggable objects, and establish communication between them. Perform the following steps:

1. Declare new classes called `InventorySlot` and `DraggableObject`.
2. To make our classes displayable in the **Inspector** view, we will need to add `@System.Serializable` before them.

3. In both classes, we will declare variables to identify a type of the object, or slot and icon that is assigned to it.
4. In the `InventorySlot` class, add extra variables, such as `location` of a `Vector2` type, `empty`, `Focused`, `ConstType` of a `boolean` type, and `Type` of a `String` type.
5. Declare a new function called `CheckFocus()` inside the `InventorySlot` class. The following is the code for the `InventorySlot` and `DraggableObject` classes:

```
@System.Serializable
class InventorySlot{
var icon : Texture;
var Type : String;
var location : Vector2;
var empty : boolean = true;
var Focused : boolean;
var ConstType : boolean = false;
function CheckFocus(){
    if (Input.mousePosition.y > (Screen.height - location.y -32)
    && Input.mousePosition.y < (Screen.height - location.y) &&
    Input.mousePosition.x > location.x && Input.mousePosition.x <
    (location.x + 32))
{Focused = true;}
    else
{Focused = false;}
    }
}
@System.Serializable
class DraggableObject{
var icon : Texture;
var Type : String;
}
```

This will allow us to easily communicate among them.

`location` is self-explanatory; it will help us to identify a location of the slot on the screen. `empty` will tell us if this slot is occupied or not; `Focused` will be used to check if the mouse is currently over that slot; and `ConstType` will determine if this slot is a generic inventory slot or is reserved for a specific type.

The `CheckFocus()` function will be used to check if the mouse is currently hovering over this slot and modify the `Focused` variable accordingly.

That is it for the `InventorySlot` class, so let's return to the `DraggableObject` class.

1. Declare the `LastSlot` and `HoveringSlot` variables of a `InventorySlot` type inside the `DraggableObject` class.
2. Declare a `new Array` variable called `InventorySet` that will contain the `InventorySlot` objects.
3. We will need a couple more slots that will be used to store weapons and armor for our character, and we also need to initialize our draggable object.
4. We will need a `boolean` variable that will be identifying if we are dragging something or not.
5. Declare the `Awake` function where we will set the values for slots, as well as create a fixed size for our slot array.

The following code snippet goes inside the `CH_Inventory` script:

```
...
var DraggedObject : DraggableObject;
var draggablesection : Rect = Rect (10, 10, 30, 30);
var ChestSlotLoc : Rect =Rect(5,130,32,32);
var ChestSlot : InventorySlot;
var RightWeaponSlotLoc : Rect =Rect(5,85,32,32);
var RightWeaponSlot : InventorySlot;
var LeftArmSlotLoc : Rect =Rect(160,85,32,32);
var LeftArmSlot : InventorySlot;
var HeadSlotLoc : Rect =Rect(5,40,32,32);
var HeadSlot : InventorySlot;
var ShoulderSlotLoc : Rect =Rect(160,40,32,32);
var ShoulderSlot : InventorySlot;
var BootsSlotLoc : Rect =Rect(160,130,32,32);
var BootsSlot : InventorySlot;
var InventorySet : InventorySlot[];
var dragging : boolean = false;
@System.Serializable
class DraggableObject{
var Type : String;
var LastSlot : InventorySlot;
var HoveringSlot : InventorySlot;
var icon : Texture;
}
function Awake(){
InventorySet = new Array(40);
ChestSlot.icon = ChestIcon;
ChestSlot.Type = "Chest";
```

```
ChestSlot.ConstType = true;
RightWeaponSlot.icon = RightWeaponIcon;
RightWeaponSlot.Type = "RightWeapon";
RightWeaponSlot.ConstType = true;
LeftArmSlot.icon = LeftArmIcon;
LeftArmSlot.Type = "LeftArm";
LeftArmSlot.ConstType = true;
HeadSlot.icon = HeadIcon;
HeadSlot.Type = "Head";
HeadSlot.ConstType = true;
BootsSlot.icon = BootsIcon;
BootsSlot.Type = "Boots";
BootsSlot.ConstType = true;
ShoulderSlot.icon = ShoulderIcon;
ShoulderSlot.Type = "Shoulder";
ShoulderSlot.ConstType = true;
}
```

`LastSlot` and `HoveringSlot` will be used to determine the slot that we are currently hovering over, when dragging an object; and which slot we acquired the object from so that it can be returned if we close the window or click in the empty space.

We will continue with creating draggable slots and implementing our slot array into the window:

1. Declare a new variable called `coord` of a `Vector2` type. It will be used to control the draggable object location on the screen:

   ```
   var coord :Vector2 = Vector2.zero;
   ```

2. In the `OnGUI` function, assign mouse position to the `coord` variable and subtract it's `y` value from the screen height. Inside the `OnGUI` function, add the following code lines:

   ```
   coord = Input.mousePosition;
   coord.y = Screen.height - coord.y;
   ```

3. If the dragging is on, we need to modify the draggable object location based on mouse information; we also need to create the draggable object and use the modified location as its coordinates:

   ```
   if (dragging){
   draggablesection = Rect(coord.x-15, coord.y-15, 30, 30);
   GUI.Box (draggablesection,GUIContent (DraggedObject.icon));
   }
   ```

We are using 15 pixels offset to locate the draggable object in the center of the selection, instead of dragging its corner.

Working with GUI windows

As our window is going to be draggable, we need to make sure that we keep up-to-date information about our weapon and armor slots. However, as slots are located inside the window, and all their location is based on window coordinates, we need to adjust our coordinates based on that. Thankfully, there is a function that does it for us—`GUIUtility.GUIToScreenPoint()`. It takes the position of the GUI within the group and returns its position relative to the screen. Perform the following steps:

1. Declare a new function called `DoMyWindow()`; it will handle all window manipulations.
2. Now, it is about time for us to check if these slots are being hovered by mouse.
3. There are two cases where we need to check if those slots are hovered—if we are currently dragging an object and hovering over the slot, or if we are trying to drag out an item from that slot.
4. In the first case, we will assign the current slot to the draggable object—`HoveringSlot`. In the second case, we will toggle dragging and make our current slot empty.
5. We will then do the same thing for the armor slot.
6. The next thing we need to do is to get those buttons on the screen.

The completed function in the `CH_Inventory` script should be as follows:

```
...
function DoMyWindow(windowID : int){
if (GUI.Button(Rect(windowRect.width - 32, 0, 32, 32), "Close"))
inventoryOpened = false;
ChestSlot.location = GUIUtility.GUIToScreenPoint(Vector2(ChestSlotLoc
.x, ChestSlotLoc.y));
RightWeaponSlot.location = GUIUtility.GUIToScreenPoint(Vector2(RightWe
aponSlotLoc.x, RightWeaponSlotLoc.y));
LeftArmSlot.location = GUIUtility.GUIToScreenPoint(Vector2(LeftArmSlot
Loc.x, LeftArmSlotLoc.y));
HeadSlot.location = GUIUtility.GUIToScreenPoint(Vector2(HeadSlotLoc.x,
HeadSlotLoc.y));
BootsSlot.location = GUIUtility.GUIToScreenPoint(Vector2(BootsSlotLoc
.x, BootsSlotLoc.y));
ShoulderSlot.location = GUIUtility.GUIToScreenPoint(Vector2(ShoulderSl
otLoc.x, ShoulderSlotLoc.y));
```

```
ChestSlot.CheckFocus();
RightWeaponSlot.CheckFocus();
LeftArmSlot.CheckFocus();
HeadSlot.CheckFocus();
BootsSlot.CheckFocus();
ShoulderSlot.CheckFocus();
if (ChestSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = ChestSlot;}
    else if (Event.current.type == EventType.MouseDrag && ChestSlot.
empty
    == false){
if (!dragging){
            DraggedObject.icon = ChestSlot.icon;
            DraggedObject.Type = ChestSlot.Type;
            DraggedObject.LastSlot = ChestSlot;
            ChestSlot.empty = true;
            ChestSlot.icon = ChestIcon;
                    }
        dragging = true;
    }
}
if (RightWeaponSlot.Focused){
if (dragging){DraggedObject.HoveringSlot = RightWeaponSlot;}
else if (Event.current.type == EventType.MouseDrag && RightWeaponSlot.
empty == false){
if (!dragging){
            DraggedObject.icon = RightWeaponSlot.icon;
            DraggedObject.Type = RightWeaponSlot.Type;
            DraggedObject.LastSlot = RightWeaponSlot;
            RightWeaponSlot.empty = true;
            RightWeaponSlot.icon = RightWeaponIcon;
                }
        dragging = true;
    }
}
if(LeftArmSlot.Focused){
if (dragging){DraggedObject.HoveringSlot = LeftArmSlot;}
else if (Event.current.type == EventType.MouseDrag&& LeftArmSlot.empty
== false){
if (!dragging){
            DraggedObject.icon = LeftArmSlot.icon;
            DraggedObject.Type = LeftArmSlot.Type;
            DraggedObject.LastSlot = LeftArmSlot;
            LeftArmSlot.empty = true;
            LeftArmSlot.icon = LeftArmIcon;
```

```
                    }
            dragging = true;
        }
    }
    if (HeadSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = HeadSlot;}
    else if (Event.current.type == EventType.MouseDrag && HeadSlot.empty
    == false){
    if (!dragging){
                DraggedObject.icon = HeadSlot.icon;
                DraggedObject.Type = HeadSlot.Type;
                DraggedObject.LastSlot = HeadSlot;
                HeadSlot.empty = true;
                HeadSlot.icon = HeadIcon;
                        }
            dragging = true;
        }
    }
    if (ShoulderSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = ShoulderSlot;}
    else if (Event.current.type == EventType.MouseDrag && ShoulderSlot.
    empty == false){
    if (!dragging){
                DraggedObject.icon = ShoulderSlot.icon;
                DraggedObject.Type = ShoulderSlot.Type;
                DraggedObject.LastSlot = ShoulderSlot;
                ShoulderSlot.empty = true;
                ShoulderSlot.icon = ShoulderIcon;
                }
            dragging = true;
        }
    }
    if (BootsSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = BootsSlot;}
    else if (Event.current.type == EventType.MouseDrag && BootsSlot.empty
    == false){
    if (!dragging){
                DraggedObject.icon = BootsSlot.icon;
                DraggedObject.Type = BootsSlot.Type;
                DraggedObject.LastSlot = BootsSlot;
                BootsSlot.empty = true;
                BootsSlot.icon = BootsIcon;
                    }
            dragging = true;
```

```
    }
}
if(GUI.RepeatButton(ChestSlotLoc,ChestSlot.icon)&&!dragging){}
if(GUI.RepeatButton(RightWeaponSlotLoc,RightWeaponSlot.icon)
&&!dragging){}
if(GUI.RepeatButton (LeftArmSlotLoc,LeftArmSlot.icon) && !dragging){}
if(GUI.RepeatButton (HeadSlotLoc,HeadSlot.icon)&&!dragging){}
if(GUI.RepeatButton (ShoulderSlotLoc,ShoulderSlot.icon)&& !dragging){}
if(GUI.RepeatButton(BootsSlotLoc,BootsSlot.icon)&&!dragging){}
```

As you may observe, there are a lot of `if` statements.

Lastly, we will add a `GUI.Label` to display the amount of money that the player has:

```
GUI.Label(Rect(60, 150, 100, 20), "Money: " + Stats.GetMoney());
```

Inventory slots

Having done that, we will have two constant slots that are being controlled and supervised. Now, we just need to do the same thing to our inventory slots that will serve as item holders and get them with scrolling bars. Perform the following steps:

1. We will start with rendering the scrolling bar.
2. At the variable section of the script, declare a new variable called `counter` of an `int` type, assign it to `0` below the last line and declare `scrollPosition` of a `Vector2` type.
3. Now, we will need to run two `for` loops to create rows and columns for our inventory slots.

   The rest of the functionality will be written inside the second `for` loop:

4. Now, we need to find the location of each inventory slot and store it inside the previously created `InventorySet` array. First, we will check if `InventorySlot` with a specific index number exists.
5. If `InventorySlot` really exists, our next step will be storing its location in the inside `location` variable of the specific `InventorySlot`. Remember that these slots are located based on window location.
6. If the slot is empty, we will need to assign an icon for it.
7. And it is about time to check if this slot is being focused by a mouse.
8. Check if the mouse is over this slot and the user is trying to drag the object out of the slot.

9. In the `CH_Inventory` script, continue the `DoMyWindow()` function as follows:

```
scrollPosition = GUI.BeginScrollView (Rect (0,200,200,100),
scrollPosition, Rect (0, 0, 300, 120));
counter = 0;
for (var i : int = 0; i < 10; i ++){
    for(var j: int = 0; j < 4; j++){
        if (InventorySet[counter] != null){
        InventorySet[counter].location = GUIUtility.
GUIToScreenPoint(Vector2(30 * i, 30 * j));
            if (InventorySet[counter].empty)
            {InventorySet[counter].icon = EmptySlotTexture;}
        InventorySet[counter].CheckFocus();
        if(InventorySet[counter].Focused == true && Event.current.
type ==
        EventType.MouseDrag && InventorySet[counter].empty ==
false){
        if (!dragging){
        DraggedObject.icon = InventorySet[counter].icon;
        DraggedObject.Type = InventorySet[counter].Type;
        DraggedObject.LastSlot = InventorySet[counter];
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        dragging = true;
        DraggedObject.HoveringSlot = InventorySet[counter];
}
```

If the mouse pointer was already dragging something, we would change the hovering object of the dragging object to the current slot. Otherwise, we assign all necessary information to the draggable object and make the slot empty.

The `i` and `j` variables will help us to align all slots so they will be located next to each other.

Another very interesting feature that we can add here is double-clicking. When a user double-clicks a weapon, it will check if a weapon slot is free and will automatically assign a weapon to that slot; or if the user double-clicks on **MedKit** from the inventory, it will increase the character's health. Perform the following steps to make it happen:

1. We need a couple more variables—`LastClick` of a `InventorySlot` type that will determine the last clicked slot and `ClickCount` of a `int` type that will count how many times we clicked on it.
2. To be done with variables, we need to create a reference to the `CH_Controller` and `CH_PlayerStats` scripts of our character to allow us to modify values in both of them.
3. Declare a `Start` function and reference the attached scripts.

4. If that was the first click we made on the slot, it will assign this slot to `LastClick` and increase `ClickCount`.
5. Otherwise, we will compare the `LastClick` location and current slot location, and if they are the same, assign an object to a slot it belongs to, or increase our health with **MedKit**.
6. Before we move out of this `else` statement, we need to reset `ClickCount`.
7. In the end of the `for` loop at the bottom, we need to display the actual slot and increment the counter.
8. Scroll view won't close on its own and we need to call the `GUI.EndScrollView` function to do that.
9. To finish this function, we need to close our scrolling and make the window that we are using as a base for our inventory `draggable`. This can be achieved with a `GUI.DragWindow` function inside the window function. We will also make sure that we can drag a window only if our mouse is holding it by the top.

The following code snippet goes into the variable section of the `CH_Inventory` script:

```
...
var LastClick : InventorySlot;
var ClickCount : int = 0;
var Stats : CH_PlayerStats;
var Controller : CH_Controller;
function Start(){
Controller = this.gameObject.GetComponent("CH_Controller");
Stats = this.gameObject.GetComponent("CH_PlayerStats");
}
...
```

The following code snippet is added after the last `if` statement of the `for` loop at the bottom:

```
...
else if (InventorySet[counter].Focused && Event.current.type ==
EventType.MouseDown){
if (ClickCount == 0){
LastClick = InventorySet[counter];
if (!dragging)
ClickCount++;
}
else{
```

```
    if(LastClick.location == InventorySet[counter].location &&
ClickCount
    == 1){
switch (InventorySet[counter].Type){
case "RightWeapon":
    if(RightWeaponSlot.empty){
        RightWeaponSlot.empty = false;
        RightWeaponSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        break;
case "LeftArm":
    if(LeftArmSlot.empty){
        LeftArmSlot.empty = false;
        LeftArmSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        break;
case "Head":
    if(HeadSlot.empty){
        HeadSlot.empty = false;
        HeadSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        break;
case "Shoulder":
if(ShoulderSlot.empty){
        ShoulderSlot.empty = false;
        ShoulderSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        break;
case "Boots":
if(BootsSlot.empty){
        BootsSlot.empty = false;
        BootsSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
```

```
        break;
case "Chest":
if(ChestSlot.empty){
        ChestSlot.empty = false;
        ChestSlot.icon = InventorySet[counter].icon;
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        }
        break;
case "MedKit":
        Stats.AddHealth(20,1);
        InventorySet[counter].empty = true;
        InventorySet[counter].icon = EmptySlotTexture;
        break;
        }
    }
ClickCount = 0;
}
}
else if (InventorySet[counter].Focused && dragging){
DraggedObject.HoveringSlot = InventorySet[counter];
    }
}
if (InventorySet[0] == null)
GUI.Box(Rect (30 * i, 30 * j, 30, 30), GUIContent(EmptySlotTexture));
else
GUI.Box(Rect (30 * i, 30 * j, 30, 30), GUIContent(InventorySet[count
er].icon));
counter ++;
    }
}
GUI.EndScrollView ();
GUI.DragWindow ();
```

Again, we need to make sure that the array is initialized before we start referencing it; that's why we need those extra `if` statements.

We have finished with the inventory function in the previous section, but not with the script.

Patching the inventory

Now, we will focus on dropping down objects while dragging them and checking if they have landed in the right slot, or whether they have missed it and need to be returned. Perform the following steps:

1. Return to the `OnGUI` function and after the first `if` statement, create a new one. But, this time check if we are dragging and releasing the button.
2. We need to explore two cases here if we are focused on our last hovering slot and it is empty, or anything else.
3. In the first `if` statement, we need to check if the hovering slot type is the same as the dragged object's, or if it is of a constant type, such as armor or weapon.
4. If that is the case, we will insert that object into the slot and clean the dragging object also, if the object that we dragged was a weapon and we inserted it into the weapon slot, we will equip that weapon.
5. To do this, we will create the `finishingDrag` function that will assign properties of the dragged object to the hovering object; or we will return the dragged object to the last slot if dragging isn't successful and we missed positioning the dragged object to the correct slot.
6. If that is not the case, then we simply return this object to where it used to belong.
7. We will have to copy that last `else` statement content to the `else` statement outside and set `dragging` to `false`:

This is how the `OnGUI` function should look at this point:

```
...
if(dragging && Event.current.type == EventType.MouseUp){
if (DraggedObject.HoveringSlot.Focused && DraggedObject.HoveringSlot.
empty){
if (DraggedObject.HoveringSlot.Type == DraggedObject.Type ||
DraggedObject.HoveringSlot.ConstType == false){
                switch (DraggedObject.HoveringSlot.Type){
                    case "RightWeapon":
                        Controller.EquipWeapon();
                        break;
                    case "Chest":
                        break;
                    case "Ammo":
```

```
                        break;
                    default:
                        break;
                }
            finishingDrag(true);
            }
            else{finishingDrag(false);}
        }
        else{finishingDrag(false);}
dragging = false;
}
function finishingDrag(successful : boolean){
switch (successful){
case true:
    DraggedObject.HoveringSlot.icon = DraggedObject.icon;
    DraggedObject.HoveringSlot.Type = DraggedObject.Type;
    DraggedObject.HoveringSlot.empty = false;
    break;
default:
    DraggedObject.LastSlot.icon = DraggedObject.icon;
    DraggedObject.LastSlot.Type = DraggedObject.Type;
    DraggedObject.LastSlot.empty = false;
}
    DraggedObject.LastSlot = null;
}
```

Character customization

In this next part of the chapter, we will talk about character customization and how to make our character change outfit by dragging different parts of equipment in inventory slots.

Perform the following steps:

1. Create a 3D avatar of our character to be displayed on the **Inventory** screen.
2. Add the **adding items** functionality.
3. Make our character swap outfits when we drag outfit item into the appropriate item slot.
4. Make our character take ammo from the inventory to reload (as a bonus).

3D character avatar

Creating a 3D character avatar isn't as hard as it might seem. Technically speaking, we don't need to create anything at all. The following are the steps that we will take to create it:

1. Create a new camera and locate it to show our character.
2. Put the character on a separate layer and order the camera to render that layer only.
3. Put the rendered image on the screen.
4. Make sure our image always aligns with the inventory when we are dragging it.

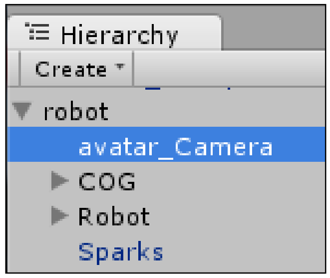
Not too hard, is it? Let's get to it.

Dealing with a camera

As mentioned previously, we need to set up our camera. Perform the following steps:

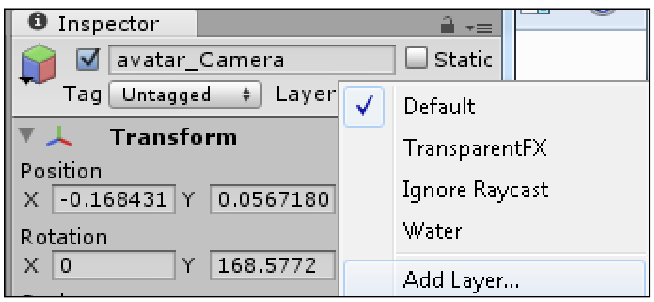
1. Create a new **Camera** object and give it a name **avatar_Camera**.
2. Locate a camera in front of the character; make sure that it doesn't cut it anywhere.

3. Attach **Camera** to the character under the **robot GameObject** in the **Hierarchy** view. This way we will make sure that, whatever happens, the camera will be facing the same direction.

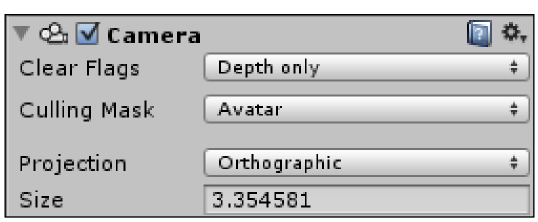


Great! Now the problem is the camera is acting just like a usual camera and rendering everything that it captures. To make the camera render our character only, we will need to put our character on a separate layer and make our camera render only that layer. Perform the following steps:

1. Having **avatar_Camera** selected, go to the **Inspector** view. Under the **Layer** drop-down menu, select **Add Layer**, as shown in the following screenshot:



2. Create a new **Layer** and call it **Avatar**.
3. Select our character and put him on **Avatar Layer**.
4. Reselect the **camera GameObject**. Go to the **Camera** component and set **Culling Mask** to **Avatar** only (deselect all other layers). **Culling Mask** will not render anything with that camera, but only the selected layer.
5. Set the **Clear Flags** value to **Depth only** to get rid of the background color.
6. Set **Projection** to **Orthographic** and adjust **Size** to fit the character in completely:



Adjusting the camera

Camera setup is almost complete, however, if you try and hit **Play** right now, you won't be able to see our **avatar_Camera** render. That is because it's being overlapped by our main camera. To prevent that from happening, we need to make sure that our **avatar_Camera** always stays on top. To do that, we need to set the **Depth** value of our camera to **100**. **Depth** is determining the layer order on the screen; the higher the value, the closer it is to the screen.

Having dealt with the depth, we have one more important thing to fix—camera screen space. It is obvious that we don't want our **avatar_Camera** to occupy the entire screen, but only a small portion of it. to make that happen, we need to adjust the **Normalized View Port Rect** values that dictate what part of the screen our camera should occupy. The **X** and **Y** values are the **Vector2D** parameters of the screen, however, they can be set between **0** and **1** to allow us to specify which percentage of the screen we want to occupy. **W** and **H** are width and height, respectively.

> Adjusting the **Normalized View Port Rect** values of two cameras can be done to create a split screen for 2 players (or more if you have more cameras).

Usually, we don't have to worry about these things and simply convert render of the second camera into a texture and apply it to any object, but this option is only available in Unity Pro. For those of you who don't have it, we will have to manually adjust the position of the render with the position of the dynamically moving inventory window. To make that happen, we will have to get back to scripting into the `CH_Inventory script`. Perform the following steps:

1. Declare a new variable to hold the camera object; call it **avatarCamera**.
2. Declare two variables of a `Vector2` type—`AvatarLoc` with a default value of `Vector2(20,-45)` and `AvatarLocation`.
3. Declare the `Update` function; it will handle the modification of render position every frame.

The following code snippet goes into the `CH_Inventory` script:

```
public var avatarCamera : Camera;
public var AvatarLocation : Vector2;
var AvatarLoc : Vector2 = Vector2(20,-45);
function Update(){}
```

To make sure that our render always moves with the window, we need to use the window position to modify the render position. The rest of the code will be written inside the `Update` function:

```
function Update(){
AvatarLocation = Vector2(windowRect.x + AvatarLoc.x, Screen.height -
windowRect.y + AvatarLoc.y  );
avatarCamera.rect = Rect ( AvatarLocation.x/Screen.width,
AvatarLocation.y/Screen.height - 0.2, 0.2, 0.2);
}
```

We are using the `AvatarLoc` value to adjust the position of the render inside the window. As GUI space has different coordinates, we need to subtract the window location from the overall screen height. All of that will give us the render location on the screen, but to adjust the render position, we need values from 0 to 1. That's why we are calculating the relation of the render position with respect to the overall screen size.

Window dragging limits

If we try to drag our window around the screen, we will notice that the render starts to shrink when we are dragging the window outside of the screen borders. Unfortunately, this cannot be helped and the only way to fix it is to make sure that our window is never being dragged outside of the screen borders. Additionally, we probably don't want our window to become draggable everywhere, that's why we need to limit, not only where it can go, but also the space where we can grab it. All the fixes will go at the very end of the `DoMyWindow` function, right before calling `GUI.DragWindow`:

```
GUI.EndScrollView ();
if(windowRect.x < 0)
windowRect.x = 0;
if(windowRect.y < 0)
windowRect.y = 0;
if (windowRect.y > Screen.height - windowRect.height)
 windowRect.y = Screen.height - windowRect.height;
 if (windowRect.x > Screen.width - windowRect.width)
 windowRect.x = Screen.width - windowRect.width;
if(Input.mousePosition.y >= Screen.height - windowRect.y-20 && Input.
mousePosition.x <= windowRect.x + windowRect.width && dragging ==
false)
GUI.DragWindow ();
```

First, we check if the window is in the wrong position and pushing it back when it should be on the screen space. Lastly, we will check the mouse position. If we are not dragging anything and pointing at the tab of the window, then we can grab and drag it.

That concludes our work with cameras.

Customization

In order to customize our character, we need an actual outfit. The easiest way to make changeable parts of the body is to attach meshes and toggle them based on the item in the slot. That way, we can have 20 weapons attached to the character, but render one at a time to save performance.

In this example we will look into something different, like changing only visuals, but not the form of robot parts. To make this happen, we will perform the following steps:

1. Set up variables and items that will be used in the inventory.
2. Add items to the inventory.
3. Make them influence the character.

Setting up items

Our outfit will be changing by changing the textures in materials that are applied to our character. Perform the following steps:

1. Declare variables for inventory slots to hold textures for items in the `CH_Inventory` script:

   ```
   public var ChestIconFull : Texture;
   public var LeftArmIconFull : Texture;
   public var RightWeaponIconFull : Texture;
   public var HeadIconFull : Texture;
   public var ShoulderIconFull : Texture;
   public var BootsIconFull : Texture;
   ```

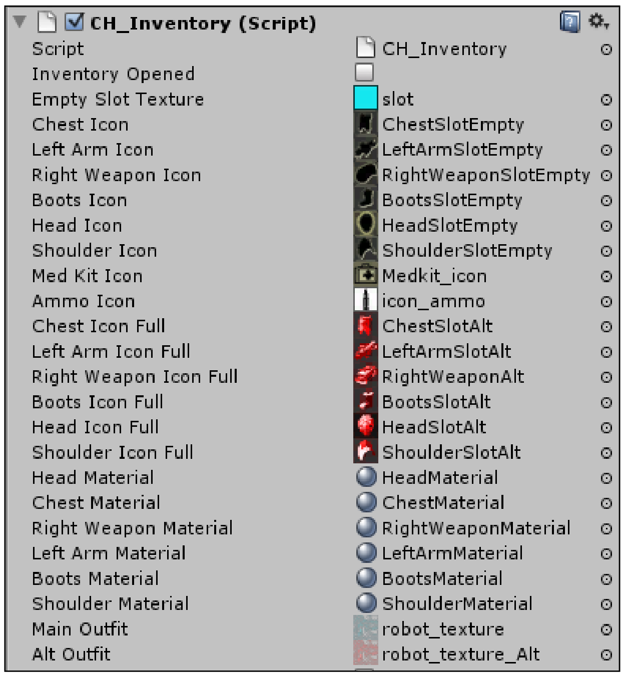
2. Declare variables to hold materials that we will need to modify:

   ```
   public var HeadMaterial : Material;
   public var ChestMaterial : Material;
   public var RightWeaponMaterial : Material;
   public var LeftArmMaterial : Material;
   public var BootsMaterial : Material;
   public var ShoulderMaterial : Material;
   ```

3. Declare variables to hold two textures that we will be modifying our materials with:

```
public var MainOutfit : Texture;
public var AltOutfit : Texture;
```

Having done that, we need to grab textures and materials from the **custom_materials** and **custom_textures** folders in the **Project** view and assign them to the variables, as shown in the following screenshot:



Adding items

To add items to the inventory, we need to create a function called `FindFirstAvailableSlot()`, which will find the first available slot and return it. To do that, we need to go through all the slots, check if they are empty, and return the last one. This function will go through all the slots in the `InventorySet` array, and it will find the first one that is empty, or return null if all are full:

In the `CH_Inventory` script, add the following code snippet:

```
...
private function FindFirstAvailableSlot(){
    for (var i : int = 0; i < 40; i ++){
        if (InventorySet[i].empty)
```

```
            return i;
            if(i == 40 && !InventorySet[i].empty)
            return null;
        }
    }
```

Now, we need a function that will take advantage of that and will add items to the inventory. All we need to pass here is a string of the object that we needs to add, when we will check it through the `if` statement and modify the slot icon and status accordingly. The following is an example of this function:

```
function AddInventory(ObjectType : String){
var emptySlot : int = FindFirstAvailableSlot();
if (ObjectType == "Chest"){
    InventorySet[emptySlot].icon = ChestIconFull;
    InventorySet[emptySlot].Type = "Chest";
    InventorySet[emptySlot].empty = false;
}
else if (ObjectType == "RightWeapon"){
    InventorySet[emptySlot].icon = RightWeaponIconFull;
    InventorySet[emptySlot].Type = "RightWeapon";
    InventorySet[emptySlot].empty = false;
    }
else if (ObjectType == "LeftArm"){
    InventorySet[emptySlot].icon = LeftArmIconFull;
    InventorySet[emptySlot].Type = "LeftArm";
    InventorySet[emptySlot].empty = false;
    }
else if (ObjectType == "Shoulder"){
    InventorySet[emptySlot].icon = ShoulderIconFull;
    InventorySet[emptySlot].Type = "Shoulder";
    InventorySet[emptySlot].empty = false;
    }
else if (ObjectType == "Head"){
    InventorySet[emptySlot].icon = HeadIconFull;
    InventorySet[emptySlot].Type = "Head";
    InventorySet[emptySlot].empty = false;
    }
else if (ObjectType == "Boots"){
    InventorySet[emptySlot].icon = BootsIconFull;
    InventorySet[emptySlot].Type = "Boots";
    InventorySet[emptySlot].empty = false;
    }
```

```
else if (ObjectType == "MedKit"){
    InventorySet[emptySlot].icon = MedKitIcon;
    InventorySet[emptySlot].Type = "MedKit";
    InventorySet[emptySlot].empty = false;
    }
else if (ObjectType == "Ammo"){
    InventorySet[emptySlot].icon = AmmoIcon;
    InventorySet[emptySlot].Type = "Ammo";
    InventorySet[emptySlot].empty = false;
}
}
```

Modifying character

As mentioned previously, we need to modify texture inside the material to change the outfit of the character. This can be done with the `SetTexture` function and specifying the type of the texture and the texture file itself. In our case, the character has two texture maps on the material—diffuse and illumination. Diffuse map sets a color of the object and illumination sets the glowing color.

Once we have changed the textures in the game, the change will remain until the next launch, as we are changing the properties of the material. To make sure that we are receiving default texture when the game loads, we need to set textures in the `Awake` function:

```
ChestMaterial.SetTexture("_MainTex", MainOutfit);
ChestMaterial.SetTexture("_Illum", MainOutfit);
RightWeaponMaterial.SetTexture("_MainTex", MainOutfit);
RightWeaponMaterial.SetTexture("_Illum", MainOutfit);
LeftArmMaterial.SetTexture("_MainTex", MainOutfit);
LeftArmMaterial.SetTexture("_Illum", MainOutfit);
HeadMaterial.SetTexture("_MainTex", MainOutfit);
HeadMaterial.SetTexture("_Illum", MainOutfit);
BootsMaterial.SetTexture("_MainTex", MainOutfit);
BootsMaterial.SetTexture("_Illum", MainOutfit);
ShoulderMaterial.SetTexture("_MainTex", MainOutfit);
ShoulderMaterial.SetTexture("_Illum", MainOutfit);
```

Next, we need to change the texture, once we are dragging the item away from the full item slot and every time we put an item in the item slot. All of that will go into the `DoMyWindow` function, which should now look like the following code snippet:

```
...
if (ChestSlot.Focused){
if (dragging){DraggedObject.HoveringSlot = ChestSlot;}
```

```
else if (Event.current.type == EventType.MouseDrag && !ChestSlot.
empty){
if (!dragging){
    DraggedObject.icon = ChestSlot.icon;
    DraggedObject.Type = ChestSlot.Type;
    DraggedObject.LastSlot = ChestSlot;
    ChestSlot.empty = true;
    ChestSlot.icon = ChestIcon;
    ChestMaterial.SetTexture("_MainTex", MainOutfit);
    ChestMaterial.SetTexture("_Illum", MainOutfit);
                    }
        dragging = true;
    }
}
if (RightWeaponSlot.Focused){
if (dragging){DraggedObject.HoveringSlot = RightWeaponSlot;}
else if (Event.current.type == EventType.MouseDrag &&
!RightWeaponSlot.empty){
if (!dragging){
    DraggedObject.icon = RightWeaponSlot.icon;
    DraggedObject.Type = RightWeaponSlot.Type;
    DraggedObject.LastSlot = RightWeaponSlot;
    RightWeaponSlot.empty = true;
    RightWeaponSlot.icon = RightWeaponIcon;
    RightWeaponMaterial.SetTexture("_MainTex", MainOutfit);
    RightWeaponMaterial.SetTexture("_Illum", MainOutfit);
                    }
        dragging = true;
    }
}
if (LeftArmSlot.Focused){
if (dragging){DraggedObject.HoveringSlot = LeftArmSlot;}
else if (Event.current.type == EventType.MouseDrag&& !LeftArmSlot.
empty){
if (!dragging){
    DraggedObject.icon = LeftArmSlot.icon;
    DraggedObject.Type = LeftArmSlot.Type;
    DraggedObject.LastSlot = LeftArmSlot;
    LeftArmSlot.empty = true;
    LeftArmSlot.icon = LeftArmIcon;
    LeftArmMaterial.SetTexture("_MainTex", MainOutfit);
    LeftArmMaterial.SetTexture("_Illum", MainOutfit);
                    }
        dragging = true;
```

```
        }
    }
    if (HeadSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = HeadSlot;}
    else if (Event.current.type == EventType.MouseDrag && !HeadSlot.empty)
    {
    if (!dragging){
        DraggedObject.icon = HeadSlot.icon;
        DraggedObject.Type = HeadSlot.Type;
        DraggedObject.LastSlot = HeadSlot;
        HeadSlot.empty = true;
        HeadSlot.icon = HeadIcon;
        HeadMaterial.SetTexture("_MainTex", MainOutfit);
        HeadMaterial.SetTexture("_Illum", MainOutfit);
                        }
            dragging = true;
        }
    }
    if (ShoulderSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = ShoulderSlot;}
    else if (Event.current.type == EventType.MouseDrag && !ShoulderSlot.
    empty){
    if (!dragging){
        DraggedObject.icon = ShoulderSlot.icon;
        DraggedObject.Type = ShoulderSlot.Type;
        DraggedObject.LastSlot = ShoulderSlot;
        ShoulderSlot.empty = true;
        ShoulderSlot.icon = ShoulderIcon;
        ShoulderMaterial.SetTexture("_MainTex", MainOutfit);
        ShoulderMaterial.SetTexture("_Illum", MainOutfit);
                        }
            dragging = true;
        }
    }
    if (BootsSlot.Focused){
    if (dragging){DraggedObject.HoveringSlot = BootsSlot;}
    else if (Event.current.type == EventType.MouseDrag && !BootsSlot.
    empty){
    if (!dragging){
        DraggedObject.icon = BootsSlot.icon;
        DraggedObject.Type = BootsSlot.Type;
        DraggedObject.LastSlot = BootsSlot;
        BootsSlot.empty = true;
        BootsSlot.icon = BootsIcon;
```

```
    BootsMaterial.SetTexture("_MainTex", MainOutfit);
    BootsMaterial.SetTexture("_Illum", MainOutfit);
                    }
        dragging = true;
    }
}
...
for (var i : int = 0; i < 10; i ++){
for(var j: int = 0; j < 4; j++){
if (InventorySet[counter] != null){
InventorySet[counter].location = GUIUtility.
GUIToScreenPoint(Vector2(30 * i, 30 * j));
if (InventorySet[counter].empty){
InventorySet[counter].icon = EmptySlotTexture;
}
InventorySet[counter].CheckFocus();
if(InventorySet[counter].Focused == true && Event.current.type ==
EventType.MouseDrag && InventorySet[counter].empty == false){
if (!dragging){
DraggedObject.icon = InventorySet[counter].icon;
DraggedObject.Type = InventorySet[counter].Type;
DraggedObject.LastSlot = InventorySet[counter];
InventorySet[counter].empty = true;
InventorySet[counter].icon = EmptySlotTexture;
}
dragging = true;
DraggedObject.HoveringSlot = InventorySet[counter];
}
else if (InventorySet[counter].Focused && Event.current.type ==
EventType.MouseDown){
    if (ClickCount == 0){
    LastClick = InventorySet[counter];
    if (!dragging)
    ClickCount++;
    }
    else{
        if(LastClick.location == InventorySet[counter].location &&
        ClickCount == 1){
            switch (InventorySet[counter].Type){
            case "RightWeapon":
                if(RightWeaponSlot.empty){
                    RightWeaponSlot.empty = false;
                    RightWeaponSlot.icon = InventorySet[counter].icon;
                    InventorySet[counter].empty = true;
                    InventorySet[counter].icon = EmptySlotTexture;
```

```
                    RightWeaponMaterial.SetTexture("_MainTex",
                    AltOutfit);
                    RightWeaponMaterial.SetTexture("_Illum",
AltOutfit);
                    }
                    break;
            case "LeftArm":
                if(LeftArmSlot.empty){
                    LeftArmSlot.empty = false;
                    LeftArmSlot.icon = InventorySet[counter].icon;
                    InventorySet[counter].empty = true;
                    InventorySet[counter].icon = EmptySlotTexture;
                    LeftArmMaterial.SetTexture("_MainTex", AltOutfit);
                    LeftArmMaterial.SetTexture("_Illum", AltOutfit);
                    }
                    break;
            case "Head":
                if(HeadSlot.empty){
                    HeadSlot.empty = false;
                    HeadSlot.icon = InventorySet[counter].icon;
                    InventorySet[counter].empty = true;
                    InventorySet[counter].icon = EmptySlotTexture;
                    HeadMaterial.SetTexture("_MainTex", AltOutfit);
                    HeadMaterial.SetTexture("_Illum", AltOutfit);
                    }
                    break;
            case "Shoulder":
                if(ShoulderSlot.empty){
                    ShoulderSlot.empty = false;
                    ShoulderSlot.icon = InventorySet[counter].icon;
                    InventorySet[counter].empty = true;
                    InventorySet[counter].icon = EmptySlotTexture;
                    ShoulderMaterial.SetTexture("_MainTex",
AltOutfit);
                    ShoulderMaterial.SetTexture("_Illum", AltOutfit);
                    }
                    break;
            case "Boots":
                if(BootsSlot.empty){
                    BootsSlot.empty = false;
                    BootsSlot.icon = InventorySet[counter].icon;
                    InventorySet[counter].empty = true;
                    InventorySet[counter].icon = EmptySlotTexture;
                        BootsMaterial.SetTexture("_MainTex",
AltOutfit);
```

```
                        BootsMaterial.SetTexture("_Illum", AltOutfit);
                        }
                        break;
                case "Chest":
                    if(ChestSlot.empty){
                        ChestSlot.empty = false;
                        ChestSlot.icon = InventorySet[counter].icon;
                        InventorySet[counter].empty = true;
                        InventorySet[counter].icon = EmptySlotTexture;
                        ChestMaterial.SetTexture("_MainTex", AltOutfit);
                        ChestMaterial.SetTexture("_Illum", AltOutfit);
                        }
                        break;
...
```

Done! now our character is fully customizable.

The last thing that we will add are defaulted items in the inventory. For that, we will go into the `Start` function and add the following code at the very bottom:

```
yield WaitForSeconds (2);
AddInventory("RightWeapon");
AddInventory("Head");
AddInventory("Shoulder");
AddInventory("LeftArm");
AddInventory("Boots");
AddInventory("Chest");
AddInventory("Ammo");
```

The following screenshot shows the **Inventory** screen:

Reloading and inventory

Now that we can have ammo in the inventory, we also need to make use of it. The idea is that every time we run out of ammo, we send an issue to the inventory and search for an ammo item. If we find an item, we destroy an item in the inventory and replenish our ammo.

First of all, let's create a `FindAmmo` function in the `CH_Inventory` script that will be searching for the ammo item in the inventory (this function is similar to the `FindFirstAvailableSlot` function that we wrote earlier). The following is an example of the `FindAmmo` function:

```
public function FindAmmo(){
    for (var i : int = 0; i < 40; i ++){
        if (InventorySet[i].Type == "Ammo"){
        InventorySet[i].icon = EmptySlotTexture;
        InventorySet[i].Type = "";
        InventorySet[i].empty = true;
        Stats.AddAmmo(1,40,1);
        }
        if(i == 40 && !InventorySet[i].empty)
        Debug.Log("No Ammo");
    }
}
```

Now, we need to call this function when we run out of ammo. The best way to do it is to call the function from the `CH_Controller` script, from the `AltShooting` function:

```
public var inventory : CH_Inventory;
function Start(){
Stats = this.gameObject.GetComponent("CH_PlayerStats");
inventory = this.gameObject.GetComponent("CH_Inventory");
...
}
function AltShooting(){
var hit: RaycastHit;
Stats.AddAmmo(1, -1, 1);
if (Stats.GetAmmo(1) == 0){
inventory.FindAmmo();
    }
...
}
```

Our weapon will now reload with ammo from the inventory.

Finishing adjustments

There are some last touch-ups that need to be done in the code.

Open the **CH_Inventory** script and perform the following steps:

1. In the `Start` function, disable `AvatarCamera`.
2. At the very top of the `OnGUI` function, we need to enable or disable the camera based on the `inventoryOpened` value and return from the function if `inventoryOpened` is `false`:

```
function Start(){
...
avatarCamera.enabled = false;
}
function OnGUI(){
avatarCamera.enabled = (inventoryOpened) ? true : false;
if (!inventoryOpened)
    return;
...
}
```

Open the **CH_Controller** script and perform the following steps:

1. In the `FixedUpdate` function, check if inventory is open before calling the `Movement` function; if it is, return.
2. In the end of the `FixedUpdate` function, check if the player has hit the *I* button and toggle the `inventoryOpened` variable inside the `CH_Inventory` script.
3. Now, we will go to the **weapon_pickUp** script and change the last line of the `OnTriggerEnter` function to this.
4. Save the **weapon_pickUp** script and open the **treasure** script. Create a reference to the **CH_Inventory** script.
5. In the `switch` statement where we are checking the type of treasure, add last case **MedKit**, reference **Inventory**, and add **MedKit** to it.

In the `FixedUpdate` function, add the following code snippet:

```
...
transform.Rotate(Vector3(transform.rotation.x, Input.GetAxis("Mouse
X"), transform.rotation.z) * Time.deltaTime * 250.0);
if (inventory.inventoryOpened)
return;
    Movement()
...
```

```
if (Input.GetKeyDown (KeyCode.I) ){
        inventory.inventoryOpened = (inventory.inventoryOpened) ?
false : true;
    }
```

Go to the **weapon_pickUp** script and change the last line of the `OnTriggerEnter` function:

```
other.gameObject.GetComponent("CH_Inventory").
AddInventory("RightWeapon");
```

That is it, we are done.

Summary

Those were the barebones of the inventory. As we saw in this example, GUI is our best friend as far as they are asked to do what they were made for—display information on the screen. In the next chapter, we will attempt a different approach to creating user interface with planes.

5
Dynamic GUI

In this chapter, we will be going over several types of **graphical user interface** (**GUI**). The GUI is extremely important for the user to find out what is going on and to be given visual feedback on their inputs. The GUI elements that will be covered are score display, objective display, pickup display, and arch targeting.

Several of these elements have multiple parts and work in unison with others, so it is important to have an understanding of GUI and the way these parts interact with each other slowly. It is easy to get lost and confused in the logic of setting up the GUI.

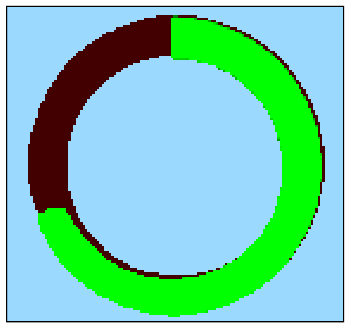
With that being said, let's now begin. The first GUI element to be explored is **Dynamic heads up display** (**HUD**). We will perform the following actions:

- We will create a circular health bar
- We will write scripts for item and weapon selection
- We will discuss text meshes
- We will create the scoring system
- We will save/read and build an arch targeting system

As stated previously, some of the GUIs have multiple parts. When it comes to HUD, there are usually several parts that can be taken into consideration. Those parts in our case are the display of health, armor, items, weapons, change camera mode, and saved score display. We will tackle these one at a time starting with health.

Radial health display

To make the creation of the health interesting and hopefully show a great way to display the health, we are going to create a radial bar. A brief description of this is that the player's health will start at 100, or max, and as it drops, it will move around in a circle appearing to be draining, as shown in the following diagram. We first stumbled across this in a Unity forum and have found it very useful for various situations, one of which has been the display of health.



Alright, let's take a look at what we are going to need to get this to work the way we want it to. We need the following for that:

Alpha gradient: This is a texture that has an alpha radial gradient transparency with a color layer. Luckily, this is already created for you, as it is the Unity material as well. It is found in the `materials` folder for this chapter.

Game manager: This is a manager game object that can hold our scripts. If you already have a manager set up, create the `Health` script.

Health script: This is a script to handle the health separate from the display of health. We will write this script in the next section.

Health display script: This is a script to handle the display of health.

Once these scripts are written, we will revisit the `Health` script to add some functions.

The Health script

Before we go ahead and begin displaying the health, we must first get the health working. This script will cover setting up the minimum health, maximum health, increase and decrease of health. To begin, let's start with initializing some variables.

We will need a variable for `currentHealth`; this can be `private`. We will also need two additional variables, `healthMax` and another for `healthMin`. These two can be `public` so that they can be modified in **Inspector**:

```
public var healthMax : float = 100;
public var healthMin : float = 0;
private var currentHealth : float = 0;
```

Next, we will want to establish a function to handle the rise and fall of our player's health. Create a function and call it `PlayerHealth`. Inside `PlayerHealth`, we want to get the influence on our `currentHealth` variable. In this case, we are using the – button to lower health and the + button to increase it. There will be two `if` statements in the function. One `if` statement will be for the increase in health and another for the decrease.

The first step is getting the pressing and holding input from the - button with the `GetKey` and `KeyCode` functions. We also need to make sure to check here that `currentHealth` is greater than the `healthMin` value.

The second step is getting the pressing and holding input from the + button with the `GetKey` and `KeyCode` functions. We also need to make sure to check here that `currentHealth` is less than `healthMax`:

```
if(Input.GetKey(KeyCode.KeypadMinus) && currentHealth >
healthMin){currentHealth -= 0.5;}
if(Input.GetKey(KeyCode.KeypadPlus) && currentHealth <
healthMax){currentHealth += 0.5;}
```

Lastly, we need to constantly check for the input coming from the player. We therefore call the `PlayerHealth` function inside of the `Update` function:

```
function Update(){PlayerHealth();}
```

We are done, at least with the `Health` script for the time being. We will have to revisit it soon.

```
public var healthMax : float = 100;
public var healthMin : float = 0;
private var currentHealth : float = 0;
PlayerHealth(){
if(Input.GetKey(KeyCode.KeypadMinus) && currentHealth > healthMin)
{currentHealth -= 0.5;}
if(Input.GetKey(KeyCode.KeypadPlus) && currentHealth < healthMax)
{currentHealth += 0.5;}
}
function Update(){PlayerHealth();}
```

Health display script

Next up, we create a script, which will display a visual indicator of our character's (or bot's) health. This script will be called `HealthBar`. Let's begin creating the variables that will help us display the health bar.

The first one which we need to create will be to hold our radial health textures to be displayed. This variable should be a `Texture2D` list and should be `public`.

We will also need a variable to hold reference to the game manager, as our `HealthBar` script will not be on the `gameManager` object. The variable will be of `GameObject` type and should be `public`.

Next, we need to set up some more reference variables for our health values. These variables are `currentHealth`, `healthMax`, and `healthMin` and can be copied from the `Health` script. They can also be changed to `private` for this script:

```
public var healthValue : Texture2D[];
public var gameManager : GameObject;
private var healthMax : float = 0;
private var healthMin : float = 0;
private var currentHealth : float = 0;
```

Now that we have our variables set up, we can set their references in the `Awake` function. Starting with our 2D texture list, we will want to set the default texture to be displayed. To do this, we need to access the renderer of `gameObject` and change its `mainTexture` material to the one in slot `0` that we have specified in our texture list:

```
renderer.material.mainTexture = healthValue[0];
```

Next, we need to set the `healthMax` and `healthMin` variables. These are from the `Health` script in the game manager so we need to access that component and read them from the return functions in the `Health` script. To do that, we use our `gameManager` variable that is referencing the game manager and the `GetComponent` function to specify which script we want access to. Then we need to call the returned functions for the specified variables as follows:

```
healthMin = gameManager.GetComponent("Health").SendHealthMin();
healthMax = gameManager.GetComponent("Health").SendHealthMax();
```

After that is set up, we want to create a function that will change the texture color based upon the health value. Call the `Health(health : float)` function. This function will have to receive information, so make sure that it has a `float` parameter.

In the `Health(health : float)` function, there will be an `if` statement and two `else if` statements. All are based upon the health value that is coming into the function. The first statement is to check if health is greater than or equal to `healthMax/3`. If it is, have `mainTexture` equal to that of our default texture—`healthValue[0]`. And in addition, we will set the emissive property of the Shader:

```
if (health >= healthMax/3 ){
    renderer.material.mainTexture = healthValue[0];
    renderer.material.SetColor("_Emission", Color.green);
    }
else if (health < healthMax/3 && health >= healthMax/1.5){
        renderer.material.mainTexture = healthValue[1];
        renderer.material.SetColor("_Emission", Color.yellow);
    }
else if (health < healthMax/1.5){
        renderer.material.mainTexture = healthValue[2];
        renderer.material.SetColor("_Emission", Color.red);
    }
```

This property allows the texture to emit the color specified, hence the reason it is being used here. Setting the color of the property is very similar to setting the texture. Instead of `mainTexture`, we use `SetColor`. `SetColor` is a built-in function with parameters. Those parameters for the color are the properties of the Shader to be affected—`_Emission`—followed by the color to be used. In this case, we will set the color to green to coincide with the texture that we are using. This process is repeated for the two `else if` statements.

For the first `else if` statement, the `if` parameter will compare whether health is less than `healthMax/3`. It will also compare whether health is greater than or equal to `healthMax/1.5`. The texture should change to the second slot and the color of `SetColor` should be changed to yellow.

The next `else if` statement will have the `if` parameter to compare whether health is less than `healthMax/1.5`. The texture again shall change, this time to the third slot and the color property will become red.

The last line that we need to write for this function is to make the `currentHealth` variable equal to the health parameter of the function:

```
currentHealth = health;
```

There are two small functions left to write and then back to the `Health` script. The next function will handle the visual increase and decrease in health. The function is called `HealthRiseFall`.

There is one line in here and it again plays with Shader attributes. This one in particular will be playing with the transparent cutoff Shader. This Shader allows one to create an alpha gradient that will dictate how the Shader handles transparency. Then, one can go in and manipulate the cutoff control to have parts of the texture appear or disappear. The Shader takes the values put in and puts minimum and maximum transparency to that. It then uses the parameter to set its transparency to slide between those two values. This is why we will be able to use it for a radial health bar. We will set its maximum transparency to `healthMax` and minimum transparency to `healthMin`. We then take the current health variable and slide the cutoff between these two values.

For our purposes, we want to use the `SetFloat` function. The parameters for this function are the Shader attribute to be affected—`_Cutoff`—and the value that it should be set at. For the value, we use another Unity built-in function called `Mathf.InverseLerp`.

This function allows us to specify the values we want to transition between and the value to transition between them. Inside this function, then, is where we have our `healthMax` as our start and `healthMin` as our end with the value determining the transition being current health:

```
renderer.material.SetFloat("_Cutoff", Mathf.InverseLerp(healthMax,heal
thMin, currentHealth));
```

The last part of this script is to create the `Update` function. In here, all we need to do is call the `HealthRiseFall` function:

```
public var healthValue : Texture2D[];
public var gameManager : GameObject;
private var healthMax : float = 0;
private var healthMin : float = 0;
private var currentHealth : float = 0;
function Awake(){
    renderer.material.mainTexture = healthValue[0];
    healthMin = gameManager.GetComponent("Health").SendHealthMin();
    healthMax = gameManager.GetComponent("Health").SendHealthMax();
}
function Health(health : float){
    if (health >= healthMax/3 ){
        renderer.material.mainTexture = healthValue[0];
        renderer.material.SetColor("_Emission", Color.green);
    }
    else if (health < healthMax/3 && health >= healthMax/1.5){
        renderer.material.mainTexture = healthValue[1];
        renderer.material.SetColor("_Emission", Color.yellow);
```

```
        }
        else if (health < healthMax/1.5 ){
            renderer.material.mainTexture = healthValue[2];
            renderer.material.SetColor("_Emission", Color.red);
        }
        currentHealth = health;
    }
    function HealthRiseFall(){
        renderer.material.SetFloat("_Cutoff", Mathf.InverseLerp(healthMax, 
    healthMin, currentHealth));
    }
    function Update () {
        HealthRiseFall();
    }
```

Well, that is it for the `HealthBar` script for now. However, we need to revisit the `Health` script and add some functions and variables.

Revisiting the Health script

As we now have values from the `Health` script that the `HealthBar` script is trying to access, we need to create two variables and two functions. Those variables will be for the health bar and the accessing of the `HealthBar` script on the health bar. The health bar variable should be `GameObject` and `public`, and the `HealthBar` script variable should be of the `HealthBar` type (the name of the script to be accessed) and `private`:

```
public var healthBar : GameObject;
private var healthBarScr : HealthBar;
```

Added to the `Awake` function is the referencing of the `HealthBar` script from the health bar `GameObject` variable:

```
healthBarScr = healthBar.GetComponent(HealthBar);
```

Now that the variables are taken care of, we can write the remaining functions. These functions will return our `healthMin` and `healthMax` variables. The `HealthBar` script is referencing certain names of functions so let's go ahead and name them the same:

```
function SendHealthMax(){return healthMax;}
function SendHealthMin(){return healthMin;}
```

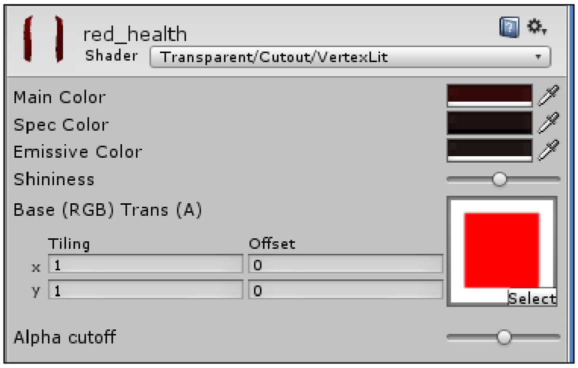
Lastly, in the `PlayerHealth` function, after all the `if` statements, we need to send the current health to the `Health` function of the `HealthBar` script:

```
healthBarScr.Health(currentHealth);
```

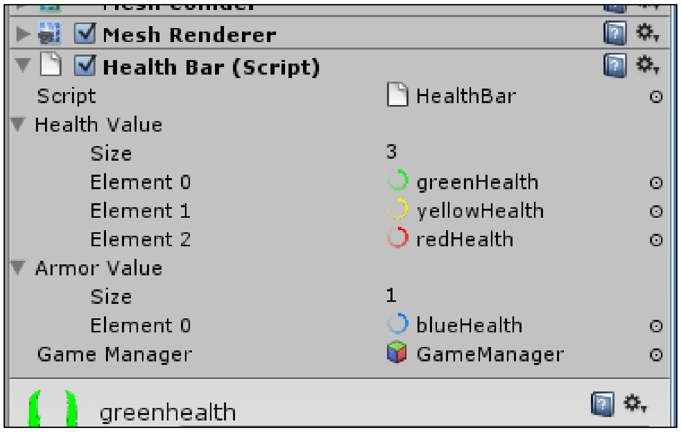
Hooking up objects to Inspector

One more step to go and you will soon see a radial health bar go up and down on your command.

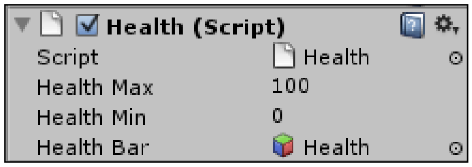
In the **Project** menu, go to the **Assets** folder for this chapter. Under **Prefabs**, you will see a prefab called **Health_Bar**. Drag this to your **Hierarchy**. Inside of the prefab, you will see that there is a **gameObject** for the background called **Health_BG** and another called **Health**. Apply red background ring material to **Health_BG**, as shown in the following screenshot:



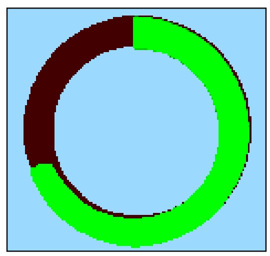
Drag the **HealthBar** script onto the **Health** `gameObject`. You should see your 2D texture list and its value set to **0**; change this to **3**. Assign **greenHealth** to the first slot, **yellowHealth** to the second, and **redHealth** to the third. Next, grab your **GameManager** and drag it into the **Game Manager** slot in the **HealthBar** script, as shown in the following screenshot:



Go to your game manager and drag the **Health** script onto it. If not already done, set **Health Max** to **100** and **Health Min** to **0**. Drag the **Health** `gameObject` into the **Health Bar** variable, as shown in the following screenshot:



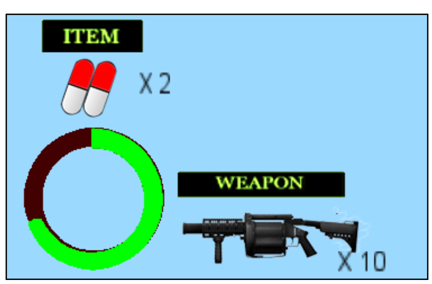
Congrats! You should have noticed by now that there is a green ring in your preview window. Upon pressing **Play**, and holding down the + or - key, you should see the green ring move around the circle. Based on amount of health, its color will change to yellow if two-third of health is left or to red if one-third is left:



Creating items

What would a game be if a player could not go and pick up items? An unfortunate game I tell you. It is true they are not needed for all games, but just in case you need to keep track of them and display them, then we can give you a hands-on way to do it.

This system will allow you to keep track of items in your inventory and display them. We will keep track of images that will dictate which item is displayed and use the name to dictate which item is displayed and the information to be displayed. Also, the use of the items and display of their information will be tackled:



Alright, let's take a look at what there is that needs to be done to get this system up and running:

1. Create the `Change_Item` script.
2. Create the `UseItem` script.
3. Create the `PlayerStats` script.
4. Create the `textMesh` script.
5. Create the `TextManager` script.
6. Revisit the `Health` Script.

The Change_Item script

This script is going to allow the player to press a button and toggle through the item array. Also, there is a function that can be called to add new items to that item array. To make this work, we will need to perform the following steps:

1. Create variables to manage texture files and values.
2. Write changing items function.
3. Write functions for addition and removal from the array.
4. Handle texture rendering.
5. Create a function for the increment control.

Setting up the code

So, first off, let's create our variables. The item array will be the first. It can be `private`. To show functionality of the script, two items have already been set up in the array. One variable will hold the `painKiller` texture and the other will have the `healthPack` texture. As we are dealing with arrays, we need a variable to handle the item to be removed and a variable for an item to be added. As we want to display item image, the item to be removed can be `String` and the item to add can be `Texture2D`. The last variable to set up at the moment is one to increment through the array:

```
public var itemLog : Array = new Array();
private var increment : int = 0;
public var painKiller: Texture2D;
public var healthPack: Texture2D;
private var itemToRemove: String;
private var itemToAdd: Texture2D;
private var item : String;
```

Next, we will need some functions. As we need to add the `painKiller` and `healthPack` items to the array, we need that done in the `Awake` function. Then, based upon the first item in the array, we need to set the `mainTexture` of `gameObject`:

```
function Awake(){
        itemLog.Add(painKiller, healthPack );
        renderer.material.mainTexture = itemLog[increment];
    }
```

Changing items

The next function that we want to create is `ChangeItem`. This function will increment through the array to the next item based upon the player's input and return the current item name. An `if` statement is used for this, and using the `GetKeyDown` function, check for the *J* key to be pressed and that our item array is larger than `0`. It is a precaution that makes sure no errors occur if we press the button when there are no items in the array. Inside, we will then call for the `IncrementControl` function to make sure that the next time we press the *I* key, it is the next item that comes up. This will be followed by the changing of `mainTexture` of the object using the current increment index. Lastly, we call the `ItemName` function to return the current item:

```
function ChangeItem(){
        if (Input.GetKeyDown(KeyCode.J) && itemLog.length > 0){
            IncrementControl();
            renderer.material.mainTexture = itemLog[increment];
            ItemName();
        }
    }
```

Addition and removal

The following two functions deal with addition of items to the array and their removal.

The first one is called `AddItemToList` and receives a `Texture2D` parameter. This function needs to be called externally. Inside the function, we want the `itemToAdd` variable to become equal to the `newItem` added parameter variable. Now, we add that item to the array by using the `Add` function and specifying `itemToAdd` as its parameter. After that, we need to set the increment value to the length of the array as the `Add` function adds the new item to the end of the array. Next, we need to call the `ItemName` and `ItemDisplay` functions:

```
function AddItemToList(newItem : Texture2D){
    itemToAdd = newItem;
    itemLog.Add(itemToAdd);
    increment = itemLog.length - 1;
    ItemName();
    ItemDisplay();
}
```

The second array function is `RemoveItemFromList` and it has a parameter variable of `String` represented by `removeItem`.

This function will take the incoming name and set it to a variable; and then using a `for` loop, we will loop through the array looking for the specific item that is to be removed. Upon finding it, we use the `RemoveAt` function. It takes an integer index as its parameter so to make sure that we have the correct spot within the array, when searching through it, we use a counter that increments with each pass through the `for` loop. After removing the item, we do a check and see if the `increment` value is equal to the length of the list. If it is, we leave it as `0` and if it is not, we decrement it by `1`. After this, we call the `ItemName` function to make sure that the appropriate information is displayed. As for the last function to call in here, it is `ItemDisplay`. The completed `RemoveItemFromList` function should be as follows:

```
function RemoveItemFromList(removeItem : String){
    var newCounter : int = 0;
    itemToRemove = removeItem;
    for(n in itemLog){
        if(n.name == itemToRemove)   {
            itemLog.RemoveAt(newCounter);
            if(itemLog.length == 0){
                increment = 0;
            }else if(increment > 0){
                increment -= 1;
            }
            ItemName();
        }
        newCounter++;
    }
    ItemDisplay();
}
```

Displaying items

The next function, which is `ItemDisplay`, controls the alpha and rendering of the texture. It consists of an `if` statement and an `else if`. The `if` statement checks if there is anything in the item array; if there is not, it sets the renderer alpha to `0`. The `else if` statement checks if the array is larger than `0` in length. If so, it sets the `mainTexture` of the material to that of the current `itemLog[increment]` and sets the alpha for the material back at `255`:

```
function ItemDisplay()
{
    if(itemLog.length == 0)
    {
```

```
            renderer.material.color.a = 0;
        }
        else if (itemLog.length > 0)
        {
            renderer.material.mainTexture = itemLog[increment];
            renderer.material.color.a = 255;
        }
    }
```

Increment controls

Next, let's create the `IncrementControl` function. This function controls the incrementing of the item array. The easiest way is to have an `if` statement to check if `increment` is not at the end of the array. This is done by simply taking the array length and subtracting `1` from it. This needs to be done because arrays start at `0` in content and `1` for counting, so we therefore need to make sure that `- 1` is there. If `increment` is not at the end of the array, add `1` to `increment`. However, if `increment` has reached the end of the array, we use an `else` statement to set `increment` back down to `0`:

```
    function IncrementControl(){
            if(increment != itemLog.length - 1){
                increment += 1;
            }
            else{
                increment = 0;
            }
        }
```

After the increment control, we need to add in one more function. Create an `Update` function and call the `ChangeItem` function:

```
    function Update (){
    ChangeItem();
    }
```

For the time being, we are done with this script. We need to build one other script before we can integrate other functionality into this one. The following code snippet shows how the `Change_Item` script should look at this point:

```
    public var itemLog : Array = new Array();
    private var increment : int = 0;
    private var itemToRemove: String;
    private var itemToAdd: Texture2D;
    public var painKiller: Texture2D;
    public var healthPack: Texture2D;
```

```
function Awake(){
        itemLog.Add(painKiller, healthPack );
        renderer.material.mainTexture = itemLog[increment];
    }
    function ChangeItem(){
        if (Input.GetKeyDown(KeyCode.J) && itemLog.length > 0){
            IncrementControl();
            renderer.material.mainTexture = itemLog[increment];
            ItemName();
        }
    }
function AddItemToList(newItem : Texture2D)
{
    itemToAdd = newItem;
    itemLog.Add(itemToAdd);
    increment = itemLog.length - 1;
    ItemName();
    ItemDisplay();
}

function RemoveItemFromList(removeItem : String)
{
    var newCounter : int = 0;
    itemToRemove = removeItem;
    for(n in itemLog)
    {
        if(n.name == itemToRemove)
        {
            itemLog.RemoveAt(newCounter);
            if(itemLog.length == 0)
            {
                increment = 0;
            }else if(increment > 0)
            {
                increment -= 1;
            }
            ItemName();
        }
        newCounter++;
    }
    ItemDisplay();
}

function ItemDisplay()
```

```
{
    if(itemLog.length == 0)
    {
        renderer.material.color.a = 0;
    }
    else if (itemLog.length > 0)
    {
        renderer.material.mainTexture = itemLog[increment];
        renderer.material.color.a = 255;
    }
}

    function IncrementControl(){
        if(increment != itemLog.length - 1){
            increment += 1;
        }
        else{
            increment = 0;
        }
    }
    function Update () {
        ChangeItem();
    }
```

One thing that you might have noticed so far, in this chapter, is that many of the scripts rely on others to make sure that their functionalities are working. We will be backtracking into the scripts and adding covers, functions, or variables, so please be patient as everything will come together in the end.

Creating the UseItem script

The `UseItem` script will have the functionality of making the selected item useable. There is only one variable that we need to set up at the moment. The first one we will create is the `itemName` variable. It can be `private` and have its type as `String`:

```
private var itemName : String = null;
```

Next, we want to go ahead and create the `SetItemName` function. It will also have the parameter of item with type of `String`. This function is going to do just as its name implies, receive an item name and then set the `itemName` variable to that received name:

```
    function SetItemName(item : String)    {
        itemName = item;
    }
```

Now that we have the item to be used, we can create our `UseItem` function. This function is going to allow us to press a key and at this moment, nothing will happen, but when we return after creating the `PlayerStats` script, we can have the pressing of the key to have the effect that is wanted.

The function has the `if` statements that check the item name and if the *Q* key was pressed. If `itemName` matches the `if` parameter, the effect occurs. As we have two items in our item array at the moment, we have two `if` statements that it is checking:

```
    function UseItem(){
if (Input.GetKeyDown(KeyCode.Q)){
        switch(itemName):
    case:"painKiller"{
        }
    case:"healthPack"{
        }
}
```

There are two more functions to write until we are done with this script. The first is the `Update` function, which contains only the call for the `UseItem` function and the second is a return function for item name:

```
function Update () {
    UseItem();
}
function ItemRemove(){return itemName;}
```

This script is done for now. There is a lot more that needs to be added into this script for functionality, but first the `PlayerStats` script needs to be written. That being said, we first need to return back to the `Change_Item` script and add some missing functionality:

```
private var itemName : String = null;
function SetItemName(item : String)    {
    itemName = item;
}
function UseItem(){
if (Input.GetKeyDown(KeyCode.Q)){
        switch(itemName):
    case:"painKiller"{
        }
    case:"healthPack"{
        }
}
function Update () {
    UseItem();
}
function ItemRemove(){return itemName;}
```

Revisiting the Change_Item script

Now that we are back in the `Change_Item` script, we can add two lines of code to the script. Thankfully, these two lines are exactly the same. What needs to be added is returning the item name to the `SetItemName` function of the `UseItem` script.

To do this, we need to create a variable to hold the `GameManager` object. After that, we need to write the `ItemName` function. In here, we have an `if` statement followed by `else`. The `if` statement sends the actual name based upon increment value. It is used when there are items in the array. When there are no items in the array, we send the `else` line. This line is identical to the preceding one in the `if` statement except that for the parameter for the `SetItemName` function, we use `placeholder` to kill it and not display anything:

```
function ItemName()
{
    if(itemLog.length > 0)
    GameManager.GetComponent("UseItem").
    SetItemName(itemLog[increment].name);
    else
    GameManager.GetComponent("UseItem").SetItemName("placeholder");
}
```

The `itemLog` line can also be copied into the `Awake` function to have the display change automatically:

```
    var GameManager : GameObject;
    function Awake(){
    GameManager.GetComponent("UseItem").
    SetItemName(itemLog[increment].name);
    }
    function ItemName()
{
    if(itemLog.length > 0)
    GameManager.GetComponent("UseItem").
    SetItemName(itemLog[increment].name);
    else
    GameManager.GetComponent("UseItem").SetItemName("placeholder");
}
```

The PlayerStats script

This script handles various statistics of the player. It is a reference script and so it contains many return statements. We will cover what we can do with this script for the time being.

Some of the variables that the script has are `painCount`, the number of `painKiller` instances, `healthCount`, the number of `healthPack` instances, a return value for the `healthPackValue` upon the use of `healthPack`, and lastly an `item` variable to handle incoming item names:

```
public var painCount : int = 2;
public var healthCount : int = 5;
public var healthPackValue : float = 25;
private var item : String = null;
```

After we have established our variables, there are a couple of functions that we need to write. These functions are non-return functions. Those will come next.

First off, we need to create the `DecrementItemCount` function. This function allows an item being used to be identified and then have its visual count number decrease upon the HUD. The function has the `item` variable become equal to the `ItemRemove` function from the `UseItem` script. These are on the same `gameObject` so it is easy using the `gameObject.GetComponent` function here. Next, there are two `if` statements. The second one is `else if`. They are basically checking the `item` variable's value against a predetermined name, in our case, `painKiller` and `healthPack`. Inside each `if` statement, we have their respective variables, `painCount` for `painKiller` instances and `healthCount` for `healthPack`. Here, we will subtract `1` from their counts. Lastly, we will write a line here, which will reflect our next two steps in putting together the HUD. We will start with the `DecrementItemCount` function that should look like the following code snippet:

```
function DecrementItemCount()
{
    item = gameObject.GetComponent("UseItem").ItemRemove();

    if(item == "painKiller")
    {
        painCount -= 1;
    }

    else if(item == "healthPack")
    {
        healthCount -= 1;
    }
}
```

This line will access the `TextManager` script and grab the `itemText` list array slot `0` from this script (this is a list of `gameObject` instances that are representative of the various screen displays of text). From there, we want to access the `textMesh` component and the `DisplayInformation` function from that component. The display information script has two parameters, one is the `item` variable identifying the particular item information to display and the other is the `Color` that the text should be identified with:

```
gameObject.GetComponent("TextManager").itemText[0].
GetComponent("textMesh").DisplayInformation(item, Color.black);
```

A single function now remains, which is the `ResetValues` function. This function is called externally from another script to reset the amount that a useable item has, for instance, health and ammo. Inside of the function, we check the amount remaining in the item and, if it is equal to 0, we then have the item's amount returned to its default value. This can be looked at as a crude form of reload:

```
function ResetValues(ammo:String, itemCount:String)
{
    if(painCount <= 0)
    painCount = 2;
    else if(healthCount <= 0)
    healthCount = 5;
}
```

Now that this is taken care of, we can write those return functions. The first one will return `painCount`, the second one will return `healthCount`, and the third one will return `healthPackValue`:

```
function ReturnPainCount(){return painCount;}
function ReturnHealthCount(){return healthCount;}
function ReturnHealthPackValue(){return healthPackValue;}
```

For now, the `PlayerStats` script is done. It will have to be revisited when dealing with weapons and ammo:

```
public var painCount : int = 2;
public var healthCount : int = 5;
public var healthPackValue : float = 25;
private var item : String = null;
function DecrementItemCount()
{
    item = gameObject.GetComponent("UseItem").ItemRemove();

    if(item == "painKiller")
    {
```

```
            painCount -= 1;
        }

        else if(item == "healthPack")
        {
            healthCount -= 1;
        }
    gameObject.GetComponent("TextManager").itemText[0].
    GetComponent("textMesh").DisplayInformation(item, Color.black);
    ]
    function ResetValues(ammo:String, itemCount:String)
    {
        if(painCount <= 0)
        painCount = 2;
        else if(healthCount <= 0)
        healthCount = 5;
    }
    function ReturnPainCount(){return painCount;}
    function ReturnHealthCount(){ return healthCount;}
    function ReturnHealthPackValue(){ return healthPackValue;}
```

The TextManager script

The `TextManager` script is a reference holder for several GUI elements in the HUD. It will help in the displaying of ammo, items remaining, score, saved game prompt, and objective display.

A very short script containing a couple of functions, it plays a huge role in keeping everything separate and clean and is very easy to use. It acts as a hub and redistributes information to referenced sources. The script has a `GameObject` variable list that holds those `gameObject` variables stated previously. Each `gameObject` in that array has a specific component on it, which allows this script to work.

So, talking about variables, let's define ours. As stated, we need the `itemText` `GameObject` variable array:

```
public var itemText : GameObject[];
```

As said before, this is a short script. The last thing to do now is write the `ReturnTextManager` function, which returns the `itemText` variable:

```
function ReturnTextManager(){return itemText;}
```

And that is literally it for this script, at least for the time being. We will return here in a little while. The `textMesh` script will be displaying all text information.

The textMesh script

Moving on to our next script to write, which will be our largest so far. The textMesh script will pretty much display our informative text all over the screen. It will look at the gameObject that it is attached to and, based upon that, determine the set of data to display.

Let's move on to variables. First off, we need a variable to determine color. This color will be our default and will be used in case a color is not specified on particular text. Next, we need a variable to hold gameManager, one for the PlayerStats script, another for referencing of an item's name, and one more to reference the text component of the Unity TextMesh component. Then, the last to be written are boolean variables for each object in the itemText variable of the textManager script. At the moment, there needs to be only one:

```
public var newColor : Color = Color.black;
public var gameManager : GameObject;
private var playerStatScr : PlayerStats;
private var itemName : String;
private var myText : TextMesh;
private var itemDisplay : boolean = false;
```

Now, there are just two more functions to write. One for the Awake function and the other for the display information.

In the Awake function, we want the myText variable to get a reference of the textMesh component on the gameObject. We then create an if statement to grab which gameObject we are on, based upon the name of the gameObject. In this case, as it is only the one, we can just create the if statement for itemMultiplier and set itemDisplay to true. Next, we will need to set the reference of the PlayerStats script by using gameManager.GetComponent. Next, we need to make sure that our default color chosen is applied at the very beginning. As the TextMesh component is directly referencing the material of the gameObject, we can just change renderer.material.color to our specified color. The last line to write in here is the calling of DisplayInformation. It is to make sure that all information that needs to be displayed, is displayed right off the bat. The variables for its parameters are itemName and newColor:

```
function Awake()
{
myText = (GetComponent("TextMesh") as TextMesh);
    if(gameObject.name == "itemMultiplier")
    {
        itemDisplay = true;
    }
```

```
playerStatScr = gameManager.GetComponent("PlayerStats");
renderer.material.color = Color.black;
DisplayInformation(itemName, newColor);
}
```

Now, let's move on to the `DisplayInformation` function. This function will be looking at the text name coming in and its color, and depending on that, it determines which information to display, where to display it, and how it should be colored. Its parameters are `String` and `Color`.

First off, we have set the `itemName` variable equal to the incoming name (`textName`) and the `newColor` variable equal to the incoming `textColor`. Then, we want to start processing our information. We do an `if` check to determine which `gameObject` this is. If `itemDisplay`, for instance, is `true`, we can proceed with item display, which we are. Next, we have two `if` statements and one `else` statement. The `if` statements are used to determine which item is selected based upon its name. Then, depending on that, the color is applied from the function parameter and the necessary text is displayed.

In order to display the necessary text, we must convert our wanted information, an integer, into a string. To do this, we have a `myText` reference to the text input and have it equal to the text that we don't want to change, which is represented by the text in quotation marks concatenated with the information that needs to be dynamically changing. Then a bracket wrapper needs to be put around the display information. For our two-item `if` statements, the information that we want to display is the current item count, which is being returned by the `PlayerStats` script.

The `else` statement that follows right after the `if` statements has a single line, which defaults the item display text to null:

```
function DisplayInformation(textName : String, textColor : Color)
{
    itemName = textName;
    newColor = textColor;
    //
    if(itemDisplay)
    {
        if(textName == "healthPack")
        {
            renderer.material.color = newColor;
            myText.text = ("X " + playerStatScr.ReturnHealthCount());
        }

        else if(textName == "painKiller")
        {
            renderer.material.color = newColor;
            myText.text = ("X " + playerStatScr.ReturnPainCount());
```

```
            }
            else
            myText.text = "";
        }
    }
```

There we go. That is our `textMesh` script. There is more to add but at least the foundation has been laid.

Revisiting the UseItem script

Now that we have our foundations laid in some other scripts, we can go back to our **UseItem** script and add some additional functionality.

First off, we need to create some new variables. One variable of `gameObject` type to hold reference to the item switching manager, another for referencing the `PlayerStats` script, a `float` variable for `healthPackValue`, and one more for referencing the `Health` script:

```
public var itemManager : GameObject;
private var playerStatScr : PlayerStats;
private var healthPackValue : float;
private var healthScr : Health;
```

Next, we need to create an `Awake` function. In here, we will be setting up the referencing for the `playerStatScr` and `healthScr` variables and calling the `SethealthPackValue` function.

Before we create the `SethealthPackValue` function, let's add a line of code at the end of the `SetItemName` function. This line is identical to the one found in the `PlayerStats` script inside the `DecrementItemCount` function except that you will have to change the parameter of `DisplayInformation` from `item` to `itemName`:

```
gameObject.GetComponent("TextManager").itemText[0].
GetComponent("textMesh").DisplayInformation(itemName, Color.black);
```

Now, we can create the `SethealthPackValue` function. In here, there is a single line that has `healthPackValue` referencing the `ReturnHealthPackValue` function from the `PlayerStats` script:

```
function SethealthPackValue()
{
    healthPackValue = playerStatScr.ReturnHealthPackValue();
}
```

Inside of the `UseItem` function, we need to add some functionality to `healthPack`—`if` statement as well as some parameters to the `if` statement itself. After checking for the input from the player for the *Q* button, we now need to add two more parameters if we want to use the health kits with our health bar.

The first parameter to be added is to check that we have health packs remaining for use and the second is to check that our current health is less than our maximum health. The health pack count is coming from the `ReturnHealthCount` function of `PlayerStats`, and `ReturnCurrentHealth` and `SendHealthMax` are from the `Health` script:

```
.... && playerStatScr.ReturnHealthCount() > 0 && healthScr.
ReturnCurrentHealth() < healthScr.SendHealthMax())
```

Inside of the `if` statement, we need to call the `DecrementItemCount` function from `PlayerStats` as well as give `healthPackValue` to the `GetHealth` function from the `Health` script as its parameter:

```
playerStatScr.DecrementItemCount();
healthScr.GetHealth(healthPackValue);
```

The last line added to the function calls the next function—`RemoveItemFromArray`, which we will write next.

The `RemoveItemFromArray` function checks if `itemName` coinciding with one it knows has no more remaining uses. If the item has no more uses, it sends the name of the currently selected item to the `RemoveItemFromList` function with the `ItemRemove` function of `UseItem` on the `ChangeItem` script. After this line, the `ResetValues` script from `PlayerStats` is called:

```
function RemoveItemFromArray()
{
    if(itemName == "healthPack" && playerStatScr.
    ReturnHealthCount() <= 0)
    {
        itemManager.GetComponent("Change_Item").
        RemoveItemFromList(ItemRemove());
        playerStatScr.ResetValues();
    }
}
```

Revisiting the Health script

We are back in the **Health** script. We should not spend much time here as there is only one function to write.

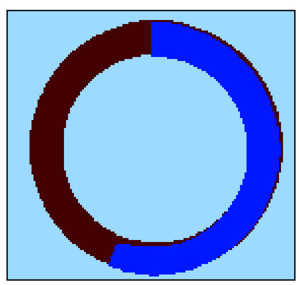
In order for us to be able to use a health pack and make it affect the visual representation of the health bar, we need to add it to the existing current health. By using the `GetHealth` function, we can receive a value and then add that value to the existing health:

```
function GetHealth(inputHealth : float)
{
    currentHealth += inputHealth;
}
```

And so ends our time in the `Health` script again.

Creating armor

Now that we can see health packs working and all the foundations of the code are in, we can go in and finish hooking up the armor scripts and functionality.



So, before we go ahead and rush towards creating the script, let's take a look at what it is that needs to be done:

1. Create the `Armor` script.
2. Revisit the `HealthBar` script.
3. Revisit the `Health` script.
4. Revisit the `UseItem` script.

The Armor script

The `Armor` script's functionality and looks are very similar to that of the `Health` script, and as a matter of fact, we are going to ask you to copy several functions and variables over and we can then tweak them.

First off, copy all of the following variables out of the `Health` script and paste them into the `Armor` script:

- `healthMax`: Rename to `armorMax`
- `healthMin`: Rename to `armorMin`
- `currentHealth`: Rename to `currentArmor`
- `healthBar`: Don't rename
- `healthBarScr`: Don't rename

Also, add a `boolean` variable—`armorActive` to track when armor has been activated.

Next, we are going to copy over the following functions and rename some variable names:

`Awake` function

  - Rename `currentHealth` and `healthMax` with their appropriate names

`PlayerHealth` function

  - Rename the `KeyCode` from - to *A*
  - Change all instances of `currentHealth` to `currentArmor`
  - Rename the `KeyCode` from + to *D*
  - Remove the `if` statement comparing the `currentHealth` amount
  - Add `ArmorDrain` function after second `if` statement
  - Wrap the `if` statements and the call of the `ArmorDrain` function in an `if` statement with the parameter of `armorActive`
  - Send current armor to the `armor` function in the `HealthBar` script, outside of the `armorActive` `if` statement
  - Rename `PlayerHealth` to `PlayerArmor`

`Update` function

  - Add `GetArmorStatus` function call
  - Rename `PlayerHealth` to `PlayerArmor`
  - Add `ResetArmor` function call

Return functions

- Rename `ReturnCurrentHealth` to `ReturnCurrentArmor`
- Rename `SendHealthMax` to `SendArmorMax`
- Rename `SendHealthMin` to `SendArmorMin`
- Make sure they are returning their values

Finally, as there are no more variables or functions to copy, we can now add some unique properties to this script.

Create a function called `ArmorDrain`. This function is going to drain the player's armor upon the activation of the armor. There is an `if` statement in here comparing to see if `currentArmor` is not yet equal to the `armorMin` value. In this statement, the `currentArmor` variable should decrement by `Time.deltaTime*2` in order to make it drain one armor per second:

```
function ArmorDrain(){
    if(currentArmor > armorMin){ currentArmor -= Time.deltaTime*2;}
}
```

If you want to turn this functionality off, comment out the function call in the `PlayerArmor` function.

The next function we want to write is the reset for `currentArmor` after the armor has been depleted. Call the `ResetArmor` function. Have an `if` statement state that if the armor is not active, it must have its `currentArmor` set to that of `armorMax` to make sure that when the activation button is next pressed, the armor is ready to go:

```
function ResetArmor(){
    if(!armorActive){
currentArmor = armorMax;}
}
```

The last function to write in here is the `GetArmorStatus` function. In here, have `armorActive` equal to `SendArmorStatus` from the `HealthBar` script:

```
function GetArmorStatus(){
armorActive = healthBarScr.SendArmorStatus();}
```

A few more tweaks here and there, well, nearly everywhere, and the `painKiller` item will have the desired effect.

Revisiting the HealthBar script

It is time to get some visual representation of the armor. Before we can really do anything, we have to create some variables. We need to create an armor variable that will hold the texture to represent it. So, just in case anyone wants to create varying degrees of armor such as the health, we will make this variable have the `Texture2D[]` type. Next, we need variables that represent `armorMax` and `armorMin` followed by one variable for `currentArmor` and the other for keeping track of its activation:

```
private var armorActive : boolean = false;
private var armorMax : float = 0;
private var armorMin : float = 0;
private var currentArmor : float = 0;
public var armorValue : Texture2D[];
```

Now, we are going to move on to function tweaking. In the `Awake` function, just under `healthMin` and `healthMax`, you might want to copy those two and paste them just underneath, and change the `healthMin` to `armorMin` and likewise for `healthMax`. Change the `GetComponent` referencing from `"Health"` to `"Armor"` and change the `SendHealthMin` and `SendHealthMax` functions to `SendArmorMin` and `SendArmorMax`, respectively.

```
armorMin = gameManager.GetComponent("Armor").SendArmorMin();
armorMax = gameManager.GetComponent("Armor").SendArmorMax();
```

Next, create a function called `Armor`. This function will have a parameter of a `float` and it will handle setting the rendering settings of the armor as well as turning the armor off when the armor is depleted.

Inside of the function, first is an `if` statement. This statement will check if the armor level has reached `0` and if it has, it will set the `armorActive` variable to `false` and call the `SendArmorStatus` function. After this statement, change `mainTexture` to be equal to that of the first index of the `armorValue` list array. After that, set the color value of the emission to blue. Set the `currentArmor` variable to the `armor` value parameter:

```
function Armor(armor : float){
    if(currentArmor <= 0){
        armorActive = false;
        SendArmorStatus();
    }

    renderer.material.mainTexture = armorValue[0];
    renderer.material.SetColor("_Emission", Color.blue);

    currentArmor = armor;
}
```

We now need to create the `ActivateArmor` function. The function will receive a `boolean` value as its parameter. The `armorActive` variable is set to the `activate` parameter:

```
function ActivateArmor(activate : boolean){
    armorActive = activate;
}
```

Next, we need to create a return function for the `armorActive` variable. Call it `SendArmorStatus`:

```
function SendArmorStatus(){
    return armorActive;
}
```

Inside of the `Health` function, we will have to put an `if` statement wrapped around everything. The parameter for the statement is to check that `armorActive` is `false`:

```
        if(!armorActive){
        ......Content.......
        }
```

There's just one more thing to do and we are done with this script. Inside the `HealthRiseFall` function, we need to wrap the health rendering in an `if` statement with the parameter of checking if `armorActive` is `false`. The second `if` statement is an `else if` checking if `armorActive` is `true`. Copy the rendering for health and paste it into this statement:

```
        if(!armorActive){
            .....Health Content.....
        }
        else if(armorActive){
            renderer.material.SetFloat("_Cutoff", Mathf.
            InverseLerp(armorMax,armorMin, currentArmor));
        }
```

Revisiting the Health script

We have a single tweak to do in this script. We need to wrap the input controls of the `PlayerHealth` function inside of an `if` statement. The parameter for the statement is going to check whether the `SendArmorStatus` function of the `HealthBar` script is not activated:

```
if(!healthBarScr.SendArmorStatus()){
    ....Content....
}
```

Revisiting the UseItem script

In order to get the armor functionality to work in the script, we need to create a `armorScr` variable that will hold reference to the `Armor` script:

```
private var armorScr : Armor;
```

In the `Awake` function, have the `armor` script variable equal the `Armor` component off the `gameObject`:

```
armorScr = gameObject.GetComponent("Armor");
```

Next, inside the `UseItem` function, we need to add the `painKiller` functionality and `if` parameters. There is an `if` wrapper that will surround the content of the function except for the bottom function—`RemoveItemFromArray`. The parameter for the wrapper is checking if the `SendArmorStatus` function of the `HealthBar` script is `false`:

```
if(!healthBar.GetComponent("HealthBar").SendArmorStatus())
    {
        if(itemName == "painKiller" && Input.GetKeyDown(KeyCode.Q)
        && playerStatScr.ReturnPainCount() > 0 )
        {
            playerStatScr.DecrementItemCount();
            return healthBar.GetComponent
            ("HealthBar").ActivateArmor(true);
        }
}
```

Creating the weapons

It's time to get us some weapon display functionality. A lot of what is going to be covered here is covered in much more detail in items. At the moment, we are going to copy many functions and scripts for the scripts required for the weapons to work. The following is what needs to be done:

1. Create the `Change_Weapon` script.
2. Create the `UseWeapon` script.
3. Revisit `PlayerStats`.
4. Revisit `textMesh`.
5. Revisit `TextManager`.

The Change_Weapon script

This script, as said before, is very similar to the `Change_Item` script, however, it will be easier for us to copy over what we need and modify the existing code. Let's begin with variables.

Copy the variables over to the `Change_Weapon` script. The variables we will be using are the `itemLog` array, `increment`, `GameManager`, `itemtoadd`, `painKiller`, and `healthPack`. These are what we need but let's rename them now. Change the `itemLog` array to `weaponLog`; `increment`, and `GameManager` can stay the same. Change `itemToAdd` to `weaponToAdd`, `painKiller` to `grenadeLauncher`, and `healthPack` to `m16`:

```
var weaponLog : Array = new Array();
private var increment : int = 0;
var GameManager : GameObject;
private var weaponToAdd : Texture2D;
var grenadeLauncher : Texture2D;
var m16 : Texture2D;
```

First off, from now on, change all `itemLog` variables to the `weaponLog` variable.

Next, we will copy over the `Awake` function. Change the `Add` instances of `painKiller` and `healthPack` to `grenadeLauncher` and `m16` respectively. Change `GetComponent("UseItem")` to `GetComponent("UseWeapon")`, the function name to `GetWeaponName`:

```
function Awake(){
    weaponLog.Add(grenadeLauncher, m16);
    renderer.material.mainTexture = weaponLog[increment];
    GameManager.GetComponent("UseWeapon").
    GetWeaponName(weaponLog[increment].name);
}
```

The next function to copy over is the `AddItemToList` function. Change the name of the function to `AddWeaponToList` and call its parameter variable `newWeapon`. Inside the function, have the `weaponToAdd` variable equal to the new weapon variable. Next, using the `Add` function for arrays, add `weaponToAdd` to `weaponLog`. Increment and then call the `WeaponName` function. As with the `Change_Item` script, this needs to be called externally:

```
function AddWeaponToList(newWeapon : Texture2D){
    weaponToAdd = newWeapon;
    weaponLog.Add(weaponToAdd);
    increment = weaponLog.length - 1;
    WeaponName();
}
```

Now, copy over the `ChangeItem` function. Change `KeyCode` to *C* and change `ItemName` to `WeaponName`:

```
function ChangeWeapon()
{
    if (Input.GetKeyDown(KeyCode.C))
    {
        IncrementControl();
        renderer.material.mainTexture = weaponLog[increment];
        WeaponName();
    }
}
```

There is one more function to copy over. Copy over the `ItemName` function and change its name to `WeaponName`. Delete everything in the function except for the line referencing the `itemLog[increment].name`. With this line, we will change some of its names and we are good to go. In `GetComponent`, put `UseWeapon` and change the function name to `GetWeaponName`:

```
function WeaponName(){
    GameManager.GetComponent("UseWeapon").
    GetWeaponName(weaponLog[increment].name);
}
function Update (){
    ChangeWeapon();
}
```

The UseWeapon script

This script is going to handle the use of the equipped gun. So to get us firing, let's set up some variables. These variables are going to be the weapon's name and a reference for the `PlayerStats` script:

```
private var playerStatScr : PlayerStats;
private var weaponName : String = null;
```

To begin the functions, let's start with the `Awake` function. In here, we want to set the reference of `playerStatScr` to that of the `PlayerStats` script:

```
function Awake(){
    playerStatScr = gameObject.GetComponent("PlayerStats");
}
```

The next function is the `GetWeaponName` function with its `weapon` parameter as `String` type. Inside of the function, we need to make `weaponName` equal to the parameter variable. Then, we want to send the color and the weapon name to the `itemText` in the second slot. This line is found in `UseItem`. Change `itemName` in `DisplayInformation` to `weaponName` and the index of `itemText` to `2`:

```
function GetWeaponName(weapon : String)
{
    weaponName = weapon;
    gameObject.GetComponent("TextManager").itemText[2].
    GetComponent("textMesh").DisplayInformation(weaponName, Color.
black);
}
```

A couple more functions to go. `UseWeapon` is the next function to write. This function will have two `if` statements that will check if the weapon name is equal to it, if the player has pressed the *F* key, and if the ammo reference from `PlayerStats` is greater than zero:

```
function UseWeapon()
{
    if(weaponName == "m-32" && Input.GetKeyDown(KeyCode.F) &&
    playerStatScr.ReturnM32AmmoCount() > 0 )
    {
        playerStatScr.DecrementAmmoCount();
    }
    if(weaponName == "m16" && Input.GetKeyDown(KeyCode.F) &&
playerStatScr.ReturnM16AmmoCount() > 0 )
    {
        playerStatScr.DecrementAmmoCount();
    }
}
```

Two functions remain. Call the `UseWeapon` function inside the `Update` function and return the weapon name with the `WeaponRemove` function:

```
function Update (){
    UseWeapon();
}
function WeaponRemove(){ return weaponName; }
```

Revisiting PlayerStats

There are a couple of variables to add to this script involving weapons. These variables will be `public`. One will be `m16Ammo` and the other will be `m32Ammo`:

```
public var m16Ammo : int = 100;
public var m32Ammo : int = 10;
```

Now, we need to create and modify some functions. Copy the `decrementItemCount` function and paste it below. Rename the function to `DecrementAmmoCount`. Create the `weapon` variable inside of the function and give it a `String` type. This variable is going to be equal to the returned weapon name from the `UseWeapon` script function—`WeaponRemove`. Next, just rename things appropriately, `weapon` in place of `item`, `"m16"` for `painKiller`, `"m-32"` for `healthPack` and change in the health and pain counts for the respective ammos. For the last line in the function that references `textMesh`, all that needs to be done is changing the list index on `itemText` to `2` and changing `item` to `weapon`:

```
function DecrementAmmoCount()
{
    var weapon = gameObject.GetComponent("UseWeapon").WeaponRemove();
    if(weapon == "m16")
    {
        m16Ammo -= 1;
    }
    else if(weapon == "m-32")
    {
        m32Ammo -= 1;
    }
    gameObject.GetComponent("TextManager").itemText[2].GetComponent
    ("textMesh").DisplayInformation(weapon, Color.black);
}
```

Inside of the `ResetValues` function, we need to add two `if` statements. Each checks a different gun's ammo and if it is at `0`, it then replaces the ammo clip:

```
    if(m32Ammo <= 0)
        m32Ammo = 10;
    else if(m16Ammo <= 0)
        m16Ammo = 100;
```

The last two functions to create are return functions. Each one returns a different gun's ammo count:

```
function ReturnM16AmmoCount(){    return m16Ammo;}
function ReturnM32AmmoCount(){    return m32Ammo;}
```

Revisiting the textMesh script

We only have a couple of small additions to be made to the script. A new variable is needed that checks to see if the `gameObject` is the `weaponMultiplier`:

```
private var weaponDisplay : boolean = false;
```

In the `Awake` function, this is used to switch the `weaponDisplay` flag to `true` if the `gameObject` name is correct:

```
if(gameObject.name == "weaponMultiplier")
    {
        weaponDisplay = true;
    }
```

There is still one more change to be made. In the `DisplayInformation` function, you need to copy the `itemDisplay` `if` statement and paste it below it. Change the `textName` to `"m16"` and `"m-32"`, and change the `playerStatScr` return functions to those that are for `m16` and `m32`. Remove the `else` statement.

```
        else if(weaponDisplay)
        {
            if(textName == "m16")
            {
                renderer.material.color = newColor;
                myText.text = ("X " + playerStatScr.ReturnM16AmmoCount());
            }

            else if(textName == "m-32")
            {
                renderer.material.color = newColor;
                myText.text = ("X " + playerStatScr.ReturnM32AmmoCount());
            }
        }
```

Scripting and displaying the score system

Now that health, items, and weapons are being displayed and working, why don't we add some scoring into the mix. We create a script that allows a player to save his score at the press of a button and save it to a text field so that he may come back at a future date and his same file will still exist. The high score will be displayed on screen and when the current score reaches and surpasses the high score, the high score will automatically start updating. Lastly, when the player saves the score, a prompt will pop up letting him know that he has saved his game.

The following steps will take us to our goal:

1. Create the `Score` script.
2. Create the `SaveScore` script.
3. Create the `timer` script.
4. Revisit the `textMesh` script.

The Score script

The `Score` script is rather small and will be our foundation for keeping track of our score. There are two variables that we need to create. One will keep track of our score and the other will be used as a reference to the `textManager` script:

```
private var currentScore : int = 0;
private var textManager : TextManager;
```

In the `Awake` function, we need to have the `textManager` reference the `TextManager` script.

The next function allows us to increment the score and send information to the score's associated `itemText` when a player presses the *P* key. The `itemText` index needed is number `3` and the `String` parameter of the `DisplayInformation` function is `"score"`:

```
function AddScore()
{
    if(Input.GetKeyDown(KeyCode.P))
```

```
        {
            currentScore += 5;
        gameObject.GetComponent("TextManager").
        itemText[3].GetComponent("textMesh").DisplayInformation
        ("score", Color.yellow);
        }
    }
```

The `Update` and `ReturnScore` functions are the last functions remaining. In the `Update` function, call the `AddScore` function and in the `ReturnScore` function, return `currentScore`:

```
function Update(){      AddScore();}
function ReturnScore(){return currentScore;}
```

This save system that we are creating is really cool. It has many possible uses but, for the time being, this use will suffice for teaching how to go about utilizing text documents, and reading and writing to and from them, respectively. As can be thought, this method can have the functionality of save systems like inventory, score, or random seeds for the building of randomly generated worlds.

For this script, we first have to import system input and output. This is represented by writing `import System.IO`:

```
import System.IO;
```

Next, we bring that in, so we need to bring in variables. We are going to have a display for when we write to the file so we need a variable for that. Make it `GameObject` and call it `saveDisplay`. The next variable is `sw`, which is of `StreamWriter` type followed by `sr`, which is of `StreamReader` type. We will then need variables for the score. One each for `savedHighScore`, `currentScore`, `displayScore`, and `highScore`. Lastly, we need reference variables for the `Score` script and `textManager`:

```
public var saveDisplay : GameObject;
private var sw : StreamWriter;
private var sr : StreamReader;
private var savedHighScore : int;
private var currentScore : int;
private var displayScore : int;
private var highScore : int;
private var scoreScr : Score;
 private var textManager : TextManager;
```

In the `Awake` function, we will first turn the renderer off on the save display by disabling it. Then, we will want to set the score reference variable to the `Score` script. The same can be done for `textManager`.

Reading from the text file

The next part here is where we begin reading from the text file, because these files need to exist before they can be accessed. We will write the code and then make the files so that you can make sure that things are working appropriately.

To read from the file, we use the `StreamReader` function. The `StreamReader` function takes the application path to the file and opens its contents. The actual name of the file is also needed. Next, we need to create a new variable called `fileContents` and have it equal to the `sr.ReadToEnd()` function. What this does is set all the information in the file into a single text but at the same time, remember where a new line was created.

Next, we no longer need the file to be open so we can just close the file by calling the `Close` function on the `sr` variable. We will then create another variable that will hold the lines of the file. To do this, we will call the `Split` function on the `fileContents` variable. We specify the type of split that we would like to perform and as we have information on different lines, we use `\n` (new line), and also where to start by stating a list index. Then, after knowing that the information has been split into individual lines and our score information is now separate and can be read, we have the `savedHighScore` variable equal to the line that the information is on by converting the `string` information or into an integer with `parseInt()`. We then proceed on and have the `highScore` variable to be equal to `savedHighScore`. Lastly, we need to send our high score information to be displayed. As the `itemText` list variable has the `highscore` variable in the fifth place, we use that index for `itemText`. So, in this case that would be `4`. The information for the `DisplayInformation` is `"highScore"` and the `Color` is `red`:

```
function Awake(){
    saveDisplay.renderer.enabled = false;
    scoreScr = gameObject.GetComponent(Score);
    textManager = gameObject.GetComponent("TextManager");
    sr = new StreamReader(Application.dataPath  +
    "/Save_ScoreFiles/Saved_High_Score.txt");
    var fileContents = sr.ReadToEnd();
    sr.Close();
    var lines = fileContents.Split("\n"[0]);
    savedHighScore = parseInt(lines[1]);
    highScore = savedHighScore;
    gameObject.GetComponent("TextManager").itemText[4].
    GetComponent("textMesh").DisplayInformation("highScore", Color.
red);
}
```

Writing to the text file

This next function deals with writing information to the text files and goes by the name `HighScore`. It will handle taking the existing score and comparing it with what is stored as the high score. If the new score is larger than the high score, `highscore` updates as `currentScore` updates.

First, we set out `currentScore` to what is being displayed, as there can sometimes be a lag in information transfer. Next, we compare the high score and the current score. If the current score is equal or higher, we continue in the statement. The `highscore` becomes equal to the current score so that it may start updating in real time. `savedHighScore` becomes equal to `highScore`, and the `StreamWriting` process starts. `sw` will be equal to the same datapath that was established in the `Awake` function. Then using the `StreamWriter` variable `sw`, use the `WriteLine` function to state what needs to be written. In this case, on the first line, we have put what is to represent and the following line that will be represented. Repeat the process of writing the line again. However, the problem that arises here is that we want to write an integer to the file. To do this, we need to convert the integer information to a string. This is accomplished by concatenating a string value, double quotes, with the value to be converted, `savedHighScore`. After that, close the file.

```
function HighScore(){
    currentScore = displayScore;
    if (currentScore >= highScore)
    {
        highScore = currentScore;
        savedHighScore = highScore;
        sw = new StreamWriter(Application.dataPath +
        "/Save_ScoreFiles/Saved_High_Score.txt");
        sw.WriteLine("Score: ");
        sw.WriteLine(savedHighScore + "");
        sw.Close();
    }
}
```

Next, it is time to set up displaying the current score as it reaches the high score and awaiting the player's response to save the high score.

We want `displayScore` to become equal to the score being returned by the `ReturnScore` function. Next, we compare that score with the high score and if it is higher, the high score equals that of the current score. This time, we change the color of the high score to show that a change has occurred:

```
displayScore = scoreScr.ReturnScore();
if(displayScore > highScore){
        highScore = displayScore;
    gameObject.GetComponent("TextManager").itemText[4].GetComponent
    ("textMesh").DisplayInformation("highScore", Color.green);
    }
```

An `if` statement follows the preceding code snippet, which checks for the players pressing of the *B* key. If it is done, we toggle the `saveDisplay` renderer to `true`, start the display timer, and save the current high score.

The last function to write is `ReturnHighScore` and it returns `highScore`.

The timer script

The `timer` script does exactly what its name implies, it times, and in this case, it acts like a countdown timer. The script requires three variables. One for the begin time, one for the cutoff time, and one to keep track of the current time:

```
public var startTime : float = 1;
public var minTime : float = 0;
private var currentTime: float = 0;
```

Next, we will want to create the `Awake` function. In here, initialize `currentTime` as `startTime`:

```
function Awake(){
    currentTime = startTime;
}
```

After the `Awake` function, we will create the `DisableRenderer` function. We do a check and see if `currentTime` is greater than `minTime`. If it is, then we decrease `currentTime` by `Time.deltaTime` and if it is not, then we turn the renderer off and reset the timer:

```
function DisableRenderer(){
    if(currentTime > minTime){
        currentTime -= Time.deltaTime;
    }
    else{
        renderer.enabled = false;
        currentTime = startTime;
    }
}
```

Lastly, we create the FixedUpdate function. This works nicely for time and gives accurate feedback:

```
function FixedUpdate(){
    DisableRenderer();
}
```

Revisiting the textMesh script

Alas, we are back in the textMesh script. We need to create four new variables. Two will be for the boolean check to see which gameObject they are on, and the other two are for accessing the Score script as well as the SaveScore script:

```
private var saveScoreScr : SaveScore;
private var scoreScr : Score;
private var scoreDisplay: boolean = false;
private var highScoreDisplay : boolean = false;
```

In the Awake function, check highScoreDisplay if they are of gameObject type. Do the same for scoreDisplay. Just below the PlayerStats referencing, set up the referencing for the SaveScore script and the Score script:

```
else if(gameObject.name == "scoreDisplay"){
        scoreDisplay = true;
    }
    else if(gameObject.name == "highScoreDisplay"){
        highScoreDisplay = true;
    }
    scoreScr = gameManager.GetComponent("Score");
    saveScoreScr = gameManager.GetComponent("SaveScore");
```

In the DisplayInformation function, create two else if statements at the end of the function, make one of them check if scoreDisplay is activated and make the other check if highScoreDisplay is activated.

Inside each, make them check if incoming textName is the one that they need to compare. Then, inside of that if statement, put the renderer color to the newColor. Lastly, set the text for score to ReturnScore from the Score script and ReturnHighScore from the SaveScore script:

```
else if(scoreDisplay){
    if(textName == "score"){
            renderer.material.color = newColor;
            myText.text = ("SCORE: " + scoreScr.ReturnScore());
        }
    }
```

```
        else if(highScoreDisplay){
            if(textName == "highScore"){
                renderer.material.color = newColor;
                myText.text = ("HIGH SCORE: " +
                saveScoreScr.ReturnHighScore());
            }
        }
```

Displaying the objectives

As this chapter has been about the creation and implementation of visual indications of various kinds, we believe it would make sense if we actually showed these functions working appropriately. We will write a `Displaying Objectives` script to do this.

This script will allow players to have the current objective displayed on screen and then upon triggering or giving an input, have that objective changed to the next one. Perform the following steps:

1. Revisit `TextManager`.
2. Revisit `textMesh`.

Revisiting TextManager

Just a few things to tackle in here to get objective display working. This script essentially is going to act as an objective manager. The player will be able to keep and display as many objectives as he wishes. As the *O* button is pressed, the objective array increments itself to display the next objective in the array.

Two variables need to be added. One will be the `String` list and the other will be `increment`:

```
var objectiveArray : String[];
private var increment : int = 0;
```

Access the `DisplayInformation` function through the `Awake` function. Create `newText` and `Color.blue` as its parameters:

```
function Awake(){
    itemText[1].GetComponent("textMesh").DisplayInformation("newText",
    Color.blue);
}
```

An increment control is required to control the flow through the array. An explanation on this is given earlier in this chapter:

```
function IncrementControl(){
    if(increment != objectiveArray.length - 1){
        increment += 1;
    }
    else{
        increment = 0;
    }

}
```

The next function is `ChangeObjective` and it does just that. Two functions are called in here. One is `Objectives` and the other is `IncrementControl`:

```
function ChangeObjective()
{
    Objectives();
    IncrementControl();
}
```

The second last function to create is the `Update` function. We have an `if` statement checking for player input for the *O* key. If the player does press the *O* key, call the `DisplayInformation` function again to get it to display the next message:

```
function Update()
{
    if(Input.GetKeyDown(KeyCode.O))
    {
        itemText[1].GetComponent("textMesh").
        DisplayInformation("newText", Color.blue);
    }
}
```

Finally, create the `Objectives` function. This function will return the current objective to be displayed:

```
function Objectives(){    return objectiveArray[increment];}
```

Revisiting textMesh

As we have done with the previous HUD elements, a new variable needs to be created. That variable is `ObjectiveDisplay` and is `boolean`:

```
private var ObjectiveDisplay : boolean = false;
```

In the `Awake` function, we need to check if this `gameObject` is `ObjectiveDisplay`; if it is, the flag is `true`:

```
else if(gameObject.name == "ObjectiveDisplay"){
    ObjectiveDisplay = true;
}
```

In the `DisplayInformation` function, create an `else if` statement with `ObjectiveDisplay` as its parameter. Inside it, create an `if` statement that checks if `textName` is equal to its own. If it is, it sets the renderer color to the parameter color. It also sets the text display of the attached `textMesh` to the current Objective being returned from `textManager` and the `Objectives` function. After this, the increment control in the text manager is activated to make sure that the next time a player presses that button, the next message appears:

```
        else if(ObjectiveDisplay){
            if(textName == "newText"){
                renderer.material.color = newColor;
                myText.text = ("Current Objective: " +
                gameManager.GetComponent("TextManager").Objectives());
                gameManager.GetComponent("TextManager").
    IncrementControl();
            }
        }
```

Hooking up HUD

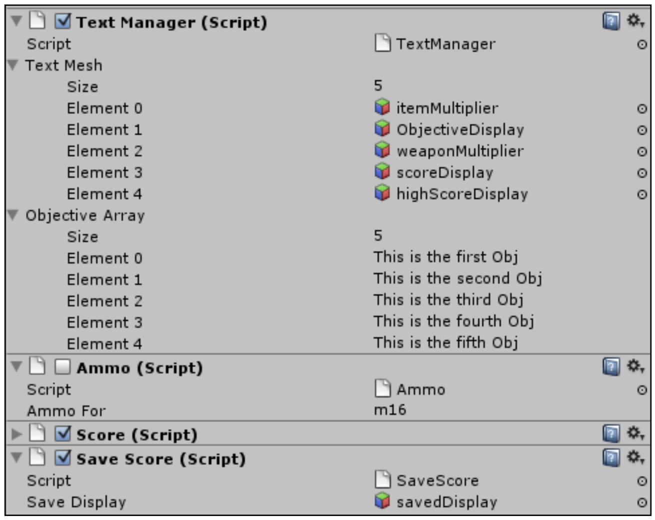
This part is going to be a checklist to make sure that everything is displaying and set up properly. The following screenshot is an example of the HUD we have been building:

We are going to go through it based upon our layout, so first off, we are going to make sure that your layout is similar to mine, so that if you have any errors or questions, it will be easy to navigate through what is going on. With that being said, let's structure our HUD layout like we have here in **Hierarchy**. Once done we will move on to each `gameObject` and make sure that the right objects are hooked up and that no questions are left unanswered towards the hooking up of this HUD.

Game manager

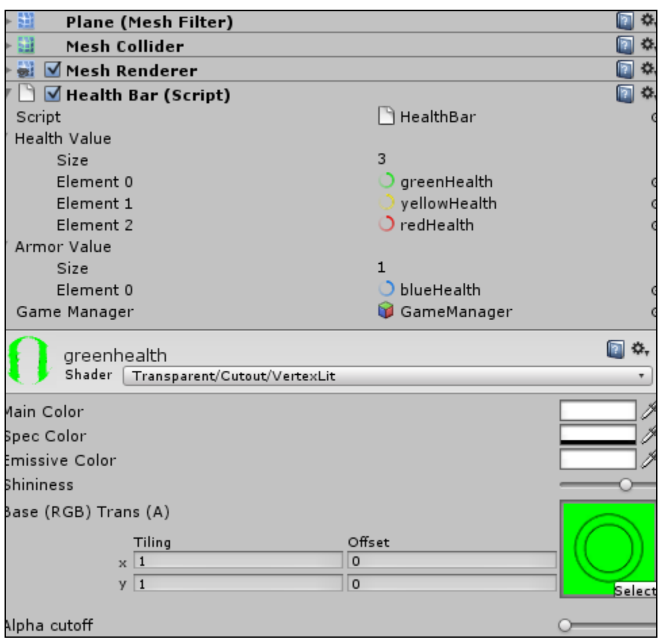
Make sure that all these scripts are attached and have correct values in them, as shown in the following screenshot:

Make sure that in the **Text Manager** script, **Text Mesh** and **Objective Array** have the correct items hooked up, as shown in the following screenshot:



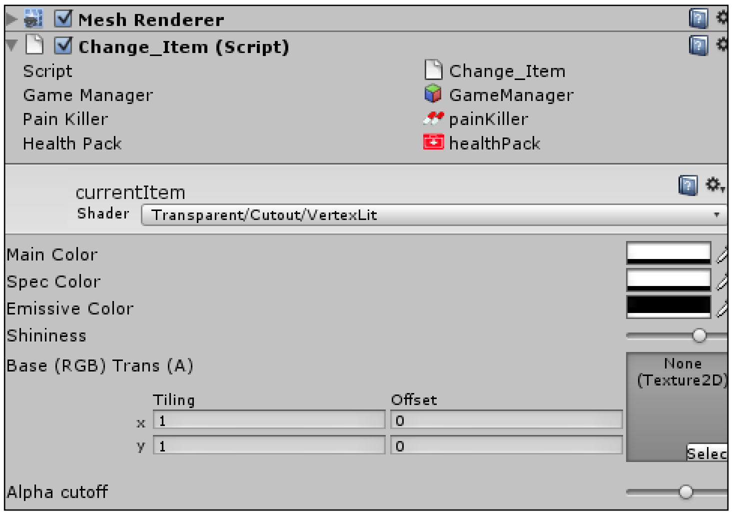
Health

This is an example of the **Health Bar** script. Make sure that all texture elements are in place, as shown in the following screenshot:



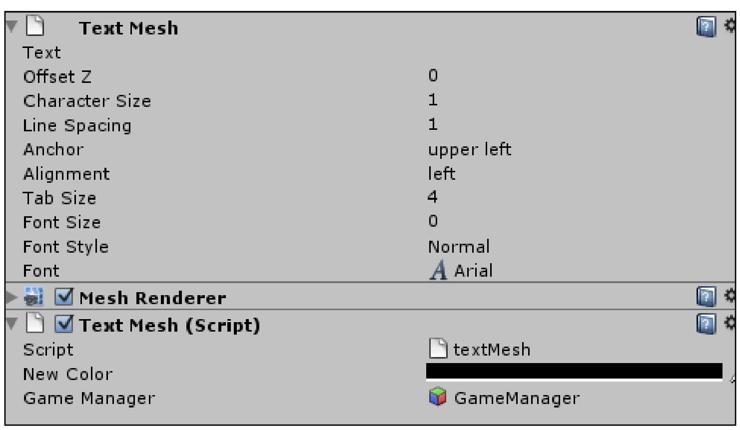
Item_Pic

Make sure that you hooked up the correct **Game Manager**, **Pain Killer**, and **Health Pack** scripts. They should have textures that represent them, as shown in the following screenshot:



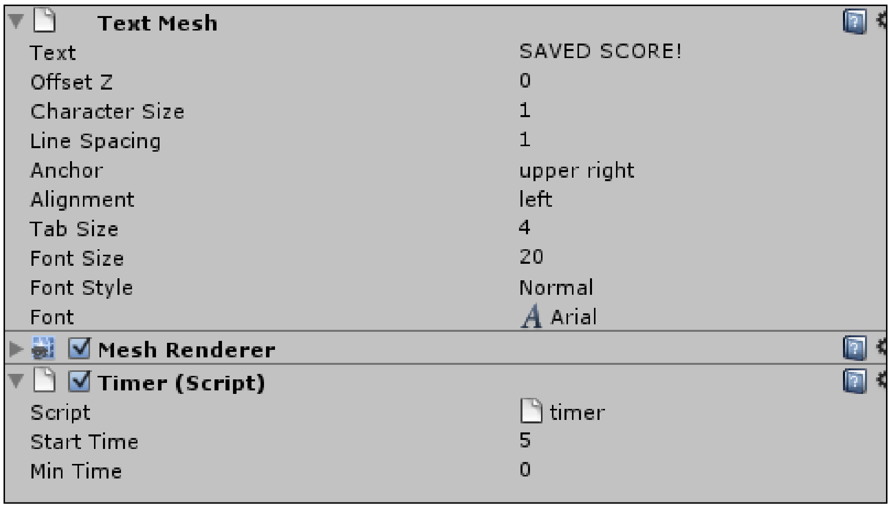
ItemMultiplier, highScoreDisplay, ObjectiveDisplay, scoreDisplay, and weaponDisplay

Everything that has to deal with text needs to have parameters like those given in the following screenshot:



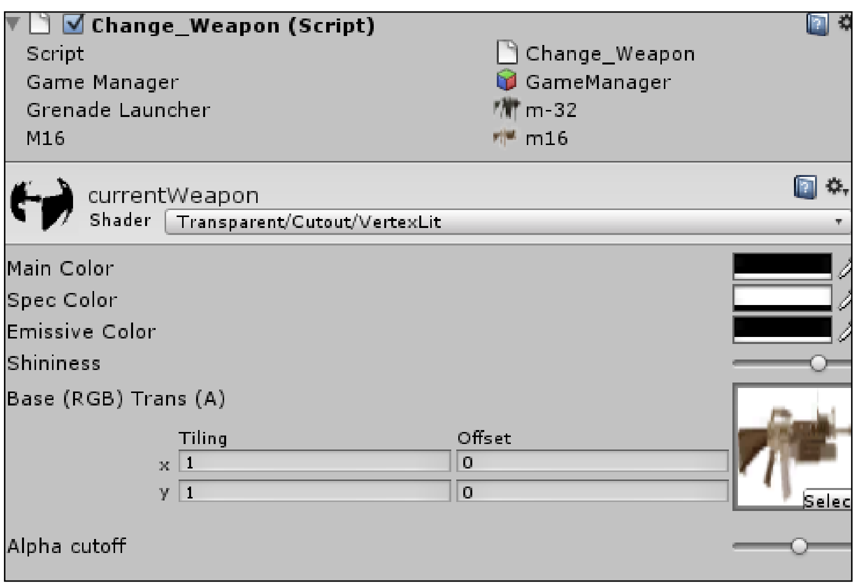
saveDisplay

Make sure that the **Timer** and **Text Mesh** scripts are set up properly, as shown in the following screenshot:



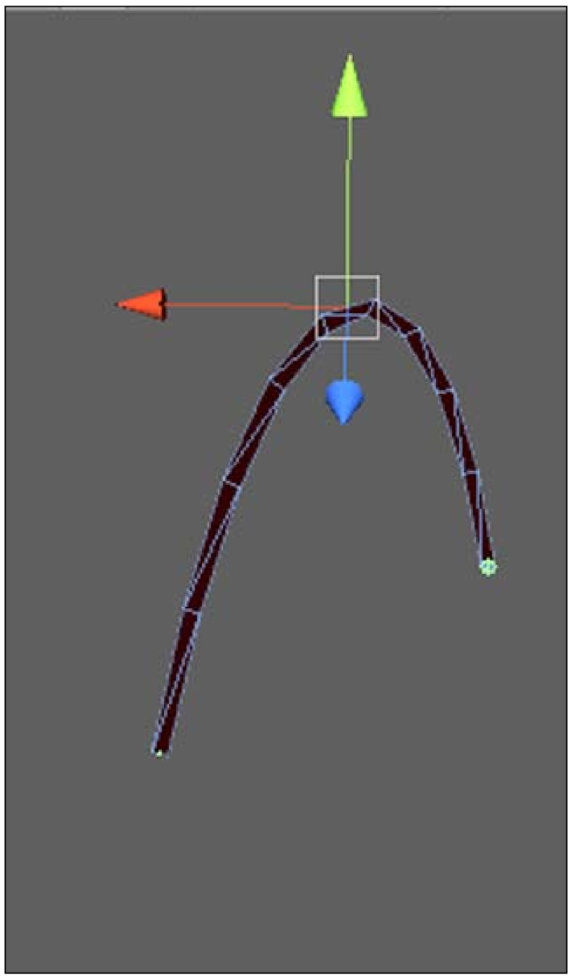
Weapon_Pic

Last, but not least, weapon pictures. Make sure that all weapons have correct pictures attached to the variables, as shown in the following screenshot:



Creating the targeting system

Having an indicator lets you know what your projectile is going to be doing. It is a great asset to have. This targeting system uses a line renderer to show what is happening with the trajectory curve, a reticule target of where it is going to land and the player has control over the steepness of shot, the distance it will shoot, and the horizontal angle.



The following are the steps that we need to take in order to create a targeting system:

1. Create the Bezier equation script.
2. Create the `ArcBehaviour` script.
3. Hook it up in editor.

Creating the Bezier equation script

The Bezier equation script will handle the equation for setting up the quadratic points to allow us to get a smooth curve along the line renderer. This was brought to my attention recently on a Unity3D forum. This Bezier equation can be written as follows:

x(t) = axt3 + bxt2 + cxt + x0

First, we need to create the variables. There are only two, and they are segments. The first one will determine the number of segments along the line, and the second is `lineRenderer` of `LineRenderer` type:

```
var sections : float = 10.0;
private var lineRenderer : LineRenderer;
```

In the `Awake` function, make the `lineRenderer` variable reference the `LineRenderer` component on its `gameObject`. After that, set the `lineRenderer` vertex count to the number of `segments`:

```
function Start(){
    lineRenderer = GetComponent(LineRenderer);
    lineRenderer.SetVertexCount(segments);
}
```

The next function that needs to be created is the `GetQuadraticCoordinates` function. This function will set up the Bezier equation. The function takes parameters, such as `t` for time, `p0` being the originating position, `c0` for the center position, and `p1` for the end position:

```
 function GetQuadraticCoordinates(t : float, p0 : Vector3 , c0 :
Vector3 , p1 : Vector3 ) : Vector3
{
    return Mathf.Pow(1-t,2)*p0 + 2*t*(1-t)*c0 + Mathf.Pow(t,2)*p1 ;
}
```

The last function to write is the `Plot` function, which sets up the location of the points that will be used to determine the positioning of the line renderer:

```
function Plot( p0 : Vector3 , c0 : Vector3 , p1 : Vector3 ){
   var t : float ;
   for (var i : int = 0 ; i < segments ; i++ )
   {
      t = i/(sections-1) ;
      lineRenderer.SetPosition (i ,GetQuadraticCoordinates(t , p0 , c0
, p1 ));
   }
}
```

ArcBehaviour

The ArcBehaviour script handles the locations of the beginning, middle, and end of the arch to be created. It then sends the position locations to the bezier.plot function for calculation.

The variables to be calculated are start, middle, and end to define the arch and they are of the Transform type. Other variables are maxHeight at which the middle point may achieve a reference to the Bezier script, a reference to the moveObject script, and the manager of the projectile:

```
public var start : Transform ;
public var middle : Transform ;
public var end : Transform ;
public var maxHeight : float = 200;
public var playerController: GameObject;
private var bezier : Bezier;
private var moveObjectScr : moveObject;
```

Inside of the Awake function, we will have the reference of the moveObject script from the manager, and the reference of the Bezier script:

```
function Awake(){
    moveObjectScr = playerController.GetComponent("moveObject");
    bezier = GetComponent(Bezier);
}
```

After this function, we will have the Update function declared. In here, we will set the position for the middle. This will determine the trajectory. We will also make the end point follow the mouse. And finally, we will set the coordinates for both the bezier.plot function and the moveObject script GetQuadraticCoordinate function:

```
function Update()
{
    var mousePos = Input.mousePosition;
    var yPosition : float ;
    yPosition = Mathf.Min( Screen.height , Mathf.Max(mousePos.y,0) );
    middle.position.y = ( yPosition / Screen.height ) * maxHeight ;
    middle.position.z = -mousePos.x;
    end.position.z = -mousePos.x;
    bezier.Plot(start.position , middle.position , end.position );
    moveObjectScr.GetQuadraticCoordinates(start.position , middle.
position , end.position );
}
```

The moveObject script

The `moveObject` script will handle the movement of an object along the project trajectory arch.

To set the script up, we will need some variables. We will need two that are of `GameObject` type; one for our projectile, the other being for the `arcManager`. We will need a reference variable for our `Bezier` script and then three variables `start`, `middle`, and `end` of the `Vector3` type to represent the start, the middle, and the end of the arch:

```
var mortars : GameObject;
var arcManager : GameObject;
private var bezierScr : Bezier;
private var start : Vector3;
private var middle : Vector3;
private var end : Vector3;
```

Inside of the `Awake` function, we will set the `arcManager` script to `Arc gameObject` and `bezierScr` will reference the `Bezier` script on `arcManager`:

```
function Awake(){
    arcManager = gameObject.Find("Arc");
    bezierScr = arcManager.GetComponent("Bezier");
}
```

Next, in the `GetQuadraticCoordinates` function, we need it to have three `Vector3` parameters, those being `stP`, `midP`, and `endP`. Inside the function, we have our coinciding variables equal to their matches:

```
function GetQuadraticCoordinates(stP : Vector3, midP : Vector3, endP :
Vector3){
    start = stP;
    middle = midP;
    end = endP;
}
```

The `Shoot` function follows after the preceding section. There is an `if` statement in it that checks for the player's input of the left mouse button. If it receives this value, it will create a mortar and fire the projectile along the arch landing in the centre of the reticule:

```
function Shoot(){
    if(Input.GetMouseButtonDown(0))
    {
        var newMortar : GameObject = Instantiate(mortars,
        Vector3(transform.position.x, transform.position.y +15,
        transform.position.z), transform.rotation);
        newMortar.transform.position =
        bezierScr.GetQuadraticCoordinates(Mathf.Lerp(0.0,1,Time.time)
,
        start ,  middle, end);
    }
}
```

Lastly, we will call the `Shoot` function in the `Update` function:

```
function Update(){Shoot();}
```

Hooking it up in the editor

Luckily for this project, the trajectory is found already put together for you inside the asset folder for this chapter. It is called `Trajectory`. Drag it to the **Hierarchy** view and you will have it set up. Remember that you can modify the end, middle, and start points to change the shape of the arch.

Summary

That's it, we are done! We hope that this example will give you a better understanding of how to create a dynamic HUD, and use of plane primitives and tricks to create it without using GUI.

In the next chapter, we will look into the creation of a game controlling system, cover sound mixing, and put all the pieces of our game together.

6

Game Master Controller

This chapter is dedicated to one of the most important parts of game development—putting all the pieces together and making them work as one solid project. In this example, we will learn how to create and set up a game manager, which will control transitions from one scene to another, track mission completing, stream new scenes as we go, save player's progress when checkpoint is hit, and play ambient music. The following list shows what we will look into:

- Game managers
- Level streaming
- Mission creation
- Saving and loading
- Audio sources and listeners
- Sound settings
- Attaching audio to the basic actions (shooting, walking, and intractable objects)

Game manager theory

Why do we use game managers? In reality, this simple question has a simple answer; the game managers help us to organize our work. For more efficient use, we can create managers to control the behavior of a particular part of the game such as the HUD, save/load system, mission management, and so on. As we create content for the game, it starts to grow and keeping track of every single function, class, and object becomes a very difficult task. To simplify navigation, we will put all the related functionality into separate managers, which makes it easier to find and keep track of in the future.

Creating game managers

There are many ways to create a game manager, and it really depends on the game and type of structure that the developer has in mind. In this example, we will create two game managers—world manager that will contain all the information about scenes, missions and will handle the saving system, and audio manager that will control ambient music. Perform the following steps:

1. Create a new script, and call it `WorldManager`.
2. Declare a new `static private` variable of a `WorldManager` type, and call it `instance`.
3. Declare a new `static public` function called `GetInstance()`.

The `WorldManager` script should be as follows:

```
static private var instance : WorldManager = null;
static public function GetInstance(){
if(!instance){
instance = FindObjectOfType(WorldManager);
if(!instance){
Debug.LogError("WorldManager does not exist");
}
return instance;
}
}
```

Here, we check if the instance of world manager has been assigned to the `instance` variable. If not, we will look for an object of a `WorldManager` type in the scene and return the instance (`FindObjectOfType` finds and returns first active loaded object of a specified type). If we don't succeed in that, we will have to report a debug error that world manager doesn't exist in the scene and needs to be created there. The `instance` variable is `static`; you will find more information about the `static` variables in the appendix. At this point, it is enough to say that the `static` variables exist throughout the entire lifetime of the project and can be accessed from any script (think of them as the advanced `public` variables, but don't use them unless you absolutely have to).

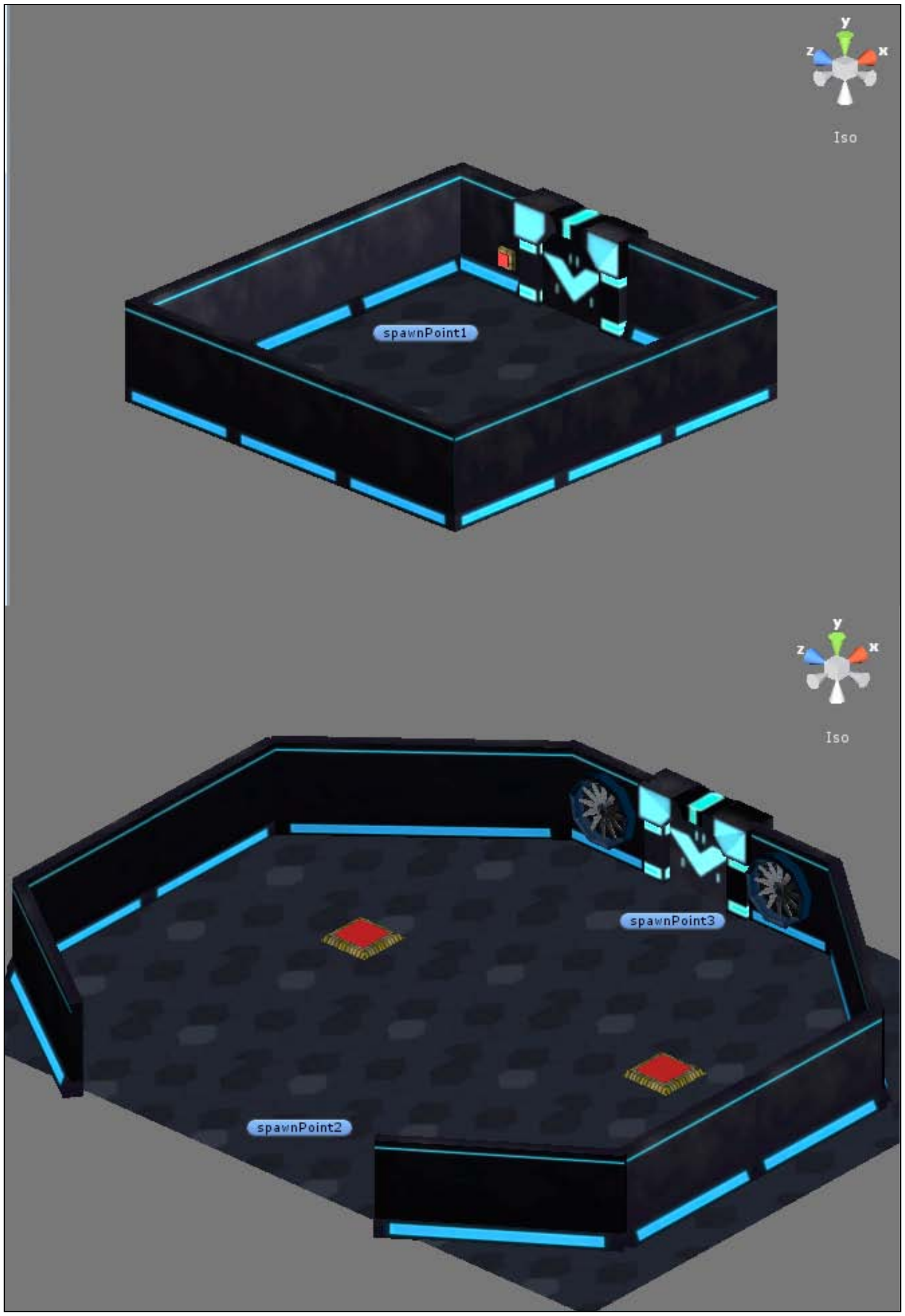
`instance` is equal to `null`, because we need to create this instance dynamically through code. We will use this function to retrieve the instance of the world manager or find one if it wasn't assigned.

Level streaming

In this example, we will look into a level-loading feature inside Unity. In the `Scenes` folder, we have five levels as follows:

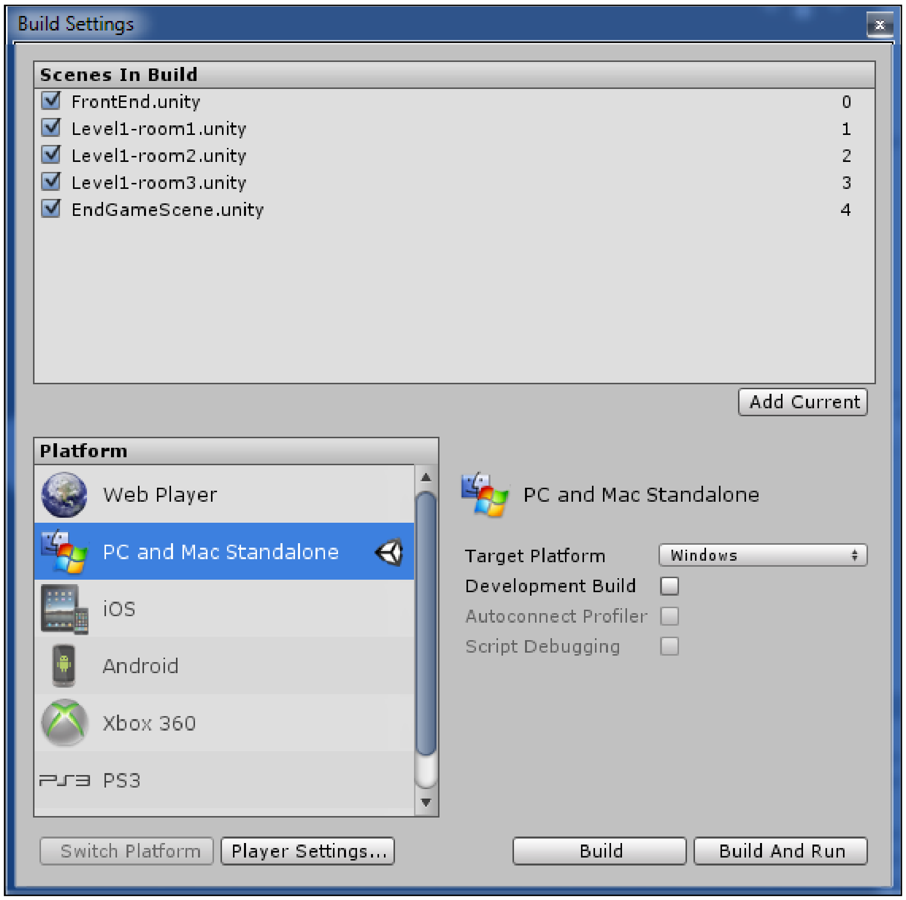
- **FrontEnd**: This will serve as a starting screen for our game
- **Level 1, Level 2, and Level 3**: These are the three levels that will be loading as our character goes through the game
- **EndGameScene**: This will be shown if a player dies or fails to complete the objective

The following screenshot shows the five levels mentioned in the preceding section:



To make Unity recognize these scenes as levels from our game, we need to manually assign them in the project properties. Perform the following steps:

1. Go to **File** | **Build Settings...**, and have the **FrontEnd** scene loaded. Click on the **Add Current** button. Current scene will be included in the game build and will be assigned an index number that we will be using to reference the scene in the code.



2. Load up each level one by one and include them in the build. Make sure that each level has an index that is the same as its number; this will make referencing easier for us. **EndGameScene** should go last.
3. In the `WorldManager` script, declare the `Awake` function:

```
function Awake(){
DontDestroyOnLoad (transform.gameObject);
}
```

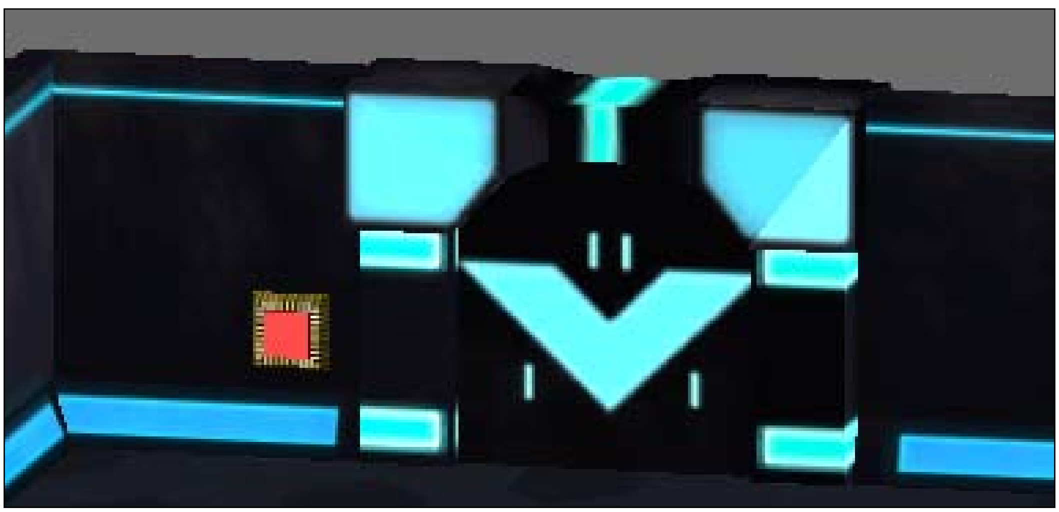
Having used scenes before, you would probably notice that as soon as you load a new scene, all objects from the old one disappear. To prevent objects from disappearing, we can tell Unity to avoid destroying the specified object.

> `DontDestroyOnLoad` is the function that tells Unity that a particular object needs to remain constant throughout the game and should not be destroyed when we change the scene.

Mission creation

Let's say we want the player to hit the button in the **Level 1** with a bio gun primary fire to open the door that will load the new level.

We will create a script for two types of buttons. The first one will be activated by the projectile and will open the door, as shown in the following screenshot:



The other button will be activated with a player stepping on it and will be used to open a second door, as shown in the following screenshot:

To create a multipurpose script, we will need variables that identify a type of specific button, and flags that will determine if buttons can be activated or if they are currently active. We determine whether a button is activated or not with the `OnCollisionEnter` function and will move the button down afterwards. Perform the following steps:

1. Declare the `ButtonType` and `Activation` variables of a `integer` type, and the `bActivated` and `canBeActivated` variables of a `boolean` type.
2. Declare the `OnCollisionEnter` function.
3. The first thing that we need to check is whether the button can be activated. If it can't be, then we will return from the function. The next step is to check the type of the button and the tag of the collided object.

4. In the first case, we need to set `bActivated` to `true` and `canBeActivated` to `false`, and increment the `Activation` variable.
5. In the second case, we also need to toggle the `bActivated` and `canBeActivated` `boolean` variables.
6. In our example, we will be opening doors whenever the buttons are activated. Let's declare the required variables.
7. In the `Awake` function, we need to store the original location of the moved doors and buttons into the variables.
8. In the `Update` function, we will handle opening the doors based on their type.
9. In the `Button` script, add the following code snippet:

```
public var Door1 : GameObject;
public var Door2 : GameObject;
public var ButtonType : int;
static var Activation: int;
public var bActivated: boolean;
private var canBeActivated: boolean;
private var DoorOneStartPosition: Vector3;
private var DoorTwoStartPosition: Vector3;
private var ThisStartPosition: Vector3;
function OnCollisionEnter(other : Collision){
if (!canBeActivated)
return;
if (ButtonType == 1 && other.gameObject.tag == "projectile"){
bActivated = true;
canBeActivated = false;
}
else if (ButtonType == 2 && other.gameObject.tag == "Player"){
bActivated = true;
Activation ++;
canBeActivated = false;
}
}
function Awake(){
    ThisStartPosition = this.transform.position;
    if(Door1)
        DoorOneStartPosition = Door1.transform.position;
    if(Door2)
        DoorTwoStartPosition = Door2.transform.position;
}
```

```
function Update(){
if(bActivated == true){
    if (ButtonType == 1){
        this.transform.position =
        Vector3.Lerp(this.transform.position, ThisStartPosition +
        Vector3(0.5, 0, 0), Time.deltaTime);
        Door1.transform.position =
        Vector3.Lerp(DoorLeft.transform.position,
        DoorLeftStartPosition + Vector3(0,7,0), Time.deltaTime);
    }
    else if (ButtonType == 2){
        transform.position = Vector3.Lerp(this.transform.position,
        ThisStartPosition + Vector3(0, -0.5, 0), Time.deltaTime);
                if (Activated == 2)
                    Door2.transform.position =
                    Vector3.Lerp(Door.transform.position,
                    DoorStartPosition + Vector3(0, 7, 0),
                    Time.deltaTime);
                            }
    }
}
```

`Activation` will be used to calculate the number of buttons activated by a player. `bActivated` will be used for the projectile activated button, and as we are not referencing any other buttons, this variable can remain `public`. `canBeActivated` will help us to disable the button when it gets activated.

The `Vector3.Lerp` function will be moving the door until it reaches the offset destination and is saved to use in the `Update` function. That is it for now; we will return to that function later after talking about level loading.

Managing levels

Now that the basic functionality of the buttons is written, we need to make them load levels for us. There are multiple ways to load scenes; one of them is to destroy the current scene and load a new one. This can be done with the `Application.LoadLevel(levelindex : int)` function. (For this function to work, our scene needs to be included into level array in the **Build Settings** window. `levelindex` is used to reference it) But, what if we don't wish to destroy the current level, and need to load a new scene on top of the existing one? For this to work, we can use the `Application.LoadLevelAdditive(levelindex : int)` function, which will load a new level without destroying the current one. This option is very useful when we are working with a limited amount of memory and want only the required pieces of levels to exist, by loading new parts and destroying the old ones.

In our case, buttons don't need to destroy previous levels, but only load new ones when the time is right.

In the OnCollisonEnter function, if our ButtonType is 1, we will load the second level on top of the current one:

```
Application.LoadLevelAdditive(2);
```

The same thing needs to be done to the Update function under the ButtonType 0:

```
else if (ButtonType == 2){
Application.LoadLevelAdditive(3);
```

Save/load system

In the previous chapter, we have already covered writing data in the text file and retrieving it back with Windows libraries. Now, we will do the same with checkpoints, plus learn how to dynamically create directories and files if they don't exist.

There is some information, which we will be saving in the save file that we need to set up, before going into save/load functions.

The following are the variables that we need to declare in the WorldManager script, which will be used in the saving system:

```
private var PlayerHealth : int;
private var AmmoPrime : int;
private var AmmoAlt : int;
private var Money : int;
static private var currentLevel : int;
static public var levelState : String;
static private var Missions : int[] = [0, 0, 0];
static public var CurrentSpawnPointIndex : int = 1;
```

Private player information (PlayerHealth, AmmoPrime, AmmoAlt, Money) will be retrieved from the CH_PlayerStats script and set back with the SetStats function. We will use them to set the player's statistics to where they were when the player was last saved. currentLevel is a static variable that will keep track of the level needed to be loaded. levelState will tell us about the state of the game. We will use that to check whether the game is in play mode, or whether the player has failed and we need to load the dying scene. Missions is an array of integers that will keep track of mission progress. CurrentSpawnPointIndex is self-explanatory; we will use it to spawn the player at the right spawnpoint.

To save the player statistics, we need to reference the script that is attached to our player. We will use the `FindWithTag` function, which will help us to find the object with a `Player` tag out of all the objects in the scene:

```
public function SetStats( PlayerHealth : int ,AmmoPrime : int, AmmoAlt
: int, Money : int){
var PlayerStats : CH_PlayerStats = GameObject.FindWithTag("Player").
GetComponent("CH_PlayerStats");
```

We will need to create a `static` function to save the game from any script. Declare a new `static` function called `SaveGame`:

```
static public function SaveGame( ){}
```

The rest of this function will write information into the text file:

```
var file = new StreamWriter( Application.dataPath + "/Saves/SavedGame.
txt");
file.WriteLine( currentLevel );
file.WriteLine( CurrentSpawnPointIndex );
if (GameObject.FindWithTag("Player")){
var PlayerStats : CH_PlayerStats = GameObject.FindWithTag("Player").
GetComponent("CH_PlayerStats");
file.WriteLine( PlayerStats.GetHealth() );
file.WriteLine( PlayerStats.GetAmmo(0) );
file.WriteLine( PlayerStats.GetAmmo(1) );
file.WriteLine( PlayerStats.GetMoney() );
}
file.WriteLine( Missions[0] );
file.WriteLine( Missions[1] );
file.WriteLine( Missions[2] );
file.Flush();
file.Close();
```

Declare a new function called `Initialize()`:

```
public function Initialize(){}
```

We will start `Initialize` with setting level state. Create a new function called `SetLevelState`; it needs to be `static` and will take `String` as an argument, which will be assigned to the current state of the game:

```
static function SetLevelState(newState : String){
levelState = newState;}
```

Later, we will need the set and get functions for the `Missions` array and the `CurrentSpawnPointIndex` variable, so we might as well set and declare them now. In the case of the `Missions` array, it's the only one that needs both the set and get functions and an extra variable to take an index of the array:

```
static public function MissionStatusCheck ( missionIndex : int ){
    return Missions[missionIndex];}
static public function SetMissionStatus ( missionIndex : int, status :
int ){
Missions[missionIndex] = status;}
static public function SetCheckPoint( newCheckPoint : int ){
    CurrentSpawnPointIndex = newCheckPoint;}
```

Back to the `Initialize` function, we need to check whether the directory exists, and if not, create it, just like we did in the `SaveGame` function. However, in this function, we also need to check if the save file exists:

```
if(!Directory.Exists("Assets/Saves/") ){Directory.
CreateDirectory("Assets/Saves/");}
if (File.Exists(Application.dataPath + "/Saves/SavedGame.txt")){}
If file indeed exists we need to read all data from it and assign to
our variables.
var file = new StreamReader( Application.dataPath + "/Saves/SavedGame.
txt" );
currentLevel = parseInt( file.ReadLine() );
CurrentSpawnPointIndex = parseInt ( file.ReadLine() );
PlayerHealth = parseInt ( file.ReadLine() );
AmmoPrime = parseInt ( file.ReadLine() );
AmmoAlt = parseInt ( file.ReadLine() );
Money = parseInt ( file.ReadLine() );
Missions[0] = parseInt ( file.ReadLine() );
Missions[1] = parseInt ( file.ReadLine() );
Missions[2] = parseInt ( file.ReadLine() );
file.Close();
}
```

On the other hand, if the file is not yet created, we need to set the default values to variables and create a file in the directory; this means that we are starting a new game:

```
else{
currentLevel = 1;
CurrentSpawnPointIndex = 1;
PlayerHealth = 100;
```

```
AmmoPrime = 20;
AmmoAlt = 20;
Money = 0;
Missions[0] = 0;
Missions[1] = 0;
Missions[2] = 0;
var file = new StreamWriter( Application.dataPath + "/Saves/SavedGame.
txt");
file.WriteLine(currentLevel);
file.WriteLine(CurrentSpawnPointIndex);
file.WriteLine(PlayerHealth);
file.WriteLine(AmmoPrime);
file.WriteLine(AmmoAlt);
file.WriteLine(Money);
file.WriteLine(Missions[0]);
file.WriteLine(Missions[1]);
file.WriteLine(Missions[2]);
file.Close();
}
```

Now that we have all the needed variables assigned to their value, we can use them to check which scenes need to be loaded. For this, we will create a new function called `LoadingLevels()`:

```
function LoadingLevels(){
if(!Missions[0]){
Application.LoadLevel(currentLevel);
}
else if(!Missions[1]){
Application.LoadLevel(currentLevel);
Application.LoadLevelAdditive(currentLevel + 1);
}
else if(!Missions[2]){
Application.LoadLevel(currentLevel);
Application.LoadLevelAdditive(currentLevel - 1);
}
}
```

> `Directory` is a part of the `System IO` library.

The completed `WorldManager` script is as follows:

```
static public function SaveGame( ){
var file = new StreamWriter( Application.dataPath + "/Saves/SavedGame.
txt");
file.WriteLine( currentLevel );
file.WriteLine( CurrentSpawnPointIndex );
if (GameObject.FindWithTag("Player")){
var PlayerStats : CH_PlayerStats = GameObject.FindWithTag("Player").
GetComponent("CH_PlayerStats");
file.WriteLine( PlayerStats.GetHealth() );
file.WriteLine( PlayerStats.GetAmmo(0) );
file.WriteLine( PlayerStats.GetAmmo(1) );
file.WriteLine( PlayerStats.GetMoney() );
}
file.WriteLine( Missions[0] );
file.WriteLine( Missions[1] );
file.WriteLine( Missions[2] );
file.Flush();
file.Close();
}
public function Initialize(){
if( !Directory.Exists( "Assets/Saves/" ) ){Directory.CreateDirectory(
"Assets/Saves/"  );}
if (File.Exists(Application.dataPath + "/Saves/SavedGame.txt")){
var file = new StreamReader( Application.dataPath + "/Saves/SavedGame.
txt" );
currentLevel = parseInt( file.ReadLine() );
CurrentSpawnPointIndex = parseInt ( file.ReadLine() );
PlayerHealth = parseInt ( file.ReadLine() );
AmmoPrime = parseInt ( file.ReadLine() );
AmmoAlt = parseInt ( file.ReadLine() );
Money = parseInt ( file.ReadLine() );
Missions[0] = parseInt ( file.ReadLine() );
Missions[1] = parseInt ( file.ReadLine() );
Missions[2] = parseInt ( file.ReadLine() );
file.Close();
}
else{
File.Create(Application.dataPath + "/Saves/SavedGame.txt");
currentLevel = 1;
CurrentSpawnPointIndex = 1;
PlayerHealth = 100;
AmmoPrime = 20;
AmmoAlt = 20;
```

```
Money = 0;
Missions[0] = 0;
Missions[1] = 0;
Missions[2] = 0;
var file = new StreamWriter( Application.dataPath + "/Saves/SavedGame.
txt");
file.WriteLine(currentLevel);
file.WriteLine(CurrentSpawnPointIndex);
file.WriteLine(PlayerHealth);
file.WriteLine(AmmoPrime);
file.WriteLine(AmmoAlt);
file.WriteLine(Money);
file.WriteLine(Missions[0]);
file.WriteLine(Missions[1]);
file.WriteLine(Missions[2]);
file.Close();
}
}
static function SetLevelState(newState : String){
levelState = newState;
}
static public function MissionStatusCheck (missionIndex : int){
    return Missions[missionIndex];
}
static public function SetMissionStatus( missionIndex : int, status :
int){
    Missions[missionIndex] = status;
}
static public function SetCheckPoint( newCheckPoint : int ){
    CurrentSpawnPointIndex = newCheckPoint;
}
function LoadingLevels(){
if(!Missions[0]){
Application.LoadLevel(currentLevel);
}
else if(!Missions[1]){
Application.LoadLevel(currentLevel);
Application.LoadLevelAdditive(currentLevel + 1);
}
else if(!Missions[2]){
Application.LoadLevel(currentLevel);
Application.LoadLevelAdditive(currentLevel - 1);
}
}
```

`Initialize` does not need to be `static` as it will be called only from within the script. `Initialize` will be used to initialize the game, by reading from the saved file and making all the arrangements to start the gameplay.

Loading with checkpoints

One way to keep track of our game status is by checking the completed missions and loading levels according to it. But, let's return to the `Initialize` function and call `LoadingLevels` from it.

```
...
LoadingLevels();
```

After the level is loaded, we need to spawn the actual player in the specified checkpoint. Declare a new function called `SpawnPlayer`, which will take `SpawnIndex` as an argument:

```
static function SpawnPlayer(spawnIndex : int){}
```

Based on the sent variable, we need to find `SpawnPoint` in the level:

```
var SpawnPlace : GameObject = GameObject.FindWithTag("spawnPoint" +
CurrentSpawnPointIndex) ;
```

We also need to specify the prefab of the player that will be spawned. To be used in the `static` function, we need to declare the `static` variable. However, `static` variables cannot be set in the **Inspector** view, therefore we need to create a pair of variables—one `public` and one `static`:

```
static public var PlayerPrefab : GameObject;
public var PlayerPrefab2 : GameObject;
```

In the `Awake` function, we need to assign the `public` variable to the `static` variable:

```
PlayerPrefab = PlayerPrefab2;
```

Now that we have that, two series of checks need to be done—we need to check if the `SpawnPlace` that we issued really exists and if `PlayerPrefab` is set, then all that's left to do is to instantiate the player:

```
if(SpawnPlace){
    if(PlayerPrefab){
    Instantiate (PlayerPrefab, SpawnPlace.transform.position,
    Quaternion.identity);}
    else
    Debug.LogError("Player Prefab is not set");}
else
Debug.Log("Spawnpoint wasn't found");}
```

Right after the player is spawned, we need to adjust his statistics with a `SetStats` function by referencing the `CH_PlayerStats` script and calling modifying functions there:

```
public function SetStats( PlayerHealth : int ,AmmoPrime : int, AmmoAlt
: int, Money : int){
var PlayerStats : CH_PlayerStats = GameObject.FindWithTag("Player").
GetComponent("CH_PlayerStats");
PlayerStats.AddHealth(PlayerHealth, 0);
PlayerStats.AddAmmo(1,AmmoPrime, 0);
PlayerStats.AddAmmo(2,AmmoPrime, 0);
PlayerStats.AddMoney(Money, 0);}
```

Now, we have all we need to finish the `Initialize` function. The first thing that we need to check is if the level with which we are trying to create our player has finished loading and that the `SpawnPoint` exists. Thankfully, we can check level progress with the `GetStreamProgressForLevel` function. This whole check needs to be done inside the `while` loop to be able to constantly check if the level has finished loading. If that doesn't work, we will simply go through the `while` loop again, but that will be too expensive to call each frame. To make sure that it doesn't, we will use the `yield` command and call the `WaitForSeconds` function with one second delay.

```
while(1){
var SpawnPlace : GameObject = GameObject.FindWithTag("spawnPoint" +
CurrentSpawnPointIndex);
if(Application.GetStreamProgressForLevel(currentLevel) == 1 &&
SpawnPlace != null){
SpawnPlayer(CurrentSpawnPointIndex);
SetStats(PlayerHealth, AmmoPrime, AmmoAlt, Money);
break;}
else{
yield WaitForSeconds(1);}
```

> `yield` is a coroutine; a function that can suspend its execution until requirements specified in `YieldInstructions` have been met.

One other function that we need to create is called `SetMissionStatus`, which will help us to set a new value to the `Mission` array:

```
static public function SetMissionStatus( missionIndex : int, status :
int){
    Missions[missionIndex] = status;
}
```

The first thing that we should have done is to set the level state to `Playing`. At the beginning of the `Initialize` function, add the following line of code:

```
SetLevelState( "Playing");
```

To finalize the `WorldManager` script, we will create menu buttons with GUI. By checking the current level and state, we will show appropriate menu options.

Continuing with the `WorldManager` script, we will add the following code snippet:

```
function OnGUI(){
if (Application.loadedLevel == 0 ){
if(GUI.Button(Rect(Screen.width - Screen.width/2, Screen.height -
Screen.height/2, 180, 20), "New Game/ Continue")){
Initialize();
}
}
else if (levelState == "Dead"){
if(GUI.Button(Rect(Screen.width - Screen.width/2, Screen.height -
Screen.height/2, 180, 20), "Load last checkpoint")){
Initialize();
}
if(GUI.Button(Rect(Screen.width - Screen.width/2, Screen.height -
Screen.height/2 + 30, 180, 20), "Go To Main Menu")){
Application.LoadLevel(0);
}
}
}
```

The following screenshot shows what our simple frontend should look like:

Now that we have finished with the `WorldManager` script, we need to add a few lines to the `Button` script. The first addition will be inside the `OnCollisionEnter` function. When the projectile activates `ButtonType 1` (inside the `else if` statement), we need to set the mission to complete, set a new level, set a new checkpoint, save the game, and load the new level.

In the `Button` script, add the following code snippet:

```
...
WorldManager.SetMissionStatus(0, 1);
WorldManager.SetCurrentLevel(2);
WorldManager.SetCheckPoint(2);
WorldManager.SaveGame();
Application.LoadLevelAdditive(2);
...
```

As we have made `WorldManager` a `static` class, we don't need to bother referencing it.

Now, we need to do the same thing, but only with the `Update` function. Inside the `else if` statement when we are comparing `ButtonType` to `0`, we need to check if we haven't completed the first mission and activated both the buttons in the `Button` script:

```
if(!WorldManager.MissionStatusCheck(1) && Activated == 2){
WorldManager.SetMissionStatus(1, 1);
WorldManager.SetCurrentLevel(3);
WorldManager.SetCheckPoint(3);
WorldManager.SaveGame();
Application.LoadLevelAdditive(3);
```

GameLoader

To load our managers, we need to create a loader script, which we will call `GameLoader`. `GameLoader` is essentially a singleton; it's being called only once, when the game starts. It calls the `GetInstance` functions in managers and is never referenced after the game is loaded.

Now, we will create a singleton mentioned in the preceding paragraph and call it `GameLoader`. Perform the following steps:

1. Create a script called `GameLoader`.
2. In the `Start` function, we need to call the `GetInstance` function from `WorldManager` and import the `System IO` libraries that need to be used in other scripts.

The following code snippet will go into the `GameLoader` script:

```
import System.IO;
function Start (){
WorldManager.GetInstance();
}
```

Dynamic camera

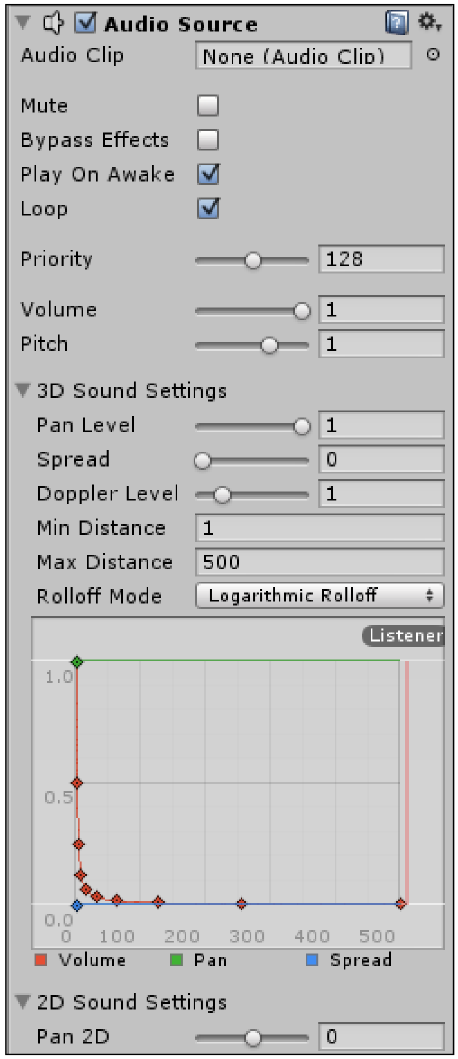
Being able to create a character has its pluses and minuses; one of them is a problem with a camera. Whenever we start with a level that doesn't have a camera, we need to create it dynamically. Perform the following steps:

1. Go to the **CH_Controller** script. Create a new `public` variable—`createCameraPrefab`. It will contain a prefab of a camera that we want to instantiate.
2. Inside the `else` statement from the `Awake` function, we need to check if an object with a `MainCamera` tag exists or not.
3. If it doesn't exist, we need to instantiate it using `createCameraPrefab` that we have declared before, and put it inside `CPrefab`.

Audio

There are two things that we need to keep in mind when dealing with audio in Unity—**Audio Listener** and **Audio Source**. **Audio Source** is a component that uses the transform information of an object to emit sound from a location. **Audio Listener** picks up all the sounds and serves as a microphone. Listeners are usually attached to cameras; however, sometimes attaching them to the character can give better results. Each scene can have only one listener and any number of sources. It is important to remember that Unity will issue an error if it finds more than one listener in the scene.

Audio Source has a few interesting parameters that we need to look at. The first of them is **Audio Clip**; this specifies a sound track that is to be played by a selected **Audio Source**. The other two parameters are **Play On Awake** and **Loop**. If we are not creating ambient sounds, it is recommended to uncheck them; it would be easier to control them through scripts.



The first thing that we need to do is to attach sound to a controllable character. Perform the following steps:

1. Open **CH_Controller** and declare three `public` variables of a `AudioClip` type.
2. Inside the `Movement` function where we apply movement to the player, we need to check if any audio is playing and the character is not moving.

3. At the beginning of the `AltShooting` function, we need to start playing the shooting sound:

```
public var ShootingAlt : AudioClip;
public var ShootingMain : AudioClip;
public var FootstepSound : AudioClip;
```

4. The following code snippet goes to the `Movement` function:

```
...
if (isGrounded){
this.transform.Translate((MoveDirection.normalized * Speed) *
Time.deltaTime);
if(audio.isPlaying == false && MoveDirection != Vector3.zero){
audio.clip = FootstepSound;
audio.Play();
}
}
...
```

5. The following code snippet goes to the `AltShooting` function:

```
...
if (audio.clip == FootstepSound || audio.isPlaying == false ||
(audio.isPlaying == false && audio.clip == ShootingAlt)){
audio.clip = ShootingAlt;
audio.Play();
}
...
```

We are assigning an audio clip that will be played from the character's audio source to play a footsteps sound.

From the character, we will switch to the environmental object, like the fan that is located in the second level. Perform the following steps:

1. Create a new script called `fan_rotation`.
2. We need to make this fan spin around constantly.
3. In the `Awake` function from the `fan_rotation` script, we need to start playing the fan sound and make it loop, as follows:

```
public var FanSound : AudioClip;
function Awake(){
    audio.clip = FanSound;
    audio.loop = true;
```

```
    audio.Play();
}
function Update () {
transform.Rotate(Vector3.right * Time.deltaTime * 100.0);
}
```

4. Create a new script called `deathTrigger`; it will be used as a killing volume in level 3, when the player falls in a river.
5. In the `Awake` function, we will assign a clip to the audio source.
6. We need `OnTriggerEnter` for decreasing the player's health, destroying the player's object, playing the audio sound, loading `EndGameLevel`, and changing the level state to `Dead`.
7. The following code snippet goes into the `deathTrigger` script:

```
public var WaterSplash : AudioClip;
function Awake(){
audio.clip = WaterSplash;
}
function OnTriggerEnter(other : Collider){
if(other.gameObject.tag == "Player"){
other.gameObject.GetComponent("CH_PlayerStats").AddHealth(-100);
WorldManager.SetLevelState("Dead");
audio.Play();
Destroy(other.gameObject);
Application.LoadLevel(4);
}
}
```

Audio manager

To design the audio manager, perform the following steps:

1. Create a new script called `AudioManager`.
2. As in world manager, we need to create the `GetInstance` function.
3. Now, we need to create a music player that will take care of switching music.
4. To initialize our audio manager, we need to call the `GetInstance` function from `GameLoader`.

5. In the `AudioManager` script, add the following code snippet:

```
static private var instance : AudioManager = null;
public var auMusicPlaying1 : AudioClip;
public var auMusicPlaying2 : AudioClip;

static public function GetInstance(){
if(!instance){
instance = FindObjectOfType(AudioManager);
if(!instance){
Debug.LogError("AudioManager does not exist");
return instance;
}
}
function MusicPlayer(){
if ( !audio.isPlaying ){
if(audio.clip == auMusicPlaying1)
audio.clip = auMusicPlaying2;
else
audio.clip = auMusicPlaying1;
audio.Play();
}
}
function Update(){
MusicPlayer();
}
function Awake(){
audio.clip = auMusicPlaying1;
}
```

6. In the `GameLoader` script, add the following code snippet:

```
function Start (){
WorldManager.GetInstance();
AudioManager.GetInstance();
}
```

Summary

All necessary sounds can be found in the `Audio` folder and can be assigned without detailed instructions.

Game managers are important if we wish to make everything fast and organized. The ultimate advantage of managers is their reusability; having created a manager once, it can be reused as a template in future projects.

In the next chapter, we will look into the basics of programming—**Artificial Intelligence** (**AI**) for video games, talk about behaviors, path finding, obstacle avoidance, and so on.

7

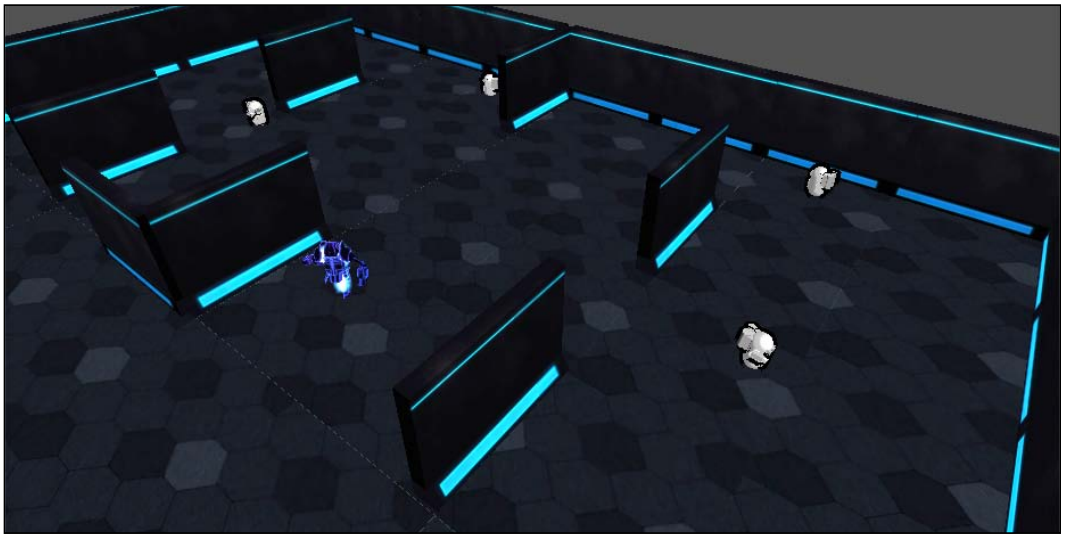
Introduction to AI Pathfinding and Behaviors

When it comes to designing a game, a person has to take a step back and ask himself, how is this game going to be played? Is it a single-person puzzler, a **first-person shooter** (**FPS**) with a campaign, a **third-person shooter** (**TPS**) action adventure, a fighter, or a **real-time strategy** (**RTS**)? Each of these can, and usually do, have a form of **Artificial Intelligence** (**AI**). There are many forms of AI that can present themselves and they appear in many ways such as pathfinding, collision detection, item collection, cover, animations, and so on. Pretty much everything that we take for granted being humans, and can do in a game without thinking about it, has to be carefully thought about and crafted to work together as if the humans were indeed in control of the AI. So, with such a huge and complex topic to explore and present, the best and most useful aspect is to show an example of enemies with behaviors and waypoint path navigation. First we will explore setting up the basic waypoint pathfinding without behaviors.

In this chapter we will look into the following topics:

- Pathfinding with waypoints
- Writing a pathfinding script for robots
- Making robots shoot and interact with a player
- Writing stats scripts for robots

- Teaching AI different kinds of behaviors



Let's get started.

Simple waypoint pathfinding

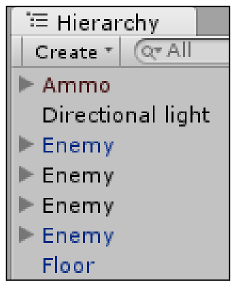
When it comes to pathfinding, there are many types and each of them has a different function and ability. For pathfinding, there are various algorithms such as breadth-first search, depth-first search, Dijkstra, and A* for pathing waypoint and nav-mesh. A **nav-mesh** is a mesh, which has its faces triangulated to form surfaces that can be transversed. Games such as Uncharted 2 and Killzone 3 use this form of navigation. With that being said, it is not uncommon to see the enemies navigating along using the different types of pathfinding techniques in different situations. The pathfinding type that we will use, as I have mentioned in the introductory section, is going to be a static waypoint navigation system.

In this section we will look into the following:

- Setting up the hierarchy
- Writing the waypoint display script
- Setting up the path arrays
- Creating the `aiSimplePath` script

Setting up the hierarchy

What we need to do, before we move any further, is to set up the scene hierarchy. In the `Assets` folder for this chapter, there is an enemy prefab and a test level called "Scene". Drag the test level and the four enemies into the **Hierarchy** view. Once this is done, you should see the enemies and the level, as shown in the following screenshot:



Now we need to decide our paths as well.

Writing the waypoint display script

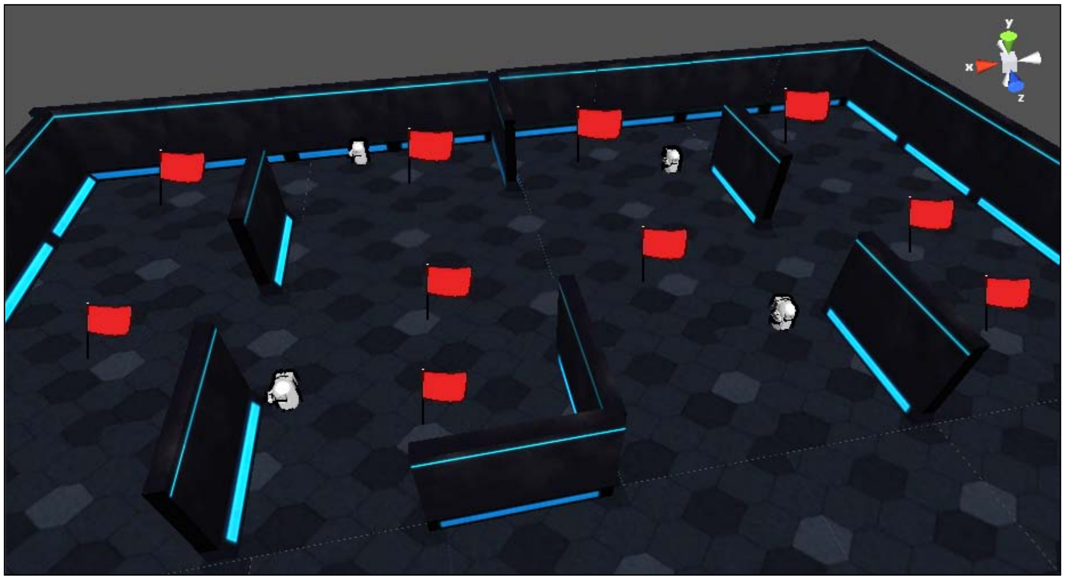
The script that we will be creating in this section will allow the user to have a gizmo icon, which is representative of our waypoints. It will allow us to turn the gizmo on and off in the scene view and game view as well. To make this happen, all we need to do is to create a new script called `WaypointNode_Display`. In here we need to call a single function called `OnDrawGizmos`. Then, in order to have the icon displayed, we need to set up the location of the image. For us, we want that location to be the location of the attached `gameObject`. The texture is located in the `Textures` folder and is called `waypointnode_icon`. To do all this, we use the `Gizmos.DrawIcon` function as follows:

```
function OnDrawGizmos(){
    Gizmos.DrawIcon(transform.position, Application.dataPath +
    "\History\Chapter 7\Custom_textures\waypoint.png");
}
```

Now that we have this script written, we can set up our waypoints for our paths. Perform the following steps:

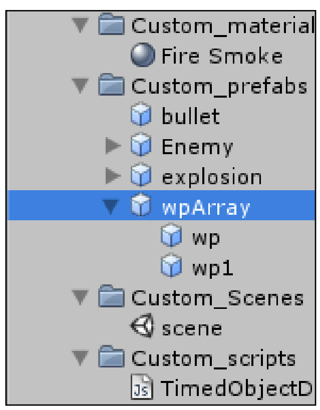
1. Create waypoint arrays.
2. Establish communication between them.
3. Make robots patrol the area using waypoints.

Have a look at the following screenshot:



Setting up the path arrays

If you would like to just move ahead to writing the pathfinding script, drag the path examples, located in the **Assets** folder for this chapter, into the **Hierarchy** view. These paths are already set up for you and have the waypoints set as well. We need to write the script before we can assign the paths to the enemies. After you do this, skip down to writing the `aiSimplePath` script.



For each enemy that is in the scene, we want to create an empty `gameObject`, which will be its path array and house the waypoints for that path. A path is not necessary for every enemy. Enemies can share paths and have multiple paths, but it will be easier to showcase the enemies' pathfinding and behavior if they have their own paths.

We want to create a waypoint prefab and drag the **waypointPointnode_Display** script onto it. After that, we will create waypoints for the enemies. So, go ahead and place them around where you want. Keep in mind when you are placing them, the order in which you place them, and where and which path they are to be associated with. Once that is done, parent the appropriate waypoints to their path `gameObject` variables. We must now write the `aiSimplePath` script.

Creating the aiSimplePath script

The `aiSimplePath` script will handle the enemy's navigation through waypoints, and later on, the tracking of injured bots, the player, and ammo. This path system will allow our enemy to be able to travel through the path and once at the last waypoint, he will have three options. The first option is to be able to loop back to the beginning of the first waypoint and the second one is to reverse and go back to the first waypoint along the path that the enemy came. A third option does present itself when an enemy is set to reverse. When the enemy is at the beginning of the array again and if `reverseLoop` is set to `true`, the enemy will then travel through the array to the last waypoint and then back again to the beginning, and so forth. With these three options, we can have the enemies acting differently from each other.

Declaring variables

First, let's define our path variables. To do this, we need to:

- Declare variables for our path
- Define the object to be pursued (in this case, the player)
- Define the speed at which we want the enemy to travel

Three of these variables, as mentioned previously, will represent the navigation of the path—`reverse`, `reverseLoop`, and `looping`. They are of the `boolean` type and `public` as well.

The next set deals with information gathering. We need to have one variable for housing the waypoint paths of the enemy, one variable that for holding all the waypoints in those paths, another one for the waypoint that we would like to get to and, lastly, a variable for the direction in which we are navigating through the waypoint array.

The enemyPath variable will be a Transform list and public. The waypointArray variable needs to be an array and, for default, set it to new Array and make it private. Create a third variable called currentWaypoint, which will handle the tracking of the current waypoint to pursue. It will be an integer and private as well. The last path variable to create is the arrayDirection variable, which will keep track of the direction in which we are navigating through the array:

```
public var reverse : boolean;
public var reverseLoop : boolean;
public var looping : boolean;
public var enemyPath : Transform[];
private var waypointArray : Array = new Array();
private var currentWaypoint : int = 0;
private var arrayDirection : int;
```

The object to be pursued is called player, and it is public and of the GameObject type. Next, we need to define the speed variable to set the speed at which the player will move, make it a float and public, too. Our last variable for now is the stopRobot variable, and it is boolean and private:

```
public var player : GameObject;
public var speed : float;
private var stopRobot : boolean;
```

Starting up functions

Once all the variables are in place, we will start our functions.

The Awake function will get all the waypoints in all the enemy paths through the enemyPath variable and add them to the waypoint array. It will then set the arrayDirection variable to the length of the waypoint array. A for loop is then used to loop through each path and a second for loop inside of the first one is used to grab the waypoints. Then, each waypoint in each path is added to the waypoint array. Naming convention for the nodes will make a difference, as it will determine the order in which enemies will be moving from one to another.

```
function Awake(){
    for(var path in enemyPath){
        for(var waypoint in path){
            waypointArray.Add(waypoint);
        }
    }
    arrayDirection = waypointArray.length;
}
```

After the `Awake` function, we will get down to the `EnemyPath` function. This is a mean beefy piece of code that allows our enemies to traverse the path.

Traversing the path

The `EnemyPath` function will set the pathfinding for the enemies in various situations, such as patrolling, pursuing the enemy, locating ammo, and locating injured enemies.

At the beginning of the function, we first need to make sure that our pathfinding variables—`reverse`, `reverseLoop`, and `looping`—are set. Remember, when `reverse` is set to `true`, `looping` is `false`. `reverseLoop` can only be `true` if `reverse` is set to `true`, and when `looping` is set to `true`, `reverse` and `reverseLoop` are `false`.

Next, we will check if the robot has come to a complete stop for whatever reason, and if not, we will start our pathfinding. We need to define three variables—`velocity` (to deal with the speed that our enemy will move at), `moveDirection` (to deal with the facing direction of the enemy), and `Target` (to define the place where the enemy wants to go). All are of the `Vector3` type.

The code for the beginning of the function is as follows:

```
function EnemyPath(){
    if(reverse){looping = false;}
    if(reverseLoop){looping = false; reverse = true;}
    if(looping){reverse = false; reverseLoop = false;}
    if(!stopRobot){
        var velocity : Vector3 = rigidbody.velocity
        var moveDirection : Vector3;
        var Target : Vector3;
}
}
```

As mentioned previously, our enemy will be searching for the player, ammo, and injured bots, but before we get to writing that code, we will first just implement the following of the waypoints, and once the `aiSimpleBehaviour` script is written, we will return to this script and add the rest of the code.

In that case, we set our enemies to patrol. Our enemy will patrol through the waypoints of our waypoint array. We will use our `arrayDirection` and `currentWaypoint` variables to determine how we navigate through the waypoints. We will perform the following steps:

1. If the `arrayDirection` variable equals our `waypointArray.length` and if our current waypoint is less than that, then it means that we are increasing through the array. Inside of the `if` statement, we need to set the `Target` and `moveDirection` variables. The `Target` variable becomes the current waypoint position from the waypoint array, and `moveDirection` is `Target` minus the enemy's position.
2. Next, we will check whether we are at the last waypoint using the `Vector3.Distance` function to check our position against the last waypoint's position. If we are not at the last waypoint, and if we are not looping or reversing, we will set the enemy's `velocity` to `Vector3.zero` (This will stop the enemy's momentum).
3. If however, we are not at the last waypoint and are within the range of 1 from our next point, we will increment the current waypoint. (We use the magnitude function on our `moveDirection` variable to determine if the distance between the enemy and its target is in range. In this case, that range value is 1.)

   If the magnitude between the enemy and the target is not less than `1` and therefore not in range, normalize our `moveDirection` variable, multiply with speed, and assign it to `velocity`.

   The code for increasing through the waypoint array `if` statement is as follows:

```
if (arrayDirection == waypointArray.length && currentWaypoint <
arrayDirection){
    Target = waypointArray[currentWaypoint].position;
    moveDirection = Target - transform.position;
if (Vector3.Distance(waypointArray[waypointArray.length-
1].position, transform.position) < 1 && !looping && !reverse &&
!reverseLoop){
    velocity = Vector3.zero;
    }
    else if(moveDirection.magnitude < 1){
    currentWaypoint++;
    }
    else{
    velocity = moveDirection.normalized * speed;
    }
}
```

That ends the increasing through the waypoint array `if` statement. Right after it though, we need an `else if` statement, which will determine if we are decreasing through the waypoint array.

4. Next, we will check the `arrayDirection` variable again. If it is equal to `0` and if our current waypoint is greater than our array direction, it means that we are decreasing through the array. The `Target` and `moveDirection` variables can stay the same, but our next `if` statement will have different parameters.
5. We need to check that our range (remember, `moveDirection.magnitude`) is less than `1`, our current waypoint is equal to `0`, and that we are not doing a reverse loop. If all of these are true, we will set our `velocity` variable to `Vector3.zero`.
6. If our range to our target is less than `1`, we will decrement the `currentWaypoint` variable. If the enemy is not in range of the target, we will normalize our `moveDirection` variable, multiply with speed, and assign it to `velocity`.

   The code for the decreasing through the waypoint array `if` statement is as follows:

```
else if(arrayDirection == 0 && currentWaypoint >= arrayDirection){
    Target = waypointArray[currentWaypoint].position;
    moveDirection = Target - transform.position;
    if(moveDirection.magnitude < 1 && currentWaypoint == 0 &&
    !reverseLoop) {
            velocity = Vector3.zero;
            }
            else if(moveDirection.magnitude < 1 ) {
            currentWaypoint--;
    }
    else {
        velocity = moveDirection.normalized * speed;
    }
}
```

   Now we need to check if our enemy is looping or going in reverse in the `else` statement that follows right after the `else if` statement.

7. If looping, we will set the `currentWaypoint` variable to `0`.
8. If reversing looping, we will set the `arrayDirection` variable back to `waypointArray.length` and increment the `currentWaypoint` variable.
9. If just reverse, set the `arrayDirection` to `0` and decrement the `currentWaypoint` variable.

10. Lastly, at the bottom of the `!stopRobot if` statement, give the enemy's `rigidbody.velocity`, the `velocity` value and use the `transform.LookAt` function with the `Target` variable as the parameter to make an enemy look at another enemy.

    The code in the `else` statement and the final lines of the function are as follows:

```
else{
    if(looping){
        currentWaypoint = 0;
    }
    else if(reverse){
    if (arrayDirection == 0 &&
        Vector3.Distance(waypointArray[0].position,
        transform.position) < 1 && reverseLoop){
                arrayDirection = waypointArray.length;
                currentWaypoint++;
    }
    else if(currentWaypoint == waypointArray.length){
                arrayDirection = 0;
                currentWaypoint--;
        }
    }
}
rigidbody.velocity = velocity;
transform.LookAt(Target);
```

Shutting down the robot

The `ShutDownRobot` function will set the `stopRobot` variable to the `stop boolean` variable of the function. If `stopRobot` is true, set `velocity` to `Vector3.zero`. The code for the `ShutDownRobot` function is as follows:

```
function ShutDownRobot(stop : boolean){
    stopRobot = stop;
    if(stopRobot){
    velocity = Vector3.zero;
    }
}
```

The `Update` function is used to call the `EnemyPath` function as follows:

```
function Update(){
    EnemyPath();
}
```

The `aiSimplePath` script is now complete, at least for the time being. We will set up **Inspector** for our enemies next.

Hooking up the aiSimplePath script on Inspector

We can now drag the **aiSimplePath** script from the **Project** view to the **Inspector** view of our enemies. You should see that our path variable will allow us to drag our respective paths to our enemies. We also need to give a speed to our enemy. After that, you can set the type of navigation of the waypoints for the enemy to do. Press **Play** in the editor, and if you have everything set up correctly, your enemy should travel along the path that you have specified for him.

We will have to come back to this script later, and add a couple of functions and some lines of code to make the enemy locate ammo and injured bots, and pursue the player. For the time being, however, the next section will deal with enemy statistics, shooting, and behaviors.

Enemy statistics, shooting, and behaviors

Now that we have our enemies moving and following a path, we should set up some statistics for them, and allow them to take damage and fire bullets. Then at the end of this section, we will write the `aiSimpleBehaviour` script. This script will give some more believability to our enemies in how they react to the situations they are in and give them unique behaviors. First up to be written is the `enemyStats` script.

In this section we will look into the following:

- Creating the `enemyStats` script
- Hooking up the `enemyStats` script on **Inspector** of each enemy
- Creating the `Shoot` script
- Hooking up the `Shoot` script on **Inspector** of each enemy
- Writing the `aiSimpleBehaviour` script
- Hooking up the `aiSimpleBehaviour` script on **Inspector** of each enemy

The enemyStats script

The `enemyStats` script will hold and track our enemy's health and ammo, and shut down the enemy if his health drops to zero. This script will also return the current values of health and ammo, and increment or decrement the ammo and health.

Setting up variables

There are a few variables for this script. We have one to represent the health, one for the ammo, and another one that will be the reference variable to the `aiSimplePath` script.

The `health` variable should be `public` and an integer; the same can be done for the `ammo` variable. The `pathScr` variable can be `private`, but make sure that it has as its type the name of the `aiSimplePath` script as follows:

```
public var health : int = 100;
public var ammo : int = 100;
private var pathScr : aiSimplePath;
```

Once we are finished with the variables, it's time to write our functions.

Setting up functions

All the functions in this script are relatively short and mainly consist of one line. Up first is the `Awake` function.

The `Awake` function makes the `pathScr` variable reference to the `aiSimplePath` script:

```
function Awake(){pathScr = gameObject.GetComponent("aiSimplePath");}
```

Retrieving functions

The `Get` functions will help us retrieve health and ammo information from robots. The retrieving functions are as follows:

- `The GetHealth` function returns the current `health` value:
  ```
  function GetHealth(){return health;}
  ```
- `The GetAmmo` function returns the current `ammo` value:
  ```
  function GetAmmo(){return ammo;}
  ```
- `The RepairBot` function will increase the `health` of an enemy by `2` when it is called:
  ```
  function RepairBot(){health += 2;}
  ```

Manipulation functions

The manipulation functions will allow us to manipulate health and ammo. They are as follows:

- The `ReceiveAmmo` function will increase `ammo` by the amount given in the function's parameter variable—`ammoAmount`:
  ```
  function ReceiveAmmo(ammoAmount : int) {ammo += ammoAmount; }
  ```

- The `DecrementAmmo` function will decrease `ammo` by the amount given in the function's parameter variable—`decrementAmount`:

  ```
  function DecrementAmmo(decrementAmount : int) {ammo -=
  decrementAmount;}
  ```

- The `DecrementHealth` function will decrease `health` by the amount given in the function's parameter variable—`decrementAmount`:

  ```
  function DecrementHealth(decrementAmount : int) {health -=
  decrementAmount;}
  ```

- The `CheckCurrentHealth` function will make sure that `health` is not greater than `100` and not less than `0`. It will also shut down the robot if the enemy's health does reach `0`. To do this, we have one `if` statement and the `Mathf.Clamp` function. If the enemy's health decreases below `0`, we will call the `ShutDownRobot` function from the path script with the function parameter of `true`.

  The complete `CheckCurrentHealth` function should look like the following code snippet:

  ```
  function CheckCurrentHealth(){
  health = Mathf.Clamp(health, 0,100);
      if(health == 0){
          pathScr.ShutDownRobot(true);
      }
  }
  ```

- The `CheckAmmo` function will make sure that `ammo` is not greater than `100` and not less than `0`. We will use two `if` statements to check the ammo count. One `if` statement will make sure that `ammo` is not greater than `100` and the other `if` statement will make sure that the `ammo` count is not less than `0`.

  The `CheckAmmo` function should look like the following code snippet:

  ```
  function CheckAmmo(){
      ammo = Mathf.Clamp(ammo, 0, 100);
  }
  ```

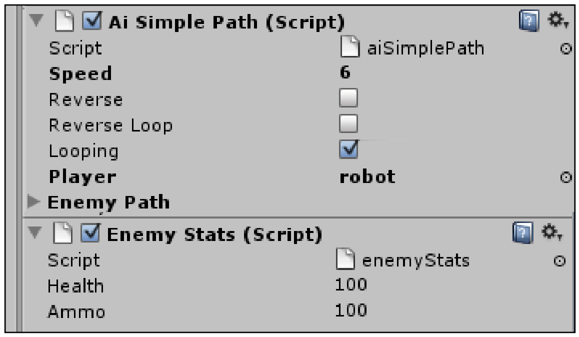
The `Update` function calls the `CheckCurrentHealth` and `CheckAmmo` functions as follows:

```
function Update (){
    CheckCurrentHealth();
    CheckAmmo();
}
```

Congrats, the `enemyStats` script is complete. We will not have to revisit this script in the future and we can now stop our enemy if their health drops to zero.

Hooking up the enemyStats script on Inspector

To test the script now, drag it onto the enemies and press **Play**. Go over to one enemy and make the `health` variable `0`. The enemy should stop. If you increase it even to 1, the enemy should resume its pathfinding.



The Shoot script

The `Shoot` script will set up the type of weapon that the enemy is firing with, where the projectiles are to be instantiated at, how fast they are to be fired, and the amount of ammo each projectile fired uses up.

Setting up the script

The following are the variables that we need to put together in this script:

- Weapon variables
- Shooting variables
- Script reference variable

There is an enumeration as well for our weapon type. The weapon variables are all `public`. The `EnemyWeapon` variable has the type of `weaponType`. The `currentWeapon` variable determines the weapon that an enemy is using from `enum Weapon`. The enemies can have weapons such as `FlameThrower`, `shortRange`, and `longRange`. The `projectile` variable is of the `GameObject` type. The last `public` variable is `launcherLoc`, which is for the launcher's location and is of the `Transform` type.

The shooting variables consist of the speed at which the bullet will fly, the amount of ammo consumed when fired, the cooldown time between each bullet fired, the initial timer for the separation, and the `canShoot` variable. `speed` is `public` and `float`; `ammoConsumeValue` is an integer; `bulletSeparationTime` is `public` and `float`; `currentTimer` is `private` and `float`; and `canShoot` is `private` and `boolean`. The default value of `canShoot` should be set to `false`.

The script reference variable is to reference the `enemyStats` script. It can be `private`, and make sure that its type is the name of the `enemyStats` script.

`enum` is for the `weaponType`. In here, put the different types of weapons that the enemy could possibly use:

```
public var EnemyWeapon : weaponType;
public enum Weapon{
FlameThrower,
shortRange,
longRange
}
public var currentWeapon : int;
public var projectile : GameObject;
public var launcherLoc : Transform;
public var speed : float;
public var AmmoConsumeValue : int = 1;
public var bulletSeparationTime : float = 1;
private var currentTimer : float = 0;
private var canShoot : boolean = false;
private var enemyStatScr : enemyStats;
enum weaponType{
    Projectile,
    Special
}
```

The `Awake` function will handle the script reference variable—`enemyStatScr` with the `enemyStats` script.

Writing shooting functionality

There are two functions that we are going to write in this section—`Shoot` and `incrementTime`. These functions will determine if an enemy can shoot and what type of weapon to use. They also manage bullet separation time. Let's begin.

The `Shoot` function will check if the enemy can shoot, and if he can, it will "fire" the projectile based upon the weapon type. After checking if he can shoot, the `incrementTime` function is called and it handles bullet separation.

If the weapon is `shortRange`, it checks if `weaponType` is `special`, and in our case it is. It's `FlameThrower`. We will set the `canShoot` variable to `false` so that bullet separation can happen. The `FlameThrower` emits a flame from the specified launcher and turns on a collider to handle particle collision, and lastly, we will decrement the ammo count based upon the specified ammo usage for the weapon.

If the weapon is `longRange`, we will again set the `canShoot` variable to `false` for bullet separation, then we will create a new bullet with the `Instantiate` function. We will create the projectile at the muzzle's position with the muzzle rotation. Then we will take `rigidbody.velocity` and multiply it by our bullet's `speed`, and assign that to `tempBullet.rigidbody.velocity`. Then we will decrement the ammo count.

Lastly, if we cannot shoot, we will put an `else` statement at the end, and in this case, with the flamethrower, we will turn off the flame emitter and disable the particle collision detection.

The code for the `Shoot` function is as follows:

```
function Shoot(fire : boolean){
    if(fire){
        incrementTime();
        if(canShoot){
            if(currentWeapon == Weapon.shortRange){
                if(EnemyWeapon.Special && currentWeapon ==
                Weapon.FlameThrower){
                    canShoot = false;
                    projectile.particleEmitter.emit = true;
                    projectile.collider.enabled = true;
                    enemyStatScr.DecrementAmmo(AmmoConsumeValue);
                }
            }
            if(currentWeapon == Weapon.longRange){
                canShoot = false;
                var tempBullet : GameObject = Instantiate(projectile,
                            launcherLoc.position,
                            launcherLoc.rotation);
                if((rigidbody.velocity).magnitude > 1){
                    tempBullet.rigidbody.velocity =
                    rigidbody.velocity * speed;
                }
                else{
                    tempBullet.rigidbody.velocity = (rigidbody.
velocity *
                    speed) * 1000;
                }
                    enemyStatScr.DecrementAmmo(AmmoConsumeValue);
            }
        }
    }
    else {
```

```
            if(currentWeapon == Weapon.FlameThrower)    {
                projectile.collider.enabled = false;
                projectile.particleEmitter.emit = false;
            }
        }
    }
```

The `incrementTime` function will increment a timer up to the bullet separation time by `Time.deltaTime` added to itself to get seconds. If timer equals the bullet separation time, the enemy can shoot and the timer resets back to `0`.

The `Increment` function should look like the following code snippet:

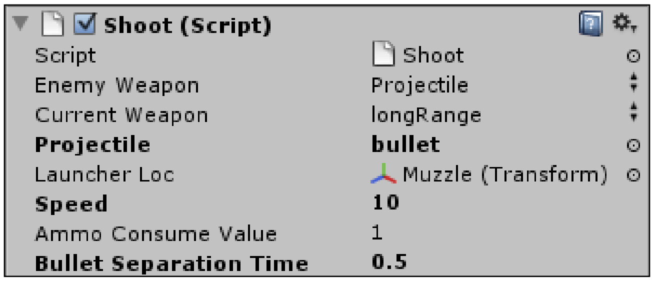
```
function incrementTime(){
    if(currentTimer < bulletSeparationTime){
        currentTimer += Time.deltaTime * 2;
    }
    else{
        canShoot = true;
        currentTimer = 0;
    }
}
```

Congrats, the `Shoot` script is complete and we don't have to modify it for this tutorial again. However, now that the script is written, we have to hook it up on enemy's **Inspector**.

Hooking up the Shoot script on Inspector

With the `Shoot` script done, we can now go ahead and drag it to each enemy's **Inspector**. What you should see is the `enum` variable at the top, allowing you to select the type of weapon the enemy will have; the `FlameThrower` boolean should follow after. The `projectile` variable needs you to drag the **bullet** prefab to be instantiated onto it. You then need to specify if it is `shortRange` or `longRange` and drag the **Muzzle** on to the launcher location variable. The speed of the bullet has to be defined along with how much ammo each bullet consumes and, lastly, the amount of time between each bullet.

Once everything is defined and hooked up, when you press `Play`, nothing will happen. That is because the behavior script runs the shooting script. With that being said, let's now start the `aiSimpleBehaviour` script.



The aiSimpleBehaviour script

The `aiSimpleBehaviour` script will set up the behaviors that the enemies can have, the behavior effects, the enemy's awareness range, and the function to locate and detect injured bots and ammo.

Setting up the script

The types of variables found in this script are enemy behavior, enemy range, time, attacking, reference, and locate variables.

The behavior variables are `passive` (the enemy does not attack the player), `defensive` (the enemy will attack only if attacked and only for a period of time), and `aggressive` (the enemy will attack the player if the player is within range). All these variables are `public` and `boolean`. A `private` `botType` variable with a `String` type is defined as well.

The enemy range variables are `enemyRange` (the enemy's awareness range), `defaultRange` (this is equal to the enemy's range), and `maxRange` (used to locate ammo). All these range variables are of the `float` type and only the `enemyRange` variable is `public`, the other two being `private`.

The single variable used for locating ammo is `lookForAmmo`, and it is `private` and `boolean`. It is used to determine if an enemy should look for ammo.

The time variables are `pursueTimeAfterAttack` (the time for which a defensive enemy will pursue the instigator), `currentTimer` (starts the pursue time at `0`), and `pursueTime` (used to start the pursue timer). `pursueTimeAfterAttack` and `currentTimer` are of the `float` type and `pursueTime` is `boolean`. The `pursueTime` and `currentTimer` variables are `private`.

Next, the attacking variables are `engagingPlayer` (used to check if the enemy should attack the player), `disengagedPlayer` (used to check if the enemy should disengage from attacking the player), and `playerEngaging` (used to see if the player is attacking the enemy). All these variables are of the `boolean` type and they are `private`.

Lastly, we have our reference variables and they are `pathingScr` (it has type of `aiSimplePath` and references the `aiSimplePath` script), `enemyStatScr` (it has type of `enemyStats` and references the `enemyStats` script), and `shootScr` (it has type of `Shoot` and references the `Shoot` script).

```
public var passive : boolean;
public var aggresive : boolean;
public var defensive : boolean;
private var botType : String;
public var enemyRange : float;
private var defaultRange : float;
private var maxRange : float = 100;
private var lookForAmmo : boolean;
public var pursueTimeAfterAttack : float = 3;
private var currentTimer : float = 0;
private var pursueTime : boolean;
private var engagingPlayer : boolean = true;
private var disengagedPlayer : boolean = false;
private var playerEngaging : boolean = false;
private var pathingScr : aiSimplePath;
private var enemyStatScr : enemyStats;
private var shootScr : Shoot;
```

The `Awake` function will set the script reference variables, set the default range to the specified enemy range, and call the `ReturnBotType` function as follows:

```
function Awake(){
    pathingScr = GetComponent("aiSimplePath");
    enemyStatScr = GetComponent("enemyStats");
    shootScr = GetComponent("Shoot");
    defaultRange = enemyRange;
    ReturnBotType();
}
```

Behavior functions

The following behavior functions will handle the bot's behavior control:

- `ReturnBotType`
- `SetIfPlayerIsAttacking`
- `CheckPlayerDistanceToEnemy`
- `PassiveBot`
- `FindAmmo`

The preceding functions may sound self-explanatory, if not, don't worry; we will look deeper into them right now.

The `ReturnBotType` function will set the `botType` variable based upon the behavior type of the enemy and return it to the `SetEnemyType` function parameter of the path script as follows:

```
function ReturnBotType(){
    if(passive){
        botType = "Passive";
    }
    else if(aggresive)    {
        botType = "Aggressive";
    }
    else if(defensive)    {
        botType = "Defensive";
    }
    pathingScr.SetEnemyType(botType);
}
```

The `SetIfPlayerIsAttacking` function will set the `playerEngaging` variable to its true/false parameter—`isAttacking` as follows:

```
function SetIfPlayerIsAttacking(isAttacking : boolean){
        playerEngaging = isAttacking;
}
```

The `CheckPlayerDistanceToEnemy` function will get the current distance between the player and the enemy using the `Vector3.Distance` function and give it to the function variable—`distanceToPlayer`. It will then compare this distance to the enemy's range to see if it is less, and therefore, within the enemy's awareness zone. It will then check the behavior type of enemy.

As our passive bot does not care for the player, we do not need to check if the player is in its range.

For aggressive enemies, we will skip right to checking if the enemy has enough ammo to shoot and that their health is greater than `0`. If the enemy is defensive, we will first check if he is engaging the player before moving on to checking ammo and health. If the health is greater than `0` and if we do have ammo to shoot, we will call the `GetBehaviourInfo` function with the `pathingScr` variable and assign it the `engagingPlayer` variable. After that, we will call the `Shoot` function with the `shootScr` variable and assign it `true` for the function parameter. The same is done for the defensive enemy.

If it happens that there is no ammo, or the enemy's health is greater than `0`, we will call the `FindAmmo` function with `true` as the function parameter. Call the `Shoot` function from the `Shoot` script and set the function parameter to `false`. Lastly, extend the enemy's range to `maxRange` to search for ammo. The same `else` statement is used for defensive enemies.

Next, we will check if the player is outside of the enemy's range, and if so, we tell the path script that we are disengaging from the player with the `GetBehaviourInfo` function and the `disengagedPlayer` variable as the function parameter. Then we will set the `Shoot` function to `false` to make the enemy stop shooting.

Lastly, we want to check if our ammo count is greater than `0`, and if it is, we will set our `enemyRange` variable back to the `defaultRange` variable and tell the enemy to stop searching for ammo. We will then set the `AmmoToLocate` function of the path script with the parameters of `null` and `false` to clear the data of the last ammo that was located.

The complete `CheckPlayerDistanceToEnemy` function should look like the following code snippet:

```
function CheckPlayerDistanceToEnemy(){
        var distanceToPlayer = Vector3.Distance
        (gameObject.FindWithTag("Player").
        transform.position,transform.position);
    if(distanceToPlayer < enemyRange)    {
        if(aggresive)    {
            if(enemyStatScr.GetAmmo() > 0
            && enemyStatScr.GetHealth() > 1){
                pathingScr.GetBehaviourInfo(engagingPlayer);
                shootScr.Shoot(true);
            }
            else    {
                FindAmmo(true);
                shootScr.Shoot(false);
                enemyRange = maxRange;
            }
        }
        else if(defensive)    {
            if(playerEngaging){
                if(enemyStatScr.GetAmmo() > 0
                && enemyStatScr.GetHealth() > 1){
                pathingScr.GetBehaviourInfo(engagingPlayer);
                    shootScr.Shoot(true);
                }
                else {
```

```
                    FindAmmo(true);
                    shootScr.Shoot(false);
                    enemyRange = maxRange;
                }
            }
        }
    }
    else if(distanceToPlayer > enemyRange)
    {
        pathingScr.GetBehaviourInfo(disengagedPlayer);
        shootScr.Shoot(false);
    }
    if(enemyStatScr.GetAmmo() > 0)
    {
        enemyRange = defaultRange;
        FindAmmo(false);
        pathingScr.AmmoToLocate(null, false);
    }
}
```

The `PassiveBot` function is for the bot having passive behavior. This bot acts as a medic and will seek out injured enemies and heal them.

To start off, we will set three array variables. The first variable grabs all the enemies in the level with the `FindGameObjectsWithTag` function and is called `getEnemies`, the second one is defined for distance and is called `distanceArray`, and the third one is defined for bots that need help and is called `botArray`.

We then check each enemy in the `getEnemies` array with a `for` loop, and for each one, we then check the health. If their health is below an acceptable limit, which in this case is `100`, they are added to `botArray`, their distance is acquired using the `Vector3.Distance` function, and they are grabbed using the `if` statement by the `botsDistance` variable. The distances are also added to `distanceArray`.

If the length of `distanceArray` is greater than `0`, it means that we have enemies to heal and we will continue.

If there are bots that need help, we will loop through `distanceArray` and check it against the first distance in the array. If the next distance is larger, we will take its distance and make it the closest distance. Once we have looped through all the distances, we will check it against the range of the medic bot. If it lies within the awareness range, we will call the `BotToHeal` function from the path script and assign it the first function parameter of the bot from `botArray` that the distance coincides with and `true` as the second parameter.

In order to get the proper bot from `botArray`, we will use a counter to keep track of the spot in the array that the closest distance is at. So, as we loop through `distanceArray` for checking distances, we will have the `counter` variable incremented each time. When we find a new closer distance we save the counter number. This counter number is then used to pull the proper bot out of the array.

Lastly, if no bots need the medic's assistance, we will set the `BotToHeal` function's first parameter to `null` and the second parameter to `false`. The complete `PassiveBot` function should look like the following code snippet:

```
function PassiveBot(){
    if(passive){
        var getEnemies : Array = new
       Array(gameObject.FindGameObjectsWithTag("Enemy"));
        var distanceArray : Array = new Array();
        var botArray : Array = new Array();
        for(var Bot : GameObject in getEnemies){
            if(Bot.GetComponent("enemyStats").GetHealth() < 100){
                var botsDistance : float =
             Vector3.Distance(Bot.transform.position,
             transform.position);
                distanceArray.Add(botsDistance);
                botArray.Add(Bot);
            }
        }
        if(distanceArray.length > 0){
            for(var dist : float in distanceArray)      {
                var counter : int = 0;
                var closDist : float = distanceArray[0];
                if(dist < closDist)    {
                    var arrayCounter = counter;
                    closDist = dist;
                }
                counter++;
            }
            if(closDist < enemyRange)    {
                pathingScr.BotToHeal(botArray[arrayCounter], true);
            }
        }
        else {
            pathingScr.BotToHeal(null, false);
        }
    }
}
```

The FindAmmo function will search for ammo in a given range. If the ammoNeeded variable is true, we need to go through a similar process with the medic and the injured bots.

We will define the three arrays. Copy and paste the ones from the PassiveBot function; and change the name from getEnemies to ammoArray for the search array, distanceArray can stay the same, and change botArray to ammoAvailable.

For the rest of the functions up to the final statement, if there are no bots that need a medic, we will copy the PassiveBot function and paste it here. Make sure that you change the names of the arrays with the appropriate ones throughout the code.

After we have the ammo sent to the AmmoToLocate function of the path script, we need to set the FindAmmo function to false. Then we have an else statement to check if any ammo is available and another one after the first if statement checking if there is any ammo in the level. In these else statements, we need to set the FindAmmo function to false, reset the AmmoToLocate function in the path script to null and false, and tell the GetBehaviourInfo to disengage from the player so that the enemy can resume his waypoint path.

```
function FindAmmo(ammoNeeded : boolean){
    lookForAmmo = ammoNeeded;
    if(lookForAmmo){
    var ammoArray : Array = new
    Array(gameObject.FindGameObjectsWithTag("Ammo"));
    var distanceArray : Array = new Array();
    var ammoAvailable : Array = new Array();
    if(ammoArray.length > 0){
        for(var ammo : GameObject in ammoArray){
            if(ammo.renderer.enabled){
                var ammoDistance : float =
                Vector3.Distance(ammo.transform.position,
                transform.position);
                    distanceArray.Add(ammoDistance);
                    ammoAvailable.Add(ammo);
                }
            }
            if(distanceArray.length > 0){
                for(var dist : float in distanceArray)    {
                    var counter : int = 0;
                    var closDist : float = distanceArray[0];
                    if(dist < closDist)    {
```

```
                        var arrayCounter = counter;
                        closDist = dist;
                    }
                    counter++;
                }
                if(closDist < enemyRange)    {
                  pathingScr.AmmoToLocate(ammoAvailable[arrayCounter],
                  true);
                  FindAmmo(false);
                }
            }
            else  {
                FindAmmo(false);
                pathingScr.AmmoToLocate(null, false);
                pathingScr.GetBehaviourInfo(disengagedPlayer);
            }
        }
        else     {
            FindAmmo(false);
            pathingScr.AmmoToLocate(null, false);
            pathingScr.GetBehaviourInfo(disengagedPlayer);
        }
    }
}
```

Additional functions

These functions will help to trigger the behavior functions and support their work.

For the defensive enemy, the `OnCollisionEnter` function checks if a bullet has collided with it. If it has, we will set the `GetIfPlayerIsAttacking` function to `true` and start the pursue time for the defensive enemy as follows:

```
function OnCollisionEnter(objCollided : Collision){
    if(defensive)    {
        if(objCollided.gameObject.tag == "projectile"){
            GetIfPlayerIsAttacking(true);
            pursueTime = true;
        }
    }
}
```

The `IncrementTime` function sets the pursue time for the defensive enemy. If `pursueTime` is true, it checks if `currentTimer` is equal to the `pursueTimeAfterAttack` value. If it is not, `currentTimer` is increased by the value given by multiplying `Time.deltaTime` with `2`. This is done to get seconds. If `currentTimer` is greater than or equal to `pursueTimeAfterAttack`, we will set `currentTimer` back to `0`, tell the enemy that the player is not attacking him anymore using `GetIflayerIsAttacking`, set `pursueTime` to `false`, and disengage from pursuing the player as follows:

```
function IncrementTime(){
    if(pursueTime){
        if(currentTimer < pursueTimeAfterAttack){
            currentTimer += Time.deltaTime * 2;
        }
        else    {
            currentTimer = 0;
            GetIfPlayerIsAttacking(false);
            pursueTime = false;
            pathingScr.GetBehaviourInfo(disengagedPlayer);
        }
    }
}
```

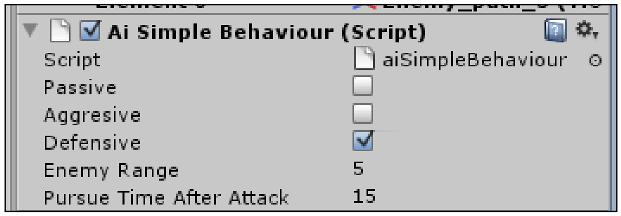
The `Update` function calls the `FindAmmo`, `PassiveBot`, `IncrementTime`, and `CheckPlayerDistanceToEnemy` functions. Make sure to assign the `FindAmmo` function the `lookForAmmo` variable.

```
function Update(){
    FindAmmo(lookForAmmo);
    PassiveBot();
    IncrementTime();
    CheckPlayerDistanceToEnemy();
}
```

Congratulations, the `aiSimpleBehaviour` script is written. Now we must hook it up on **Inspector** and add some functionality to the path script again, and we will be able to run this script with no errors.

Hooking up the aiSimpleBehaviour script on Inspector

After you drag the **aiSimpleBehaviour** script onto each enemy, we need to set the enemy behavior, the enemy range, and the pursue time for each enemy. However, we will be able to play without having errors only if we add the extra functions to the pathfinding script. Let's add that functionality now.



Returning to the aiSimplePath script

Now that we have our behavior script, we can add the additional functionality to the `aiSimplePath` script.

The additional functionality is a couple of functions that will deal with receiving path information and additional functionality for the enemy path script so as to find which bot is injured and needs to be healed, to locate the ammo, and to pursue the player.

Pursue functionality

We will have to add some variables before we add in the new functions and code. The variables that we will have to add are the pursue variables. These variables are used to make the enemy pursue different objectives.

Those variables are `pursuePlayer` (it is of the `boolean` type and used to determine if the enemy should pursue the player), `healBotLoc` (it is of the `GameObject` type and is used to locate the injured bot that needs to be healed by the passive bot), `healBot` (it is of the `boolean` type and used to determine whether the bot to be healed should be healed), `locateAmmo` (it is of the `boolean` type and used to determine whether the enemy should grab the ammo), and lastly, `ammoToFind` (it is of the `GameObject` type and used to determine the location of the ammo). The last variable to define is the script reference variable—`enemyStatScr`. It can be `private` and make sure that its type is `enemyStats`.

```
public var player : GameObject;
private var pursuePlayer : boolean;
private var healBotLoc : GameObject;
private var healBot : boolean;
private var locateAmmo : boolean;
private var ammoToFind : GameObject;
private var enemyStatScr : enemyStats;
```

The GetBehaviourInfo function will get the behavior information from the aiSimpleBehaviour script and set the pursuePlayer variable to the engagePlayer variable's value.

The SetEnemyType function will get the enemy's type as string and set the enemyType variable to the String value.

The BotToHeal function will grab the bot that the medic (passive bot) needs to heal, and it will also set the healBot variable.

The AmmoToLocate function will grab the ammo that the enemy should pursue and set the locateAmmo variable.

```
function GetBehaviourInfo(engagePlayer: boolean){pursuePlayer =
engagePlayer;}
function SetEnemyType(type: String){enemyType = type;}
function BotToHeal(Bot : GameObject, heal : boolean){healBotLoc = Bot;
healBot = heal;}
function AmmoToLocate(Ammo : GameObject, find : boolean){ammoToFind =
Ammo; locateAmmo = find;}
```

Revisiting the EnemyPath function

The functionality that is being added to this function is the functionality of the enemy to pursue and repair a damaged bot, the ability to locate and pursue ammo, and the ability to locate and pursue an enemy. Right after the declarations of the three Vector3 variables, we want to add in an if statement that will hold the if statements for pursuing the player, pursuing the ammo, and locating the injured bots. We need to perform the following steps:

1. If we are pursuing the player, we will set Target to the player's position, moveDirection will become the facing angle of the player, and we will make velocity equal to rigidbody.velocity. We will check the magnitude of moveDirection to see how close we are to our target, and if we are within our specified range, we will set velocity to Vector3 zero. If we are not within the specified range, we will normalize our direction, multiply it by our speed variable, and assign that to our velocity variable.
2. If we are a passive bot and if we are looking to heal a bot, we will continue pursuing the player till the Target variable becomes equal to the location of the bot to be healed and the range becomes less. If our target is within the range, we will call the RepairBot function from the enemyStats script.

3. If we are pursuing ammo, we will again follow the same procedure, except that the `Target` variable becomes equal to the location of the ammo to find and the range becomes less. We won't call any functions if within the range. We will just set `velocity` to `Vector3 zero`.

The code should look like the following code snippet:

```
if(pursuePlayer){
    Target = player.transform.position;
    moveDirection = Target - transform.position;
    if(moveDirection.magnitude < 3){
        velocity = Vector3.zero;
    }
    else{
        velocity = moveDirection.normalized * speed;
    }
}
        if(enemyType == "Passive" && healBot)    {
            Target = healBotLoc.transform.position;
            moveDirection = Target - transform.position;
            if(moveDirection.magnitude < 2) {
                velocity = Vector3.zero;
    healBotLoc.GetComponent("enemyStats").RepairBot();
            }
            else{
                velocity = moveDirection.normalized * speed;
            }
        }
        else if(locateAmmo)    {
            Target = ammoToFind.transform.position;
            moveDirection = Target - transform.position;
            if(moveDirection.magnitude < 0.5)    {
                velocity = Vector3.zero;
            }
            else{
                velocity = moveDirection.normalized * speed;
            }
        }
```

After placing those three statements in the function, we need to add an `else if` statement around the original waypoint path navigation. That `else if` statement is to check if the enemy is indeed pursuing the player.

With that done, the `aiSimplePath` script is completed. We have added the full functionality to it. Now all that is left is to press **Play**, and the behavior that has been applied and the path attributes should take effect.

The bulletCollision, ammoCollision, and AmmoInfo scripts

This section will allow the projectiles that we are firing to be able to identify what it is that they are hitting at and to apply damage to the object if it has the right tag. The `ammoCollision` script will handle the amount of ammo that will be given upon contact and the type of ammo. The `AmmoInfo` script will handle the respawn time and the enabling and disabling of the ammo.

In this section we will look into the following:

- Creating the `bulletCollision` script
- Hooking up the `bulletCollision` script on bullet's **Inspector**
- Creating the `ammoCollision` script
- Hooking up the `ammoCollision` script on bullet's **Inspector**
- Creating the `AmmoInfo` script
- Hooking up the `AmmoInfo` script on bullet's **Inspector**

Creating the bulletCollision script

The `bulletCollision` script handles how the bullet will react upon contact with an object. Depending upon the collided object's tag, a specified effect will happen. It will also set up a life timer on the bullet, and the bullet will explode when it has run out of time to live.

We will create the variables such as `explosion` for the explosion that will occur upon the bullet's contact with something, `bulletDamage` to determine the amount of damage that the projectile will do to the collided object when the tag matches, `bulletLifeTime` to determine how long until the bullet explodes, and `currentTimer` that will be the tracker for the timing of the bullet. Each variable is `public` except `currentTimer`, which is `private`. The `explosion` variable is of the `GameObject` type, `bulletDamage` is of the integer type, the `bulletLifeTime` variable is float, and so is `currentTimer`.

```
public var explosion : GameObject;
public var bulletDamage : int;
public var bulletLifeTime : float = 2;
private var currentTimer : float = 0;
```

The `OnCollisionEnter` function will handle the bullet's collision with objects. If it has collided with a wall and the player, it instantiates an explosion at the coordinates of its collision and destroys itself. If it is an enemy, it also causes damage to the enemy.

```
function OnCollisionEnter(collidedObj : Collision){
    if( collidedObj.gameObject.layer == "Wall" ||
    collidedObj.gameObject.tag == "Player"){
        Instantiate(explosion, transform.position, transform.
rotation);
        Destroy(gameObject);
    }
    else if(collidedObj.gameObject.tag == "Enemy"){
    collidedObj.gameObject.GetComponent("enemyStats").
    DecrementHealth(bulletDamage);
        Instantiate(explosion, transform.position, transform.
rotation);
        Destroy(gameObject);
    }
}
```

The `incrementTime` function handles the life of the bullet. If it has not collided with anything by the time the timer has run out, it creates an explosion and destroys itself. This is similar to how flax cannons work.

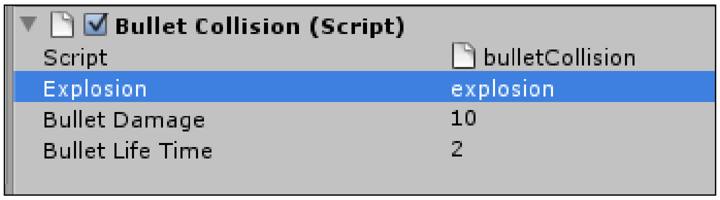
```
function incrementTime(){
    if(currentTimer < bulletLifeTime){
        currentTimer += Time.deltaTime * 2;
    }
    else    {
        Instantiate(explosion, transform.position, transform.
rotation);
        Destroy(gameObject);
    }
}
Update function calls the incrementTime function.
function Update(){
    incrementTime();
}
```

Hooking up the bulletCollision script on bullet's Inspector

Now that the script is written, we want to create a prefab that is called **bullet** and drag a cube or a sphere or any object that you would like to be our projectile. Drag the **bulletCollision** script onto the object as well and set the damage of the bullet and lifetime of the bullet. Once that is done, make sure that it is in the prefab and drag it to the **projectile** slot in every enemy's `Shoot` script.

Now when you press **Play** and the player comes into the range of an enemy, the player will be pursued if:

- The enemy is aggressive and the player is in range of the enemy
- The player attacks a defensive enemy and is in range of the enemy



Creating the ammoCollision script

The `ammoCollision` script, as mentioned, will destroy the ammo when hit, grab the ammo's information, and return it to the `enemyStats` script and the `ReceiveAmmo` function.

Declaring variables for this function is easy. All we have to define is a script reference variable for the `enemyStats` script. The variable will be `private` and its type will be the name of our `enemyStats` script:

```
private var enemyStatScr : enemyStats;
```

The `Awake` function will set the `enemyStatScr` variable to the `enemyStats` script as follows:

```
function Awake(){
    enemyStatScr = GetComponent("enemyStats");
}
```

The `OnCollisionEnter` function will determine the amount of ammo that the enemy receives upon contact with ammo. If the enemy collides with ammo, we will turn the ammo's renderer off by setting `objCollided.gameObject.renderer.enabled` to `false` and make it noncollidable by setting `objCollided.gameObject.collider.isTrigger` to `true`.

Three variables are declared—one variable to hold the script reference to the `AmmoInfo` script, a second one to get the type of ammo collided, and the third one to hold how much ammo should be given. The `ammoAmount` variable is an integer and the `ammoType` variable is `string`.

If `ammoType` is `projectile`, we will call the `GetProjectileAmmoAmount` function from the `AmmoInfo` script and if it is `special`, we will call the `GetSpecialAmmoAmount` function instead.

Lastly, we will set the `ReceiveAmmo` function of the `enemyStats` script to the `ammoAmount` variable.

```
function OnCollisionEnter(objCollided : Collision){
    if(objCollided.gameObject.tag == "Ammo"){
        objCollided.gameObject.renderer.enabled = false;
        objCollided.gameObject.collider.isTrigger = true;
        var ammoInfoScr : AmmoInfo =
        objCollided.gameObject.GetComponent("AmmoInfo");
        var ammoType : string = ammoInfoScr.GetAmmoType();
        var ammoAmount : int;
        if(ammoType == "projectile"){
            ammoAmount = ammoInfoScr.GetProjectileAmmoAmount();
        }
        else if(ammoType == "special")    {
            ammoAmount = ammoInfoScr.GetSpecialAmmoAmount();
        }
        enemyStatScr.ReceiveAmmo(ammoAmount);
    }
}
```

Hooking up the ammoCollision script on enemy's Inspector

Not much to do to hook this script up. Drag it onto **Inspector** of each enemy and you're done.

Creating the AmmoInfo script

The `AmmoInfo` script allows for ammo to have value when it is picked up by an enemy. The script will have the amount worth for the ammo and respawn the ammo after it has been disabled.

There are a couple of variables needed to be defined for this script.

We need an enum variable that handles kindOfAmmo. It has the enum name of weapon type and we will call it ammoType.

The next two are the ammo worth variables. The first one is projectileAmmoAmount and the second one is specialAmmoAmount. Both of them are public and are of the integer type.

The next set of variables contains the time variables. They are the respawnTime, currentTimer, and startTime variables. respawnTime is public and the other two are private. respawnTime and currentTimer are of the float type, and startTime is boolean.

The last variable is the kindOfAmmo enum. This one, at this moment, holds reference to projectile type ammo and special type ammo.

```
public var ammoType : kindOfAmmo;
public var projectileAmmoAmount : int = 50;
public var specialAmmoAmount : int = 25;
public var respawnTime : float = 5;
public var currentTimer : float = 0;
private var startTime : boolean = false;
enum kindOfAmmo{
    projectile,
    special
}
```

The GetAmmoType function will return ammoType based upon the selection from the enumeration (projectile or special).

The GetProjectileAmmoAmount function returns projectileAmmoAmount.

The GetSpecialAmmoAmount function returns specialAmmoAmount.

The incrementTime function will start the respawn time of the ammo after the ammo has been disabled. The timer will count up to the respawn time by adding Time.deltaTime to itself. When the timer reaches the respawn time, we will set the startTime variable to false, turn the ammo's renderer back on by setting gameObject.renderer.enabled to true, make the ammo collidable by setting gameObject.collider.isTrigger to false, and reset the timer to 0.

The Update function will check if the renderer of the gameObject is enabled, and if it is not, it will set startTime (respawn time) to true and call the incrementTime function.

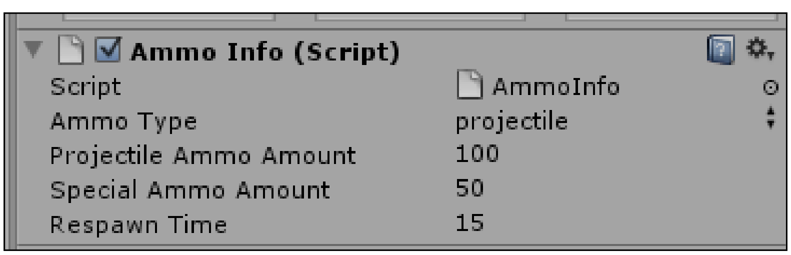
```
function GetAmmoType() : String {
    if(ammoType.projectile)
    return "projectile";
    if(ammoType.special)
```

```
    return "special";
}
function GetProjectileAmmoAmount(){return projectileAmmoAmount;}
function GetSpecialAmmoAmount(){return specialAmmoAmount;}
function incrementTime(){
    if(startTime)    {
        if(currentTimer < respawnTime){
            currentTimer += Time.deltaTime * 2;
        }
        else     {
            startTime = false;
            gameObject.renderer.enabled = true;
            gameObject.collider.isTrigger = false;
            currentTimer = 0;
        }
    }
}
function Update(){
    if(gameObject.renderer.enabled == false){
        startTime = true;
        incrementTime();
    }
}
```

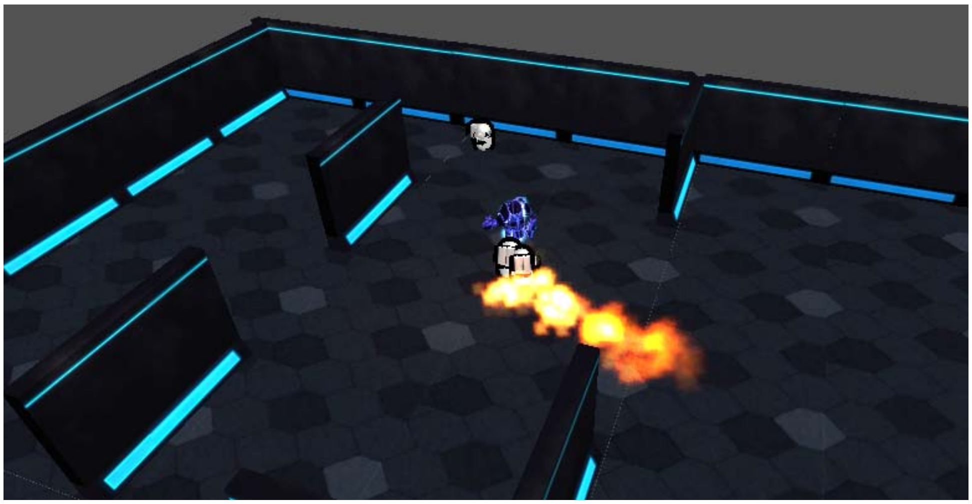
Hooking up the AmmoInfo script on ammo's Inspector

Once the `AmmoInfo` script is written, we will create a new prefab called **ammo**. Create a single cylinder and apply a basic material to it so that it can be differentiated from the background color of the level.

After that, apply the script to the `gameObject` and set the values of the script. For testing purposes, the `enum` defaults to `projectile` or whichever one you have at the first place. Set **Projectile Ammo Amount** to **100**, **Special Ammo Amount** to **50**, **Respawn Time** to **15** (this represents seconds). Tag the **gameObject Ammo** and drag the **gameObject** onto the prefab. Place your ammo around the level and you are done.



Now place the player in a position where an enemy will interact with him. So, when you go and press **Play**, the enemies will follow their paths, but if they come across the player, and if their behavior type is aggressive, they can shoot the player but only for a specific time. With each bullet fired, their health goes down and now they have to search out ammo. If the ammo has been out of range of the player when the enemies pick it up, they disengage from the player and resume their waypoint path, at least, until their paths cross again.



Summary

This chapter has shown a fair beginning to AI. A person should be able to identify aspects of what needs to go into designing systems for AI to work. In this chapter, we explored concepts such as identifying the path that the enemy should navigate along, how the enemy should react at the end of a waypoint array, how behaviors can help expand and enrich the player's experience, and bring some challenge to enemies. We saw how tweaking values can give us different battle situations, and therefore, if we have a simple battle setup for bots, it is very doable and easy to accomplish for the player.

As mentioned throughout this book, these concepts and principles that are being shown are merely the basics and they are very simple. In order to truly get results, whether it be AI, tethering, cameras, or controllers, there is a more sophisticated way that allows for optimization and more freedom with the code. This is merely a gateway. I am hoping you want to learn more and explore the boundaries of Unity and then surpass them. In the reading material of the *Appendix*, *Object-oriented Programming in Unity*, there are several excellent websites, wikis, forums, and blogs to visit, which will help with questions you may have.

Object-oriented Programming in Unity

The book has almost come to an end, but there are a lot of things to learn ahead. In this appendix, we will cover object-oriented programming inside Unity. Those of you who used to work with C/C++/Java/AS3 should be familiar with all the concepts of OOP and will find their application in Unity. For others, it would be a next step of how to improve oneself in programming.

Object-oriented programming – basics

Object-oriented programming, or OOP, is a programming paradigm that uses data structures called **objects** to store parameters and methods. To make it simple, every object has a number of parameters and functions that it can execute, and whole programming is based upon object manipulation.

Encapsulation

One of the four fundamentals of OOP is **encapsulation**, the ability of objects to hide properties and methods. It might sound as if its only purpose is to protect them, and in general it is. There is no better way to hide the source code that you've been working on for so long if you decide to give your program to somebody to try. But, imagine that you are not working alone and wish to save your peers from reading and trying to understand thousands of lines of your code. The best way to do that is to hide everything that they wouldn't need to use or even look at, and provide them only with everything necessary for their work. That's when encapsulation comes in handy.

There are three types of access levels that are used:

- `private`: visible only to the current class
- `public`: visible to everybody
- `protected`: visible to the current class and inherited classes

Unity has a very interesting way to incorporate encapsulation by letting us choose which properties we want to be modified inside the editor and be visible in the **Inspector** view. Basically, if we want any variables to be modified or seen in the editor, we need to give them `public` access level; all others will remain saved and hidden.

Classes

Before we can create an object, we need to create its prototype or a template, which is called **class**. A class is a construct that can create instances of itself. A class defines constituent members that enable its instances to have state and behavior. In general, class is referred to as a noun (cow, table, inventory). Inside the class, we can specify properties (data) and methods (functions).

Constructors

The interesting feature of classes is a function that executes upon creation. Whenever we create a new instance of an object, we automatically call the so-called constructor function, which can even take arguments and do basic setup for the object. The constructor function must be named after class to be recognized as a constructor.

Code

Perform the following steps:

1. Create the constructor function for `ParentClass`:
   ```
   function ParentClass(){Debug.Log("Constructor function is
   called");}
   ```
2. Instantiate `ParentClass` from the `Awake` function:
   ```
   var inheritedClassSample= new ParentClass();
   ```

If we run the code now, as soon as the instance of `ParentClass` is instantiated, we will see a debug message that `"Constructor function is called"`.

Inheritance

One of the most wonderful features of classes is **inheritance**. Whenever we create a new class, we can extend from an already existing class and inherit all its data and methods. Of course, we can regulate access level by making data private, which will make it accessible only by its original class.

Inheritance is widely used in OOP. Imagine that you need to create a number of weapons; you can create each class for the weapon separately, however, if we look closer, each weapon has some things in common. These common data and methods can be stored in a template class, and we can create each individual weapon class as a child of the template class. The template classes that are never instantiated, but use inheritance to create child classes, are called **abstract**.

If the class that we created is not meant to be extended from, we can declare it as final, which will prevent it from being inherited from.

Preparations

Let's take a look at some examples of class manipulations. In this example, we will create a script that contains a couple of classes, and they will inherit from each other. Each class will have data and methods with different access levels to demonstrate how they work:

1. Create a new script, and call it `OOP`.
2. Create a new scene, and place a random object in there.
3. Attach the `OOP` script to this object.

Code

Perform the following steps:

1. Declare a new `public` class and call it `ParentClass`:

   ```
   public class ParentClass{}
   ```

2. Inside `ParentClass`, declare the `private`, `protected`, and `public` variables:

   ```
   protected var protectedVar : int = 1;
   private var privateVar : int = 1;
   public var publicVar : int = 1;
   ```

3. Create three functions—`private`, `public` and `protected`:

```
private function voidFunction() : int
{Debug.Log("Private function called");}
public function getPrivateFunction():void
{voidFunction();}
protected function protectedFunction() : int
{return privateVar;}
```

4. Create a new class, and call it `inheritedClass`. It will inherit from `ParentClass` and will be the final class:

```
public final class InheritedClass extends ParentClass{}
```

5. Declare the `Awake` function outside the class, create a new instance of `inheritedClass`, and try calling the `public` and `protected` functions from the parent class:

```
function Awake(){
 var ClassSample = new InheritedClass();
ClassSample.getPrivateFunction();
Debug.Log( ClassSample.protectedFunction ());
}
```

The result is predictable; the child class recognized the `public` and `protected` functions of the parent and successfully called them.

Polymorphism

Inheritance is an amazing feature that allows us to save on copying and pasting the same code from one class to another and making our code more organized, but what if the methods that we wish to use in child classes need to be tweaked a bit? Thankfully, we don't have to create new functions; we can simply rewrite existing ones. This concept in OOP is called **polymorphism**. Polymorphism has two features that we need to keep in mind:

- **Method overloading**: This allows us to declare functions with the same naming signature within the class, but a different argument list
- **Method overriding**: This allows us to declare functions with the same naming signature as the parent class, but change what is actually being executed in them

The argument list, really, is the only way to control an overloaded function, but what if we need to call an overridden function from the parent class? This can be done by calling `super.functionname()`. `super` is a reference to a parent class and can be called from anywhere in the child class.

Code

Perform the following steps:

1. Inside `InheritedClass`, override `protectedFunction` and create a constructor function:

   ```
   function InheritedClass(){}
   public function protectedFunction(): int {return 5.1;}
   ```

2. In the constructor function, call both the original and overridden `protectedFunction` functions:

   ```
   Debug.Log("Calling super: " + super.protectedFunction());
   Debug.Log("Calling protected: " + protectedFunction());
   ```

When we start the program, the constructor function will call two functions with the same naming signature, but from different classes.

Nested classes

Classes can be the storage for class properties, as well as other classes. Classes that are stored within other classes are called **nested classes**. Nested classes can get access to the `public` and `protected` functions, methods, and properties of upper class.

Summary

Most of the modern programming and scripting languages use an object-oriented programming paradigm, which makes the knowledge gained in this appendix universal. The more you work with classes, the more ways of using them you will find, which will make your work easier to communicate and more efficient.

In the end, it's all about the experience, and about how much effort and time you spend on solving problems, which will make you a better programmer. Go ahead now! Start working on personal and group projects, read other people's code, and find efficient solutions to the problems! We can't wait to see what you come up with.

Index

Symbols

A

B

C

D

H

S

T

U

V

W

Y

Thank you for buying
Unity 3.x Scripting

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

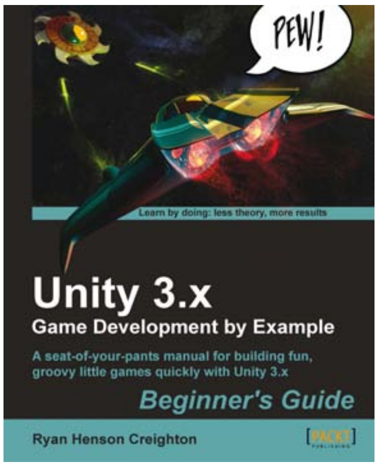
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Unity 3.x Game Development by Example Beginner's Guide

ISBN: 978-1-84969-184-0 Paperback: 408 pages

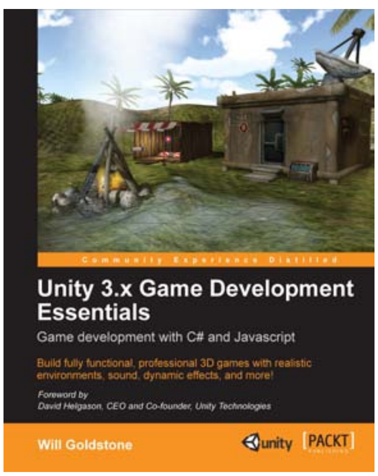
A seat-of-your-pants manual for building fun, groovy little games quickly with Unity 3.x

1. Build fun games using the free Unity game engine even if you've never coded before

2. Learn how to "skin" projects to make totally different games from the same file – more games, less effort!

3. Deploy your games to the Internet so that your friends and family can play them

---



Unity 3.x Game Development Essentials

ISBN: 978-1-84969-144-4 Paperback: 488 pages

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

1. Kick start your game development, and build ready-to-play 3D games with ease

2. Understand key concepts in game design including scripting, physics, instantiation, particle effects, and more

3. Test & optimize your game to perfection with essential tips-and-tricks

4. Written in clear, plain English, this book takes you from a simple prototype through to a complete 3D game with concepts you'll reuse throughout your new career as a game developer

Please check **www.PacktPub.com** for information on our titles