

Building an RPG with Unity 2018

Second Edition

Leverage the power of Unity 2018 to build elements of an RPG



Packt>

www.packt.com

By Vahé Karamian

Building an RPG with Unity 2018

Second Edition

Leverage the power of Unity 2018 to build elements of an RPG.

Vahé Karamian



BIRMINGHAM - MUMBAI

Building an RPG with Unity 2018

Second Edition

Copyright © 2018 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Noyonika Das
Content Development Editor: Francis Carneiro
Technical Editor: Ralph Rosario
Copy Editor: Safis Editing
Project Coordinator: Devanshi Doshi
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Jason Monteiro
Production Coordinator: Aparna Bhagat

First published: March 2016
Second edition: July 2018

Production reference: 1270718

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

ISBN 978-1-78862-346-9

www.packtpub.com

*To my wife Armineh for being my loving partner throughout our joint life-journey.
To my son and daughter, Maximilian and Makayla.*

– Vahé Karamian



`mapt.io`

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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.

Contributors

About the author

Vahé Karamian is a software consultant and author based in Los Angeles, CA. He is the founder of Noorcon Inc., a software consulting firm in Burbank, CA, providing services in the field of augmented reality and virtual reality used in the entertainment, architecture, medical, and manufacturing industries. He has lectured in state and private universities on topics including Introduction to Computer Science, Data Structures and Algorithms, Operating Systems, and Game Design and Development.

I would like to acknowledge my wife, Armineh, and our two beautiful children, Maximilian and Makayla, for their support through the long nights in the preparation of this book. Thank you and love you.

About the reviewers

Adam Larson started programming professionally in 2005. He has shipped 14 console games, 2 PC titles, and 8 mobile games. In 2012, he founded a company that focused on using Unity for business applications. He spent the following 5 years building a million dollar agency that built high-end software. Recently, he moved to another start-up that focuses on revolutionizing the banking industry. When he isn't doing something with technology, you will find him spending time with his wife and three kids.

Francesco Sapia has a master's of science in engineering in AI and Robotics, scoring summa cum laude. Now, he is a PhD researcher. He is a Unity and Unreal expert. Among his many projects, he developed Gea2 (Sapienza), Sticker Book (Dataware Games), and consulted for Prosperity – Italy 1434, Mi Rasna (Entertainment Game Apps, Inc), and Projekt I.D (RMIT). He is an active writer on game development, authoring *Getting Started with Unity 5.x 2D Game Development*, *Unity UI Cookbook*, and many others. Finally, Francesco loves math, philosophy, logic, and puzzle solving, but mostly loves creating video games.

I'm deeply thankful to my family for their enthusiasm and support throughout my life; they have always encouraged me to do better with the Latin expressions "Ad maiora" and "Per aspera ad astra". Finally, a huge thanks to all the special people around me whom I love, in particular to my sweetheart; I'm grateful for all your help in everything.

Michael Miles is a manufacturing engineer working in the aerospace industry by day and a Unity game designer, C#, JavaScript, and Python developer at all other times. With these skills and upon discovering Unity3D and Blender, Michael began creating interactive training and production-ready instruction tools for the assembly of the aircraft his company builds.

When not in front of a computer screen, he likes to play games with family and friends or immerse himself in books from fantasy to theoretical physics.

Thanks to my parents for always being there, supporting every decision I made, however unwise you thought it was at the time.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: What is an RPG?	5
A brief history of the genre	5
Characteristics of an RPG	7
Story and setting	7
A glimpse of a cRPG story	8
Exploration and quests	8
A glimpse at our exploration and quests	9
Inventory system	10
Character attributes and actions	13
Experience and leveling	15
Combat system	17
User interaction and graphics	18
Existing or upcoming RPG games	21
MU Legend	21
Titan Siege	23
Citadel: Forged with Fire	24
Cyberpunk 2077	25
Patterns in RPG	26
Terminology	27
Contest tree	29
Last Man Standing	30
Negotiated Contest	31
Summary	32
Chapter 2: Planning the Game	33
Building our RPG	33
The story of the Zazar dynasty	34
Backstory	34
Exploration and quests	34
Awakening	35
The village	35
Broken forest – the horizon	36
The kingdom	38
Asset inventory	39
Environment assets	39
Medieval Environment Pack	40
Terrain Toolkit 2017	41
Nature Starter Kit 2	41
Character assets	42

Barbarians	43
Orcs	44
Villagers	44
Free Assets	45
Level design	46
Setting the stage	48
Terrain toolkit in a nutshell	51
The Awakening	53
Using the terrain model	55
Using a custom toolkit	57
Skybox	65
Testing the level	66
Creating the main menu	68
Creating the GameMaster script	69
Summary	73
Chapter 3: RPG Character Design	74
Character definitions	74
Character assets	75
Base character class attributes	75
Character states	77
Character model	80
Default character models	80
Barbarian	80
Villager	81
Orc	83
Let's get started	83
Rigging your model	85
Character motion	87
Animator Controller	87
Animation states	88
Character controller	93
Modification to animations	99
Inverse Kinematics	101
Setting the animation curve	107
Summary	108
Chapter 4: The Game Mechanics	109
Customizing the player character	110
Customizable parts	113
User interface	115
The code for character customization	118
Preserving our character state	122
Recap	123
Non-player characters	123
Non-player character basics	125
Setting up the non-player character	125

NPC Animator Controller	130
NPC Attack	133
NPC AI	136
PC and NPC interaction	144
Summary	152
Chapter 5: GameMaster and Game Mechanics	153
GameMaster	154
Managing game settings and audio	156
Managing scenes	160
Improving GameMaster	163
Level controller	163
Audio controller	165
Player data management	169
PC class enhancements	169
Character customization class update	172
Changes to the game level controller	180
Testing	180
Summary	182
Chapter 6: Inventory System	184
Designing an inventory system	185
Weighted inventory	185
Determining item types	186
Creating an inventory item	193
Creating the prefab	194
Adding an inventory item agent	196
Inventory items defined as prefabs	200
Designing an inventory interface	201
Creating the inventory UI framework	203
Designing a dynamic item viewer	206
Adding a scroll view	207
Adding elements to PanelItem and Scroll View	211
Adding txtItemElement dynamically	213
Building the final inventory item UI	215
Integrating the UI with the actual inventory system	217
Hooking the category buttons and displaying the data	217
Testing the inventory system	222
Inventory items and the player character	228
Applying inventory items	229
How it looks	243
Summary	245
Chapter 7: User Interface and System Feedback	246
Designing a heads-up display	247
HUD basics	247

Our design	248
HUD framework	249
Completing our HUD design	250
Character info panel	250
Active inventory items panel	255
Special items panel	259
Integrating the code	261
Enemy stats in the HUD	275
NPC stats user interface	276
Creating the NPC canvas	276
NPC taking a hit	280
Enhancing the code	285
Summary	290
Chapter 8: Multiplayer Setup	292
Challenges of a multiplayer game	292
Initial multiplayer game	293
Fundamental networking components	294
My tank networking project	295
Adding a player character	297
Variable synchronization	300
Network callbacks	301
Sending commands	301
Client RPC calls	302
Creating the cannonball for the tank	304
Creating the tank prefab and configuring the network lobby manager	306
Adding the enemy tank	310
Building and testing	315
Network-enabling RPG characters	318
Creating a scene for our RPG	318
Networked player character	321
Networked non-player character	326
Synchronizing player customization and items	332
Spawning NPCs and other items	333
Testing our network-enabled PC and NPC	334
What's next?	340
Summary	341
Other Books You May Enjoy	342
Index	345

Preface

Everyone wants to make a game—today, this is possible more than ever due to the democratization of the game industry and the tools that are used to design and develop games. This book is written with several purposes in mind. During the process of writing this book, Unity went through several major updates and releases. At the time of writing, Unity stands at version 2018.1.1f1, and the new release has added many great features to the game engine. Simply put, it is impossible to cover everything in one single book!

This book is intended as a reference guide for individuals who want to learn about Unity and apply their skills for the creation of a **Role-Playing Game (RPG)**.

Who this book is for

This book is written for individuals who want to learn and apply their Unity skills for the creation of an RPG. It is assumed that the reader has a basic understanding of the concepts of programming and is comfortable with the basics of Unity's IDE. This book gives a strong and solid foundation of the core concepts and topics that can be applied to build your own game experience.

What this book covers

Chapter 1, *What is an RPG?*, provides a good background of what an RPG is. It covers some historical aspects and gives examples of existing RPGs. It discusses the main aspects of an RPG, covers some terminology, and prepares the reader for the rest of the book.

Chapter 2, *Planning the Game*, is where we take a look at character definitions, character class attributes, character states, and how to set up and rig character models, which includes an exploration of motion, controllers, and inverse kinematics.

Chapter 3, *RPG Character Design*, continues expanding on player character customization and looks at preserving character states, the setup of **Non-Player Characters (NPCs)**, and introducing NPC **Artificial Intelligence (AI)** and interaction.

Chapter 4, *The Game Mechanics*, is where we start planning the game. We discuss the different types of assets and resources we will need during the creation of our game, introduce a third-person character controller, and create our initial level and scripts. We also take a look at the Terrain Toolkit for terrain generation.

Chapter 5, *GameMaster and Game Mechanics*, looks at enhancing the Game Master script, introduces Level Controller and Audio Controller scripts, discusses the storage of character data and the character customization state, and explores options for the initial user interface for the main menu.

Chapter 6, *Inventory System*, covers the creation of a generic inventory system; the creation of the necessary scripts, assets, and prefabs that represent the inventory items; designing the inventory user interface; and how to represent the inventory system and its items.

Chapter 7, *User Interface and System Feedback*, discusses the design and implementation of a heads-up display, player character information panel, and the active inventory items panel. The special inventory items panel is designed and implemented, as are the non-player character health bar and UI.

Chapter 8, *Multiplayer Setup*, discusses multiplayer programming using Unity's Unet architecture. This chapter illustrates the concepts using two sample projects. The initial project is a tank game that illustrates the concepts of server client and data synchronization. The second project applies what we have learned to create a scene supporting our character models.

To get the most out of this book

You will need to have a good understanding of the C# language. You will also need to have a good understanding of the basics of the Unity IDE.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Building-an-RPG-with-Unity-2018-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/BuildinganRPGwithUnity2018SecondEdition_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace com.noorcon.rpg2e
{
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1 What is an RPG?

Before we get started, it would be best to get a brief history of the genre and understand some of the key elements you will need to consider throughout the design of your RPG.

Here is an overview of what we will be covering in this chapter:

- A brief history of the genre
- Characteristics of an RPG
- Story and setting
- Exploration and quests
- Inventory systems
- Character development
- Experience and leveling
- Combat systems
- User interaction and graphics
- Existing or upcoming RPGs
- Patterns in RPGs

You might have just started one of your most challenging tasks thus far.

A brief history of the genre

What is an RPG? In short, it is a game in which players assume the roles of characters in a fictional setting, in which the participants assume the roles of characters and collaboratively create stories. Each player determines the actions of their characters based on their characterization, and the actions taken to succeed or fail according to the rules of the game.

There are three varieties of RPGs, as follows:

- Tabletop
- Live action
- **Computer RPG (cRPG)**

Tabletop and **pen-and-paper (PnP)** RPGs are conducted through discussions in a small social gathering. There is usually a **Game Master (GM)** who describes the game world and its inhabitants. The other players describe the intended actions of their characters, and the GM describes the outcomes. This is the format in which RPGs were first popularized, namely through **Dungeons & Dragons (D&D)**.

Live Action Role Playing (LARP) is played more like improvisational theatre. Participants act out their character's actions instead of describing them, and the real environment is used to represent the imaginary setting of the game world. SomeLARPs use rock-paper-scissors or a comparison of attributes to resolve symbolically, while otherLARPs use physical combat with simulated arms. A movie can be considered a simple LARP; the difference being that in a movie, all actions are scripted, and the players do not have much decision-making to do, whereas in a LARP, the characters can change the outcome of their actions based on their decisions.

cRPGs are tabletop RPGs that have been translated into an electronic format. Early cRPGs influenced all types of electronic gaming, as well as spanning the role-playing video game genre. In short, a cRPG is a video game genre where the player controls the actions of the main character while immersed in a well-defined world.

This book will concentrate on the design and development of a cRPG.



NOTE: Going forward, when we mention RPG in the book, we are referring to cRPG.

Computer role-playing games take their root from tabletop versions of the genre. Many of the same terminologies, settings, and game mechanics have been carried over from the original tabletop games. Some of these similarities include story-telling and narrative elements, where, throughout the story, the player character will continuously develop their skills and abilities to meet the objective of the game.

Characteristics of an RPG

Role-playing video games typically rely on a highly developed story and setting, which is divided into a number of quests or levels. Players typically control one or more characters by issuing commands, which are then performed by the player character based on their defined abilities and attributes. Throughout the game, these attributes increase or decrease, and set the personality of the character.

An RPG usually also has a complex and dynamic interaction mechanism, defined and developed between the player character and the world within which they are immersed. This includes interaction with the world environment, as well as other non-character players defined within the world. Due to these factors, there is usually more time allocated to designing and developing the codebase that deals with the behavior and **artificial-intelligence (AI)** handling such events throughout the game.

The key elements of an RPG are as follows:

- Story and setting
- Exploration and quests
- Items and inventory
- Character actions and abilities
- Experience and levels
- Combat
- Interface and graphics

Story and setting

The premise of most role-playing games tasks the player with saving the world, or whichever level of society is threatened. There are often twists and turns as the story progresses, such as the surprise appearance of estranged relatives, or enemies who become friends or vice versa. The game world tends to be set in a historical, fantasy, or science fiction universe, which allows players to do things they cannot do in real life, and helps players suspend their disbelief about the rapid character growth.

As stated previously, RPGs are heavily invested in story-telling. This is one of the main key entertainment factors of the genre. Due to this fact, when you are developing your RPG, you will need to pay close attention to how you develop your story and the characters that are within your story. This, in turn, translates into the kind of environments and settings you will have for your game, and characters within the game.

Traditionally, RPGs progress the plot based on decisions that the player character makes during gameplay. This puts a great deal of pressure on the game designer, who needs to be able to integrate such forks in the gameplay with the main storyline of the game. This also raises the issue of how to program the game to take into consideration of all the different paths within the story.

To make the game more interesting and attractive, the game designer can introduce special triggers within the story to make it more interesting or challenging. This is usually done by introducing new characters and/or areas to discover within an existing level. The following is a simplified description of the storyline and the setting we will be building for our cRPG.

A glimpse of a cRPG story

Once upon a time, there was a great kingdom ruled by the great King Zazar. The ruler of the kingdom was a generous lord to his subjects. The kingdom under the rule of Zazar was peaceful and prosperous; however, over time, internal family rivalry and struggle caused cracks in the strong bond that kept the kingdom intact.

Due to mysterious events, the great king decided to move his family away from the kingdom and trust his son to one of the wise elders that he trusted. The kingdom was never the same. Until now! Expanding on our story, in the second edition, we will implement more gameplay, as well as the introduction of quests.

Exploration and quests

The whole idea behind an RPG is the ability of the player to have the freedom to explore the world in which they have been immersed. The more well defined the world is, the more interesting it will be for the player to explore and in return retain their curiosity and engagement throughout the gameplay.

This is achieved by the narrative of the story developed for the RPG. Players will be specifically given the opportunity to walk around the world and explore their surroundings in order to meet their objectives.

In an open-world RPG, the player is free to roam in the world event after they have met their objective, set by the storyline. In such cases, the player can still explore any area which is no longer needed for the continuation of the quest, but they can spend time exploring the area and maybe meet some other non-player characters that they hadn't previously met while completing their mission. But generally speaking, this is not done by the player; once they meet their objective, they are eager to move on to the next quest. Hence, the question is, how much time and effort do the game designers and developers apply to a region after the main objective has been met? The answer would be not much.

Historically, the player follows a linear sequence of quests in order to realize their goals and objectives within the game. To make the game more engaging, the developer can introduce mini-quests within the main plot of the game at that particular location, to give the player the ability to explore and gain more skills and/or abilities. Since these are not part of the main storyline, they can be triggered anytime a player enters a specific area.

For instance, assume the player has completed the main objective of the level and is ready to move on to the next objective. Now, imagine that we have created an open-world environment where the user can revisit the world anytime they choose. If the player decides to go back and explore a certain area of the world they just completed, and they happen to trigger the event to launch this mini-quest, wouldn't that be a great surprise for the player? Keep in mind that these mini-quests should not affect the main storyline, but they can be used to enhance the player experience. These types of decisions are important when you are developing your game. If they choose not to take on the challenge, you should not penalize them, except if you want to be really mean.

Quests may involve defeating one or many enemies, rescuing a non-player character, item-fetch quests, or location puzzles, such as mysteriously locked doors.

A glimpse at our exploration and quests

Our game will have a total of four quests. Each quest will have unique objectives that the player will need to complete. The design and development of each quest will be discussed throughout the book as we progress.

Here is a list of the levels we will be designing:

- Awakening
- The village
- Broken forest - the horizon
- The kingdom

The game will start by immersing the player in the environment, where our hero will be given the basic training he will need to complete his mission.

Inventory system

One of the main functions and features of an RPG is the inventory system. Throughout the game, the user will come across a vast amount of collectible items that can be used for different purposes within the game to help them progress through the journey. Therefore, RPGs need to provide a mechanism to help the player store, retrieve, and organize the content relevant to their journey. Take a look at the following screenshot:



Inventory

As the player progresses throughout their journey in an RPG, they interact with the world they are immersed in. The storyline of the game usually forces the player to interact with the surrounding world and other non-player characters. These interactions are usually in the form of some sort of an exchange. Whether this exchange is done through narration, to provide the player with a better sense of the quest, or real exchange, in terms of items, is up to the game designers and developers.

The game needs a way to keep track of all the interactions between the player character and everything and everyone else. One system that is used to keep track of these interactions is the inventory system.

During the gameplay, players usually start off as a very simple character and part of the gameplay is to elevate their character by exploring the world and collecting items that will help them increase their skills and abilities.

For instance, a player might start their journey with very basic clothes. Throughout the quest, they will either interact with a non-character player, such as a merchant who will provide them with a better set of clothes, and/or some sort of a weapon to get them started. These items will be stored and managed by the inventory system, an example of which is shown in the following screenshot:



Items will be stored and managed by the inventory system

The simplicity or complexity of the inventory system will be defined by the complexity of the game and the complexity of the characters within the game. In our game, we will be designing and developing a generic inventory system that can be applied to many different types of items.

Here are some of the items that are usually collected in a game:

- Weapons
- Armor
- Clothing
- Special objects

Some of the items are collected or discovered by world exploration, and some of the items are specifically traded throughout the game. If you are setting up a trade system in a game, then you will need to provide the mechanics for the trade. A trade usually takes place while interacting with a non-player character, usually a merchant, and it will use a special window to enable the interaction of the trade to take place.

There is usually a cost associated with any trade. In general, there is a cost associated with everything the player does within the game, and the cost usually either increases the player character's ability and/or experience or decreases it. This can get pretty complex if you dig deeply into it.

The main point to keep in mind is that everything that the player will need to collect and/or manage will be done through the inventory system. Hence, this is one of the most critical features that you will need to put effort into as a game designer and developer.

One other element that can be used to enhance the gameplay for the player, and push them to strategize their quest, is limiting the amount of items they can carry in their inventory.

For instance, in real life, a warrior will have limited ability to carry different types of weapons. Let's assume that in the real world, a warrior can carry a maximum of five different types of weapon at any given time. Now, in the game world, there might be 20 different types of weapon. Do you allow the player to carry all 20 different types when they discover them? Or are you going to limit the quantity to five?

These are the small things that make the gameplay more interesting when planned out properly. There is more to an inventory system, and we will take a look at it in more detail in later chapters.

Character attributes and actions

As with any other part of RPG development, character attributes and actions are highly defined by the storyline of the game. These actions, are performed indirectly within the game when the player commands the character to perform a specific task.

For instance, in a given RPG there will be at least a couple of character classes. The following are some sample class types:

- Barbarians
- Orcs
- Magicians / Wizards
- Zombies
- Humans

Each character class might even have subclasses of its own, with its own uniquely defined attributes. Again, this will be tightly coupled to your storyline for your RPG.

For instance, we will have the player-character, who is technically the hero of our story and of the game. The hero is usually of a certain character class; let's assume the hero is part of the Human class.

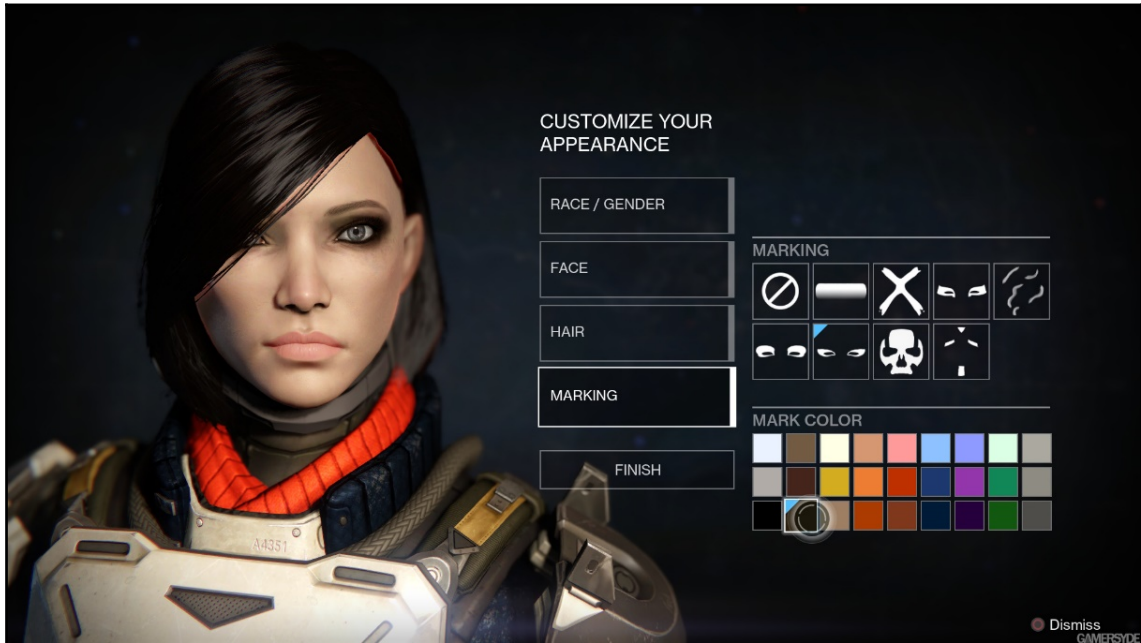
The Human class or race, then, will have some specific characteristics that will be inherited by the player character, or any other non-player character of the same type or class.



Note: The character class and race usually determine the abilities of a character within the game, which then define the types of actions the character can perform.

The strength of a character within the game is defined by the character class it belongs to and the type of actions it can perform. The performance of a character is defined by the value of the attributes defined within the character's class and race.

For instance, if we take two different character classes and compare them side-by-side, such as a human and an orc, the orc will have far superior strength and brute force than the human. However, the human may have higher intelligence and problem-solving skills, which will out-rate the strength of the Orc if applied properly. Take a look at the following screenshot:



This is another key area where the RPG designer will have to spend a lot of time defining and specifying the specifications for the design and development of the characters within the game. The sky is the limit when it comes to designing and defining your characters, but there are some key attributes that you will need to consider for any RPG.

Most RPGs allow the player to modify their characters before the game starts, or even during game play. By default, every character class will have some default attributes, and the player is allowed to adjust the values based on some modifier. The basic fundamental features allowed for modification are the sex, class, or race of a character.

It all comes down to the budget and resources that are available to you during the production of the game. In some games, you can also introduce ethical attributes into the characteristics of the character. For instance, if you allow the ability to kill or rob innocent bystanders within the game, then the player will become less liked by the friendly non-player characters, and they may not be as friendly or helpful as needed to complete your quest. In other words, you will live by the consequences of your actions!

As a final takeaway, character classes define your character attributes and hence define your character's strengths and weaknesses. These physical attributes can be simplified into the following: dexterity and strength, which determine the performance of a character during battle!

Experience and leveling

To engage the player and to get them hooked on the game, game designer use mechanics to enhance the performance of the player-character. The progress is what is termed leveling or experience in RPGs.

Leveling and experience are a key element of any role-playing game. A good leveling or experience tree will be defined for any RPG. This allows the player to develop their avatar through game play and become functionally more powerful by gaining more skills, points, and other resources necessary to complete their quest.

The ability to acquire new weapons, armor, clothing, and/or any other game play item defined in the world, the player will need to meet some specific thresholds within the game. These thresholds can be a combination of the player's acquired experience points, financial gains, and/or combat experience. There is no right or wrong when it comes to designing any of these hierarchies and/or systems. You will need to see which one works for your specific needs and how to best apply them. Take a look at the following screenshot:



In RPGs, the progress of the character player is measured by counting some defined attributes specified by the game designer. Usually, the advancements are defined by the player completing a certain task to get experience points, and slowly, the tasks and the points rewards are increased throughout the game. The player then can use the experience points to enhance his or her avatar within the game.

Again, this is highly integrated with the story line, character classes, and/or race the player has selected. Some common ways to acquire points are by killing enemies, combating non-player characters of no importance, and performing quests that have been defined within the game.

Just like in real-life, the more you play and apply your skills, the more experienced you become. The accumulation of your experience will then allow you to acquire better weapons and/or armor to strengthen your attack or defense for the next quest. Some games might give the player 100+ points and allow the player to distribute the points across the available character attributes for their avatar. Sometimes, the game automatically applies all of the experience to a specific area, such as strength.

Gaining experience will also allow the user to unlock more features and skills to be acquired by the player during game play. This is a great way to monetize your games. In reality, most free games use this principle. They provide the basics of the world and the character for free, and they monetize the game through what are called in-game purchases, to increase either resources and/or character performance.

How is this implemented? Just like the inventory system, we need a way to keep track of the progress of the player's skills. This is usually done through a skill tree. Learning or acquiring a particular skill in the tree will unlock more powerful skills and give the player the ability to utilize the skills in the game.

Combat system

Time for battle! This is the moment every player looks forward to during their journey to kill the boss, the bad guy, or the evil warlord! Every RPG has some type of combat or battle component built into the game play. This is when the player gets to use all of their acquired skills and experience to destroy the opponent or be destroyed by the opponent, depending on how the day goes.

Historically, there were three basic types of RPG combat system. What type of combat system you choose to implement for your game will have a big impact on the game play, as well as the implementation of the game.

The three types of combat system are as follows:

- Traditional turn-based system
- Real-time combat
- Real-time with pause

Historically, role-playing games used to implement turn-based combat systems. This type of combat system is as follows: only one character could act at a given time. During this time, all the other characters had to remain still. In other words, they could not take any action. This type of combat system is designed to put more emphasis on rewarding strategic planning.

The next type is the real-time with pause combat system. This type of combat system is also strictly turn-based, but with a catch. If the player waits more than a certain period of time to make a move or issue a command, the game will automatically pass on the command to the other player. This will allow the other player, that is, the enemy, to take a turn and attack the player.

In this book, we will be using a real-time combat system. Real-time combat imports features from action games and creates a hybrid action RPG game genre. Action RPG combat systems combine the RPG mechanics of role-playing with the direct, reflex-oriented, arcade-style, real-time combat systems of action games, instead of the more traditional battle systems of RPGs.



Shigeru Miyamoto is best known as the creator of many of Nintendo's most beloved characters and franchises, including Mario, Donkey Kong, The Legend of Zelda, and many others. He was also the chief designer of Nintendo's Touch! Generation console series, which includes the Nintendo DS, Wii, and 3DS.

For more information, visit <https://www.giantbomb.com/action-rpg/3015-8592/>.

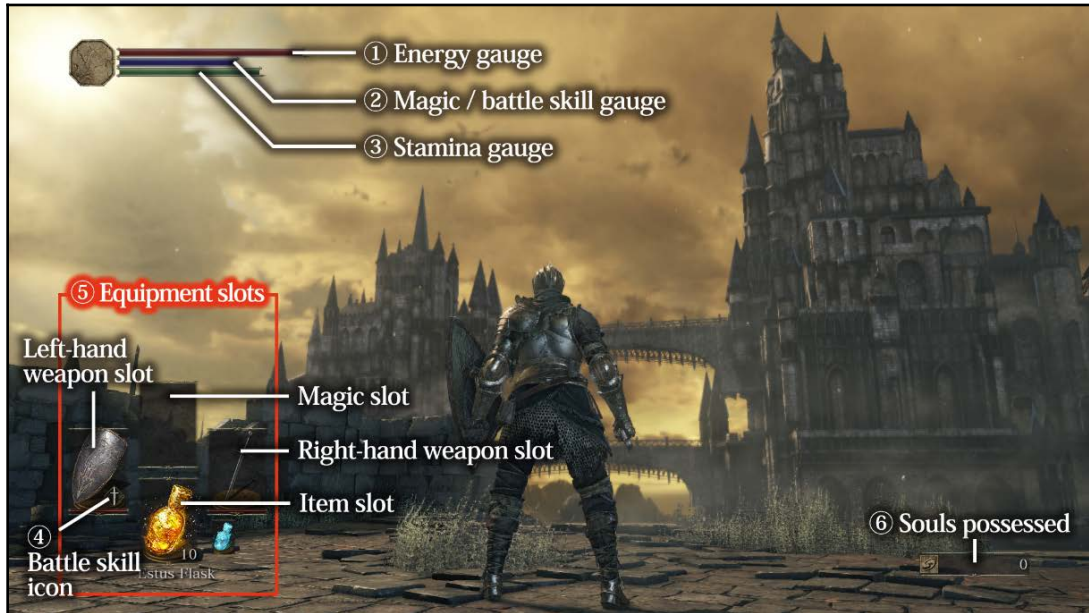
User interaction and graphics

The question arises, how do we present our game world to the player? What kind of user interface are we going to provide for our game? What kind of view are we going to allow for our game? Are we going to design our world to be viewed as a top-down camera view? Are we going to create an isometric view of the world? Or are we going to create a first-person or a third-person perspective of our world?

Answering these questions is crucial, as when you are designing your game assets you will need to understand how they will be viewed in the game world. For instance, when designing your characters and/or 3D models for the game, if you know that you will be using an isometric view, then you will approach your modeling differently than, for instance, when you are designing for a first-person or third-person camera.

In our game, we will be using a third-person camera view for the presentation of our world.

The next question would be how to provide critical information to the player in a simple and meaningful fashion. Role-playing games require the player to manage a large amount of information and frequently make use of windowed interfaces to arrange the data for the player. This is usually designed and implemented through a **Heads-Up Display (HUD)**. Take a look at the following screenshot:



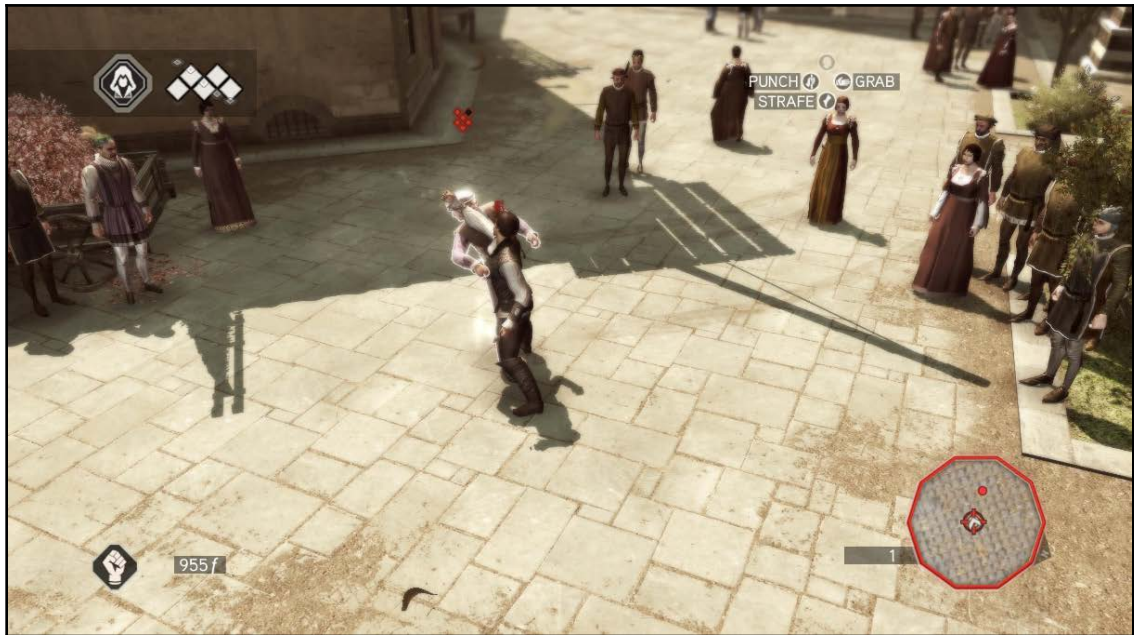
HUD is frequently used to simultaneously display several pieces of information, including the main character's health, items, and indication of game progression. You can think of the HUD as the access point for all the information the user will be required to have access to and interact with during game play.

The design of the HUD is crucial for RPG games. Typically, there are a few key data elements that you would like to continuously communicate with the player throughout the game play, which are as follows:

- Health
- Energy
- Stamina
- Active weapon
- Active shield

- Special items
- Number of lives
- Access to main menu
- Access to inventory
- Access to skills

Once again, the design of your HUD is derived by the type of game you are designing, as well as the type of information that will need to be available to the player during game play. Take a look at the following screenshot:



Since most RPGs collect and store large amounts of data for the player character, it is very important to create an easy-to-use, yet clean HUD.

A very important thing to remember when designing a HUD is that it should never overpower the screen or become a distraction. It usually takes a few stabs to come up with a great HUD design for your game, from initial artistic concepts, to the actual implementation and testing by gamers to get some feedback before finalizing the design and internal workings.

At the end of the day, the HUD is supposed to simplify the game play for the player, and not make it more confusing. Today, many games are moving away from traditional HUDs, leaning more towards cinematic or extremely simplistic experiences during the game play. This enables the game designer to immerse the player into the world and not to distract them with a constant, static HUD.

Making a HUD that will fit into the game play and style of your game is basic. While an element-rich HUD might be extraordinary for a few games, an oversimplified HUD can be similarly successful, or more so. Everything relies upon the player encounter you need. So when you're prepared to make the heads-up display for your next game, ensure you're outlining the HUD to upgrade the player's involvement, and never over-burden the player with data.

Existing or upcoming RPG games

This section of the book will take a look at some of the existing or upcoming RPGs on the market. The main idea behind this section is to provide you with a point of reference to multiple RPGs, as well as the game design implemented. It is also a good idea to research existing or upcoming games to get ideas for your own!

MU Legend

This is the sequel to a popular Korean MMORPG game. MU2 is a title that has been produced and published by Webzen Inc. The game is set on the mythical mainland of Mu which, as indicated by the hypothesis of James Churchward, was occupied by the antiquated human progress of Naacal.

Players expect the parts of saints looking for fortune, undertakings, and a few beasts to chase in this amazing domain. Game play in MU2 centers around powerful battle taking after regular hack-and-slice mechanics. Take a look at the following screenshot:



MU Legend

The game tells the story of a war between the god of light, Rugard, and the god of destruction known as Sekneum that raged for many years. It finished with the thrashing of the latter at the hand of Akneria, the Supreme Creator, yet the peace did not last long, and Sekneum acquired a human body and came back to the mortal world. In any case, the Great Mage Kundun saw through the fallen god's ruse and imprisoned him inside his own body. In any case, at last, he crumpled under Sekneum's strength and emerged as Dark Lord, expediting the world towards its end. This is where the players enter the world, going with the last sage, Icarus, as Devias Knights in a last endeavor to spare Mu from fast-approaching annihilation.

You can find more information at <http://mulegend.webzen.com/en>

Titan Siege

An allowed to-play MMORPG dream game including smaller-scale exchanges. The phenomenal setting of Titan Siege combines elements taken from Norse, Greek, and Egyptian folklore. The players take the parts of legends, who safeguard humankind from Titan groups taking up arms against each other. The journey begins with the character creation—the player can pick the sex, appearance, and battle details of his hero.

The game play expands on open-world investigation, finishing journeys allocated by NPCs, battling experienced adversaries, and building up the fundamental hero's aptitudes—this in the run of the mill MMORPG way. Throughout the game, the players acquire new things, such as captivated parchments, and bits of protective clothing, which help their characters during a fight. Travel through the immense areas can be accelerated by acquiring a steed. The two pets and inviting players can help us in fights. Players can likewise provoke us to a duel—the champ gets the hardware of the fallen legend. Take a look at the following screenshot:



The story in Titan Siege is set in a fantastical world, where elements of Norse, Greek, and Egyptian legends mix together to make a one-of-a-kind setting. The domain has turned into a combat zone, on which two groups of Titans take up arms against each other. Humankind chose to take matters in their own hands and proclaimed a war themselves, with the objective to put an end to the disarray caused by the Titans' war. Human armed forces are driven by saints—who are controlled by the players.

You can find more information at <https://steamcommunity.com/sharedfiles/filedetails/?id=561159376>.

Citadel: Forged with Fire

A multiplayer dream RPG created and distributed by Blue Isle. The generation is implied exclusively for online play and is sandbox in nature, giving players a chance to pick their own particular profession ways.

The players play the part of magical adepts and utilize their spell-learning to investigate a fantastical domain that is loaded with threats and privileged insights, traveling, building fortresses and attack vehicles, and challenging other individuals. Gamers can locate their own homes which are counterparts of organizations and sets up their principles. At that point, we can battle against comparable gatherings for impact, an area, and fortunes close by our confidants. The game-play mechanics are controlled by the material science motor, so the various accessible spells sensibly affect the articles in the earth. The game is more enjoyable due to the likelihood of flying. Take a look at the following screenshot:



The activity is set in a fantastical domain known as Ignus and gives the players a chance to accept the part of arcane expressions adepts who set out looking for undertakings and magnificence. The diversion happens on a guide with a region of 36 square kilometers. It incorporates fields, thick timberlands, mountains, swamps, solidified tundra, and old demolishes that shroud significant fortunes and great curios.

You can find more information at <http://www.citadelgame.com/>.

Cyberpunk 2077

A sci-fi RPG title in the style of a table-top Cyberpunk game. This game was produced by the CD Projekt RED studio. The story takes you to the year 2077, including in fact progressed, though defiled universe without bounds. The game is set in Night City, a place well known to devotees of the Cyberpunk universe, where dismal interests and insider facts flourish.

While investigating the game's open world, you take in the complex, multi-string story through various exchanges, which regularly force you to make difficult decisions. The classes and character movement framework, battle show, and additionally many in-game expansions were motivated of the first pretending framework. The visual side of the game is based on an upgraded form of Red Engine. Take a look at the following screenshot:



Cyberpunk 2077 highlights an intense and develop approach towards the theme, drawing motivation from preparations, for example, Blade Runner. The title itself transports us to the year 2077 and presents us with a dream without bounds world, horrid and loaded with interest; the move of the title makes put in the famous Night City, which the fanatics of the Cyberpunk framework will as of now be acquainted with. Curiously, the CD Projekt RED studio has chosen to fabricate the amusement around the open world game play demonstrate, which enables the players to unreservedly investigate the world and finish side missions at whatever point they want.

You can find more information at <https://www.cyberpunk.net/en>.

Patterns in RPG

Just like any other technical project that might have patterns defined, an RPG can also utilize similar patterns that have been documented by Whitson John Kirk III in his book titled *Design Patterns for Successful Role-Playing Games*.



Note: Whitson was inspired by the book *Design Patterns: Elements of Reusable Object-Oriented Software*. His objective was to see if any special patterns existed in existing RPGs, and he approached this by examining specific patterns in successful games in the genre to detect and identify them.

In this section, we will be looking at some of the design patterns that have been identified, and that can be utilized for your own games.

Anytime you start a new project, regardless of what type, you need to have some clear idea of what exactly it is that you are trying to accomplish. This is even truer for designing a game. Since designing a game has many different components to it, you will need to identify what your game is going to be about. Some questions to start the thinking process are as follows:

- What are you trying to accomplish?
- What mood are you trying to evoke?
- What do the characters do?
- What do the player or players, in a multiplayer environment, do?
- What kind of activities do you want to reward and what kinds of reward do you want to provide?
- What age group does your game targets?
- Is your game going to have sequences?
- Will the game and the story extend with supplemental assets?

These are all important questions that will affect the design of your game. As you read this chapter and the book in general, keep a pen and paper handy so that you can write down all ideas that flash into your mind. This way, you can keep track of all your thoughts, and if you need to expand on them you can do so at a later time.

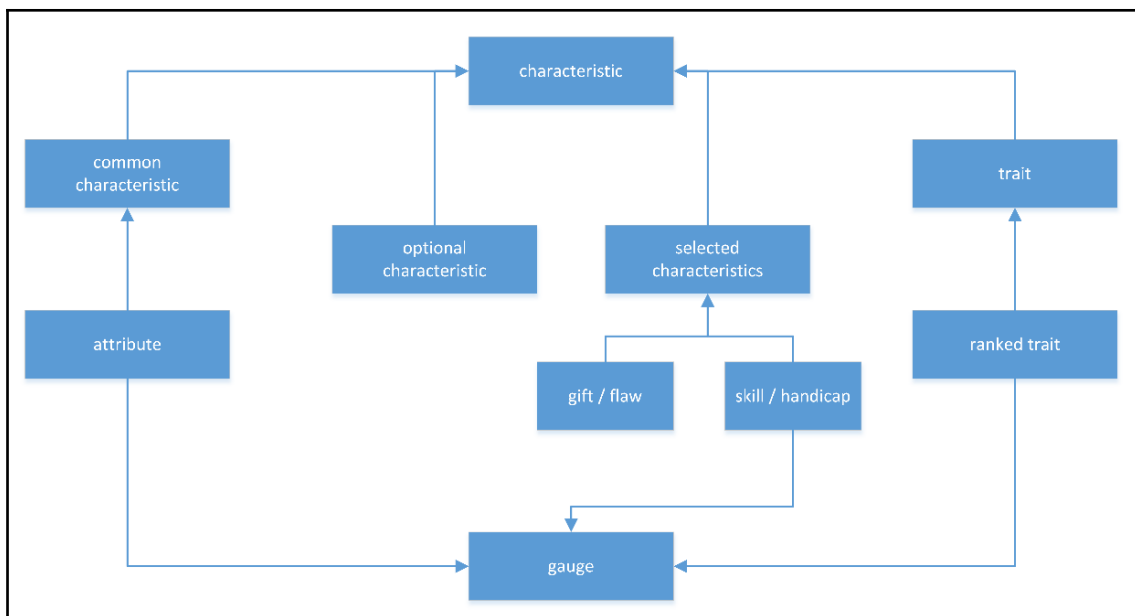
Terminology

Every discipline has its own terminology. The following is a list of terminology that is used in RPG games. It is a good idea to take a moment and study them, to expand your vocabulary or to refresh your memory:

- **Attribute:** A gauge that is a common characteristic, a commonality.
- **Character:** A person in a game portrayed by a player, including, possibly, the Game Master.
- **Characteristics:** An aspect of a character. A character's name, height, age, beauty, and strength are some possible characteristics.
- **Common Characteristics:** Characteristics common to all characters of a given type in a game. A character's name, height, age, beauty, and strength are frequently common characteristics.
- **Conflict:** Contention between characters, players, and/or game forces, especially contention that shapes the game's plot. This includes oppositions between two or more players concerning what facts should be introduced into a game world.
- **Contest:** A conflict that is resolved through mechanical means.
- **Derived Attributes:** An attribute whose value is determined by a formula. Typically, the formula uses other attribute values to generate a number.
- **Drama:** an outcome-based purely on story consideration. Outcomes in a drama are exclusively determined by what would be most entertaining for the participants.
- **Flaw:** a selected characteristic that is specifically not also a gauge. A character either has a flaw or he does not. Flaws are structurally very similar to gifts. But, flaws are generally considered detrimental to a character rather than beneficial.
- **Fortune:** An outcome that is at least partly based on random factors. This may include rolling dice, drawing cards, or some other random value generator.
- **Game Master:** Traditionally, a player assigned responsibilities to manage the game flow. With computer RPGs, the Game Master (GM) is the glue that holds everything together.
- **Gauge:** A graduated value generally associated with a name. Commonly the graduated values are numeric values.

- Gift: A selected characteristic that is specifically not a gauge. A character either has a gift or not. In general, gifts are considered beneficial to a character's well-being.
- Karma: An outcome based on non-random value comparison. A karma-based contest directly compares two values to determine an outcome.
- Non-Player Character (NPC): Any character portrayed by the Game Master as part of the role.
- Optional Characteristics: A characteristic that is not common to all characters of a given type.
- Player: Any person participating in a role-playing game.
- Player Character (PC): a character portrayed by any player while not assuming the role of a Game Master.
- Primary Attribute: An attribute whose value is set directly by a player rather than being derived by a formula from other attributes. Commonly, primary attributes are used in formulas to determine the values of Derived Attributes, but their own values are not determined by formulas. Typically, they are generated by random numbers or set by spending some resources.
- Rank: The specific value of a gauge skill, handicap, or ranked trait. Also used as an adjective in place of gauge when describing such skills and traits.
- Ranked Trait: A trait that is also a gauge.
- Selected Characteristic: A characteristic selected from a pre-defined list of choices.
- Shared Gauge: A gauge that is shared by many characters.
- Skill: A selected characteristic that is also a gauge and is generally considered beneficial to a character.
- Trait: A characteristic made up by a player without drawing it from a pre-defined list of choices.

To get a better understanding about the relationships between the attributes and characteristics, we have put together a visual help explain it:



Contest tree

The intent of a contest tree is to provide a mechanical means to create rising tension within a game. This is also known as Escalating Conflict.

Contest trees are high-level conflict resolution systems made up of many levels of contests arranged in a hierarchical fashion. The way they work is that lower-level contests feed into the higher-level contests and hence affect the outcome of the higher-level contests.

In other words, the higher-level contest could be to kill the big boss, but before you get to the big boss, there could be other mini battles that you will have to complete, and the outcome of the mini-battles will drive the outcome of the big battle. A simple example would be the amount of experience points you have gained before reaching the main boss.

Since the higher-level contests are somehow related to the lower-level contests, players do pay attention to the outcome of the lower-level contests; therefore, tensions arise concerning the eventual success or failure of a higher-level contest as a lower-level contest succeeds or fails.

It is best to use a contest tree when you want to create a sense of rising tension in your game. This can be achieved by applying different mechanics as the player progresses through the levels. Creating suspense in a cRPG is very simple, as you have a lot of control over the way levels and game play are designed. Since we have the ability to create our 3D worlds as we like, it would be easy to incorporate suspense into the game.

A few key points for creating tension in your game are as follows:

- The hero and enemy should be evenly matched.
- The hero and enemy should both periodically fail in their attempts, providing they are worthy adversaries.
- The hero and enemy's successes and failures are never so great that all hope of success of attaining the high-level goals is eliminated from either side.
- A highlight concerning contest trees is that it can only resolve high-level conflicts dealing with the mechanical inputs by the system. That is, if **Damage** and **Remaining Hit Points** are the only gauges used as an input into a conflict resolution, then the mechanics can only resolve issues dealing with **Damage** and **Hit Points**. To design a flexible contest tree, you will need to consider both the inputs as well as the outputs.

Last Man Standing

The Last Man Standing conflict system provides a generalized contest tree to resolve which side attains victory in battle.

The Last Man Standing is also one of the most traditional forms of generalized contest trees. The basic idea behind the pattern is to identify who the winner is by the simple fact of who manages to destroy the opponent. He who does is the winner. This is also one of the simplest ways to implement a contest tree.

It is used when there is a strong emphasis on tactical combat. Keep in mind that you are not obliged to use one pattern or the other. You can very well combine several patterns together, and you should, to make it more interesting. For instance, if your game has a great emphasis on combat, but you want to also introduce some negotiation into the conflict resolution, you can very much do so. Again, it all depends on you and your game design.

Something to keep in mind: if the only way to resolve a high-level contest is through battle, then players will focus their efforts on being the best they possibly can at winning battles. In other words, if a game only provides a single tool to resolve disputes, then you can be assured that players will become very adept and focused at using that tool.

Negotiated Contest

A Negotiated Contest provides a mechanical means to resolve disputes where the set of inputs and possible outcomes is negotiated by the player and non-player characters specifically for the conflict.

Designing and developing a Negotiated Contest mechanism is pretty complex. In order for the pattern to work properly, you will need to consider all of the inputs and outputs of the negotiations. The challenge of developing such a system is not so much the actual technical implementation, but the database that you will have to create and retain, based on the available selection by the player and the outcome of each input.

The implementation of the mechanics can be as simple as a few options and outcomes, and as complex as a variety of options and their eventual outcomes down the line. The important point is that you will have to introduce a negotiation mechanism after a conflict has been introduced, but before any action has been taken! The outcome is obviously what a win/lose concerns with, based on the negotiation before the conflict.



Note: The Negotiated Contest pattern requires that players be allowed to negotiate the effects of success and failure before the conflict is mechanically resolved.

Use the Negotiates Contest pattern when your design goals include one of more of the following:

- A desire to unambiguously decide whether the outcome of a contest means a player wins or loses his stated goals, rather than whether or not his character succeeds in performing discrete actions
- A need to scale the resolution of contests to levels of granularity different than that of individual actions
- A willingness to allow players a great deal of narrative freedom in describing the results of contests, both good and bad

For computer role-playing games, the negotiated contest will have some restrictions, as we cannot afford to create an AI system to be open-ended. But we can design a simpler means to provide the player a sense of some control on the negotiations, as part of the game play.

Negotiation can be a great mechanics for exchanging information, and good with non-player characters in your role-playing game. There are three parts to a Negotiation pattern, as follows:

- **Initiation:** This is the phase at which a character action is introduced into the game world
- **Execution:** This is the phase in which the success or failure of a character action is determined
- **Effect:** This is the phase in which the results of a character's actions are determined.

Here are some questions to consider when designing a Negotiation system:

- What does the winner get?
- What does the loser get?
- How do we know who's the winner and who's the loser?
- What do we need to establish before resolution begins, and how?

Summary

In this chapter, we covered in great detail what a role-playing game is. We briefly covered some of the historical aspects of the genre and looked at the different varieties of RPG. We discussed the key elements for designing a role-playing game and provided some examples to demonstrate.

We looked into the characteristics of an RPG and discussed how to plan for your game, such as story and setting, exploration and quests within your game, different types of inventory systems, character development, user interaction, and some combat system patterns. We also covered some basic terminology that is used in RPGs.

By the end of the chapter, you should have a clear idea of what you will need to prepare for and the kind of effort it will take for making a role-playing game.

In the next chapter, we will start planning the design and development of our RPG.

2

Planning the Game

In the previous chapter, you were given a good overview and historical background of RPG, and hopefully, it also sparked your imagination. In this chapter, we will start to lay the groundwork for our own RPG. We will start by defining the story of our game, as well as the plot and the quests that will make the game playable. We'll look at the assets that will be required to create our environment, characters, and finally, we'll design the levels.

The following is a breakdown of the topics we will be covering in this chapter:

- Building our RPG
- The story of the Zazar dynasty
- Asset inventory
- Level design
- The awakening
- Testing the level
- Creating the main menu

Awaken your creativity and let your imagination go wild!

Building our RPG

As discussed, building a role-playing game is no small task, but once you start down the path, you will come to the realization that it is not as difficult as it seems initially. The idea is to get started, and as you put your ideas down on paper and start the design process, more and more ideas will come into perspective.

As we have learned, there are some key elements that we will need to establish for our RPG. Let's recall them, and maybe even fine-tune them as we go along.

Our game's key elements are as follows:

- Story and setting
- Exploration and quest
- Items and inventory
- Character action and abilities
- Experience and levels
- Combat system
- User interface

The story of the Zazar dynasty

The premise of most role-playing games tasks the player with saving the world. There are often twists and turns as the story progresses, such as the surprise appearance of an estranged relative, or enemies who become friends and vice versa. We will create our story and game based on such a story.

Backstory

Once upon a time, there was a great kingdom, ruled by the great King Zazar. The ruler of the kingdom was a generous lord to his subjects. The kingdom under the rule of Zazar was peaceful and prosperous; however, over time, internal family rivalries and struggles caused cracks in the strong bond that kept the kingdom intact.

Due to mysterious events, the great king decided to move his family away from the kingdom and trust his son—who will become the player character—with one of the wise elders that he trusted. The kingdom was never the same... Until now!

Exploration and quests

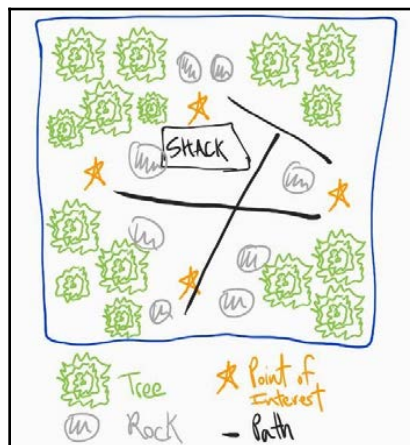
Now that we have defined the setting for our game, we can start working on developing the story further and breaking it down into different levels. To keep things simple, we will concentrate on basic quests and level design; the important point is to understand the concepts and apply them to your own story.

Awakening

The game will start by immersing the player in the environment, where our hero has been raised and trained by the elder, who was entrusted by the great King Zazar.

The main objectives of this level will be for the player to engage with the environment and learn how to interact with their surroundings, as follows:

- **Objective(s):**
 - Introduce the player to the user interface
 - Move the character
 - Interact with non-player characters
 - Interact with the environment
- **Outcome:**
 - Player gets points for completing in-game tasks
 - Player gets his first weapon
 - Player learns how to interact with the surrounding world
- **General sketch:**



The village

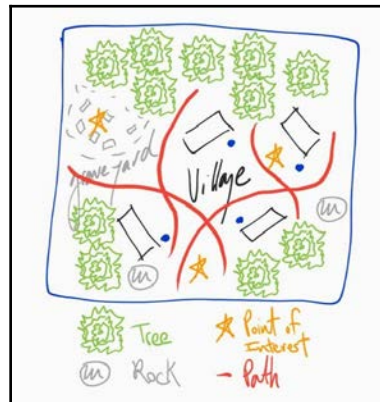
Our hero starts his journey of self-fulfillment. He will be traveling in the outskirts of the kingdom and arriving at one of the villages that has been terrorized by the thugs and mercenaries hired by the evil overlord, Shaquil.

Our hero, himself unaware of who he is and why he is on this journey, will find out about the austerity that has been going on since the departure of his father. This will mostly be accomplished through interaction with the village peasants.

The primary objective of this level will be for the player to learn social skills, engage with the villagers, and create relationships.

Rumour has it that there are spies in the village, and that everyone is suspicious of each other, and the unity that once was the strength of the village is crumbling. Our character's objectives are now as follows:

- **Objective(s):**
 - Interact with the village peasants to acquire social skills
 - Create trust between the hero and the villagers
 - Seek out who the spy is among the villagers
- **Outcome:**
 - Improved social skills
 - Relationships that can be tapped into at a later point
 - Basic combat skills learned
- **General sketch:**



Broken forest – the horizon

Our hero will be traveling along his quest into the horizon. The horizon is the initial exposure to the main kingdom's borders, where the main castle and inner city is within reach.

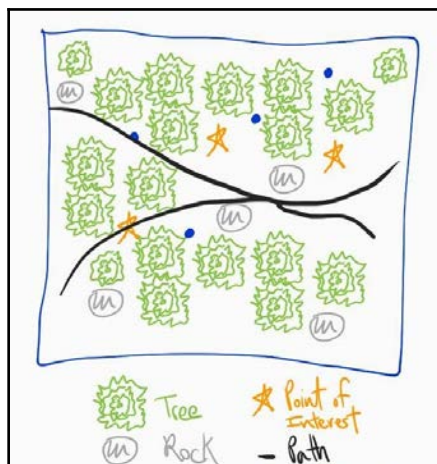
It is basically a vast, lush forest that protects the main domain of the kingdom from outside threats. It also has a few secrets and surprises for the uninitiated passerby. The forest is where the barbarians reside and cause havoc in the surrounding areas. What is not apparent at the time is the connection between the barbarians and the current overlord of the kingdom.

As far as the hero is concerned, he or she will need to be able to safely pass through the forest.

The horizon will have several unexpected surprises for the hero. The outcome of the quest will heavily rely on the way the player interacts with the surrounding environment, as well as with the non-player characters.

Now, our hero's objectives are as follows:

- **Objective(s):**
 - Pass through the forest without getting killed
- **Outcome(s):**
 - Hero can be captured by the barbarians
 - Hero could face other life-threatening scenarios and/or non-player characters
 - Hero successfully passes through the forest and is ready to take on the next challenge
 - Hero establishes new relationships to enhance his skills and abilities
- **General sketch:**

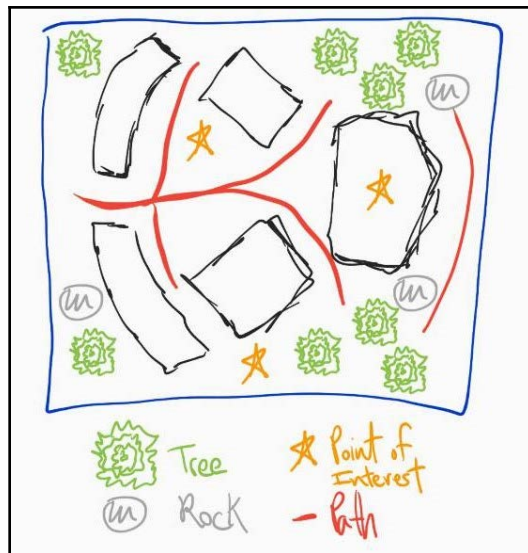


The kingdom

The hero has progressed through the previous quests and is now ready to take down the evil overlord and retake what is inherently his. Our hero has progressed and acquired a vast amount of skills and abilities throughout the quests, and now he will undertake one of the most difficult and epic battles in the game.

Our hero is surprised by the overlord's vast army. He will need to figure out a way to pass through the city and into the main castle to defeat the enemy. His objectives are as follows:

- **Objective(s):**
 - Kill the overlord and retake his kingdom
- **Outcome(s):**
 - Call to action the relationships he has established over the gameplay
 - Use his negotiation skills and wisdom to trick more powerful enemies
 - Destroy the enemy
- **General sketch:**



Ah, the pure joy of defeating your nemesis and taking over your kingdom!

Asset inventory

Now is a good time to discuss some of the basic assets that will be required for the development of our RPG. Our game assets are defined by the scenes we describe for our game. For our RPG, we have described four unique scenes. Each one has been described in enough detail for us to get an idea of the types of assets we will require.

Environment assets

The general theme of our game will be medieval. There are several ways to go about this. The first and preferred way is to either create the environment models by yourself or with a teammate; the second is to find a freely available model that has been created by a third party; and the third is to purchase 3D models created by a third party.

The **Asset Store** is a great place for you to start hunting for great content if you do not have the ability to create your own 3D models. You can use the Asset Store to search for medieval-themed environments that can be used for the game.

One of my favorites is called **Medieval Environment Pack**. You might want to consider searching for a few more that are to your liking and taste.

The following are some things to consider as part of your environment assets:

- Buildings
- Props and add-ons:
 - Banners
 - Barrels
 - Windows
 - Boxes
 - Wagons
- Rocks / plants / trees
- Particle assets:
 - Fire
 - Fog
 - Smoke
 - Water
- Skyboxes

This list is just a starting point, but it is a starting point for your environment assets.

I am using the following environment assets as a demonstration for the book, but you are more than welcome to choose any other package you wish.

Medieval Environment Pack

This is a great package if you are serious about creating an RPG with good quality graphics. It is very extensive and ideal to meet your environmental needs for medieval or fantasy settings.

The package includes the following:

- 70 modular buildings
- 180 add-ons – stalls, barrels, carts, and so on
- 25 rock formations
- Particles
- Textures

It is a bit pricey, but it is worth it if you lack the resources for a 3D modeler. The package is highly recommended. This is great if you are making a desktop or console RPG, but it is not suitable for mobile games.

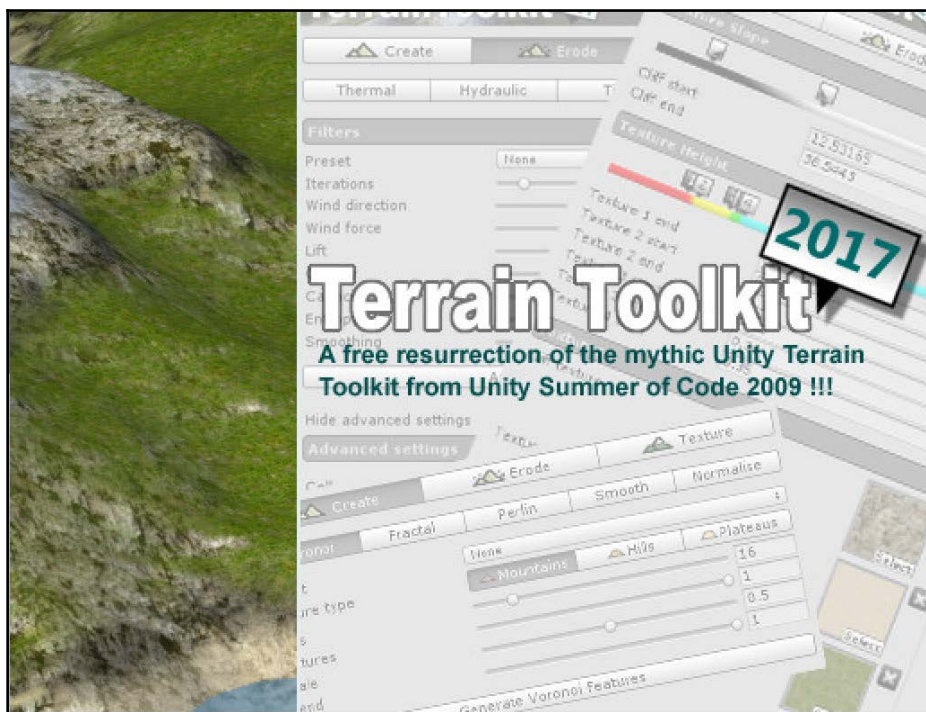
Take a look at the following screenshot:



Medieval Environment asset

Terrain Toolkit 2017

This toolkit is a great utility for terrain generation within Unity. It is from Unity Summer of Code 2009 and is made by Sándor Moldán. The toolkit has been upgraded to be usable in Unity 5.x and above. It is an enhancement to the original tool (V1.0.2). If you have been following along with the first edition of the book, you will also notice great enhancement in the actual user interface of the tool. The following screenshot shows the Terrain Toolkit asset:



Terrain Toolkit asset

Nature Starter Kit 2

The Nature Starter Kit 2 is a great asset that contains trees and bushes compatible with the built-in tree generator. This enables you to generate all kinds of new variations of your own! The package contains the following:

- Six bushes
- Four trees

- Ground textures
- Grass textures
- Skybox

The following screenshot gives you an idea of how the natural environment of your game will look if you use this asset:



Nature Starter Kit 2 asset

Character assets

RPGs are heavily based on characters. Therefore, the next important game asset will be the characters themselves. The models that you need to define for your game are again heavily related to your storyline and setting. The Asset Store provides a wealth of character models that you can download and use as a proof of concept for your game.

For our game, the following are the characters that are required:

- Humans: These will represent the hero as well as the villagers and other non-player character types of human

- Barbarians: These are some of the characters that the hero must confront during the gameplay
- Orcs: These are animals in their own right

You can either get the free models or the paid models to represent your characters. We will get more into the character assets in future chapters.

I have been working with a creator named Polygon for the specific models I will be using for the demonstration in the book. These are great characters to get you started. You can check them out in the **Asset Store**.

Barbarians

The Barbarians package comes with 15 unique textures. Each model goes from 1,700 to 3,000 triangles (no gear to full gear), with over 20 animations. They are Mecanim-ready and include Face Rig. You can see some examples of barbarians in the following screenshot:



Barbarians asset

Orcs

The Orcs package comes with unique models, accessories, armor, and weapons. Each model goes from 1,600 to 2,500 triangles (no gear to full gear) and includes 28 animations. It is also ready for Mecanim. Take a look at the following screenshot:



Orc assets

Villagers

This package provides a crowd of medieval humans ready to use. Great for fillers and non-player characters. They can be used to integrate inside the story as the hero progresses. Their total poly-count goes from 1,500 to 2,000 (no gear to full gear), and they are fully rigged characters with 20 animations. See the following screenshot:



Villagers asset

Free Assets

And finally, you have the ability to get a great amount of free assets from the Asset Store. Some of them are really of good quality. However, chances are that if you are looking for a specific asset type, you will need to spend a little bit of money:



Top free packages

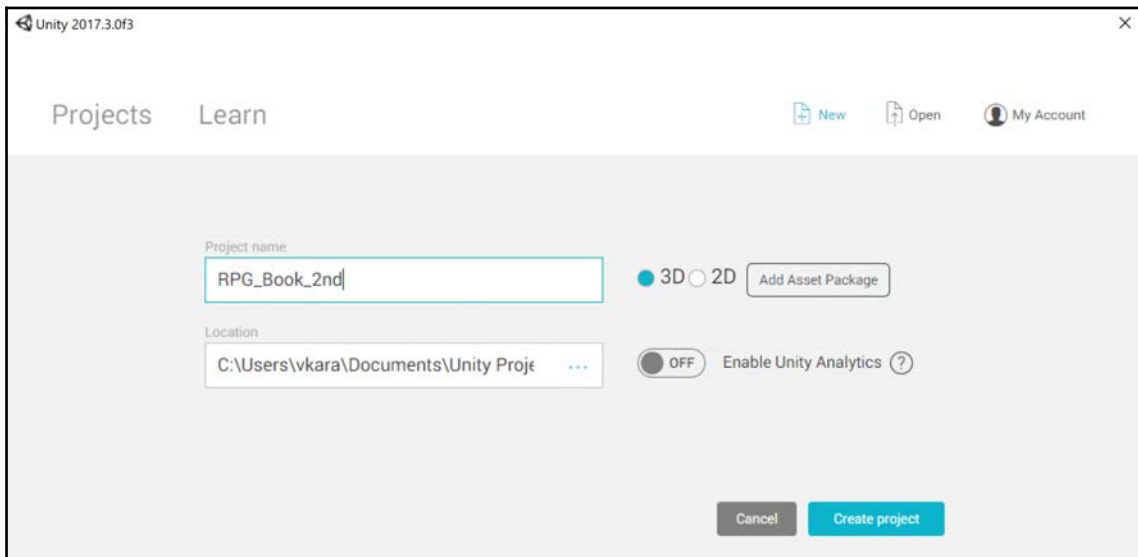
Level design

Now that we have our game story on paper and have an idea of what we want to achieve, it is time to apply our skills to actually making it happen.

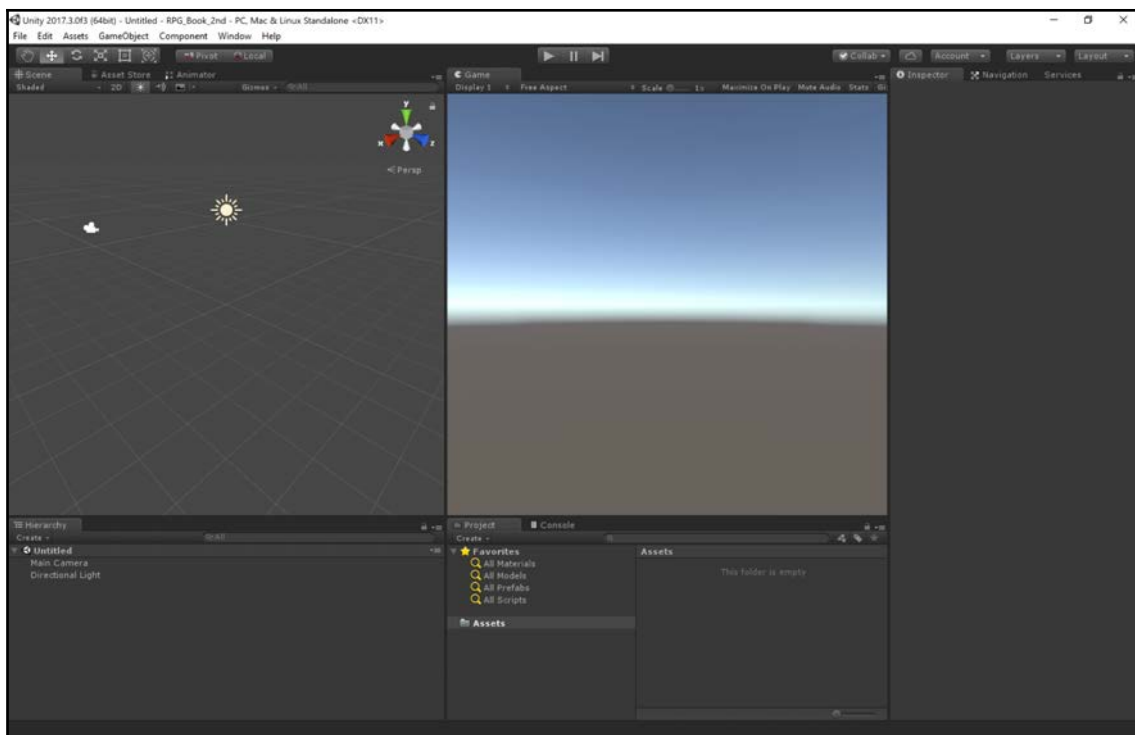


Since this book targets an audience that is already familiar with the basics of Unity, we will not cover the fundamental aspects of the software.

To get started, we need to launch Unity. I am using the 64-bit edition of Unity 2017.x Pro. You do not need to have the Pro version of Unity to complete the project in this book. See the following screenshot:



Go ahead and select a location and a name you desire for your project, and click the **Create project** button. At this point, Unity will create an empty project for you and display the Unity IDE. It should look something like the following:



New project



For better quality of images, download the graphics bundle from https://www.packtpub.com/sites/default/files/downloads/BuildinganRPGwithUnity2018_ColorImages.pdf

Your view might be a bit different, depending on how you have configured your Unity layout. If this is the first time you have launched Unity, you will need to get up to speed with the basics, since we will not be covering them in this book.



Note: If you have not used Unity before, you should get familiar with the IDE before you continue reading.

Setting the stage

The first thing we would like to do is to create a landscape for our first level, which is called **Awakening**.

Unity itself has some good tools for creating terrains, but truth be told, it is not a practical means of achieving nice, beautiful terrains for the game. For this purpose, we will use another set of tools, called Terrain Toolkit 2009, which was developed by *Sánder* as part of the *Unity Summer of Code 2009*.



The toolkit is available on the Asset Store:

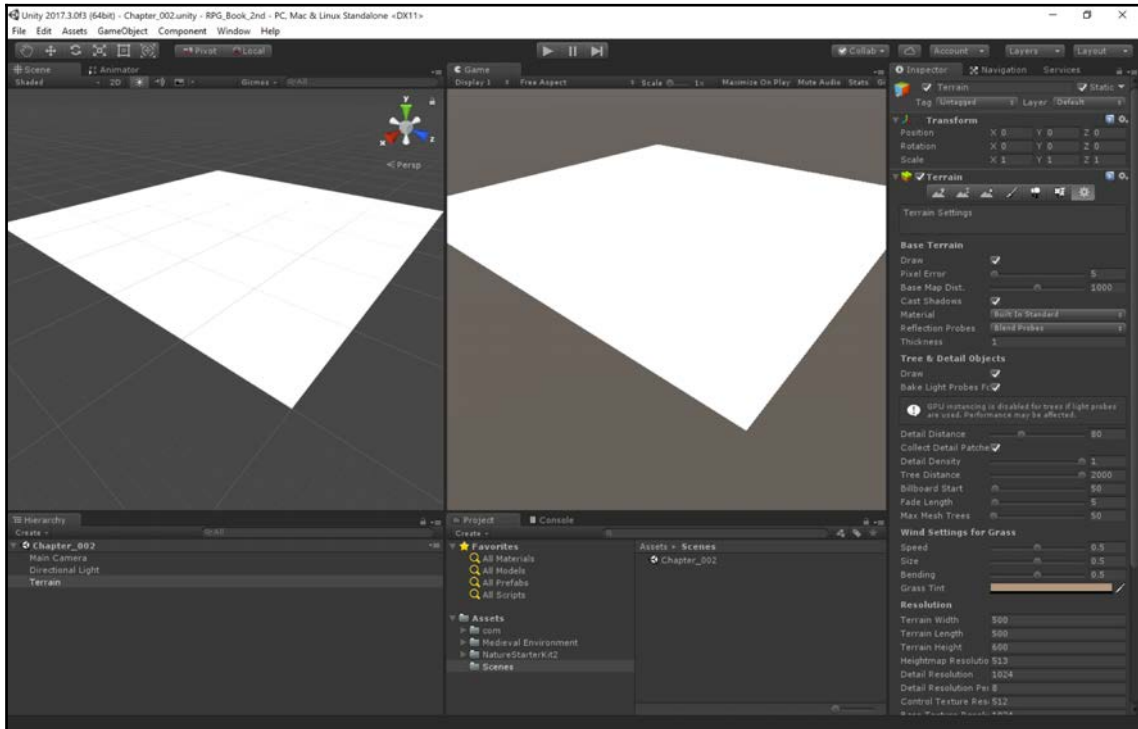
<https://www.assetstore.unity3d.com/en/#!/content/83490>.

I have also included the library as part of the download provided by this book, just in case the original link gets deprecated in the future.

Once you download the asset from the Asset Store, you will need to import it into the project. Simply click the **Import** button, if you have not done so already, to start importing the package:



At this point, let's take a moment and go back into Unity and actually create a **Terrain** **GameObject**, and see the built-in tool for terrain modification. To create a terrain, you will need to select the following from the main menu: **GameObject** | **3D Object** | **Terrain**. This will create a default terrain in your scene, which should look something like the following:



Demo of terrain

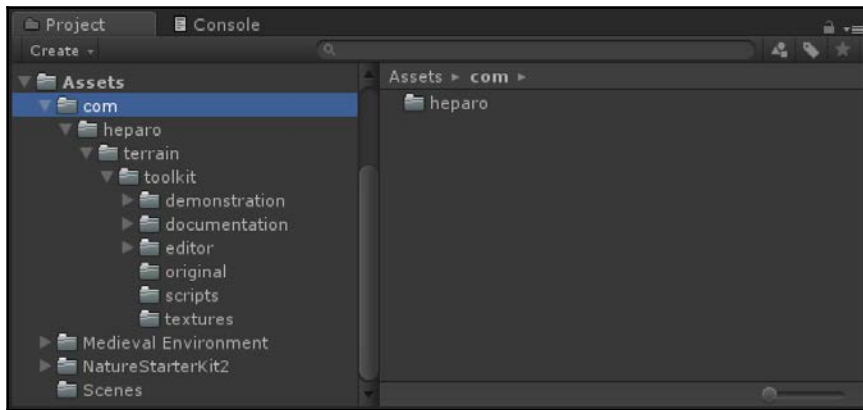
When you select the **Terrain** **GameObject** in the **Hierarchy** window, you will see the **Inspector** window displaying the properties and components that are accessible through the designer for the *Terrain* **GameObject**. As you can see, there are a lot of attributes that you can modify, and by doing so create a nice-looking terrain. When you start playing around with the terrain tool, you will soon realize that it is not practical for large terrain models, nor for natural-looking terrains.

To enhance our terrain generation, we will use the **Terrain Toolkit 2017**. You will need to import the package into the project if you have not already done so. If you have not already downloaded the package from the Asset Store, go ahead and download it, and then import it into the project.



Sometimes when you import older Unity assets, Unity will prompt you to automatically upgrade to the latest version. This is usually OK, so just accept it and let Unity do what it needs to do.

When Unity imports Terrain Toolkit 2017, you will notice a new folder under your **Project** window called **TerrainToolkit**. All the code will be listed under the folder if you want to make any modifications to it. There is also a `readme` file, which you can use to get started. See the following screenshot:



You will also notice that a new **Unity Editor** feature has been added under the **Component Menu** called **Terrain | Terrain Toolkit**. All you need to do to apply the Terrain Toolkit to your existing terrain is to select that option, and it will automatically attach the correct component to the Terrain GameObject for you:



If you have used the older version of the Terrain Toolkit, you will immediately see a great improvement on the user interface.

You will notice a few options that are now available to you through the Terrain Toolkit for the generation of more natural and realistic terrains. You should take the time to get familiar with each attribute and play around with the values, to get an idea of how they affect the terrain generation algorithm.

The new toolkit has predefined Terrain Models that you can use for a quick start. Here is a list of available models:

- Desert Mesa
- Desert Sands
- Sea Shore
- Snowy Mounts
- Pampa
- Path of the Flesh

I will let you play around with them and see the results for yourself. One hint would be to adjust your `Terrain` attributes a bit from the original values to get better visuals.

Terrain toolkit in a nutshell

There are a number of pre-defined generators in the toolkit for the creation of the terrain. They are **Voronoi**, **Fractal**, and **Perlin**. The following is a brief explanation of each:

- **Voronoi**: This creates an arbitrary tallness delineate of a progression of mountain-like pinnacles utilizing a Voronoi outline, and applies it to the territory protest
- **Fractal**: This produces an arbitrary stature outline of the cloud or plasma fractal calculation, and applies it to the landscape protest
- **Perlin**: This generates an irregular tallness delineate Perlin clamor, and applies it to the territory protest

There are also two filter types that can be applied after the generation of the terrain. These are the **Smooth** and **Normalize** filters, as follows:

- **Smooth:** This is a channel which applies smoothing to the territory protest more than once finished various emphases.
- **Normalize:** This is a channel which standardizes the territory protest by setting the most astounding point in the present landscape stature guide to the greatest, and the least point to the base. Every single other point is added between the most extreme and the least.

The next step is to apply some erosion to the terrain. There are three built-in erosion types in the toolkit: **Thermal Erosion**, **Hydraulic Erosion**, and **Tidal Erosion**. You can apply these erosion types either by a brush or by the actual erosion filters:

- **Thermal Erosion:** This expels material from territories with a slant more noteworthy than the base slant, and stores it additionally down the incline. This tends to smooth and straighten slants in the landscape.
- **Hydraulic Erosion:** This expels material from zones with an incline less than the most extreme slant and stores it additionally down the slant. This has a tendency to steepen slants in the territory and further smooth and straighten different zones.



Note: There are three different Hydraulic Erosion types.

- **Tidal Erosion:** This applies smoothing at the picked ocean level, with the exception of territories where the incline surpasses a given esteem. This reenacts the erosive activity of waves around a shoreline and makes shorelines.

The final step is to apply texturing. This will give our terrain a more realistic look and feel at runtime. The toolkit provides procedural terrain texturing, which automatically textures the terrain object using the slope and altitude attributes of the terrain to determine which texture will be used.

The Awakening

The setting and the atmosphere of our first level will be in a secluded area within the forest. We will now generate our terrain using the Terrain Toolkit discussed in the previous section.

Let's create a new scene and call it **Awakening**. By default, the scene will just have a camera and a directional light GameObject defined.

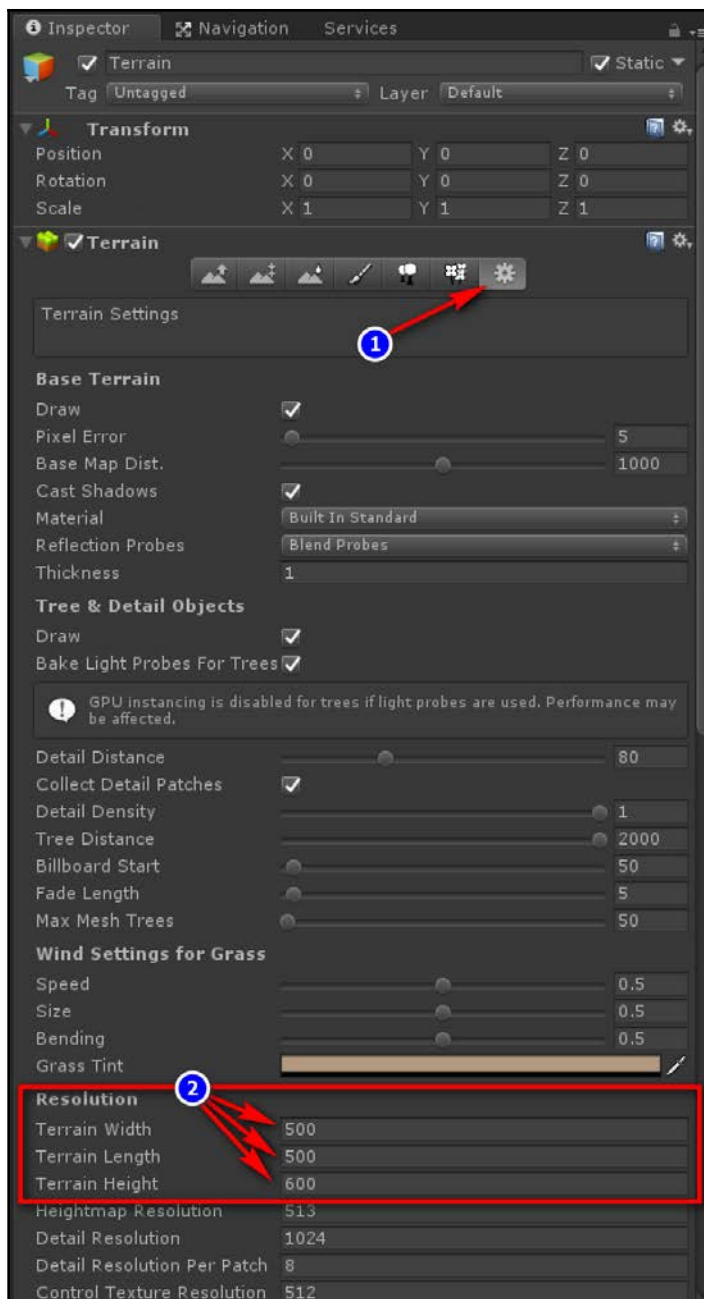


You can save your scenes and assets within the **Asset** folder without much thought. However, it is usually a good idea to have some file structure in place to make organization of your assets easier as well as finding them faster.

A preferred folder structure will include scenes, prefabs, textures, audio, and models. Within each folder, you can then create subfolders and so forth, for your own organizational purposes.

Now we are ready to add a Terrain GameObject to the scene. Go ahead and select **GameObject | 3D Object | Terrain**. This will place a **Terrain GameObject** in the scene. Double-click on the **Terrain GameObject** in the **Hierarchy** window to make it center of the **Scene View**.

By default, the Terrain object will be very large, so let's go ahead and make some adjustments before we do anything else. The following screenshot shows how to access the settings:



To make the adjustments to the terrain size, select the **Settings** icon, as indicated in the preceding screenshot. This will display the basic attributes of the terrain. As you can see, there are a bunch of properties that can be adjusted to make it behave to your liking. We are mostly concerned with the size of the terrain and also the maximum height that the terrain can rise to. Therefore, scroll down until you get to the **Resolution** section.

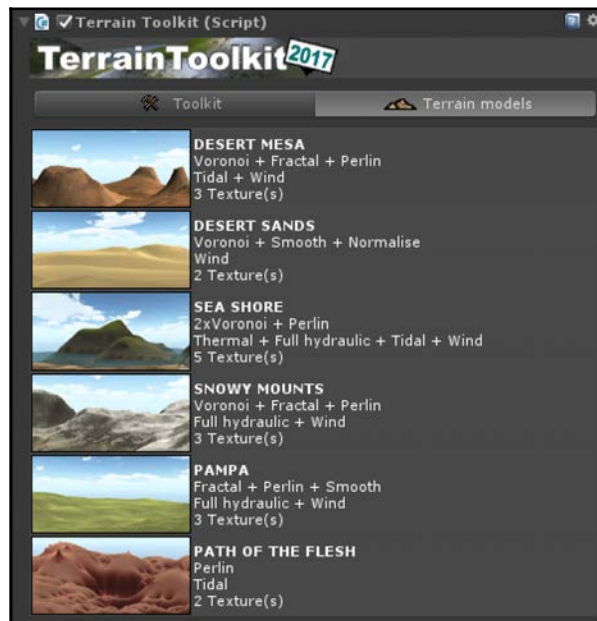
Change the **Terrain Width** and **Terrain Length** to **100**. Change the **Terrain Height** to **333**. This will change the dimension so that we can handle our scene easily. Our original terrain size was very large and it would have taken us a long time to design it.

Now we have a good size terrain. Assuming you have already imported the Terrain Toolkit, go ahead and select **Component** | **Terrain** | **Terrain Toolkit** from the main menu.

There are two ways we can approach this. Since the new toolkit has given us some prebuilt settings for terrain models, we can use that to quickly generate the random terrain, or we can use the default method of generating our own custom terrains.

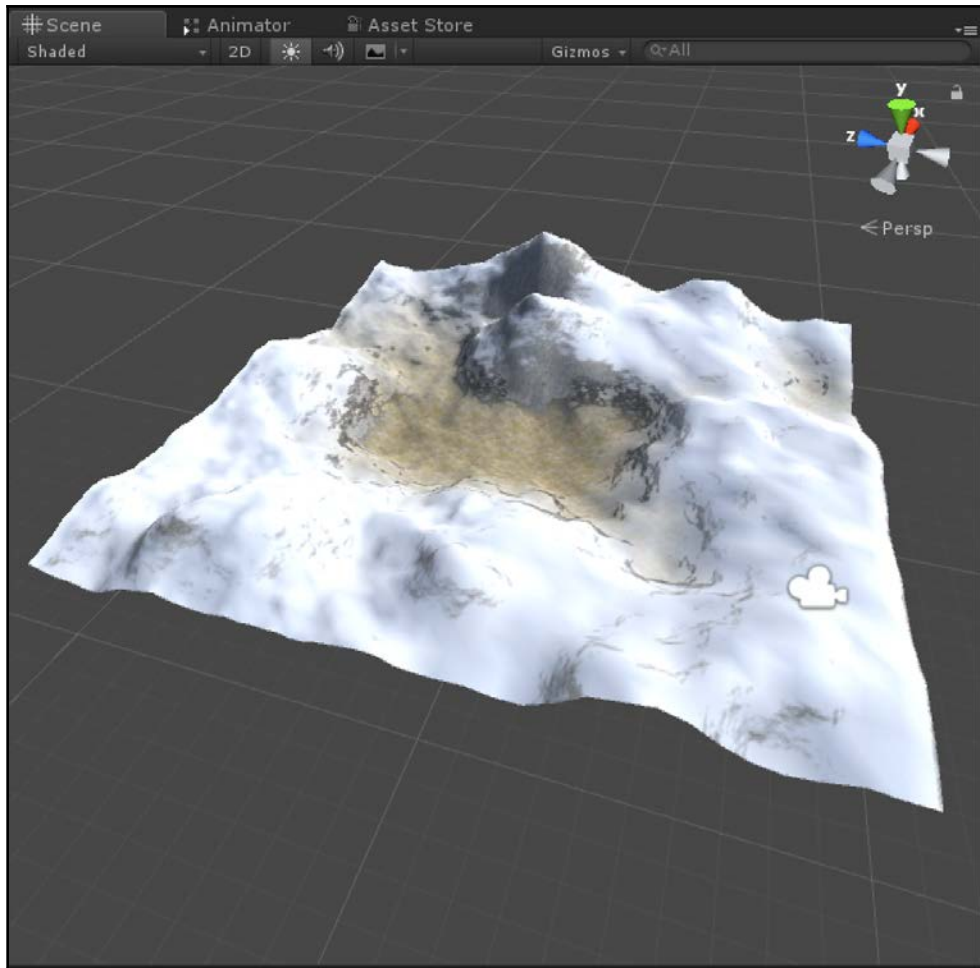
Using the terrain model

Select the **Terrain GameObject** and, using the **Inspector** window, go to the **Terrain Toolkit** and select the **Terrain Models** tab, as shown in the following screenshot:



As you can see, we have six pre-configured terrain models that we can use. Let's try out the **SNOWY MOUNTAINS** model. Go ahead and click on it, and Unity will start generating the model.

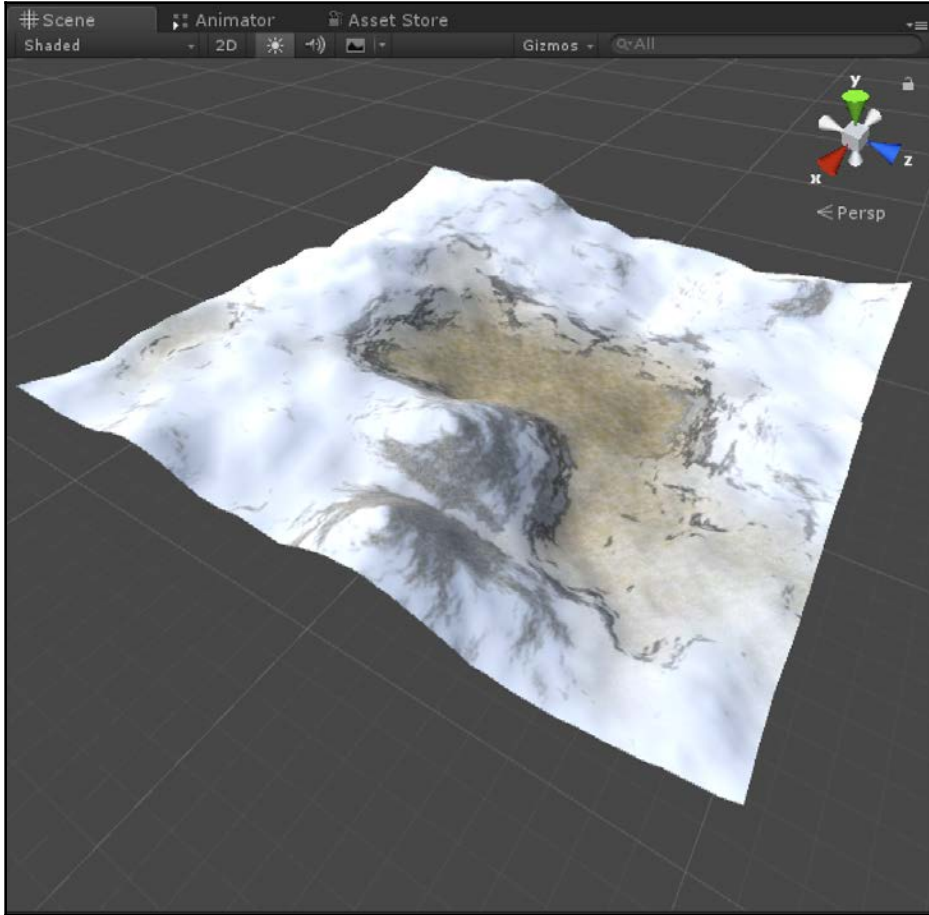
Here is what my terrain model looks like:



It looks pretty cool. Now let's try creating a terrain using custom functions.

Using a custom toolkit

Using the toolkit to generate a random terrain, I have used the **Fractal** terrain generator function with a **Delta** of 0.4 and **Blend** of 0.445. This will generate a nice-looking terrain with a good proportion of hills and valleys. Since the terrain is randomly generated, yours may not look exactly like mine, but it should look something similar to the following screenshot:



I usually apply the **Smooth** filter after my terrain generation to make things look even nicer.

You will see the difference the filter makes once it has been applied. Let's go ahead and now apply some textures to make it nice. Selecting the **Texture** tab within the **Terrain Toolkit**, you will have several options. We would like to apply at least two textures to give our terrain a more realistic look. You can apply up to four textures if you choose to! Click the **Add Texture** button twice to create the texture placeholder.



Since I used the **Terrain Model** to generate the original terrain and then applied the toolkit with the settings mentioned, the default textures for the model are in place. You can update them as needed.

Take a look at the following screenshot:



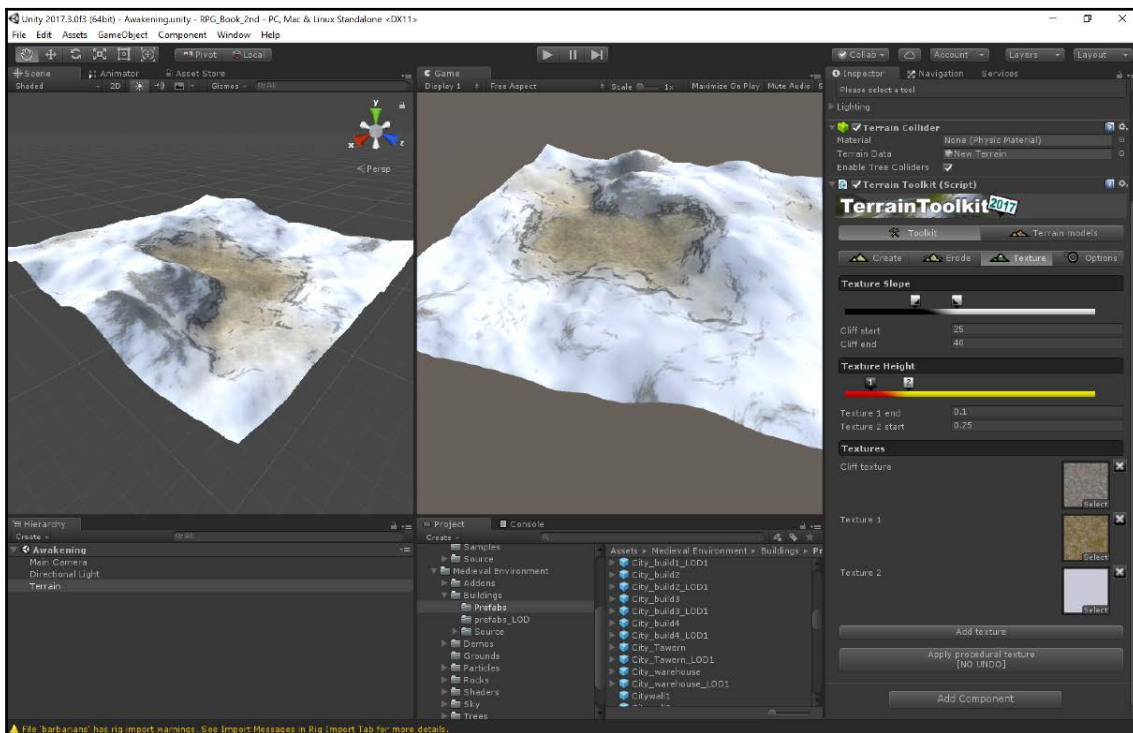


Textures are very important in graphics, and especially games. The better and higher resolution your textures, the better your scene will look. However, this is a catch-22. Usually, higher resolution textures take up more resources. So you have to find the right balance for your game.

Now is a good time to stop and discuss one of the main advantages of Unity: the **Asset Store**. The Asset Store is a great online community where Unity developers can either acquire assets to be used in their games or develop assets that will be used by other developers. You can get free assets from the Asset Store, or you can get better quality ones for a little bit of money.

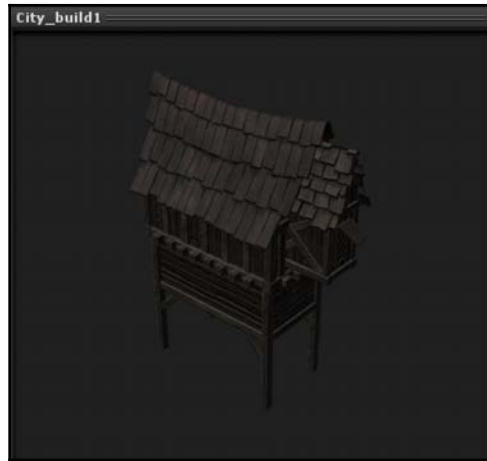
For our game, I will be using some free assets and some paid assets. If you want to use the same assets that I am using in the book, you will need to purchase them.

Take a look at the following screenshot:



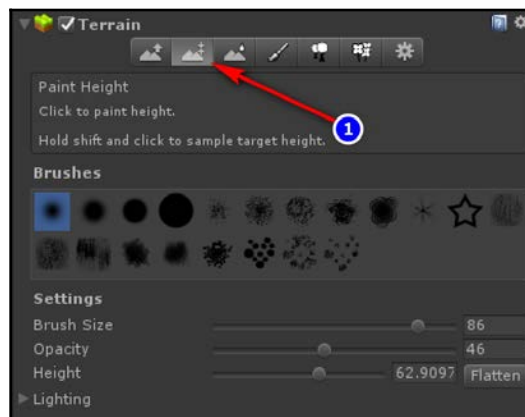
Sample of generated terrain

The next thing I like to do is locate a position on the terrain where I will create the scene objects necessary to play out the level. For this particular scene, I want to use an asset that represents an old cottage in the woods, where the hero will awaken at the beginning of the game. See the following screenshot:



This is a model from Medieval Environment Pack. The model does not have any interior. This is OK, since I am not planning to have any gameplay inside the structure. It will be used as an eye-catching object in the scene and a point of reference.

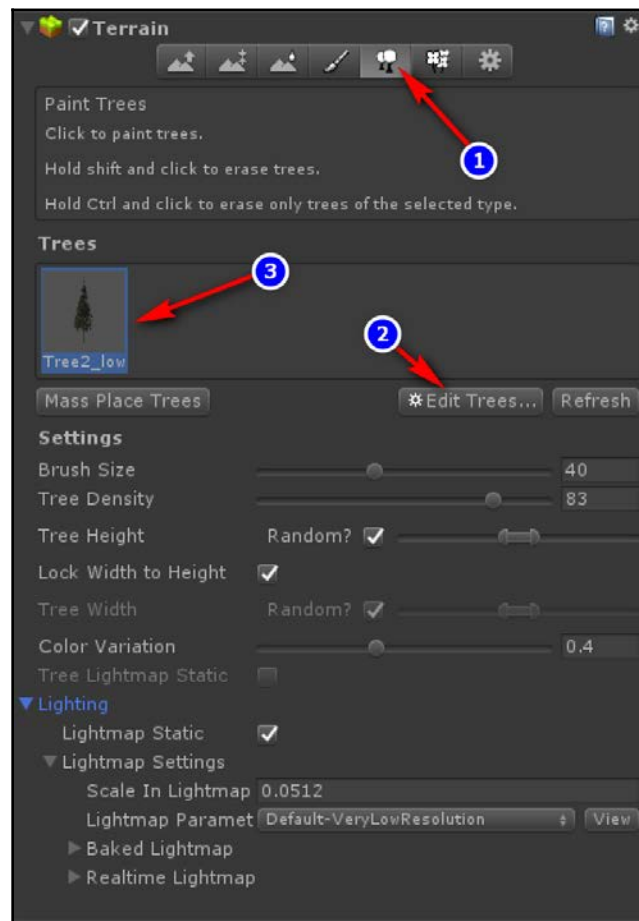
You can use the **Terrain** tool to level the terrain before placing the object into the scene. If you notice, our terrain does not have any level areas where we can properly place the shack. We will need to use the terrain object's terrain components to make some more changes, as shown in the following screenshot:



Again, select the Terrain GameObject in the **Hierarchy** view and use the **Inspector** window to select the **Paint Height** tab, shown as (1) in the preceding screenshot, to enable the feature.

This is a great feature to sample the terrain height at a particular point, and apply the same height using the brush to any other region. This will level the terrain to the same height that was sampled. It is a great way to quickly level a region and place your items, if those items need to be on level ground, like a house or a shelter.

The next step is to place some tree models on our terrain to create the forest look and feel. To achieve this, we will need to select the terrain object in the scene and use the **Inspector Window** to select the **Tree Placement** feature. You will then need to select the **Edit Trees...** button to add a tree model, as shown in the following screenshot:

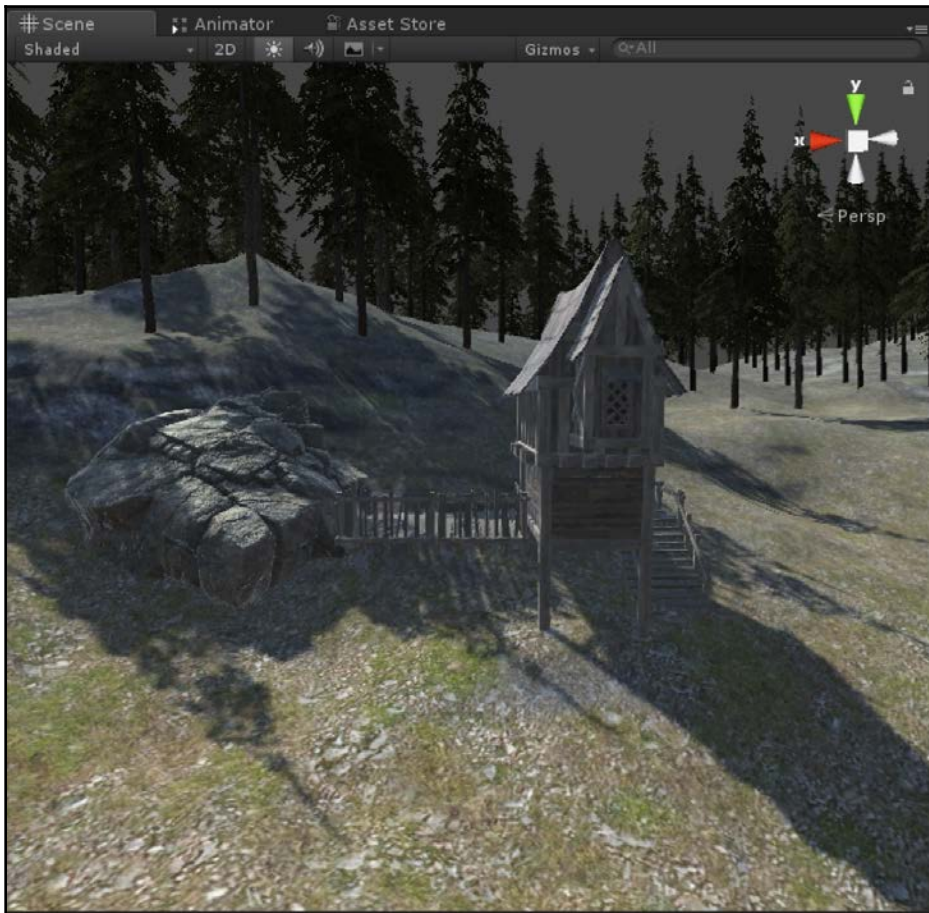


Using the **Inspector** window, you will need to select the **Tree Placement** tab, select the **Edit Tree... Add Tree** feature, locate a tree prefab and you are done!

Take a look at the settings and change the brush size to meet your needs. In my case, I have changed it to 40. I have left the rest of the attributes at the default value, but you can change them as needed.

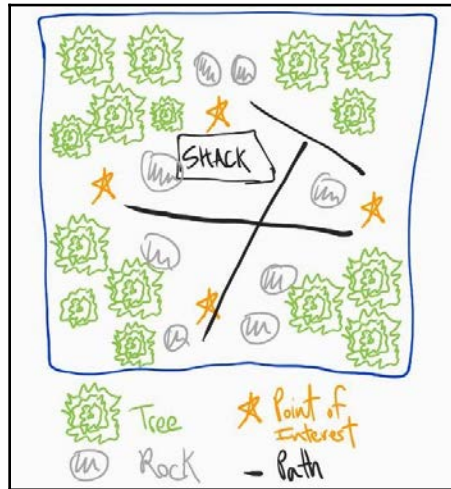
Now, when you move your mouse in the **Scene** view, you will notice a brush like highlight. This is where the trees will be placed during design time.

The following screenshot displays how my scene looks after I have made some of the adjustments to the placement of my trees and the building:



The next step of the process is to fill the level with other environmental assets, such as rocks, vegetation, and other props, to make the level come to life. The idea here is to make it interesting and at the same time functional. It is a good idea to have some sort of a sketch of your level design. This way you can have a good idea of how you will develop your level.

Keep in mind that this can also be used as a means of communication with your team, the level designers, and artists, providing them a direction. The following figure is a top-down view of our intended level design:

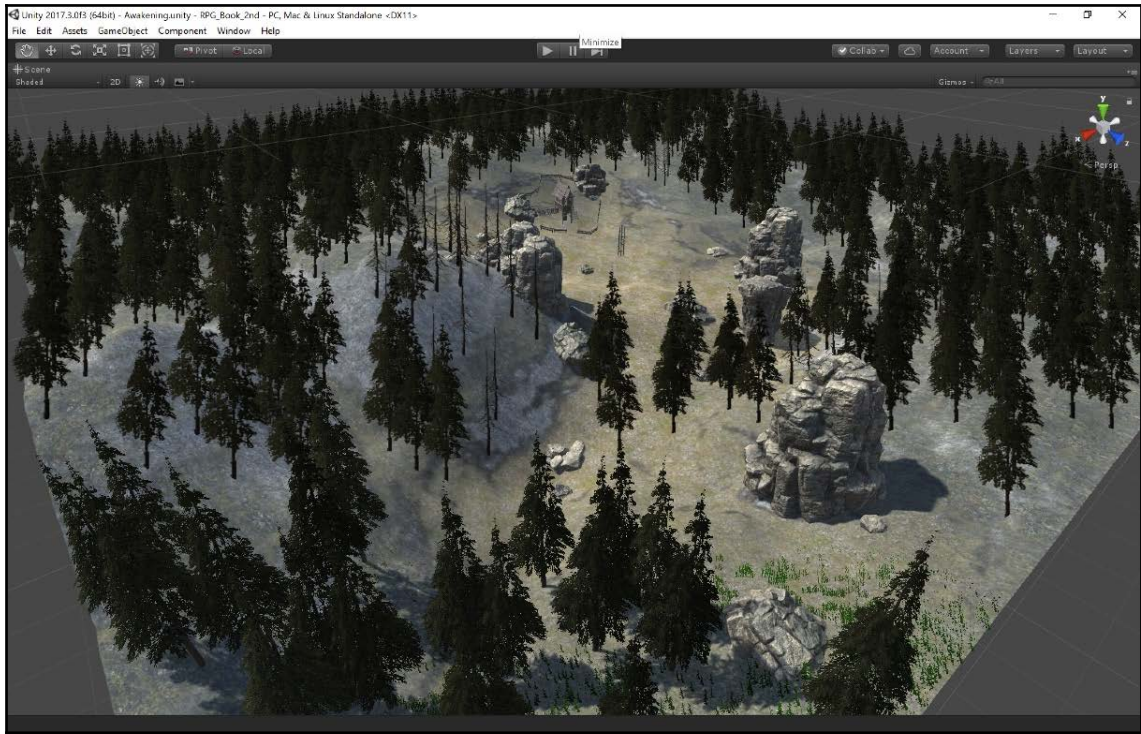


Graveyard

I will now build my level based on the layout I have. Now is the time to get creative and use your imagination to design your level. This part of the exercise is free-form; you are the designer, so you will decide how to go about placing and creating your level, as long as it meets the requirements.

Keep in mind the following important point: the player will be interacting with the environment or non-player characters at the designated points of interest. Make sure that when you design your level, they have an easy way to access the areas where they need to get to, so that they can perform the given task.

Take a look at the following screenshot:



Level design

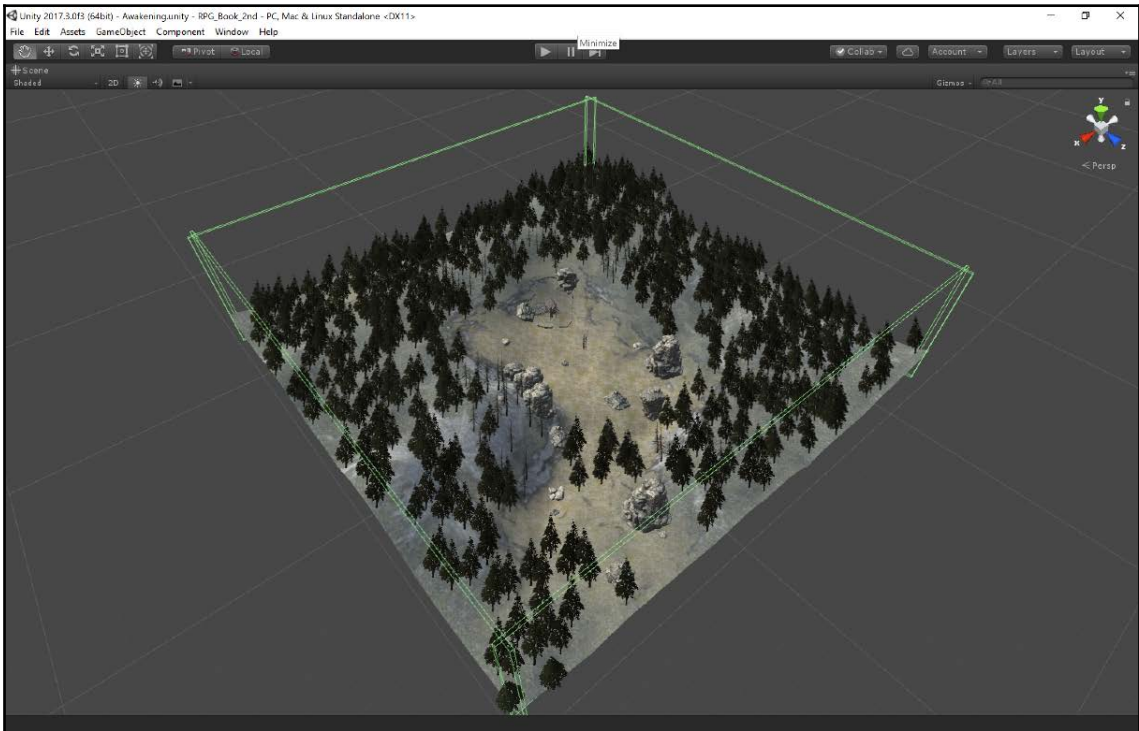
One thing to note is the limitation of the terrain as we have it defined here. When the player goes to the edge of the terrain, they will fall through! Yes, they will have a free-fall forever! We don't want that to happen. We need to incorporate some boundaries in the level design that will prevent the player from going overboard, basically.

It is very simple to create some restrictions and boundaries in such cases. We can use wooden fences, or we can use the actual environment to restrict the level of access by the player to the danger zones in the level. This method can be very time consuming if your level is large.

Another way to solve this problem is to create four colliders that will be used on each side of the terrain. The colliders will be attached to an Empty GameObject, and they will stop the player from moving forward upon contact. This is an easy method, and will take less time to place in the scene.

Go ahead and create an Empty GameObject by choosing **GameObject | Create Empty**. Using the **Inspector** window, adjust the **Transform** component to set the **Position** to $\langle 100, 50, 100 \rangle$. Next, we need to **Add Component** to the newly created GameObject. Use the **Add Component** button to select **Physics | Box Collider**. This will add a box collider, and now we can change the **Size** of the collider, using the **Inspector** window, to $\langle 200, 50, 1 \rangle$.

Once you are satisfied with the look and feel, you will need to duplicate it and place it on all the side edges to protect the player from falling, as shown in the following screenshot:



Level design

Skybox

Since we have our preliminary level design, we can go ahead and add another nice touch to it. Let's use a skybox to give a more realistic atmospheric look to the gameplay.

As always, you can use the Asset Store to search for existing skyboxes. Go ahead and search for one that you like and import it into your project.

In the **Hierarchy** window, select the **Main Camera** and use the **Add Component** button to attach a Skybox component to the camera: **Rendering** | **Skybox**. Assign a skybox material to the **Custom Skybox** property. Take a look at the following screenshot:



Adding Skybox

It is time to do a test run. We can use the third-person character controller provided in the Standard Assets to quickly drop our character placeholder and roam around the level to get a feel for it.

Testing the level

At some point, you will want to test out the level and look at it through the eyes of the camera. We can use the built-in third-person character controller that comes in the Standard Assets and do a quick walk-through of the level.

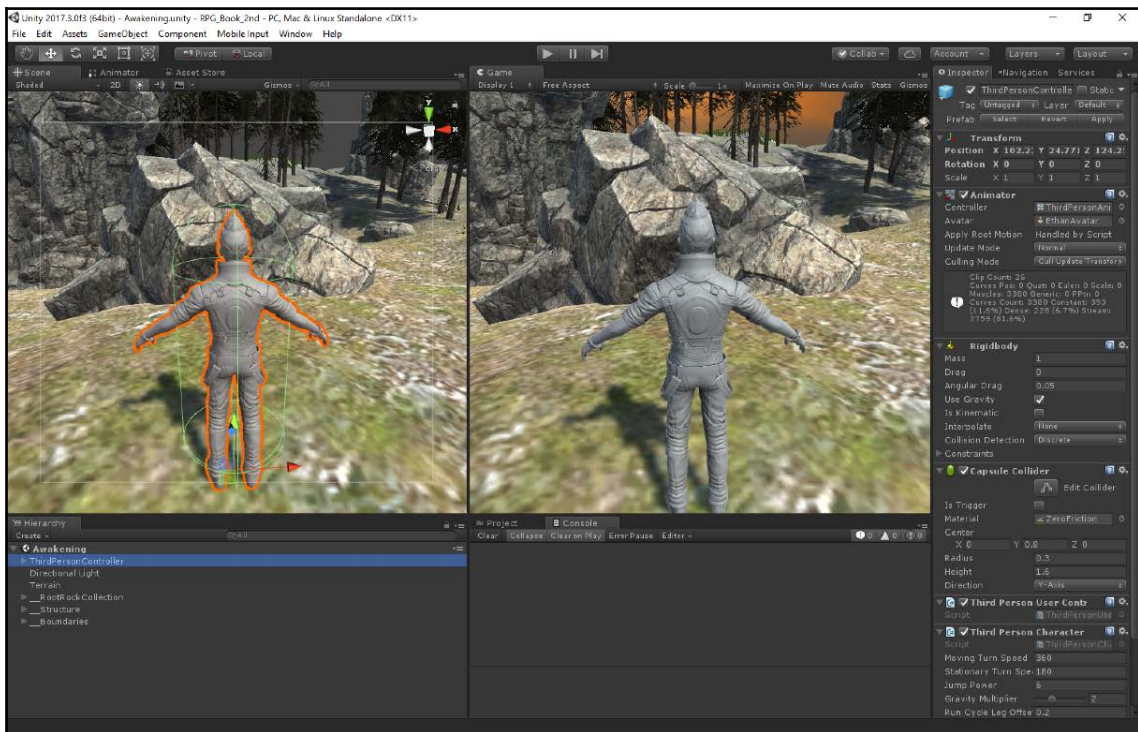


If you did not import the Standard Assets when you created the project, you will need to import them by selecting **Assets** followed by **Import Package|Characters**.

In your **Project** window, you will see a folder called **Standard Assets**; there is a subfolder called **Character Controllers**. You will need to select the **3rd Person Controller Prefab** and drop it somewhere on the current scene. A good location is next to the shack. Make sure that the **3rd Person Controller (3rdPC)** is above the terrain so it does not fall through!

You will have to attach a **Rigidbody** component to the **3rdPC** GameObject. This is needed to make sure that our **player-character (PC)** uses the built-in physics for collision detection.

Before you run the level, let's make sure the camera is following the character. Simply make the **Main Camera** a child to the **ThirdPersonController** and adjust the orientation of the camera to look something like the following:



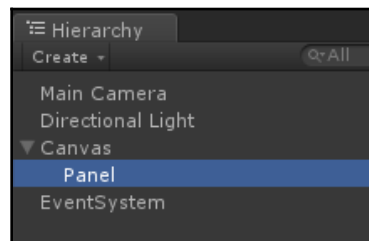
Standard character controller

Go ahead and run the level and walk-through the scene. Test and make sure that the PC is behaving the way it is supposed to when you are navigating through the environment. Make sure that the character is respecting all the colliders and that there are no minor errors in the design part.

Creating the main menu

Now is a good time to create the starting point for our game. Go ahead and save the current scene. We will make a new scene that will be used as the starting point of our game. To create the new scene, you will need to select **File | New Scene**. Go ahead and save the scene. I called my scene **MainMenu**.

Now we have a clear canvas that we can work with to create our **Main Menu**. In the **Hierarchy** window, right-click and select **UI | Panel**. This will create a **Canvas** GameObject and an **EventSystem** GameObject, and place them in your **Hierarchy** window. You will notice that the **Panel** UI Object is a child of the canvas. All UI elements will be a child of a canvas. Your **Hierarchy** should look something like the following screenshot:



There are several key aspects that we want to make sure are set properly. These are namely on the **Canvas** GameObject. Select the **Canvas** GameObject and look at the **Inspector Window**.

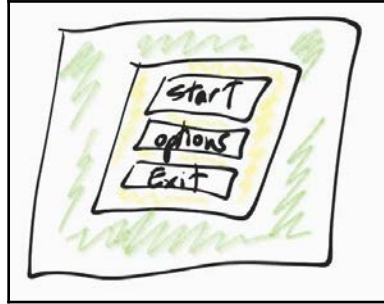
For this particular canvas, make sure that the **Render Mode** is set to **Screen Space - Overlay**. The next property you need to check is the **UI Scale Mode**. Change this to **Scale With Screen Size**. This will make sure that the UI will always be scaled to the screen size of the device that the game is being run in.



For best results, you will want to create multiple menus, for different device types.

For now, let's go ahead and create a button that will basically load our *Awakening* level.

Right-click on the **Panel Object** in the **Hierarchy** window and select **UI | Button**. This will place a button on the canvas as a child to the `Panel` object. The parent-child relationships are important to consider when you are building your user interface. When you place a UI element as part of a child to another UI element, the child will be scaled and moved according to the parent's scale and location.



You can learn more about UI development in *Chapter 5, Game Master and Game Mechanics*, of *Introduction to Game Programming: Using C# and Unity 3D*. Eventually, we would like to have a main menu that looks something like the preceding screenshot.

We will spend time fine-tuning our menu in future chapters. Change the caption of the button to `Start Game`. It is also a good idea to name your scene object appropriately, just to keep things nice and organized. I have changed the name of the button to `butStartGame`. This can be done by selecting the `Button` object in the **Hierarchy** window and changing the name in the **Inspector** window.

Creating the GameMaster script

As discussed in *Chapter 1, What is an RPG?*, we will need a way to manage our game. We will create a script called `GameMaster`. This will be the core of the game that glues everything together. As we progress with the book, you will see how we will modify the core to meet our needs.

For now, we are just going to create a simple C# script and name it `GameMaster.cs`. We will then create the code that will be used to handle some of the basic events we want to perform at this point, namely, navigating from scene to scene.

From your **Project Window**, under your `scripts` folder, right-click and select **Create | C# Script**. Name it `GameMaster.cs`. Double-click your script to start your code editor and place the following code in there:

```
using UnityEngine; using UnityEngine.SceneManagement; namespace
com.noorcon.rpg2e {
    public class GameMaster : MonoBehaviour
    {

        // Use this for initialization
        void Start()
        {

        }

        // Update is called once per frame
        void Update()
        {

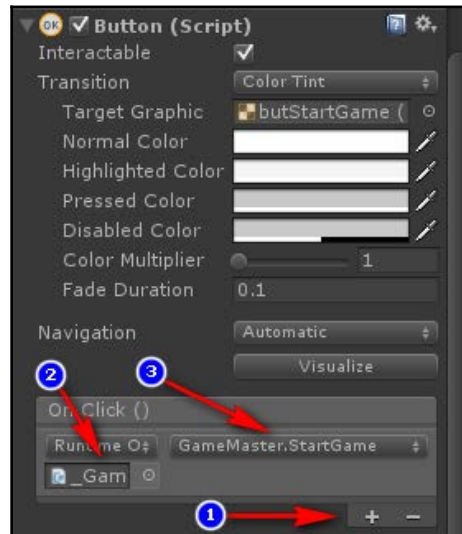
        }

        public void StartGame()
        {
            // You should put in the name of the scene that represents your
            // start level
            SceneManager.LoadScene("Awakening");
        }
    }
}
```

In the **Hierarchy** window, you will need to create an empty `GameObject`. The best way to do this is by right-clicking and selecting `Create Empty`. An empty `GameObject` will be created. Select it and change the name to `_GameMaster`.

We need to attach our script to the `_GameMaster` `GameObject` in our scene. Select the `GameMaster.cs` script and drag and drop it onto `_GameMaster`. This will attach the script to the `_GameMaster` object and make it available in the scene.

The next step is to create the event call from the button. This can easily be achieved by selecting the `butStartGame` button element, and from the **Inspector** window, adding a new event call on the `OnClick()` component. Click the **(+)** button to create a new event, as shown in the following screenshot:



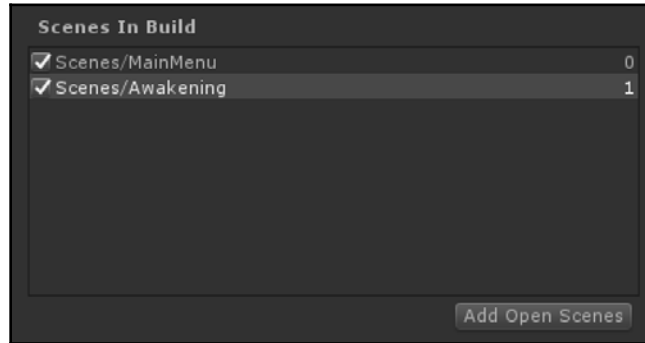
We need to call the function we created in the `GameManager.cs` script. To do so, we would need to somehow refer to it. This is done really easily. We can drag and drop the `_GameManager` GameObject into the slot, as indicated in the preceding screenshot by the number 2.

Once you place your `_GameManager` GameObject in the slot, you will need to select the script from the drop-down menu, as indicated in the screenshot by the number 3.

That's all there is to it! We have now connected our button click event to the code that will be responsible for loading our first level.

Now is a good time to save your scene and test your application. When you run the application for the first time, you will get an error. Don't be surprised, we have not done anything wrong. But there is one more step that we need to do before we can actually run our game successfully.

In order to be able to load scenes in the game, you will need to make sure they are listed in **Build Settings**. To do so, select **File | Build Settings...**, and add the current scene to the list by selecting **Add Open Scenes**. Your build settings should look as follows:



Load the **MainMenu** scene once more, and run the application. Nice! It is working as expected. The only other item I would like to add to this chapter before we move on, is the following code in the `GameMaster.cs` script:

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace com.noorcon.rpg2e
{
    public class GameMaster : MonoBehaviour
    {
        // Use this for initialization
        void Start()
        {
            DontDestroyOnLoad(this);
        }

        // Update is called once per frame
        void Update()
        {
        }

        public void StartGame()
        {
            // You should put in the name of the scene that represents
            // your
            // start level
            SceneManager.LoadScene("Awakening");
        }
    }
}
```



```
    }  
  }  
}
```

The single line of code in the `Start()` function will make sure that the `_GameMaster` `GameObject` does not get destroyed when we move from one scene to the next. This is important, because we will be storing all of our game configuration, stats, and so on in this particular `GameObject`. When you run the game now from the **Main Menu** scene, you will notice that when you load level 1, the `_GameMaster` `GameObject` comes over automatically from the main menu. This is cool!

Summary

In this chapter, we have established a good sense of the RPG. We have defined our levels and the settings for each level, and defined clear objectives for each level and the outcome for each level. We also took the first level, called *Awakening*, and created the environment. We looked at how to use our assets and the Asset Store to incorporate 3D models in our scene. We also looked at how to plan the layout of the level. We introduced a third-person character controller into the scene to help us visualize how the level looks from the player's perspective and help us fine-tune it as needed.

Here is a list of the assets that were used for the design of this level:

- For the terrain, I used the Terrain Toolkit 2017 asset
- For the models within the level, I used the Medieval Environment Pack asset

By the end of the chapter, we also developed our **Main Menu** scene and our initial `GameMaster` script that will be used to glue the core of the game together. In the next chapter, we will start creating our player character and enhancing our **Game Master** and **Main Menu** system.

3

RPG Character Design

We are now at an interesting point in our development. In this chapter, we will discuss the design of our RPG characters and look at some of the attributes and characteristics that we need to design and implement.

Here is a breakdown of the topics covered in this chapter:

- Character definitions
- Base character class attributes
- Character states
- Character model:
 - Rigging your model
 - Character motion
 - Animator controller
 - Animation states
 - Character controller
 - Modification to animations
- Inverse kinematics

Character definitions

To have a meaningful and interesting RPG, the game should usually have more than one character class. In [Chapter 1](#), *What is an RPG?* We defined the following class types:

- Barbarians
- Orcs
- Villagers

We won't be able to implement all of the character types, due to time. The demonstration of the implementation of one or two character types should give you a good foundation to develop your own character classes. After all, that is the overall objective of this book.

One of the main characters is, of course, the **Player Character (PC)**. Let's go ahead and concentrate on the implementation of the PC, and then we can start defining and designing the `Barbarian` class, the `Villager` class, and perhaps the `Orc` class.



My character models will be from the Asset Store. You may either download the same characters or design your own. You can also use different types of character models. The point is to implement the character based on the specifications, which will be defined in this chapter and beyond.

Character assets

I will be using the following assets from the Asset Store for my character models:

- Fantasy Horde Barbarians
- Fantasy Horde - Villagers
- Fantasy Horde - Orc

Let's take a look at some of the attributes that our player will have in general.

Base character class attributes

Let's start laying down the foundation we will need for the implementation of our character classes. The following is a list of attributes that will be part of the Base Character class:

- Character class name
- Character class description
- List of attributes:
 - Strength
 - Dexterity
 - Endurance
 - Intelligence
 - Social standing

- Agility
- Alertness
- Vitality
- Willpower

The attributes you define for your characters depend on the character type, but there will be some similarities between all character attributes. We would like to implement these similarities in a base class that will be shared with all character classes.



The list provided is just a sample, and you can add or subtract as you see fit.

Let's keep things simple. We will use only the five primary statistics for now, as follows:

- **Strength:** Strength is a measure of how physically solid a character is. Quality controls the most extreme weight the character can convey, skirmish assault as well as harm, and some of the time hit focuses. Protective layer and weapons may likewise have a strength prerequisite.
- **Defense:** Defense is a measure of how flexible a character is. Protection more often than not diminishes harm taken by either a rate or a settled sum for each hit.
- **Dexterity:** Dexterity is a measure of how deft a character is. Skill controls assault, development speed, and precision, and also dodging an adversary's assault.
- **Intelligence:** Intelligence is a measure of a character's critical thinking capacity. Knowledge regularly controls a character's capacity to understand remote dialects and their aptitude in enchantment. Now and again, insight controls what number of aptitude focuses the character gets at level up. In a few recreations, it controls the rate at which encounter focuses are earned, or the sum expected to level up. This is once in a while joined with shrewdness, and additionally self control.
- **Health:** Health decides whether the character is alive or dead.

The attributes listed are attributes that will be inherited by all character classes. Now let's put this into code. Create a new C# script and name it `BaseCharacter.cs`. Open the script and place the following code in the file:

```
namespace com.noorcon.rpg2e
{
    public class BaseCharacter
    {
        public string Name;
        public string Description;

        public float Strength;
        public float Defense;
        public float Dexterity;
        public float Intelligence;
        public float Health;
    }
}
```

Character states

States are an important part of the character design. They will also drive the kind of actions and movement you will need to create for each state. For instance, at a minimum our character will need to have the following states implemented:

- Idle
- Walking
- Running
- Jumping
- Attacking
- Die

Your character may have more states defined; this is something that you, as the designer of the game, will need to identify and eventually implement. Each one of the states identified will need to be implemented as an animation. The person creating the character models will usually also develop the animations for the character.

For instance, the Orc model has the following states/animations defined:



Clips	Start	End
idle	5.0	120.0
idle break 1	120.0	190.0
idle break 2	190.0	290.0
salute	290.0	340.0
walk	385.0	415.0
run	420.0	445.0
run fast	455.0	478.0
crouch	485.0	499.0
crouch walk	500.0	550.0
hit	554.0	575.0
jump	580.0	623.0
falling	625.0	665.0
die soft	668.0	740.0
die hard	745.0	770.0
idle fight	775.0	820.0
idle break fight	820.0	870.0
blocking	870.0	890.0
axe attack	890.0	910.0
sword attack	910.0	930.0
sword double attack	930.0	960.0
arch attack	960.0	1015.0
prepare spear	1015.0	1025.0
spear attack	1025.0	1045.0
casting A	1060.0	1090.0
casting B	1090.0	1115.0

You may consider implementing all of the states or some of the states. The theory and practice are the same either way.

I can also use RawMocap data to animate the models, since the models I am using are Mecanim-ready, including Face Rigs. In fact, we will use some RawMocap data to animate the models.

With the release of Unity 5, the *Mecanim Animation System* was introduced, which is used to create easy workflow and setup of animations on humanoid characters, retargeting animation from one character to the next, previewing the animation clips, managing complex interactions between animations with a visual tool, and animating different body parts with different logic.

In fact, let's go ahead and download the asset from the **Asset Store** right now. Do a search on *Raw Mocap Data for Mecanim* in the **Asset Store**.



The package contains several raw motion capture data files for your use. Be aware that you might have to make some adjustments on your own.

When creating your character models, it is a good idea to follow the proper bone structure set up for your characters. This will help make controlling the states and the animations of your character easier, as well as re-using your Animation Controller on multiple characters. This is also true if you are going to use a character from the Asset Store.

Character model

You should now consider how your player character is going to look. There are several approaches that can be taken. An easy way would be to have a predefined hero where the player does not have many options or much choice when it comes to customization of the character. The other way would be to provide the player the ability to change and modify their character to an extent or fully. This all really depends on your budget!

Here, we are going to do something in-between, to get the benefit of both worlds.



You may use the Asset Store to download predefined characters that can be used as placeholders for your game while you create your own. You can even use some of the characters that are freely available through the Asset Store and modify them for your needs.

Once you have determined your character model, the next step is to configure it and customize it for your game. The character model I have can be visually modified to represent several unique characters.

For instance, let's take a look at the default character models we will be working with, as detailed in the following sections.

Default character models

Here is a set of character models that have been purchased from the Asset Store. I like this set of models as they are simple and straightforward to work with.

Barbarian

This model contains several body types—fat, regular, and skinny—which are set up by blendshapes on the body and cloth. It comes with 15 different body and accessories textures, one weapons texture, and two shield textures. This gives us a good range of unique character definitions and customization to enhance the range of different NPCs for the RPG.

We will take a look at how to utilize them when doing character customization. See the following screenshot for an example of a barbarian:



The Barbarian model

Villager

The Villager models give us two groups of models: child and adult. In the adult group, we have the male, female, and monk types. There are 16 male, three monk, eight female, four child, and two equipment textures provided. For the female type, there are two types of texture: standard hair, used for loose hair, and hat, used with head accessories such as a hat.

There are two different sets of animations: one for the `Adult` mesh type and one for the `Child` mesh type. The child type has fewer states compared to the adult. See the following screenshot, depicting a male, female, and child villager:



The Villager models

Orc

The Orc model will be a bit different in its structure. All body-type meshes will be part of the main structure; there are three main models: Fat, Average, and Athletic. There might also be some legacy animations, which we will need to take care of when we start the configuration and programming of the `Orc` class. See the following screenshot of an orc:

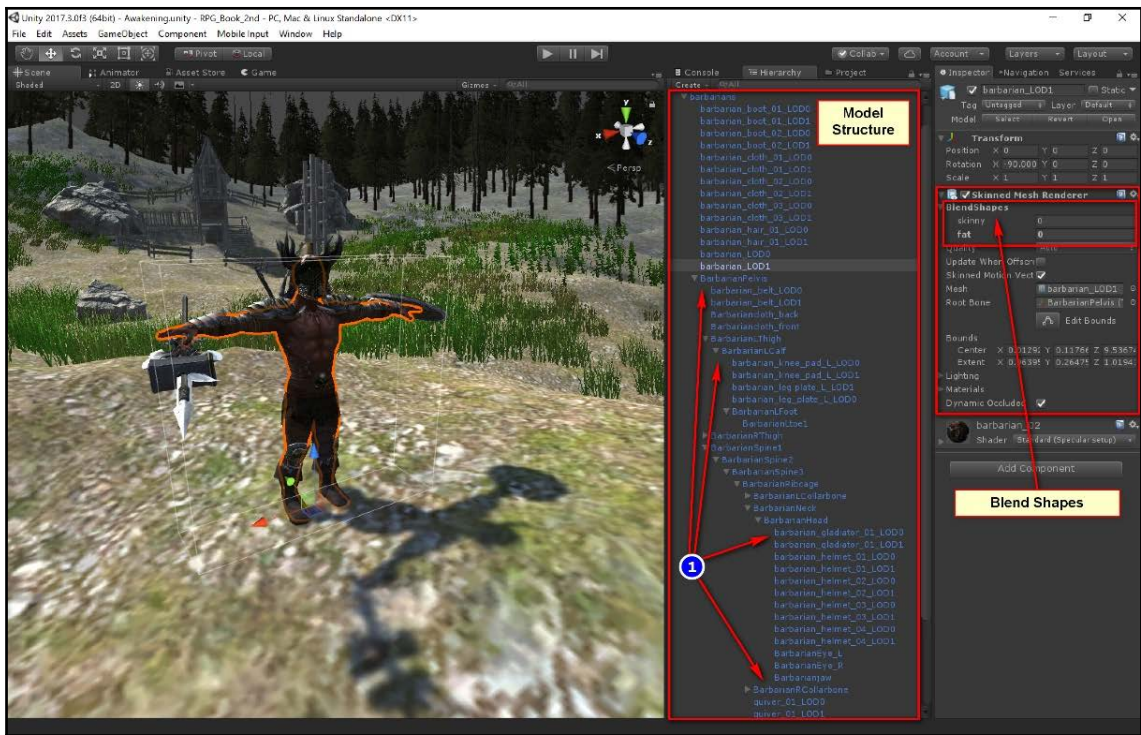


The Orc model

Let's get started

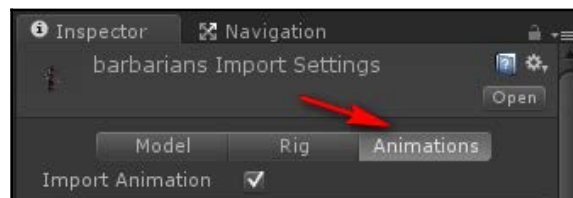
Drag and drop the Barbarian model from your project into the scene. You will need to study your character model carefully, and understand how it is built so that you can modify it during design time, as well as during runtime if necessary.

This particular model has several visual elements attached for weapons, clothing, and so on. Your model may have been configured differently; if so, you will need to create your own attachment points and instantiate the weapons and/or other character related assets accordingly. See the following screenshot:

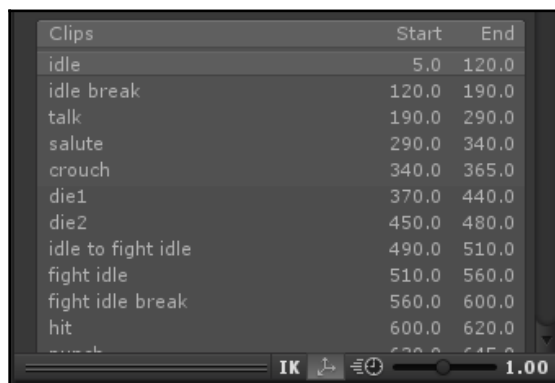


Model hierarchy

Select your character model and investigate the structure of the model. You will notice that there is a certain pattern and naming convention to the model hierarchy, as shown in the preceding screenshot. Some models might have animations attached. To check them, you will need to select the model from the **Project** window and select the **Animation** tab in the **Inspector** window to get a list of the embedded animations for the model, as shown in the following screenshot:



In the **Inspector** window, select the **Animations** tab shown in the preceding screenshot, and notice the **Clips** section for all animations developed for your character model, as indicated in the following screenshot:

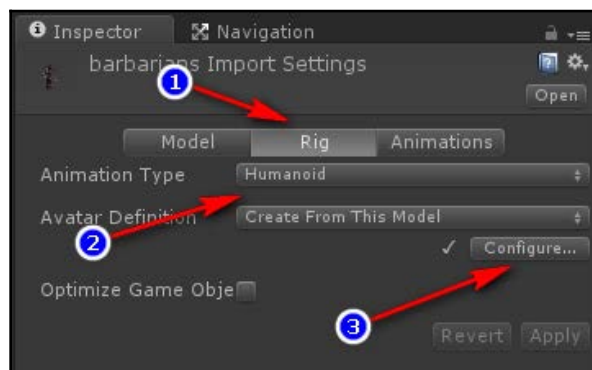


Clips	Start	End
idle	5.0	120.0
idle break	120.0	190.0
talk	190.0	290.0
salute	290.0	340.0
crouch	340.0	365.0
die1	370.0	440.0
die2	450.0	480.0
idle to fight idle	490.0	510.0
fight idle	510.0	560.0
fight idle break	560.0	600.0
hit	600.0	620.0
death	620.0	645.0

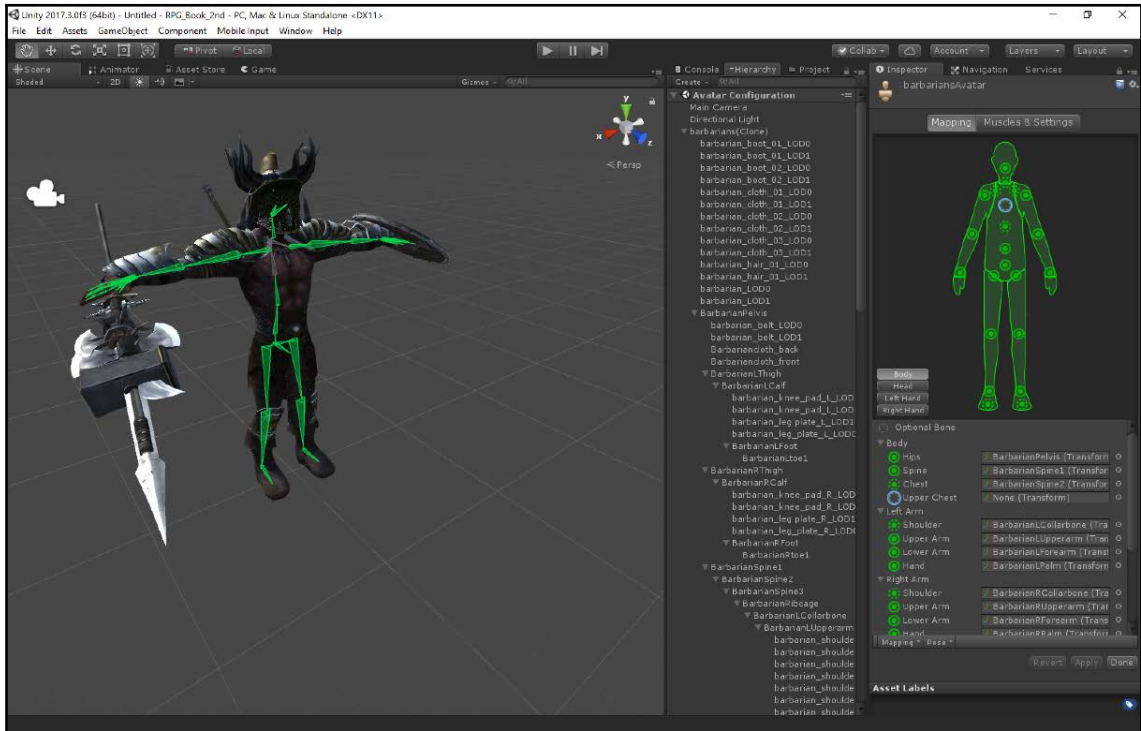
Notice that the animation clips have a start time and an end time. The actual character model is visually displayed at the bottom of the **Inspector** window.

Rigging your model

There might be times that you will need to **Rig** your model to make it suitable for your game. This can be achieved by selecting your model source, and from the **Inspector** window selecting the **Rig** tab, as shown in the following screenshot:



In the **Rig** tab, there are several options that you can apply to your model. Assuming that your character is of **Humanoid** type, you will need to select the **Humanoid Animation Type** if not already selected. The **Avatar Definition** can also be either created from the model or assigned if you have an avatar defined. Finally, you can click on the **Configure...** button to see the configuration of the rigged model. See the following screenshot:



Humanoid bone structure

Notice from the preceding screenshot that your model has a mapping defined for its skeleton. If your model is of **Humanoid** type, and if your model structure has been named properly, the system will automatically assign the correct bones and joints. If your naming is not per Unity specification, you can navigate your model structure and manually assign each point in the **Body**, **Head**, **Left Hand**, and **Right Hand**.

The **Muscles** and **Settings** tabs will enable you to define and restrict the movement of the joints for your model. These can be very useful and practical for creating more realistic movements for your characters. You can study these topics further on your own, as they would require a whole chapter or two to cover them.

Character motion

Traditionally, the motion and movement of the characters were done separately through code. With the introduction of *Mecanim*, you are now able to apply what is called *Root Motion*. This modifies the character's in-game transform based on the data in root motion.

We are going to use root motion for our characters. Root motion works with the **Animator Controller** and the Animation State Machine. The Body Transform and Orientation are stored in the Animation Clip. This makes it easier to create a state machine that plays the appropriate animation clip through the **Animator Controller**.

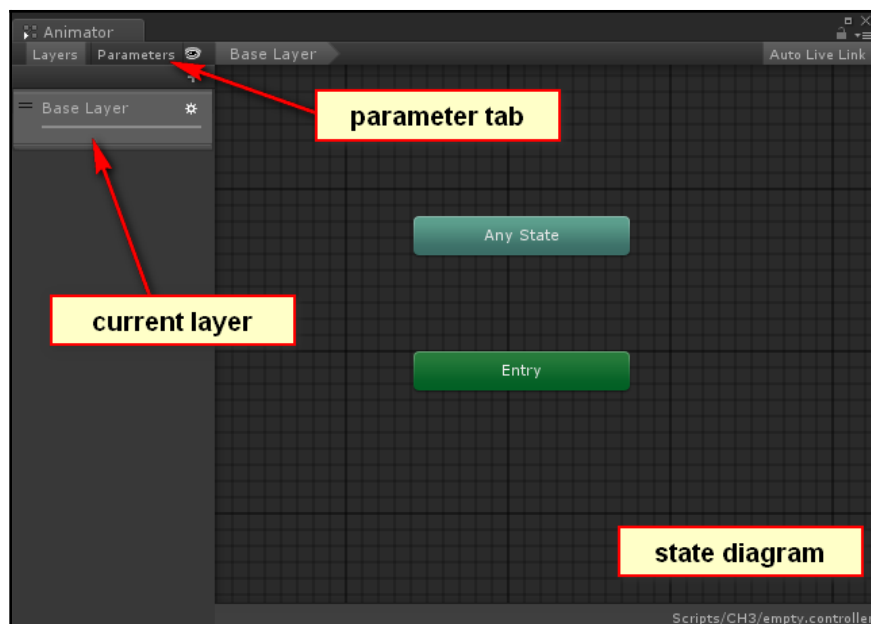
Animator Controller

In this section, we will use the new **Animator Controller** to create our character states and determine the criteria for a change of states. Let's make a new folder in the **Project** window and name it `Animator`. Select the newly created folder.

To create an **Animator Controller**, in the **Project** window, right-click and select **Create | Animator Controller**. Give it a name. I have called mine `BaseAnimatorController`. Double-click the controller to open the **Animator** window.

The **Animator Controller** is a very complex tool, and it will take you some time to study the different aspects and features that are available to you through it. The following screenshot is a snapshot of an empty controller. I have marked the main sections of the **Animator** window. There are two visible tabs, the **Layers** tab and the **Parameters** tab. In the **Layers** tab you will be able to create different layers that hold your animation states and the relevant **Transitions** from one state to the next. The **Parameters** tab is where you define your parameters that will be accessed and modified by the **Animator Controller**, as well as through your code.

There is a wide range of topics that you will need to know to fully appreciate the Mecanim system. We won't be going through all of the aspects in this book, but we will touch on some of the key aspects that are needed for our game. See the following screenshot:



Animation states

To create a new state, you can simply drag and drop an animation from your **Project** window. This will name and assign the relevant animation to the state in the layer. You can also create an empty state by right-clicking in the layer and selecting **Create State | Empty**. When a state is created, you can click on the state and observe its properties in the **Inspector** window, as shown in the following screenshot:

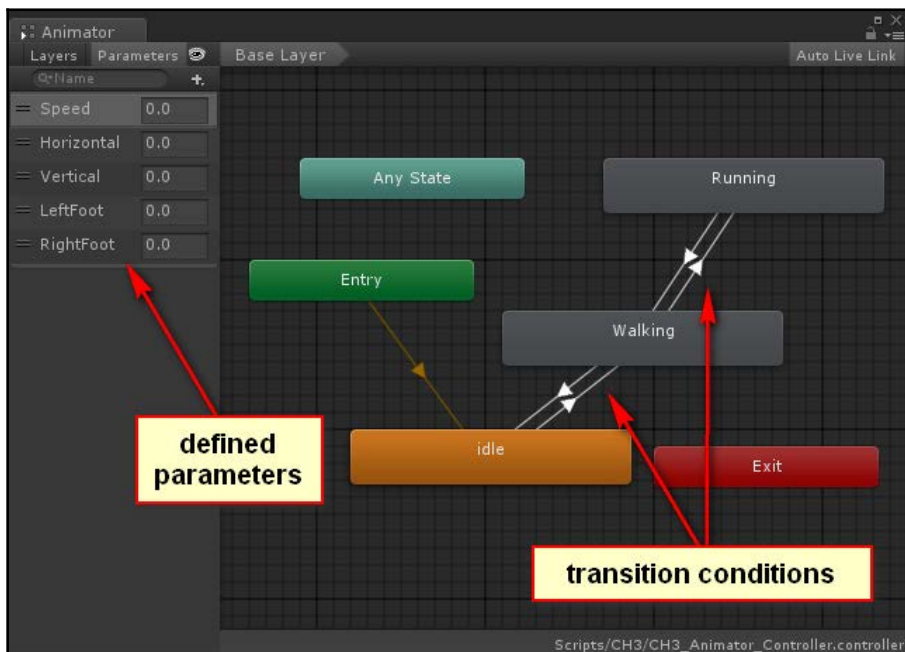




Your model may or may not have animations attached to it. The whole idea of the Mecanim system is to enable character modelers to work on their models, while animators can use the skeleton of a humanoid avatar to animate the character. This in turn makes it easier and better to have a set of animations applied to different types of character models!

To identify the state, it is best to provide it with a unique name that can be easily recognized in the state diagram. You will need to assign a Motion to it; this is the animation clip that will be playing when the state is active. The next important property would be the **Transitions** property. A transition will determine the condition for which state will be moving to another state, if there is such a requirement.

For instance, when the character is in an idle state, what condition is for the character to change its state to a walking state, to a running state and so forth, as shown in the following screenshot:



In the preceding screenshot, you will see I have defined three different states: **Idle**, **Walking**, and **Running**. You will also notice that, in the **Parameters** tab, I have defined some parameters. These parameters are used to determine when to move from idle to walking, to running, and back. The parameters are there to help you create the conditions for your state machine.

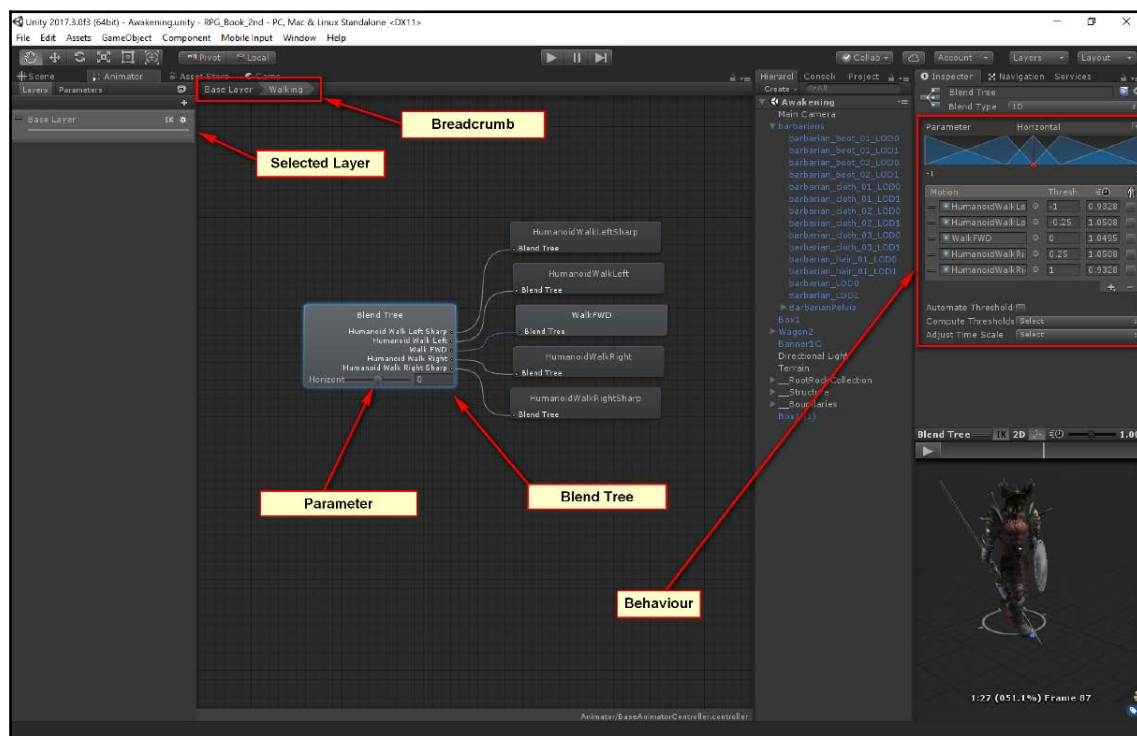
To create a **Transition** from one state to the next, right-click your state, and select **Make Transition**, then select the state it will transition to. This will create the visual arrow from the start state to the end state. Select the *Transition* arrow to get its properties, and set the conditions in the **Inspector** window.

The **Walking** and **Running** states are actually a Blend Tree in this instance. A Blend Tree is used to make the transition from one animation state to the next more natural. In order for the blended motion to make sense, the motions that are blended must be of similar nature and timing.



Blend Trees are used to allow multiple animations to be blended smoothly by incorporating parts of them all to a varying degree. The amount that each of the motions contributes to the final effect is controlled using a blending parameter, which is just one of the numeric animation parameters associates with the **Animator Controller**.

For instance, the walking state could look something like the following screenshot:



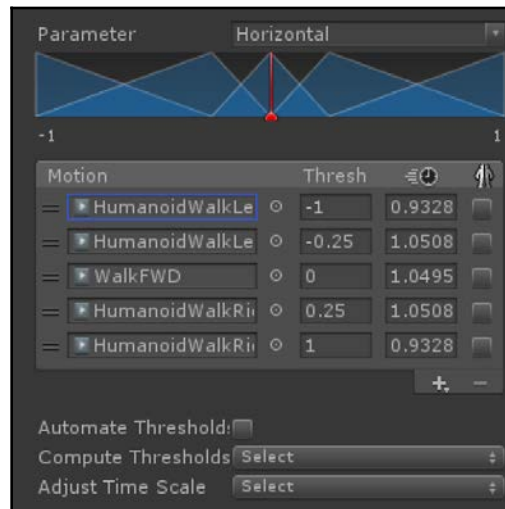
Blend Tree example

In our first **Blend Tree** node, we have five outputs: HumanoidWalkLeftSharp, HumanoidWalkLeft, WalkFWD, HumanoidWalkRight, and HumanoidWalkRightSharp. These are the animation clips that will be playing based on the value of the parameter called **Horizontal**.



These animations are from *Raw Mocap Data for Mecanim*.

In the behavior region, you will notice a few thresholds that have been set up for the parameter; these thresholds are what determines the animation to be played. The value of the **Horizontal** parameter is set through our C# code by passing in the value of the **Horizontal Axis**, which is defined in the **Input Manager**. See the following screenshot:

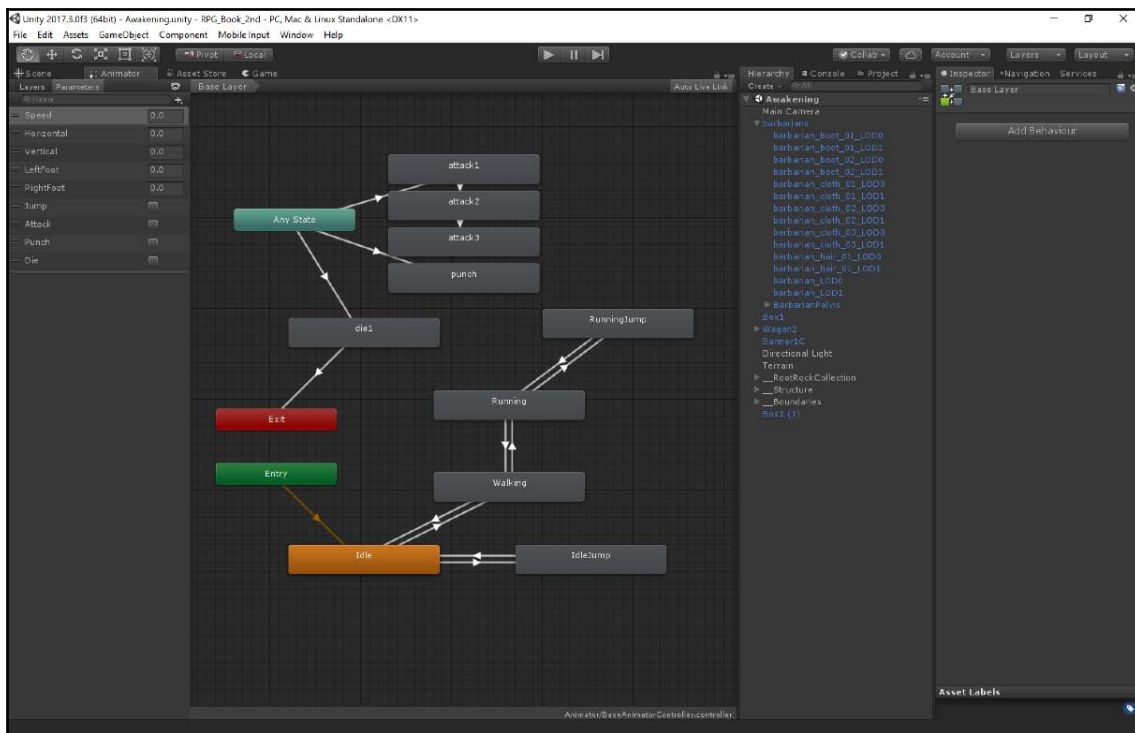


When you select a *Blend Tree* node, your **Inspector** window will give you the ability to add or remove the different animation states, as well as the parameter and the threshold of the parameter that will determine which animation will be rendered.



The key to have a smooth looking blending in your animation is to pay attention to your animation data.

Let's take a look at our final state diagram:



State diagram

At this stage, I have gone ahead and implemented the state diagram for **Idle, Walking, Running, Idle Jump, Running Jump, Attack1/2/3, Punch, and Die**.

The parameter that defines the transition from the **Idle** state to the **Walking** and **Running** states is the **Speed** parameter. If the **Speed** value is greater than 0.1, it will transition from **Idle** to **Walking**; if it is greater than 0.6, it will transition from **Walking** to **Running**. The opposite is true for going from **Running** to **Walking**, and from **Walking** to **Running**.

Notice, however, that the character can only enter the **Jump** state from either the **Idle** state or from the **Running** state. The parameter that controls this transition is the `Jump` parameter, which is a Boolean value set by pressing the Space bar button on the keyboard.

There are also three attack states that can be triggered from the **Any** state, as well as a **Die** state that can be entered from the **Any** state. Well, this is because your character can die at any given time if you are not careful!

Let's take a look at how we can control these parameters.

Character controller

It is time to enable our character to move around the scene. This is generally handled by the character controller. The character controller will be used to handle most of the interactions the player will have with the character in the game.

Create a new C# script and call it `BarbarianCharacterController.cs`. Enter the following code in the `BarbarianCharacterController` class. At the moment the code is very basic. Let's get a listing of the code and we can start discussing the different parts of the code after the listing:

```
using UnityEngine;

namespace com.noorcon.rpg2e
{
    public class BarbarianCharacterController : MonoBehaviour
    {
        public Animator animator;
        public float directionDampTime;

        public float speed = 6.0f;
        public float h = 0.0f;
        public float v = 0.0f;

        bool attack = false;
        bool punch = false;
        bool run = false;

        bool jump = false;
        bool die = false;
        bool dead = false;

        // Use this for initialization
        void Start()
        {
            this.animator = GetComponent<Animator>() as Animator;
        }
    }
}
```

```
// Update is called once per frame
private Vector3 moveDirection = Vector3.zero;

void Update()
{
    if (dead)
    {
        if (die)
        {
            die = !die;
            animator.SetBool("Die", die);
        }
        return;
    }

    if (Input.GetKeyDown(KeyCode.C))
    {
        attack = true;
    }
    if (Input.GetKeyUp(KeyCode.C))
    {
        attack = false;
    }
    animator.SetBool("Attack", attack);

    if (Input.GetKeyDown(KeyCode.P))
    {
        punch = true;
    }
    if (Input.GetKeyUp(KeyCode.P))
    {
        punch = false;
    }
    animator.SetBool("Punch", punch);

    if (Input.GetKeyDown(KeyCode.LeftShift))
    {
        this.run = true;
    }
    if (Input.GetKeyUp(KeyCode.LeftShift))
    {
        this.run = false;
    }
    animator.SetBool("Run", run);

    if (Input.GetKeyDown(KeyCode.Space))
    {

```

```
        jump = true;
    }
    if (Input.GetKeyUp(KeyCode.Space))
    {
        jump = false;
    }
    animator.SetBool("Jump", jump);

    if (Input.GetKeyDown(KeyCode.I))
    {
        die = true;
        dead = true;
    }

    animator.SetBool("Die", die);
}
```

In the `Start()` function, we will get a reference to the **Animator Controller**. We will be using the `FixedUpdate()` function to perform our updates for the character movement, as shown in the following code:

```
void FixedUpdate()
{
    // The Inputs are defined in the Input Manager
    // get value for horizontal axis
    h = Input.GetAxis("Horizontal");
    // get value for vertical axis
    v = Input.GetAxis("Vertical");

    speed = new Vector2(h, v).sqrMagnitude;

    // Used to get values on console
    Debug.Log(string.Format("H:{0} - V:{1} - Speed:{2}", h, v,
    speed));

    animator.SetFloat("Speed", speed);
    animator.SetFloat("Horizontal", h);
    animator.SetFloat("Vertical", v);
}
}
```

What is the difference between the `Update()` function and the `FixedUpdate()`? The `Update()` function is called every frame, and is used regularly to update the moving of non-physics objects, simple timers, and input processing. The update interval time varies for the `Update()` function. `FixedUpdate()` is called every physics step. The interval is consistent and used for adjusting physics on `Rigidbody`.

In the `FixedUpdate()` function, we get the inputs for our Horizontal and Vertical axes, we calculate the *speed* value, and set the parameters defined in the **Animator Controller** using the `animator.SetFloat()` function. These parameters are then used by the animator controller to decide which state the character is at.

For instance, to go from an idle state to the walking state, the *speed* parameter needs to be greater than 0.1, and from walking to running, the *speed* parameter will need to be greater than 0.6, and the *run* parameter needs to be true. The opposite is true when you want to go back from the running state to the walking state and from the walking state to the idle state. The Horizontal and Vertical parameters control the movement for turning left or turning right. All these three parameters combined control what state and what animation the character is rendering.

The next step is for us to enable the **Jump**, **Die** and **Attack** states. The **Jump** state can be entered while the character is idle or running and the Jump Boolean variable is set to true. The jump condition is set in the `Update()` function when the space bar is pressed by the player. This sets the variable to true and passes the variable to the animator controller.

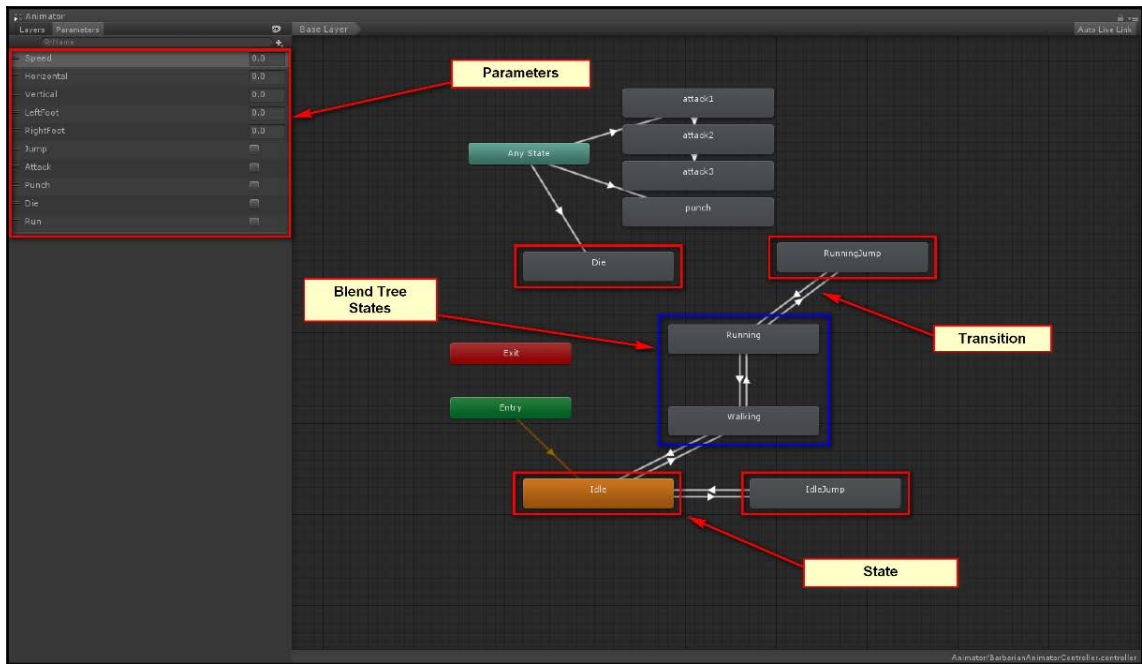
The same mechanism is used for the attack and punch state. This is mapped to the following keys on the keyboard: *C* and *P*. Each one will set its Boolean value to true and pass it into the animator controller. The player can only enter these states from the **Any** state. We will leave it as is for now.

Finally, the **Die** state is implemented, and for now we are using the keyboard input *I* to test it out. The main difference between the **Die** state and the other states so far is that the **Die** state can be entered from any state.



We are not using Blend Trees for these states as there is only one type of animation for the state. You will also notice that the states can only be transitioned to from the **Idle** state. This is due to how the animations and model were set up initially. Yours could be different.

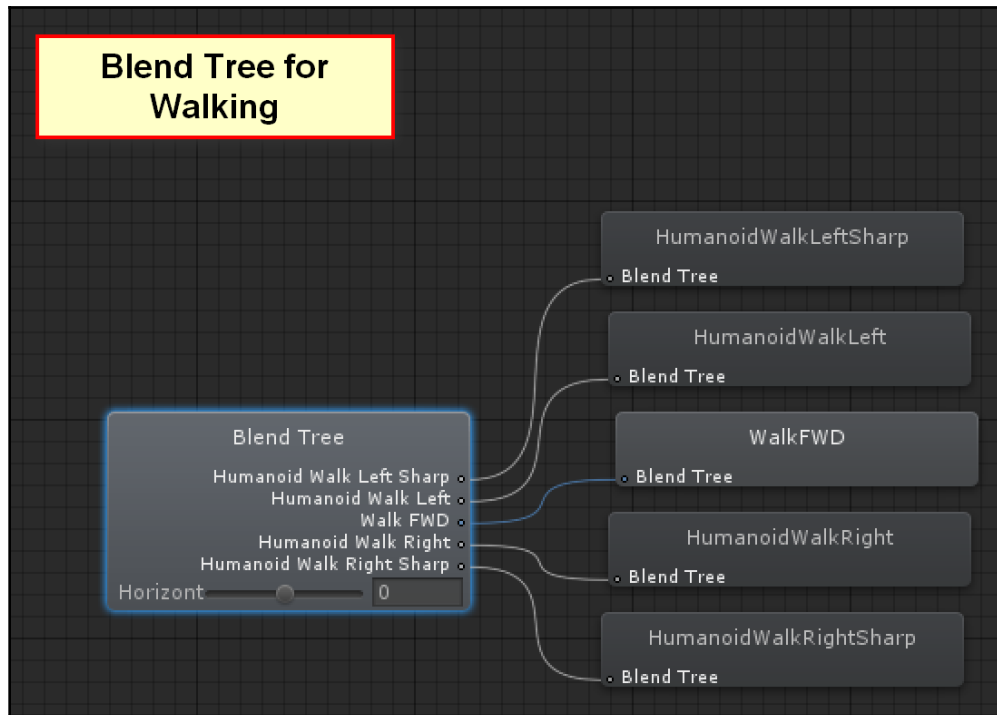
Take a look at the following screenshot:



Animator parameters

The character can get into the **Die** and **Attack** states from any state. That is, your character player can die at any time in the game, whatever state he or she is in at the time. The jump states can be triggered from two states, idle and the running. You can improve these transitions and states based on the level of your animation complexity, but for now, this should do.

Take a look at the following screenshot:

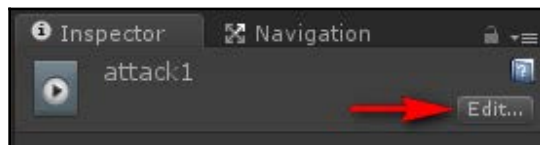


These states are controlled through Boolean parameters defined in the animator. At this stage, you should be able to use your model to test the scene, as well as your character animations and states.

Modification to animations

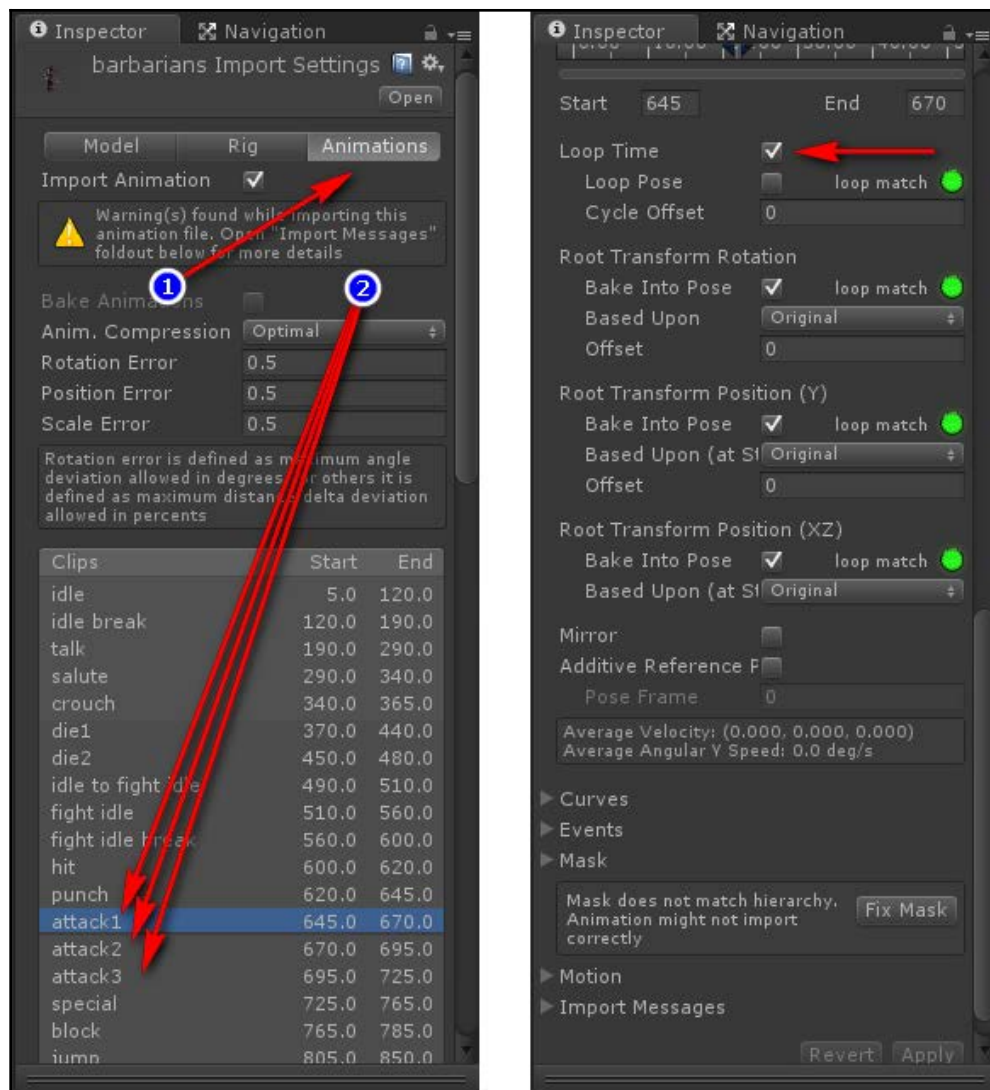
There might be times that you will need to make some changes and/or modifications to the existing modification that will make it work properly with your game and the state machine.

The attack animations prepared for my character model need to be adjusted to make them loop while the character is still in that particular state. For instance, if I use the existing animation and the character state goes into attack mode, the animation will play only once. This is not what I intend to do; I am building the attack input to perform the attack while the attack key is pressed down. Changing the animation loop setting is easy. To do so, select the animation from your **Project** window and select the **Edit...** button from the **Inspector** window, as shown in the following screenshot:



You will now be in the **Edit** mode of the animation, as displayed in the following screenshot. I have placed the **Inspector** window next to it, to illustrate the **Animation** tab, selecting each animation we want to modify, one at a time, and setting the **Loop Time** property to **True**.

In this particular screenshot, you will also notice several other important properties for the animation, such as **Root Transform Rotation**, **Mirror**, **Curves**, **Events**, **Mask**, and **Motion**. We will use the **Curves** property when we set out Inverse Kinematics for some of our animations regarding our character. This basically sets the values of predefined parameters that can be used to set or get them through *Mecanim*. Take a look at the following screenshot:



If your animations are attached to your model and your animations and models are older, you will most likely need to make some modifications to them.

For instance, one of the main properties that you might have to set for a particular animation clip would be the **Loop Time** property, as shown in the preceding screenshot. This will make sure that the animation will loop as long as you are in the state which is running the animation. If looping is not enabled, the animation will run once and stop, even if you are still in the state representing the animation.

Make sure the **Loop Time** property is set for the idle, walking, running and attacking animations. At the same time, not all animation clips need to be looped, for instance, the jump and die animations just need to be played once. You will need to do your diligence and check all of these properties.

Other animation will need to be modified to enable baking the transform into the model. For instance, the die and jump animations have the following properties checked: **Root Transform Rotation** and **Root Transform Position (Y)**; make sure that the **Bake Into Pose** property is checked. This is important, to make sure the animation and the skeletal movement of the character are harmonized at the root transform position.



Your animation might seem funky if these properties are not set properly. If there is something weird going on, make sure to double-check these properties.

If you have not done so by now, you should attach your `BarbarianharacterController.cs` script to your player character.

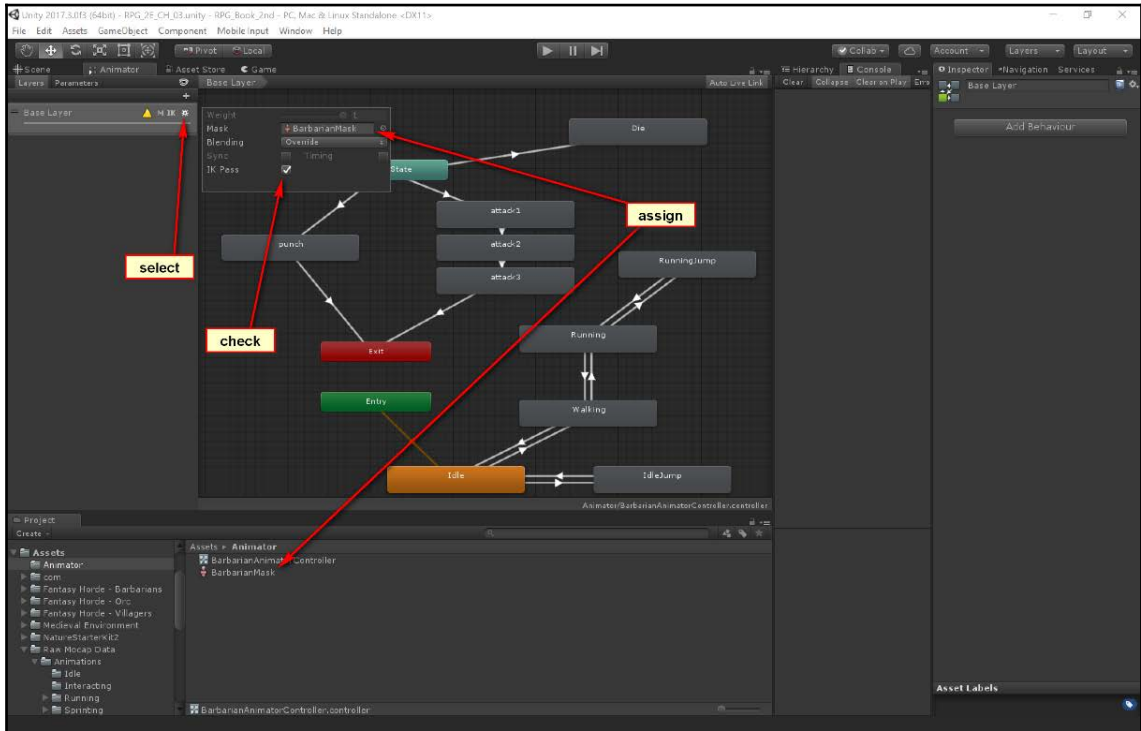
Inverse Kinematics

Inverse Kinematics (IK) is important in game programming. It is typically used to make the character's movement more realistic. One of the main uses of IK is the calculation of the player's feet and how they relate to the ground they are standing on.

In short, IK is used to determine the position and rotation of the joints of a character based on a given position in space. For instance, to make sure the foot of a player lands properly on the terrain it is walking on.

Unity has a built-in IK system that can be used to do some basic calculations in this regard. Let's go ahead and implement the foot IK for our character. There are a few things that you will need to set up before we can enable IK for our humanoid character.

The first thing to do is check your layer in the **Animator Controller** and use the **Engine** icon to enter the **Settings** window. Make sure that **IK Pass** is checked, as shown in the following screenshot. You will also need to provide a Mask if you have not done so already. The mask is used to dictate which parts of the skeleton are affected by the IK. Take a look at the following screenshot:



Mask for Inverse Kinematics

Once you have set this up, the fun begins. We need to create a C# script that will handle our IK. Create a C# script and call it `IKHandle.cs`. Type the following code into the script:

```
using UnityEngine;

namespace com.noorcon.rpg2e
{
    public class IKHandle : MonoBehaviour
    {
        Animator anim;

        #region USED FOR MANUAL TESTING
    }
}
```

```
Transform leftIKTarget;
Transform rightIKTarget;

Transform hintLeft;
Transform hintRight;

float ikWeight = 1f;
#endregion

// to make it dynamic
[Header("Dynamic IK Values")]
Vector3 LeftFootPosition;
Vector3 RightFootPosition;

Quaternion LeftFootRotation;
Quaternion RightFootRotation;

float LeftFootWeight;
float RightFootWeight;

public Transform LeftFoot;
public Transform RightFoot;

[Header("Adjustment Properties for IK")]
public float OffsetY;

public float LookIkWeight = 1.0f;
public float BodyWeight = 0.7f;
public float HeadWeight = 0.9f;
public float EyesWeight = 1.0f;
public float ClampWeight = 1.0f;

public Transform LookPosition;

// Use this for initialization
void Start()
{
    anim = GetComponent<Animator>();

    LeftFoot = anim.GetBoneTransform(HumanBodyBones.LeftFoot);
    RightFoot = anim.GetBoneTransform(HumanBodyBones.RightFoot);

    LeftFootRotation = LeftFoot.rotation;
    RightFootRotation = RightFoot.rotation;
}
```

This script is a bit involved. In order for the Inverse Kinematic, that is, IK, to work properly, we need to identify a couple of important points in space. One of these points is the position of the target in space that we want our foot to move to, and the second point in space is the hint. These two points in space are used to control the movement and translations of the skeleton for a particular joint to be made, in order to successfully complete the IK for the target position. Take a look at the following code:

```
// Update is called once per frame
void Update()
{
    // we can set the look position here somewhere
    Ray ray = new Ray(Camera.main.transform.position,
        Camera.main.transform.forward);

    Debug.DrawRay(Camera.main.transform.position,
        Camera.main.transform.forward * 15, Color.cyan);

    //lookPosition.position = ray.GetPoint(15);

    RaycastHit leftHit;
    RaycastHit rightHit;

    Vector3 lpos = LeftFoot.TransformPoint(Vector3.zero);
    Vector3 rpos = RightFoot.TransformPoint(Vector3.zero);

    if (Physics.Raycast(lpos, -Vector3.up, out leftHit, 1))
    {
        LeftFootPosition = leftHit.point;
        LeftFootRotation = Quaternion.FromToRotation(transform.up,
            leftHit.normal) * transform.rotation;
    }

    if (Physics.Raycast(rpos, -Vector3.up, out rightHit, 1))
    {
        RightFootPosition = rightHit.point;
        RightFootRotation = Quaternion.FromToRotation(transform.up,
            rightHit.normal) * transform.rotation;
    }
}

void OnAnimatorIK()
{
    LeftFootWeight = anim.GetFloat("MyLeftFoot");
    RightFootWeight = anim.GetFloat("MyRightFoot");

    anim.SetIKPositionWeight(AvatarIKGoal.LeftFoot,
        LeftFootWeight);
}
```



```
anim.SetIKPositionWeight (AvatarIKGoal.RightFoot,
RightFootWeight);
anim.SetIKPosition (AvatarIKGoal.LeftFoot, LeftFootPosition +
new Vector3(0f, OffsetY, 0f));
anim.SetIKPosition (AvatarIKGoal.RightFoot, RightFootPosition +
new Vector3(0f, OffsetY, 0f));

anim.SetIKRotationWeight (AvatarIKGoal.LeftFoot,
LeftFootWeight);
anim.SetIKRotationWeight (AvatarIKGoal.RightFoot,
RightFootWeight);
anim.SetIKRotation (AvatarIKGoal.LeftFoot, LeftFootRotation);
anim.SetIKRotation (AvatarIKGoal.RightFoot, RightFootRotation);

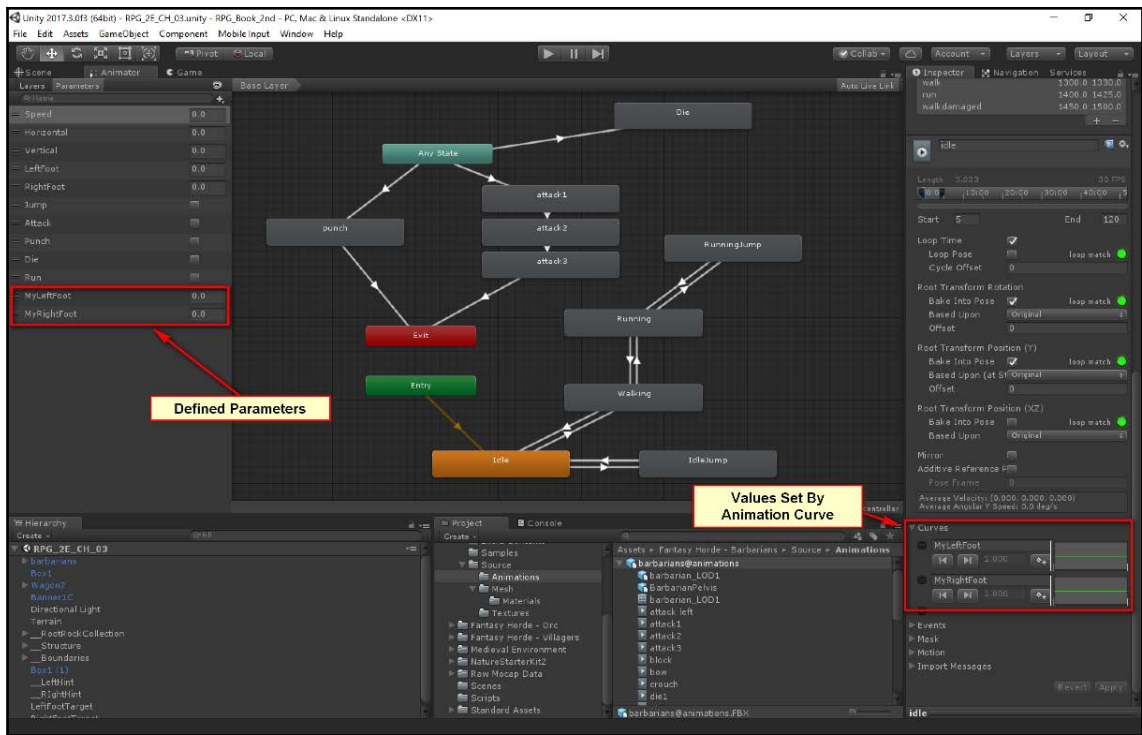
    }
}
}
```

The `LeftFootPosition` and `RightFootPosition` variables are used to represent the target position for the left and right foot during runtime. `LeftFootRotation` and `RightFootRotation` are used to store the rotation of the left and right foot.

We also need two variables to actually reference our left and right foot in the model. This is done by the `LeftFoot` and `RightFoot` variables.

Some of these variables are initialized in the `Start()` function. Specifically, we get a reference to the left and right foot from the **Animator Controller** bone structure defined for humanoids.

In the `Update()` function, we use `Physics.Raycast()` to perform some raycasting, to determine the position of our left and right foot. This data is then used and stored in the `LeftFootPosition` and `RightFootPosition` variables, with their equivalent rotation data in the `LeftFootRotation` and `RightFootRotation` variables. Take a look at the following screenshot:



Animation curves

The actual IK animation is applied in the `OnAnimatorIK()` function. The `LeftFootWeight` and `RightFootWeight` variables are used to get the parameter values set for `MyLeftFoot` and `MyRightFoot` in the **Animator Controller**, through the animation clip Curve function.



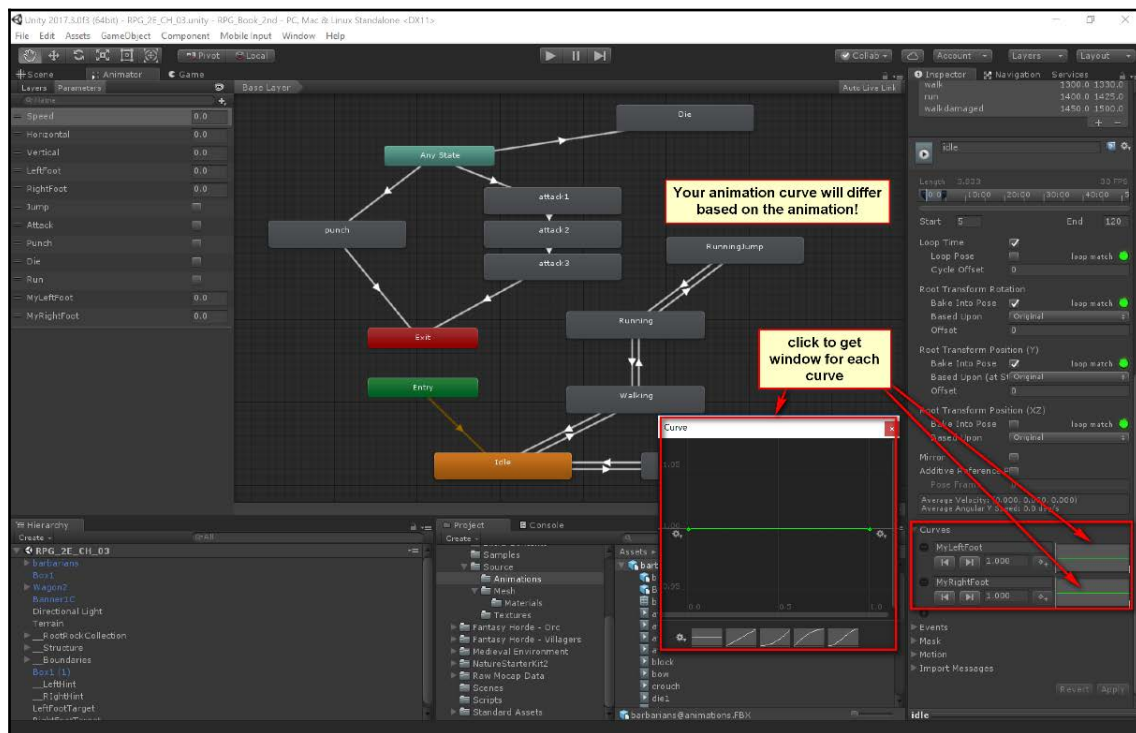
The key here is to properly define the curve of the animation clip that will be used to drive the weight of the IK. The preceding screenshot only shows the curve of the idle state. Both feet are on the ground, therefore the value is set to 1. For your walking and running clips, your curve will be different.

Finally, the `SetIKPositionWeight()` and `SetIKPosition()` functions are used to properly adjust the position and rotation of the feet relative to the ground! Notice that this is performed for each foot separately.

Attach the `IKHandle.cs` script to your character and do a test run. Notice the difference in your character and the way it is interacting with the floor or the terrain you have set up.

Setting the animation curve

This step is very important for the IK to work. I am going to use the **Idle** animation to demonstrate what needs to be configured, to make sure the parameters in the animator controller are set properly. Take a look at the following screenshot:



Animation curve modification

For the IK to work properly and look nice, you will need to set the curve for each animation that has to do with the motion of the feet. Since we have five sets of animations for walking and running, you will need to perform the same for each animation curve to properly set the weight value that will be passed to the IK script.

Summary

We covered a lot of content in this chapter. We discussed the different character definitions we will be using for our game, looked at the base character class attributes that will be shared by all of our characters, and created the `BaseCharacter` class to be used later in the game. We also discussed the primary states our character will have in the game, and how to implement them using the **Animator Controller**.

We looked at how to rig our character model to be prepared for the Mecanim system, as well as how to use the Mecanim system to create animation and state diagrams that will determine how the character behaves during game play. Then we implemented our initial character controller script that handles the state of our character. This gave us the opportunity to look at the Blend Trees and transition from one state to the next using parameters. We then looked at how to modify animation clips if there is a need for it.

Finally, we learned about Inverse Kinematics, which will help our character to behave more realistically in the game environment.

Having reached the end of the chapter, you should have a good grasp of all of the different components that work together to make your character look, behave, and move in the game environment.

In the next chapter, we will be introducing non-character behaviors.

4

The Game Mechanics

In [Chapter 3](#), *RPG Character Design*, we covered a wide range of topics to prepare your character model for the game. We looked at how to import and set up our character model, created the `BaseCharacter` class, used the Animator Controller to set up the state diagram, created the initial character controller to handle the motion and behavior of our character model, and finally, looked at some basic inverse kinematics for the foot.

In this chapter, we will expand on the character player and also the non-character player, covering the following topics:

- Customizing the player character:
 - Customizable parts (model)
 - C# code for customization
 - Preserving character state
 - Recap
- Non-player characters:
 - Non-player character basics
 - Setting up the non-player character
 - Navmesh setup
 - NPC Animator Controller
 - NPC Attack
 - NPC AI
- PC and NPC interaction

Customizing the player character

One of the key features of an RPG is to be able to customize your character player. In this section, we will take a look at how we can provide a means to achieve this.



Once again, the approach and concept are universal, but the actual implementation might be a little different based on your model structure.

Create a new scene and name it `CharacterCustomization`. Create a Cube prefab and set it to the origin. Change the Scale of the cube to `<5, 0.1, 5>`. You can also change the name of the `GameObject` to `Base`. This will be the platform that our character model stands on while the player customizes his/her character before game play.

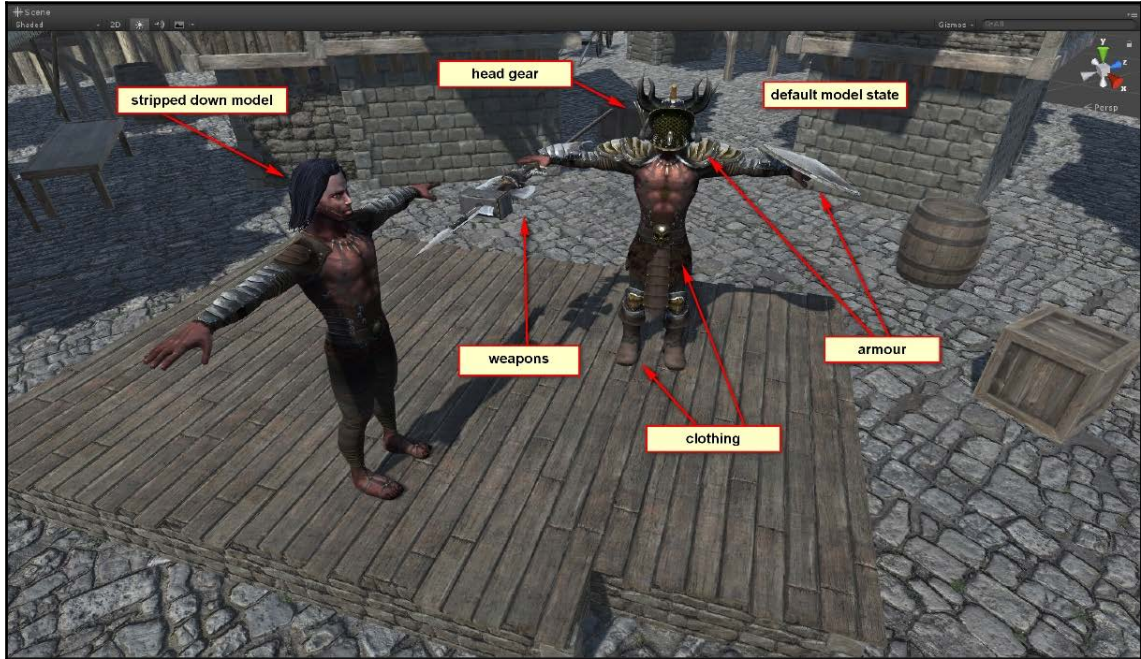


I used my environment assets to create the stage. This takes more time, but it is more attractive. This is totally up to you, the creator and designer of the game, and the sky is the limit!

Drag and the drop the prefab representing your character model into the Scene View. The next few steps will entirely depend on your model hierarchy and structure as designed.

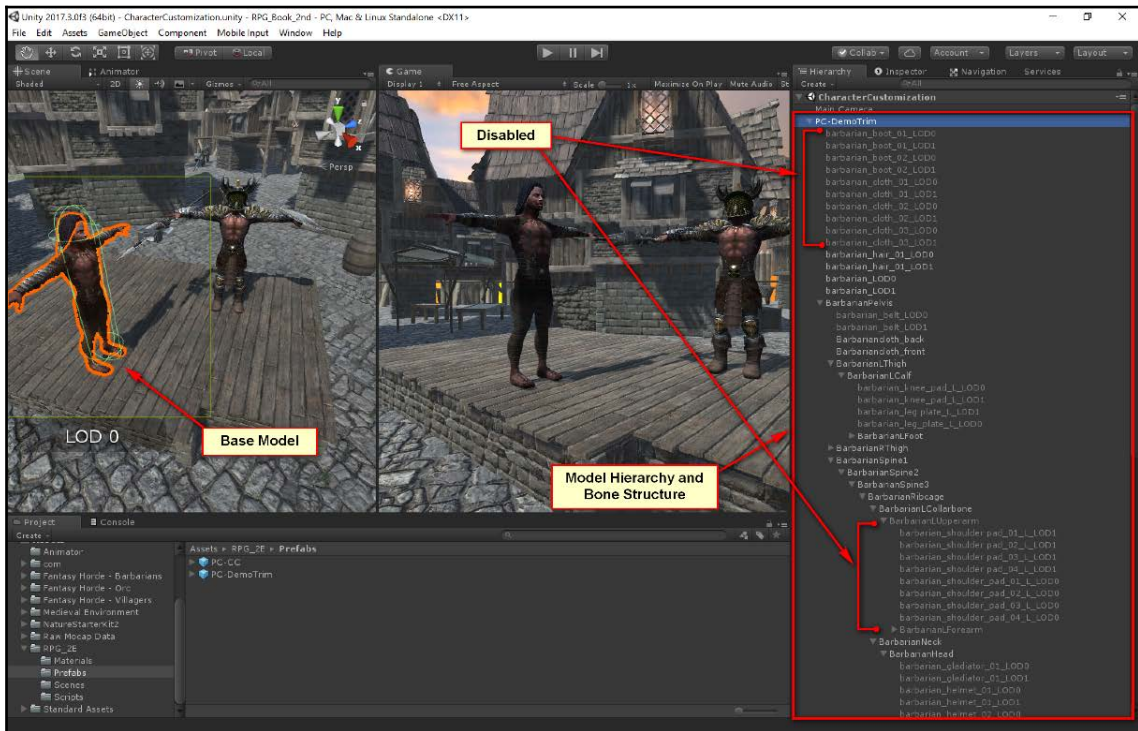
I am using the Barbarian model to illustrate the structure.

To illustrate the point, I have placed the same model in the scene twice. The one on the left is the model that has been configured to display only the basics, and the model on the right is the model in its original state, as shown in the following screenshot:



Barbarian model: simple and fully geared

Notice that the particular model I am using has everything attached. This includes the different types of weapons, shoes, helmets, armor, and skins. The instantiated prefab on the left-hand side has turned off all of the extras from the model's hierarchy. Here is how the hierarchy looks in Hierarchy View:



Barbarian model structure

The model has a very extensive hierarchy in its structure. The preceding screenshot is a small snippet to demonstrate that you will need to navigate the structure and manually identify and enable or disable the mesh representing a particular part of the model.

Model root:

Pelvis:

Left thigh:

- Left calf

Right thigh:

- Right calf

Spine:

- Ribcage:
 - Left collarbone:
 - Left upper arm:
 - Left forearm
 - Neck:
 - Head
 - Right collarbone:
 - Right upper arm:
 - Right forearm:
 - Right palm

Each one of your character models will have their own unique hierarchy and bone structure. You will need to study this, as mentioned previously, to understand and plan how you will configure and program them during game play.

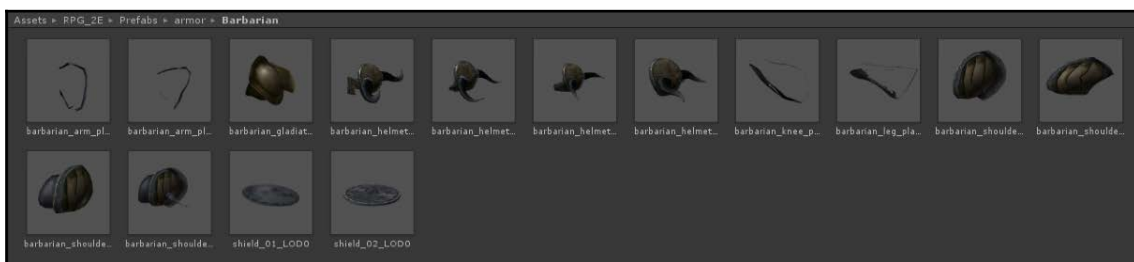
Customizable parts

Using my Barbarian model, I can customize a few items with it. I can customize the shoulder pads, the body type, the weapons, the armor, the helmets, the shoes, and, finally, the texture or skin of the model to give it different and unique looks.

Let's get a list of all the different customizable items we have for our character:

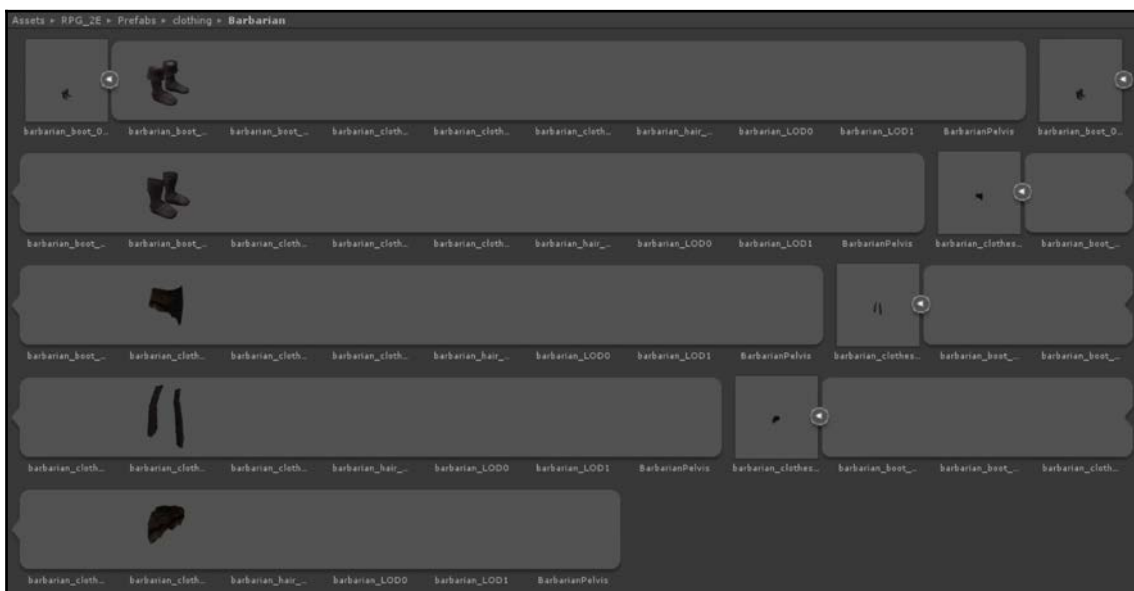
- **Shields:** There are two types
- **Body Type:** There are three body types: skinny, buff, and chubby

- **Armor:** knee pad, leg plate
- **Boots:** There are two types of boots
- **Helmet:** There are four types of helmets
- **Weapons:** There are 13 different types of weapons
- **Skins:** There are 13 different types of skins



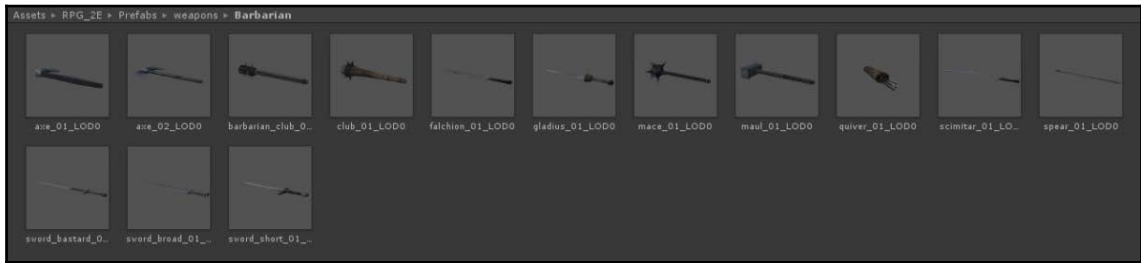
Model assets 1

You can easily extract each accessory from the main model and create a prefab of the individual weapon, armor, and clothing:



Model assets 2

We are separating the items in such a way to give the player the ability to level-up or find the required weapon or armor during game play:



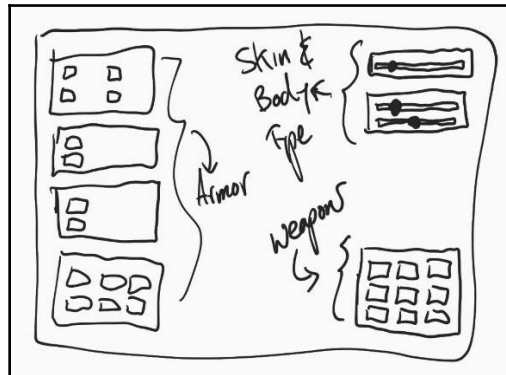
Model assets 3

Once an item is picked up, we would have it in our inventory and the player can easily access it.

User interface

Now that we know what our options are for customizing our player character, we can start thinking about the **User Interface (UI)**. The UI will be used for the customization of the character.

The following is a sketch of an idea for the UI. As we start implementing the UI, we might have to make some adjustments to accommodate the usability of the original concept:

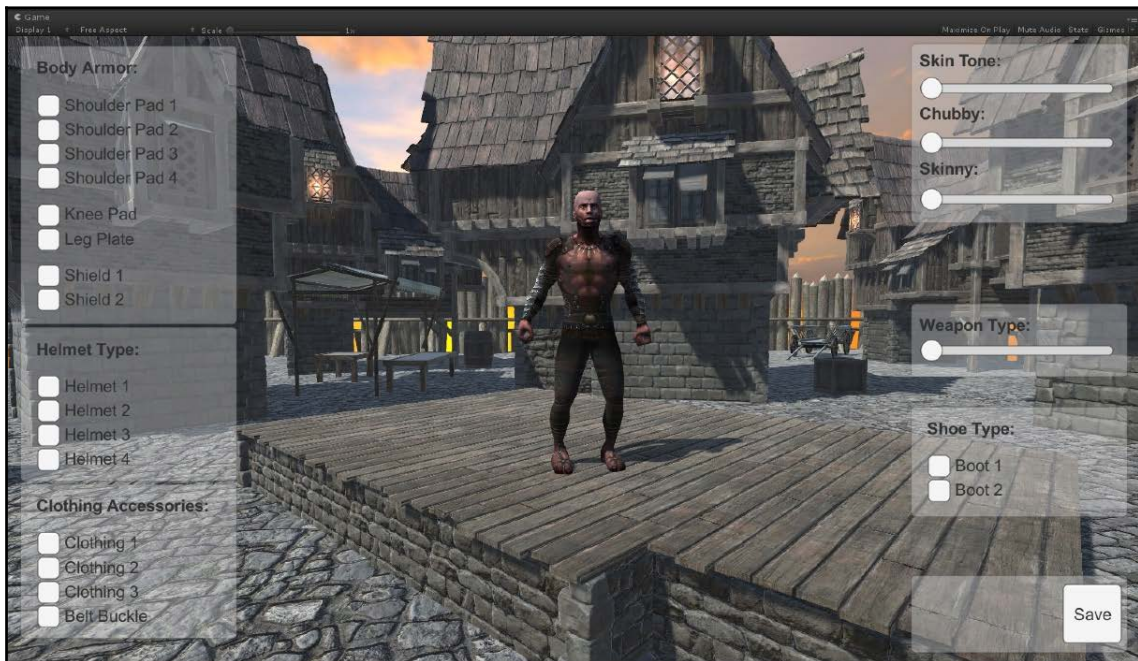


To design our UI, we will need to create a Canvas GameObject. This is done by right-clicking in the **Hierarchy View** and selecting **Create | UI | Canvas**. This will place a Canvas GameObject and an `EventSystem` GameObject in the **Hierarchy View**.



It is assumed that you already know how to create a UI in Unity. If you do not, please refer to *Introduction to Game Programming: Using C# and Unity 3D*, Chapter 5, *GameMaster and Game Mechanics*, at <https://www.amazon.com/Introduction-Game-Programming-Using-Unity-ebook/dp/B01BCPRRCU/>.

I am going to use panels to group the customizable items. For the moment, I will be using checkboxes for some items and scroll bars for the weapons and skin texture. The following screenshot illustrates how my UI for customization looks:



Character customization UI

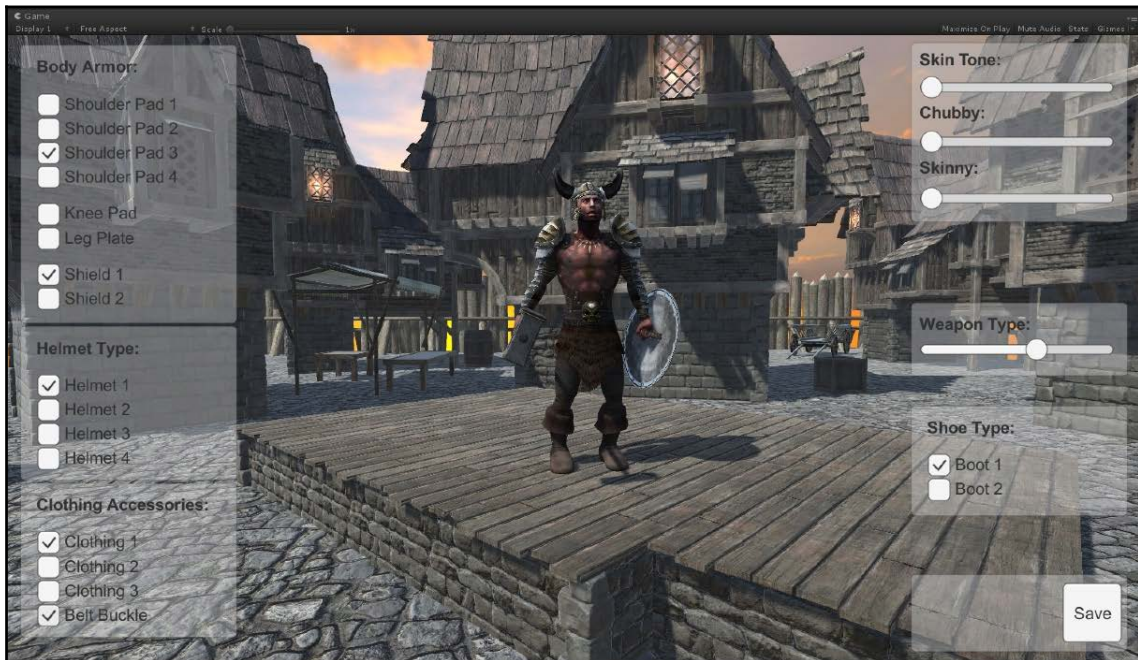
These UI elements will need to be integrated with event handlers that will perform the necessary actions for enabling or disabling certain parts of the character model.

For instance, using the UI, I can select Shoulder Pad 4, use the scroll bar to increase or decrease the chubbiness of the body, and move the weapons type scroll bar until the Hammer weapon shows up. Selecting the second Helmet checkbox, selecting Shield 1, and Boot 2, my character will look the like following screenshot.

We need a way to refer to each one of the meshes representing the different types of customizable objects on the model. This will be done through a C# script. The script will need to keep track of all the parts we are going to be managing for customization.



Some models will not have the extra meshes attached. You can always create empty GameObjects at a particular location on the model, and you can dynamically instantiate the prefab representing your custom object at the given point. This can also be done for our current model. For instance, if we have a special space weapon that somehow gets dropped by aliens in the game world, we can attach the weapon to our model through C# code. The important thing is to understand the concept, and the rest is up to you!



Character customization in action

The code for character customization

Things don't happen automatically. We need to create some C# code that will handle the customization of our character model. The script we create here will handle the UI events that will drive the enabling and disabling of different parts of the model mesh.

Create a new C# script and call it `BarbarianCharacterCustomization.cs`. Create an empty `GameObject` called `__Base` and attach the script to `__Base` `GameObject` in the scene. Here is a listing of the script:

path of the code

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace com.noorcon.rpg2e
{
    public class BarbarianCharacterCustomization : MonoBehaviour
    {
        public GameObject PLAYER_CHARACTER;

        public PlayerCharacter PlayerCharacterData;

        public Material[] PLAYER_SKIN;

        public GameObject CLOTH_01LOD0;
        public GameObject CLOTH_01LOD0_SKIN;
        public GameObject CLOTH_02LOD0;

        // Player Character Defense Weapons
        public GameObject SHIELD_01LOD0;
        public GameObject SHIELD_02LOD0;

        public GameObject QUIVER_LOD0;
        public GameObject BOW_01_LOD0;

        // Player Character Calf - Right / Left
        public GameObject KNEE_PAD_R_LOD0;
        public GameObject LEG_PLATE_R_LOD0;

        public GameObject KNEE_PAD_L_LOD0;
        public GameObject LEG_PLATE_L_LOD0;

        public GameObject BOOT_01LOD0;
        public GameObject BOOT_02LOD0;
```

```
// Use this for initialization
void Start()
{
    PlayerCharacterData =
    PLAYER_CHARACTER.GetComponent<PlayerAgent>().playerCharacterData;
}

public bool ROTATE_MODEL = false;

// Update is called once per frame
void Update()
{
    if (Input.GetKeyUp(KeyCode.R))
    {
        ROTATE_MODEL = !ROTATE_MODEL;
    }

    if (ROTATE_MODEL)
    {
        PLAYER_CHARACTER.transform.Rotate(new Vector3(0, 1, 0), 33.0f *
        Time.deltaTime);
    }

    if (Input.GetKeyUp(KeyCode.L))
    {
        Debug.Log(PlayerPrefs.GetString("Name"));
    }
}

void DisableShoulderPads()
{
    SHOULDER_PAD_R_01LOD0.SetActive(false);
    SHOULDER_PAD_R_02LOD0.SetActive(false);
    SHOULDER_PAD_R_03LOD0.SetActive(false);
    SHOULDER_PAD_R_04LOD0.SetActive(false);

    SHOULDER_PAD_L_01LOD0.SetActive(false);
    SHOULDER_PAD_L_02LOD0.SetActive(false);
    SHOULDER_PAD_L_03LOD0.SetActive(false);
    SHOULDER_PAD_L_04LOD0.SetActive(false);
}
```

This is a long script but it is straightforward. At the top of the script, we have defined all of the variables that will be referencing the different meshes in the model character. All variables are of type `GameObject` with the exception of the `PLAYER_SKIN` variable, which is an array of `Material` data type. The array is used to store the different types of texture created for the character model:

```

    public void SetShoulderPad(Toggle id)
    {
        try
        {
            PlayerCharacter.ShoulderPad name
            =
            (PlayerCharacter.ShoulderPad)Enum.Parse(typeof(PlayerCharacter.ShoulderPad)
, id.name, true);
            if (id.isOn)
            {
                PlayerCharacterData.SelectedShoulderPad = name;
            }
            else
            {
                PlayerCharacterData.SelectedShoulderPad
                = PlayerCharacter.ShoulderPad.none;
            }
        }
        catch
        {
            // if the value passed is not in the enumeration set it to none
            PlayerCharacterData.SelectedShoulderPad
            = PlayerCharacter.ShoulderPad.none;
        }

        // disable before new selection
        DisableShoulderPads();

        switch (id.name)
        {
        case "SP01":
            {
                SHOULDER_PAD_R_01LOD0.SetActive(id.isOn);
                SHOULDER_PAD_L_01LOD0.SetActive(id.isOn);
                break;
            }
        ...
        case "SP04":
            {
                SHOULDER_PAD_R_04LOD0.SetActive(id.isOn);
                SHOULDER_PAD_L_04LOD0.SetActive(id.isOn);
            }
        }
    }

```



```

        break;
    }
}

public void SetShoulderPad(PlayerCharacter.ShoulderPad id)
{
    // disable before new selection
    DisableShoulderPads();

    switch (id.ToString())
    {
        case "SP01":
        {
            SHOULDER_PAD_R_01LOD0.SetActive(true);
            SHOULDER_PAD_L_01LOD0.SetActive(true);
            break;
        }
        ...
        case "SP04":
        {
            SHOULDER_PAD_R_04LOD0.SetActive(true);
            SHOULDER_PAD_L_04LOD0.SetActive(true);
            break;
        }
    }
}
...https://github.com/PacktPublishing/Building-an-RPG-with-Unity-2018-
Second-Edition

```

There are a few functions defined that are called by the UI event handler. These functions are: `SetShoulderPad(Toggle id)`, `SetBodyType(Toggle id)`, `SetKneePad(Toggle id)`, `SetLegPlate(Toggle id)`, `SetWeaponType(Slider id)`, `SetHelmetType(Toggle id)`, `SetShieldType(Toggle id)`, `SetSkinType(Slider id)`, `SetBodyFat(Slider id)`, and `SetBodySkinny(Slider id)`;

All of the functions take a parameter that identifies which specific type should enable or disable.

We have just created a tool that will enable you quickly visualize character customization.

You can also use the system we just built to create all of the different variations of your player character or non-character player models and store them as prefabs! Wow! This will save you so much time and effort in creating your characters representing different barbarians!

Preserving our character state

Now that we have spent the time to customize our character, we need to preserve our character and use it in our game. In Unity, there is a function called `DontDestroyOnLoad()`.

This is a great function that can be utilized at this time. What does it do? It keeps the specified `GameObject` in memory going from one scene to the next. We can use these mechanisms for now. Eventually, though, you will want to create a system that you can save and load your user data.

Go ahead and create a new C# script and call it `DoNotDestroy.cs`. This script is going to be very simple. Here is the listing:

```
using UnityEngine;
using System.Collections;

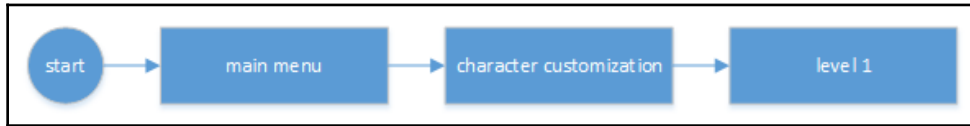
public class DoNotDestroy : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
        DontDestroyOnLoad(this);
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

After you create the script, go ahead and attach it to your character model prefab in the scene. Not bad; let's do a quick recap of what we have done so far.

Recap

By now you should have three scenes that are functional. We have our scene that represents the main menu, we have our scene that represents our initial level, and we just created a scene that is used for character customization. Here is the flow of our game thus far:



We start the game, see the main menu, select the **Start Game** button to enter the character customization scene, do our customization, and when we click the **Save** button, we load level 1.

For this to work, we have created the following C# scripts:

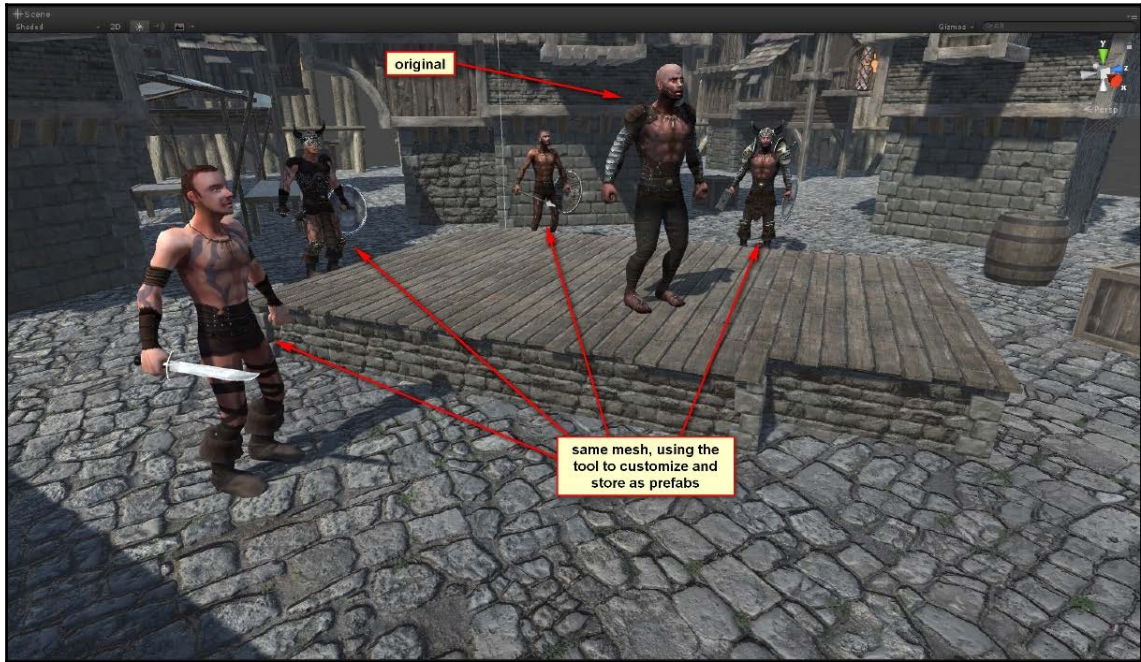
- `GameMaster.cs`: This is used as the main script to keep track of our game state
- `BarbarianCharacterCustomization.cs`: This is used exclusively for customizing our character
- `DoNotDestroy.cs`: This is used to save the state of a given object
- `BarbarianCharacterController.cs`: This is used to control the motion of our character
- `IKHandle.cs`: This is used to implement inverse kinematics for the foot

When you combine all of this together, you now have a good framework and flow that can be used to extend and improve as we go along.

Non-player characters

Until now, we have concentrated on the player character. In this section, we will start thinking about our non-player characters. Let's start with our Barbarians. We can use our Character Customization scene to quickly create a few prefabs that will represent our unique Barbarians.

Using the tool we have just developed, you can make your adjustments, and when satisfied with your model, drag and drop the `GameObject` representing your character player into the `Prefabs` folder. This will create a copy of the instance of the `GameObject` as you see it and save it into a prefab. The following screenshot demonstrates the two characters I have created and stored as a prefab:



Creating unique characters using the tool

What I have shown you, if done properly, could save you hours of tedious work to manually go down the model structure and individually enable and disable the different meshes. In other words, we not only create a scene that allows us to customize the in-game player character, we have also created a tool that can help us customize our own character models quickly for use within the game!

Another point to emphasize here is the power of prefabs. Think of a prefab as a storage that can be used to save the state of a given `GameObject` and used over and over again within your game environment. When you update your prefab, all instances of the prefab will automatically get updated! This is great, but at the same time you have to be careful not to break anything for the same reason. When you update a code logic on a script that is attached to a prefab, all instances of the prefab will use the updated script, so a bit of planning on your part can save a lot of time and headache in the long run.

Non-player character basics

We are going to be using the newly created prefabs to implement our non-player characters. Since there are some similarities in the character models, we can re-use some of the assets that we have created so far.

For instance, all character will be inheriting the `BaseCharacter` class defined in *Chapter 3, RPG Character Design*. They will also incorporate the same states we have already created for the player character and extend a few more states specifically for the NPC, such as searching and seeking.

We have used our character customization tool to create and save our non-player character; hence, we are OK with the modeling part. What we need to concentrate on is the motion of our non-player characters. We would need to create a new Animator Controller that will handle the states of our NPCs.

Setting up the non-player character

One of the main difficulties for implementing an NPC is the ability to give it realistic intelligence. This can be achieved easily by identifying and implementing several key areas for our NPCs.

There are a few new components we would need to attach to our NPCs. Using the prefab we have saved, we will need to add the following components:

- New Sphere Collider, this will be used to implement the range of sight for our NPC.
- We already have an Animator component attached, but we will need to create a new Animator Controller to capture new states for the NPC.
- We also need to add a Nav Mesh Agent component. We are going to use the built-in navigation and pathfinding system for our NPC.

To add the Sphere Collider, you will need to select the prefab defined for the NPC, and in the **Inspector** Window. Select **Add Component** | **Physics** | **Sphere Collider**. This will attach a Sphere Collider to our prefab.

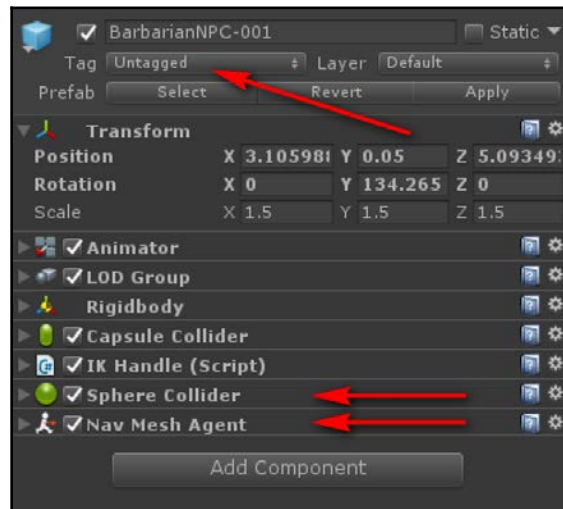
Next, we need to add Nav Mesh Agent. Again, from the Inspector Window, select **Add Component** | **Navigation** | **Nav Mesh Agent**. OK, so now we have set up our main built-in components that are going to be used for the NPC.

Since our prefab is an instance of our player character, we will need to remove some of the script components that have been carried over. If your NPC prefab contains any scripts attached to it, go ahead and remove them now.



Make sure you also change the Tag property to `Untagged`, if you have not done so already.

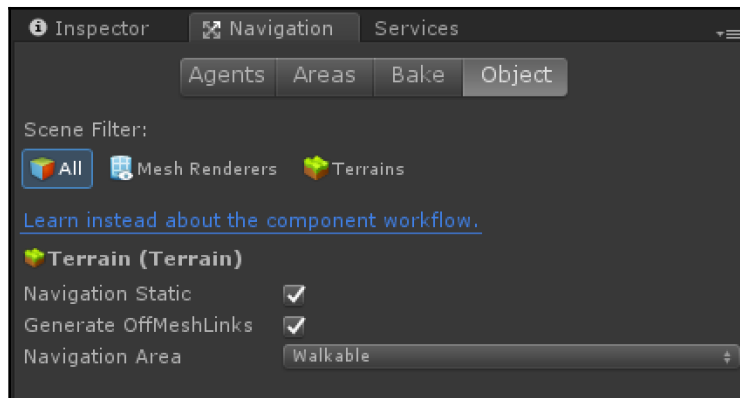
The following screenshot illustrates the components that we have so far on the NPC. This will include both the existing components, including the scripts we have brought over from the player character, and the newly added components that will be used for the NPC:



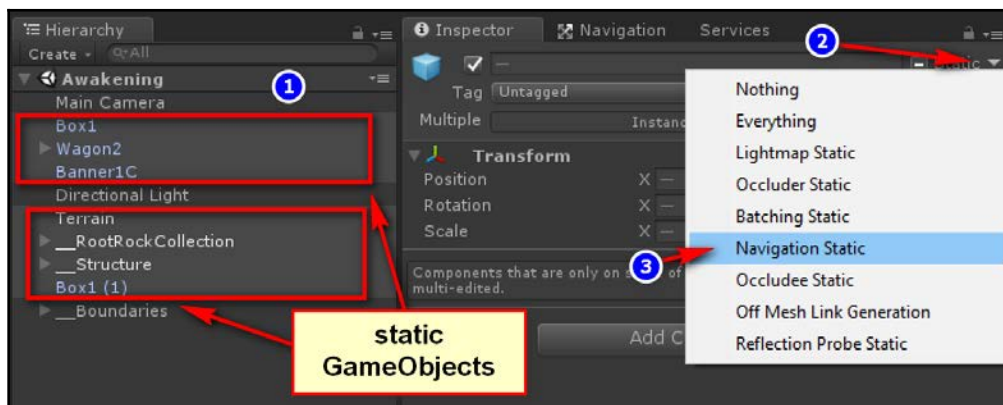
Before you perform the next step, make sure you are in one of the level scenes. I am going to use the Awakening scene.

Switch to one of your playable game scenes.

The next step is to set up our Navmesh. To create a Navmesh, we need to get into the **Navigation** window, by selecting **Window | Navigation**:



In order for the navmesh to work properly, we will need to mark all GameObjects that are going to be static in the scene as **Navigation Static**. This will create a navmesh based on the static objects in the scene; that is, GameObjects that are not going to be moving throughout the lifespan on the scene:

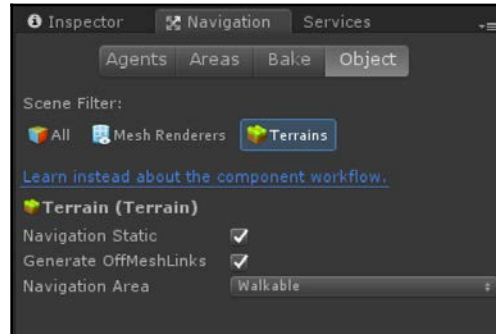


In your active scene, select the GameObjects that are going to be set as **Navigation Static**, as shown in the preceding screenshot (1), use the **Static** drop-down menu (2), and select the **Navigation Static** option (3). If your GameObject is a parent GameObject with children, Unity will ask if you want to apply the property change to all children.

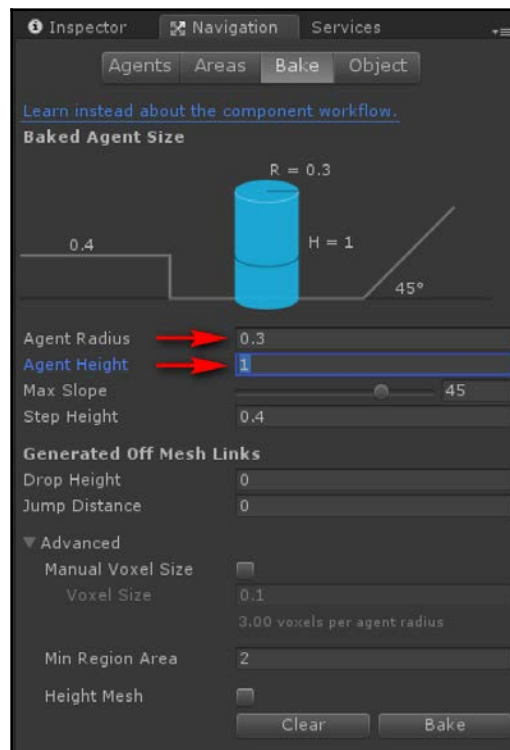


Notice that I have placed all of my environment GameObjects under a GameObject called `__Structure`, `__RootRockCollection` and a few others that have been added at a later time. This way, if I have many static objects, I can apply the property change to the parent and the children will automatically inherit the change as well. But make sure everything in the group will be static!

Once this is complete, we need to go back to the **Navigation** window and make some adjustments. In the **Navigation** tab, select **Terrains** and make sure it is set to **Navigation Static**, and the **Navigation Area** is set to **Walkable**:

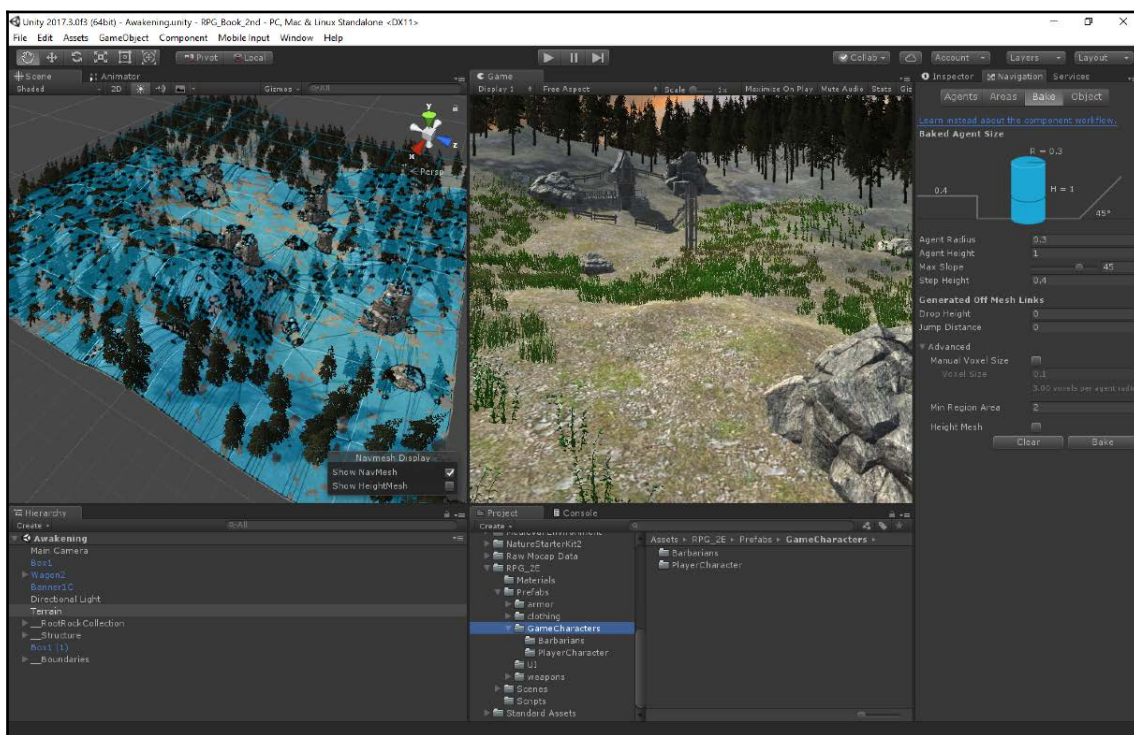


In the **Bake** tab, change the **Agent Radius** to **0.3** and **Agent Height** to **1**. Keep the rest of the properties the same. This will give the NPC more freedom to pass through tight corners:



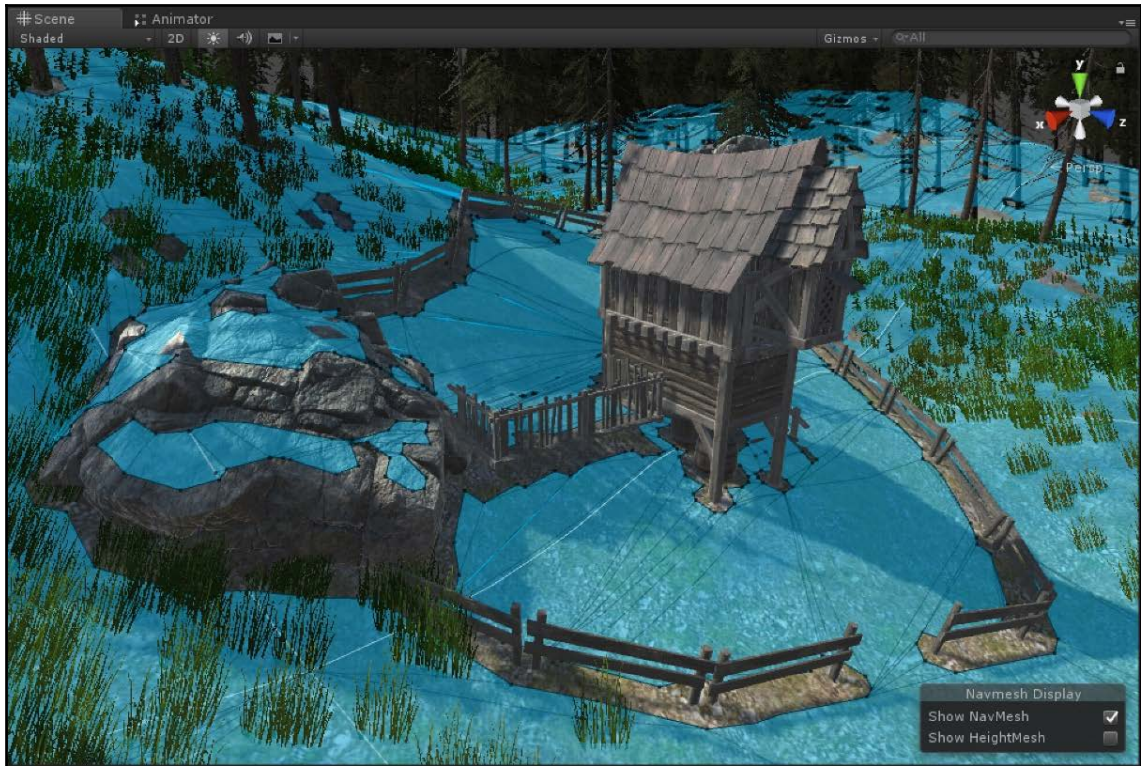
When you are ready, you can select the **Bake** button at the bottom of the **Navigation** window.

Unity will take some time to generate the Navmesh for your scene. This will depend on the complexity of your level. If all is done correctly, you will see something similar to the following screenshot displaying your Navmesh:



Navmesh generation

The blue areas you see are all the regions that the NPC can actually navigate to:

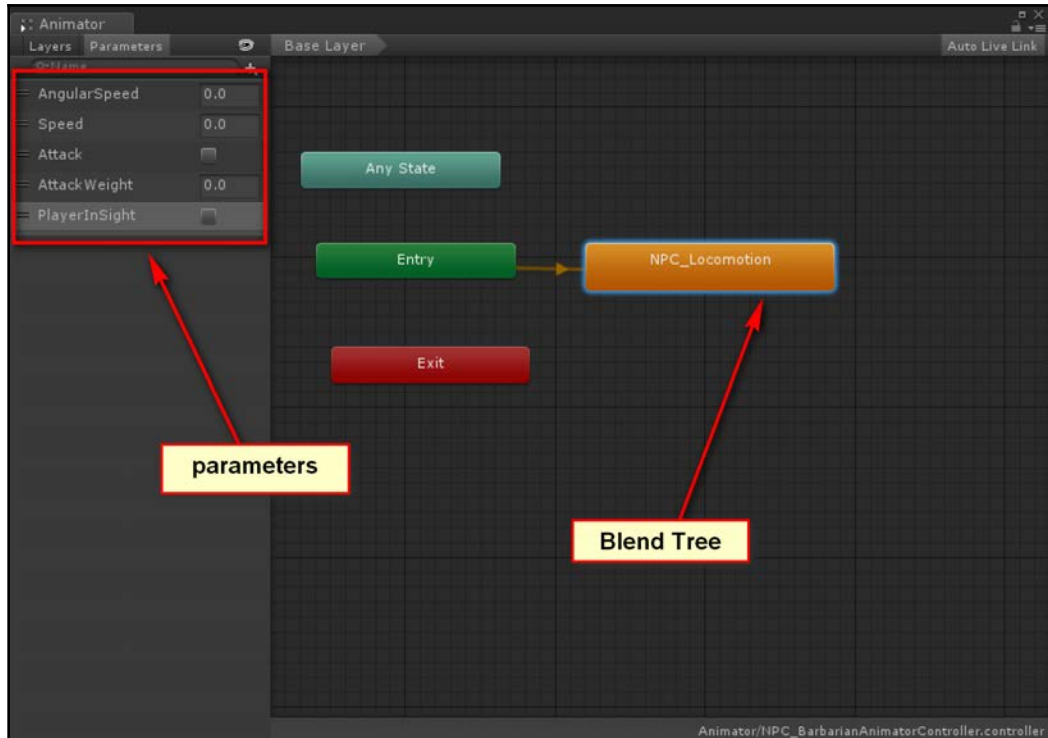


NPC Animator Controller

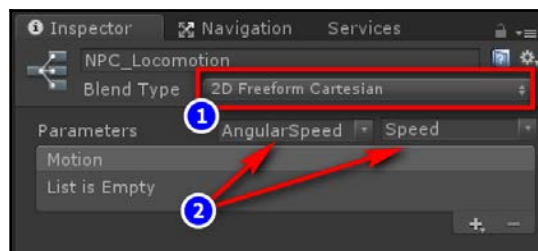
We now need to create the **Animator Controller (AC)** for our NPC. The Animator Controller will use input from the `MeshAgent` to control and change the state of our NPC. We also need to define a few parameters for our NPC AC. These are going to be:

- `AngularSpeed`: This will be used for directional movement
- `Speed`: This will be used to determine how fast the NPC will be moving
- `Attack`: This will be used to determine if it needs to attack
- `AttackWeight`: This might be used
- `PlayerInSight`: This will be used to determine if the PC is in sight

Go ahead and create a new Animator Controller in your project and name it `NPC_BarbarianAnimatorController`. Open the Animator Window. Create a new Blend Tree by right-clicking in the Animator Window and selecting **Create State | From New Blend Tree**. Change the name to `NPC_Locomotion`. Double-click it so that you can edit the Blend Tree. Change the node name to `NPC_Locomotion` as well:



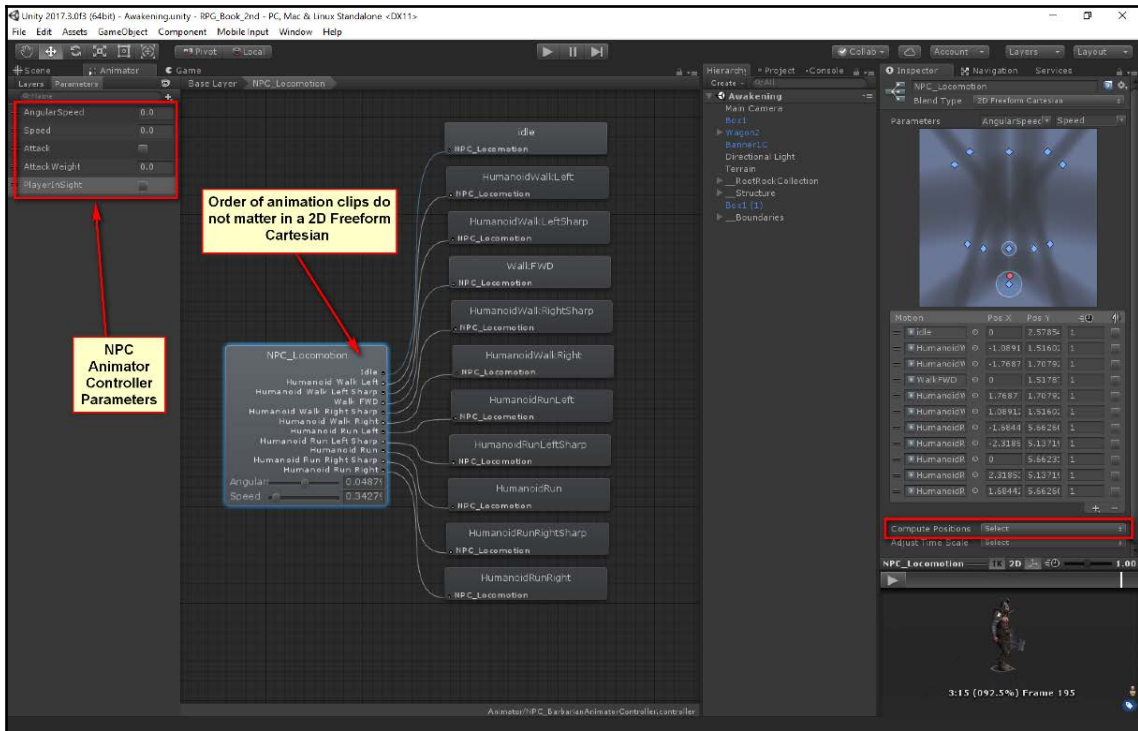
From the Inspector Window, change the *Blend Type* to *2D Freeform Cartesian*:



The x -axis will be represented by the *AngularSpeed*, and the y -axis will be represented by the *Speed* parameters.

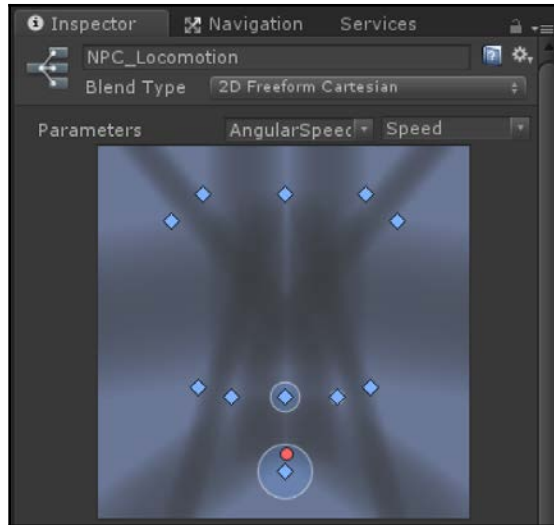
The Blend Tree is going to hold all of the different locomotion animation states. These are going to be the idle, walking, and running states.

I have set up 11 different animation states for the locomotion of my NPC. The following screenshot will give you an overview of the Blend Tree:



NPC Blend Tree

Once you include all of the animations states in the Blend Tree, you will need to compute the positions of your animations. An easy way to do this, is to select the *Compute Positions* dropdown and select *AngularSpeed and Speed*. This will place the animation position based on the root motion, as illustrated in the following screenshot:

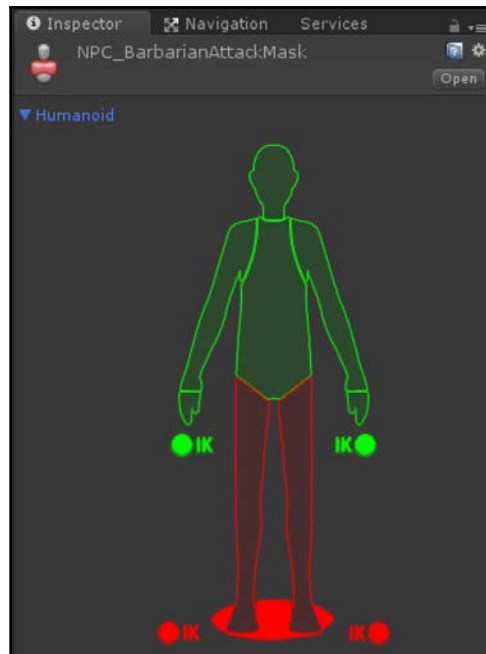


You can use your mouse to drag the red point shown in the screenshot to preview your animation states in action.

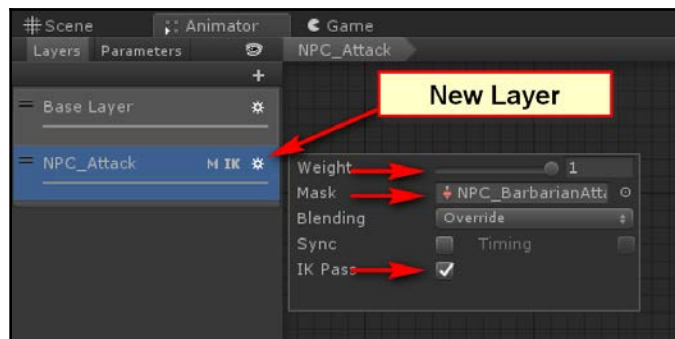
NPC Attack

In order to implement our attack mode, we will need to create a new layer in the Animator Controller. Go ahead and create a new layer and call it `NPC_Attack`. This layer will be responsible for animating our character when we enter the attack mode.

We need to create a new mask for the layer. The mask will be used to determine which parts of the humanoid body will be affected by the layer animation. To create a mask, right-click in your Project Window and select **Create | Avatar Mask**. Name the new mask `NPC_BarbarianAttackMask`. Use the Inspector Window to disable the body parts that we don't want to be affected by the layer animation, as illustrated in the following screenshot:



Your layer setup should look like the following screenshot:

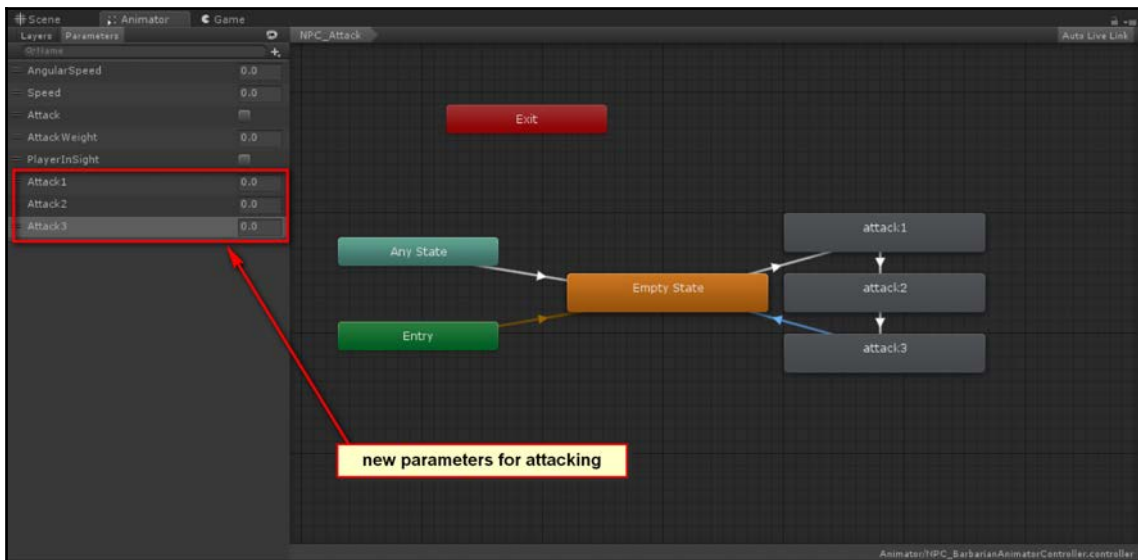


Make sure you change the Weight property to 1, the Mask property assigned to the Avatar Mask we created, and also that the IK property is checked. Now we are ready to create our attack state machine.

Right-click in the Animator Window and select **Create State | Empty**. Drag and drop your attack animation(s). The empty state is used to have a nice transition between the main layer and back.

After you have dropped your attack animation(s) into the Animator, you will need to connect them using the transition conditions. I have added three more parameters to the parameter list named **attack1**, **attack2**, and **attack3**. These parameters, in connection with the Attack parameter, will determine which attack state our NPC will transition to.

The following screenshot shows the NPC_Attack layer as configured up to this point:



New parameters

Finally, you want to assign the new NPC_BarbarianAnimatorController to the NPC prefab(s).

NPC AI

Now it is time to give some intelligence to our NPCs. One of the scripts we will need to create is to give the NPC the ability to detect the player. This script will be called `NPC_BarbarianMovement.cs`. The script will be used to detect if the player is in-sight, calculate the field of view for the NPC, and calculate the path from the NPC to the player character.

Here is a listing of the source code:

```
using UnityEngine;
using UnityEngine.AI;
namespace com.noorcon.rpg2e
{
    public class NPC_BarbarianMovement : MonoBehaviour
    {
        // reference to the animator
        public Animator animator;
        // these variables are used for the speed
        // horizontal and vertical movement of the NPC
        public float speed = 0.0f;
        public float h = 0.0f;
        public float v = 0.0f;
        public bool attack = false; // used for attack mode 1
        public bool jump = false; // used for jumping
        public bool die = false; // are we alive?
        // used for debugging
        public bool DEBUG = false;
        public bool DEBUG_DRAW = false;
        // Reference to the NavMeshAgent component.
        private NavMeshAgent nav;
        // Reference to the sphere collider trigger component.
        private SphereCollider col;
        // where is the player character in relation to NPC
        public Vector3 direction;
        // how far away is the player character from NPC
        public float distance = 0.0f;
        // what is the angle between the PC and NPC
        public float angle = 0.0f;
        // a reference to the player character
        public GameObject player;
        // is the PC in sight?
        public bool playerInSight;
        // what is the field of view for our NPC?
        // currently set to 110 degrees
        public float fieldOfViewAngle = 110.0f;
        // calculate the angle between PC and NPC
```



```
public float calculatedAngle;
void Awake()
{
    // get reference to the animator component
    animator = GetComponent<Animator>() as Animator;
    // get reference to nav mesh agent
    nav = GetComponent<NavMeshAgent>() as NavMeshAgent;
    // get reference to the sphere collider
    col = GetComponent<SphereCollider>() as SphereCollider;
    // get reference to the player
    player = GameObject.FindGameObjectWithTag("Player") as GameObject;
    // we don't see the player by default
    playerInSight = false;
}
// Use this for initialization
void Start()
{
}
void Update()
{
    // if player is in sight let's slerp towards the player
    if (playerInSight)
    {
        transform.rotation =
        Quaternion.Slerp(this.transform.rotation,
        Quaternion.LookRotation(direction), 0.1f);
    }
}
// let's update our scene using fixed update
void FixedUpdate()
{
    h = angle; // assign horizontal axis
    v = distance; // assign vertical axis
    // calculate speed based on distance and delta time
    speed = distance / Time.deltaTime;
    if (DEBUG)
    Debug.Log(string.Format("H:{0} - V:{1} - Speed:{2}", h, v, speed));
    // set the parameters defined in the animator controller
    animator.SetFloat("Speed", speed);
    animator.SetFloat("AngularSpeed", v);
    animator.SetBool("Attack", attack);
}
```

OK, so let's actually take a look and see what this code is trying to do. In the `Awake()` function we are initializing our variables that will be used in the script. We have a reference to the `NavMeshAgent`, the `SphereCollider`, and the `Animator` components attached to the NPC. These are stored in the `nav`, `col`, and `anim` variables, respectively:

```
// if the PC is in our collider, we want to examine the location
// of the player
// calculate the direction based on our position and the
// player's position
// use the DOT product to get the angle between the two vectors
// calculate the angle between the NPC forward vector and the PC
// if it falls within the field of view, we have the player in
// sight
// if the player is in sight, we will set the nav agent destination
// if we are within a certain distance from the PC, the NPC has
// the ability to attack
void OnTriggerStay(Collider other)
{
    if (other.transform.tag.Equals("Player"))
    {
        // Create a vector from the enemy to the player and store
        // the angle between it and forward.
        direction = other.transform.position - transform.position;
        distance = Vector3.Distance(other.transform.position, transform.position) -
        1.0f;
        float DotResult = Vector3.Dot(transform.forward,
        player.transform.position);
        angle = DotResult;
        if (DEBUG_DRAW)
        {
            Debug.DrawLine(transform.position + Vector3.up, direction * 50,
            Color.gray);
            Debug.DrawLine(other.transform.position, transform.position, Color.cyan);
        }
        playerInSight = false;
        calculatedAngle = Vector3.Angle(direction, transform.forward);
        if (calculatedAngle < fieldOfViewAngle * 0.5f)
        {
            RaycastHit hit;
            if (DEBUG_DRAW)
            Debug.DrawRay(transform.position + transform.up, direction.normalized,
            Color.magenta);
            // ... and if a raycast towards the player hits something...
            if (Physics.Raycast(transform.position + transform.up,
            direction.normalized, out hit,
            col.radius))
            {

```

```
// ... and if the raycast hits the player...
if (hit.collider.gameObject == player)
{
    // ... the player is in sight.
    playerInSight = true;
    if (DEBUG)
        Debug.Log("PlayerInSight: " + playerInSight);
}
}
}
if (playerInSight)
{
    nav.SetDestination(other.transform.position);
    CalculatePathLength(other.transform.position);
    if (distance < 1.1f)
    {
        attack = true;
    }
    else
    {
        attack = false;
    }
}
}
```

We also need to get a reference to the player and the player animator component. These is done through the `player` variable. We are also setting the `playerInSight` variable to false by default:

```
void OnTriggerExit(Collider other)
{
    if (other.transform.tag.Equals("Player"))
    {
        distance = 0.0f;
        angle = 0.0f;
        attack = false;
        playerInSight = false;
    }
}
// this is a helper function at this point
// in the future we will use it to calculate distance around
// the corners
// it currently is also used to draw the path of the nav mesh
// agent in the
// editor
float CalculatePathLength(Vector3 targetPosition)
{
}
```

```

// Create a path and set it based on a target position.
NavMeshPath path = new NavMeshPath();
if (nav.enabled)
nav.CalculatePath(targetPosition, path);
// Create an array of points which is the length of the number
// of corners in the path + 2.
Vector3[] allWayPoints = new Vector3[path.corners.Length + 2];
// The first point is the enemy's position.
allWayPoints[0] = transform.position;
// The last point is the target position.
allWayPoints[allWayPoints.Length - 1] = targetPosition;
// The points inbetween are the corners of the path.
for (int i = 0; i < path.corners.Length; i++)
{
allWayPoints[i + 1] = path.corners[i];
}
// Create a float to store the path length that is by default 0.
float pathLength = 0;
// Increment the path length by an amount equal to the
// distance between each waypoint and the next.
for (int i = 0; i < allWayPoints.Length - 1; i++)
{
pathLength += Vector3.Distance(allWayPoints[i], allWayPoints[i + 1]);
if (DEBUG_DRAW)
Debug.DrawLine(allWayPoints[i], allWayPoints[i + 1], Color.red);
}
return pathLength;
}
}
}

```

The `Update()` function is not performing anything major at this point. It is just checking to see if the player character is in sight, and, if so, it makes sure that the NPC is orienting itself to look at the player.

Most of the meat of our code is in the `OnTriggerStay()` function. The first things we need to do is make sure the object that has entered our collider is the player object. This is done by checking the tag attribute on the other collider.

If the player is within our collider, then we go ahead and calculate the direction, the distance, and the angle of the player relative to the NPC. This is done with the following lines:

```

direction = other.transform.position - transform.position;

distance = Vector3.Distance(other.transform.position, transform.position) -
1.0f;

```

```
float DotResult = Vector3.Dot(transform.forward,player.transform.position);  
angle = DotResult;
```

Then, if the angle is smaller than the `fieldOfViewAngle` variable, we can use ray casting to determine if we can hit the player. If that is the case, the player is in NPC's sight:



Debug rays

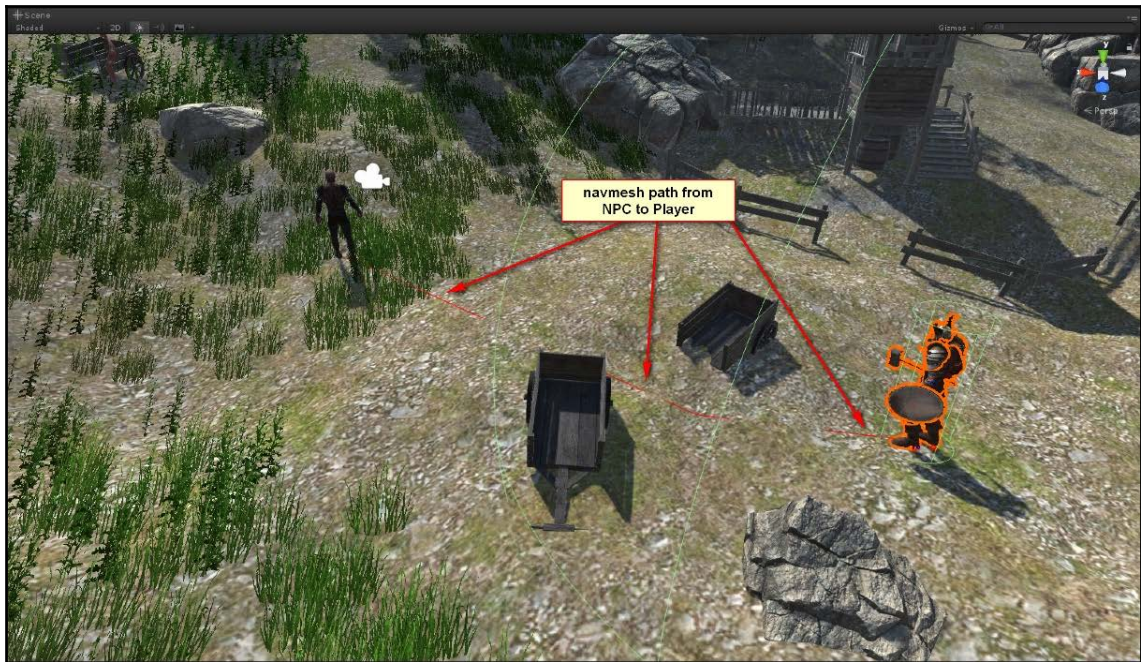
There is one more critical calculation the NPC needs to perform. That is how to get to the player once it is in range. Once we have established that the player is in range and that we are facing the player, we need to make the NPC find its way to the player. This is where the `NavMesh` and the `NavMeshAgent` come into play.

The `CalculatePathLength()` is a function that takes the position of the player and, using the mesh data, it calculates the best path to navigate from the NPC's location to the player's location.

However, there is one more additional calculation we are performing, and that is, we are calculating the length of the path between the two points. This length calculation will be used in the future to perform the following:

If the length of the path is larger than a threshold we have set, then we won't make the NPC attack. If it is within mean, then we can make the NPC move towards the player to engage in battle.

In the last function `OnTriggerExit()`, we set the `playerInSight` variable to false. This will stop the NPC from pursuing the player:



Navmesh path

The preceding screenshot illustrates the path between the NPC and the player based on real-time calculation.

Go ahead and attach the script to the NPC prefab if you have not done so already, and run the application to test it out. If all things are good, then you will be able to move the player character around the level, and once the player character enters the NPC field of view, the NPC will start moving towards the player and when close enough, it will attack:



At this point, your NPC should have the following components attached to its prefab:

- Animator
- Rigidbody
- Capsule and sphere colliders
- NavMesh Agent
- `NPC_BarbarianMovement` script

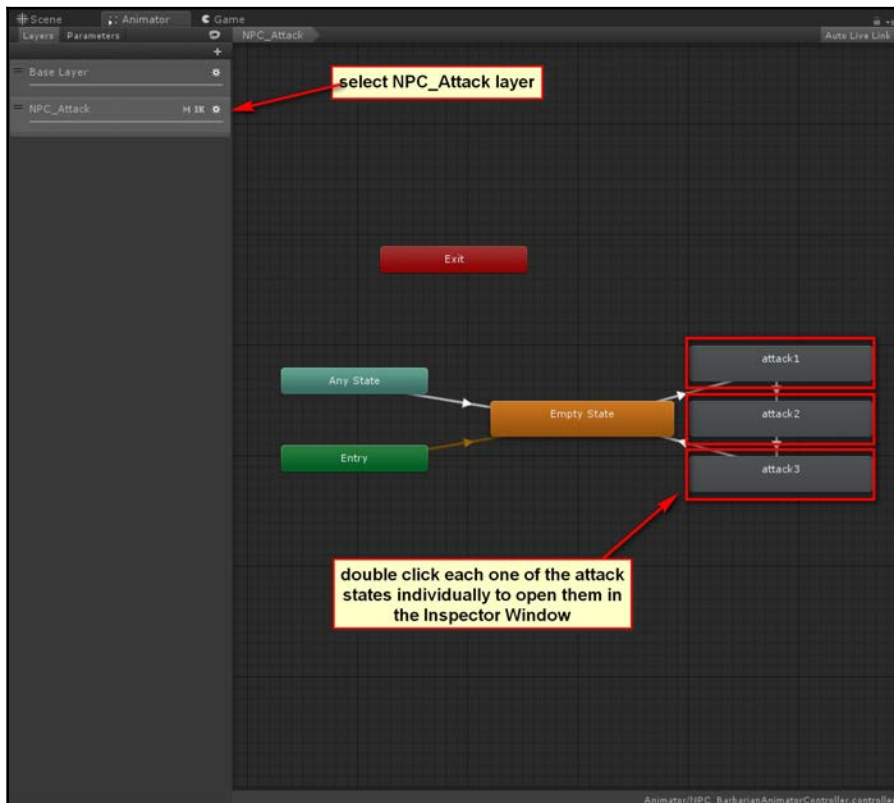
We have covered a lot of information. I would encourage you to take the time to read through it one more time and understand the concepts before moving forward.

PC and NPC interaction

Thus far, we have created the basics movement for both our PC and NPC. The next item I would like to complete is the attack mechanism for the PC and the NPC characters. Let's start by implementing the hit for the NPC.

Our NPC detects the player character based on the code we just created in the previous section. When the player character is in sight, the NPC will find the shortest path to the player character, and at a given range, it will attack the player character. We have the movement and animation mechanics completed. The next objective is to keep track of the hit points when the NPC is attacking.

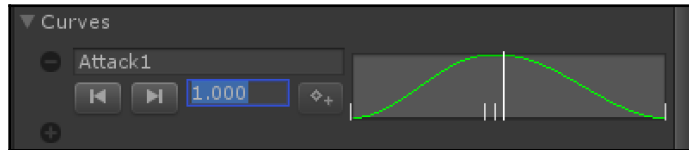
There are a few adjustments we need to make in `NPC_Animator_Controller`. Open the **Animator** window, and select the `NPC_Attack` layer:



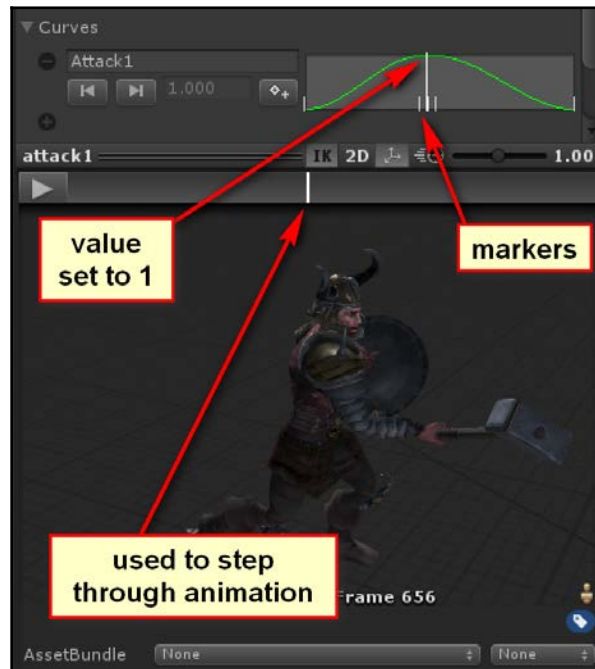
NPC attack layer

Double-click on the *attack1* state, or the attack state you have defined in your state machine. This will open the related animation in the **Inspector** window.

In the **Inspector** window, scroll down to the *Curves* section. We are going to create a new curve by selecting the (+) sign under the *Curves* section. We are also going to create a new parameter called *Attack1C* to represent the value of the curve. This parameter should be of type float:



The curve displayed in the preceding screenshot will be based on your animation:



In the preceding screenshot, I have marked the important parts of the interface that you will need to work with to configure the curve of an animation. The first step would be to actually preview your animation and get a feeling for it.

The next step for my particular animation sequence was determining when the right arm of the model is coming along and I set a marker in the curve. I make another marker a bit more into the animation where the right arm has crossed a good deal from the right side to the left side.

These markers will indicate a hit point during the animation when the NPC is in attack mode:



OK, why do we do this? Simple. This will help us only generate a hit based on the curve of the animation. This way, we don't hit the player and reduce the health of the player while the weapon is away from the player's body.

Next, we need to update our `NPC_BarbarianMovement.cs` code to program the NPC attack.



Note: I have only listed the portions that have been updated.

Here is an updated listing of the code:

```
using UnityEngine;
using System.Collections;

public class NPC_Movement : MonoBehaviour
{
    ...
    void Update()
    {
        // if player is in sight let's slerp towards the player
        if (playerInSight)
        {
            this.transform.rotation =
                Quaternion.Slerp(this.transform.rotation,
                                Quaternion.LookRotation(direction), 0.1f);
        }

        if(this.player.transform.GetComponent<CharacterController>().die)
        {
            animator.SetBool("Attack", false);
            animator.SetFloat("Speed", 0.0f);
            animator.SetFloat("AngularSpeed", 0.0f);
        }
    }

    // let's update our scene using fixed update
    void FixedUpdate()
    {
        h = angle;           // assign horizontal axis
        v = distance;        // assign vertical axis

        // calculate speed based on distance and delta time
        speed = distance / Time.deltaTime;

        if (DEBUG)
            Debug.Log(string.Format("H:{0} - V:{1} - Speed:{2}", h, v,
speed));

        // set the parameters defined in the animator controller
        animator.SetFloat("Speed", speed);
        animator.SetFloat("AngularSpeed", v);
        animator.SetBool("Attack", attack1);
        animator.SetBool("Attack1", attack1);

        if(playerInSight)
        {
            if (animator.GetFloat("Attack1C") == 1.0f)
```

```

        {
            this.player.GetComponent<PlayerAgent>().playerCharacterData.HEALTH -= 1.0f;
        }
    }
    ...
}

```

The new addition to the code checks to see if the player is in sight, and if that is the case, we check to see if we are in range to be able to attack. If that is the case, we get into the attack mode. If we are in the attack mode, the attack animation is played.

In the code, we check to get the value of the newly created parameter called `Attack1`, and if it happens to be of value `1.0`, then we go ahead and reduce the health of the player character.

If the player dies while the NPC is attacking, it will stop attacking and go back into the idle state.

OK, you might be wondering how did we get the ability to get the information from the player character. This is because we need to make some more additional C# scripts. Let's go ahead and do so now. Create the following C# scripts:

- `PlayerCharacter.cs`: This is going to be our player character class, which inherits from the `BaseCharacter` class we have defined previously
- `PlayerAgent.cs`: This is going to be used to store the PC data and also inherit `MonoBehaviour`
- `NPC.cs`: This is going to be our non-player character class, which inherits from the `BaseCharacter` class as well
- `NPC_Agent.cs`: This is going to be used to store the NPC data and also inherit `MonoBehaviour`

I have made some modification to the `BaseCharacter.cs` script to make it more accessible through the editor. Here is the new listing:

```

using System;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class BaseCharacter
    {
        [SerializeField]
        public string Name;
        [SerializeField]

```

```
        public string Description;
        [SerializeField]
        public float Strength;
        [SerializeField]
        public float Defense;
        [SerializeField]
        public float Dexterity;
        [SerializeField]
        public float Intelligence;
        [SerializeField]
        public float Health;
    }
}
```

I have gone ahead and made the class and the fields serializable.

Let's take a look at the listing for `PlayerCharacter.cs`:

```
using System;
using UnityEngine;

namespace com.noorcon.rpg2e
{
    [Serializable]
    public class PlayerCharacter : MonoBehaviour
    {

    }
}
```

There is nothing much going on there at this point. Now let's take a look at the `PlayerAgent.cs`:

```
using System;
using UnityEngine;

namespace com.noorcon.rpg2e
{
    [Serializable]
    public class PlayerAgent : MonoBehaviour
    {
        public PlayerCharacter playerCharacterData;

        void Awake()
        {
            PlayerCharacter tmp = new PlayerCharacter();
            tmp.Name = "Maximilian";
            tmp.Health = 100.0f;
        }
    }
}
```

```
        tmp.Defense = 50.0f;
        tmp.Description = "Our Hero";
        tmp.Dexterity = 33.0f;
        tmp.Intelligence = 80.0f;
        tmp.Strength = 60.0f;

        playerCharacterData = tmp;
    }

    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (playerCharacterData.Health < 0.0f)
        {
            playerCharacterData.Health = 0.0f;

            transform.GetComponent<BarbarianCharacterController>().die =
true;
        }
    }
}
```

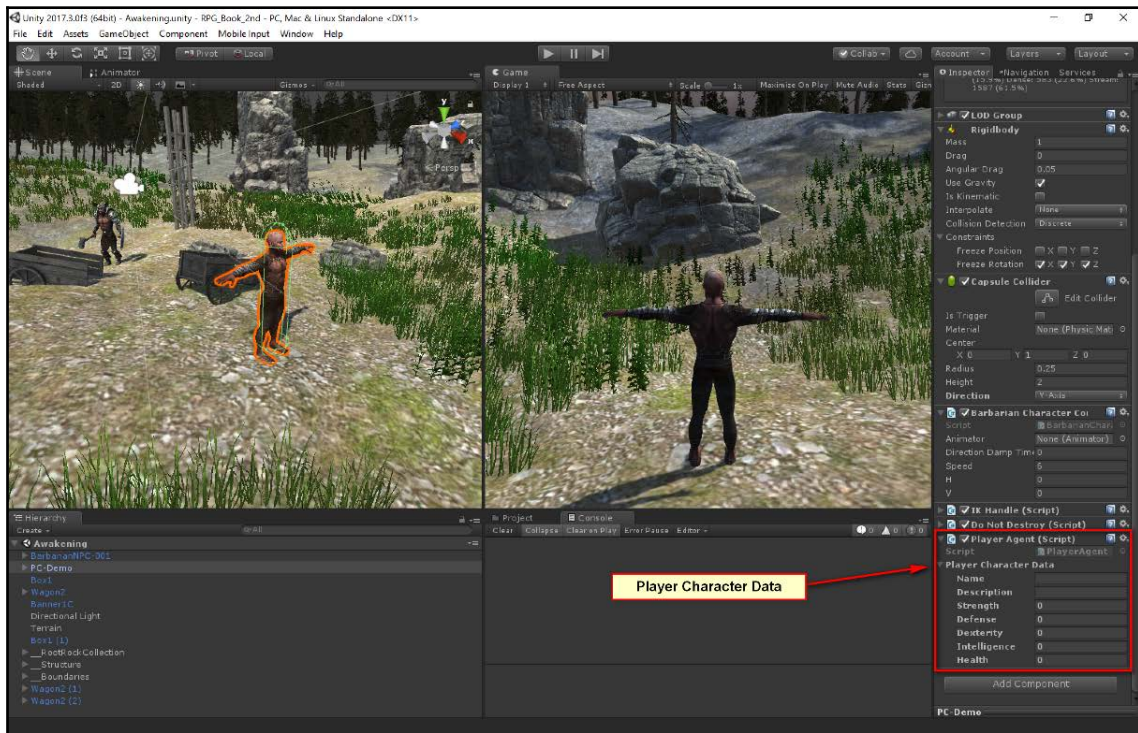
In the player agent code, we are initializing some default values for our PC data in the `Awake()` function. Since the class has been serialized, we can actually see the data during runtime for debugging purposes.

In the `Update()` function, we check to see if the health of our PC is less than 0.0f, and if it is, then this indicates the player has died. Then we use the `CharacterController` component we have created to set the *die* property to true. The `CharacterController` then will use the new value and communicate with the `Animator Controller` for the player character to get into the die state.



Notice that our `NPC_BarbarianMovement.cs` script is accessing the exact same PC data through the reference we have created in the script.

You will need to attach the `PlayerAgent.cs` script to your player character in the scene:



Player character data

In the preceding screenshot, you can see the additions we have done to the scripts and how they look during runtime. We will have a listing for the `NPC.cs` and `NPC_Agent.cs` in future chapters. At this point, they are not used.

Summary

This chapter was a very involved one. We covered some very important topics and concepts in the chapter that can be used and enhanced for your games. We started the chapter by looking into how to customize your player character. The concepts you take away from the section can be applied to a wide variety of scenarios.

We looked at how to understand the structure of your character model so that you can better determine the customization methods. These are the different types of weapons, clothing, armor, shields, and on.

We then looked at how to create a user interface to help enable us with the customization of our player character during game play. We also learned that the tool we developed can be used to quickly create several different character models (customized) and store them as prefabs for later use. Great time saver! We also learned how to preserve the state of our player character after customization for game play.

We looked at the non-player characters next. We went through the basics of setting up the NPC with the different necessary components. We then looked at how to create a Navmesh and how to work with Navmesh Agent and Pathfinding using the Navmesh.

We created a new Animator Controller for the NPC. We created a 2D Freeform Cartesian Blend Tree that was used for the animation of the NPC. We looked at how to create multiple layers in the Animation Controller and enable IK for different regions of the humanoid skeleton. We created the initial NPC AI script to detect and determine if the player is close enough for it to make a move and attack. Finally, we created new scripts to make interaction between the NPC and the player character possible.

By the end of the chapter, you should have a good grasp of how everything is interrelated and have an idea of how to approach your project.

In the next chapter, we will create a better way to manage our game state.

5

GameMaster and Game Mechanics

In chapters 1 through 4, we learned how to make some of the necessary components needed for the design and implementation of our RPG. For instance, you should have a good understanding of how to organize and arrange your player character and non-player character assets and components.

Here is a breakdown of the chapter:

- GameMaster
 - Managing game settings and scenes
 - Managing scenes
- Improving GameMaster
 - Level controller
 - Audio controller
- Player data management
 - PC class enhancements
 - Character customization class update
- Changes to the UI controller
- Testing

In this chapter, we are going to make more adjustments and updates to everything we have done so far.

GameMaster

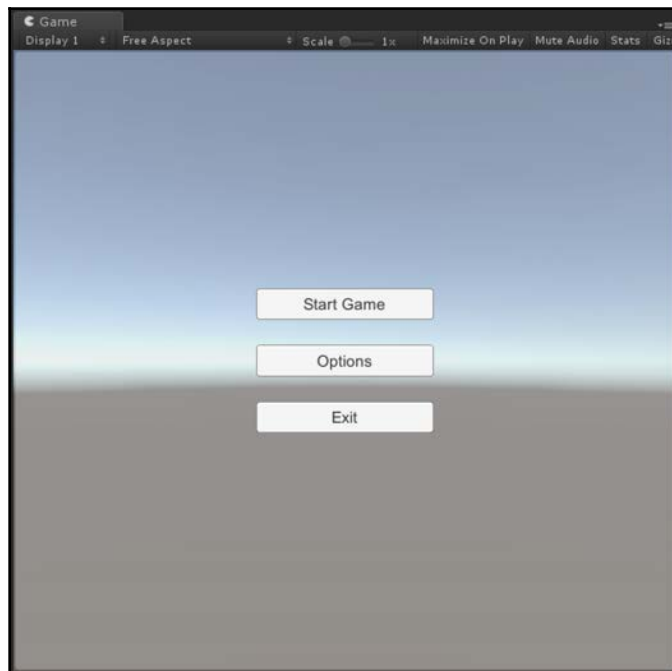
Even though we have created a `GameMaster.cs` script, we have not really utilized it to manage our game. We have created bits and pieces of our game assets and used them to do quick testing. Now is the time to start looking at how to combine everything together and create a better game manager for our RPG.

There are a few things that I want `GameMaster.cs` to perform. These are as follows:

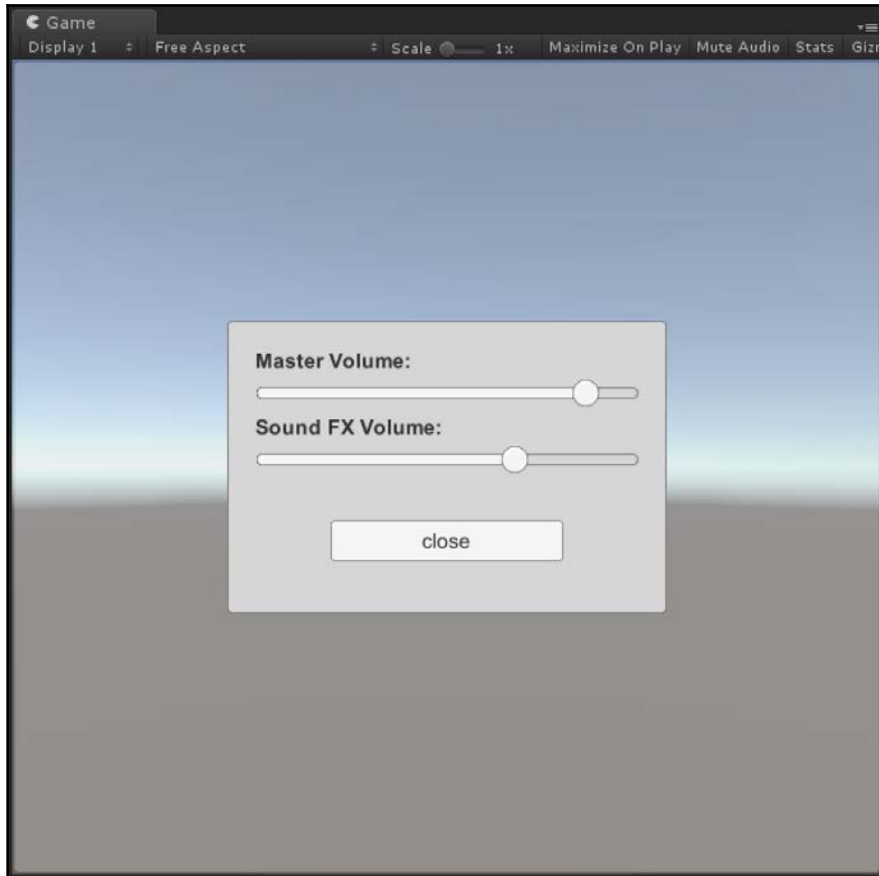
- Having a reference to the UI controller for each particular scene
- Having a reference to the player character in the scene
- Having a reference to the non-player character(s) in the scene
- Having a reference to the audio source for control
- There should always be one instance of the `GameMaster` class available

As we create our `GameMaster`, we will add or subtract some of the elements as we see fit. Let's start by integrating the User Interface with the `GameMaster`.

Open up your **Main Menu** scene. It should look something like the following screenshot:

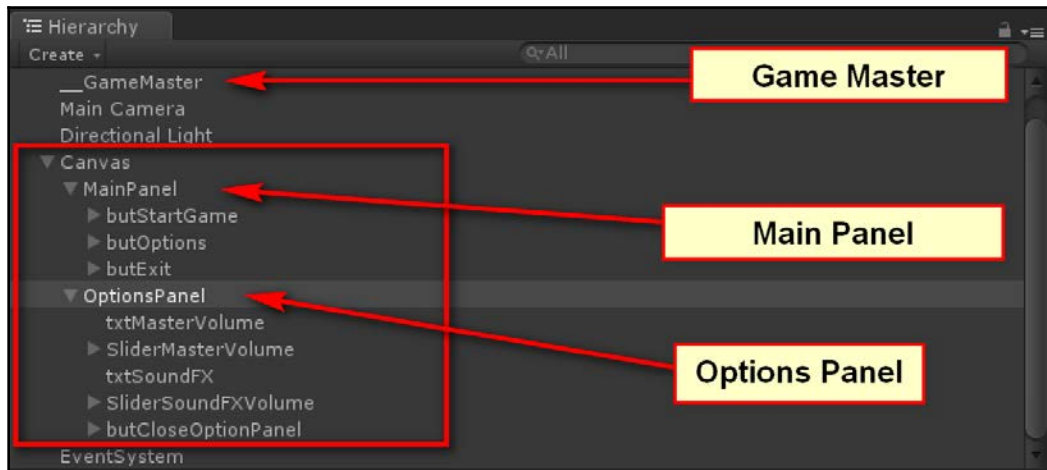


The following screenshots display the main menu we have designed for our RPG:



It is pretty simple, just to illustrate the concept. When the player clicks the **Options** button, they will get the **Options** window, where they will have the ability to adjust the volume within the game. When they are done, they will click the **close** button and go back to the main menu.

Here is a screenshot of the **Hierarchy** window for the **Main Menu** scene:



Managing game settings and audio

Create an Empty **GameObject** and name it `uiController`. We now need to create a UI Controller script that will handle the user interaction. Create a new C# script and name it `UiController.cs`.



Note: The scripts in this chapter will be updated and modified as we progress.

A listing of the UI controller is as follows:

```
using UnityEngine;
using UnityEngine.UI;

namespace com.noorcon.rpg2e
{
    public class UiController : MonoBehaviour
    {
        public RectTransform OptionsPanel;
        public Slider ControlMainVolume;
        public Slider ControlFXVolume;

        public void Update()
        {

```

```
    }

    public void DisplaySettings()
    {
        GameManager.instance.DisplaySettings =
        !GameManager.instance.DisplaySettings;
        OptionsPanel.gameObject.SetActive (GameManager.instance.DisplaySettings);
    }

    public void MainVolume()
    {
        GameManager.instance.MasterVolume (ControlMainVolume.value);
    }
    public void FXVolume()
    {
        GameManager.instance.SoundFxVolume (ControlFXVolume.value);
    }
}
}
```

Currently, we have just a few functions defined: `DisplaySettings()` and `MainVolume()`. The functions are really simple; they are referencing the UI components needed to display the settings panel, as well as to retrieve the value of the volume control slider. The information is then passed to the `GameManager.cs` script for further processing.

We need to make several changes to the `GameManager.cs` script. A listing of the code is as follows:

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace com.noorcon.rpg2e
{
    public static class SceneName
    {
        public const string MainMenu = "MainMenu";
        public const string CharacterCustomization =
        "CharacterCustomization";
        public const string Level_1 = "Awakening";
    }

    public class GameManager : MonoBehaviour
    {
        public static GameManager instance;

        // let's have a reference to the player character
```

```
// and start position of player character
public GameObject Pc;
public GameObject StartPosition;
public GameObject CharacterCustomization;

// let's have a reference to the current scene/level
public Scene CurrentScene;

// Ref to UI Elements ...
public bool DisplaySettings = false;
public UiController uiController;

public int Level = 0;

// initial audio levels for background and
// sound FX
public float AudioLevel = 0.33f;
public float FxLevel = 0.33f;

void Awake()
{
    // simple singleton
    if (instance == null)
    {
        instance = this;
    }
    else if (instance != this)
    {
        Destroy(this);
    }

    // keep the game object when moving from
    // one scene to the next scene
    DontDestroyOnLoad(this);
}

// Use this for initialization
void Start()
{
    // let's find a reference to the UI controller of the loaded scene
    if (GameObject.FindGameObjectWithTag("UI") != null)
    {
        instance.uiController =
        GameObject.FindGameObjectWithTag("UI").GetComponent<UiController>();
    }

    instance.uiController.OptionsPanel.gameObject.SetActive(instance.DisplaySettings);
}
```

```
    }

    // Update is called once per frame
    void Update()
    {

    }

    public void MasterVolume(float volume)
    {
        instance.AudioLevel = volume;
        instance.GetComponent().volume = instance.AudioLevel;
    }

    public void StartGame()
    {
        // NOTE: Start the game, load the scene that allows the player
        // to customize their character
        SceneManager.LoadScene(SceneName.CharacterCustomization);
    }
}
}
```

This code needs a little bit of explanation. The first and most important concept to take away is the concept of a Singleton. This is done by first defining a static variable, which will be used to hold our `GameMaster` instance:

```
public static GameMaster instance;
```

Then, in our `Awake()` function, we need the following code:

```
void Awake()
{
    // simple singleton
    if (instance == null)
    {
        instance = this;
    }
    else if (instance != this)
    {
        Destroy(this);
    }
    // keep the game object when moving from
    // one scene to the next scene
    DontDestroyOnLoad(this);
}
```

In the `Awake()` function, we are checking to see if the `instance` variable has been initialized. It sets the `instance` variable once. The next check ensures that we always have one instance. In other words, if the `GameMaster` object gets instantiated a second time by mistake, it will destroy it. The last line of code, `DotDestroyOnLoad()`, will ensure that the `GameObject` does not get destroyed when we move from one scene to the next.

In the `Start()` function, we are checking to see if `uiController` is present, and if it is, we get a reference to it. Once we have a reference to `uiController`, we make sure that the **Settings** panel is disabled by default, that is, hidden.

The `MasterVolume()` function gets called from the `UIController.cs` script, which then passes the actual value from the slider defined to control the volume of the background music.

Managing scenes

The next item I want to make is to have the `GameMaster` control loading the different scenes for the game. Let's look at how `GameMaster.cs` will look with the new addition of scene management:

```
using UnityEngine;
using UnityEngine.SceneManagement;
namespace com.noorcon.rpg2e
{
    public static class SceneName
    {
        public const string MainMenu = "MainMenu";
        public const string CharacterCustomization =
            "CharacterCustomization";
        public const string Level_1 = "Awakening";
    }
    [RequireComponent(typeof(AudioSource))]
    public class GameMaster : MonoBehaviour
    {
        public static GameMaster instance;
        // let's have a reference to the player character
        // and start position of player character
        public GameObject Pc;
        public GameObject StartPosition;
        public GameObject CharacterCustomization;
        // let's have a reference to the current scene/level
        public Scene CurrentScene;
        // Ref to UI Elements ...
        public bool DisplaySettings = false;
```



```
public UiController Ui;
public int Level = 0;
// initial audio levels for background and
// sound FX
public float AudioLevel = 0.33f;
public float FxAudioLevel = 0.33f;
void Awake()
{
    // simple singleton
    if (instance == null)
    {
        instance = this;
    }
    else if (instance != this)
    {
        Destroy(this);
    }
    // keep the game object when moving from
    // one scene to the next scene
    DontDestroyOnLoad(this);
}
// for each level/scene that has been loaded
// do some of the preparation work
void OnLevelWasLoaded()
{
    instance.CurrentScene = SceneManager.GetActiveScene();
    if (instance.CurrentScene.name.Equals(SceneName.CharacterCustomization))
    {
        if (GameObject.FindGameObjectWithTag("BASE") != null)
        {
            instance.CharacterCustomization = GameObject.FindGameObjectWithTag("BASE")
            as GameObject;
        }
    }
    // If we are at any other scene except character customization
    // let's go ahead and get reference to player and player
    // stat position
    if (!this.CurrentScene.name.Equals(SceneName.CharacterCustomization))
    {
        // let's get a reference to our player character
        if (instance.Pc == null)
        {
            if (GameObject.FindGameObjectWithTag("Player") != null)
            {
                instance.Pc = GameObject.FindGameObjectWithTag("Player") as GameObject;
            }
        }
    }
}
```

```
if (GameObject.FindGameObjectWithTag("START_POSITION") != null)
...

```

We have already discussed what the `Awake()` function is doing, so let's take a look at the next important function, `OnLevelWasLoaded()`. Take a look at the following code:

```
public void MasterVolume(float volume)
{
    instance.AudioLevel = volume;
    instance.GetComponent<AudioSource>().volume = instance.AudioLevel;
}
public void SoundFxVolume(float volume)
{
    instance.FxAudioLevel = volume;
}
public void StartGame()
{
    // NOTE: Start the game, load the scene that allows the player
    // to customize their character
    SceneManager.LoadScene(SceneName.CharacterCustomization);
}
public void LoadLevel()
{
    switch (instance.Level)
    {
        // load level 1
        case 1:
        {
            instance.Pc = GameObject.FindGameObjectWithTag("Player") as GameObject;
            SceneManager.LoadScene(SceneName.Level_1);
            break;
        }
    }
}
}
}
}
}
}

```

The `OnLevelWasLoaded()` function is called by Unity after the scene was loaded. We are using this function in the `GameMaster` script to perform a few tasks. The first thing we do is get the current scene we are in. This information will be used later to determine what `GameMaster` will do.

We check to see if we are in the character customization scene. This is where the player can customize the PC before they start playing the game. If we are in the character customization scene, we want to get a reference to the `Base GameObject` in the scene. If you recall, the `Base GameObject` has the `CharacterCutomization.cs` script attached to it, which is used to, well, customize the character.

If we are in any other scene, then we want to get a reference to the player character, as well as the starting position of the player character at the beginning of the scene, if there is one.

We then use the `DetermineLevel()` function to determine the level we are currently on, to make some more configurations.

The two functions currently implemented for starting the game and loading the levels are handled by the `StartGame()` function and the `LoadLevel()` function.

```
public static class SceneName
{
    public const string MainMenu = "MainMenu";
    public const string CharacterCustomization = "CharacterCustomization";
    public const string Level_1 = "Awakening";
}
```

The `SceneName` class is designed to make it easier to refer to the scene names in the C# code. This makes it easier to change the actual scene name within the project, but have a consistent call name in the code.

This is all good so far, but we can try to make it better.

Improving GameMaster

The code we have so far works, but it is not very clean. Let's go ahead and structure the code a little better. Let's create a new script, called `GameLevelController.cs`. This new script will be handling the logic for our level management.

Level controller

A listing of `GameLevelController.cs` is as follows:

```
using UnityEngine;
using UnityEngine.SceneManagement;
namespace com.noorcon.rpg2e
{
```

```
public static class SceneName
{
    public const string MainMenu = "MainMenu";
    public const string CharacterCustomization = "CharacterCustomization";
    public const string Level_1 = "Awakening";
}

public class GameLevelController
{
    // let's have a reference to the current scene/level
    public Scene CurrentScene
    {
        get { return SceneManager.GetActiveScene(); }
    }
    // keep the numerical level value
    public int Level = 0;
    public void OnLevelWasLoaded()
    {
        // if we are in the character customization scene,
        // let's get a reference to the Base game object for future use.
        if (CurrentScene.Equals(SceneName.CharacterCustomization))
        {
            if (GameObject.FindGameObjectWithTag("Base") != null)
            {
                GameManager.instance.CharacterCustomization =
                GameObject.FindGameObjectWithTag("Base") as GameObject;
            }
        }
        // If we are at any other scene except character customization
        // let's go ahead and get reference to player and player
        // stat position
        if (CurrentScene.name.Equals(SceneName.CharacterCustomization))
        {
            // let's get a reference to our player character
            if (GameManager.instance.Pc == null)
            {
                if (GameObject.FindGameObjectWithTag("Player") != null)
                {
                    GameManager.instance.Pc = GameObject.FindGameObjectWithTag("Player") as
                    GameObject;
                }
            }
            if (GameObject.FindGameObjectWithTag("StartPosition") != null)
            {
                GameManager.instance.StartPosition =
                GameObject.FindGameObjectWithTag("StartPosition") as GameObject;
            }
            if (GameManager.instance.StartPosition != null && GameManager.instance.Pc !=
            null)
```

```

{
    GameMaster.instance.Pc.transform.position =
    GameMaster.instance.StartPosition.transform.position;
    GameMaster.instance.Pc.transform.rotation =
    GameMaster.instance.StartPosition.transform.rotation;
}
}
// determine what level we are on
DetermineLevel();
}
// this function will set a numerical value for our levels
private void DetermineLevel()
{
    switch (CurrentScene.name)
    {
        case SceneName.MainMenu:
        case SceneName.CharacterCustomization:
        {
            Level = 0;
            break;
        }
        case SceneName.Level_1:
        {
            Level = 1;
            GameMaster.instance.Pc.GetComponent<IKHandle>().enabled = true;
            break;
        }
        default:
        {
            Level = 0;
            break;
        }
    }
}
.....

```

What I have done is basically move all of the code that deals with level management into `GameLevelController.cs`. Our `GameMaster` script drives the `LevelController` class. We will see this a bit later.

Audio controller

The next code clean-up I want to do is for the audio. Let's create a new script, called `GameAudioController.cs`. The code for the new script is as follows:

```
using UnityEngine;
```

```
namespace com.noorcon.rpg2e
{
public class GameAudioController : MonoBehaviour
{
// initial audio levels for background and
// sound FX
public float AudioLevel = 0.33f;
public float FxAudioLevel = 0.33f;
public AudioSource audioSource;
public void SetDefaultVolume()
{
audioSource.volume = AudioLevel;
}
public void MasterVolume(float volume)
{
AudioLevel = volume;
audioSource.volume = AudioLevel;
}
public void SoundFxVolume(float volume)
{
FxAudioLevel = volume;
}
}
}
```

The code is pretty straightforward. Now, let's take a look at how `GameMaster.cs` looks:

```
using UnityEngine;
namespace com.noorcon.rpg2e
{
[RequireComponent(typeof(AudioSource))]
public class GameMaster : MonoBehaviour
{
public static GameMaster instance;
// let's have a reference to the player character
// and start position of player character
public GameObject Pc;
public GameObject StartPosition;
public GameObject CharacterCustomization;
public GameLevelController LevelController;
public GameAudioController AudioController;
// Ref to UI Elements ...
public bool DisplaySettings = false;
public UiController Ui;
void Awake()
{
// simple singleton
if (instance == null)
```

```
{
instance = this;
// initialize level controller
instance.LevelController = new GameLevelController();
// initialize audio controller
instance.AudioController = new GameAudioController();
instance.AudioController.audioSource =
instance.GetComponent();
instance.AudioController.SetDefaultVolume();
}
else if (instance != this)
{
Destroy(this);
}
// keep the game object when moving from
// one scene to the next scene
DontDestroyOnLoad(this);
}
// for each level/scene that has been loaded
// do some of the preparation work
void OnLevelWasLoaded()
{
instance.LevelController.OnLevelWasLoaded();
}
// Use this for initialization
void Start()
{
// let's find a reference to the UI controller of the loaded scene
if (GameObject.FindGameObjectWithTag("Ui") != null)
{
instance.Ui =
GameObject.FindGameObjectWithTag("Ui").GetComponent<UiController>();
}
instance.Ui.OptionsPanel.gameObject.SetActive(instance.DisplaySettings);
}
// Update is called once per frame
void Update()
{
}
public void MasterVolume(float volume)
{
instance.AudioController.MasterVolume(volume);
}
public void SoundFxVolume(float volume)
{
instance.AudioController.SoundFxVolume(volume);
}
public void StartGame()
```

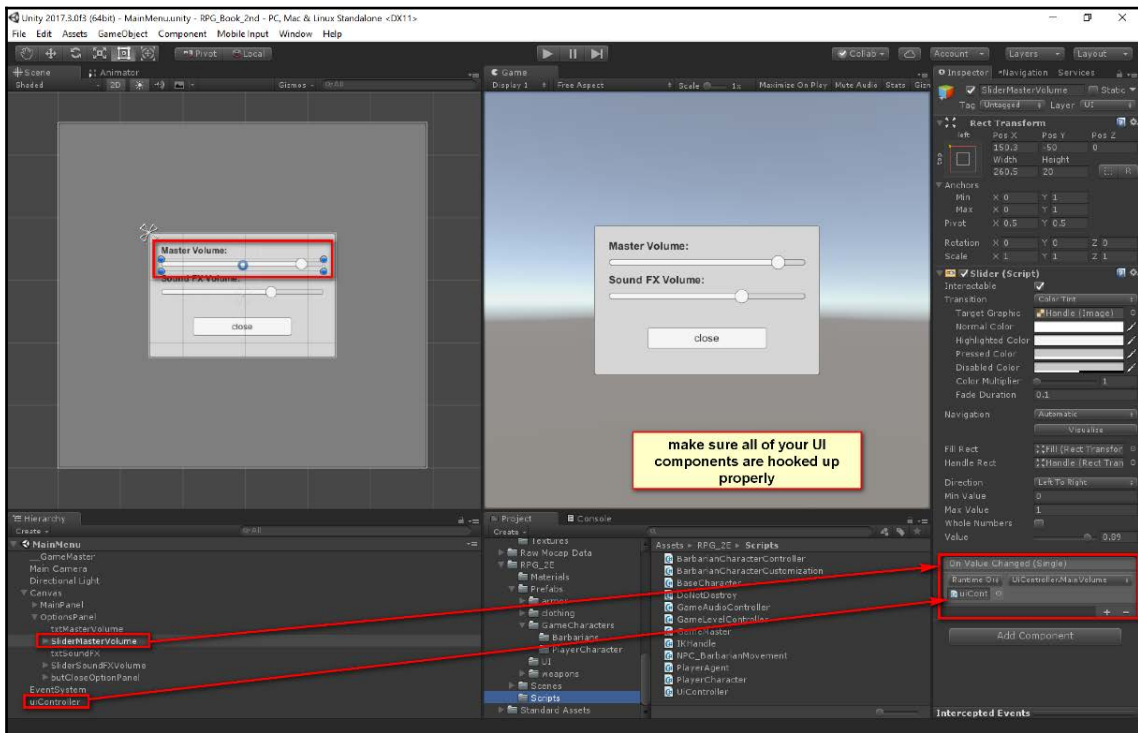
```

{
instance.LoadLevel();
}
public void LoadLevel()
{
instance.LevelController.LoadLevel();
}
}
}

```

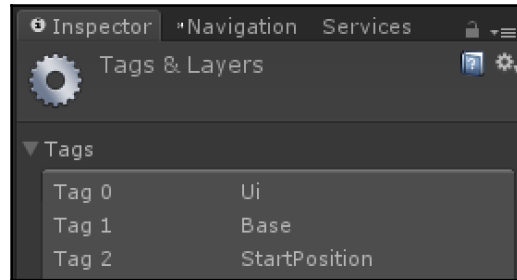
As you can see, the code is easier to read, and it is also structured better. GameMaster is using the controllers to perform each specific task. This also makes it easier to maintain code for different tasks within our game. For instance, all of the audio-related code can be now implemented in the controller, and so on.

For all this to work, you have to make sure that you have properly hooked up your `uiController` `GameObject` to communicate with the `UiController.cs` class when the user interacts with the Options/Settings menu, as shown in the following screenshot:



UI element events

Your project should also have the following **Tags** defined by now:



These are used within the C# code to identify GameObjects at runtime.

Player data management

We have not saved the actual data representing the customization of our player. The next step is to enhance our `PlayerCharacter.cs` and `BarbarianCharacterCustomization.cs` scripts to actually save the selected data in our PC object.

PC class enhancements

To do this, we need to modify our `PlayerCharacter.cs` code. The new listing is as follows:

```
using System;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class PlayerCharacter : BaseCharacter
    {
        public enum ShoulderPad
        {
            none = 0,
            SP01 = 1,
            SP02 = 2,
            SP03 = 3,
            SP04 = 4
        };
        public enum BodyType { normal = 1, BT01 = 2, BT02 = 3
    };
};
```

```
// Shoulder Pad
public ShoulderPad selectedShoulderPad = ShoulderPad.none;
public BodyType selectedBodyType = BodyType.normal;
public bool kneePad = false;
public bool legPlate = false;
public enum WeaponType
{
    none = 0,
    axe1 = 1,
    axe2 = 2,
    club1 = 3,
    club2 = 4,
    falchion = 5,
    gladius = 6,
    mace = 7,
    maul = 8,
    scimitar = 9,
    spear = 10,
    sword1 = 11,
    sword2 = 12,
    sword3 = 13
};
public WeaponType selectedWeapon = WeaponType.none;
public enum HelmetType { none = 0, HL01 = 1, HL02 = 2, HL03 = 3, HL04 = 4 };
public HelmetType selectedHelmet = HelmetType.none;
public enum ShieldType { none = 0, SL01 = 1, SL02 = 2 };
public ShieldType selectedShield = ShieldType.none;
public int SKIN_ID = 1;
public enum ShoeType { none = 0, BT01 = 1, BT02 = 2 };
public ShoeType selectedBoot = ShoeType.none;
}
```

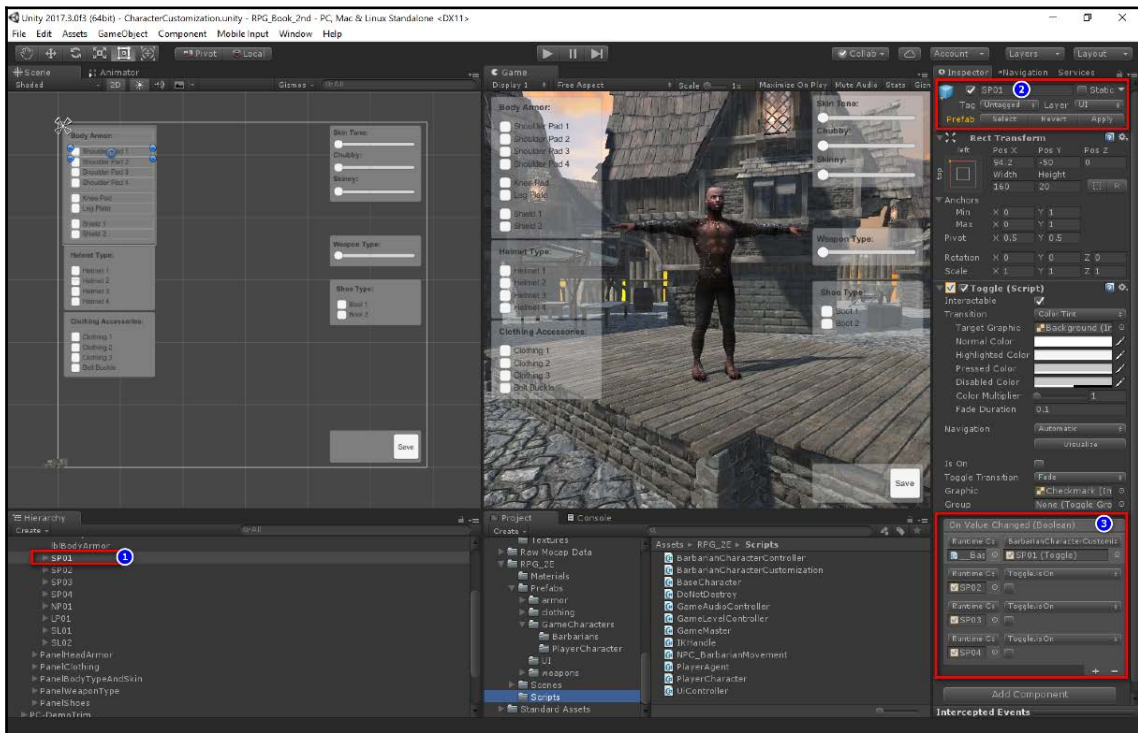
We have defined several enumeration types that describe the different parts of the player character's customization. There are several advantages to using enumeration in our code, a few of them being named constants, the name describes what they are for, type safety, and it is easier to change the value of the enumeration without having to check a hundred different places within your code.

As stated in previous chapters, the character customization code is heavily related to your character model and how you have rigged up your character model to be used in the game.



You will need to modify the name of your UI elements to match the new code.

Take a look at the following screenshot:



Hooking up UI element events

There are a few things you will need to configure to make sure the code works properly. First, you will need to name your UI elements properly to match the enumeration. The preceding screenshot illustrates one of the UI elements representing a shoulder pad.

Character customization class update

The events that drive character customization are attached to the Base prefab, which has the `CharacterCustomization.cs` script as a component. The `CharacterCustomization.cs` script is as follows:

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace com.noorcon.rpg2e
{
    public class BarbarianCharacterCustomization : MonoBehaviour
    {
        public GameObject PLAYER_CHARACTER;

        public PlayerCharacter PlayerCharacterData;

        public Material[] PLAYER_SKIN;

        ...

        // Use this for initialization
        void Start()
        {
            PlayerCharacterData =
            PLAYER_CHARACTER.GetComponent<PlayerAgent>().playerCharacterData;
        }

        public bool ROTATE_MODEL = false;

        // Update is called once per frame
        void Update()
        {
            if (Input.GetKeyUp(KeyCode.R))
            {
                ROTATE_MODEL = !ROTATE_MODEL;
            }

            if (ROTATE_MODEL)
            {
                PLAYER_CHARACTER.transform.Rotate(new Vector3(0, 1, 0), 33.0f *
                Time.deltaTime);
            }

            if (Input.GetKeyUp(KeyCode.L))
            {

```

```
        Debug.Log(PlayerPrefs.GetString("Name"));
    }

}

void DisableShoulderPads()
{
    SHOULDER_PAD_R_01LOD0.SetActive(false);
    SHOULDER_PAD_R_02LOD0.SetActive(false);
    SHOULDER_PAD_R_03LOD0.SetActive(false);
    SHOULDER_PAD_R_04LOD0.SetActive(false);

    SHOULDER_PAD_L_01LOD0.SetActive(false);
    SHOULDER_PAD_L_02LOD0.SetActive(false);
    SHOULDER_PAD_L_03LOD0.SetActive(false);
    SHOULDER_PAD_L_04LOD0.SetActive(false);
}

public void SetShoulderPad(Toggle id)
{
    try
    {
        PlayerCharacter.ShoulderPad name
            =
            (PlayerCharacter.ShoulderPad) Enum.Parse(typeof(PlayerCharacter.ShoulderPad)
, id.name, true);
        if (id.isOn)
        {
            PlayerCharacterData.SelectedShoulderPad = name;
        }
        else
        {
            PlayerCharacterData.SelectedShoulderPad
                = PlayerCharacter.ShoulderPad.none;
        }
    }
    catch
    {
        // if the value passed is not in the enumeration set it to none
        PlayerCharacterData.SelectedShoulderPad
            = PlayerCharacter.ShoulderPad.none;
    }

    // disable before new selection
    DisableShoulderPads();

    switch (id.name)
```

```

        {
            case "SP01":
            {
                SHOULDER_PAD_R_01LOD0.SetActive(id.isOn);
                SHOULDER_PAD_L_01LOD0.SetActive(id.isOn);
                break;
            }
        }
        ...
        case "SP04":
        {
            SHOULDER_PAD_R_04LOD0.SetActive(id.isOn);
            SHOULDER_PAD_L_04LOD0.SetActive(id.isOn);
            break;
        }
    }
}
...

```

In the code listed, what we have done is add a new variable of type `PlayerCharacter` named `PlayerCharacterData`. The `PlayerCharacter` class is the player character class we have defined and enhanced to contain the data for our player character as follows:

```

public void SetKneePad(Toggle id)
{
    KNEE_PAD_R_LOD0.SetActive(id.isOn);
    KNEE_PAD_L_LOD0.SetActive(id.isOn);
}

public void SetLegPlate(Toggle id)
{
    LEG_PLATE_R_LOD0.SetActive(id.isOn);
    LEG_PLATE_L_LOD0.SetActive(id.isOn);
}

void DisableWeapons()
{
    AXE_01LOD0.SetActive(false);
    AXE_02LOD0.SetActive(false);
    CLUB_01LOD0.SetActive(false);
    CLUB_02LOD0.SetActive(false);
    FALCHION_LOD0.SetActive(false);
    GLADIUS_LOD0.SetActive(false);
    MACE_LOD0.SetActive(false);
    MAUL_LOD0.SetActive(false);
    SCIMITAR_LOD0.SetActive(false);
    SPEAR_LOD0.SetActive(false);
    SWORD_BASTARD_LOD0.SetActive(false);
}

```

```
        SWORD_BOARD_01LOD0.SetActive(false);
        SWORD_SHORT_LOD0.SetActive(false);
    }

    public void SetWeaponType(Slider id)
    {
        try
        {
            PlayerCharacter.WeaponType weapon =
            (PlayerCharacter.WeaponType) Convert.ToInt32(id.value);

            PlayerCharacterData.SelectedWeapon = weapon;
        }
        catch
        {
            PlayerCharacterData.SelectedWeapon =
            PlayerCharacter.WeaponType.none;
        }

        // disable weapons
        DisableWeapons();

        switch (Convert.ToInt32(id.value))
        {
            case 0:
            {
                DisableWeapons();

                break;
            }
            case 1:
            {
                AXE_01LOD0.SetActive(true);
                break;
            }
            ...
            case 13:
            {
                SWORD_SHORT_LOD0.SetActive(true);
                break;
            }
        }
    }

    public void SetWeaponType(PlayerCharacter.WeaponType id)
    {
        // disable Weapons
        DisableWeapons();
    }
}
```

```

switch (Convert.ToInt32(id))
{
    case 0:
    {
        DisableWeapons();
        break;
    }
    case 1:
    {
        AXE_01LOD0.SetActive(true);
        break;
    }
    ...
    case 13:
    {
        SWORD_SHORT_LOD0.SetActive(true);
        break;
    }
}
}

```

The next logic we need to implement is to detect which option the player has selected through the character customization UI, and appropriately set the data in the `PlayerCharacter` object. See the following code:

```

void DisableHelmets()
{
    HELMET_01LOD0.SetActive(false);
    HELMET_02LOD0.SetActive(false);
    HELMET_03LOD0.SetActive(false);
    HELMET_04LOD0.SetActive(false);
}

public void SetHelmetType(Toggle id)
{
    try
    {
        PlayerCharacter.HelmetType helmet
        =
        (PlayerCharacter.HelmetType)Enum.Parse(typeof(PlayerCharacter.HelmetType),
        id.name, true);
        if (id.isOn)
        {
            PlayerCharacterData.SelectedHelmet = helmet;
        }
        else
        {
            PlayerCharacterData.SelectedHelmet

```



```
        = PlayerCharacter.HelmetType.none;
    }
}
catch
{
    // if the value passed is not in the enumeration set it to none
    PlayerCharacterData.SelectedHelmet
        = PlayerCharacter.HelmetType.none;
}

    // disable helmets
    DisableHelmets();

switch (id.name)
{
    case "HL01":
    {
        HELMET_01LOD0.SetActive(id.isOn);
        break;
    }
    case "HL02":
    {
        HELMET_02LOD0.SetActive(id.isOn);
        break;
    }
    case "HL03":
    {
        HELMET_03LOD0.SetActive(id.isOn);
        break;
    }
    case "HL04":
    {
        HELMET_04LOD0.SetActive(id.isOn);
        break;
    }
}
}

...

public void SetShieldType(PlayerCharacter.ShieldType id)
{
    switch (id.ToString())
    {
        case "SL01":
        {
            SHIELD_01LOD0.SetActive(true);
            SHIELD_02LOD0.SetActive(false);
        }
    }
}
```

```

        break;
    }
    case "SL02":
    {
        SHIELD_01LOD0.SetActive(false);
        SHIELD_02LOD0.SetActive(true);
        break;
    }
}

public void SetSkinType(Slider id)
{
    PlayerCharacterData.SkinId = Convert.ToInt32(id.value);
    SKN_LOD0.GetComponent<Renderer>().material =
    PLAYER_SKIN[System.Convert.ToInt32(id.value)];
}

...

```

The implementation concept is the same for all the different parts of the player character that can be customized. One of them is as follows:

```

public void SetClothingType(Toggle id)
{
    try
    {
        PlayerCharacter.ClothingType clothing
        =
        (PlayerCharacter.ClothingType)Enum.Parse(typeof(PlayerCharacter.ClothingType), id.name, true);
        if (id.isOn)
        {
            PlayerCharacterData.selectedClothing = clothing;
        }
        else
        {
            PlayerCharacterData.selectedClothing
            = PlayerCharacter.ClothingType.none;
        }
    }
    catch
    {
        // if the value passed is not in the enumeration set it to
        none
        PlayerCharacterData.selectedClothing
        = PlayerCharacter.ClothingType.none;
    }
}

```

```
switch (id.name)
{
    case "CT01":
    {
        CLOTH_01LOD0.SetActive(id.isOn);
        CLOTH_02LOD0.SetActive(false);
        CLOTH_03LOD0.SetActive(false);
        break;
    }
    case "CT02":
    {
        CLOTH_01LOD0.SetActive(false);
        CLOTH_02LOD0.SetActive(id.isOn);
        CLOTH_03LOD0.SetActive(false);
        break;
    }
    case "CT03":
    {
        CLOTH_01LOD0.SetActive(false);
        CLOTH_02LOD0.SetActive(false);
        CLOTH_03LOD0.SetActive(id.isOn);
        break;
    }
    case "CT04":
    {
        BELT_LOD0.SetActive(id.isOn);
        break;
    }
}
```

The preceding code is for the customization of the body type of the player character. The first thing it tries to do is to parse and convert the value passed to the function by the UI component. Next it sets the `selectedClothing` variable in the `PlayerCharacter` object. If, for some reason, the value passed does not exist in the enumeration, we will assign the default value to the `selectedClothing` variable. There are also debug statements to give you feedback about the current value.

Changes to the game level controller

The game level controller will also need to be updated now, to make the necessary changes to the GameMaster object. We will need to update the `LoadLevel()` function as follows:

```
public void LoadLevel()
{
    switch (GameMaster.instance.LevelController.Level)
    {
        case 0:
        {
            SceneManager.LoadScene(SceneName.CharacterCustomization);
            break;
        }

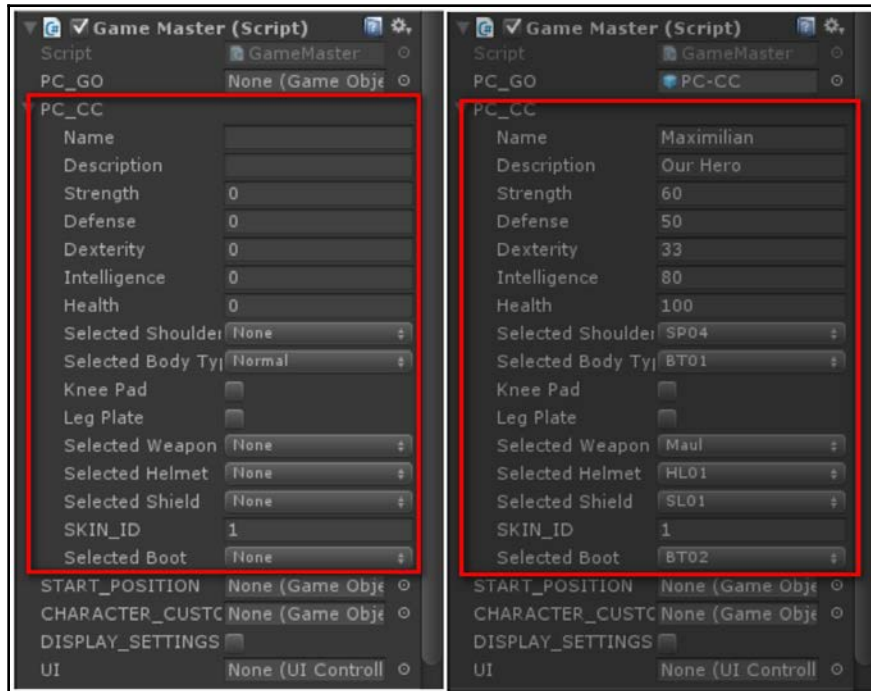
        // load level 1
        case 1:
        {
            GameMaster.instance.PlayerCharacterGameObject =
            GameObject.FindGameObjectWithTag("Player") as GameObject;
            SceneManager.LoadScene(SceneName.Level_1);
            break;
        }
    }
}
```

This will make sure that GameMaster is updated with the proper player character data. Let's go ahead and test the code.

Testing

Starting from the Main Menu scene, make sure that you have the following GameObjects in the scene: `uiController` and `_GameMaster`. The `uiController` GameObject should have `UiController.cs` attached, and `_GameMaster` should have the following components attached: `GameMaster.cs` and an `AudioSource` component that will be used for the background music.

Have the `_GameMaster` GameObject selected in the **Hierarchy** window, and run the game. Select the **Start Game** button. This will load the character customization scene. The `_GameMaster` GameObject should still be selected. If not, go ahead and select it from the **Hierarchy** window, do some of the character customization, and click the **Save** button. Refer to the following screenshot:



The first level should have been loaded with your character and the customization you have made to your character in the previous step. So visually, your character has retained all of the customization you have done, and from a data point of view, when you look at the `_GameMaster` GameObject in the **Inspector** window, you will notice that the data has been saved properly, as shown in the preceding screenshot.

Summary

This chapter was mostly code. We enhanced the `GameMaster` class to handle the game settings and scene management. We began the chapter by making `GameMaster` handle the user interface, the player character data, and the game settings, which currently is just the volume for the background music.

We added a new UI element that displays the settings panel for the game. At the moment, it only contains the main volume control. Next, we added the necessary code in the `UiController` class and the `GameMaster` class to handle the display of the settings window, as well as the slider value passed from the UI component to `UiController` to the `GameMaster` class.

We also made the `GameMaster` class into a singleton. A singleton in software engineering is a design pattern that restricts the instantiation of a class to one object. This pattern fits perfectly for `GameMaster`, as we only need to have one instance of it active at any given time throughout the lifespan of the game.

We also looked at how to perform scene management. We defined a static class named `SceneName` that contains constant string variables identifying the scene references in our game.

We then took the next step to improve `GameMaster` and the internal structure for our code. We created a new class, called `GameLevelController.cs` that handles the scene management, which in turn is driven by `GameMaster`. We practically took the logic for level handling from within the `GameMaster` class, and reworked and improved it in the `GameLevelController` class.

Next we developed an `AudioController` class that basically manages the audio for our game. This class is also driven by `GameMaster`. By this time, `GameMaster` is a lean script that manages all of the other components.

The next big challenge was how to handle the player character data. Specifically, how to save the character customization data for the player character internally, after the player has customized the character. In order to save the data, we had to modify the `PlayerCharacter.cs` class.

We created several enumerations representing each part of the character that could be customized, such as the shoulder pad, the body type, the weapon type, the helmet type and so on. We used enumeration to make it easier to reference them within the code.

This approach forced us to make some modifications to the existing character customization setup that we implemented previously. We had to update the UI components to reflect the enumeration defined for each customizable type, and we also had to modify the `BarbarianCharacterCustomization.cs` class to handle the new changes.

The `BarbarianCharacterCustomization` class implemented a `PC` type variable to keep track of the customizations and finally pass the data along to `GameMaster`. During the process, we also improved the case handling of the `BarbarianCharacterCustomization` class for default values and so on.

Finally, we did a test run of the game to double-check everything worked as designed and implemented.

We have created a lot of code in this chapter. In the next chapter, we will start building our inventory system, and yes, that is going to involve more code!

6 Inventory System

The inventory system is one of the most critical components of an RPG. It will be used to store all important game elements that the player will need in your game environment. This chapter will guide you on how to create a simple generic inventory system that can be utilized and extended as you see fit.

Here is a breakdown of the chapter:

- Designing an inventory system
 - Weighted inventory
 - Determining item types
- Creating an inventory item
 - Creating the prefab
 - Adding an inventory item agent
 - Inventory items defined as prefabs
- Designing an inventory interface
 - Creating the inventory UI framework
 - Designing a dynamic item viewer
 - Adding a scroll view
 - Adding elements to `PanelItem` and `Scroll View`
 - Adding `txtItemElement` dynamically
 - Building the final inventory item UI
- Integrating the UI with the actual inventory system
 - Hooking the category buttons and displaying the data
 - Testing the inventory system
- Inventory items and the player character
 - Applying inventory items
 - How it looks

There is a lot of work ahead of us in this chapter. Let's get started!

Designing an inventory system

As with everything else we have discussed thus far, designing your **inventory system** will also be heavily dependent on your game. There are many different types of inventory system mechanics that you can study and choose based on its relevance to your game.

Weighted inventory

I will be leaning towards implementing what is called the **weighted inventory**. In this type of inventory system, each item or piece of equipment is assigned a numerical value that represents the weight of the item. This, in return, is used to determine how much inventory the player can carry at any given time during game play. This makes sense for our RPG, if you think about it.

Consider the following as an example: assume you are a hiker who wants to climb Mount Ararat. The climb itself will take a certain amount of time. During the climb, you will need to carry with you the necessary equipment to complete the journey or climb. Realistically, there are several crucial items that you, as the hiker, will need to carry with you, such as those shown in the following list:

- Clothing
- Tents
- Sleeping bags
- Boots
- Ice breakers
- Food
- Light source
- Personal items

Each one of the categories listed above has a specific weight associated to it in real life. Therefore, when you are planning your hike, you will need to plan ahead and see how you can meet your climbing needs, while in the meantime also reducing the amount of items and the total weight of the items you will need to carry on your back during the journey. The actual logistics are a little more involved, but you get the picture.

It is no different in our RPG. The player character can only carry a certain number of items and/or equipment with them for their journey. For instance, the player character cannot carry twenty different types of weapon at any given time! It would be just impossible, realistically speaking. So it would be a nice touch to put some realism into the game play.

Also, just like in real life, the heavier the equipment one has to carry, the more energy it will take. So, we can also incorporate such a system for our game. For instance, carrying too many weapons will have a major effect on the player character over a long period of time. First of all, it will reduce its speed and movement drastically; secondly, it can have a major impact on the health of the player. This is where your creativity and design skills will come into play. You are the master of the game, and you determine how you want to implement it!

I am going to keep it simple for the sake of demonstration!

Determining item types

For starters, we will concentrate on some of the basic item types that we would like to define in our game, such as weapons, armor, and clothing. On top of this, we can also add health packets, potions, and collectibles.

We will create three new scripts named `BaseItem.cs`, `InventoryItem.c`, and `InventorySystem.cs`. The `BaseItem` class will hold the generic properties for all items, just like the `BaseCharacter` class we defined previously. The `InventoryItem` class will inherit the `BaseItem` class and define the item type.

A listing of `BaseItem.cs` is as follows:

```
using System;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class BaseItem
    {
        public enum ItemCategory
        {
            Weapon = 0,
            Armour = 1,
            Clothing = 2,
            Health = 3,
            Potion = 4
        }
        [SerializeField]
```

```
private string name;
[SerializeField]
private string description;
public string Name
{
    get { return name; }
    set { name = value; }
}
public string Description
{
    get { return description; }
    set { description = value; }
}
}
```

The main idea in the preceding code is `ItemCateogry`. At the moment, I have kept it to only five different types of category that the inventory would keep track of.



A category could have multiple item types. For instance, there are different types of weapons and so on.

A listing of `InventoryItem.cs` is as follows:

```
using System;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class InventoryItem : BaseItem
    {
        [SerializeField]
        private ItemCatrgory category;
        [SerializeField]
        private float strength;
        [SerializeField]
        private float weight;
        public ItemCatrgory Category
        {
            get { return category; }
            set { category = value; }
        }
        public float Strength
        {
            get { return strength; }
        }
    }
}
```

```
set { strength = value; }  
}  
public float Weight  
{  
get { return weight; }  
set { weight = value; }  
}  
}  
}
```

The preceding code implements more properties or attributes for the items to be used in the inventory. For now, let's just keep it the way it is; we can always change it in the future.

The next important script is the actual script that will be used to manage the inventory. There are many ways to implement the logic for the inventory system. Again keeping things simple, the current script will have five `List` datatypes of type `InventoryItem`, one for each item category.

A listing of `InventorySystem.cs` is as follows:

```
using System.Collections.Generic;  
using UnityEngine;  
namespace com.noorcon.rpg2e  
{  
public class InventorySystem  
{  
[SerializeField]  
private List<InventoryItem> weapons = new List<InventoryItem>();  
[SerializeField]  
private List<InventoryItem> armour = new List<InventoryItem>();  
[SerializeField]  
private List<InventoryItem> clothing = new List<InventoryItem>();  
[SerializeField]  
private List<InventoryItem> health = new List<InventoryItem>();  
[SerializeField]  
private List<InventoryItem> potion = new List<InventoryItem>();  
private InventoryItem selectedWeapon;  
private InventoryItem selectedArmour;  
public InventoryItem SelectedWeapon  
{  
get { return selectedWeapon; }  
set { selectedWeapon = value; }  
}  
public InventoryItem SelectedArmour  
{  
get { return selectedArmour; }  
set { selectedArmour = value; }  
}}
```

```
}
public InventorySystem()
{
    ClearInventory();
}
public void ClearInventory()
{
    weapons.Clear();
    armour.Clear();
    clothing.Clear();
    health.Clear();
    potion.Clear();
}
// this function will add an inventory item
public void AddItem(InventoryItem item)
{
    switch (item.Category)
    {
    case BaseItem.ItemCatrgory.Armour:
    {
        armour.Add(item);
        break;
    }
    case BaseItem.ItemCatrgory.Clothing:
    {
        clothing.Add(item);
        break;
    }
    case BaseItem.ItemCatrgory.Health:
    {
        health.Add(item);
        break;
    }
    case BaseItem.ItemCatrgory.Potion:
    {
        potion.Add(item);
        break;
    }
    case BaseItem.ItemCatrgory.Weapon:
    {
        weapons.Add(item);
        break;
    }
    }
}
```

We won't have direct access to the lists that will be used to contain the inventory items. For now, we have implemented two functions, `AddItem()` and `DeleteItem()`, which will handle the two basics features of the inventory, adding an item to it and removing an item from it. These two functions will take an `InventoryItem` object, and based on `ItemCategory`, be added or removed from the appropriate list within the inventory, as shown in the following code:

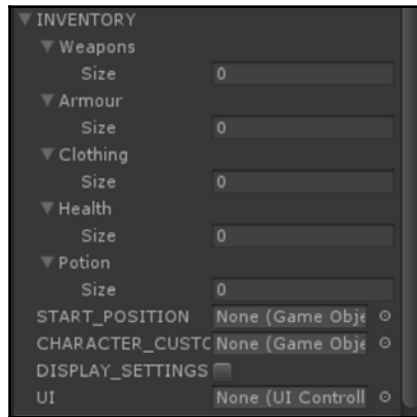
```
// this function will remove an inventory item
public void DeleteItem(InventoryItem item)
{
    switch (item.Category)
    {
        case BaseItem.ItemCatrgory.Armour:
        {
            armour.Remove(item);
            break;
        }
        case BaseItem.ItemCatrgory.Clothing:
        {
            clothing.Remove(item);
            break;
        }
        case BaseItem.ItemCatrgory.Health:
        {
            health.Remove(item);
            break;
        }
        case BaseItem.ItemCatrgory.Potion:
        {
            potion.Remove(item);
            break;
        }
        case BaseItem.ItemCatrgory.Weapon:
        {
            weapons.Remove(item);
            break;
        }
    }
}
```

The basics are in place. Now we will need to integrate this with the `GameMaster.cs` script. To do so, we will need to create a new variable of type `InventorySystem` named `INVENTORY`, initialized in the `Awake()` function of the `GameMaster.cs` script.

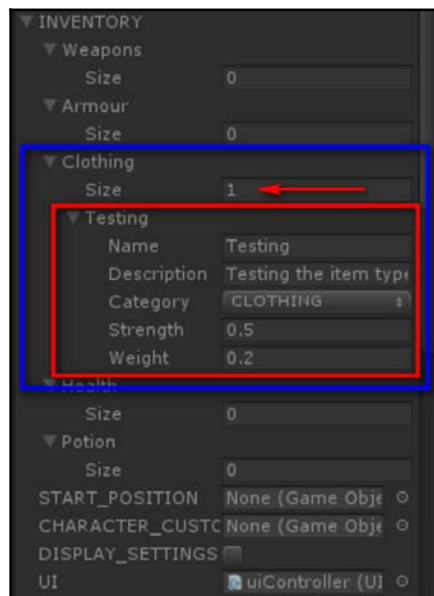
The following listing illustrates just the new addition:

```
public static GameManager instance;
// let's have a reference to the player character
// and start position of player character
public GameObject PlayerCharacterGameObject;
public GameObject StartPosition;
public GameObject CharacterCustomization;
public PlayerCharacter PlayerCharacterData;
public InventorySystem Inventory;
public GameLevelController LevelController;
public GameAudioController AudioController;
// Ref to UI Elements ...
public bool DisplaySettings = false;
public UiController Ui;
void Awake()
{
    // simple singleton
    if (instance == null)
    {
        instance = this;
        // initialize level controller
        instance.LevelController = new GameLevelController();
        // initialize audio controller
        instance.AudioController = new GameAudioController();
        instance.AudioController.audioSource =
        instance.GetComponent();
        instance.AudioController.SetDefaultVolume();
        // initialize Inventory System
        instance.Inventory = new InventorySystem();
        InventoryItem tmpInvItem = new InventoryItem();
        tmpInvItem.Category = BaseItem.ItemCategory.Clothing;
        tmpInvItem.Name = "Testing";
        tmpInvItem.Description = "Testing clothing item type";
        tmpInvItem.Strength = 0.5f;
        tmpInvItem.Weight = 0.2f;
        instance.Inventory.AddItem(tmpInvItem);
    }
    else if (instance != this)
    {
        Destroy(this);
    }
    // keep the game object when moving from
    // one scene to the next scene
    DontDestroyOnLoad(this);
}
```

Notice that we are actually creating an `InventoryItem` object and inserting it into `InventorySystem` for testing purposes. Another great feature is that you can see `InventorySystem` during design time since we have serialized the classes and the fields. Take a look at the following screenshot:



The preceding screenshot displays the inventory system as seen in the **Inspector Window** when you select the `GameManager` object. When you run the game to test it, you will see the following update:



Notice how the data reflects appropriately in the inventory system, as expected! The **CLOTHING** list has increased its size to 1, and the `InventoryItem` object within the list is properly stored and displayed for testing and debugging. We have one clothing item named **Testing**, with the given description, a **Strength** of 0.5, and a **Weight** of 0.2.

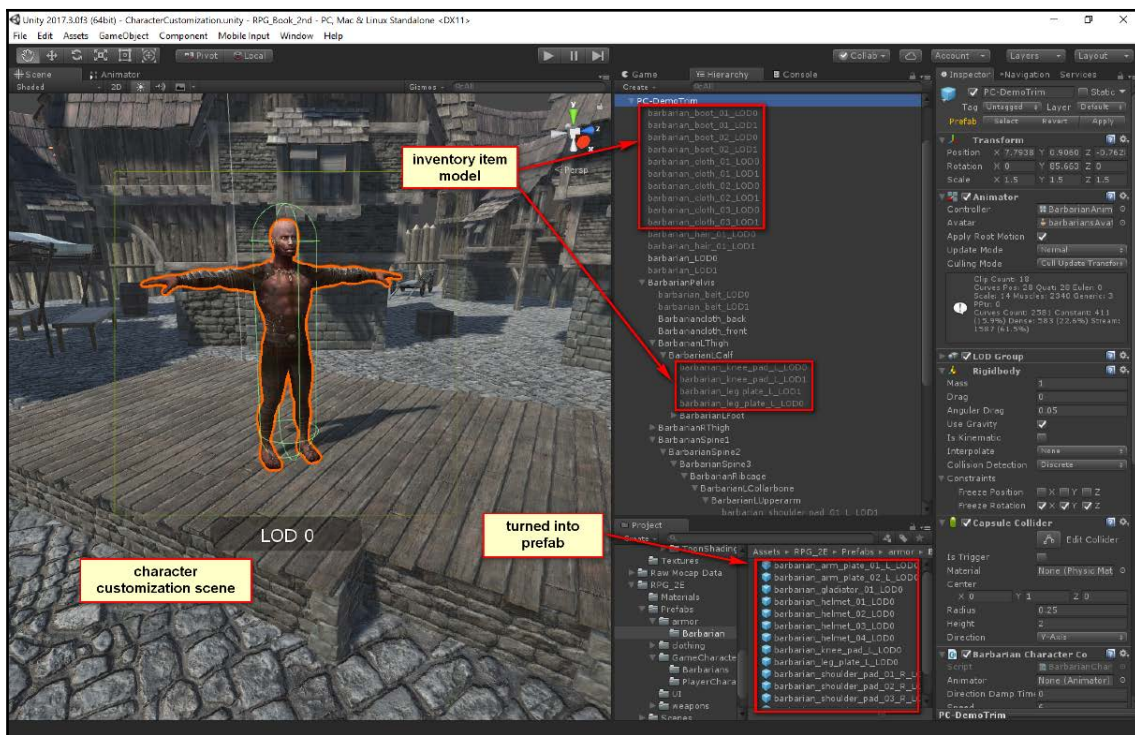
So far, so good. Now we need to actually create the items that will be used to visually represent our inventory items! This is discussed in the following section.

Creating an inventory item

It is now time to actually create the items (assets) we are going to use for our inventory system. I will be creating one item type from each item category to keep things simple. This section will again be highly dependent on how you have modeled your character models. As discussed earlier in the book, in my particular model, all of the character's essential parts are embedded within the **fbx**. In this case, you will need to navigate down your model's hierarchy and extract the mesh for the specific armor or weapon, or anything else you will be using for the inventory.



You can also use independent models representing your inventory items that may or may not be related to your character model's mesh. These items are just used for visual representation within the world, so that the player can pick them up.



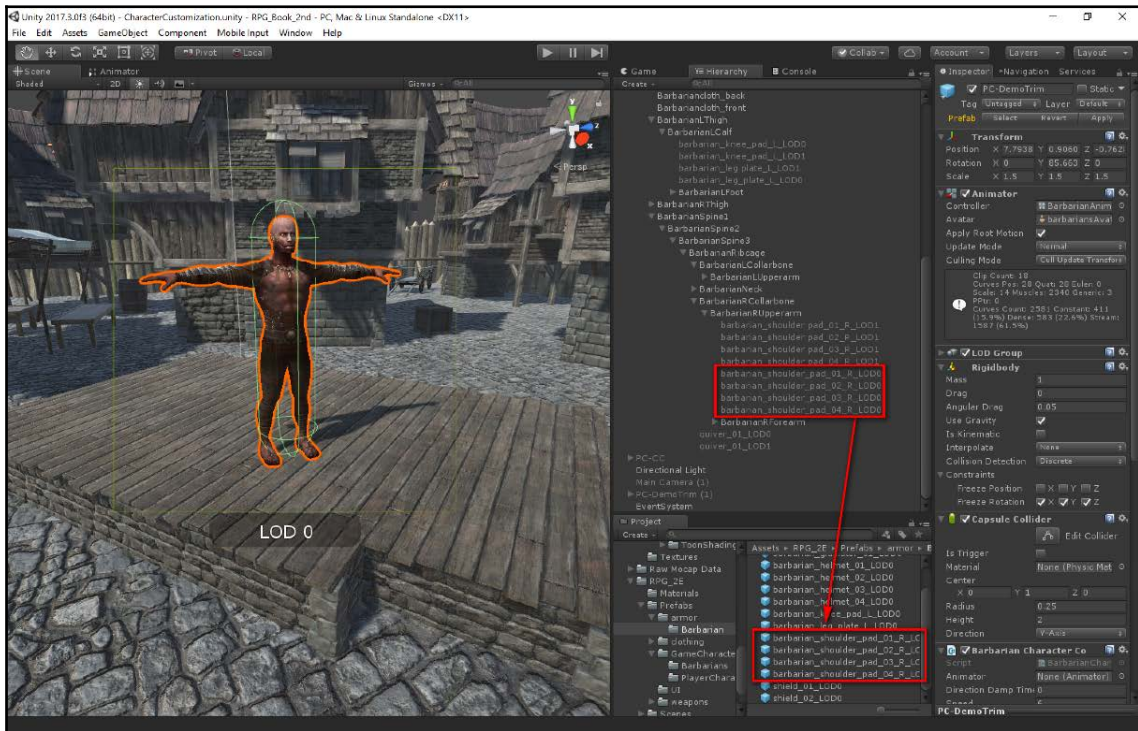
Creating asset prefabs

If you recall from the character customization scene, we had already gone through the model and identified the parts we wanted to have the player be able to enable or disable based on the selection they made through the interface.

Creating the prefab

If you have not already done so, go ahead and create a folder in your **Project Window** named **Prefabs**. Within this folder, go ahead and create a new folder and name it **InventoryItems**, and then create a sub-folder named **ShoulderPads**. You are welcome to use a different naming and folder structure if you choose, as long as you are comfortable and it is organized for you to work with.

To create a prefab, you simply need to take an existing GameObject that is present in the **Scene Window** and drag it into the **Project Window**. To keep things organized, we will be using the structure defined in the previous paragraph. So you will need to navigate to the **ShoulderPads** folder in the project window, and then simply drag one of the shoulder pad meshes from your model and drop it into the **ShoulderPads** folder. Take a look at the following screenshot:



Creating asset prefabs

Observe, when you create a prefab, the prefab will be an exact copy of the GameObject in the active scene! In this case, my mesh is disabled in the scene; therefore, when I create a prefab of the mesh, it will also be disabled! Since it is disabled, when you drag the newly created prefab into the scene as a new GameObject, it will be invisible; you will need to enable it.

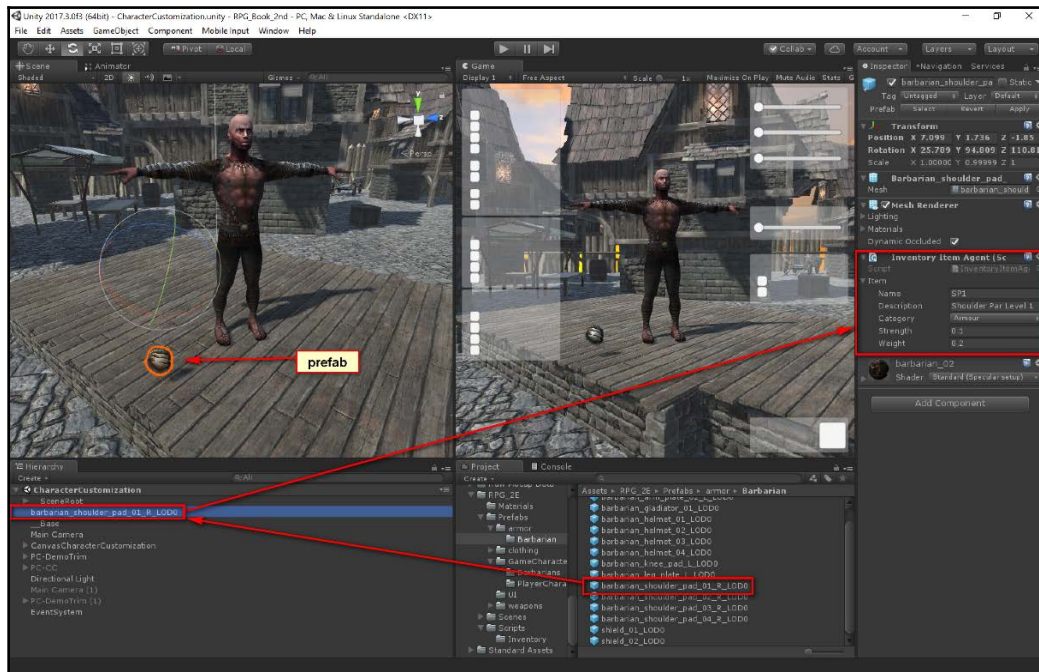
Adding an inventory item agent

We need the means to interact with our inventory items. In order to do this, we will need to create a new script that will handle our interaction with the inventory items during game play. This will be coded in the `InventoryItemAgent.cs` script. At the moment, the script will just enable us to interact with the `InventoryItem` object through the IDE.

A listing of the script is as follows:

```
using UnityEngine;
namespace com.noorcon.rpg2e
{
    public class InventoryItemAgent : MonoBehaviour
    {
        public InventoryItem Item;
    }
}
```

Very simply, in order for us to be able to interact with the `GameObject`, we would need to use a script that inherits `MonoBehaviour`. Go ahead and attach this script to your prefab. Now you can easily set up your inventory items visually. Take a look at the following screenshot:



Inventory item agent

In the preceding screenshot, you can see that we have created a `GameObject` from the prefab and, using the **InventoryItemAgent** component, we have access to the properties of the **InventoryItem** object. Utilizing this concept, you can now create your prefabs for the different types of inventory item.



If you are applying your changes in the scene window, make sure you apply them to the original prefab so that it keeps it in memory.

Caution: When you apply changes to a prefab, all instances of the prefab get updated with the new attributes.

At the moment we have implemented an easy way to define our inventory items, but we still need to implement user interaction with the items. The logic for the interaction will be implemented in the `InventoryItemAgent.cs` script. First we need to identify who we are colliding with; in this case, we want to make sure it is the player that is going to collect the item. Second, we need to store the data in `GameMaster`, and also remove `GameObject` from the active scene. The last two parts will be handled by `GameMaster`, as you will see.

The new code listing for `InventoryItemAgent.cs` is as follows:

```
using UnityEngine;
namespace com.noorcon.rpg2e
{
    public class InventoryItemAgent : MonoBehaviour
    {
        public InventoryItem Item;
        public void OnTriggerEnter(Collider c)
        {
            // make sure we are colliding with the player
            if (c.gameObject.tag.Equals("Player"))
            {
                // Make a copy of the Inventory Item Object
                InventoryItem myItem = new InventoryItem();
                myItem.CopyInventoryItem(Item);
                // Add the item to our inventory
                GameMaster.instance.Inventory.AddItem(myItem);
                // Destroy the GameObject from the scene
                GameMaster.instance.RpgDestroy(gameObject);
            }
        }
    }
}
```

I have created a new function in the `InventoryItem.cs` script called `CopyInventoryItem()`. This function is used to make a copy of one `InventoryItem` object into another one. The code for the newly added function in the `InventoryItem` class is as follows:

```
public void CopyInventoryItem(InventoryItem item)
{
    Category = item.Category;
    Description = item.Description;
    Name = item.Name;
    Strength = item.Strength;
    Weight = item.Weight;
}
```

We have already seen how to add an item to the inventory using `GameMaster`. However, we need to add a new function that will handle the destruction of `GameObjects` in our game. This is done by the `RPG_Destroy()` function.

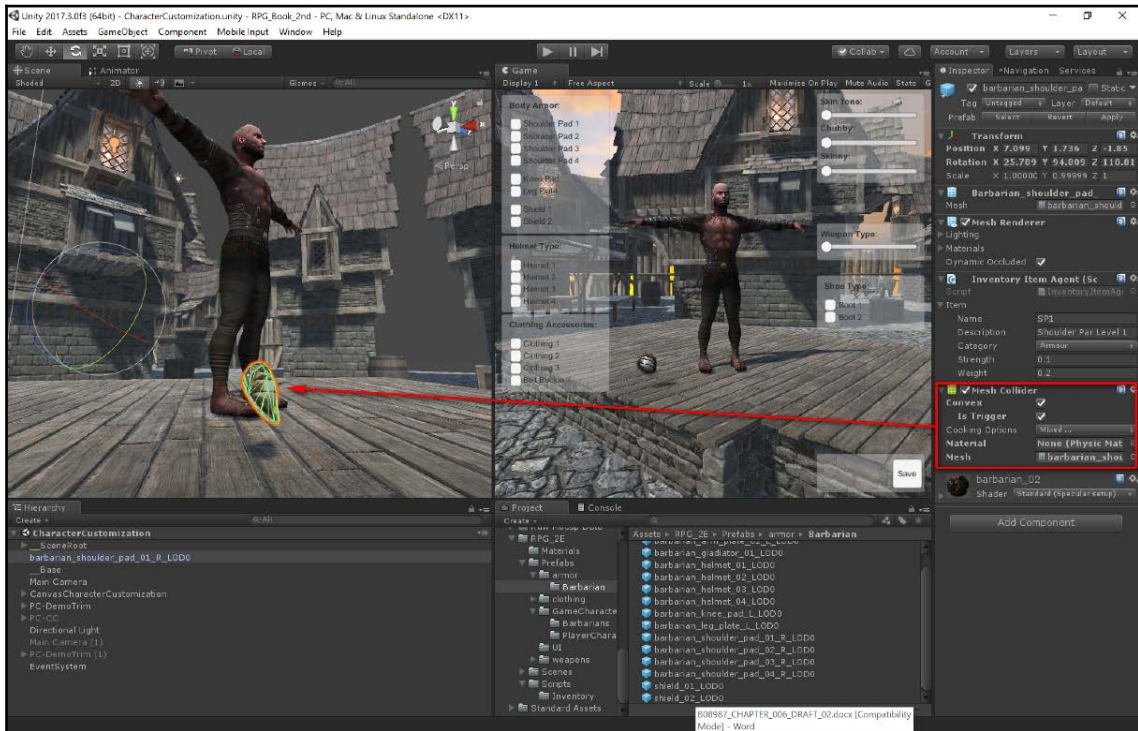


You cannot use `Destroy()`, `DestroyImmediate()`, or `DestroyObject()` since they are part of all `GameObjects` in Unity. Therefore, be cautious of your naming convention within your own classes.

The listing of the new function is as follows:

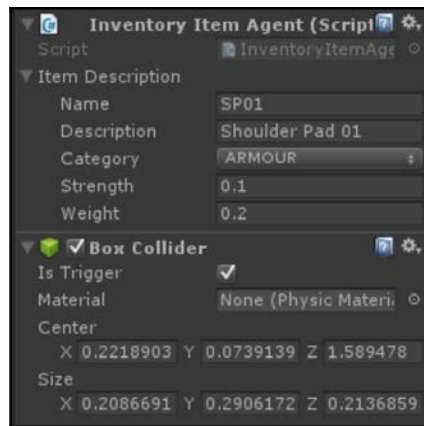
```
public void RpgDestroy(GameObject obj)
{
    Destroy(obj);
}
```


One final component that needs to be added to your prefabs representing the inventory items is a *Collider*. Take a look at the following screenshot:



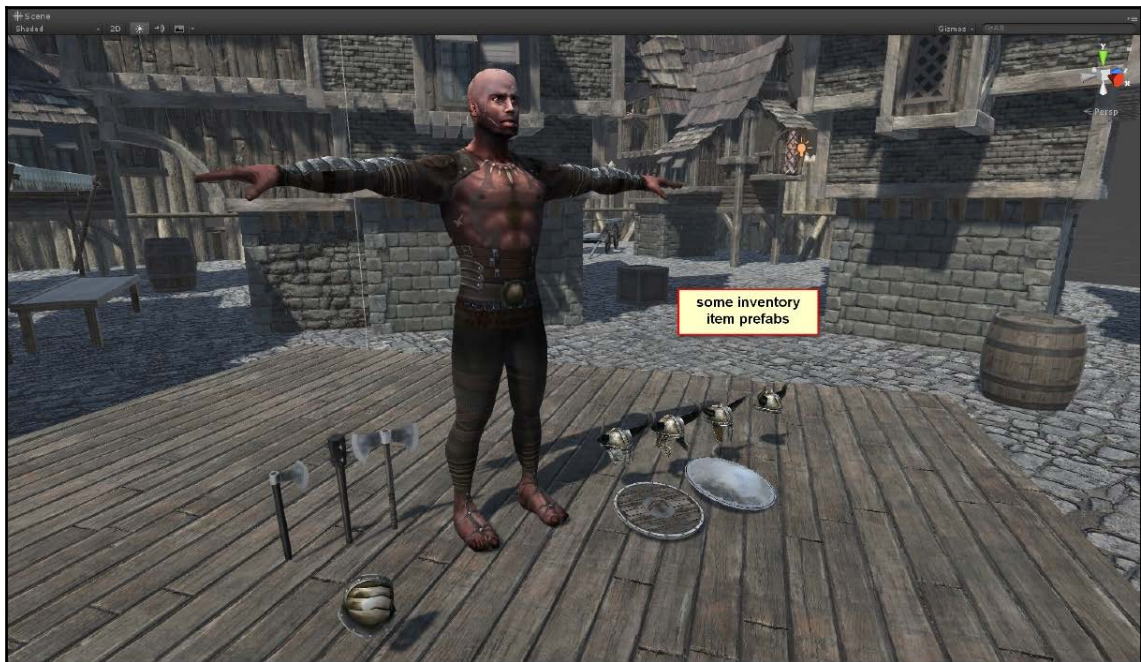
Mesh collider

I used a **Mesh Collider** to keep things simple. A collider can be added by selecting the **Add Component | Physics | Mesh Collider** from the **Inspector Window**, as shown in the following screenshot:



Inventory items defined as prefabs

The following figure shows some of the inventory item prefabs I have created for demonstration purposes:



Inventory prefab sample

The key for all of this to work is to make sure that your prefabs have the `InventoryItemAgent.cs` script as well as a `Collider` component attached to the prefabs. Then you will need to provide the inventory item data through the IDE uniquely identifying each one.

The following table lists the data for each inventory item defined:

Prefab	Name	Description	Category	Strength	Weight
Helmet	HL01	Brass Helmet with Two Horns	Armor	0.2	0.2
	HL02	Brass Helmet Face Protection	Armor	0.3	0.25
	HL03	Bronze Helmet Protecting Face	Armor	0.3	0.3
	HL04	Bronze Helmet	Armor	0.2	0.25
Shield	SL01	Iron Shield	Armor	0.3	0.3
	SL02	Wooden Shield	Armor	0.2	0.2
Shoulder Pads	SP01	Shoulder Pad 01	Armor	0.1	0.2
	SP02	Shoulder Pad 02	Armor	0.1	0.2
	SP03	Shoulder Pad 03	Armor	0.15	0.25
	SP04	Shoulder Pad 04	Armor	0.2	0.25
Weapons	Axe1	Single Head	Weapon	0.2	0.1
	Axe2	Double Head	Weapon	0.25	0.2
	Club1	Wooden Club	Weapon	0.2	0.1

Again, the data is arbitrary: you decide what best suits your game and game design. Usually, you will have a table that would be loaded with such information, and your game manager will handle preloading it at runtime from the server.

Designing an inventory interface

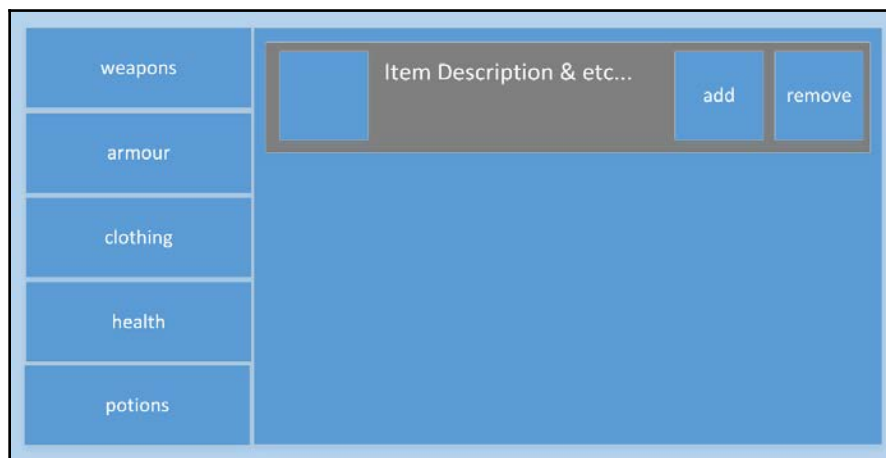
It is now time to think about how we are going to visualize the inventory during game play. Creating a user interface for any game is a challenging task. You need to have a balanced approach about the amount of information you want to display on the screen at playtime, without interfering with the game play. In the meantime, you want to make sure that the player has the most crucial and important information necessary to complete their mission to hand.

With that said, let's look at how we can design a simple user interface to enable the player with the basics of interacting with the inventory system. Here is a list of minimum features that the player should be able to perform:

- Display the inventory at any time during game play
- Navigate based on category
- See what items are listed under each category
- Be able to remove an item from the inventory
- Be able to consume an item from the inventory
- See what inventory items are already in use by the player

The preceding list will give us a good start for implementing our inventory interface. Let's start by identifying the categories that will need to be displayed. The categories are defined as an **enum** named `ItemCategory` in the `BaseItem` class.

We have the following: weapons, armor, clothing, health, and potion. Take a look at the following screenshot:



The preceding screenshot shows a concept I am leaning towards for the implementation of the inventory interface. The interface can be constructed by utilizing the following UI elements:

- Buttons
- Panels
- Text
- Images

Each category will have a button, and there will be one main panel containing the list of items per category, as illustrated in the preceding diagram. Each item will be contained in its own panel, which will contain an image of the inventory item, the item description, and two buttons that can be used to add or remove the item from the inventory system.

Creating the inventory UI framework

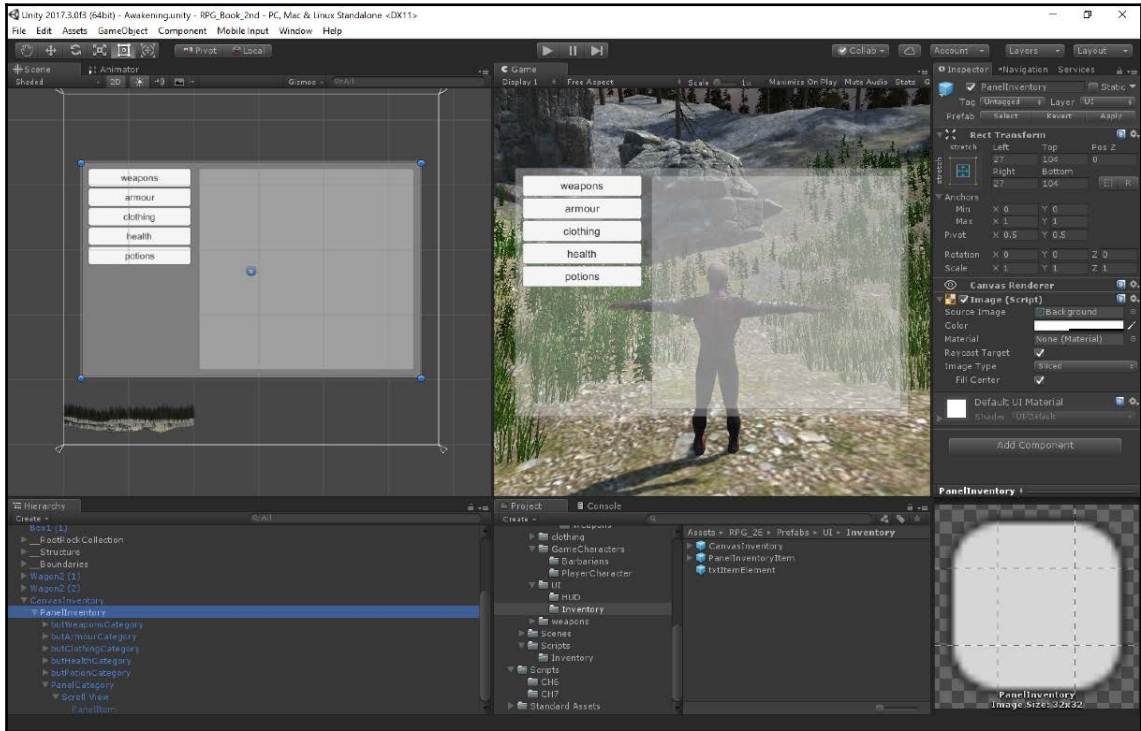
Let's start by first implementing the initial framework for our inventory system graphical interface. In the main scene of your project, go ahead and create a new *Canvas* *GameObject* if you have not done so already.

To do so, right-click on the **Hierarchy Window**, and select **UI | Panel**. This will automatically create a **Canvas** *GameObject* and a Panel UI Element as a child to the canvas.

Rename this panel **PanelInventory**. This will be the main panel that will contain everything else. Now, let's go ahead and start building the buttons that will represent our main categories.

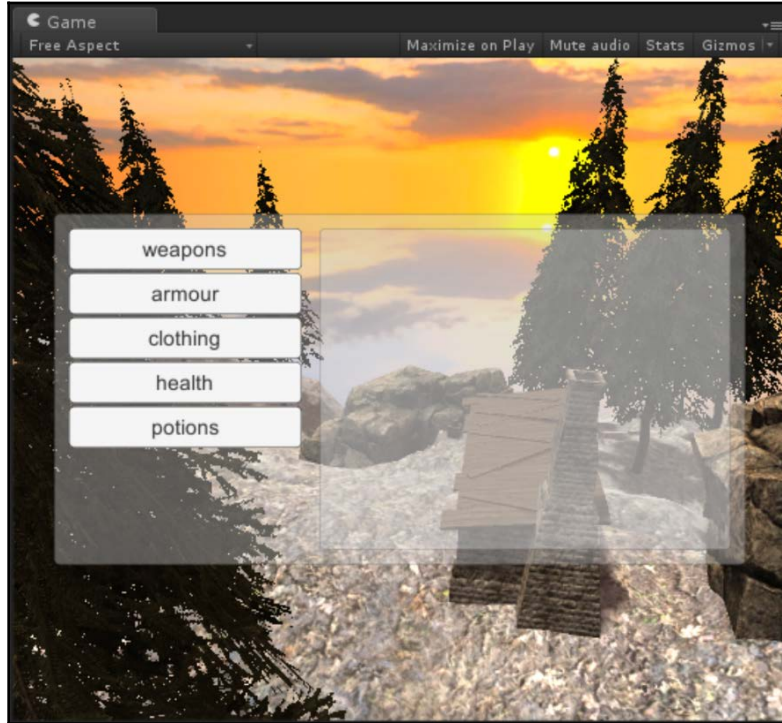
Similarly, right-click on the **PanelInventory** *GameObject* and select **UI | Button**. This will make sure that the newly created button becomes a child of the **PanelInventory**. If, for whatever reason, this is not the case in the **Hierarchy Window** after the creation of the button(s), simply drag the newly created button(s) under the **PanelInventory** panel. Do this for all five categories. Rename the buttons appropriately, such as **butWeaponsCategory** and so on.

Change the caption of the button so that it reflects the button's function. Also, rename the **Text** element to something like the following: **txtWeaponsCategory** and so on. Take a look at the following screenshot:



Inventory UI

Finally, add a new **Panel** element to the **PanelInventory** again, by selecting the **PanelInventory** GameObject and right-clicking and selecting **UI | Panel**. Rename the newly created panel to **PanelCategory**. Take a look at the following screenshot:



Your inventory user interface should look something like the preceding screenshot. Before we get more involved, let's go ahead and hook up some of the basics for showing and hiding the inventory interface for the player. To do this, we will need to modify the `UiController.cs`, `GameLevelController.cs`, and also the `GameMaster.cs` scripts.

I will not be listing the whole source file as we will do this later on in the chapter. The changes for each script for now are as follows:

- `UiController.cs`: Added a new function named `DisplayInventory()` and a new variable to reference the inventory canvas named `InventoryCanvas`. See the following code:

```
public void DisplayInventory()
{
    InventoryCanvas.gameObject.SetActive(GameMaster.instance.DisplayInventory);
    Debug.Log("Display Inventory Function");
}
```

```
}
```

- **GameLevelController.cs:** Updated the `OnlevelWasLoaded()` function to assign the `uiController` `GameObject` to the `GameManager` instance, if one is present. See the following code:

```
if (GameObject.FindGameObjectWithTag("Ui"))
{
    GameManager.instance.Ui
    =
    GameObject.FindGameObjectWithTag("Ui").GetComponent<UiController>();
}
```

- **GameManager.cs:** Modified the `Update()` function to check and see if the J key was pressed and released. This in turn toggled a Boolean variable to see if we are supposed to show or hide the inventory interface. See the following code:

```
void Update()
{
    if (instance.LevelController.CurrentScene.name != SceneName.MainMenu)
    {
        if (Input.GetKeyUp(KeyCode.J))
        {
            instance.DisplayInventory = !DisplayInventory;
            instance.Ui.DisplayInventory();
        }
    }
}
```

If you test your scene from the main menu, you will be able to test out the interface and toggle it on and off.



Don't forget that you will need to disable the Canvas for the inventory system during design time, or at runtime when the game loads initially.

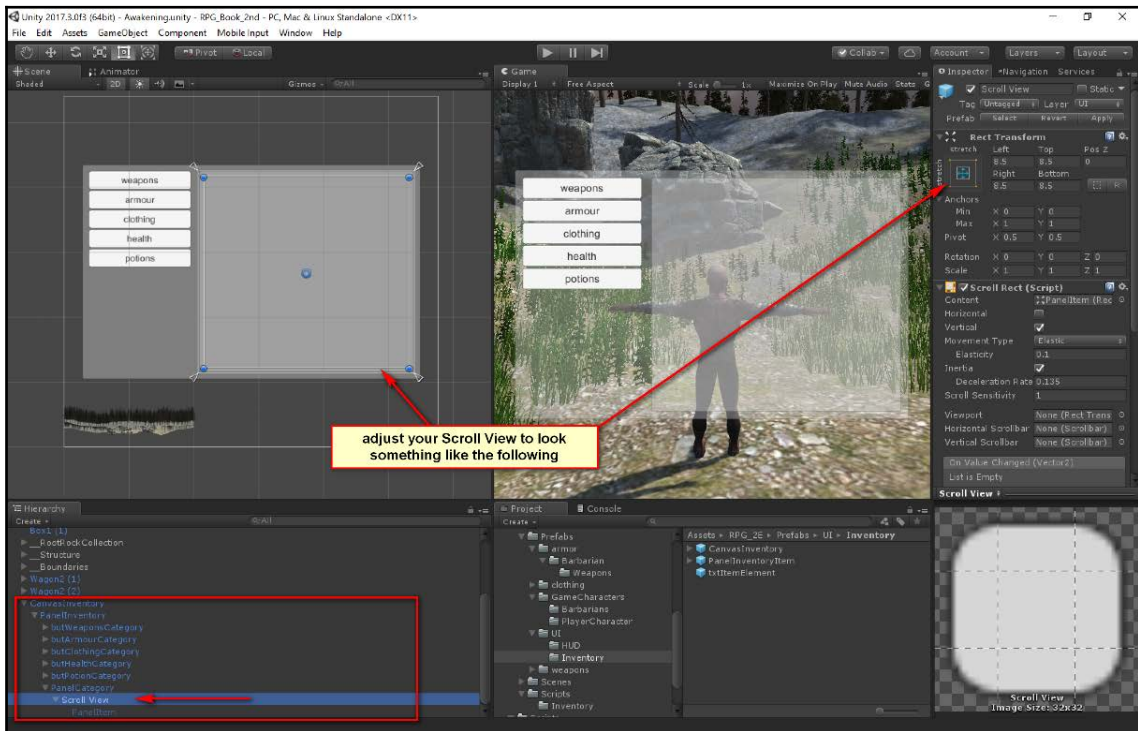
Designing a dynamic item viewer

The next challenge for us is to create a method to dynamically populate the inventory items and displaying them properly on the user interface. We are going to use two new UI elements that we have not used before. We will be using a *scroll view* to give us the ability to scroll through the items when needed.

Let's first get the scroll view set up and look at how to add a simple UI prefab to the scroll view. Once this is done, we can go ahead and enhance the UI prefab to handle what we have outlined in the previous section.

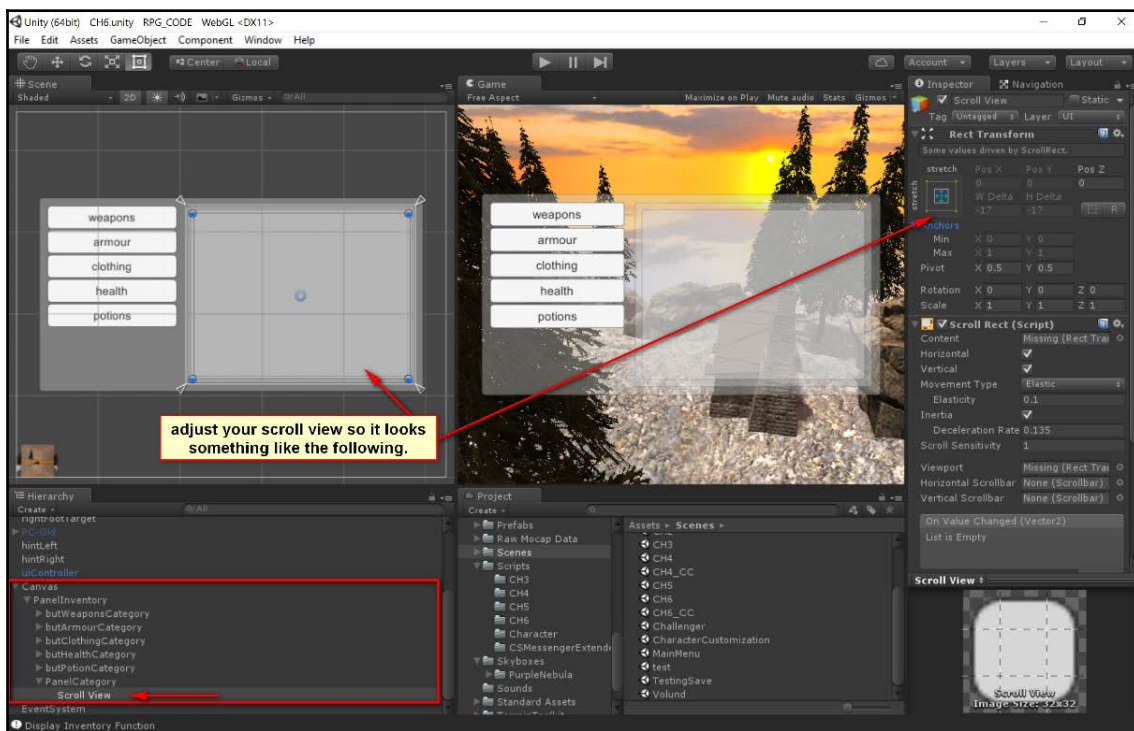
Adding a scroll view

The next few screenshots will illustrate how to add a scroll view functionality to your user interface:



Adding a scroll view

Go to the scene where you have created your Inventory UI, and select the *PanelCategory* in the *Canvas*. Right-click and select *UI | Scroll View* to add a scroll view UI element. Take a look at the following screenshot:



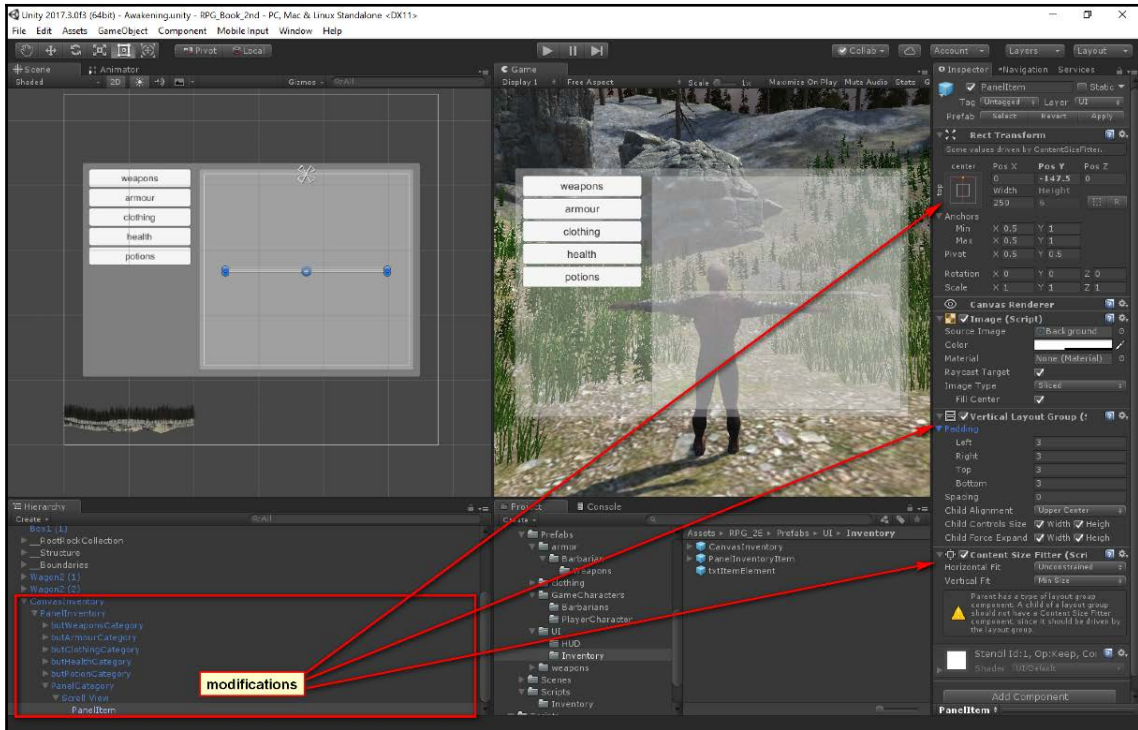
Adding Scroll View

You should now have a scroll view UI element with associated children under your *PanelCategory* panel. The children will be *Viewport*, *Scrollbar Horizontal*, and *Scrollbar Vertical*.



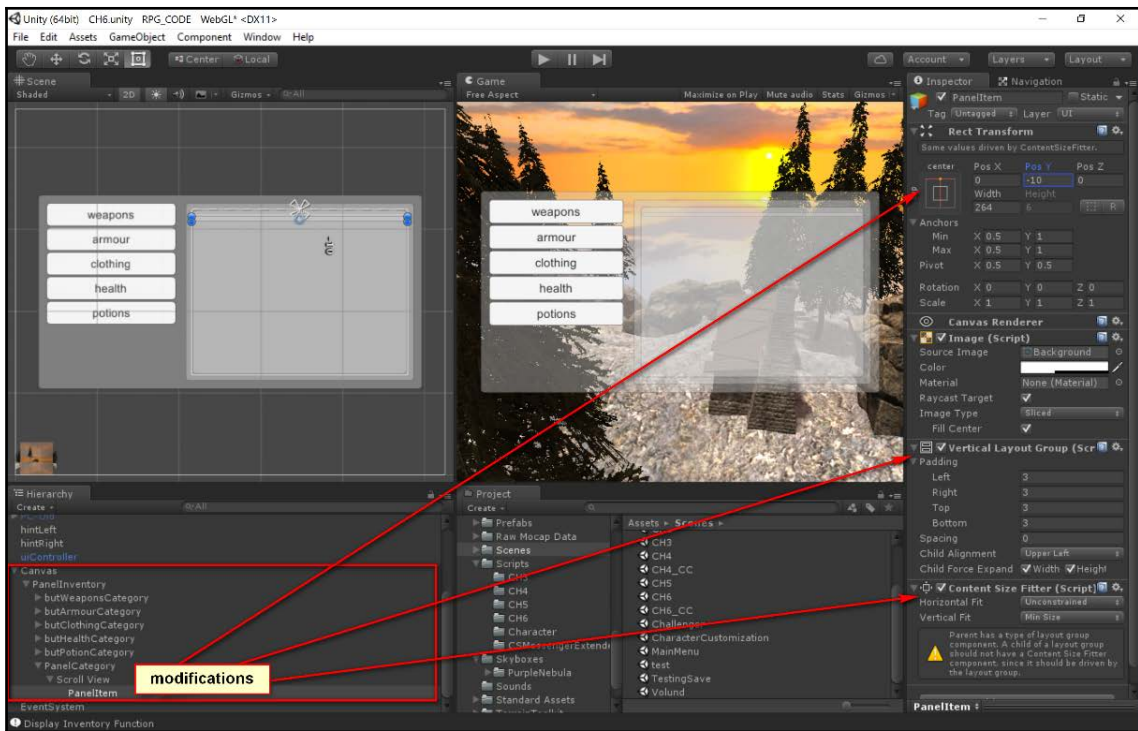
Make your adjustment to the scroll view UI element before you delete the children.

We are going to make some modifications to the default scroll view. Go ahead and delete the following from *Scroll View*: the *Scrollbar Horizontal*, *Scrollbar Vertical*, and the *Viewport* child elements. After you are done, your screen should look something like the preceding screenshot. Now take a look at the following screenshot:



Layout adjustment

Next, we need to add a **Panel** element as a child to our **Scroll View**. Go ahead and select the scroll view, right-click, and select **UI | Panel**. Rename the newly added panel **PanelItem**. We need to add two layout components to our **PanelItem**. To do this, select **PanelItem**, and from the **Inspector Window** select **Add Component | Layout | Vertical Layout Group**, and once more select **Add Component | Layout | Content Size Filter**. Take a look at the following screenshot:



Layout adjustment

Go ahead and modify the following attributes under the **Vertical Layout Group** components. Set the **Left**, **Right**, **Top**, and **Bottom Padding** to **3**. Set the **Spacing** to **0**, change the **Child Alignment** to **Upper Left**, and set the **Child Force Expand** to **True** for both **Width** and **Height**.

For the **Content Size Filter** component, set the **Horizontal Fit** to **Unconstrained** and the **Vertical Fit** to **Min Size**.

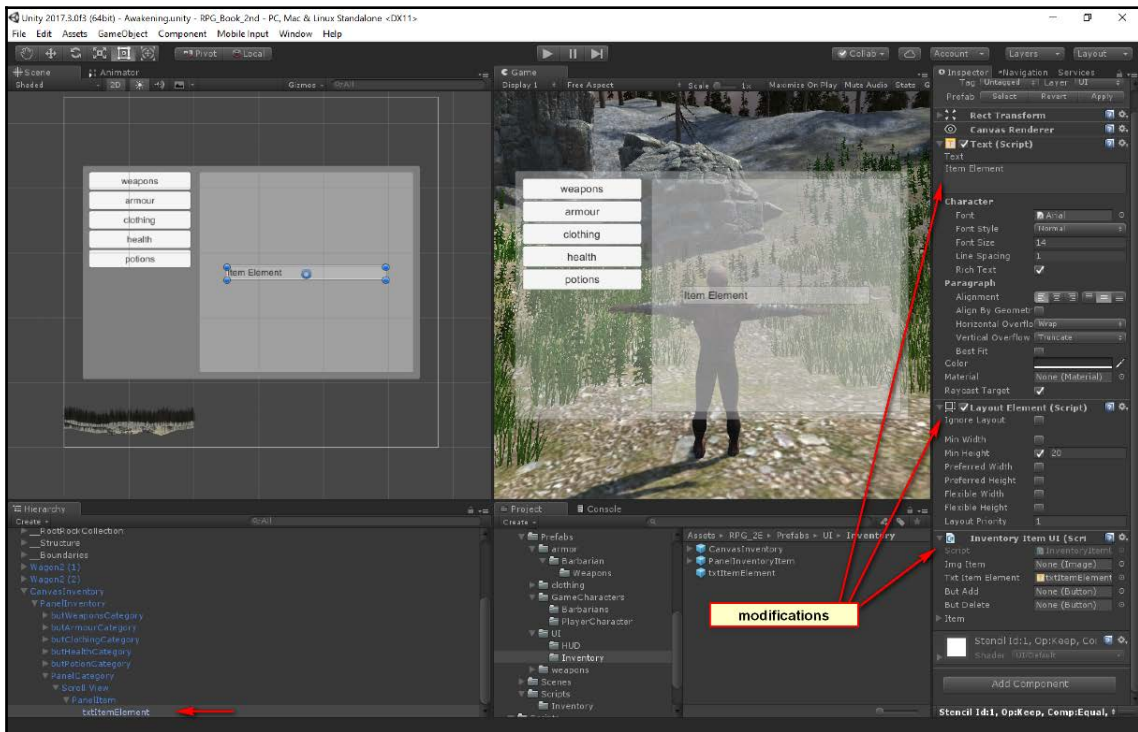
Finally, in the **Rect Transform** component, change the **Anchor Point** to **Top Center**, and modify the **Pos Y** to **-10**.

At this point we have the basic framework in place. The next step it to populate our newly created **ScrollView**!

Adding elements to PanelItem and Scroll View

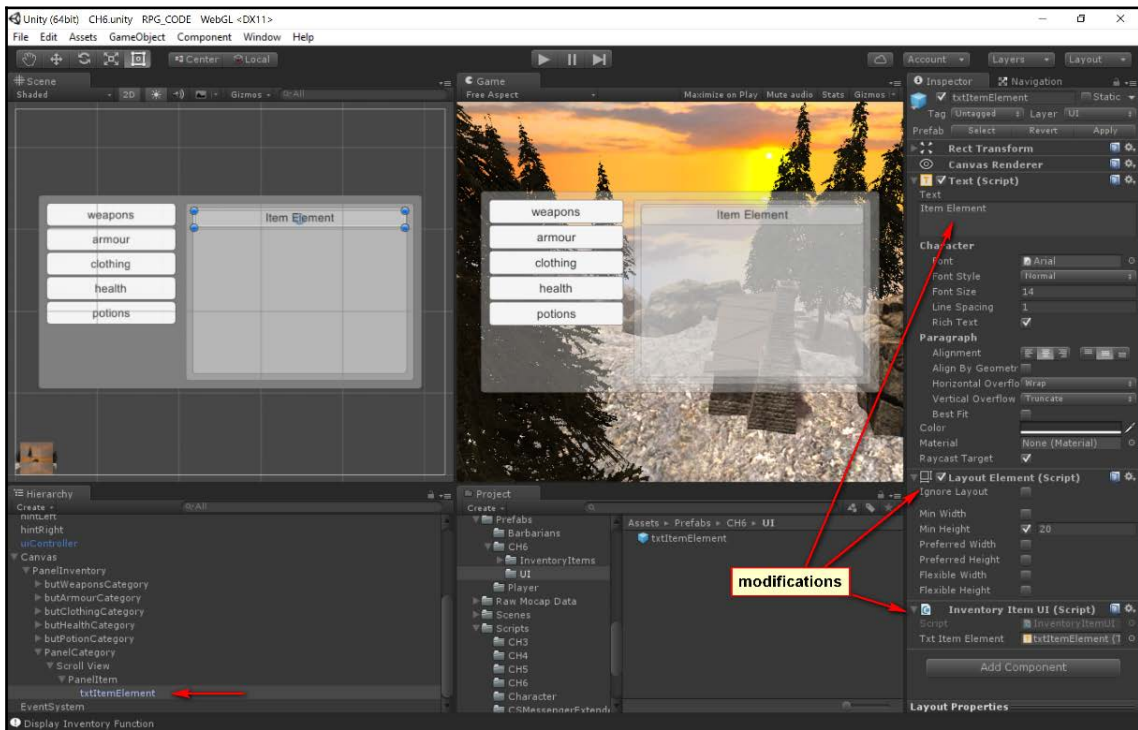
For starters, let's go ahead and add a *Text* element under the *PanelItem* panel. Again select the *PanelItem* element and right-click and select **UI | Text**. Next select the text element and rename it to *txtItemElement*. We need to add a new component to the *Text* element, from the **Inspector Window** go ahead and select **Add Component | Layout | Layout Element**.

Modify the **Min Height** attribute of the **Layout Element** component to **20**. Take a look at the following screenshot:



Modification

The preceding screenshot illustrates the setup of the `InventoryItem` element and UI. Now take a look at the following screenshot:



Adding an inventory item

We need a means to access and modify the **Text** attribute of the new `Text` UI element. In order to do this, we need to create a new script called **InventoryItemUI.cs**. The code will just have a public variable that will reference the `Text` element. The listing is as follows:

```
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class InventoryItemUi : MonoBehaviour
    {
        public Text txtItemElement;
    }
}
```

Finally, drag and drop the `Text` element from the hierarchy window into the **TextItemElement** attribute of the **InventoryItemUI** component attached to the **txtItemElement** object. Refer to the preceding screenshot.



The script is used to self-reference. We will use it to modify the text component of the `Text` UI element.

Now we will need to create a **Prefab** of the **txtItemElement** by dragging and dropping it into a designated folder. I have created a new folder under my **Prefabs** folder named it **UI** and have created the prefab in that folder. Refer to the preceding screenshot.

You can now delete the **txtItemElement** from the **Hierarchy Window** under the **PanelItem** object. We will be adding them dynamically during runtime.

There is one last configuration you will need to do before we move forward. You will need to add a **Mask** component to the **ScrollView** UI element. Select the **Scroll View** from the **Hierarchy Window**, and from the **Inspector Window** select **Add Component | UI | Mask**. After the addition of the **Mask** component, make sure that the **Show Mask Graphics** attribute is *unchecked*.

Adding txtItemElement dynamically

Now it is time to add our inventory item placeholder dynamically to the **PanelItem** UI element. To do so, we will use the `UiController.cs` script. Go ahead and open up the script and add the following variable to the class:

```
public Transform InventoryPanelItem
public GameObject InventoryItemElement;
```

In the designer, you will need to assign the **PanelItem** UI element from the *Canvas* **GameObject** and the **txtItemElement** prefab from the prefab folder.

Next, we are going to modify the `Update()` function so that when we press the **H** key on the keyboard, it will go ahead and instantiate a new **InventoryItemElement** and make it a child element of the **PanelItem** object.

The listing of the code is as follows:

```
public void Update()
{
    if (Input.GetKeyUp(KeyCode.H))
```

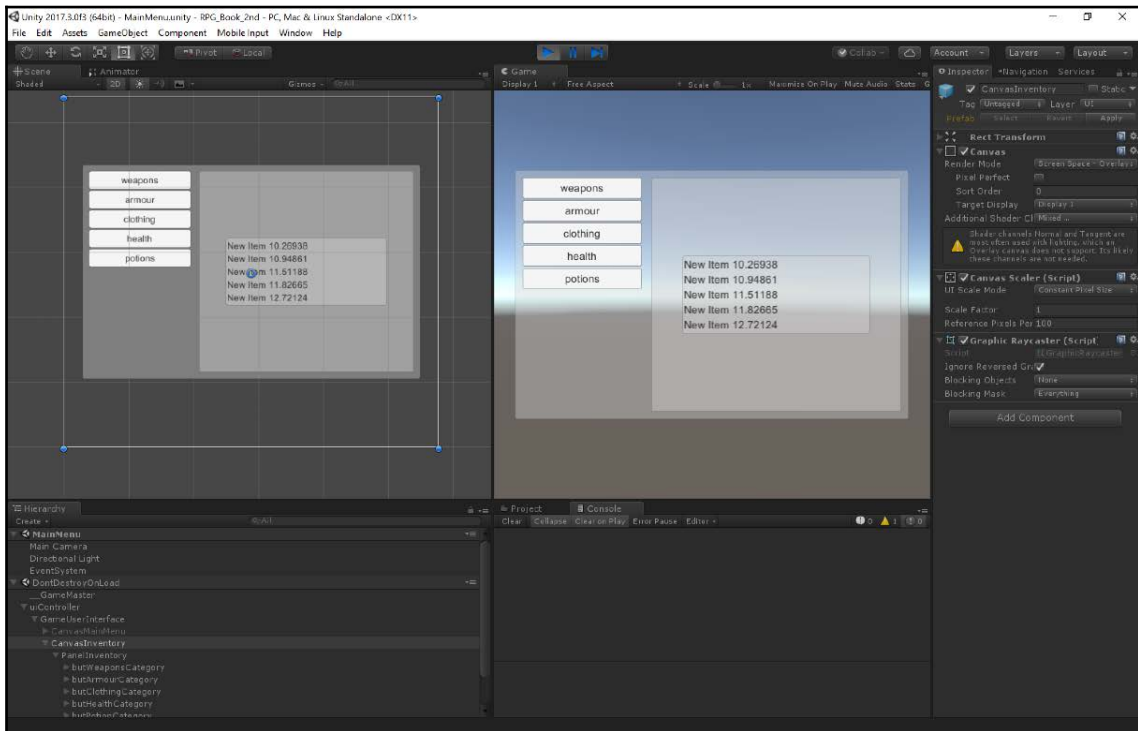
```

{
    GameObject newButton
    = Instantiate(InventoryItemElement) as GameObject;
    InventoryItemUi txtItem
    = newButton.GetComponent<InventoryItemUi>();
    txtItem.txtItemElement.text
    = string.Format("New Item {0}", Time.time);
    newButton.transform.SetParent(InventoryPanelItem);
}
}

```

The preceding code listing simply instantiates the prefab and makes it a child of the `PanelItem` element. We are also changing the caption of the element and placing it with a timestamp to see the uniqueness of each UI element.

The outcome is shown in the following screenshot:



Inventory view

At this point, we have put together the main elements to have our inventory interface list items dynamically and be able to scroll through them.

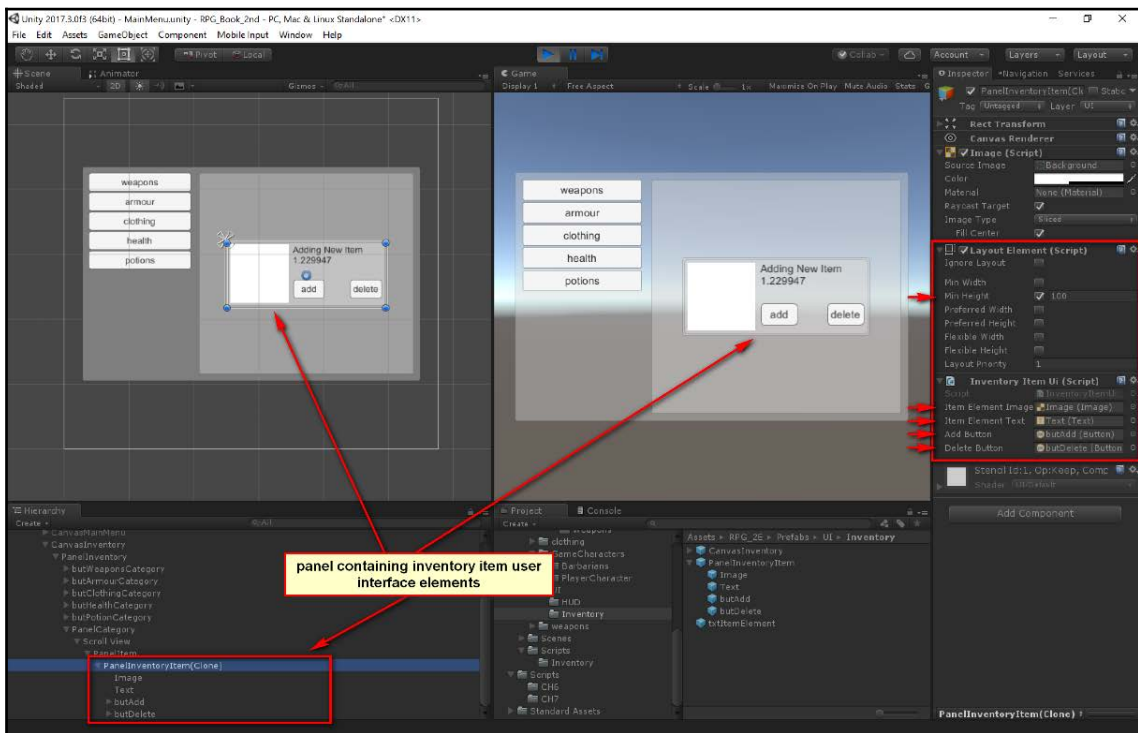
Building the final inventory item UI

To create the actual **Inventory Item** user interface, we are going to need to use several UI elements. We will need a panel to be the container of the item. Within the *Panel*, we are going to need to use an *Image*, a *Text*, and two *Button* UI elements.



Note: I will not be going through the steps of how to put the panel together. You should know how to create user interfaces by now.

Just make sure that you add the *Layout Element* component and *Inventory Item UI* script to the panel that will be the base for the inventory item.



Panel containing inventory item UI elements

The following figure illustrates the UI component that has been developed for displaying the inventory item. Since the UI component has been modified, we also have to update the `InventoryItemUI.cs` script to contain a reference to all of the new UI elements in the *Panel*.

The listing of the new `InventoryItemUi.cs` is as follows:

```
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class InventoryItemUi : MonoBehaviour
    {
        public Image ItemElementImage;
        public Text ItemElementText;
        public Button AddButton;
        public Button DeleteButton;
    }
}
```

We also need to update the `UiController.cs` script to handle the new prefab accordingly.

The listing for the new UI prefab in `UiController.cs` is as follows:

```
public void Update()
{
    if (Input.GetKeyUp(KeyCode.H))
    {
        GameObject newItem
        = Instantiate(InventoryItemElement) as GameObject;
        InventoryItemUi txtItem
        = newItem.GetComponent<InventoryItemUi>();
        txtItem.ItemElementText.text
        = string.Format("Adding New Item {0}", Time.time);
        // button triggers
        txtItem.AddButton.GetComponent<Button>().onClick.AddListener(() => {
            Debug.Log(string.Format("You have clicked ADD Button for {0}",
                txtItem.ItemElementText.text));
        });
        txtItem.DeleteButton.GetComponent<Button>().onClick.AddListener(() => {
            Debug.Log(string.Format("You have clicked DELETE Button for {0}",
                txtItem.ItemElementText.text));
            Destroy(newItem);
        });
    }
}
```



```
newItem.transform.SetParent(InventoryPanelItem);  
}  
}
```

In the preceding listing, the main concept I want to point out is the implementation of the `onClick()` event handler for the buttons within the prefab.

Since we are dynamically generating our UI, and hence the buttons, we need to be able to trigger the `onClick()` function somehow; this is done by adding a listener, as shown in the code.

For now, when you click the **AddButton** button, you will get an output on the **Console Window** with the appropriate caption. When you click the **DeleteButton** button, you will get another output on the **Console Window**, with the appropriate caption. Then the item will be destroyed, that is, removed from the inventory.

Integrating the UI with the actual inventory system

We have seen and implemented the concepts necessary to make our inventory system UI work properly. Now it is time to actually fill the user interface with the actual data that is stored in `GameMaster`.

Hooking the category buttons and displaying the data

Using the `UiController.cs` script, we are going to create five new methods that will handle the proper visualization of our inventory system. We are going to add the following five functions:

- `DisplayWeaponsCategory()`
- `DisplayArmourCategory()`
- `DisplayClothingCategory()`
- `DisplayHealthCategory()`
- `DisplayPotionsCategory()`

We also need to clear the existing inventory items from the panel when the user switches from one category to the next. This will require a private function named `ClearInventoryItemPanel()`, which will just do that.

The listing for the new `UiController.cs` script is as follows:

```
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class UiController : MonoBehaviour
    {
        [Header("Settings Window")]
        public RectTransform OptionsPanel;
        public Slider ControlMainVolume;
        public Slider ControlFXVolume;
        [Header("Inventory Window")]
        public RectTransform InventoryCanvas;
        [Tooltip("root for inventory items")]
        public Transform InventoryPanelItem;
        [Tooltip("prefab representing invenotry item UI")]
        public GameObject InventoryItemElement;
        public void Update()
        {
            if (Input.GetKeyUp(KeyCode.H))
            {
                GameObject newItem
                = Instantiate(InventoryItemElement) as GameObject;
                InventoryItemUi txtItem
                = newItem.GetComponent<InventoryItemUi>();
                txtItem.ItemElementText.text
                = string.Format("Adding New Item {0}", Time.time);
                // button triggers
                txtItem.AddButton.GetComponent<Button>().onClick.AddListener(() => {
                    Debug.Log(string.Format("You have clicked ADD Button for {0}",
                        txtItem.ItemElementText.text));
                });
                txtItem.DeleteButton.GetComponent<Button>().onClick.AddListener(() => {
                    Debug.Log(string.Format("You have clicked DELETE Button for {0}",
                        txtItem.ItemElementText.text));
                    Destroy(newItem);
                });
                newItem.transform.SetParent(InventoryPanelItem);
            }
        }
        public void DisplaySettings()
        {

```

```

    GameManager.instance.DisplaySettings = !GameManager.instance.DisplaySettings;
    OptionsPanel.gameObject.SetActive (GameManager.instance.DisplaySettings);
}
public void MainVolume()
{
    GameManager.instance.MasterVolume (ControlMainVolume.vale);
}
public void FXVolume()
{
    GameManager.instance.SoundFxVolume (ControlFXVolume.value);
}
#region INVENTORY UI FUNCTIONS
public void DisplayInventory()
{
    InventoryCanvas.gameObject.SetActive (GameManager.instance.DisplayInventory);
    Debug.Log("Display Inventory Function");
}
public void DisplayWeaponsCategory()
{
    if (GameManager.instance.DisplayInventory)
    {
        ClearInventoryPanelItems();
        foreach (InventoryItem item in GameManager.instance.Inventory.Weapons)
        {
            GameObject newItem
            = Instantiate(InventoryItemElement) as GameObject;
            InventoryItemUI txtItem
            = newItem.GetComponent<InventoryItemUI>();
            txtItem.txtItemElement.text =
            string.Format("Name: {0}, Description: {1}, Strength: {2}, Weight: {3}",
            item.Name,
            item.Description,
            item.Strength,
            item.Weight);
            ...

```

We also had to some modifications to the `InventorySystem.cs` script to make it possible for us to access properties storing the data more easily.

The new listing of the script is as follows:

```

using System;
using System.Collections.Generic;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]

```

```
public class InventorySystem
{
    [SerializeField]
    private List<InventoryItem> weapons = new List<InventoryItem>();
    [SerializeField]
    private List<InventoryItem> armour = new List<InventoryItem>();
    [SerializeField]
    private List<InventoryItem> clothing = new List<InventoryItem>();
    [SerializeField]
    private List<InventoryItem> health = new List<InventoryItem>();
    [SerializeField]
    private List<InventoryItem> potion = new List<InventoryItem>();
    public List<InventoryItem> Weapons
    {
        get { return weapons; }
    }
    public List<InventoryItem> Armour
    {
        get { return armour; }
    }
    public List<InventoryItem> Clothing
    {
        get { return clothing; }
    }
    public List<InventoryItem> Health
    {
        get { return health; }
    }
    public List<InventoryItem> Potion
    {
        get { return potion; }
    }
    private InventoryItem selectedWeapon;
    private InventoryItem selectedArmour;
    public InventoryItem SelectedWeapon
    {
        get { return selectedWeapon; }
        set { selectedWeapon = value; }
    }
    public InventoryItem SelectedArmour
    {
        get { return selectedArmour; }
        set { selectedArmour = value; }
    }
    public InventorySystem()
    {
        ClearInventory();
    }
}
```

```
public void ClearInventory()
{
    weapons.Clear();
    armour.Clear();
    clothing.Clear();
    health.Clear();
    potion.Clear();
}
// this function will add an inventory item
public void AddItem(InventoryItem item)
{
    switch (item.Category)
    {
        case BaseItem.ItemCatrgory.Armour:
        {
            armour.Add(item);
            break;
        }
        case BaseItem.ItemCatrgory.Clothing:
        {
            clothing.Add(item);
            break;
        }
        case BaseItem.ItemCatrgory.Health:
        {
            health.Add(item);
            break;
        }
        case BaseItem.ItemCatrgory.Potion:
        ...
    }
}
```

Notice that I have removed the code from the `Update()` function in the `UiController.cs` script as it was only for testing purposes.

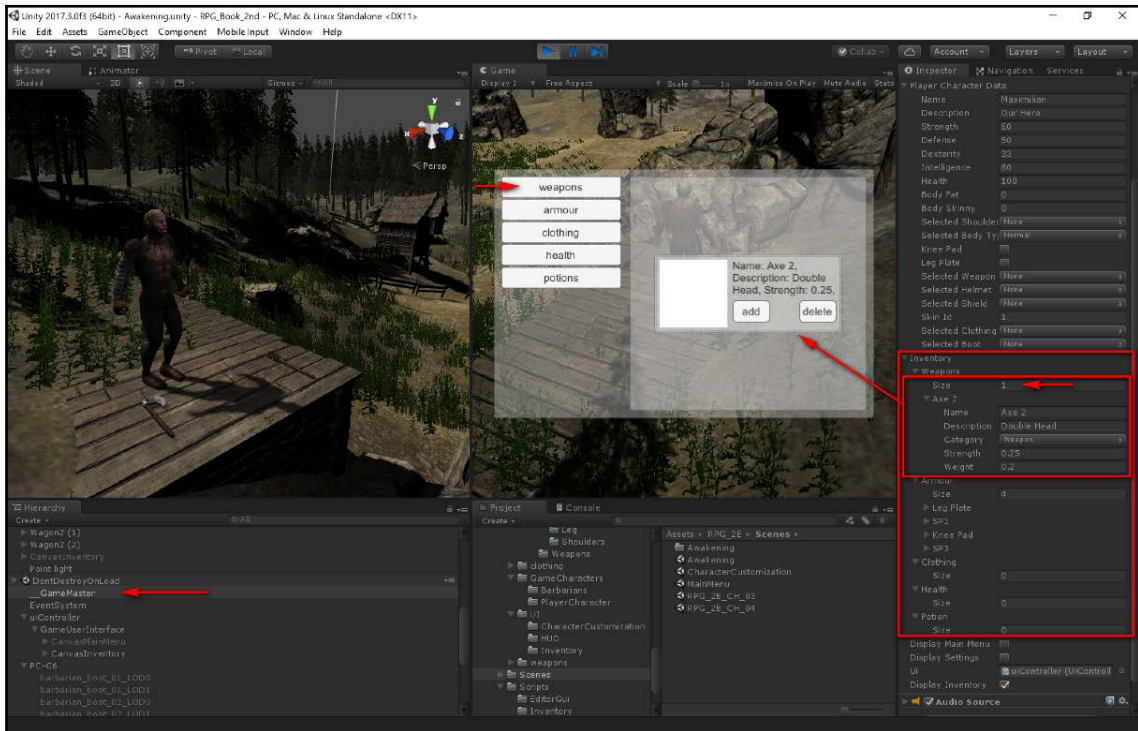
Testing the inventory system

For testing purposes, I have placed a number of the inventory item prefabs created earlier in this chapter. Take a look at the following screenshot:



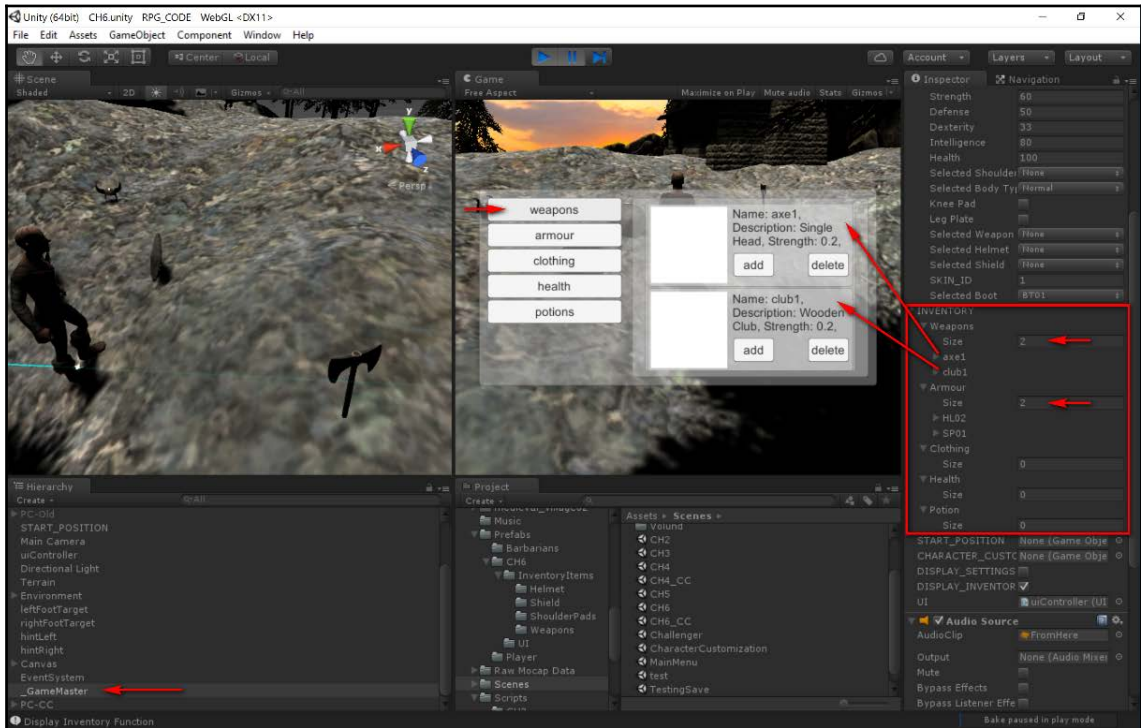
Inventory items to be collected

Start the game from the main menu and go through the character customization scene to save the character player and start the game. Once you are in the playable scene, go ahead and collect a few of the items that have been placed throughout the scene. Take a look at the following screenshot:



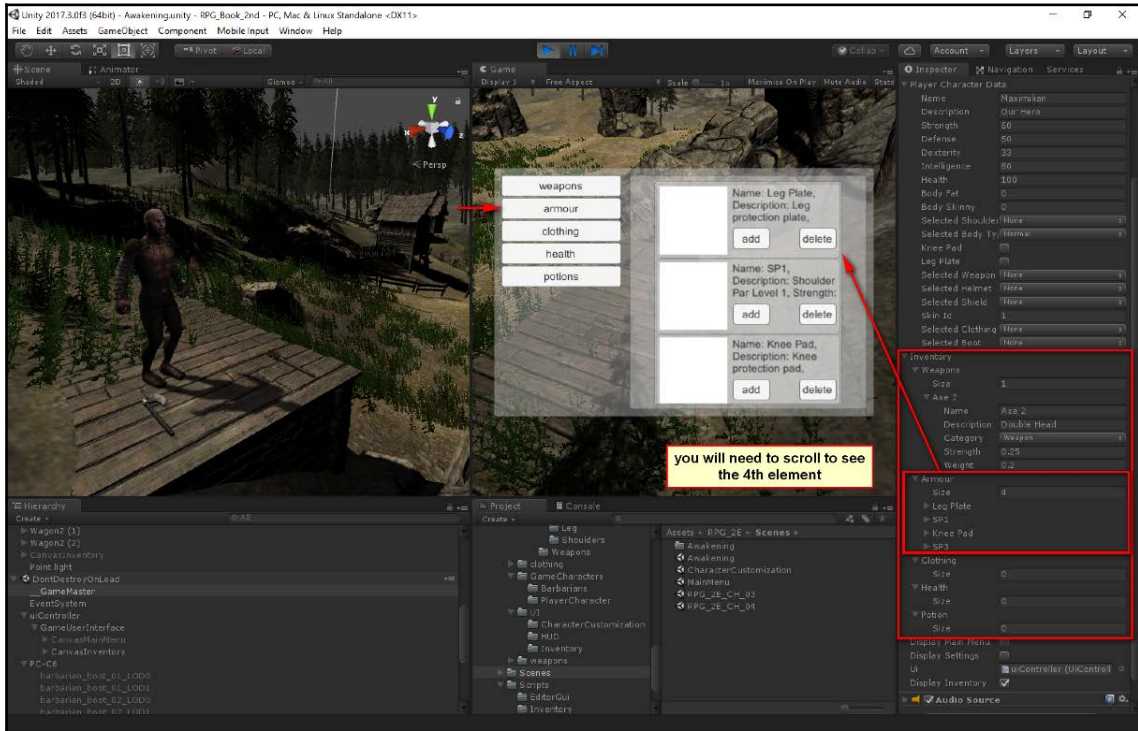
Item inserted into inventory - 1

Notice in the following screenshot that I have selected the `_GameMaster` GameObject to display the **Inventory** data in the **Inspector Window**:



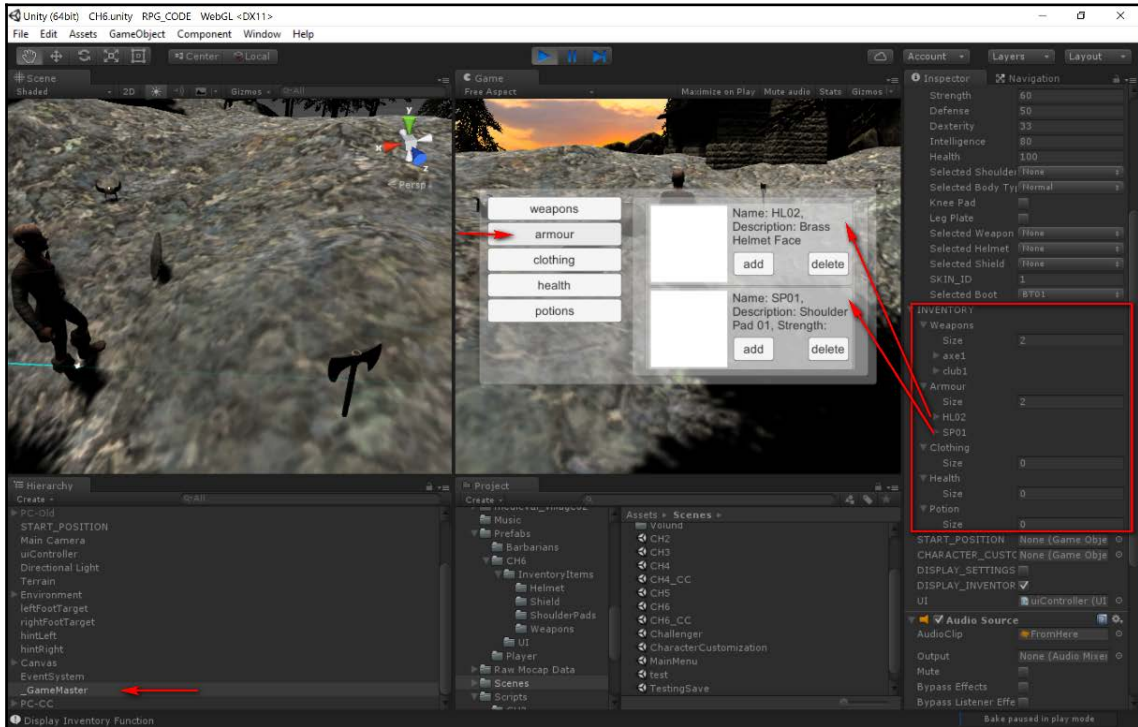
Item inserted into inventory - 2

We have picked up two weapon types and two armor types. The weapons items we have picked up are: **Axe 2**. The armor items we have picked up are: **Leg Plate**, **SP1**, **Knee Pad**, and **SP3**, as indicated in the **Inspector Window**. Take a look at the following screenshot:



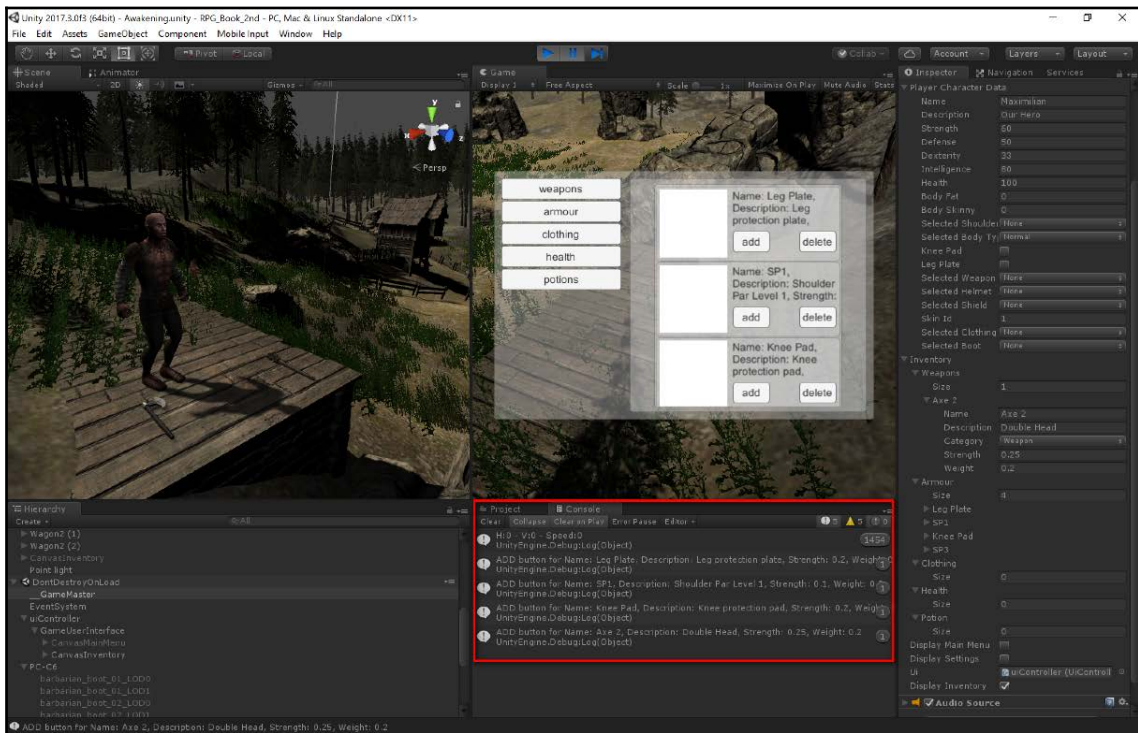
Item inserted into inventory - 3

Notice that in the **Game Window**, when we bring up the **Inventory Window** for display and click the **weapons** button, we get two listings. The listings display the proper data for each inventory item in the category. Take a look at the following screenshot:



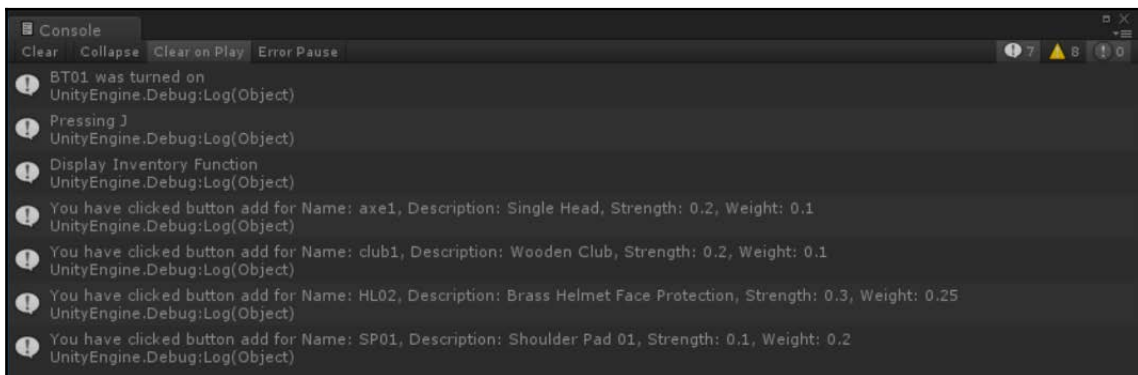
Armour category

To illustrate the `onClick()` event for the add button, please see the following screenshot from the **Console Window**:



Inventory item properties

To list the armor items in the inventory, we will click the **armour** button. The following screenshot displays the items in the **armour** category in the inventory based on our data:



We have come a long way. Let's take a moment to put things in perspective.

We first created the following scripts to lay the foundation for our inventory system in the game:

```
BaseItem.cs  
InventoryItem.cs  
InventoryItemAgent.cs  
InventorySystem.cs
```



The next step was to create the prefabs for each inventory item and add them to the `InventoryItemAgent.cs` script. This in turn allowed us to assign the necessary data to identify the prefab as an inventory item during game play.

Next, we started work on the design and development of the user interface for the inventory system. We created a sketch of how we would like the inventory window to look and implemented the framework using the build in UI architecture.

Slowly adding to the UI and applying different concepts and new elements, we built the final user interface for the inventory system.

Finally, we used the prefabs to test the complete addition and removal of the inventory items from the user interface.

The next challenge we face is how to actually apply the inventory items to the player character.

Inventory items and the player character

Now that we have seen how to create the inventory system, we need to be able to utilize it during game play to apply changes to our player character. In this section, we will examine how to do just that!

Here are some of the new features we need to work on:

- Applying a selected inventory item to the player character
- Performing accounting on both the player character and the inventory system
- Updating the game state accordingly

Applying inventory items

We need to make some design decisions on how we are going to handle applying the inventory items to the player character, and in turn how the system will handle the event. For instance, let's assume the player character has acquired several weapons; let's say Weapons A, B, and C.

Let's also assume that initially, the player does not have any active weapons. Now, the player selects to activate Weapon A. For this scenario, we would just use the inventory item data and activate Weapon A for the player, taking into consideration all of the accounting that comes with the weapon.

Now, the player wants to change his/her weapon to Weapon B because it is more powerful and they will need it to defeat the boss. Since the player already has Weapon A active, what are we going to do with it before we activate Weapon B? Do we put it back into the game world, or do we put it back into our inventory for later use?

In our case, once you have an item in the inventory, it will stay with you until you actually delete it from the inventory, in which case it will be destroyed. We need to make a few code modifications, and also some prefab modifications, to have everything working together.

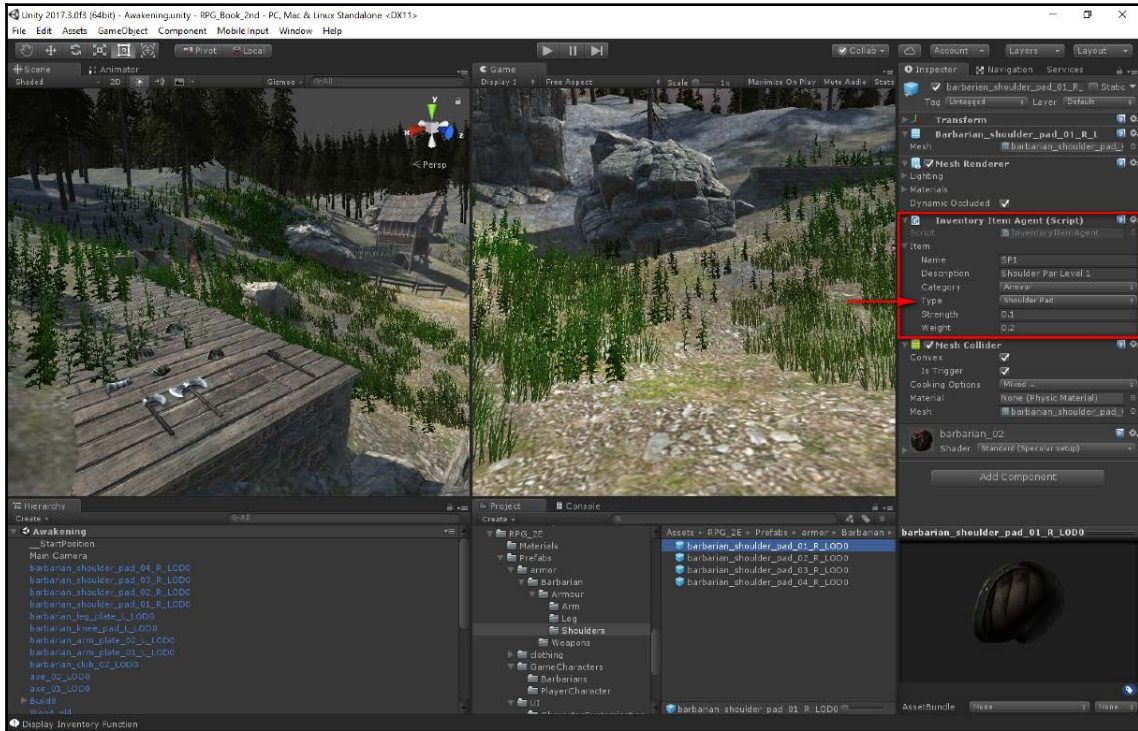
Let's start with the `InventoryItem.cs` script. We are going to add new data to store the type of the inventory item. This is necessary because we have a category, and within the category we have different types of items. This is especially true for the **Armour** category! For instance, we have a Helmet, a Shield, a Shoulder Pad, and so on.

The code listing is as follows:

```
using System;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class InventoryItem : BaseItem
    {
        public enum ItemType
        {
            Helmet = 0,
            Shield = 1,
            ShoulderPad = 2,
            KneePad = 3,
            Boots = 4,
            Weapon = 5
        }
        [SerializeField]
```

```
private ItemCatrgory category;
[SerializeField]
private ItemType type;
[SerializeField]
private float strength;
[SerializeField]
private float weight;
public ItemCatrgory Category
{
    get { return category; }
    set { category = value; }
}
public ItemType Type
{
    get { return type; }
    set { type = value; }
}
public float Strength
{
    get { return strength; }
    set { strength = value; }
}
public float Weight
{
    get { return weight; }
    set { weight = value; }
}
public void CopyInventoryItem(InventoryItem item)
{
    Category = item.Category;
    Description = item.Description;
    Name = item.Name;
    Strength = item.Strength;
    Weight = item.Weight;
}
}
```


When you make the update to your script, make sure to go back into the IDE and select the proper type for each prefab we have created, to represent your inventory items. Take a look at the following screenshot:



Inventory item type field

You will need to update the **Type** field for each prefab you have created for your inventory items.

We also need to update the `PlayerCharacter.cs` script. We are going to make the original data variables private and create public properties to access them. This way, if we need to perform any extra work prior to, or after, setting or getting the property value, we can do so easily.

The listing for the `PlayerCharacter.cs` script is as follows:

```
using System;
using UnityEngine;
namespace com.noorcon.rpg2e
{
```

```
public delegate void WeaponChangedEventHandler(PlayerCharacter.WeaponType
weapon);
[Serializable]
public class PlayerCharacter : BaseCharacter
{
    public enum ShoulderPad
    {
        none = 0,
        SP01 = 1,
        SP02 = 2,
        SP03 = 3,
        SP04 = 4
    };
    // Older version of the model
    public enum BodyType { normal = 1, BT01 = 2, BT02 = 3 };
    // New support for character model
    public float BodyFat = 0.0f;
    public float BodySkinny = 0.0f;
    // Shoulder Pad
    public ShoulderPad selectedShoulderPad = ShoulderPad.none;
    public BodyType selectedBodyType = BodyType.normal;
    public bool kneePad = false;
    public bool legPlate = false;
    public enum WeaponType
    {
        none = 0,
        axe1 = 1,
        axe2 = 2,
        club1 = 3,
        club2 = 4,
        falchion = 5,
        gladius = 6,
        mace = 7,
        maul = 8,
        scimitar = 9,
        spear = 10,
        sword1 = 11,
        sword2 = 12,
        sword3 = 13
    };
    [SerializeField]
    private WeaponType selectedWeapon = WeaponType.none;
    public WeaponType SelectedWeapon
    {
        get { return selectedWeapon; }
        set { selectedWeapon = value; }
    }
    public enum HelmetType { none = 0, HL01 = 1, HL02 = 2, HL03 = 3, HL04 = 4
```



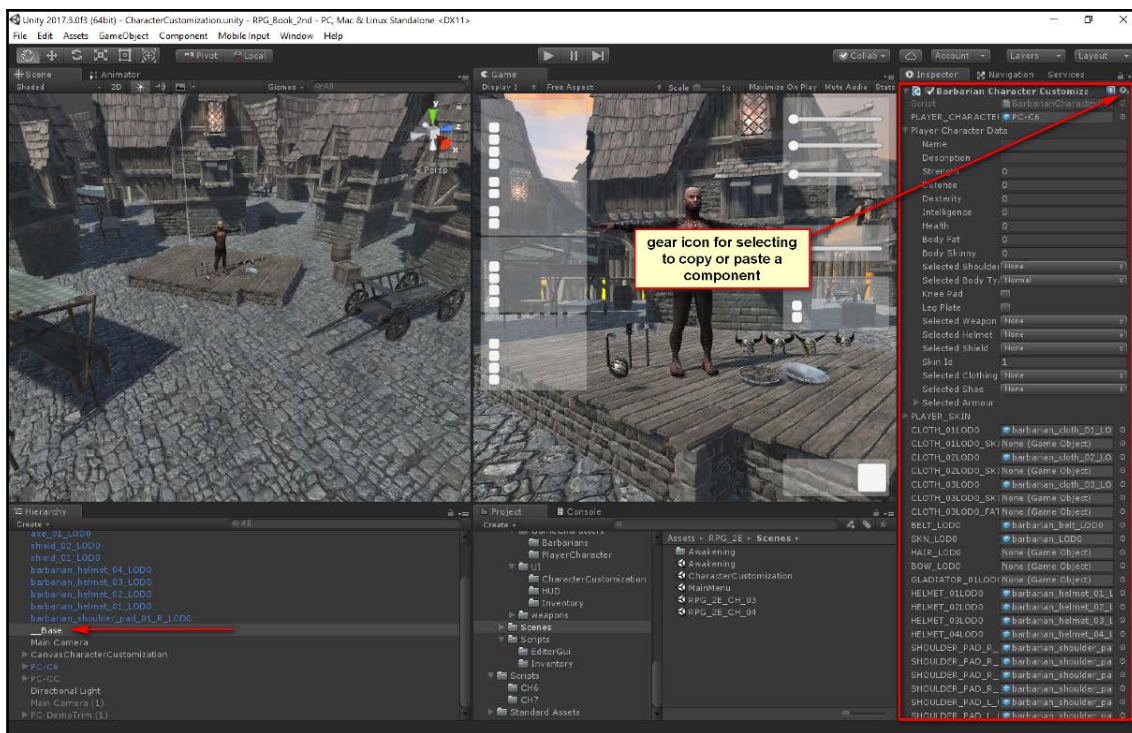
```
};
[SerializeField]
private HelmetType selectedHelmet = HelmetType.none;
public HelmetType SelectedHelmet
{
    get { return selectedHelmet; }
    set { selectedHelmet = value; }
}
public enum ShieldType { none = 0, SL01 = 1, SL02 = 2 };
[SerializeField]
private ShieldType selectedShield = ShieldType.none;
public ShieldType SelectedShield
{
    get { return selectedShield; }
    set { selectedShield = value; }
}
public int SkinId = 1;
public enum ClothingType { none=0, CT01=1, CT02=2, CT03=3, CT04=4 };
[SerializeField]
private ClothingType selectedClothing = ClothingType.none;
public ClothingType SelectedClothing
{
    get { return selectedClothing; }
    set { selectedClothing = value; }
}
public enum ShoeType { none = 0, BT01 = 1, BT02 = 2 };
[SerializeField]
private ShoeType selectedShoe = ShoeType.none;
public ShoeType SelectedShoe
{
    get { return selectedShoe; }
    set { selectedShoe = value; }
}
[SerializeField]
private InventoryItem selectedArmour;
public InventoryItem SelectedArmour
{
    get { return selectedArmour; }
    set { selectedArmour = value; }
}
}
```

The next code modification will be on the `BarbarianCharacterCustomization.cs` script. Since this script has been used in the character customization scene, we can utilize the same script and expand it to apply the visual changes to our player character. But before we can utilize this script, we will need to copy the actual component from the `_Base` GameObject defined in our character customization scene, and paste it into the `PC_C6` GameObject representing our player character!



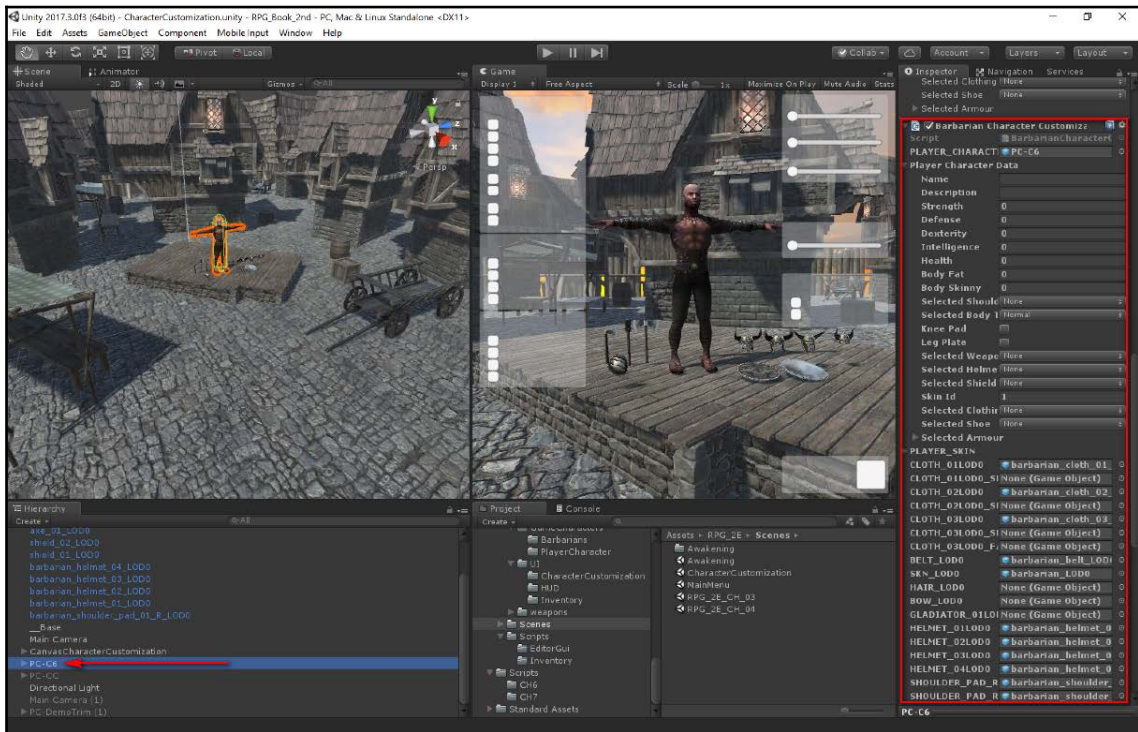
When you copy a component using the gear menu in the *Inspector Window*, all the configurations, links, and references stay intact! When you paste the component, again using the gear menu in the *Inspector Window*, you will have an exact copy of the component. This will eliminate the need for us to rewire all of the GameObjects to their references in the script.

The following two screenshots illustrate the copy of the component from the `_Base` GameObject to the `PC_C6` GameObject:



Using the gear icon to copy and paste a component

The `_Base` GameObject has the customization script attached to it, as illustrated in the preceding screenshot, copying the values from the `_Base` GameObject to transfer over to PC-C6 object. Take a look at the following screenshot:



Using the gear icon to copy and paste a component

This works because the script is actually referencing the different parts of the `PC_C6` GameObject hierarchy in the first place. The difference was that it used to be attached to the `_Base` GameObject for the customization.



Note: Can they both be active at the same time, in the same scene, at this point? Yes! However, if you have the time, you might want to redo your UI event triggers to use the `PC_CC` GameObject, and then you can remove the `BarbarianCharacterCustomization.cs` script from the `_Base` GameObject.

Now we actually need to modify the `BarbarianCharacterCustomization.cs` script to activate the different parts of the player character model using the data it will receive from the `GameMaster.cs` script.

A partial listing of the `CharacterCustomization.cs` script is as follows:

```
void DisableWeapons()
{
    AXE_01LOD0.SetActive(false);
    AXE_02LOD0.SetActive(false);
    CLUB_01LOD0.SetActive(false);
    CLUB_02LOD0.SetActive(false);
    FALCHION_LOD0.SetActive(false);
    GLADIUS_LOD0.SetActive(false);
    MACE_LOD0.SetActive(false);
    MAUL_LOD0.SetActive(false);
    SCIMITAR_LOD0.SetActive(false);
    SPEAR_LOD0.SetActive(false);
    SWORD_BASTARD_LOD0.SetActive(false);
    SWORD_BOARD_01LOD0.SetActive(false);
    SWORD_SHORT_LOD0.SetActive(false);
}

public void SetWeaponType(Slider id)
{
    try
    {
        PlayerCharacter.WeaponType weapon =
            (PlayerCharacter.WeaponType) Convert.ToInt32(id.value);

        PlayerCharacterData.SelectedWeapon = weapon;
    }
    catch
    {
        PlayerCharacterData.SelectedWeapon =
            PlayerCharacter.WeaponType.none;
    }

    // disable weapons
    DisableWeapons();

    switch (Convert.ToInt32(id.value))
    {
        case 0:
        {
            DisableWeapons();

            break;
        }
    }
}
```

```
case 1:
{
    AXE_01LOD0.SetActive(true);
    break;
}
case 2:
{
    AXE_02LOD0.SetActive(true);
    break;
}
case 3:
{
    CLUB_01LOD0.SetActive(true);
    break;
}
case 4:
{
    CLUB_02LOD0.SetActive(true);
    break;
}
case 5:
{
    FALCHION_LOD0.SetActive(true);
    break;
}
case 6:
{
    GLADIUS_LOD0.SetActive(true);
    break;
}
case 7:
{
    MACE_LOD0.SetActive(true);
    break;
}
case 8:
{
    MAUL_LOD0.SetActive(true);
    break;
}
case 9:
{
    SCIMITAR_LOD0.SetActive(true);
    break;
}
case 10:
{
    SPEAR_LOD0.SetActive(true);
```

```

        break;
    }
    case 11:
    {
        SWORD_BASTARD_LOD0.SetActive(true);
        break;
    }
    case 12:
    {
        SWORD_BOARD_01LOD0.SetActive(true);
        break;
    }
    case 13:
    {
        SWORD_SHORT_LOD0.SetActive(true);
        break;
    }
}
}

```

I have not listed the whole script as it would take many pages. But the basic concept is to overload the `SetXXXXX()` functions so that they will perform the necessary tasks based on the parameters coming in, as for the preceding example, `PlayerCharacter.WeaponType`.

The next script that needs to be modified is the `UiController.cs` script. This is where we will modify the five functions we created previously, to actually apply the changes to the player character. Let's look at one of the functions that have been modified, without listing the whole code, as follows:

```

public void DisplayWeaponsCategory()
{
    if(GameMaster.instance.DISPLAY_INVENTORY)
    {
        this.ClearInventoryItemsPanel();

        foreach (InventoryItem item in GameMaster.instance.INVENTORY.WEAPONS)
        {
            GameObject objItem =
            GameObject.Instantiate(this.InventoryItemElement) as GameObject;
            InventoryItemUI invItem = objItem.GetComponent<InventoryItemUI>();
            invItem.txtItemElement.text =
                string.Format("Name: {0}, Description: {1}, Strength: {2},
                Weight: {3}",
                                item.NAME,
                                item.DESRIPTION,
                                item.STRENGTH,
                                item.WEIGHT);
        }
    }
}

```

```
        invItem.item = item;

        // add button triggers
        invItem.butAdd.GetComponent<Button>().onClick.AddListener(() =>
        {
            Debug.Log(string.Format("You have clicked button add for {0},
{1}", invItem.txtItemElement.text, invItem.item.NAME));

            // let's apply the selected item to the player character
            GameManager.instance.PC_CC.SELECTED_WEAPON =
(PC.WEAPON_TYPE)Enum.Parse(typeof(PC.WEAPON_TYPE), invItem.item.NAME);
            GameManager.instance.PlayerWeaponChanged();
        });

        // delete button triggers
        invItem.butDelete.GetComponent<Button>().onClick.AddListener(() =>
        {
            Debug.Log(string.Format("You have clicked button delete for {0}",
invItem.txtItemElement.text));
            Destroy(objItem);
        });

        objItem.transform.SetParent(this.PanelItem);
    }
}
```

Notice that we are also saving the *item* from the `foreach` loop into the `invItem.item` variable. This is important, to make sure the `OnClick()` listener is picking up the current `InventoryItem` variable from the list.

The bulk of the work is done in `onClick.AddListener()` for each button. We basically set the selected weapon using `GameManager.instance` to store, and then we are calling the `PlayerWeaponChanged()` function to handle some more features. This will be demonstrated in the next code listing.



You will need to handle each add button listener in a similar fashion, based on how you have designed and implemented your code and your prefabs.

Finally, we are going to make some modifications to the `GameMaster.cs` script. The listing is as follows:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

using System.Collections;
using System;

public class GameMaster : MonoBehaviour
{
    public static GameMaster instance;

    // let's have a reference to the player character GameObject
    public GameObject PC_GO;

    // reference to player Character Customization
    public PC PC_CC;
    public InventorySystem INVENTORY;

    public GameObject START_POSITION;

    public GameObject CHARACTER_CUSTOMIZATION;

    public LevelController LEVEL_CONTROLLER;
    public AudioController AUDIO_CONTROLLER;

    // Ref to UI Elements ...
    public bool DISPLAY_SETTINGS = false;
    public bool DISPLAY_INVENTORY = false;

    public UIController UI;

    void Awake()
    {
        // simple singleton
        if (instance == null)
        {
            instance = this;

            instance.DISPLAY_INVENTORY = false;
            instance.DISPLAY_SETTINGS = false;

            // initialize Level Controller
            instance.LEVEL_CONTROLLER = new LevelController();

            // initialize Audio Controller
```



```

        instance.AUDIO_CONTROLLER = new AudioController();
        instance.AUDIO_CONTROLLER.AUDIO_SOURCE =
    GameManager.instance.GetComponent();
        instance.AUDIO_CONTROLLER.SetDefaultVolume();

        // initialize Inventory System
        instance.INVENTORY = new InventorySystem();

    }
    else if (instance != this)
    {
        Destroy(this);
    }

    // keep the game object when moving from
    // one scene to the next scene
    DontDestroyOnLoad(this);
}

....

```

The only function I want you to take a note of is the `PlayerArmourChanged()` function. It is because of this function that we have to add the new *Type* data variable and datatype to the `InventoryItem` class. Take a look at the following code:

```

    // for each level/scene that has been loaded
    // do some of the preparation work
    void OnLevelWasLoaded()
    {
        GameManager.instance.LEVEL_CONTROLLER.OnLevelWasLoaded();
    }

    // Use this for initialization
    void Start()
    {
        // let's find a reference to the UI controller of the loaded scene
        if (GameObject.FindGameObjectWithTag("UI") != null)
        {
            GameManager.instance.UI =
    GameObject.FindGameObjectWithTag("UI").GetComponent<UIController>();
        }

        GameManager.instance.UI.SettingsCanvas.gameObject.SetActive(GameManager.instance.DISPLAY_SETTINGS);
    }

    // Update is called once per frame
    void Update()

```

```
{
    // only when we are in the game level
    if(instance.LEVEL_CONTROLLER.CURRENT_SCENE.name==SceneName.Level_1)
    {
        if (Input.GetKeyUp(KeyCode.J))
        {
            //Debug.Log("Pressing J");
            instance.DISPLAY_INVENTORY = !instance.DISPLAY_INVENTORY;
            instance.UI.DisplayInventory();
        }
    }
}

public void MasterVolume(float volume)
{
    GameMaster.instance.AUDIO_CONTROLLER.MasterVolume(volume);
}

public void StartGame()
{
    GameMaster.instance.LoadLevel();
}

public void LoadLevel()
{
    GameMaster.instance.LEVEL_CONTROLLER.LoadLevel();
}

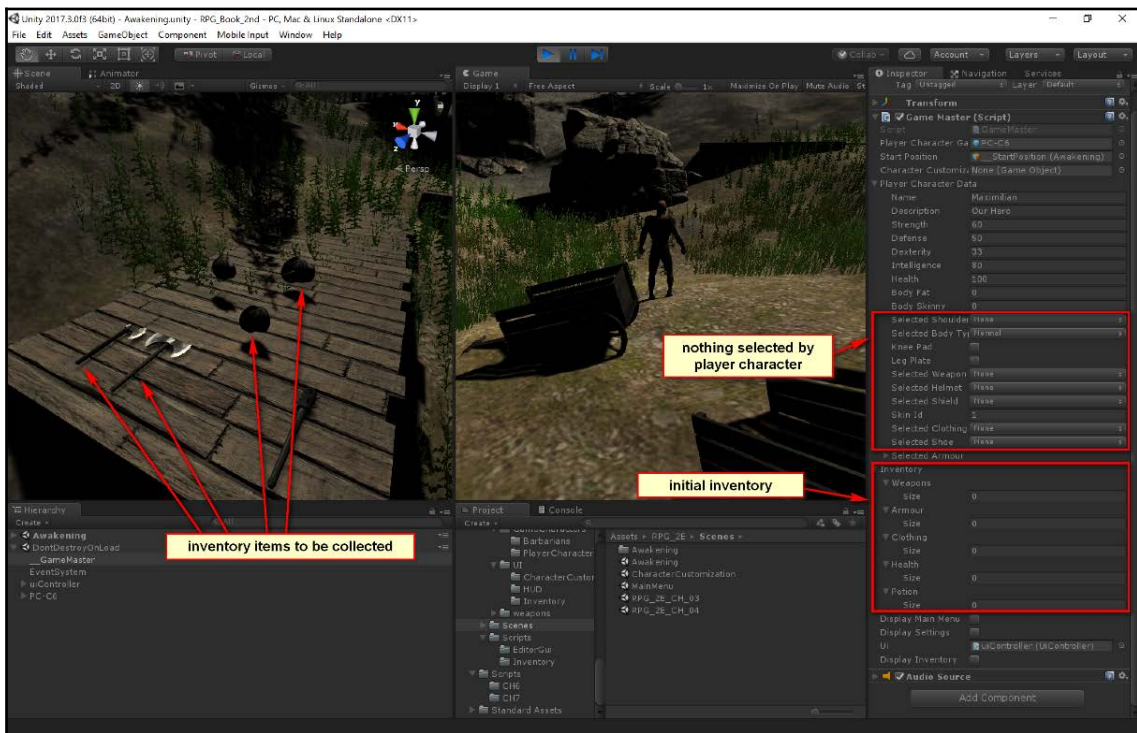
public void RPG_Destroy(GameObject obj)
{
    Destroy(obj);
}
}
```

We have a lot of different types of armor, and we needed a way to distinguish between them. Based on the armor type, we would then call the appropriate function to activate them on the player character.

How it looks

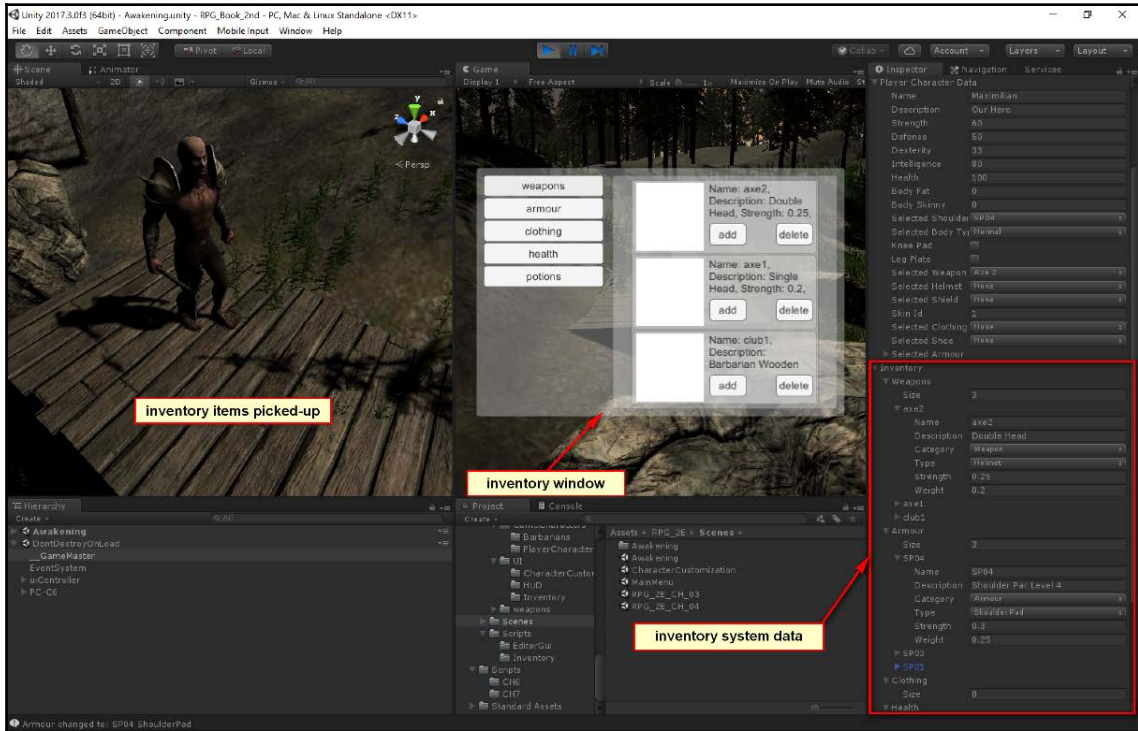
This chapter was a bit involved with regard to configuring your GameObjects and prefabs, and, more importantly, the code that went along to glue everything together. I have tried to keep things as simple as possible. Having said that, there are a few items that we have not discussed and incorporated in the chapter—the topics relating to leveling up, XP, and Quests. If you truly study and understand the Inventory System, you can very easily drive and expand it to cover all other parts. Unfortunately, we won't be able to implement everything in one book.

Following is a screenshot illustrating the player character prior to picking up the inventory items placed on the level. Notice that `INVENTORY` is empty in the `GameMaster` object. Also, there are no selected items in the `PC-C6` object:



Player character data

After I move the player character to pick up the inventory items, I will use the hot-key programmed to bring up the inventory window. In my case, the J key. The following screenshot captures the interaction:



Player Character Data Inventory

Now, let's look at how things change when we apply a few of the inventory items to the player character. You can see the changes applied in the preceding screenshot, in the inventory collection.

Summary

There is a lot covered in this chapter. The core of the chapter was to create a usable inventory system. We began the chapter by discussing weighted inventories, and gave a brief overview of the concept. We then determined the item types we are going to be using for our game.

We created the scripts `BaseItem.cs`, `InventoryItem.cs`, and `InventorySystem.cs`. These scripts were then utilized as a starting point to design and develop our inventory. We then updated the `GameMaster.cs` script, to test the basics of the newly created scripts and be able to view the data within the Unity IDE by serializing the attributes. We did this by instantiating an `InventoryItem` and inserting it into the `InventorySystem`, and validated the operation visually through the IDE.

The next step was to actually create the inventory item prefabs. This section covered how to navigate and find your customizable inventory items if your model included everything on the actual fbx model, how to extract it from the model, and how to convert it into a prefab for later use. Then we created a new script, called `InventoryItemAgent.cs`, which was attached to every prefab created to represent an inventory item. This script basically gives us the ability to set the data for each inventory item from within the IDE. Very useful! We also had to attach a collider to each prefab, to handle collision and trigger a pick-up call when the player collides with the object.

Once we had the basics in place, we started looking at how to design and implement the user interface for our inventory system. We discussed the categories that we want to represent, as well as how the items within each category will be listed/displayed for the player during game play. We implemented the initial framework for the inventory window and integrated it with the game.

Now we are ready to discuss how to create a dynamic item viewer that can be populated during runtime and represent the inventory items we have collected correctly. We introduced some new user interface concepts, such as scroll view and how to utilize layouts in our interface. We did a quick test with a simple placeholder to just display the name of the item. Once we got the mechanics working, we implemented the main inventory item control panel and converted it into a prefab, to be instantiated at runtime when needed.

Finally, we worked on the integration of the inventory system with the inventory UI, with the `GameMaster` and player character, to have the final implementation. This required that we update and modify more scripts.

By the end of the chapter, you have a fully functional inventory system that can be expanded as needed.

7

User Interface and System Feedback

Have you ever seen the top of an iceberg? Well, so far, that's what we have done throughout the previous six chapters. In this chapter, we will keep on improving on our game's user interface and feedback system. We will create a heads-up display that will be responsible for managing the user interaction with the system menus, as well as the system giving feedback to the player.

In this chapter, we will cover the following:

- Designing a heads-up display
- HUD basics
- Our design
- HUD framework
- Completing our HUD design
- Character info panel
- Active items panel
- Special items panel
- Integrating the code
- Enemy stats in the HUD
- NPC stats user interface
- Creating the NPC canvas
- NPC taking a hit
- Enhancing the code

Let's get started!

Designing a heads-up display

Designing a **Heads-Up Display (HUD)** is a very challenging task. The HUD is the interface through which your players can interact with the virtual world and receive feedback from the virtual world environment. As with everything else we have designed so far, the HUD design is also heavily related to the type and the needs of the game you are trying to make.

For instance, a **Real-Time Strategy (RTS)** game will have a very different type of HUD design to a **First-Person Shooter (FPS)** or a **Role-Playing Game (RPG)**. They will have some things in common, but their individual design is very distinctive, as well as some of the features and functionality.

We could have a whole book just about the design and development of user interfaces and how to approach them in a scientific manner. But that is outside the scope of this book, and we are concerned with the theories.

HUD basics

Any simple heads-up display will have, at a minimum, a way to display the following information:

- Basic information about the player character
- Health
- Mana
- Strength
- Level
- Current inventory items consumed by the player character
- Current weapon used by the player
- Current armor used by the player
- Available potions and/or health
- Feedback from the game environment
- Anything useful pertaining to the game
- Power ups
- Level ups

Let's go ahead and design our HUD. Once again, we will start with a simple framework and slowly build on top of it as and when we need to.

Our design

Taking everything into consideration, let's go ahead and design a HUD that will be useful for our game. At the same time, we will keep it simple but useful. We should have a HUD that will display the basic player character information in a manner that does not block the gameplay, but at the same time gives critical information to the player regarding their character's state.

We should also design a way to display the current inventory items the player has activated to be used, such as the weapon or the armor. Finally, we should also have a simple way for the player to use any health packets and/or potions they might have during gameplay.

The following figure shows a quick sketch of how I want my HUD to look. Again, you are free, and in fact, I encourage you, to come up with your own design:



I have clearly marked roughly how I would like my HUD to look during gameplay.

Notice that I have kept it simple. In the top-left corner, I have placed the immediate information that the player will need to have, such as their health and perhaps their strength.

In the bottom-left corner, I have placed a scroll-able panel that will list all of the active inventory items that the player may have active on their player character, and on the right-hand side of the screen I have three slots that will be used for immediate access to things such as health packets and/or potions that the player might need to use during their gameplay.

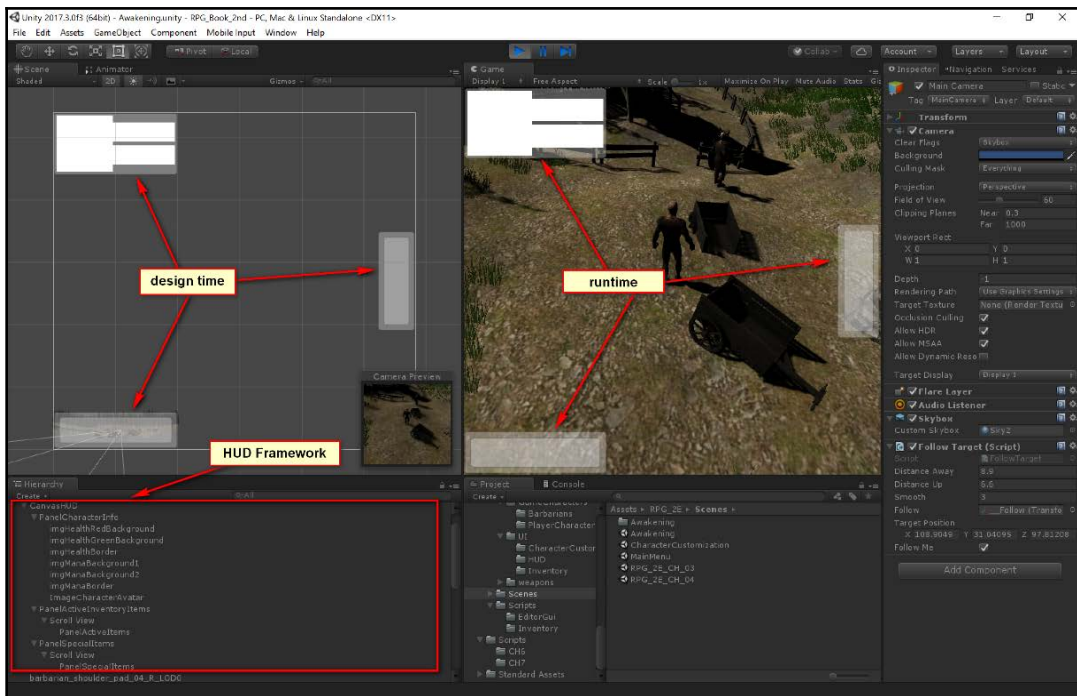
HUD framework

Now that we have an understanding of how we want our UI to look, let's start implementing it in Unity. We need to create a new canvas to hold our HUD. To create the canvas, right-click in the **Hierarchy** window and select **UI | Canvas**. Rename the new canvas `GameObject CanvasHUD`.

Let's go ahead and put into place all of the different sections for our UI. We will need three main panels for each section, as indicated in the following screenshot.

We need a panel to hold the character information in the top-left corner of the screen. We will also need a panel to hold the active inventory items in the bottom-left corner of the screen, and another panel to hold the special items on the right-hand side of the screen.

Create each panel by right-clicking the **Hierarchy** window and selecting **UI | Panel**. Make sure the panels are a child of the `CanvasHUD` `GameObject`. Rename each panel accordingly. I have named mine: `PanelCharacterInfo`, `PanelActiveItems`, and `PanelSpecialItems`. Take a look at the following screenshot:



Initial HUD outline

The preceding screenshot gives you a feeling of the HUD framework.

Completing our HUD design

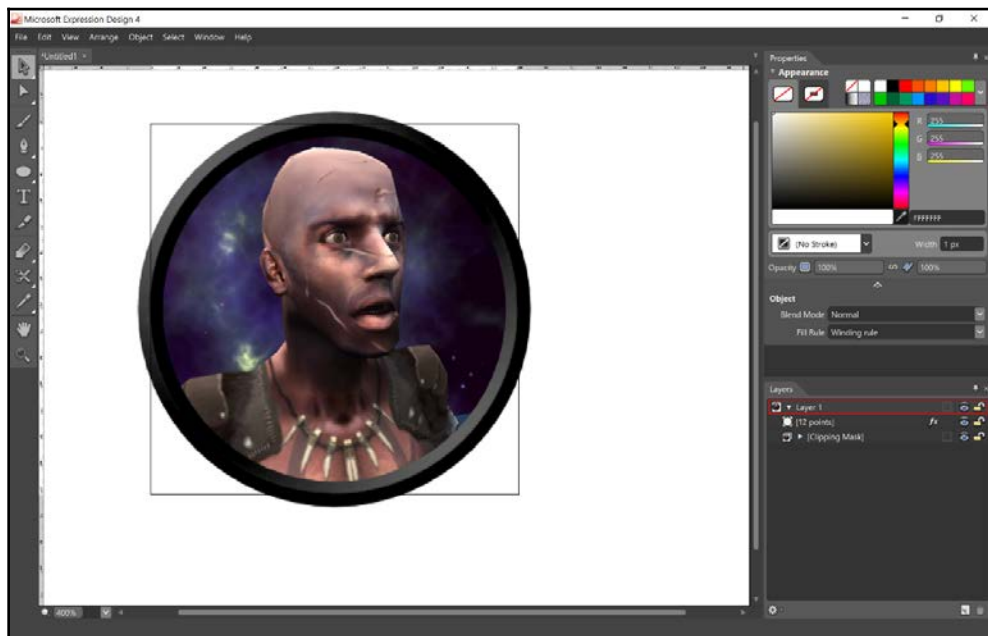
Now that we have the framework in place, let's go ahead and complete each section individually. I would like to start with the `PanelCharacterInfo`.

Character info panel

From a design point of view, the panel that will contain the visual components for the character will be very complex. The panel will be composed of five images.

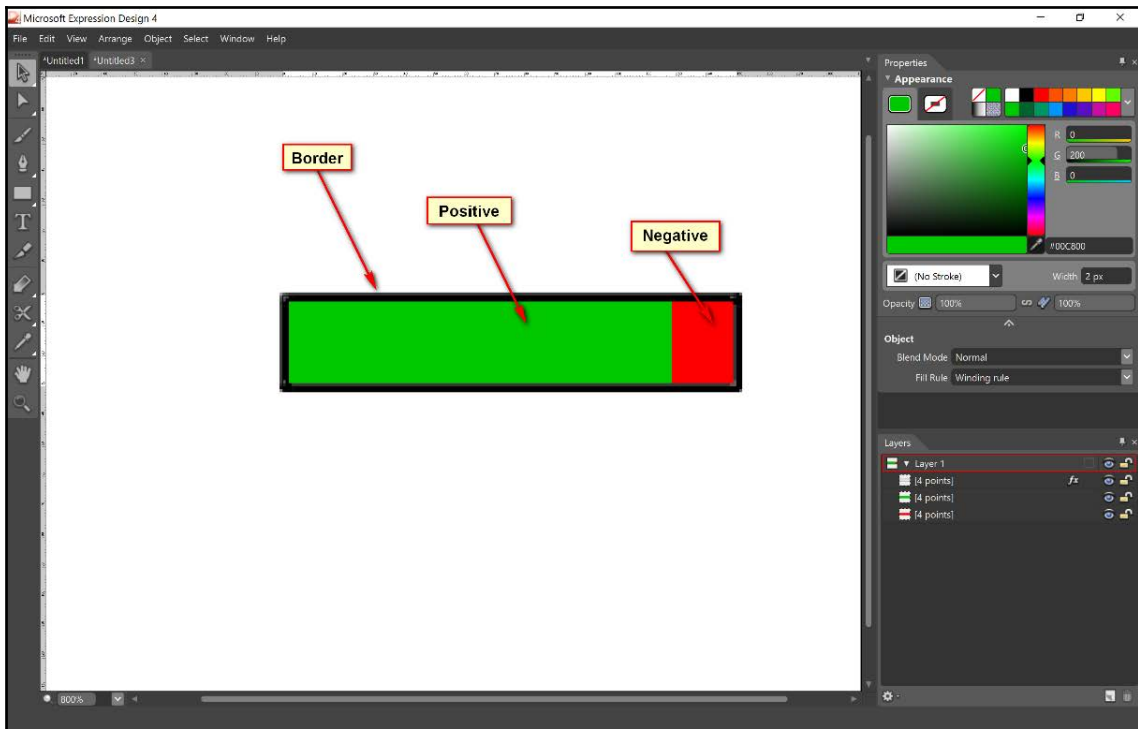
The main image will be used to hold an avatar of the character. The other four images will be used to display the health and mana of the character. Since these values will be displayed in a bar format, we will be using two images per item. One of the images will host the border, and the other the representation of the actual value.

To come up with the images, I will use external tools such as Photoshop; Microsoft Expression Design is a good tool for creating frames and so on. Take a look at the following screenshot:



In the preceding screenshot, I have made a nice image portraying the avatar of the player character. You should take into consideration the actual size of the image you will be placing inside the `PanelCharacterInfo` panel. The image size I have generated is 301 × 301 pixels.

To create the graphics for the bars representing the health and the mana for our character player, we will actually need to have three images. One image representing the negative value for the bar, one image representing the positive value of the bar, and the third will be the boarder image for the bar. They will be overlaid on top of each other to give the illusion of our graphic bars, as shown in the following screenshot:



Health bar graphics

Creating the three distinct sprites and overlaying them will give you a good illusion of what you are looking for.

After exporting our images, we need to import them into Unity. Use your file system to move your images from their original location to the `Assets` folder under your Unity project.

I have placed my textures in the following directory: Assets | RPG_2E | Textures.

Once you have moved them into the desired location within your Unity project, you will need to convert the images to sprites. Select all of the images that will be used for GUI, and from the **Inspector** window, change the **Texture Type** to **Sprite (2D and UI)**. Take a look at the following screenshot:

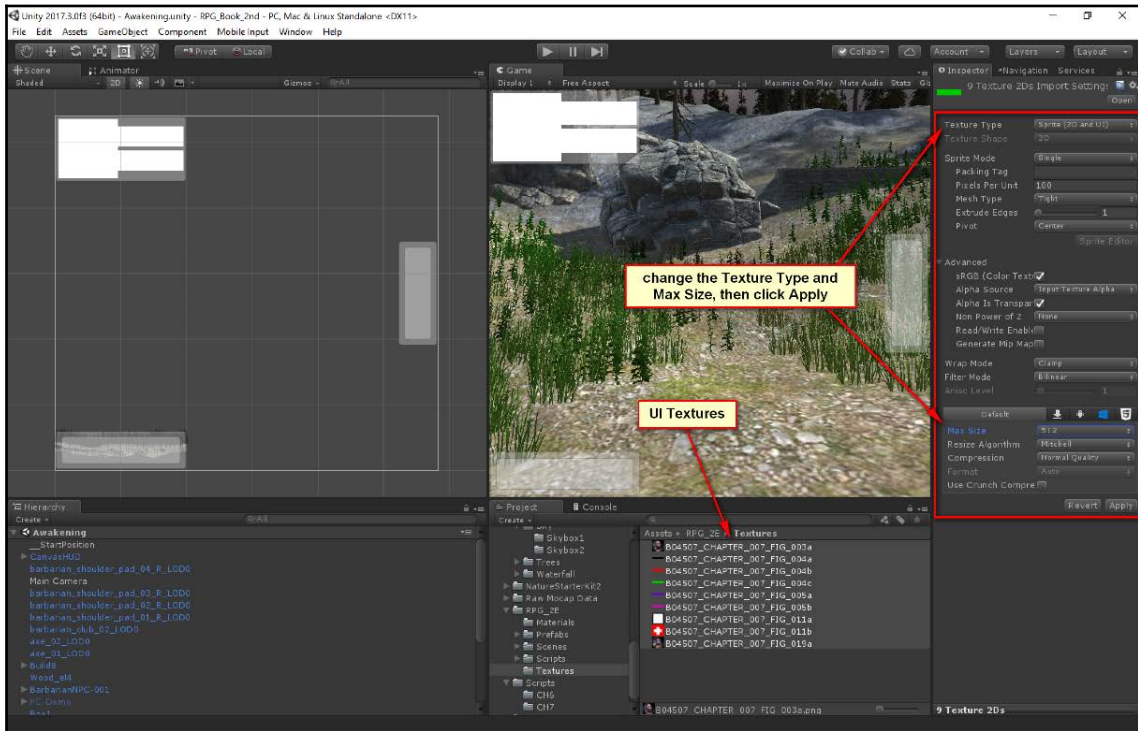


Image import properties

It's time to apply our textures to the actual UI elements we have defined under the `PanelCharacterInfo` panel within the Canvas object.

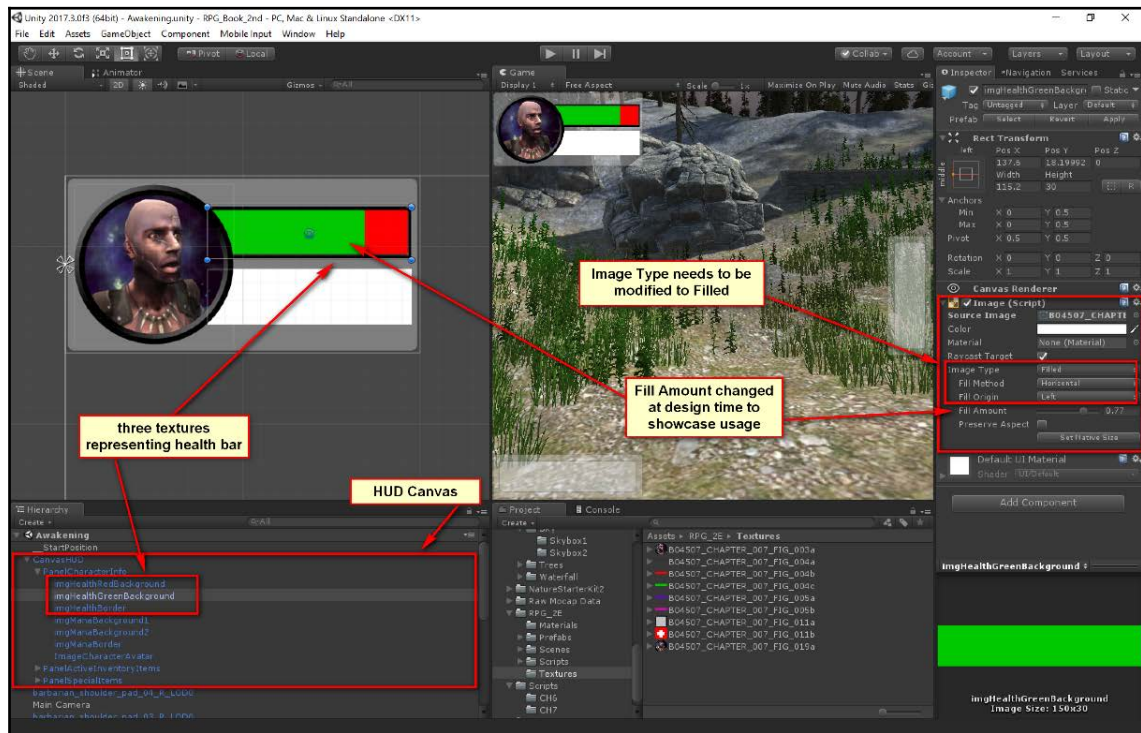


Note: There are a few steps that need to be performed before we can fully apply the UI elements to the HUD.

The first thing you should do if you have not done so already is creating three new UI image elements under the `PanelCharacterInfo` panel by right-clicking the panel and selecting **UI | Image**.

I have named my three images `imgHealthRebBackground`, `imgHealthGreenBackground` and `imgHealthBorder`. The order of the images does matter. You should take a note of it when you are designing the UI. Generally speaking, if a UI element is lower in the hierarchy, it will be rendered on top of the other elements.

Take a look at the following screenshot:



Health-bar hookup

Notice the order of the images representing the health bar. The image that represents the green bar will need to be modified using the **Inspector** window. Select it, and change the **Image Type** to **Filled**, change the **Fill Method** to **Horizontal** and the **Fill Origin** to **Left**. We will be using the **Fill Amount** to control the visual part of our health bar. Notice that I have set it to 0.77 for demonstration purposes.

By default, when the game starts, we will be starting at a **Fill Amount** of 1, which is equivalent to 100 percent for the player character's health. 0.77 is equivalent to 77 percent and so on.

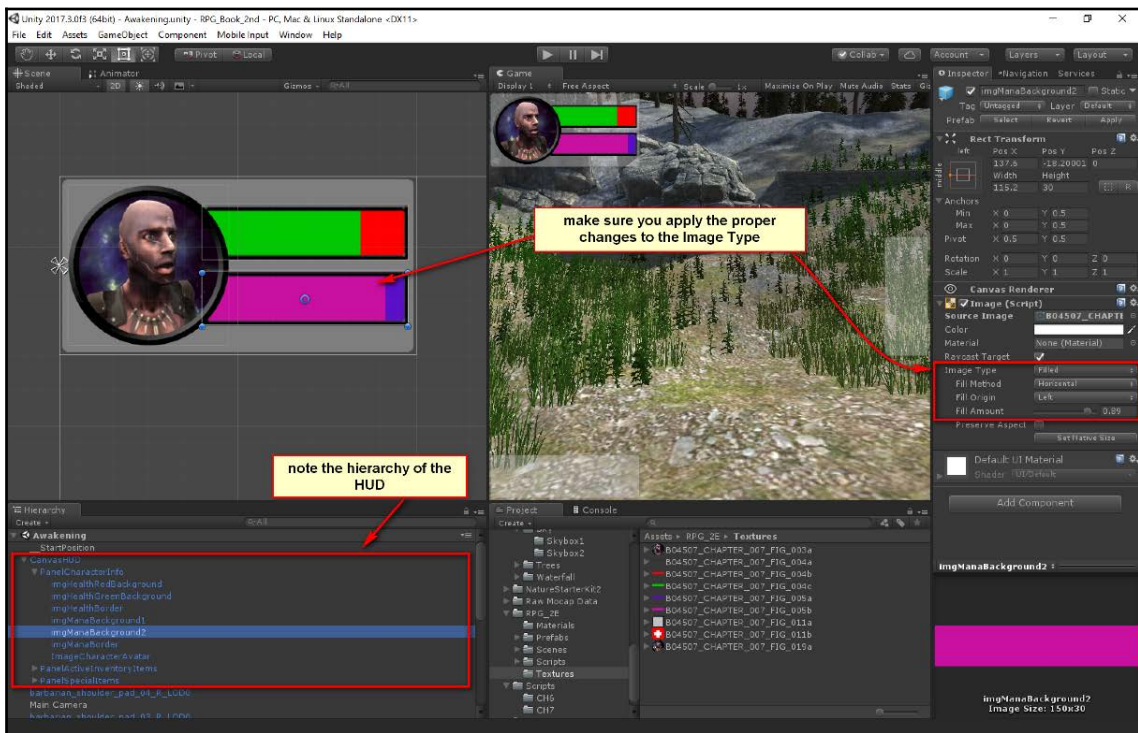
We will apply the same technique for our mana bar. Go ahead and create two more images, which will represent the two backgrounds for our mana bar.



Note: We will be using the same border image for both bars.

Again, don't forget you will need to make the appropriate changes to the imported textures within Unity. Convert them to **Sprite (2D and UI) Texture Type**.

Create the necessary image UI elements under the panel, and apply the textures to the image element within the canvas. You should have something like the following screenshot:



Mana-bar hookup

That's all there is to it! Not bad for a person with no artistic background!

Active inventory items panel

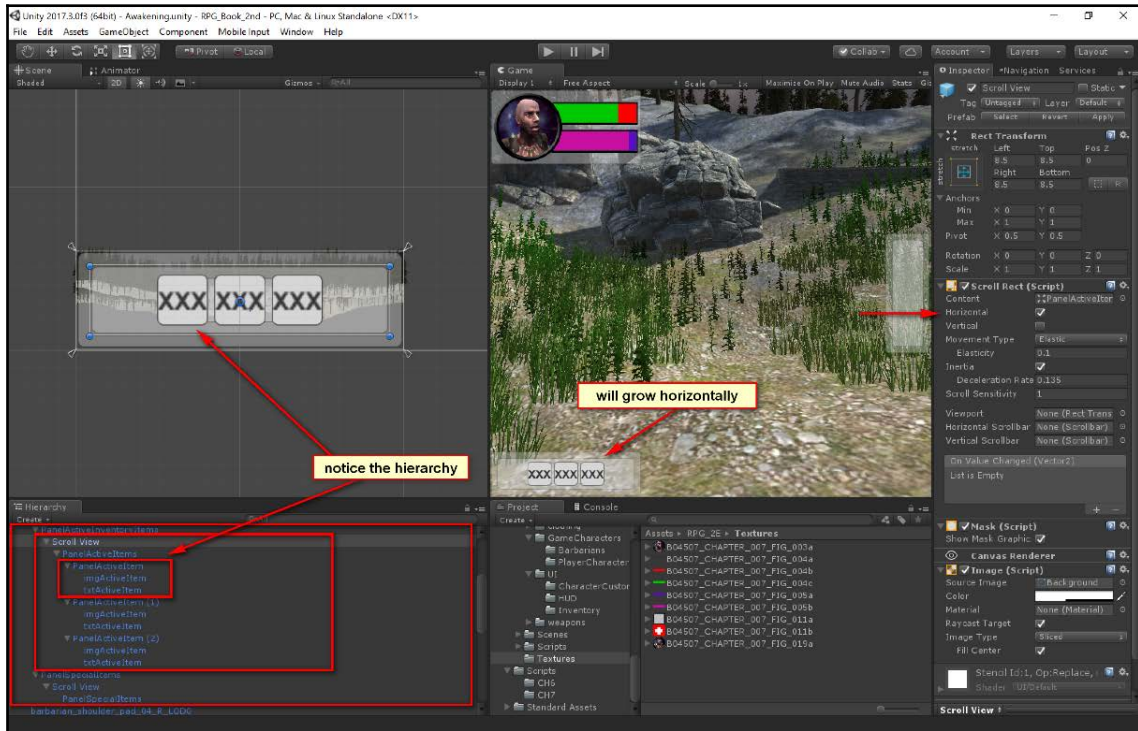
Creating the UI for active inventory items is similar to what we have done in [Chapter 6, *Inventory System*](#). The difference is that we will be listing only the items that have been consumed by the player character using the inventory system.

In other words, the **Active Inventory Items** display is a visual indication of the items that have been activated within the inventory. It is important to keep in mind that we are more interested in learning the concepts and applying them in a simple example that you can expand upon and improve on your own.

The basic idea is to create a scroll-able panel that will be used to add items as needed. We have already seen how to set up the scroll-able view and how to configure the UI components to support what we are trying to achieve. I won't be getting into the details here again; please refer to [Chapter 6, *Inventory System*](#) for the necessary steps if needed.

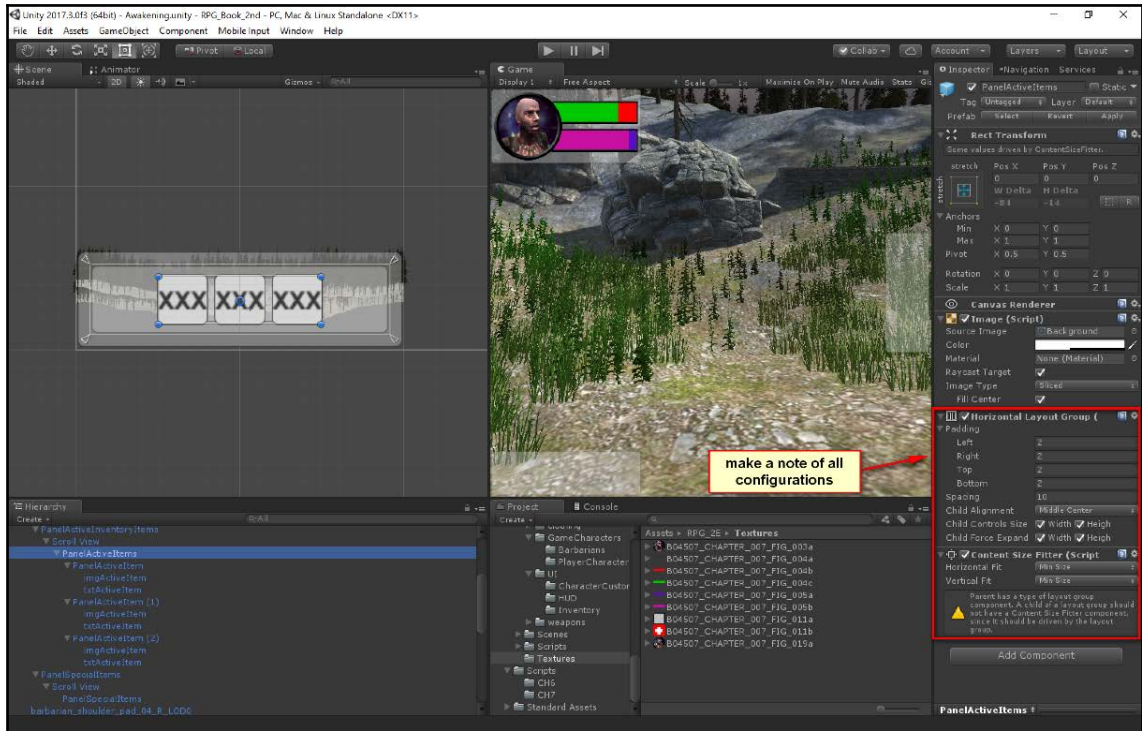
From the **Hierarchy** window right-click on **PanelActiveInventoryItems** and select **UI | Scroll View**. Go ahead and remove the `Viewport`, `Scrollbar Horizontal` and `Scrollbar Vertical` children that have been created with the `Scroll View` element.

You just need to make sure that the layout configuration you are applying is for horizontal and not vertical, as we did in Chapter 6, *Inventory System*. Take a look at the following screenshot:

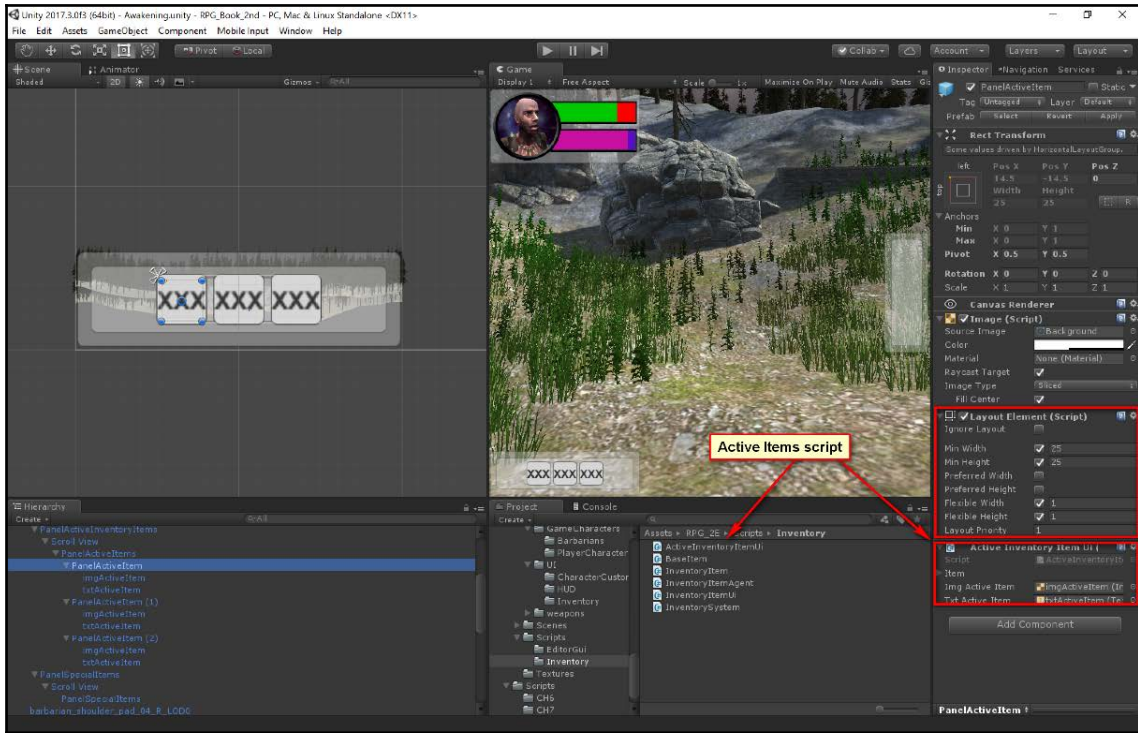


Active inventory panel

Double-check your anchors and alignments in the UI elements:



And finally, take a look at the following screenshot:



The three preceding figures illustrate the different parts of the configuration for the **Active Inventory Items** panel. If you are unsure of how to put this together, please go back to Chapter 6, *Inventory System*, and read the *Designing a Dynamic Item Viewer* section.

Don't forget to make a prefab of the UI element that will be representing your active inventory item in the panel.

You will also need a script to reference the UI elements designated for your items. I have called this script `ActiveInventoryItemUi.cs`, and currently, there are two attributes; one is a reference to the `Image` element and the other a reference to the `Text` element.

The listing for the script is as follows:

```
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class ActiveInventoryItemUi : MonoBehaviour
```

```

{
public InventoryItem item;
public Image imgActiveItem;
public Text txtActiveItem;
}
}

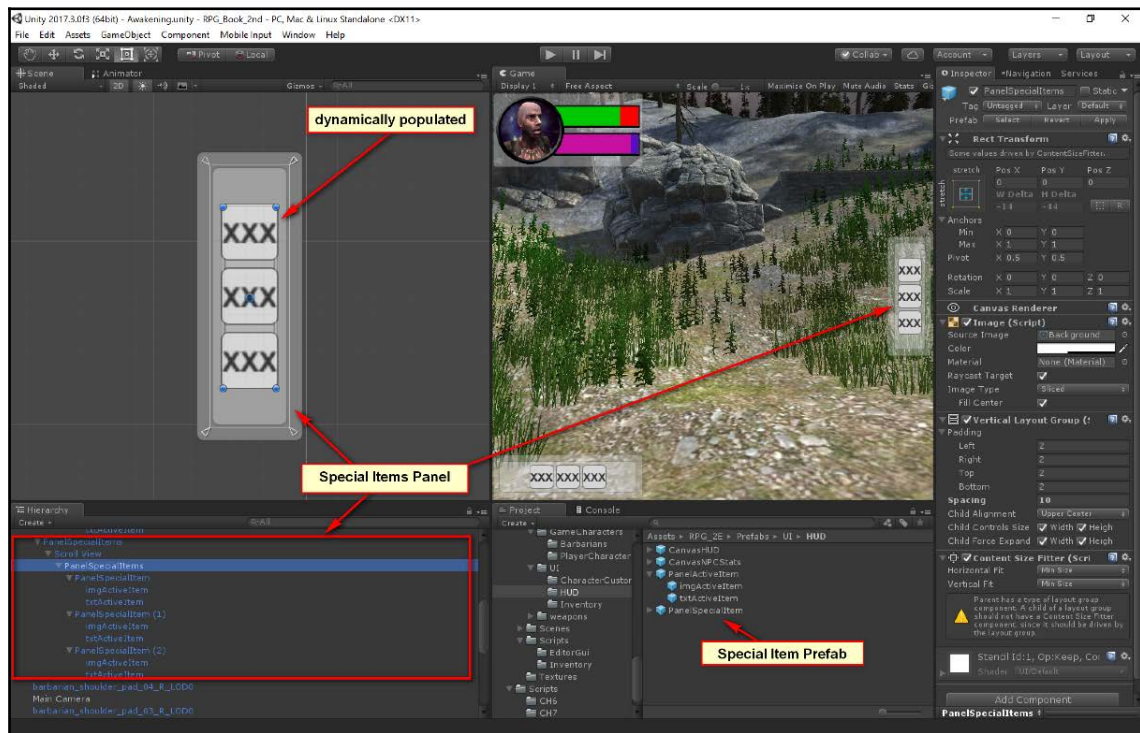
```

We will eventually need to integrate all of these scripts together to make things work properly.

Special items panel

Now we will look at the design of our last panel. The main difference between this panel and the last one we developed is the orientation. Everything else will be exactly the same. However, for this panel, our orientation will be vertical instead of horizontal.

Here is a screenshot capturing everything at once:



Special items panel

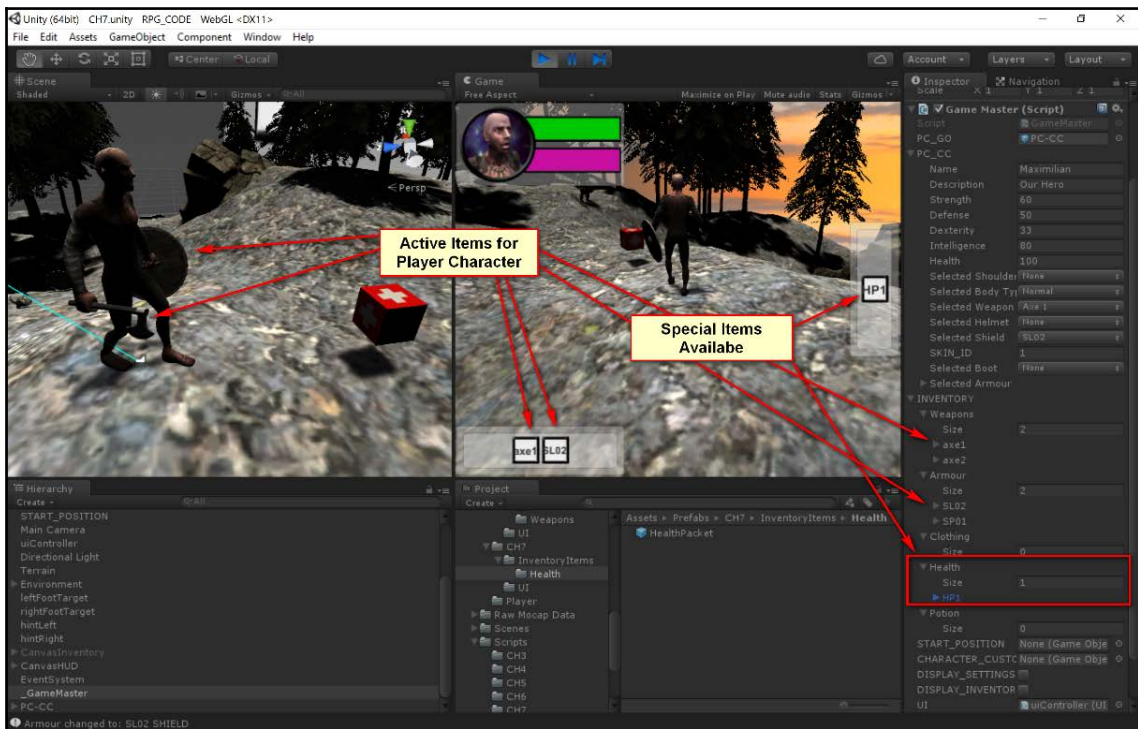
The procedure to create the panel has already been discussed several times, and you should not have any trouble creating it.



I have let you make your own textures and images to apply to the UI elements.

While designing the **Special Items** panel, I came up with a better idea of how to improve the panel UI. You might want to have a static icon representing each special item, and have a counter attached to the UI representing how many of each item you have. Each time you collect one, it will increase, and each time you consume one it will decrease.

The following screenshot shows how the current HUD looks based on our design:



Active inventory and special inventory runtime

We now need to start thinking about the integration of the HUD user interface with the code base we have developed so far.

Integrating the code

Now that we have our HUD design up and running, we will need to integrate the UI elements with the actual code that will be deriving them. There are a few scripts that will be created to support the new UI features and a few that will be updated to glue everything together.

The following scripts have been created: `ActiveInventoryItemUi.cs`, `ActiveSpecialItemUi.cs`, and `HudElementUi.cs`.

The listing of these scripts is as follows:

```
using UnityEngine.EventSystems;
namespace com.noorcon.rpg2e
{
    public class ActiveSpecialItemUi : EventTrigger
    {
        public override void OnPointerClick(PointerEventData data)
        {
            InventoryItem iia =
            gameObject.GetComponent<ActiveInventoryItemUi>().item;
            switch (iia.Category)
            {
                case BaseItem.ItemCatrgory.Health:
                {
                    // add the item to the special items panel
                    GameManager.instance.Ui.ApplySpecialInventoryItem(iia);
                    Destroy(gameObject);
                    break;
                }
                case BaseItem.ItemCatrgory.Potion:
                {
                    break;
                }
            }
        }
    }
}

using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class HudElementUi : MonoBehaviour
    {
        public Image imgHealthBar;
        public Image imgManaBar;
        public GameObject activeInventoryItem;
```

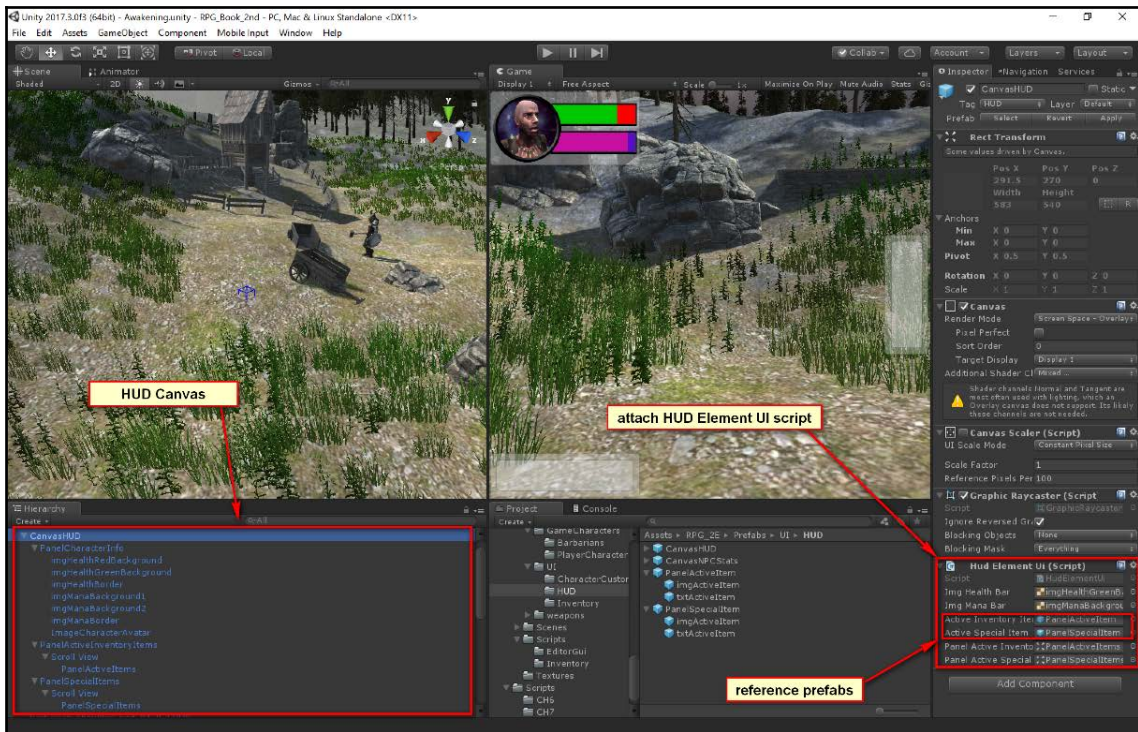


```

public GameObject activeSpecialItem;
public Transform panelActiveInventoryItems;
public Transform panelActiveSpecialItems;
}
}

```

These scripts will be used on the HUD user interface to give us access to the elements. For instance, you will need to attach the `HudElementUi.cs` script to the `CanvasHUD` GameObject. Take a look at the following screenshot:

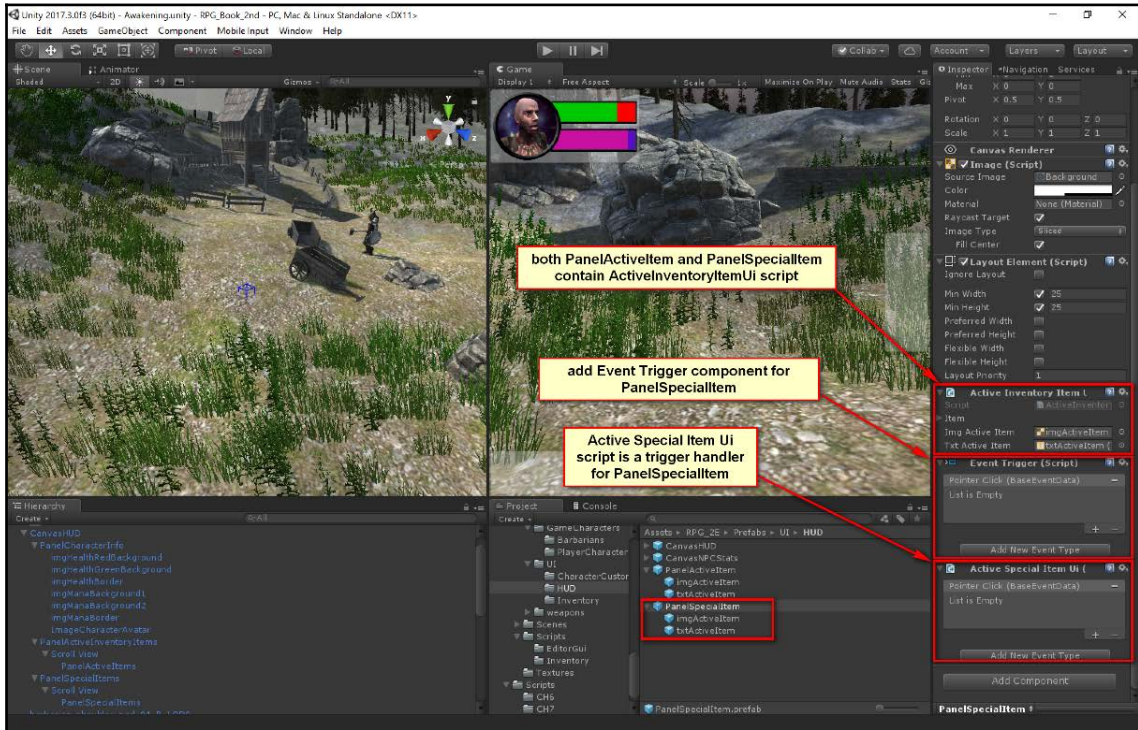


HUD canvas

The preceding screenshot illustrates how the HUD canvas is configured with the `HUDElementsUI.cs` script.

Now let's take a look at the prefabs we have created to represent the UI elements to be used for the panels. There are two; I have named them `PanelActiveItem` and `PanelSpecialItem`.

I will discuss `PanelSpecialItem` as it contains everything `PanelActiveItem` contains plus an additional script, which is attached to it for event handling. Take a look at the following screenshot:



HUD event triggers

What we have just covered was the implementation of the scripts that are used to have access to the proper UI elements within the HUD canvas.

You will notice that for the `PanelSpecialItem` prefab, we have two new and very important components that we have attached to it. One is the **Event Trigger** within Unity, and the other is the `ActiveSpecialItemUi.cs` script, which is used to handle the `PointerClick` event for the special item.

What this means is that we are basically making the item clickable, and when the player clicks on the item, something happens. In this case, it applies the special item to the player character.

Now we are ready to update the other scripts we have already developed to incorporate the HUD functionality. The scripts that will need to be modified are `InventorySystem.cs` and `UiController.cs`.

The listing of `InventorySystem.cs` is as follows:

```
using System;
using System.Collections.Generic;
using UnityEngine;
namespace com.noorcon.rpg2e
{
    [Serializable]
    public class InventorySystem
    {
        [SerializeField]
        private List<InventoryItem> weapons
        = new List<InventoryItem>();
        [SerializeField]
        private List<InventoryItem> armour
        = new List<InventoryItem>();
        [SerializeField]
        private List<InventoryItem> clothing
        = new List<InventoryItem>();
        [SerializeField]
        private List<InventoryItem> health
        = new List<InventoryItem>();
        [SerializeField]
        private List<InventoryItem> potion
        = new List<InventoryItem>();
        public List<InventoryItem> Weapons
        {
            get { return weapons; }
        }
        public List<InventoryItem> Armour
        {
            get { return armour; }
        }
    }
}
```



```
public List<InventoryItem> Clothing
{
    get { return clothing; }
}
public List<InventoryItem> Health
{
    get { return health; }
}
public List<InventoryItem> Potion
{
    get { return potion; }
}
private InventoryItem selectedWeapon;
private InventoryItem selectedArmour;
public InventoryItem SelectedWeapon
{
    get { return selectedWeapon; }
    set { selectedWeapon = value; }
}
public InventoryItem SelectedArmour
{
    get { return selectedArmour; }
    set { selectedArmour = value; }
}
public InventorySystem()
{
    ClearInventory();
}
```

The listing for `ClearInventory()` and `AddItem()` is as follows:

```
public void ClearInventory()
{
    weapons.Clear();
    armour.Clear();
    clothing.Clear();
    health.Clear();
    potion.Clear();
}
// this function will add an inventory item
public void AddItem(InventoryItem item)
{
    switch (item.Category)
    {
        case BaseItem.ItemCatrgory.Armour:
        {
            armour.Add(item);
            break;
        }
    }
}
```

```
}
case BaseItem.ItemCatrgory.Clothing:
{
clothing.Add(item);
break;
}
case BaseItem.ItemCatrgory.Health:
{
health.Add(item);
GameManager.instance.Ui.AddSpecialInventoryItem(item);
break;
}
case BaseItem.ItemCatrgory.Potion:
{
potion.Add(item);
break;
}
case BaseItem.ItemCatrgory.Weapon:
{
weapons.Add(item);
break;
}
}
}
```

DeleteItem() will remove a given InventoryItem from the list. See the following code:

```
// this function will remove an inventory item
public void DeleteItem(InventoryItem item)
{
switch (item.Category)
{
case BaseItem.ItemCatrgory.Armour:
{
armour.Remove(item);
break;
}
case BaseItem.ItemCatrgory.Clothing:
{
clothing.Remove(item);
break;
}
case BaseItem.ItemCatrgory.Health:
{
// let's find the item and mark it for removal
InventoryItem tmp = null;
foreach (InventoryItem i in this.health)
{
```

```
if (item.Category.Equals(i.Category)
&& item.Name.Equals(i.Name)
&& item.Strength.Equals(i.Strength))
{
    tmp = i;
}
}
health.Remove(tmp);
break;
}
case BaseItem.ItemCatrgory.Potion:
{
    // let's find the item and mark it for removal
    InventoryItem tmp = null;
    foreach (InventoryItem i in this.health)
    {
        if (item.Category.Equals(i.Category)
&& item.Name.Equals(i.Name)
&& item.Strength.Equals(i.Strength))
        {
            tmp = i;
        }
    }
    potion.Remove(tmp);
    break;
}
case BaseItem.ItemCatrgory.Weapon:
{
    weapons.Remove(item);
    break;
}
}
}
}
```

The listing for `UiController.cs` is as follows:

```
using System;
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class UiController : MonoBehaviour
    {
        [Header("Main Menu Canvas")]
        public RectTransform MainMenuCanvas;
        [Header("Settings Window")]
```

```
public RectTransform OptionsPanel;
public Slider ControlMainVolume;
public Slider ControlFXVolume;
[Header("Inventory Window")]
public RectTransform InventoryCanvas;
[Tooltip("root for inventory items")]
public Transform InventoryPanelItem;
[Tooltip("prefab representing invenotry item UI")]
public GameObject InventoryItemElement;
[Header("HUD Window")]
public RectTransform HudCanvas;
public HudElementUi HudUi;
...

public void DisplaySettings()
{
    GameManager.instance.DisplaySettings = !GameManager.instance.DisplaySettings;
    OptionsPanel.gameObject.SetActive(GameManager.instance.DisplaySettings);
}

public void MainVolume()
    GameManager.instance.MasterVolume(ControlMainVolume.value);
}

public void FXVolume()
{
    GameManager.instance.SoundFxVolume(ControlFXVolume.value);
}

#region INVENTORY UI FUNCTIONS
public void DisplayInventory()
{
    InventoryCanvas.gameObject.SetActive(GameManager.instance.DisplayInventory);
    Debug.Log("Display Inventory Function");
}

public void DisplayWeaponsCategory()
{
    if (GameManager.instance.DisplayInventory)
    {
        ClearInventoryPanelItems();
        foreach (InventoryItem item in GameManager.instance.Inventory.Weapons)
        {
            GameObject newItem
            = Instantiate(InventoryItemElement) as GameObject;
            InventoryItemUi InventoryItemControl
            = newItem.GetComponent<InventoryItemUi>();
            InventoryItemControl.ItemElementText.text =
            string.Format("Name: {0}, Description: {1}, Strength: {2}, Weight:
            {3}",
            item.Name,
            item.Description,
```

```

    item.Strength,
    item.Weight);
InventoryItemControl.Item = item;
// button triggers
InventoryItemControl.AddButton.GetComponent<Button>().onClick.AddListener((
) =>
{
    Debug.Log(string.Format("ADD button for {0}",
InventoryItemControl.ItemElementText.text));
// apply selected weapon
GameMaster.instance.PlayerCharacterData.SelectedWeapon
= (PlayerCharacter.WeaponType)Enum.Parse(
typeof(PlayerCharacter.WeaponType), InventoryItemControl.Item.Name);
GameMaster.instance.PlayerWeaponChanged();
AddActiveInventoryItem(InventoryItemControl.Item);
});
InventoryItemControl.DeleteButton.GetComponent<Button>().onClick.AddListene
r(() =>
{
    Debug.Log(string.Format("DELETE button for {0}",
InventoryItemControl.ItemElementText.text));
Destroy(newItem);
});
newItem.transform.SetParent(InventoryPanelItem);
}
}
}

```

A portion of script is listed as follows. Please refer to the download package for the full listing:

```

public void DisplayArmourCategory()
{
    if (GameMaster.instance.DisplayInventory)
    {
        ClearInventoryPanelItems();
        foreach (InventoryItem item in GameMaster.instance.Inventory.Armour)
        {
            GameObject newItem
            = Instantiate(InventoryItemElement) as GameObject;
            InventoryItemUi InventoryItemControl
            = newItem.GetComponent<InventoryItemUi>();
            InventoryItemControl.ItemElementText.text =
            string.Format("Name: {0}, Description: {1}, Strength: {2}, Weight:
            {3}",
            item.Name,
            item.Description,
            item.Strength,

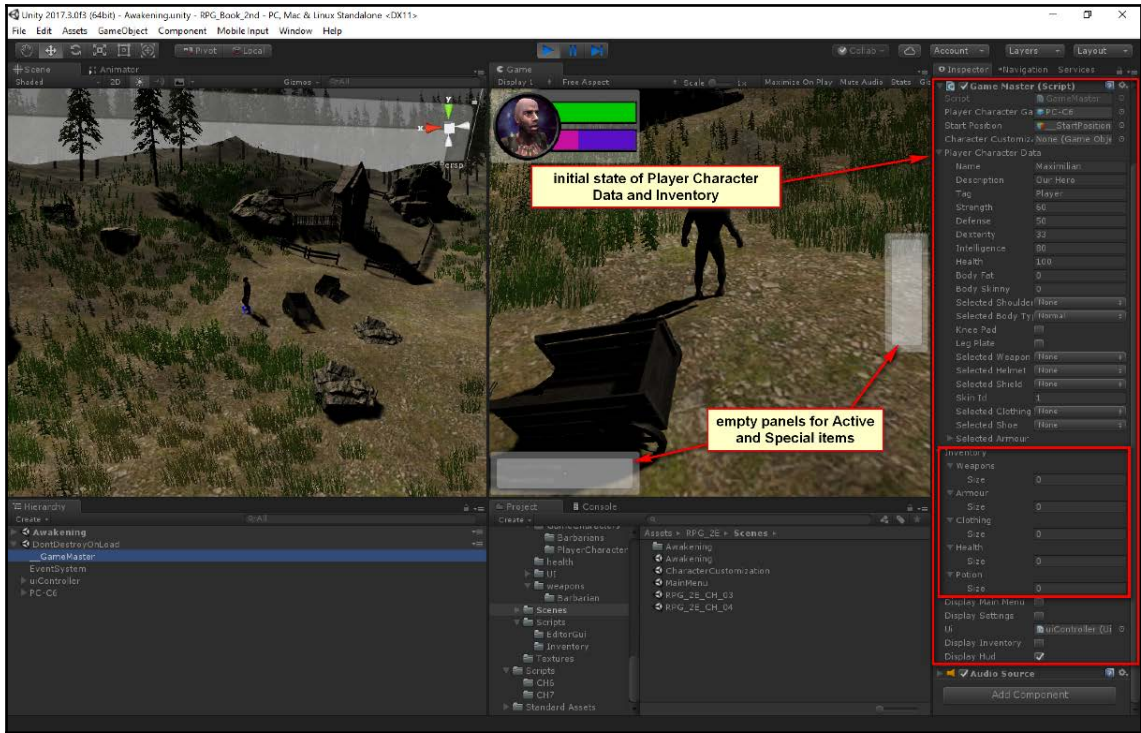
```

```
item.Weight);
InventoryItemControl.Item = item;
// button triggers
InventoryItemControl.AddButton.GetComponent<Button>().onClick.AddListener((
) =>
{
Debug.Log(string.Format("ADD button for {0}",
InventoryItemControl.ItemElementText.text));
// apply selected weapon
GameManager.instance.PlayerCharacterData.SelectedArmour =
InventoryItemControl.Item;
GameManager.instance.PlayerArmourChanged(InventoryItemControl.Item);
SetActiveInventoryItem(InventoryItemControl.Item);
});
InventoryItemControl.DeleteButton.GetComponent<Button>().onClick.AddListene
r(() =>
{
Debug.Log(string.Format("DELETE button for {0}",
InventoryItemControl.ItemElementText.text));
Destroy(newItem);
});
newItem.transform.SetParent(InventoryPanelItem);
}
}
}
...
```

Now that you have everything in place, you can go ahead and test run the game to make sure everything is working as expected. This is also a good time to test/debug your code and your project settings if you have not done so already.

I have to make the following point once again: the idea is to grasp the concept. We are looking at one way to implement what we want to achieve; you might come up with a better method along the way, or decide to do something totally different. I encourage that!

Take a look at the following screenshot:

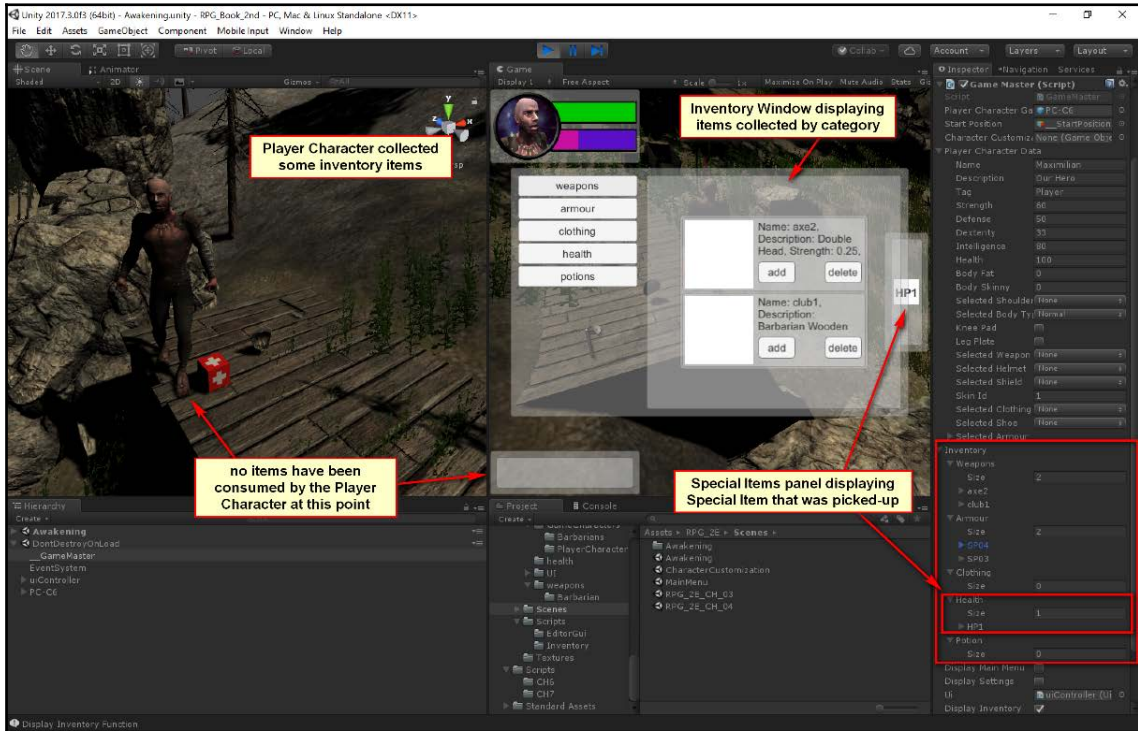


Initial state

The preceding screenshot illustrates the state of the player character and that of the inventory when the level initially loads. I have indicated the critical parts that we are testing and will track, to make sure our code works properly.

In the next figure, the player character has picked up a few of the inventory items we have laid on the level. When you bring up the **Inventory** window and click on any one of the categories defined, such as **weapons**, you will get a listing of all the weapons that we have in our inventory, and so forth.

We have collected one weapon type, one health packet, and a couple of defensive items. Note that our **Special Items** panel is displaying an item. This is the health packet we have picked up:

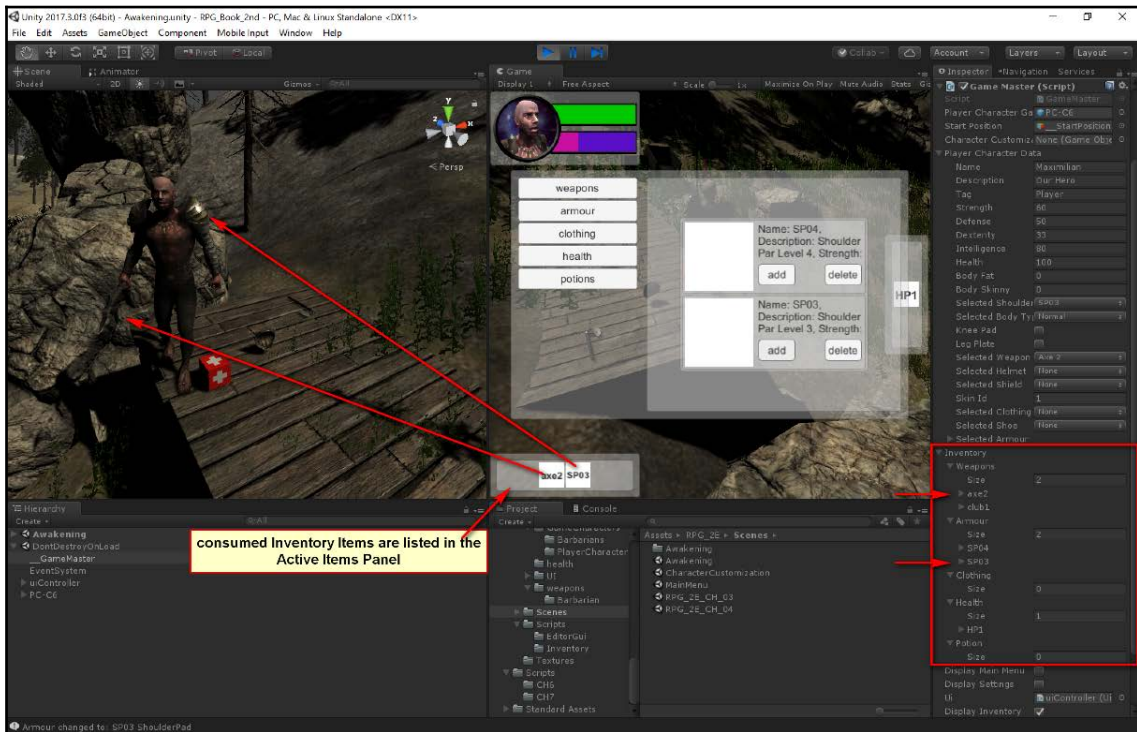


Special item interaction

The next screenshot illustrates how the HUD updates itself when the player starts to consume some of the inventory items by adding them using the **Inventory** window during gameplay.

Notice that we have activated three inventory items: two weapons, named **axe2** and **club1**, and two armors of type **SP04** and **SP03** respectively.

You can visually see them on the player character as well as the panel holding the active inventory items. Pretty cool:

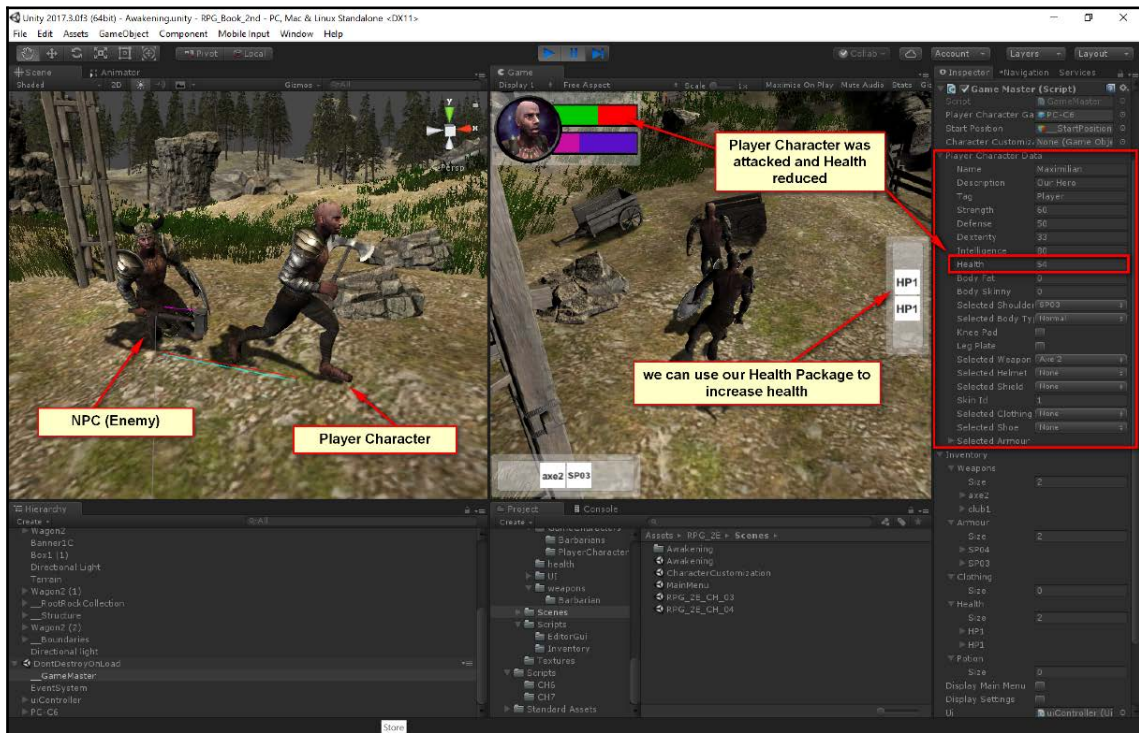


Active panel interaction

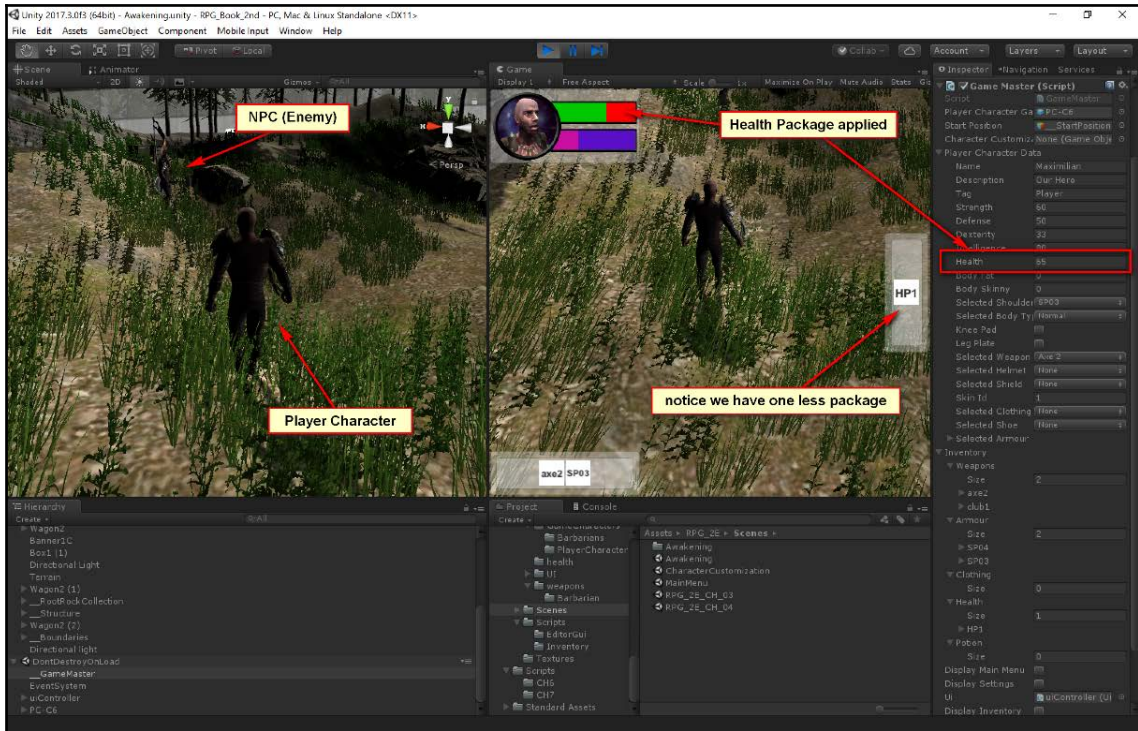
It's time to go and meet the enemy. We have not discussed much about the interaction between the player character and the **non-player characters (NPCs)**. We will do this shortly.

We have applied some of the inventory items from our inventory to the player character, and now we can actually go and face the enemy. We will allow the enemy to attack us, to see how our health reduces. Then we will use the health packet from our special items panel to increase our health once more.

The two following screenshots illustrate this scenario:



The following screenshot shows us running away and applying our health packet:



Notice how, when we apply the health packet, it removes itself from the inventory system as well as the special items panel.

Enemy stats in the HUD

We have not really discussed how to handle and manage the statistics and the visual representation of the NPC with the player. It is now time to do just that! We need to decide what it is that we want to display as information to the player. At the moment, let's keep it simple and just display the basic health and strength of the enemy.

The question is, what is the best way to display this information? Should we display the information based on a distance threshold between the player character and the NPC, or should we display it when the player requests it at some time during gameplay?

Let's go ahead and take the first scenario. We will display the information of the NPC when there is a certain distance between the player character and the NPC. We can even make this distance the same as the line of sight we have set for the NPC! This is good, because, if they can see us, then they are close enough for us to see their stats! Let's get to work!

NPC stats user interface

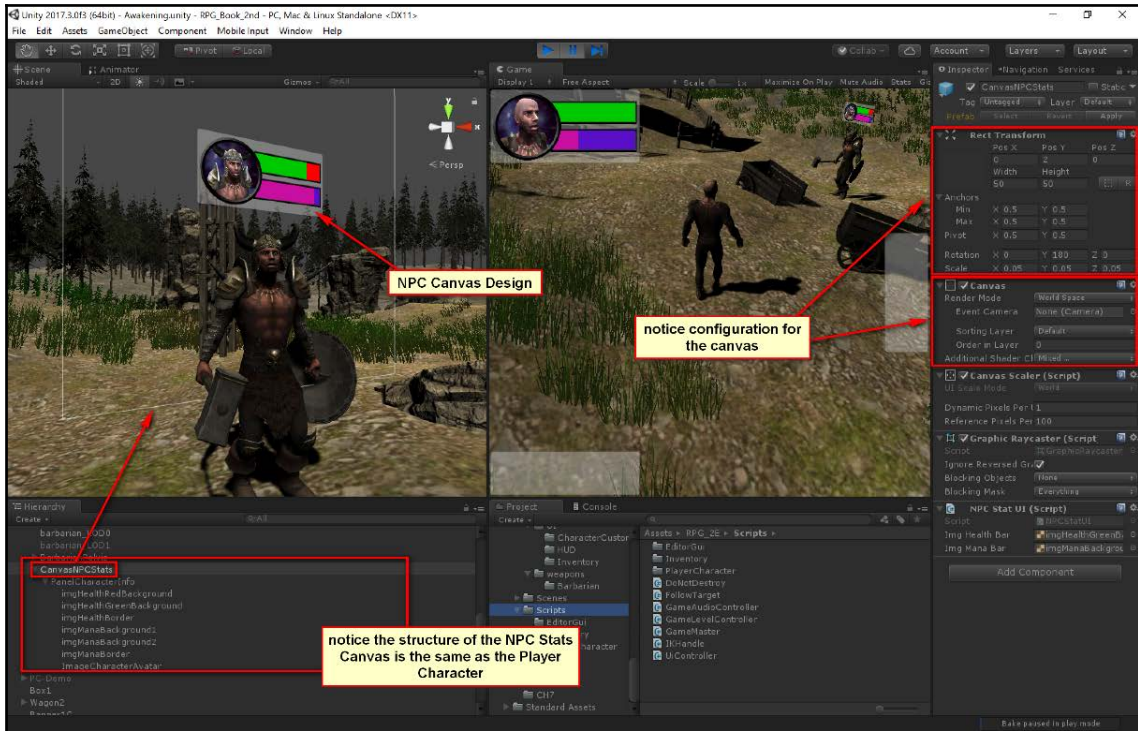
We will be using some of the existing textures that we have created for our player character, such as the textures for the health bar and strength bar. We just need to create a canvas that will be in the world space and attached to the NPC character.

Creating the NPC canvas

The main differences between the canvas we will create for the NPC and that of the one we have been creating for the player are some of the configurations.

One of the main differences will be the **Render Mode** of the canvas. The NPC canvas will have a **World Space Render Mode**. This will allow us to position the canvas as another `GameObject` within the scene. The next important difference will be the **Rect Transform** attributes and, more importantly, the **Scale** and **Rotation** attributes.

Take a look at the following screenshot:



NPC health bar

To make life easier, all you need to do is create the canvas and change the properties of the canvas as shown in the preceding screenshot. For the next step, you can copy the whole `PanelCharacterInfo` we developed in the previous sections, and paste it as a child of the new canvas. This way, you will not have to recreate each UI element one by one, which will save a lot of time. However, you will need to change the **Scale** and the **Transform** properties on the `PanelCharacterInfo` panel—the new one—to arrange it so that it renders above the NPC's head!

The next step is for us to be able to control the values of the stat bars from the code. For this, we will create a new script called `NPCStatUi.cs` and attach it to the canvas object we just created for the NPC stats.



I have renamed the canvas `CanvasNPCStats`.

The listing of the script is as follows:

```
using UnityEngine;
using UnityEngine.UI;
namespace com.noorcon.rpg2e
{
    public class NpcStatusUi : MonoBehaviour
    {
        public Image imgHealthBar;
        public Image imgManaBar;
    }
}
```

The script we just created will only give us a reference to the image elements. We still need to be able to have a method to update the values.

We need to find a way to reference all the NPC characters in a given scene. Once that is determined, we will need to set the initial values of the health and strength bar. Then, during gameplay, we will need to be able to update each NPC's stats according to the state of the game.

In order for us to identify the NPCs in a given scene, we will use the `Tag` element defined in each `GameObject`. We need to create a new `Tag` element named **Enemy**, and every NPC that is of enemy type will need to be tagged as such. This is an easy way to do a quick search and get a list of `GameObjects` based on their `Tag` value.

You should also start thinking about how are you going to dynamically attach the NPC stat canvas to the NPC at runtime. At the moment, for testing purposes, I will leave it attached to the model. But the question is, where do you actually attach it? Well, we have an empty `GameObject` named **Follow** attached to our model prefab. Since this is driven from our player character model, we have embedded **Follow** as a placeholder for the main camera during gameplay. For the NPC, we will use it to attach the NPC canvas as a child `GameObject` to the `Follow` `GameObject` in the model hierarchy. You can see these in the preceding screenshots.

We will use the `NpcAgent.cs` script to initialize the NPC status canvas prefab and the appropriate values of the UI elements. This is the best place to place the initialization because it will be self-contained. The new listing for the script is as follows:

```
using UnityEngine;
namespace com.noorcon.rpg2e
{
    public class NpcAgent : MonoBehaviour
    {
        [SerializeField]
        public Npc NpcData;
        [SerializeField]
        public Transform CanvasAttachmentPoint;
        [SerializeField]
        public Canvas CanvasNpcStats;
        [SerializeField]
        public GameObject CanvasNpcStatsPrefab;
        public void SetHealthValue(float value)
        {
            CanvasNpcStats.GetComponent<NpcStatusUi>().imgHealthBar.fillAmount =
            value;
        }
        public void SetStrengthValue(float value)
        {
            CanvasNpcStats.GetComponent<NpcStatusUi>().imgManaBar.fillAmount =
            value;
        }
        /// Use this for initialization
        void Start()
        {
            // let's go ahead and instantiate our stats
            GameObject tmpCanvasGO = Instantiate(
            CanvasNpcStatsPrefab,
            new Vector3(CanvasAttachmentPoint.position.x, 2, 0),
            CanvasNpcStatsPrefab.transform.rotation) as GameObject;
            tmpCanvasGO.transform.SetParent(CanvasAttachmentPoint, false);
            CanvasNpcStats = tmpCanvasGO.GetComponent<Canvas>();
            CanvasNpcStats.GetComponent<NpcStatusUi>().imgHealthBar.fillAmount = 1.0f;
            CanvasNpcStats.GetComponent<NpcStatusUi>().imgManaBar.fillAmount = 1.0f;
            Npc tmp = new Npc();
            tmp.Tag = "Enemy";
            tmp.CharacterGameObject = transform.gameObject;
            tmp.Name = "B1";
            tmp.Health = 100.0f;
            tmp.Defense = 50.0f;
            tmp.Description = "The Beast";
            tmp.Dexterity = 33.0f;
            tmp.Intelligence = 80.0f;
        }
    }
}
```

```
tmp.Strength = 60.0f;
NPCData = tmp;
}
//// Update is called once per frame
void Update()
{
    if (NPCData.Health < 0.0f)
    {
        NPCData.Health = 0.0f;
        transform.GetComponent<NPCBarbarianMovement>().die = true;
    }
}
}
```

Note that you will need to assign `canvasNPCStatsAttachment`, which will be used to store a reference to the `GameObject` we will attach to the NPC canvas.

`canvasNPCStatsPrefab` will be used to assign the prefab representing the NPC status canvas at design time. If you run the game now, you will have the prefab instantiated dynamically and attached to the `Follow` `GameObject` in the hierarchy with the fill values set to `1f`, that is, 100 percent.

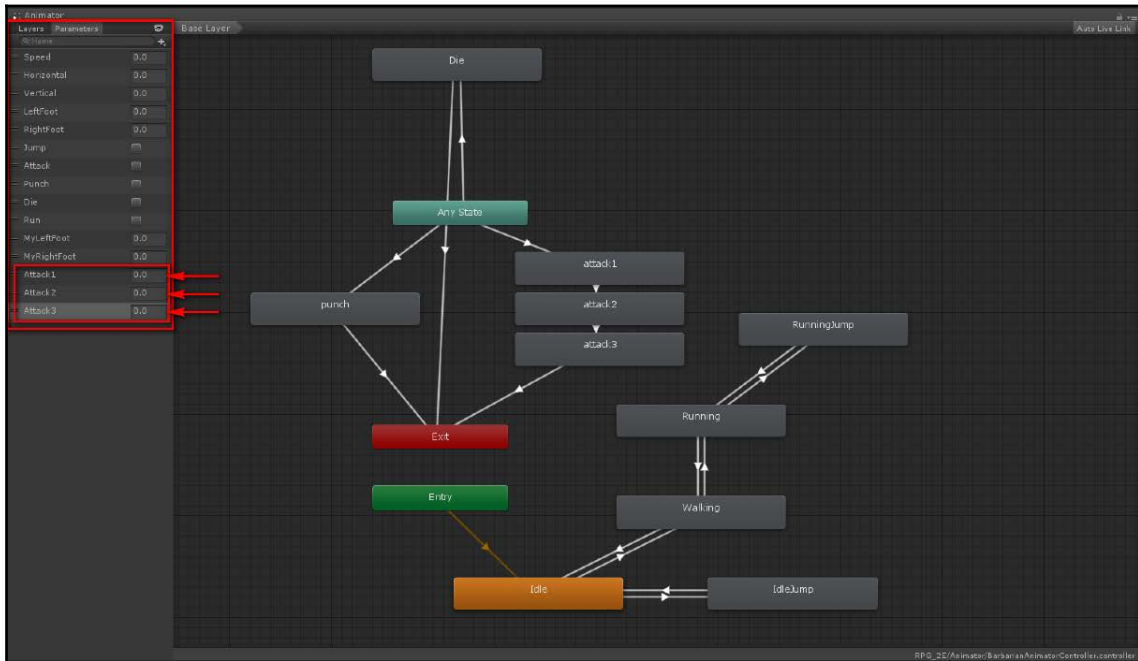
NPC taking a hit

We need to take a moment and go back to some of the initial scripts and configurations we did in earlier chapters, where we defined the player character's **Animator Controller** and `BarbarianCharacterController.cs` script.



Note: Please refer to Chapter 3, *RPG Character Design*, to refresh your memory on **Animator Controller** and **Curves**.

Open the animator controller we created in Chapter 3, *RPG Character Design*, named `BarbarianAnimatorController`. Select the **Parameters** tab and create new parameters called **Attack1**, **Attack2**, and **Attack3** of the `float` datatype. Take a look at the following screenshot:



Animator State Machine

For a refresher, go back to Chapter 4, *The Game Mechanics*, section *PC and NPC Interaction*, and you will recall how we defined and configured the curve to assign the parameter based on the animation.



We have defined the curve only for one of the attack animations.

Once you have configured the parameter on the **Animator Controller** for the player character, then we have to update the `BarbarianCharacterController.cs` script to trigger an attack based on the parameter value.

The following listing is a partial listing of the script, displaying just the modified portion:

```
void FixedUpdate()
{
    // The Inputs are defined in the Input Manager
    // get value for horizontal axis
    h = Input.GetAxis("Horizontal");
    // get value for vertical axis
    v = Input.GetAxis("Vertical");
    speed = new Vector2(h, v).sqrMagnitude;
    animator.SetFloat("Speed", speed);
    animator.SetFloat("Horizontal", h);
    animator.SetFloat("Vertical", v);
    if (animator.GetFloat("Attack1") == 1.0f)
    {
        GameManager.instance.AttackEnemy();
    }
}
```

In the code, we check to see if any of the attack modes are active and if so we check to see what the curve parameter **Attack1** is at the moment of the animation. If we are at 1.0f, then we call the `GameManager` object to perform the rest.

Now we need to take a look at a few functions we have defined/modified in the `GameManager.cs` script:

```
// for each level/scene that has been loaded
// do some of the preparation work
private void OnLevelWasLoaded(int level)
{
    instance.LevelController.OnLevelWasLoaded();
    // find all NPC GameObjects of Enemy type
    if (GameObject.FindGameObjectsWithTag("Enemy").Length > 0)
    {
        var tmpGONPCEnemy = GameObject.FindGameObjectsWithTag("Enemy");
        instance.NpcEnemyListGameObjects.Clear();
        foreach (GameObject goTmpNPCEnemy in tmpGONPCEnemy)
        {
            instance.NpcEnemyListGameObjects.Add(goTmpNPCEnemy);
        }
    }
}

public void AttackEnemy()
{
    Npc npc =
    instance.ClosestNpcEnemy.GetComponent<NpcAgent>().NpcData;
    npc.Health -= 1;
}
```

Some explanation is needed here. So the `OnLevelWasLoaded()` function is called each time a new scene is loaded at runtime. This is where we query all `GameObjects` that are tagged `Enemy`. We then store them internally, for further processing down the line.



For testing purposes and due to the simplicity of the scene, there is only one enemy present for testing. I am also setting the `closestNPCEntity` object to the last `GameObject` tagged `Enemy`. This variable is later used in the `PlayerAttachEntity()` function to set the NPC's `Health` property.

When the `PlayerAttackEntity()` function is called, we get a reference to the NPC component of the NPC character and reduce the health based on the attack.

Now, this also forces us to update the `BaseCharacter.cs` script. A listing of the modification is as follows:

```
[SerializeField]
private float health;
public float Health
{
    get { return health; }
    set
    {
        health = value;
        if (Tag.Equals("Player"))
        {
            if (GameManager.instance.Ui.HudUi != null)
            {
                GameManager.instance.Ui.HudUi.imgHealthBar.fillAmount
                = health / 100.0f;
            }
        }
        else
        {
            CharacterGameObject.GetComponent<NPCAgent>().SetHealthValue(health /
            100.0f);
        }
    }
}
```

In the `Health` property, we check to see if we are the player, or an NPC. If we are the player, we need to use `GameManager` to update our `Stats UI`, if we are going to update our own NPC `Stats UI`.

This means that when you create your player character and/or NPC, you will need to make sure you are assigning the data elements properly; see the following code:

```
void Awake()
{
    PlayerCharacter tmp = new PlayerCharacter();
    tmp.Name = "Maximilian";
    tmp.Tag = transform.gameObject.tag;
    tmp.CharacterGameObject = transform.gameObject;
    tmp.Health = 100.0f;
    tmp.Defense = 50.0f;
    tmp.Description = "Our Hero";
    tmp.Dexterity = 33.0f;
    tmp.Intelligence = 80.0f;
    tmp.Strength = 60.0f;
    playerCharacterData = tmp;
}
```

The preceding code snippet is the `Awake()` from the `PlayerAgent.cs` script. You will need to perform the same for the `NpcAgent.cs` script; see the following code:

```
void Start()
{
    // let's go ahead and instantiate our stats
    GameObject tmpCanvasGO = Instantiate(
        CanvasNpcStatsPrefab,
        new Vector3(CanvasAttachmentPoint.position.x, 2, 0),
        CanvasNpcStatsPrefab.transform.rotation) as GameObject;
    tmpCanvasGO.transform.SetParent(CanvasAttachmentPoint, false);
    CanvasNpcStats = tmpCanvasGO.GetComponent<Canvas>();
    CanvasNpcStats.GetComponent<NpcStatusUi>().imgHealthBar.fillAmount = 1.0f;
    CanvasNpcStats.GetComponent<NpcStatusUi>().imgManaBar.fillAmount = 1.0f;
    Npc tmp = new Npc();
    tmp.Tag = "Enemy";
    tmp.CharacterGameObject = transform.gameObject;
    tmp.Name = "B1";
    tmp.Health = 100.0f;
    tmp.Defense = 50.0f;
    tmp.Description = "The Beast";
    tmp.Dexterity = 33.0f;
    tmp.Intelligence = 80.0f;
    tmp.Strength = 60.0f;
    NpcData = tmp;
}
```

The code and scripts we have looked at have been used to test the ideas we have put forth. The results are positive. You might have noticed that when the player character attacks, we do not take into consideration its position relative to that of the enemy. We are also automatically assigning the closest NPC character in `GameManager` to eventually be the last element of the query we give each time a level loads.

Enhancing the code

One last code implementation I would like to make before I close out the chapter, is to make sure that when we are in attack mode for the player character, the hit points will affect the NPC that it is intended for automatically. In other words, determine which NPC is closest to us based on our distance and view angle toward the NPC.

We have already created the logic to determine these quantities for the NPC character. We now need to implement something similar for the player character. Let's take a look at a partial listing of the code changes we need to make for the `BarbarianCharacterMovement.cs` script:

```
using UnityEngine;
namespace com.noorcon.rpg2e
{
    public class BarbarianCharacterController : MonoBehaviour
    {
        public Animator animator;
        public float directionDampTime;
        public float speed = 6.0f;
        public float h = 0.0f;
        public float v = 0.0f;
        bool attack = false;
        bool punch = false;
        bool run = false;
        bool jump = false;
        [HideInInspector]
        public bool die = false;
        bool dead = false;
        public bool EnemyInSight;
        public GameObject EnemyToAttack;
        Quaternion StartingAttackAngle = Quaternion.AngleAxis(-25, Vector3.up);
        Quaternion StepAttackAngle = Quaternion.AngleAxis(5, Vector3.up);
        Vector3 AttackDistance = new Vector3(0, 0, 2);
        // Use this for initialization
        void Start()
        {
            animator = GetComponent<Animator>() as Animator;
```

```
EnemyInSight = false;
}
...
void FixedUpdate()
{
    // The Inputs are defined in the Input Manager
    // get value for horizontal axis
    h = Input.GetAxis("Horizontal");
    // get value for vertical axis
    v = Input.GetAxis("Vertical");
    speed = new Vector2(h, v).sqrMagnitude;
    animator.SetFloat("Speed", speed);
    animator.SetFloat("Horizontal", h);
    animator.SetFloat("Vertical", v);
    #region used for attack range
    RaycastHit hitAttack;
    var angleAttack = transform.rotation * StartingAttackAngle;
    var directionAttack = angleAttack * AttackDistance;
    var posAttack = transform.position + Vector3.up;
    for (var i = 0; i < 10; i++)
    {
        Debug.DrawRay(posAttack, directionAttack, Color.yellow);
        if (Physics.Raycast(posAttack, directionAttack, out hitAttack, 1.0f))
        {
            var enemy = hitAttack.collider.GetComponent<NPCAgent>();
            if (enemy)
            {
                //Enemy was seen
                EnemyInSight = true;
                EnemyToAttack = hitAttack.collider.gameObject;
                GameManager.instance.ClosestNPCEnemy = hitAttack.collider.gameObject;
            }
            else
            {
                this.EnemyInSight = false;
            }
        }
        directionAttack = StepAttackAngle * directionAttack;
    }
    #endregion
    if (EnemyInSight)
    {
        if (animator.GetFloat("Attack1") == 1.0f)
        {
            PlayerCharacter pc
            = gameObject.GetComponent<PlayerAgent>().playerCharacterData;
            float impact = (pc.Strength + pc.Health) / 100.0f;
            GameManager.instance.AttackEnemy(impact);
        }
    }
}
```

```

}
}
}
}
}

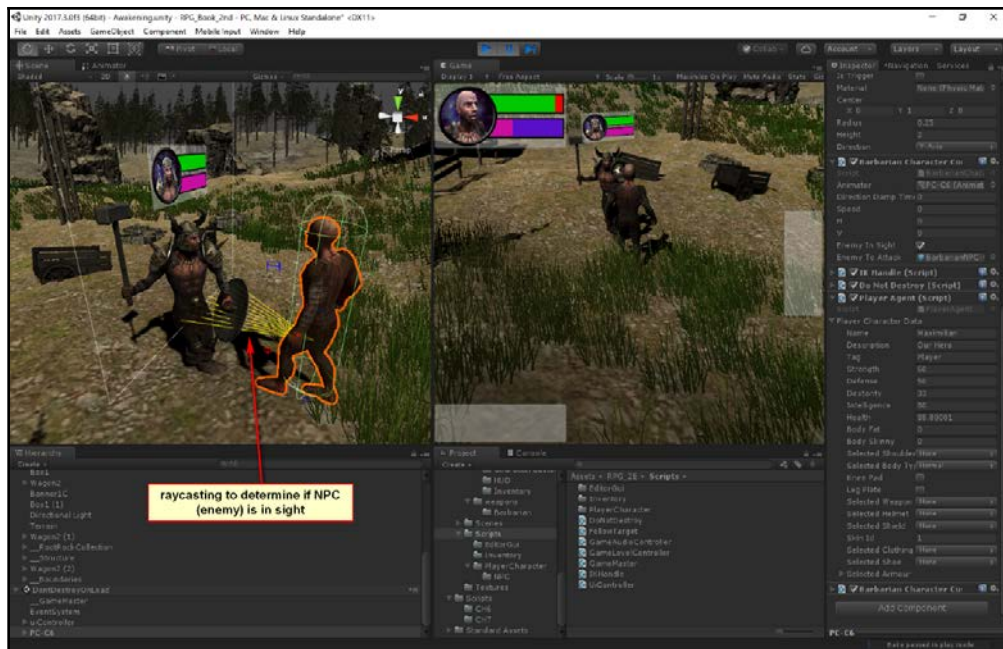
```

The way we calculate the sight and distance of the enemy NPCs is through **Raycasting**. This is done only when we are in attack mode; we check to see if the NPC is in front of us and, if so, we set the `ClosestNpcEnemy` object on `GameMaster` and set the `enemyInSight` flag, where we then perform the necessary subtraction from the health of the NPC.

Notice that I have also changed the way we compute the impact of the hit based on a simple equation, as follows:

$$impact = \frac{pc.STRENGTH + pc.HEALTH}{100}$$

Here, *pc* is the object reference to our player character. The same equation is used on the NPC objects. This is just a simple demonstration that the impact of the hit point of the player or the NPC is based on the strength and the health of the actors in the scene. Take a look at the following screenshot:



Determining if the NPC is in sight

The preceding screenshot illustrates how we detect if an NPC is in attack range or not.

In turn, you can derive the strength value from the components that the player or the NPC has activated throughout the gameplay.

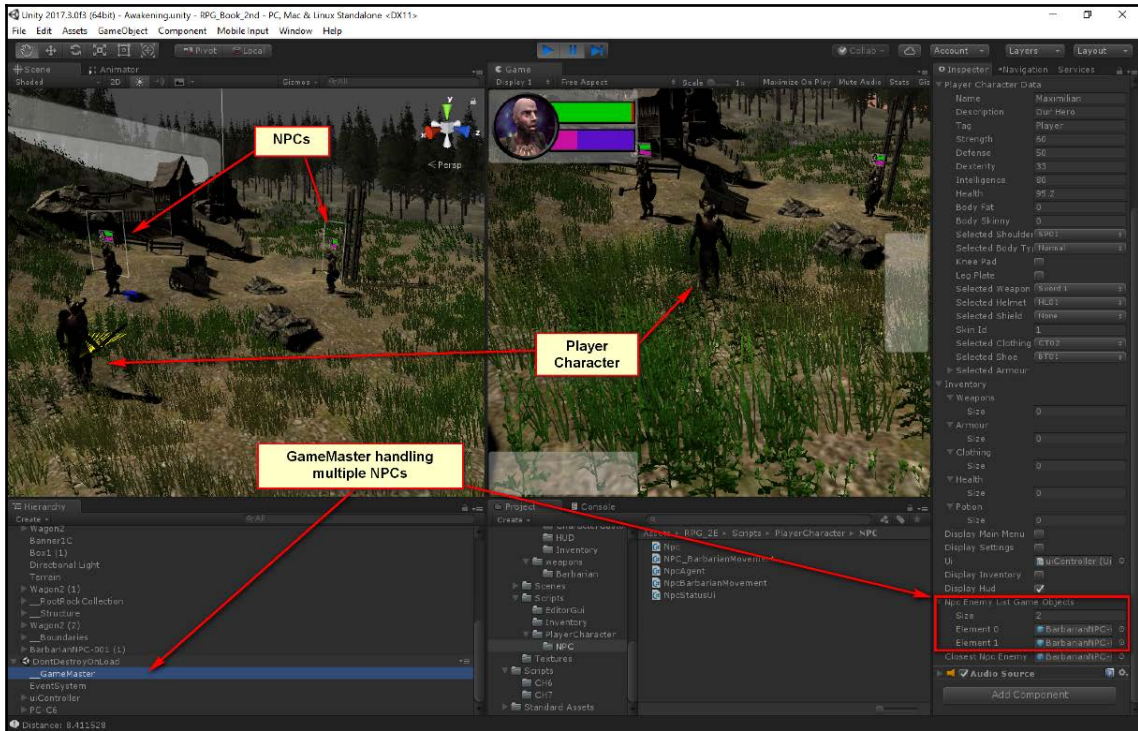
A partial listing of `BaseCharacter.cs` illustrating the `HEALTH` property is as follows:

```
[SerializeField]
private float health;
public float Health
{
    get { return health; }
    set
    {
        health = value;
        if (Tag.Equals("Player"))
        {
            if (GameMaster.instance.Ui.HudUi != null)
            {
                GameMaster.instance.Ui.HudUi.imgHealthBar.fillAmount
                = health / 100.0f;
            }
        }
        else
        {
            CharacterGameObject.GetComponent<NPCAgent>().SetHealthValue(health /
            100.0f);
        }
    }
}
```



There are more code changes and updates; please refer to the associated files provided for details.

During the implementation process, I have modified a few other code locations, which are not listed within the book due to physical limitations. Here are the scripts that have been modified: `BaseCharacter.cs`, `BarbarianCharacterController.cs`, `GameMaster.cs`, `NpcAgent.cs`, `PlayerAgent.cs` and `NpcBarbarianMovement.cs`. Take a look at the following screenshot:



GameMaster handling multiple NPCs

You are encouraged to do some research and try different types of mechanics and implementation to enhance your skills.

Summary

In this chapter, we have expanded on our idea and looked at how to integrate all of the major pieces together. The main objective of the chapter was to create a heads-up display (HUD) for our game.

We started out with a design concept that was of interest to us, and created a layout for our HUD before the actual implementation. Once we concluded how the HUD should look, we started building the framework for it. We designed the three main sections of the HUD and referred to them as the following: `PanelCharacterInfo`, `PanelActiveItems`, and `PanelSpecialItems`.

Next, we started to build the UI elements and the code necessary to make the panels work with our code. We started with `PanelCharacterInfo`, which represented the stats for our character player, that is, a reference to the player's avatar, a reference to the health, and a reference to the strength of the character. In the process, we had to create or update several of the scripts to work with the new UI.

Next, we designed and developed the `PanelActiveItems` panel. The implementation and approach to this specific panel was a little more involved. The purpose of this panel is to display all of the current active inventory items that the player has consumed. We had to make the panel scroll-able since we don't know how many items the player will be consuming at any given time. We created the necessary prefabs to be placeholders for the inventory items, as well as the scripts that make them work together.

The design for `PanelSpecialItems` was very similar to that of `PanelActiveItems`, with two main differences. First, we had to make sure that the panel was vertical instead of horizontal, so we had to make sure that the proper configuration was applied. Secondly, the main functionality was different for this panel. The items displayed were supposed to be intractable, which meant that we had to create custom event handlers, apply the necessary values to the player character, and update the whole game state.

Once we were satisfied with the design of our HUD, we started building the necessary scripts to integrate the UI elements with the `GameMaster` and other scripts. This was basically making sure that our UI was always reflecting the state of the object that was of interest to us. The health, stamina, and inventory are the main items we used to communicate the concepts.

In the last section of the chapter, we concentrated on the implementation of the player's character movement and detection of the NPCs, and how to track the hit points between the player character and the NPC, which was not done in previous chapters.

We also had to do some backtracking and make some adjustments to the animation controller we had defined for our player character, to have curves defined for our attack animation values based on the motion.

During the process, we had to solve the following challenges: How do we know if we are in close enough range that we can actually attack and hit the NPC character? How are we going to detect which NPC is closer to us? More importantly, how is the data going to be passed along from the action of attacking to the actual hit on the NPC?

We have done a lot in a short period of time and in a small number of pages. Some of the functions have been left for the reader to solve on their own. For instance, we have not discussed how to delete an inventory item and so on. I felt this to be trivial, and that the reader will be comfortable enough to implement the function on their own once they see the bigger scope and how to connect everything together.

With that said, let's move on to the next and final chapter, covering multiplayer!

8

Multiplayer Setup

The desire of every indie game developer is to make a multiplayer game. The reality is, creating multiplayer games is difficult. There are a lot of scenarios that you need to take into consideration as a game designer/developer. Besides the technical complexities that are involved in the nature of creating online multiplayer games, there are also game-play elements that you will need to consider.

The purpose of this chapter is to give you a good overview of the out-of-box networking functionality using the new Unity networking paradigm. This is a complex topic, and as such we cannot cover everything in this chapter. A whole new book will be required to really dive into the details.

Having said that, I have prepared this chapter to include one simple project, which will be used to illustrate the fundamentals of networking. I will then show you how to network-enable our own game objects.

In this chapter, we will cover the following:

- Heads-up display
- Completing a HUD design
- Integrating the code

Here we go!

Challenges of a multiplayer game

A general rule of thumb is, if you don't need to enable your game to be multiplayer, don't! It just adds a whole lot of complexity and extra requirements and specifications that you will need to start worrying about. But if you must, then you must!

You probably know by now that creating even the simplest multiplayer game has its own challenges you will need to address as the game designer. There are different types of online multiplayer game design, as follows:

- Real-time multiplayer games
- Turn-based multiplayer games
- Asynchronous multiplayer games
- Local multiplayer games

The most challenging out of all the different types of multiplayer games is real-time multiplayer gaming. This is because all players have to be synchronized in a proper and effective way with the latest game state at any given time.

That is, if we have Player A perform a specific action, Player B will see the action at the same time on his or her screen. Now, consider we have another player join, say Player C; Players A and B will need to synchronize with Player C, and in turn Player C will need to synchronize its own environment with Player A's and Player B's state.

Not only does the actual position/rotation of the players have to be synchronized, but also all of the player data. Now, imagine what will happen when you multiply this by 100, 1,000, or 1,000,000 connected players.

For a real-world multiplayer game, what we will cover here is not enough, and what Unity provides out of the box is not enough either. Chances are you will need to write your own server-side code to handle the player data.

Now you can see the challenge involved in designing and developing multiplayer games, we can start by building our first multiplayer game.

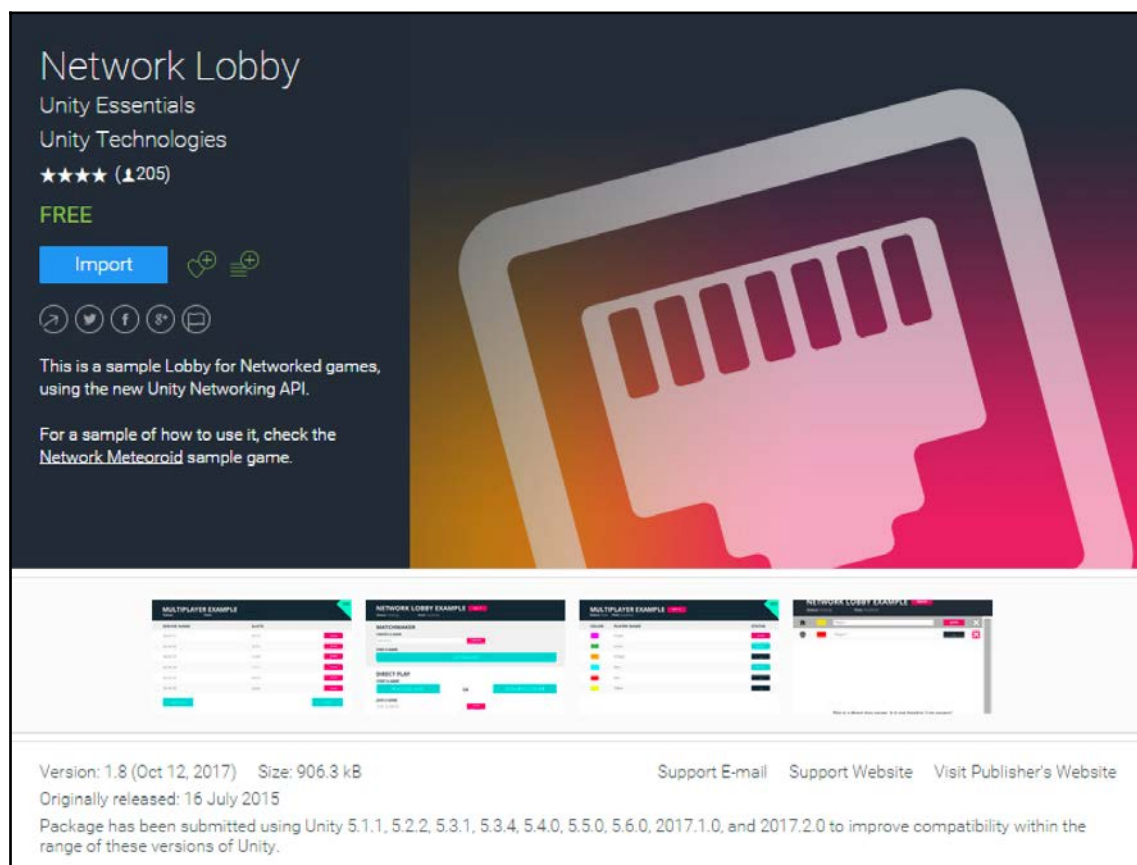
Initial multiplayer game

The best way to learn about multiplayer games is by looking at a simple example. The following project is based on the Unity networking tutorial but has been extended to have some other features implemented that will be helpful in the implementation of networking in our RPG.

Fundamental networking components

Since the first edition of the book, a lot of technical and architectural changes have occurred within the Unity engine. Networking is one of the areas that has improved drastically since the publication of the first edition.

Let's take a look and see how we can quickly get started. The first thing you will need to do is download the **Network Lobby** asset from the Asset Store, as shown in the following screenshot:



This is a great starting point, as we have a generic network lobby to start off with, so we don't have to waste time creating everything from scratch!

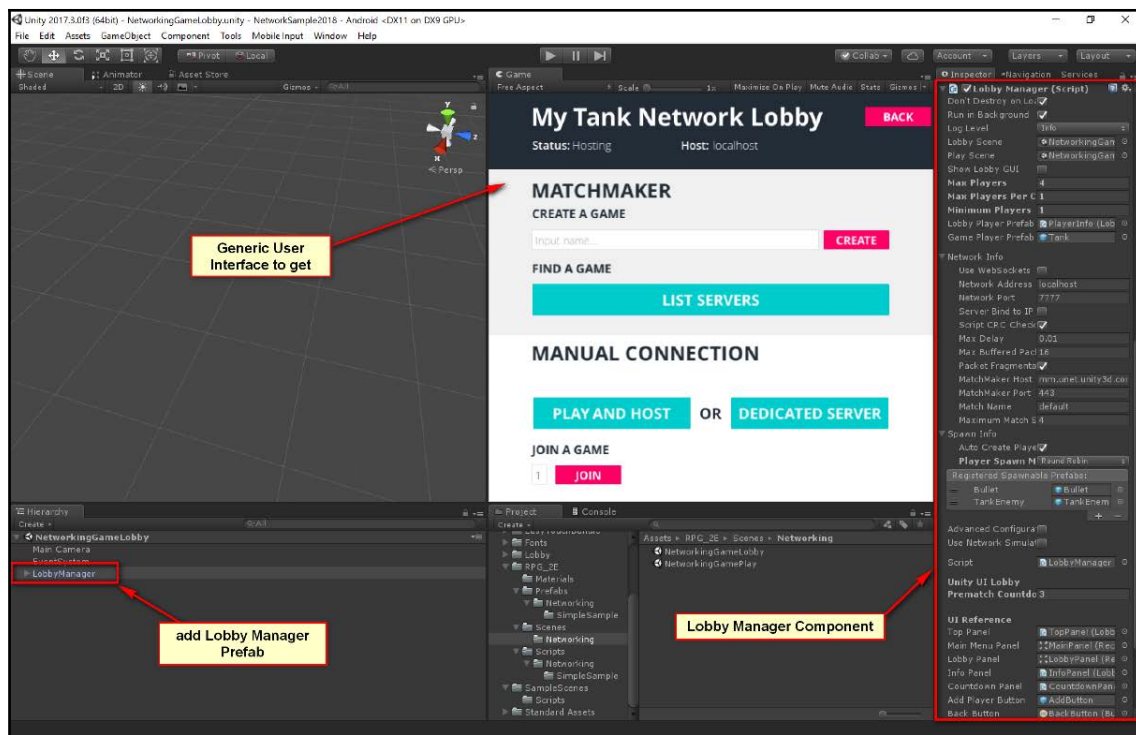
We need to get familiar with some networking components that will be used for the creation of our network-enabled games. These components are as follows:

- **Network Manager:** `NetworkManager` is a high-level class that allows you to control the state of a networked game. It provides an interface in the editor to control the configuration of the network, the prefabs used for spawning, and the scenes to use for different network game states.
- **Network Lobby Manager:** `NetworkLobbyManager` is a specialized type of `NetworkManager` that provides a multiplayer lobby before entering the main play scene of the game. Ideal for matchmaking.
- **Network Manager HUD:** This provides a default user interface for controlling the network state of the game. It also shows information about the current state of `NetworkManager` in the editor.
- **Network Identity:** The `NetworkIdentity` component is at the heart of the new networking system. This component controls an object's network identity, and it makes the networking system aware of it.
- **Network Transform:** The `NetworkTransform` component synchronizes the movement of game objects across the network. This component takes authority into account, so `LocalPlayer` objects synchronize their position from the client to the server, then out to other clients. Other objects (with server authority) synchronize their position from the server to clients.

My tank networking project

It is assumed that you have downloaded and imported the Network Lobby asset. We will be using two scenes to illustrate the concept. The first scene will be the lobby scene, and the second scene will be where the game will take place.

Go ahead and create a scene, and name it `NetworkingGameLobby`. This is where the game will start. Take a look at the following screenshot:

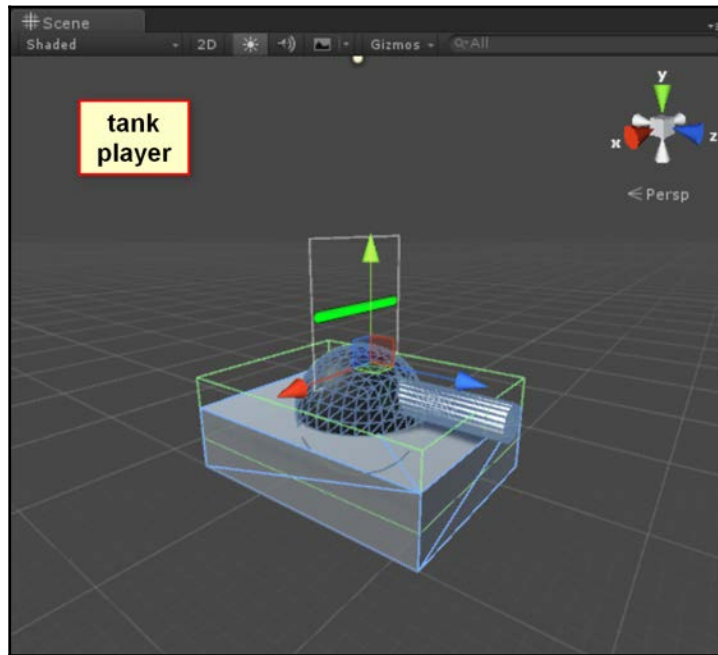


My tank network lobby

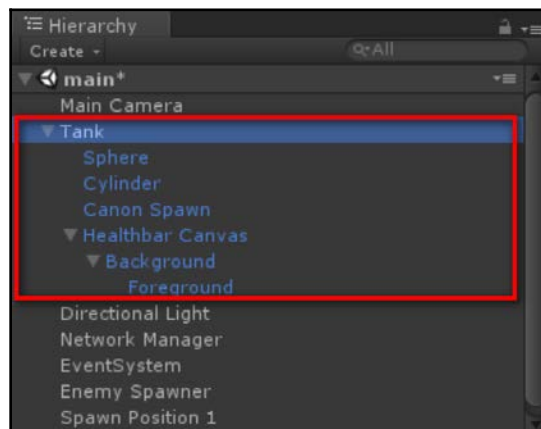
Add the `LobbyManager` prefab to the scene. Select the `LobbyManager` `GameObject` in the scene, and note the `LobbyManager` component in the **Inspector** window. There are several things to notice here. First, you will need to assign the **Lobby Scene** and the **Play Scene** properties. We need to assign our `NetworkingGameLobby` scene to the **Lobby Scene** property, and the `NetworkingGamePlay` scene to the **Play Scene** property. Next, you will need to assign the `Game Player Prefab` property with the `Player` prefab. Don't worry, we will create the `Player` prefab next. Finally, we need to configure the `Spawn Info` section, and register the prefabs (`GameObjects`) that will be spawned during game time; in our case, these will be the `Bullet` and the `Enemy Tanks`.

Adding a player character

We will now add a simple player character. You can really use any primitive GameObject to represent your PC; I will create my player to take the shape of a simple tank. Take a look at the following screenshot:



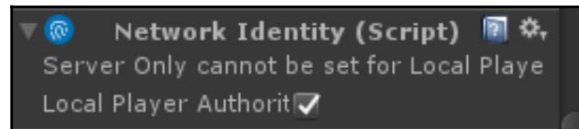
The following screenshot will illustrate the hierarchy of the Tank GameObject:



I will not cover how to create the `GameObject`, as you should be able to do that very easily by now. What I will cover, is how to enable the new network-enabled Tank `GameObject`.

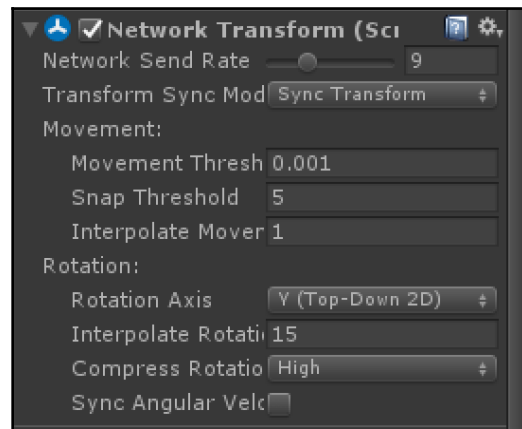
We will attach two network components to the Tank `GameObject`. The first one will be `NetworkIdentity`, which can be added by selecting the Tank `GameObject`, and from within the **Inspector** window, select **Add Component** | **Network** | **Network Identity**.

When you are done adding the component, make sure to check the **Local Player Authority** property checkbox, as shown in the following screenshot:



Local Player Authority allows the object to be controlled by the client that owns it.

Next, we need to add the `NetworkTransform` component to the Tank `GameObject`. Again, selecting the Tank `GameObject`, from the **Inspector** window, click **Add Component** | **Network** | **NetworkTransform** to add the component:



We will keep the default values for the `NetworkTransform` component. With the exception of the rotation properties, since we are only rotating on the Y-axis, we don't have to send all XYZ information, so change that to **Y (Top-Down 2D)** and increase the interpolate Rotation field to 15, and change the Compress Ratio to High. You can read more about the different properties on your own using the online documentation. The main attribute you might want to adjust is the **Network Send Rate**.

Next, we want to create a script that will allow us to control the movement of the Tank. Go ahead and create a new C# script and name it `MyPlayerController.cs`.

Here is a listing of the script:

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;
public class MyPlayerController : NetworkBehaviour
{
    public Transform mainCamera;
    public float cameraDistance = 16f;
    public float cameraHeight = 16f;
    public Vector3 cameraOffset;
    [SyncVar]
    public Color myColor;
    public GameObject bulletPrefab;
    public Transform bulletSpawn;
    private void Start()
    {
        GetComponent<MeshRenderer>().material.color = myColor;
        cameraOffset = new Vector3(0f, cameraHeight, -cameraDistance);
        mainCamera = Camera.main.transform;
        MoveCamera();
    }
    void Update()
    {
        // only execute the following code if local player ...
        if (!isLocalPlayer)
            return;
        #if UNITY_EDITOR
        var x = Input.GetAxis("Horizontal") * Time.deltaTime * 150.0f;
        var z = Input.GetAxis("Vertical") * Time.deltaTime * 3.0f;
        #else
        var x = ETCInput.GetAxis("Horizontal") * Time.deltaTime * 150.0f;
        var z = ETCInput.GetAxis("Vertical") * Time.deltaTime * 3.0f;
        #endif
        transform.Rotate(0, x, 0);
        transform.Translate(0, 0, z);
        #if UNITY_EDITOR
        if (Input.GetKeyDown(KeyCode.Space))
        {
            CmdFire();
        }
        #else
        if (ETCInput.GetButtonDown("ButtonFire"))
        {
            CmdFire();
        }
    }
}
```

```
}
#endif
MoveCamera();
}
[Command]
void CmdFire()
{
    // Create the Bullet from the Bullet Prefab
    var bullet = Instantiate(
        bulletPrefab,
        bulletSpawn.position,
        bulletSpawn.rotation) as GameObject;
    // Add velocity to the bullet
    bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward *6;
    bullet.GetComponent<Bullet>().myColor = myColor;
    // Spawn the bullet on the Clients
    NetworkServer.Spawn(bullet);
    // Destroy the bullet after 2 seconds
    Destroy(bullet, 2.0f);
}
void MoveCamera()
{
    mainCamera.position = transform.position;
    mainCamera.rotation = transform.rotation;
    mainCamera.Translate(cameraOffset);
    mainCamera.LookAt(transform);
}
}
```

The code is straightforward, but there are some important concepts that we need to discuss. First and foremost, you will notice that we are inheriting from `NetworkBehaviour` instead of `MonoBehaviour`.

`NetworkBehaviour` is used to work with objects with the `NetworkIdentity` component. This allows you to perform network-related functions such as `Commands`, `ClientRPCs`, `SyncEvents`, and `SyncVars`.

Variable synchronization

Synchronizing variables is one of the important aspects of a multiplayer game. If you recall, one of the challenges of multiplayer games is the ability to make sure all of the key data for the game is synchronized across the server and the clients. This is accomplished by the `SyncVar` attribute. You will see how this is applied in the next script, which we will create for the health of the unity.

Network callbacks

Network callbacks are functions that are invoked on the `NetworkBehaviour` script for various network events. They are listed as follows:

- `OnStartServer()` is called when an object is spawned on the server, or when the server is started for objects in the scene
- `OnStartClient()` is called when the object is spawned on the client, or when the client connects to a server for objects in the scene
- `OnSerialize()` is called to gather state to send from the server to the clients
- `OnDeserialize()` is called to apply state to objects on clients
- `OnNetworkDestroy()` is called on clients when the server tells the object to be destroyed
- `OnStartLocalPlayer()` is called on clients for player objects for the local client only
- `OnRebuildObservers()` is called on the server when the set of observers for an object is rebuilt
- `OnSetLocalVisibility()` is called on a host when the visibility of an object changes for the local client
- `OnCheckObserver()` is called on the server to check the visibility state for a new client

In the `PlayerController.cs` script, you will notice that we are using `OnStartClient()` to highlight the local player by changing its material color to blue.

Sending commands

Commands are how clients request a function to be performed on the server. In a server authoritative system, clients can only do things through commands. Commands are run on the player object on the server that corresponds to the client that sent the command. This routing happens automatically, so it is impossible for a client to send a command for a different player.

A command must begin with the prefix "Cmd" and have the `[Command]` custom attribute on it.

In our `PlayerController.cs` script, when the player fires, it sends a `Command` to the server using the `CmdFire()` function.

Client RPC calls

Client RPC calls are a way for server objects to cause things to happen on client objects. This is the reverse direction to how commands send messages, but the concepts are the same. Client RPC calls, however, are not only invoked on player objects; they can also be invoked on any `NetworkIdentity` object. They must begin with the prefix "Rpc" and have the `[ClientRPC]` custom attribute.

You will see an example of this on the `Health.cs` script, which we will be creating next.

We will also need a way to keep track of our player character's health. This will be done using a new script called `Health.cs`.

A listing of the script is as follows:

```
using UnityEngine;
using UnityEngine.Networking;
public class Health : NetworkBehaviour
{
    public const int maxHealth = 100;
    [SyncVar(hook = "OnChangeHealth")]
    public int currentHealth = maxHealth;
    public RectTransform healthBar;
    public bool destroyOnDeath;
    public GameObject[] listOfPlayers;
    private void Start()
    {
        healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);
    }
    public void TakeDamage(int amount)
    {
        currentHealth -= amount;
        if (currentHealth <= 0)
        {
            if (destroyOnDeath)
            {
                RpcDied();
                listOfPlayers = GameObject.FindGameObjectsWithTag("Player");
                if (listOfPlayers.Length < 1)
                {
                    Invoke("BackToLobby", 3.0f);
                }
            }
            else
            {
                currentHealth = maxHealth;
            }
        }
    }
}
```

```
// called on the Server, will be invoked on the Clients
RpcRespawn();
}
}
}
void OnChangeHealth(int health)
{
healthBar.sizeDelta = new Vector2(health, healthBar.sizeDelta.y);
}
[ClientRpc]
void RpcRespawn()
{
if (isLocalPlayer)
{
// move back to zero location
transform.position = Vector3.zero;
}
}
[ClientRpc]
void RpcDied()
{
gameObject.tag = "Untagged";
GetComponent<Renderer>().material.color = Color.black;
if (GetComponent<MyPlayerController>() != null)
{
GetComponent<MyPlayerController>().enabled = false;
}
if (GetComponent<EnemyController>() != null)
{
GetComponent<EnemyController>().enabled = false;
}
}
void BackToLobby()
{
FindObjectOfType<NetworkLobbyManager>().ServerReturnToLobby();
}
}
```

Notice that, in this script, we are also inheriting from `NetworkBehaviour`. The two main items I want to bring to your attention are the `SyncVar`, the `ClientRpc`, and the `Start()` functions.

We want to synchronize the player's health across the network. To do this, we use the `SyncVarNetworkBehaviour`. `SyncVar` can be any basic type, not classes, lists, or other collections.

When the value of *SyncVar* is changed on the server, it will be sent to all of the ready clients in the game. When objects are spawned, they are created on the client with the latest state of all *SyncVars* from the server.

The `OnStartClient()` function makes sure that each object with the `Health.cs` script attached to it will have the most up-to-date value to display on the health bar UI.

I want to take a moment and make sure I give you a crucial pointer here. Assume we are running a networked game session, and we have the `Host`, `Player A`, and `Player B` connected and going about their business. During the gameplay, `Player A` and `Player B` have their health value changed. Now, we have a third player connect to the game, `Player C`. If `Start()` is not implemented, the client for `Player C` will have the correct data synchronized for all of the `GameObjects` with the `Health.cs` script; however, the data will not reflect correctly on the UI because we need to have a trigger for that to happen. This can be handled in the `Start()` function, as shown in the code.

The next function is the `RpcRespawn()` function. In the `TakeDamage()` function, we check the health of the current `GameObject`. If the health drops below zero, we check to see if the `destroyOnDeath` Boolean variable is set. If it is not set, we go ahead and reset the `currentHealth` value to `maxHealth` value, and we use the `RpcRespawn()` method to re-spawn the player at the origin. Remember this function is executed on all clients!

Within the function, we check to see if the caller is the local player by checking the `isLocalPlayer` variable. Yes, creating a multiplayer game does get confusing! This will become more apparent as you start experimenting with it more.

Creating the cannonball for the tank

Let's create a prefab that will represent our cannonballs! Very simply, create a sphere and make it the same size as the nozzle of your tank gun.

We will need to attach the following components to the Canon Ball `GameObject`: `NetworkIdentity`, `NetworkTransform`, `Rigidbody`, and `Bullet.cs` scripts.

Make sure that you set the `Use Gravity` property to `False` on the `Rigidbody` component. Also, make sure that both **Server Only** and **Local Player Authority** properties are *False* on the `NetworkIdentity` component. On the `NetworkTransform` component, change `Network Send Rate` to 0. Once we generate the object on the server, the physics will take care of the motion on each client.

Create a new C# script called `Bullet.cs`.

The listing for the script is as follows:

```
using UnityEngine;
using UnityEngine.Networking;
public class Bullet : NetworkBehaviour
{
    [SyncVar]
    public Color myColor;
    private void Start()
    {
        GetComponent<MeshRenderer>().material.color = myColor;
    }
    void OnCollisionEnter(Collision collision)
    {
        var hit = collision.gameObject;
        var health = hit.GetComponent<Health>();
        if (health != null)
        {
            health.TakeDamage(10);
        }
        Destroy(gameObject);
    }
}
```

All we are doing here is detecting a collision. If there is a collision, we get the `Health` component. If the `Health` component is not null, we call the `TakeDamage()` function and pass it a value.

If you recall from the `Health.cs` script, the `TakeDamage()` function reduces `currentHealth` of the player, which in return is a `SyncVar` that gets updated on all active clients.

One item we did not discuss is the idea of a hook. A `SyncVar` can have a hook. Think of a hook as an event handler. The hook attribute can be used to specify a function to be called when the `SyncVar` changes value on the client:

```
[SyncVar(hook = "OnChangeHealth")]
public int currentHealth = maxHealth;
```

The `OnChangeHealth()` function is responsible for updating the UI canvas for displaying our health value:

```
void OnChangeHealth(int health)
{
    healthBar.sizeDelta = new Vector2(health, healthBar.sizeDelta.y);
}
```

Go ahead and also make a prefab of the cannonball and delete the instance from the scene.

Make sure you have assigned the proper prefab associations that are required on each script. For instance, the `Tank GameObject's PlayerController.cs` script needs a reference to the cannonball prefab, and also the canon spawn location. The `Health.cs` script needs a reference to the `HealthBar` foreground image and so on.

Creating the tank prefab and configuring the network lobby manager

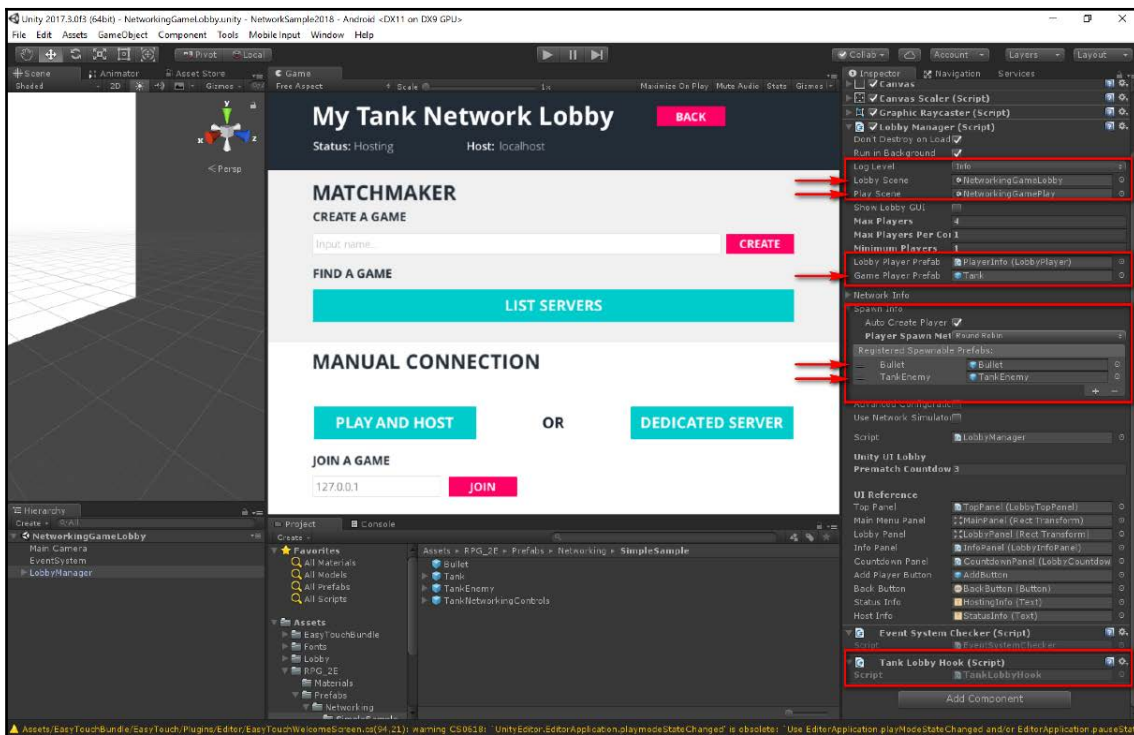
Now that we have created our `Tank GameObject` and attached all of the necessary components and scripts to it, we need to make a prefab of it. This is because we will let `NetworkLobbyManger` spawn our player character, and in order for it to be able to do so, it needs to refer to a prefab that is a representation of your player character.

The `NetworkLobbyManager` has a `Spawn Info` section that you can assign the `Player Prefab`, determine if the `NetworkLobbyManager` can `Auto Create Player` and the `Player Spawn Method`.

There is also a section for `Registered Spawnable Prefabs`. We need to register all of the `GameObjects` that will be spawned by the `NetworkServer`. For instance, the `Canon Ball` prefab will need to be registered here so that we can spawn it across the network on different clients.

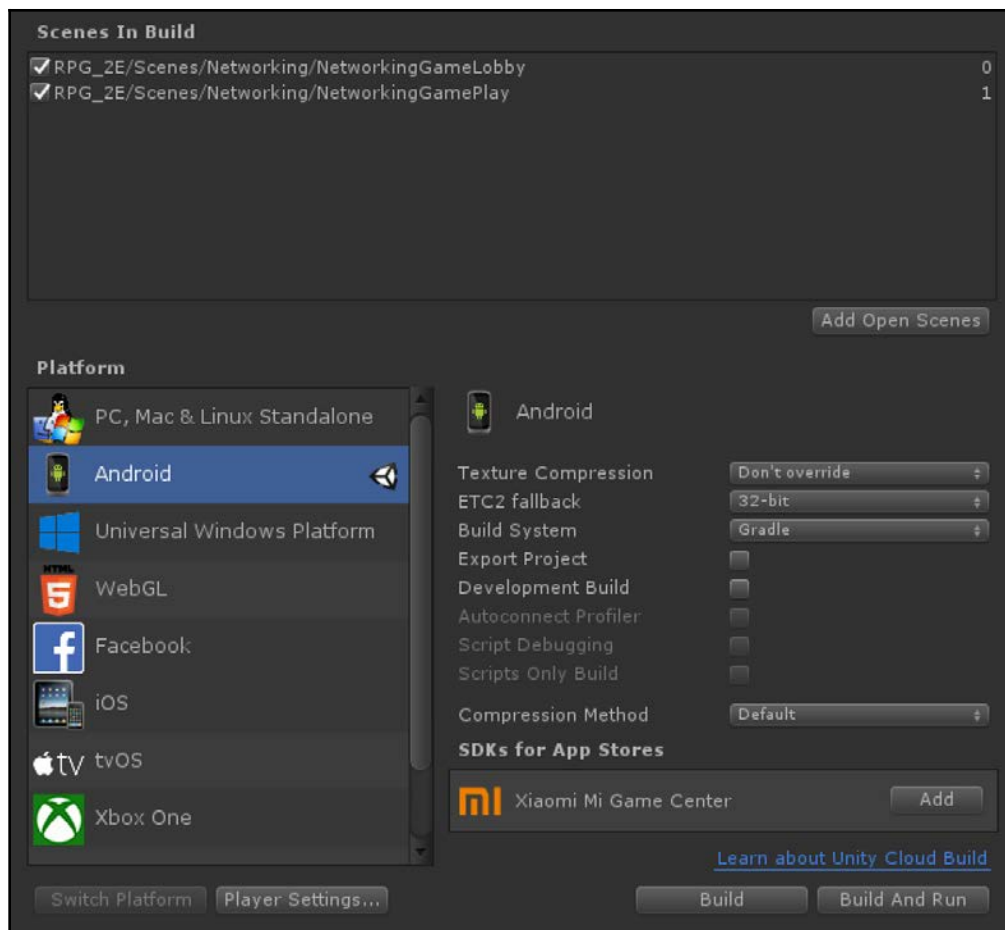
Select the `Network Lobby Manager GameObject` in the scene, and in the **Inspector** window assign the appropriate prefabs as needed.

Here is a screenshot of how the NetworkLobbyManager should look at this point:

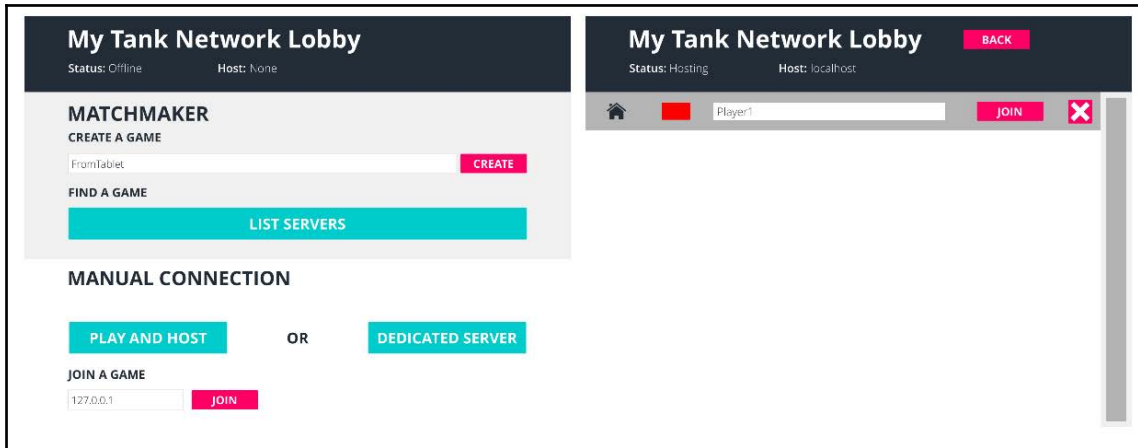


Lobby manager

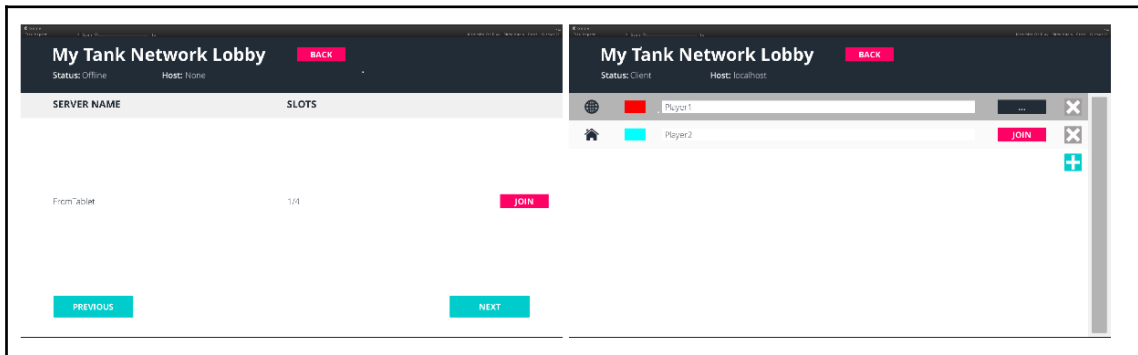
At this point, you are ready to test out what we have built so far. Go ahead and create a stand-alone version of your game using the **Build Settings** window. Once you have your build ready, launch two instances of the application. We will use one instance to host the game, and the other to connect as a client, as shown in the following screenshot:



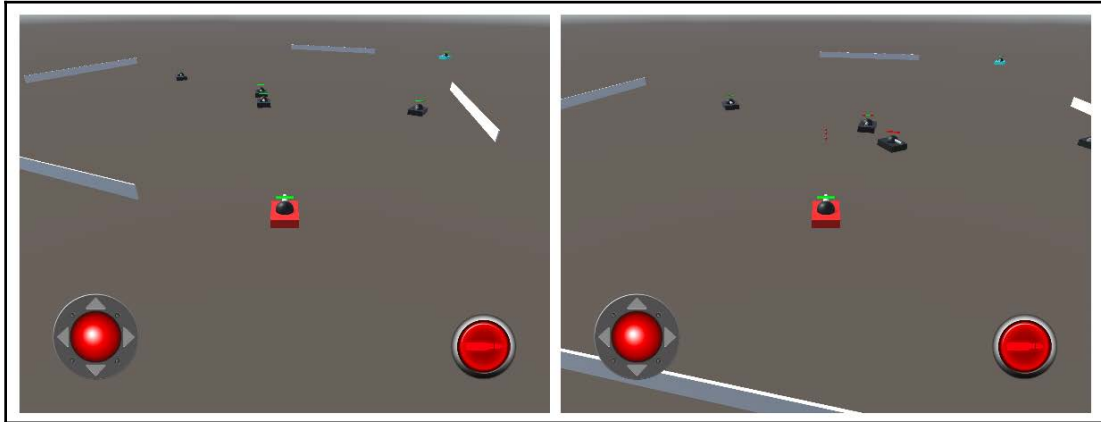
The following screenshot illustrates how your game instance will look when you run it:



The following screenshots illustrate how your screen will look during runtime. The game is created on an Android tablet:



The second player selects **LIST SERVERS** and gets available matches, and clicks **JOIN** to enter the room, as shown in the following screenshot:



Once everyone is ready, the game starts. Notice that each player has a unique tank color assign based on their color selection in the lobby.

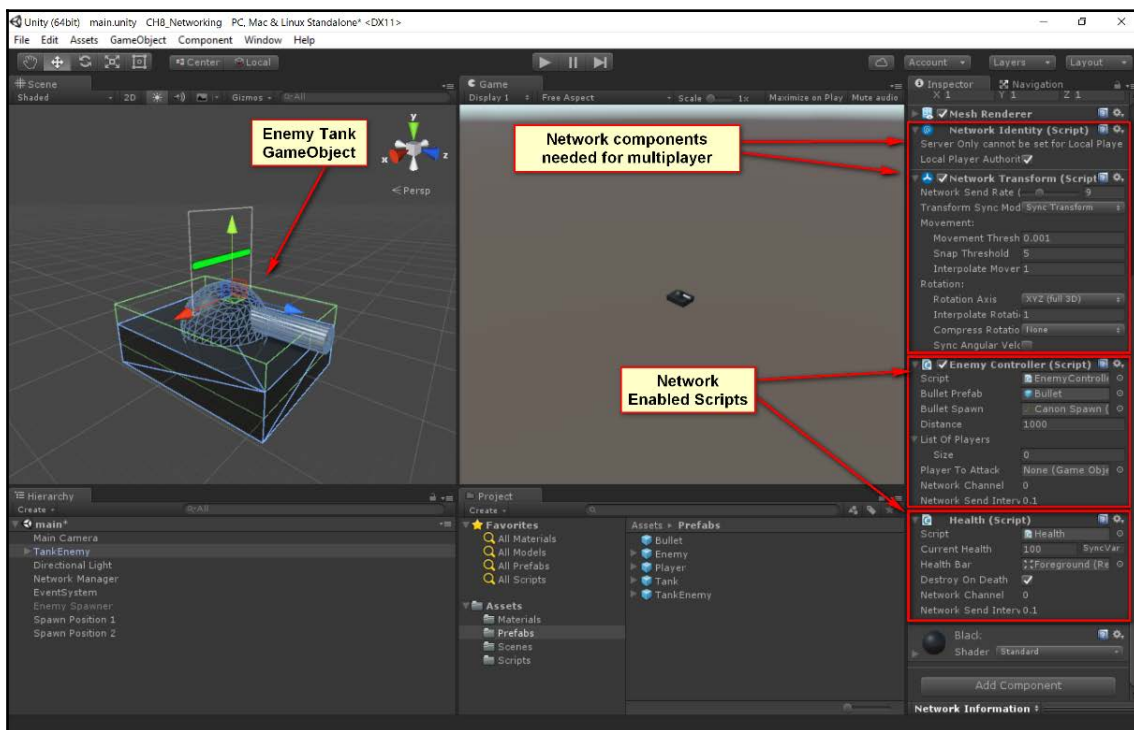
In the preceding screenshot, note that each client has highlighted the player character it controls, that is, the tank it controls. It will be difficult to capture the fire command, but you can go ahead and use the Space Bar to fire the canon, and it will be triggered accordingly on all active clients.

You will also notice that the health of each tank will be reflected accurately if they do get a hit. Now we are ready to create an enemy to illustrate the non-player character in the game.

Adding the enemy tank

Now it is time to add some of the non-player characters to our multiplayer demo. Adding the Enemy Tank will be simple as we will use our Tank prefab as a base. Go ahead and drop the Tank prefab into the scene, and change the name to `TankEnemy`.

Remove the `MyPlayerCharacter.cs` script from the `GameObject`. We will create a separate script as the controller for the enemy tank. I have also gone ahead and applied different material to the enemy tank so that we can visually distinguish which tanks will be controlled by players and which ones will be non-player. Take a look at the following screenshot:



Enemy tank setup

The preceding screenshot illustrates how your `Tank` and `TankEnemy` prefabs should look. The main difference between the two is the controller script. `Tank` has the `MyPlayerController.cs` script, and `TankEnemy` has the `EnemyController.cs` script.

A listing of the `EnemyController.cs` script is as follows:

```
using UnityEngine;
using UnityEngine.Networking;
public class EnemyController : NetworkBehaviour
{
    public GameObject bulletPrefab;
    public Transform bulletSpawn;
```

```
public float distance = 1000;
public GameObject[] listOfPlayers;
[SyncVar(hook = "OnChangePlayerToAttack")]
public GameObject playerToAttack;
float coolOffTime = 0.0f;
void Update()
{
    // only execute the following code if local player ...
    if (!isServer)
        return;
    listOfPlayers = GameObject.FindGameObjectsWithTag("Player");
    if (listOfPlayers.Length > 0)
    {
        float distance = 100f;
        foreach (var player in listOfPlayers)
        {
            float d = Vector3.Distance(transform.position, player.transform.position);
            if (d < distance)
            {
                distance = d;
                playerToAttack = player;
            }
        }
        if (playerToAttack != null)
        {
            Vector3 direction = playerToAttack.transform.position - transform.position;
            transform.rotation =
                Quaternion.Slerp(transform.rotation,
                Quaternion.LookRotation(direction), 0.1f);
            float d = Vector3.Distance(transform.position,
            playerToAttack.transform.position);
            if (d < 15.0f)
            {
                if (coolOffTime < Time.time)
                {
                    CmdFire();
                    coolOffTime = Time.time + 1.0f;
                }
            }
        }
    }
    void OnChangePlayerToAttack(GameObject player)
    {
        playerToAttack = player;
    }
    [Command]
    void CmdFire()
```



```
{  
    // Create the Bullet from the Bullet Prefab  
    var bullet = Instantiate(  
        bulletPrefab,  
        bulletSpawn.position,  
        bulletSpawn.rotation) as GameObject;  
    // Add velocity to the bullet  
    bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward * 6;  
    // Spawn the bullet on the Clients  
    NetworkServer.Spawn(bullet);  
    // Destroy the bullet after 2 seconds  
    Destroy(bullet, 2.0f);  
}  
}
```

The script continuously searches for all players that are active in the scene and makes a list of them. Then it finds the closest one to itself. Once it determines which player is closest, it rotates to face the player.

After that, it calculates the distance between itself and the selected player; if the distance is shorter than the acceptable threshold, then it starts firing at the player. Each time the enemy tank fires, it actually calls a [Command] named `CmdFire()`.

This function is run on the server; it instantiates a cannonball prefab and spawns it on the network.

The `EnemyController.cs` script also has a `SyncVar` for the `playertoAttack` variable, with a hook attached as an `OnChangePlayerToAttack()` function. This, in turn, makes sure that all clients get updated with the latest data on each Enemy Tank GameObject.

The `Health.cs` script works the same as it does on the Tank GameObject.

There is one more item we need to cover: the spawning of the Enemy Tanks by the server. We can do this easily by creating another empty GameObject and naming it `Enemy Spawner`. We need to attach a `NetworkIdentity` component and make sure we set the `Server Only` property to `True`. This will make sure that only the server can instantiate the enemy objects.

The next step is to create the `EnemySpawner.cs` scripts. The listing is as follows:

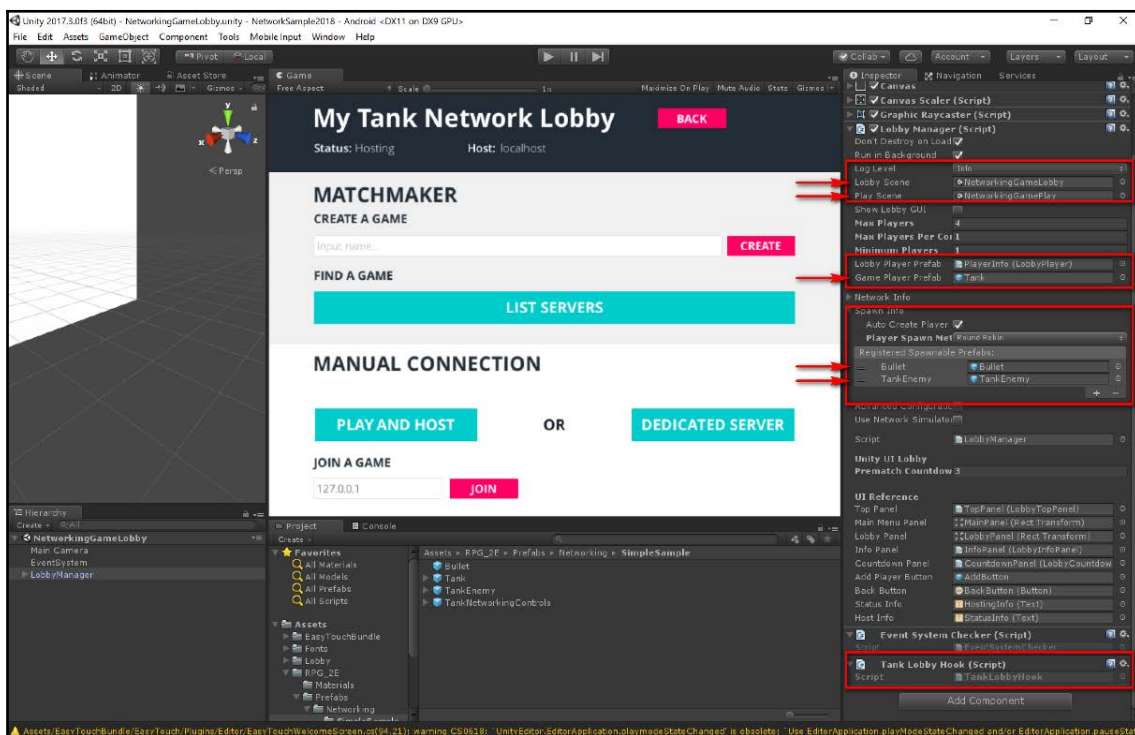
```
using UnityEngine;
using UnityEngine.Networking;
public class EnemySpawner : NetworkBehaviour {
    public GameObject enemyPrefab;
    public int numberOfEnemies;
    public override void OnStartServer()
    {
        for (int i = 0; i < numberOfEnemies; i++)
        {
            var spawnPosition = new Vector3(
                Random.Range(-20.0f, 20.0f),
                0.0f,
                Random.Range(-20.0f, 20.0f));
            var spawnRotation = Quaternion.Euler(
                0.0f,
                Random.Range(0, 180),
                0.0f);
            var enemy = (GameObject)Instantiate(enemyPrefab, spawnPosition,
                spawnRotation);
            NetworkServer.Spawn(enemy);
        }
    }
}
```

This code technically takes the prefab provided as the enemy tank and randomly spawns each enemy tank within a range over the network.

Make sure all of your prefabs have been assigned in the **Inspector** window for both the `Enemy Spawner` `GameObject` and the `TankEnemy` `GameObject`. Create a prefab of your `TankEnemy` if you have not done so already, and delete it from the scene. Do not delete the `Enemy Spawner`.

We need to register the `TankEnemy` prefab with the `NetworkLobbyManager`. Go ahead and select the `Network Manager` `GameObject`, and from the **Inspector** window, add a new Prefab to the `Registered Spawnable Prefabs` option.

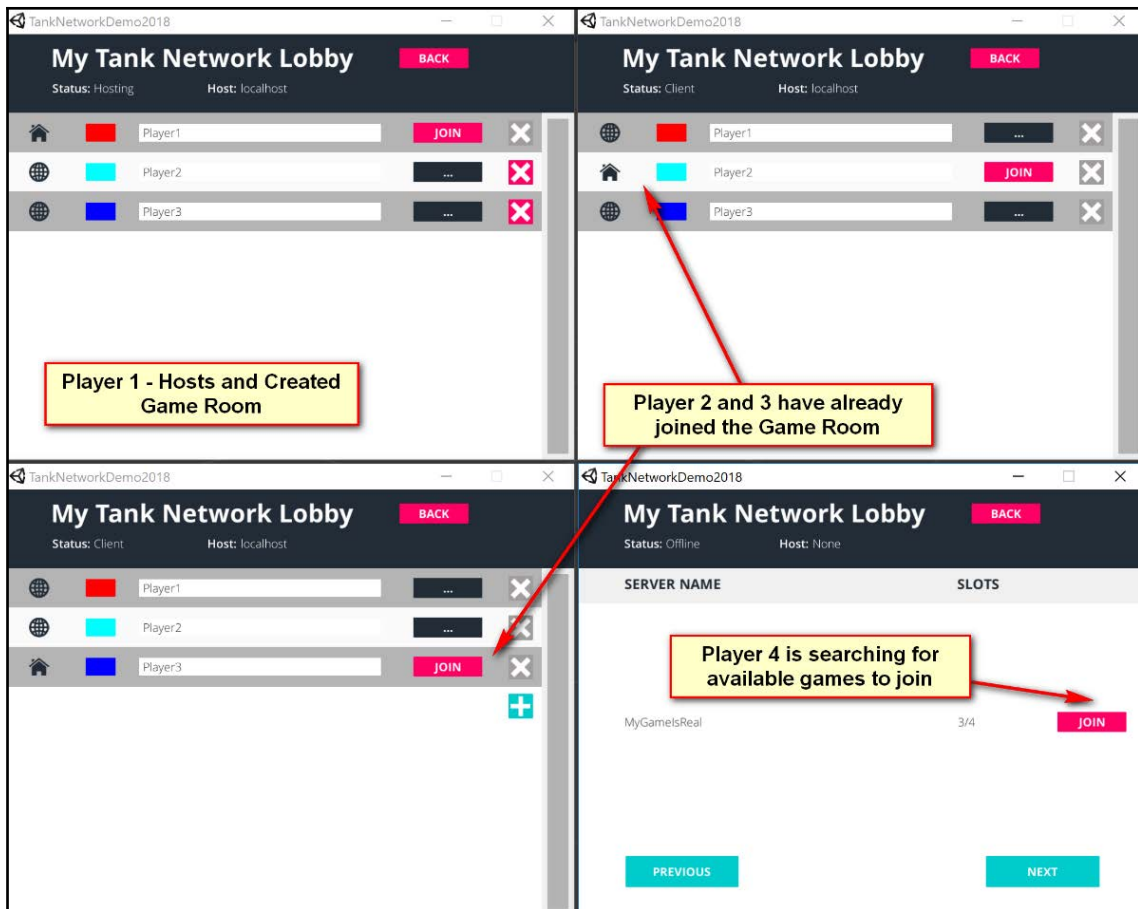
Your Network Manager should now look as follows:



Lobby Network Manager Setup

Building and testing

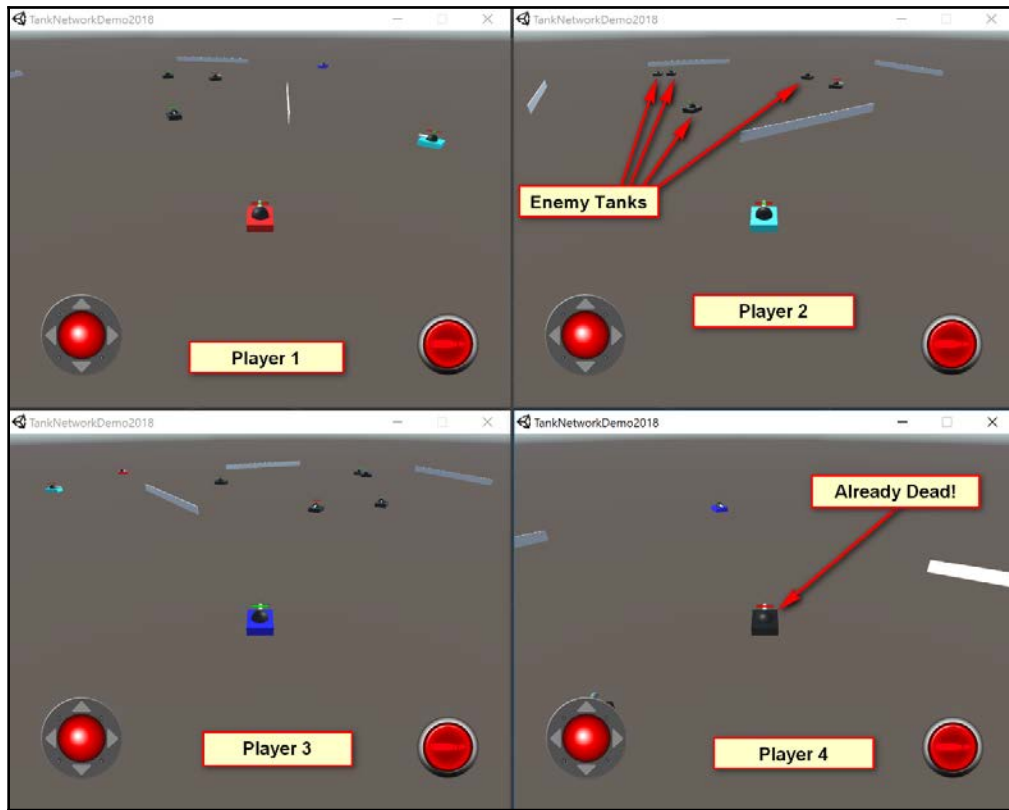
We are ready to do our final test. Go ahead and build the stand-alone version of the project, and launch a new instance of the game. **Player 1** will create the room, and the rest of the players will join the game, as shown in the following screenshot:



Joining a game demo

You will also notice that right after initialization, all of the enemy tanks will rotate toward the player character tank and if within range, they will start firing at it.

The following screenshot illustrates the runtime of the game:



Notice that while I was trying to capture the screen, the enemy tanks were merciless and fired at **Player 4** continuously. You can see that my health bar has reduced drastically. You can also see that one of the enemy tanks has also received some damage.

I assure you that I had nothing to do with the damage taken by the enemy tank; it was actually caused by friendly fire. Yes, at the moment, the enemy tanks are not smart enough to hold fire if another team member is in the line of fire!

I will let you handle the implementation of that on your own. Shouldn't be too complex.



Use Raycasting to make sure there is no object between the enemy tank and the player prior to firing.

Congratulations! You have just created your first multiplayer game! As mentioned previously, creating, maintaining, and hosting a multiplayer game is no small task, and covering every single aspect on how to do it is simply impossible in a few pages.

The idea here is to give you the foundation and the fundamentals that you can take and expand upon. I would encourage you to take some time to study what we have just covered and do some more reading on the material, even though not much exists. The truth is that you will need to do a lot of experimentation and trial and error on your own.

Now that we know the basics, let's go ahead and apply what we have learned to our RPG assets.

Network-enabling RPG characters

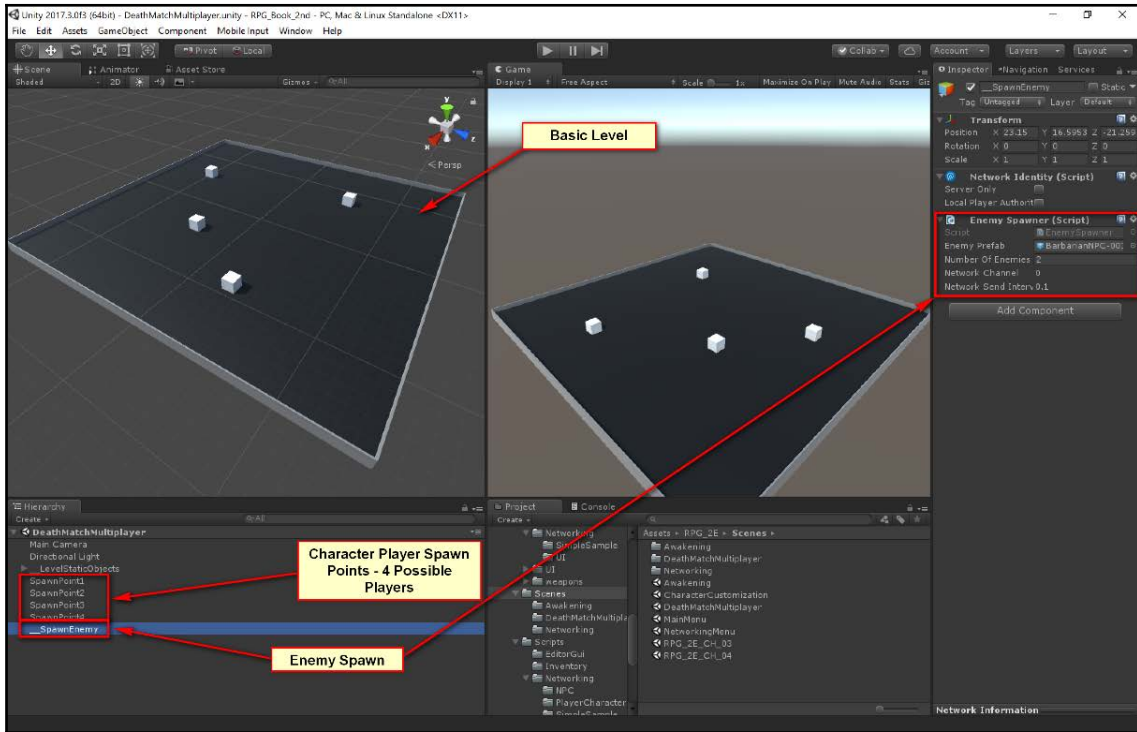
In order to make life easier, I have decided to create a new scene that will be used to test and implement our network-enabled characters. This example will show you how to network-enable the player character, and also how to synchronize the player character data such as inventory items across the network, as well as provide you with the ability to network-enable the non-player character and make it synchronize its data across the clients.

Creating a scene for our RPG

Just like the tank demo in the previous section, we will use two new scenes to illustrate the RPG networking concept here. We will create two scenes, one called `NetworkingMenu` and the other `DeathMatchMultiplayer`.

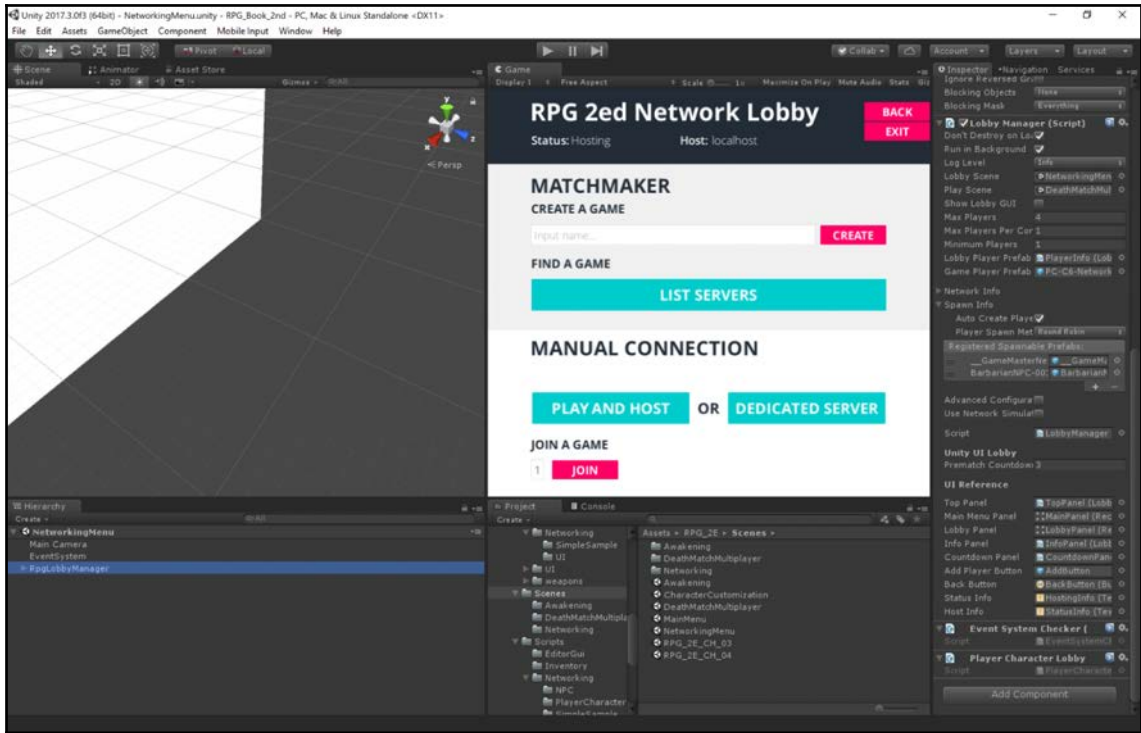
The `NetworkingMenu` scene is our lobby scene, so it will be exactly the same as the Tank Multiplayer lobby scene, with the exception that the player character will be replaced by the Barbarian prefab, and the `Registered Spawnable Prefabs` will be different. We shall see this later.

In the DeathMatchMultiplayer scene, I have the following level design:



Level design spawn enemy

Your lobby scene should look something like the following:



The next step is to make sure our GameObjects are network-enabled.

Networked player character

Go ahead and drag the player prefab you have created into the scene. We will use it as a base to create a new prefab, which will be used for the networked version of the game.

Go ahead and remove the existing `BarbarianCharacterController.cs` and `BarbarianCharacterCustomization.cs` components from the instance. We will create new scripts that are network-enabled and use them. Rename the PC GameObject instance to PC-C6-Network. Now make a prefab of the instance. You should now have a new prefab named PC-C6-Network.

Go ahead and attach the following components to the prefab: `NetworkIdentity`, `NetworkTransform`, and `NetworkAnimator`, by using the Inspector Window and selecting **Add Components | Network | <component name>**.

On the `NetworkIdentity` component, set `Local Player Authority` to `True`. On the `NetworkTransform` component, change **Transform Sync Mode** to **Sync Rigidbody 3D**. On the `NetworkAnimator` component, you will need to drag the `Animator` components attached to the GameObject into the `Animator` slot.



You will need to select the `Animator` component and drag it right down into the `Animator` slot on the `NetworkAnimator` components.

Next, we need to create a new character controller so that it is network compatible.

Create a new C# script and call it `BarbarianCharacterNetworkController.cs`. Attach the script to the PC-C6-Network prefab. The new character controller is a stripped-down version of the original character controller.

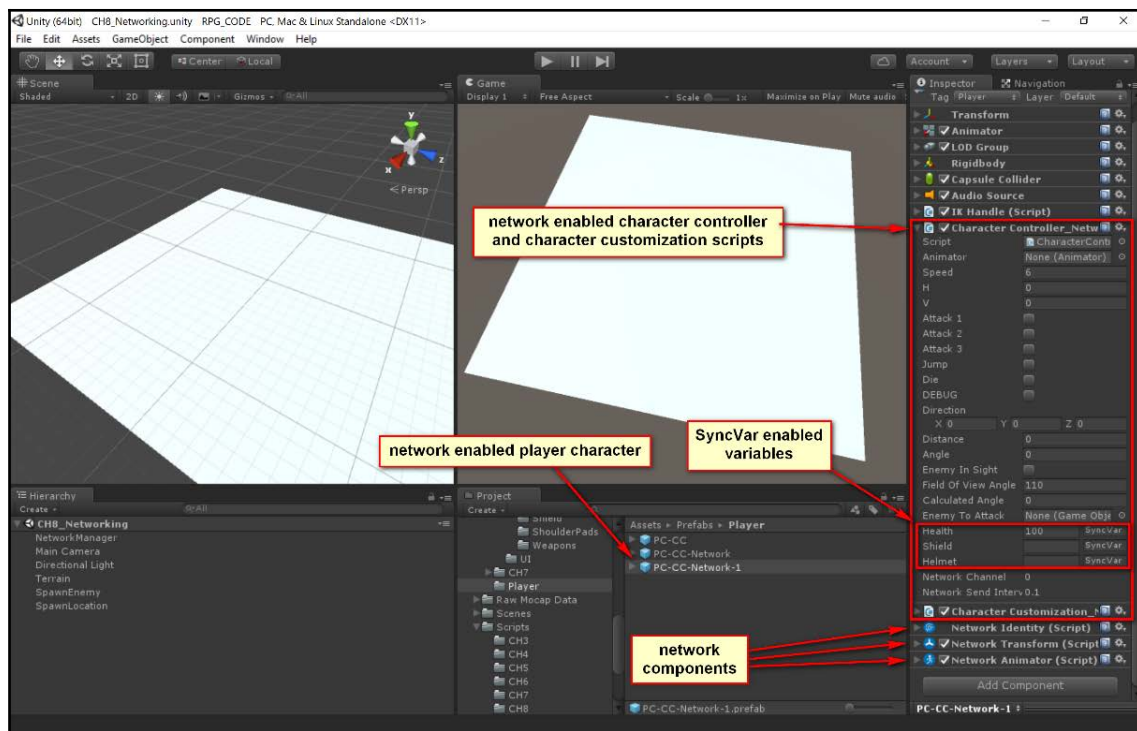
The listing for it is as follows:

```
using UnityEngine;
using UnityEngine.Networking;
namespace com.noorcon.rpg2e
{
    public class BarbarianCharacterNetworkController : NetworkBehaviour
    {
        public Transform mainCamera;
        public float cameraDistance = 16f;
        public float cameraHeight = 16f;
        public Vector3 cameraOffset;
        public Animator animator;
```

```
public float directionDampTime;
public float speed = 6.0f;
public float h = 0.0f;
public float v = 0.0f;
bool attack = false;
bool punch = false;
bool run = false;
bool jump = false;
[HideInInspector]
public bool die = false;
bool dead = false;
[SyncVar]
public bool EnemyInSight;
public GameObject EnemyToAttack;
Quaternion StartingAttackAngle = Quaternion.AngleAxis(-25, Vector3.up);
Quaternion StepAttackAngle = Quaternion.AngleAxis(5, Vector3.up);
Vector3 AttackDistance = new Vector3(0, 0, 2);
// Use this for initialization
void Start()
{
    cameraOffset = new Vector3(0f, cameraHeight, -cameraDistance);
    mainCamera = Camera.main.transform;
    MoveCamera();
    animator = GetComponent<Animator>() as Animator;
    EnemyInSight = false;
}
// Update is called once per frame
private Vector3 moveDirection = Vector3.zero;
void Update()
{
    if (!isLocalPlayer)
        return;
    if (dead)
    {
        animator.SetBool("Die", false);
        return;
    }
    if (Input.GetKeyDown(KeyCode.C))
    {
        attack = true;
        this.GetComponent<IKHandle>().enabled = false;
    }
    if (Input.GetKeyUp(KeyCode.C))
    {
        attack = false;
        this.GetComponent<IKHandle>().enabled = true;
    }
    animator.SetBool("Attack", attack);
}
```

```
if (Input.GetKeyDown(KeyCode.P))
{
    punch = true;
    this.GetComponent<IKHandle>().enabled = false;
}
if (Input.GetKeyUp(KeyCode.P))
{
    punch = false;
    this.GetComponent<IKHandle>().enabled = true;
}
animator.SetBool("Punch", punch);
if (Input.GetKeyDown(KeyCode.LeftShift))
{
    this.run = true;
    this.GetComponent<IKHandle>().enabled = false;
}
if (Input.GetKeyUp(KeyCode.LeftShift))
{
    this.run = false;
    this.GetComponent<IKHandle>().enabled = true;
}
animator.SetBool("Run", run);
if (Input.GetKeyDown(KeyCode.Space))
{
    jump = true;
    this.GetComponent<IKHandle>().enabled = false;
}
if (Input.GetKeyUp(KeyCode.Space))
{
    ...
}
```

The first thing you should notice is that we are inheriting from `NetworkBehaviour` instead of `MonoBehaviour`. This is necessary if we want to enable certain network behaviors on the `GameObject`. Take a look at the following screenshot:



Level setup for RPG network

Next, let's look at some of the variables that need to be synchronized across the network for each player character that is connected. These variables are `enemyToAttack` and `Health`. There are two more variables, `Shield` and `Helmet`, which we will discuss later.

In the `Update()` function, we need a way to check and make sure that it is the local player before giving the controller the chance to execute the player. This is done by having the following code check to see if the current client is the local player:

```
if (!isLocalPlayer)
    return;
```

This will make sure that the code runs only for the current client (player). The rest of the code in the `Update()` function checks to see if the enemy is in sight, and makes sure the player character is facing the enemy to attack.

If the player is in attack mode and the enemy is in our view, we set `enemyInSight` to `True` and `enemyToAttack` to the enemy `GameObject`, which is stored in the `hitAttack` variable of type `RaycastHit`. The important element here is the `CmdEnemyToAttack()` function. The client needs to send a command to the server telling the server who the target of attack is:

```
[Command]
void CmdEnemyToAttack(GameObject go)
{
    this.enemyInSight = true;
    this.enemyToAttack = go;
}
```

This will make sure that the data is registered correctly on the server and that it is synchronized to other clients. We also have another function, called `CmdEnemyTakeDamage()`, which is used to reduce the health of the enemy character on the server. The server then calls the `RpcEnemyTakeDamage()` function to synchronize the health value of the enemy across all clients:

```
[Command]
void CmdEnemyTakeDamage(float value)
{
    RpcEnemyTakeDamage(value);
}

[ClientRpc]
void RpcEnemyTakeDamage(float value)
{
    if (this.enemyToAttack != null)
        this.enemyToAttack.GetComponent<NPC_Movement_Network>().Damage(value);
}
```

This idea is a bit confusing at first, but it will be clearer as you start to study it more carefully.

We also have the following function to send commands to the server when the player dies:

```
[Command]
void CmdPlayerCharacterIsDead()
{
    RpcPlayerCharacterIsDead();
}
[ClientRpc]
void RpcPlayerCharacterIsDead()
{
    this.die = true;
    Destroy(this.gameObject, 2.0f);
}
```

The preceding functions make sure that the player character is dead and destroyed on all connected clients at the moment of the game.

And finally, the following hook functions are used by the `SyncVar` on the `Health` and `enemyToAttack` variables:

```
// Var Sync hook function ...
void OnChangePlayerHealth(float health)
{
    this.Health = health;
}
// Var Sync hook function
void OnChangeEnemyToAttack(GameObject enemy)
{
    this.enemyToAttack = enemy;
}
```

This idea is also a bit confusing at first, but it will be clearer as you start to study it more carefully.

If you have not done so already, apply and save all of your changes to your `PC-CC-Network` prefab.

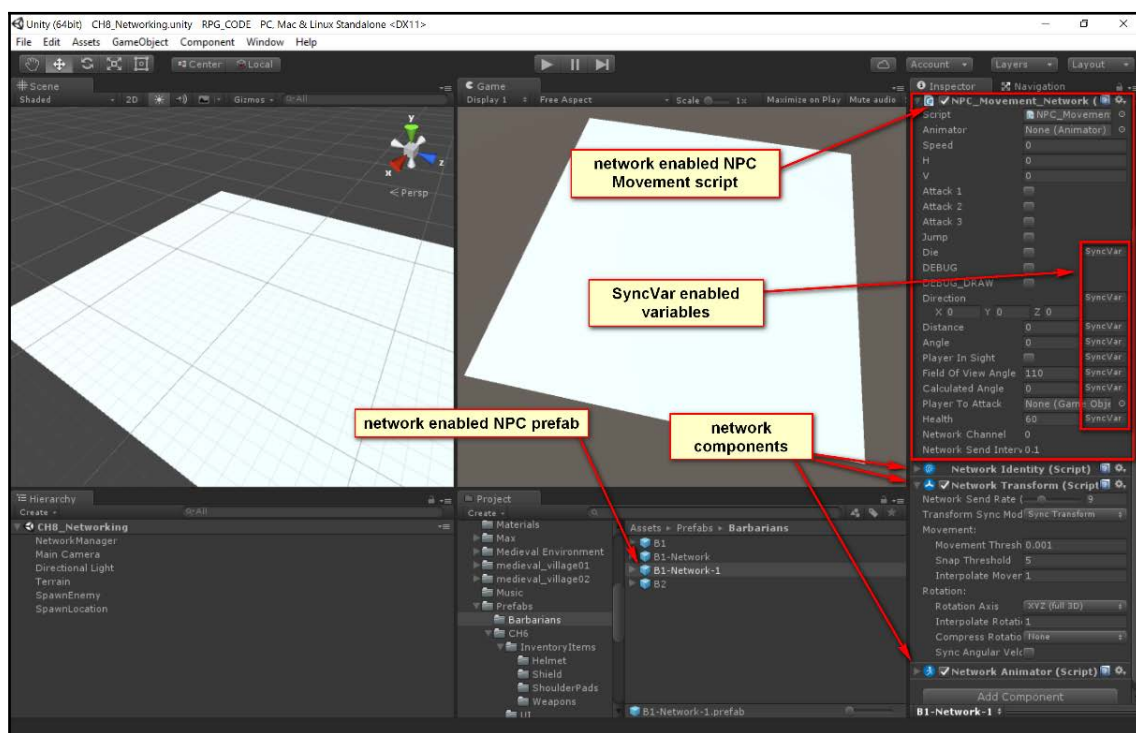
At this stage, your character is ready to be integrated with `NetworkManager`; you can drag and drop the prefab into the `Player Prefab` slot, and build a stand-alone version to test out your character movement and synchronization.

Networked non-player character

Just like the player character network-enabled prefab, we will use the non-player character prefab as our base to get started. Go ahead and create an instance of your NPC in the scene.

Go ahead and remove the existing `NPC_Movement.cs` component from the prefab. Rename the Prefab to `B1-Network` and attach the following components to it: `NetworkIdentity`, `NetworkTransform`, and `NetworkAnimator`, by selecting **Add Component | Network | <component name>** from the **Inspector** window.

On the `NetworkIdentity` component, set `Local Player Authority` to `True`; in the `NetworkTransform` component, set `Transform Sync Mode` to `Sync Transform`; and for the `NetworkAnimator` component, set the `Animator` slot to the `Animator` controller attached to the prefab, by dragging it and dropping into the slot, as shown in the following screenshot:



We now need to create a new script for our NPC movement that is network-enabled. Go ahead and create a new C# script, and name it `NPC_Movement_Network.cs`. A listing of the scripts is as follows:

```
using UnityEngine;
using UnityEngine.Networking;

using System.Collections;
```

```
public class NPC_Movement_Network : NetworkBehaviour {
    // reference to the animator
    public Animator animator;

    public bool jump = false;        // used for jumping

    [SyncVar(hook = "OnNPCIsDead")]
    public bool die = false;        // are we alive?

    ...

    // what is the field of view for our NPC?
    // currently set to 110 degrees
    [SyncVar]
    public float fieldOfViewAngle = 110.0f;

    // calculate the angle between PC and NPC
    [SyncVar]
    public float calculatedAngle;

    [SyncVar(hook = "OnChangePlayerToAttackInNPC")]
    public GameObject playerToAttack;

    [SyncVar(hook = "OnChangeNPCHealth")]
    public float Health = 100.0f;

    void Awake()
    {
        // get reference to the animator component
        this.animator = GetComponent<Animator>() as Animator;

        // get reference to nav mesh agent
        this.nav = GetComponent<NavMeshAgent>() as NavMeshAgent;

        // get reference to the sphere collider
        this.col = GetComponent<SphereCollider>() as SphereCollider;

        // we don't see the player by default
        this.playerInSight = false;
    }

    void Update()
    {
        // only execute the following code if local player ...
        if (!isServer)
            return;

        this.CmdUpdateNetwork();
    }
}
```



```
}

[Command]
void CmdUpdateNetwork()
{
    this.RpcUpdateNetwork();
}

[ClientRpc]
void RpcUpdateNetwork()
{
    // if player is in sight let's slerp towards the player
    if(this.playerToAttack!=null)
    {
        if (playerInSight)
        {
            this.transform.rotation =
                Quaternion.Slerp(this.transform.rotation,
                    Quaternion.LookRotation(direction), 0.1f);

            if
            (this.playerToAttack.transform.GetComponent<CharacterController_Network>().
            die)
            {
                animator.SetBool("Attack", false);
                animator.SetFloat("Speed", 0.0f);
                animator.SetFloat("AngularSpeed", 0.0f);
                this.playerInSight = false;
                this.playerToAttack = null;
            }
        }
    }

    if(this.Health<=0.0f)
    {
        this.die = true;
        this.Health = 0.0f;

        animator.SetBool("Attack", false);
        animator.SetFloat("Speed", 0.0f);
        animator.SetFloat("AngularSpeed", 0.0f);
        this.playerInSight = false;
        this.playerToAttack = null;
    }
}
```

```

        animator.SetBool("Die", die);
    }
    ...

```

There are a few variables that have been indicated as SyncVars; these are: die, distance, direction, angle, playerInSight, fieldOfViewAngle, calculatedAngle, playerToAttack, and Health.

Take a look at the following code:

```

void OnTriggerExit(Collider other)
{
    if (other.transform.tag.Equals("Player"))
    {
        distance = 0.0f;
        angle = 0.0f;
        this.attack1 = false;
        this.playerInSight = false;

        this.playerToAttack = null;
    }
}

// this is a helper function at this point
// in the future we will use it to calculate distance around the corners
// it currently is also used to draw the path of the nav mesh agent in
the
// editor
float CalculatePathLength(Vector3 targetPosition)
{
    // Create a path and set it based on a target position.
    NavMeshPath path = new NavMeshPath();
    if (nav.enabled)
        nav.CalculatePath(targetPosition, path);

    // Create an array of points which is the length of the number of
    corners in the path + 2.
    Vector3[] allWayPoints = new Vector3[path.corners.Length + 2];

    // The first point is the enemy's position.
    allWayPoints[0] = transform.position;

    // The last point is the target position.
    allWayPoints[allWayPoints.Length - 1] = targetPosition;

    // The points inbetween are the corners of the path.
    for (int i = 0; i < path.corners.Length; i++)
    {

```

```

        allWayPoints[i + 1] = path.corners[i];
    }

    // Create a float to store the path length that is by default 0.
    float pathLength = 0;

    // Increment the path length by an amount equal to the distance between
    each waypoint and the next.
    for (int i = 0; i < allWayPoints.Length - 1; i++)
    {
        pathLength += Vector3.Distance(allWayPoints[i], allWayPoints[i + 1]);

        if (DEBUG_DRAW)
            Debug.DrawLine(allWayPoints[i], allWayPoints[i + 1], Color.red);
    }

    return pathLength;
}
}

```

Some of the SyncVars have a hook; these are Health, playerToAttack, playerInSight, and die.

In the Update() function, we check to make sure we are the server by the following line:

```

// only execute the following code if server ...
if (!isServer)
    return;

```

If we are the server, we use the CmdUpdateNetwork() and RpcUpdateNetwork() functions to perform our duties. These are just for the movement and action for the NPC. The key here is the SyncVars and hook functions that are used to synchronize the NPC data to all clients:

```

public void OnChangePlayerPlayerInSight(bool value)
{
    this.playerInSight = value;
}
// Var Sync hook function ...
void OnChangeNPCHealth(float health)
{
    this.Health = health;
}
void OnNPCIsDead(bool value)
{
    die = true;
}

```

```
void OnChangePlayerToAttackInNPC(GameObject player)
{
    this.playerToAttack = player;
}
```

That is all we need for the NPC. Go ahead and add the script to the prefab and apply the changes. Save it.

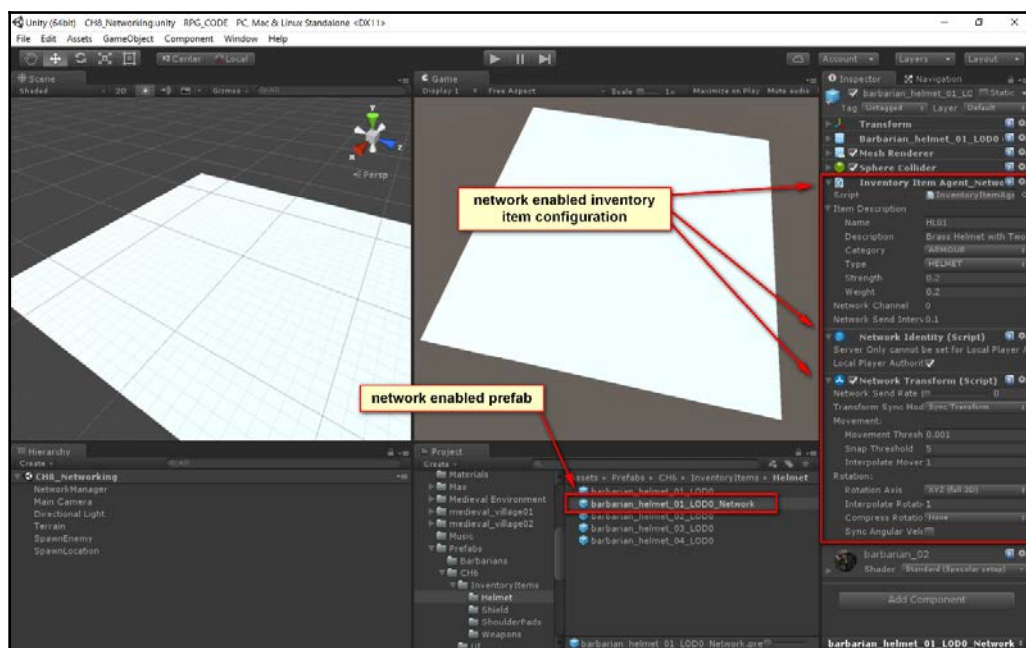
Your new NPC prefab should have the following components attached:

Synchronizing player customization and items

In order for this to work, we need to perform configure and create several more new inventory item prefabs. I will use two inventory items to demonstrate this particular point.

I will use one of the `Helmet` prefabs from my inventory items. Duplicate it, and remove the `InventoryItemAgent.cs` component. We will create a new script that is network-enabled, as we did for our PC and NPC.

Attach the following components to the instance: `NetworkIdentity` and `NetworkTransform`, using **Add Component | Network | <component name>** from the **Inspector** window:



Create a new script named `InventoryItemAgent_Network.cs`. The listing is as follows:

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class InventoryItemAgent_Network : NetworkBehaviour {

    public InventoryItem ItemDescription;

    public void OnTriggerEnter(Collider c)
    {
        // make sure we are colliding with the player
        if (c.gameObject.tag.Equals("Player"))
        {
            // Make a copy of the Inventory Item Object
            InventoryItem myItem = new InventoryItem();
            myItem.CopyInventoryItem(this.ItemDescription);

            c.gameObject.GetComponent<CharacterController_Network>().PlayerArmourChange
            d(myItem);
        }
    }
}
```

All this script does is assign the inventory item to the player character using the `PlayerArmourChanged()` function in the `CharacterController_Network.cs` script.

The `PlayerArmourChanged()` function uses another script we need to create that is network enabled, and that is the `CharacterCustomization_Network.cs` script. I will not list the script here as it is very long. You can look at the script in the code supplied with the book.

Spawning NPCs and other items

We need a way to spawn our NPC as well as the inventory items we will be using for the next demonstration.

In the **Hierarchy** window, right-click and select **Create Empty**; this will create an `Empty GameObject`. Rename it `SpawnEnemy`, and add a `NetworkIdentity` component to it by selecting **Add Component** | **Network** | **NetworkIdentity** from the **Inspector** window.

We will create a new script called `EnemySpawn_Network.cs`. The listing is as follows:

```
using UnityEngine;
```

```
using UnityEngine.Networking; // used for chapter 8

using System.Collections;

public class EnemySpawn_Network : NetworkBehaviour
{
    public GameObject enemyPrefab;
    public Transform spawnLocation;

    public GameObject inventoryItemPrefab;
    public GameObject inventoryItemShield;

    public override void OnStartServer()
    {
        GameObject go = GameObject.Instantiate(enemyPrefab,
spawnLocation.position, Quaternion.identity) as GameObject;
        NetworkServer.Spawn(go);

        GameObject goInventoryItem1 =
GameObject.Instantiate(inventoryItemPrefab, new Vector3(2, 1, 2),
Quaternion.identity) as GameObject;
        NetworkServer.Spawn(goInventoryItem1);

        GameObject goInventoryItem2 =
GameObject.Instantiate(inventoryItemShield, new Vector3(3, 1, 2),
Quaternion.identity) as GameObject;
        NetworkServer.Spawn(goInventoryItem2);

    }
}
```

The script is very simple, as you can see. We are just referencing the GameObjects that represent the prefabs for the NPC and inventory items prefab.

Attach the new script to the `SpawnEnemy` prefab in the **Hierarchy** Window.

Testing our network-enabled PC and NPC

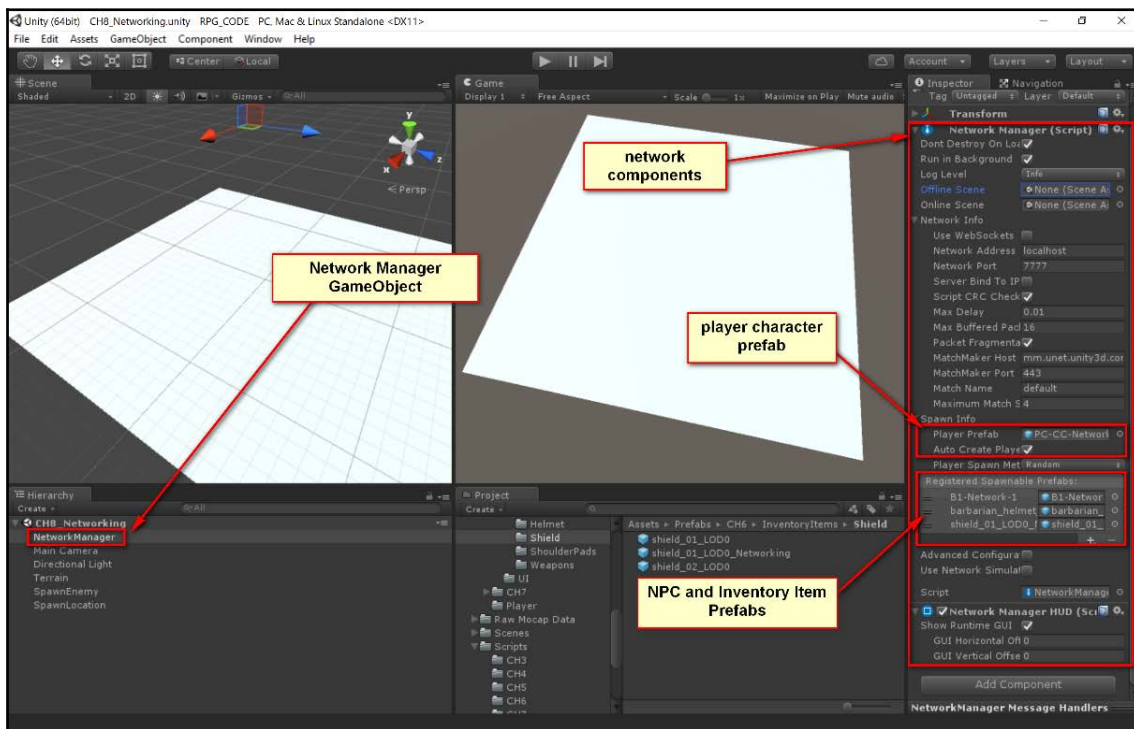
At this point, we have all of the assets needed to test out our network-enabled RPG characters. There is one final step that we need to perform if you have not done so already.

Select the `NetworkManager` GameObject in the **Hierarchy** window, and from the **Inspector** window, you will need to make sure certain things have been assigned in the **Spawn Info** section.

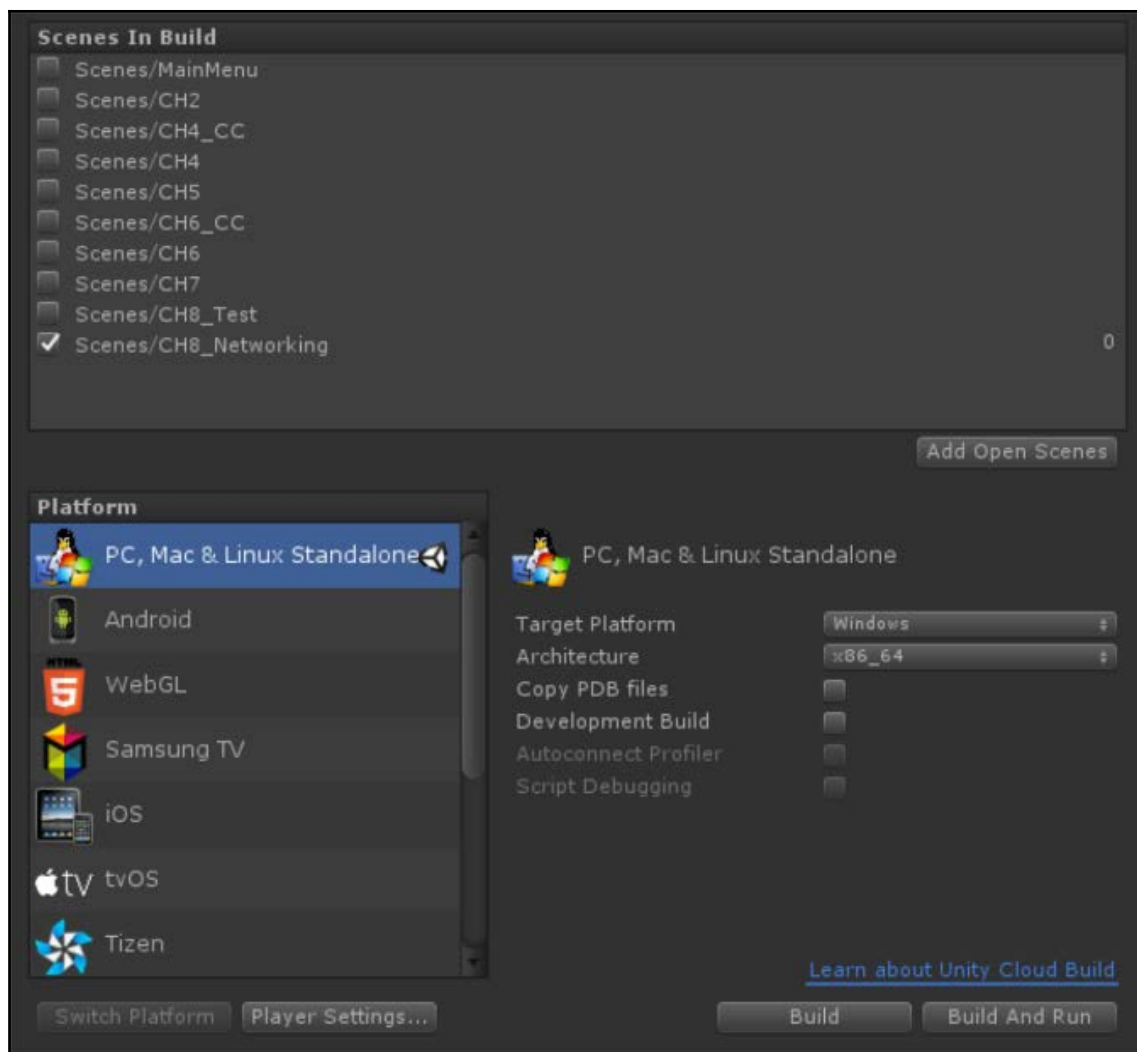
Player Prefab should be assigned to your player character prefab. Mine is named PC-CC-Network-1. Make sure **Auto Create Player** is set to True.

You will also register your NPC prefabs and other network-enabled non-character prefabs in the **Registered Spawnable Prefabs**. I have the barbarian prefab named B1-Network-1 assigned, barbarian_helmet_01_LOD0_Network, and shield_01_LOD0_Network.

Take a look at the following screenshot:

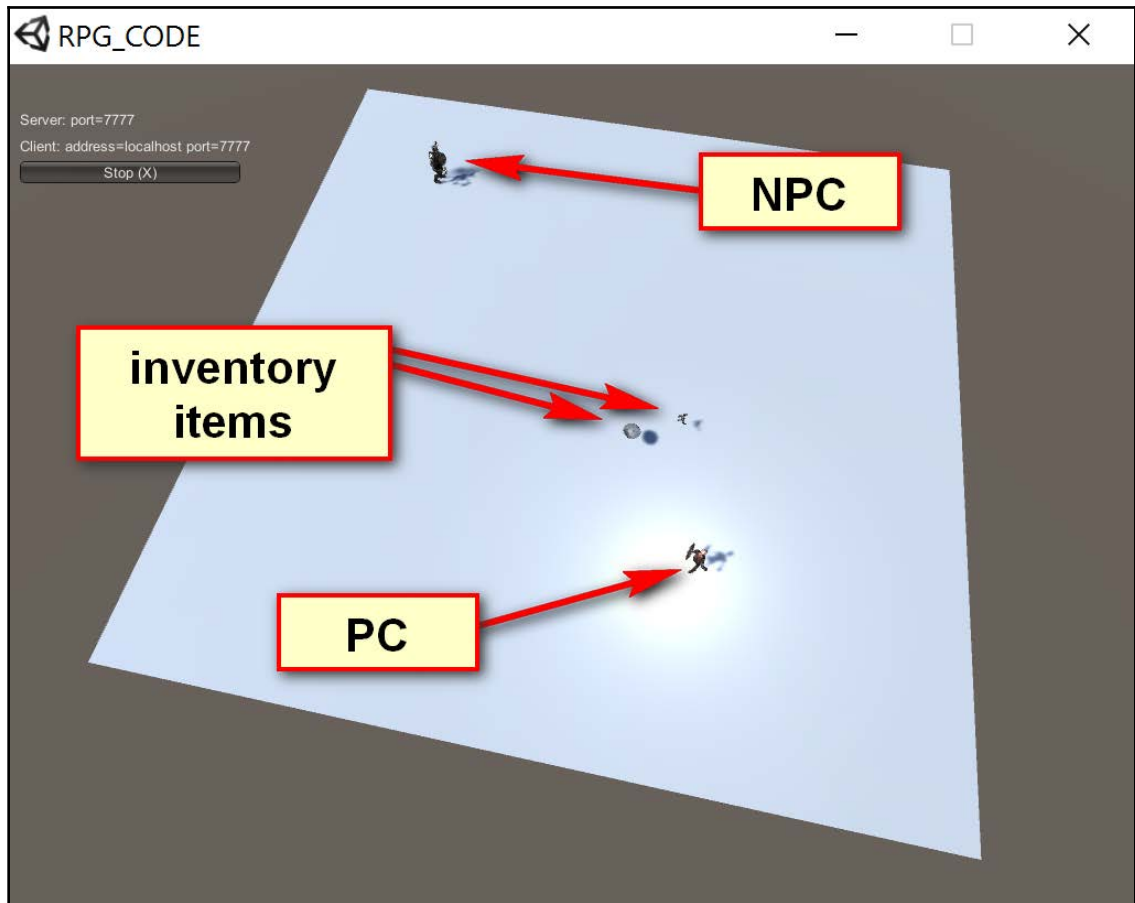


Alright, at last, we can do a build. Let's go ahead and make a stand-alone build of our game. Make sure that the current scene is in the build configuration:

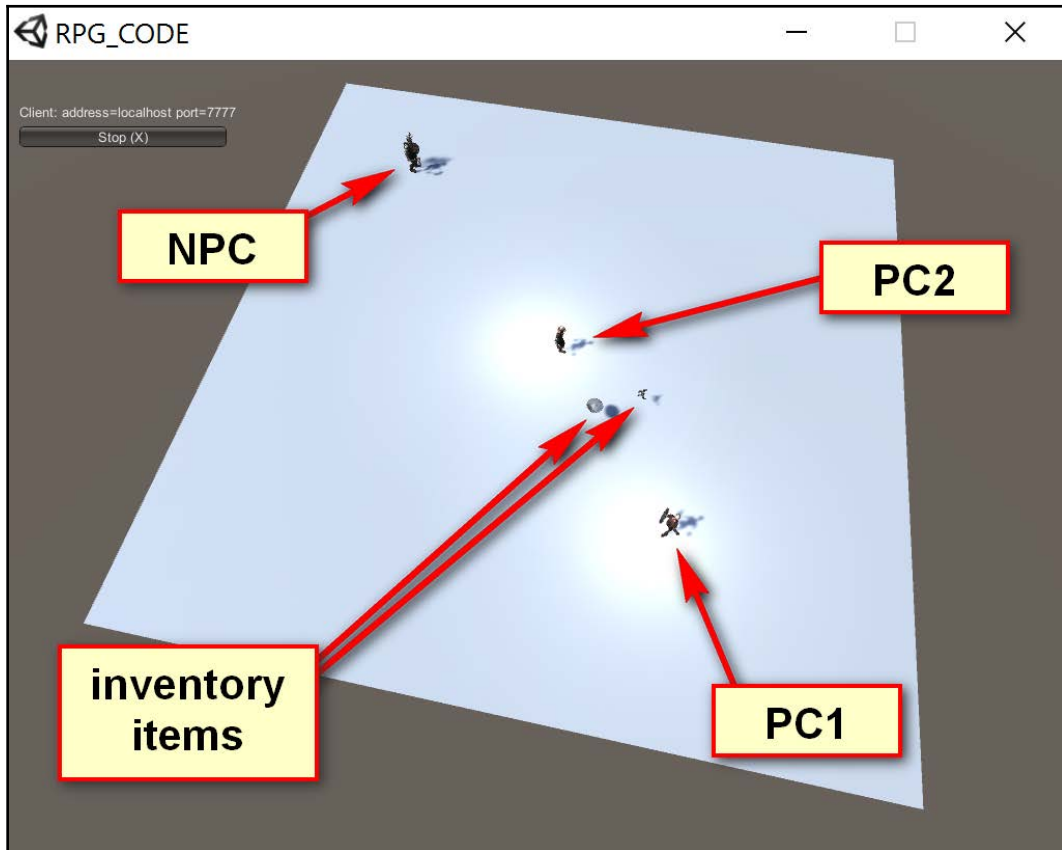


Go ahead and launch two instances of the build. Make one of them the host, and the other the client.

Take a look at the following screenshot:

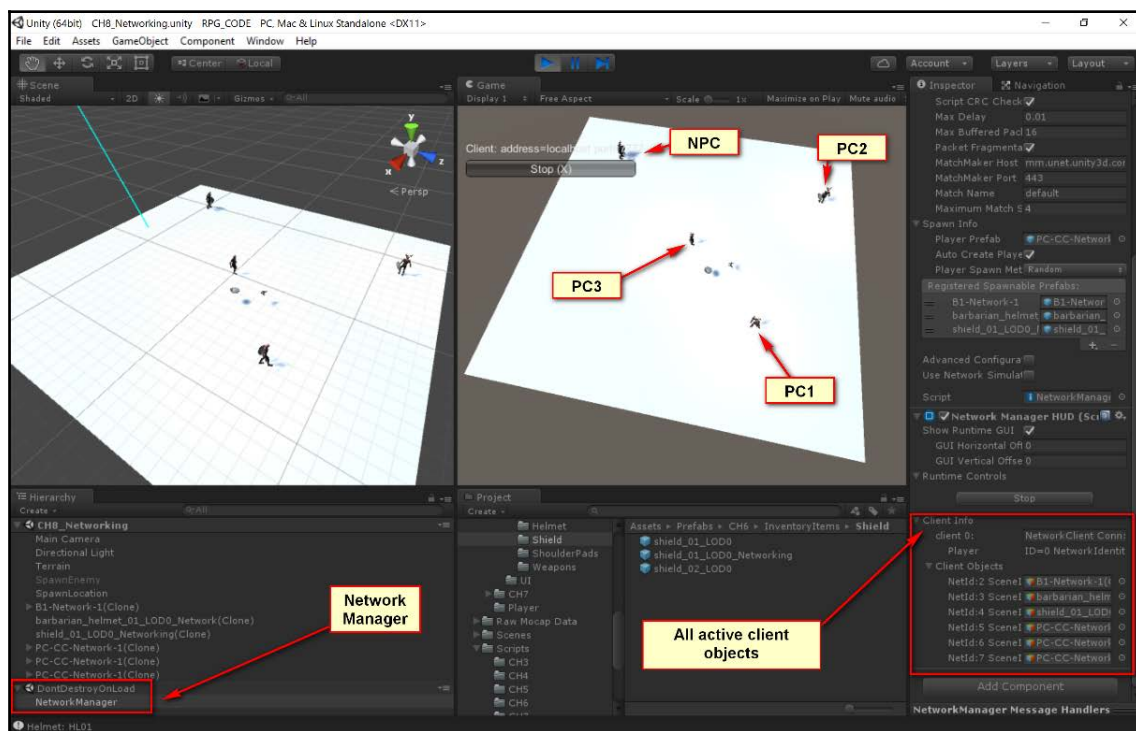


In the preceding screenshot, we have started a client as a server and the player character has picked up one of the inventory items, a shield. When we connect the second client, it should correctly take into consideration the current state of all PCs and NPCs active in the game:



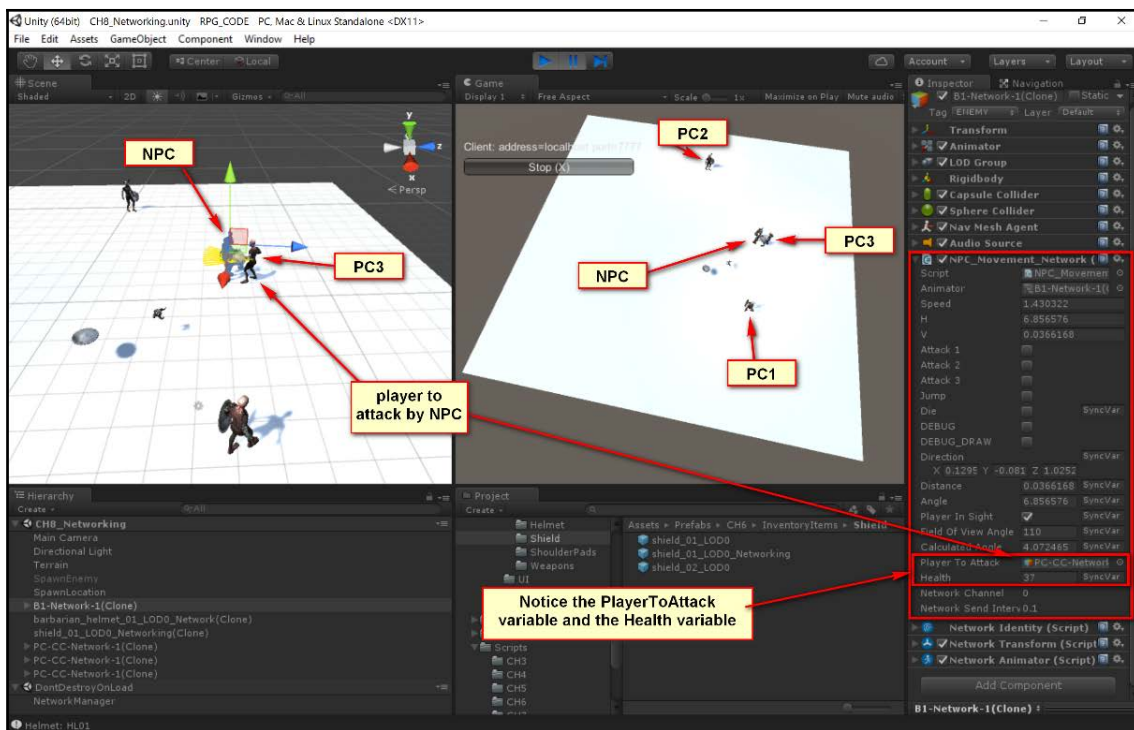
Keeping both of the instances running, use the Unity IDE to connect the third client. You can use the client to perform debugging on the client end, and also to see what is going on.

Take a look at the following screenshot:



In the preceding screenshot, you can see all of the player characters and how they have been accurately synchronized with one another. Select B1-Network-1 GameObject from the Hierarchy Window, and use one of the client instances to take the player character and attack the NPC.

We will pause the editor and inspect the variables and how they have been properly synchronized, as shown in the following screenshot:



What's next?

As you have witnessed, network programming is simple, but at the same time, it can be difficult. The difficulty lies in managing and understanding the synchronization of the data between all players in an efficient and meaningful way.

It can actually get a bit more involved if you are truly considering creating a game with a large number of clients. Unity networking will not be able to handle that; you will need to create your own backend server managers and messaging systems.

What we have covered in this chapter will give you a good understanding to take it to the next level. Keep on coding until we meet again!

Summary

In this chapter, we looked at network programming using the Unity networking components. The main objective of the chapter was to introduce you to the fundamentals of networking in Unity by implementing two samples.

We started the chapter by discussing some of the challenges you will face as a game designer and developer for a multiplayer game. One of the main questions raised is whether you really need to invest the time and energy to create a multiplayer mode for your game. Assuming you really do want or need to create a multiplayer game, we started by looking at the different types of multiplayer games popular today.

We then went on with our first example of a simplified multiplayer game. The multiplayer game we developed are real-time, that is, all clients are synchronized with one another based on the activity on each active player's state; namely, the position, rotation, movement, and other important data needs to be synchronized with all clients connected to the game session.

We looked at the fundamentals of Unity's networking components, such as the Network Manager, Network Manager HUD, Network Identity, and Network Transform. These are the base components that are used to illustrate multiplayer programming in Unity. Once we understood what these components are used for, we started with a simple sample.

We created a simple tank game that demonstrated how to put together all of the essential components for a multiplayer game. We created the necessary player character prefab with the appropriate network-enabled scripts and components. We also created the non-player character prefab with its own network-enabled scripts. The game demonstrates how to spawn, and how to synchronize between the player characters and non-player characters.

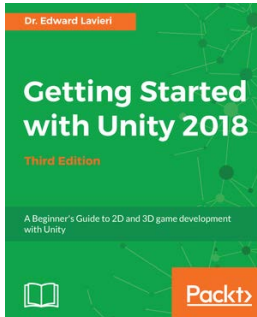
During the construction of the tank game, we covered how to synchronize variables and use network callbacks that are crucial for the development of a multiplayer game. We also covered what *Commands* and *ClientRPC* calls are and how to use them properly.

Next, we applied what we learned to the RPG characters we developed in previous chapters. We used the existing prefabs as a base, and extended them to include the networking components and create new network-enabled scripts to handle the character movement, character customization, and NPC movement scripts.

One of the crucial elements that we covered is the synchronization of each player character's inventory items visually with the rest of the players. We closed the chapter by testing and discussing how to debug your code on the client and the server while developing your multiplayer game.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

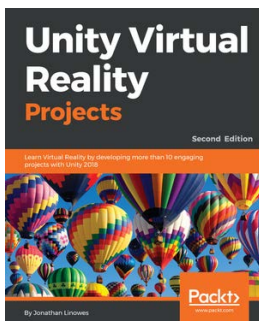


Getting Started with Unity 2018 - Third Edition

Dr. Edward Lavieri

ISBN: 978-1-78883-010-2

- Set up your Unity development environment and navigate its tools
- Import and use custom assets and asset packages to add characters to your game
- Build a 3D game world with a custom terrain, water, sky, mountains, and trees
- Animate game characters, using animation controllers, and scripting
- Apply audio and particle effects to the game
- Create intuitive game menus and interface elements
- Customize your game with sound effects, shadows, lighting effects, and rendering options
- Debug code and provide smooth error handling



Unity Virtual Reality Projects - Second Edition

Jonathan Linowes

ISBN: 978-1-78847-880-9

- Create 3D scenes with Unity and other 3D tools while learning about world space and scale
- Build and run VR applications for specific headsets, including Oculus, Vive, and Daydream
- Interact with virtual objects using eye gaze, hand controllers, and user input events
- Move around your VR scenes using locomotion and teleportation
- Implement an audio fireball game using physics and particle systems
- Implement an art gallery tour with teleportation and data info
- Design and build a VR storytelling animation with a soundtrack and timelines
- Create social VR experiences with Unity networking

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- animation curve
 - setting 107
- Animator Controller
 - animation states, creating 88, 89, 90, 91, 92
 - used, for creating character motion 87
- artificial-intelligence (AI) 7
- asset inventory, Zazar dynasty
 - about 39
 - character assets 42
 - environment assets 39
- Asset Store 39
- Awakening scene, Zazar dynasty
 - creating 53, 55
 - custom toolkit, using 57, 59, 60, 61, 62, 63, 64, 65
 - skybox, using 65, 66
 - terrain model, using 55, 56

B

- Base Character class
 - attributes 75, 77
- built-in erosion types, Terrain toolkit
 - Hydraulic Erosion 52
 - Thermal Erosion 52
 - Tidal Erosion 52

C

- character assets, Zazar dynasty
 - about 42
 - Barbarians 43
 - Free Assets 45
 - Orcs 44
 - Villagers 44
- character controller
 - animation, modifications 99, 101

- character model
 - about 80
 - default character models 80
 - modifying 83, 84, 85
 - rigging 85, 86
- character motion
 - creating, with Animator Controller 87
 - handling, with character controller 93, 95, 96, 98
 - implementing 87
- character
 - assets 75
 - definitions 74
 - states, implementing 77, 78, 79
- Citadel
 - about 24
 - reference 24
- combat system
 - about 17
 - types 17
- contest tree 29, 30
- cRPG story
 - about 8
 - exploration 9
 - quests 9
- customizable parts, player
 - armor 113
 - body type 113
 - boots 114
 - helmet 114
 - shields 113
 - skins 114
 - weapons 114
- Cyberpunk 2077
 - about 25
 - reference 25

D

default character models

- about 80
- Barbarian 80
- Orc 83
- Villager 81

Dungeons & Dragons (D&D) 6

dynamic item viewer

- designing 206
- elements, adding to PanelItem 211, 212
- elements, adding to ScrollView 211, 212
- scroll view, adding 207, 208, 209, 210
- txtItemElement, adding dynamically 213, 214

E

environment assets, Zazar dynasty

- about 39
- Medieval Environment Pack 40
- Nature Starter Kit 2 41
- Terrain Toolkit 2017 41

F

fbx 193

filter, types

- Normalize 52
- Smooth 52

First-Person Shooter (FPS) 247

G

game level controller

- modifying 180

Game Master (GM) 6

GameMaster

- about 154, 155
- audio controller, creating 165, 168, 169
- audio, managing 156, 157, 159, 160
- code, testing 180, 181
- game settings, managing 156, 157, 159, 160
- improving 163
- level controller 163, 165
- scenes, managing 160, 162, 163

genre

- history 5

H

Heads-Up Display (HUD)

- about 19
- active inventory items panel, creating 255, 257, 258
- character info panel, creating 250, 251, 252, 253, 254, 255
- design, completing 250
- designing 247, 248
- enemy stats, displaying 275
- framework, creating 249, 250
- items panel, creating 259, 260
- NPC canvas, creating 276, 277, 278, 280
- NPC stats user interface, creating 276
- player character, attacking 280, 281, 283, 284, 285

I

inventory interface

- designing 201, 203
- dynamic item viewer, designing 206
- final inventory item UI, building 215, 216, 217
- UI framework, creating 203, 205, 206

inventory item

- agent, adding 196, 197, 198, 199
- applying 229, 231, 234, 235, 238, 239, 241, 242
- applying, to player character 243, 244
- as prefabs 200, 201
- category buttons, hooking 221
- creating 193, 194
- player character, modifying 228
- prefab, creating 194

inventory system

- about 10, 11, 12
- category buttons, hooking 217
- data, displaying 217, 219, 221
- designing 185
- item types, determining 186, 187, 188, 190, 192, 193
- testing 222, 223, 225, 226, 227, 228
- UI, interacting 217
- weighted inventory 185, 186

Inverse Kinematics (IK)

- about 101, 102, 104, 105, 106

animation curve, setting 107

L

Last Man Standing 30

level design, Zazar dynasty

about 46, 47

Awakening scene, creating 53, 55

stage, setting 48, 49, 51

terrain toolkit 51, 52

testing 66, 68

Live Action Role Playing (LARP) 6

M

Medieval Environment Pack 40

MU Legend

about 21, 22

reference 22

multiplayer game

building 315, 317, 318

challenges 292

enemy tank, adding 310, 311, 313, 315

implementing 293

my tank networking project, creating 295

networking components 294, 295

player character, adding 297, 298, 299, 300

testing 315, 317, 318

N

Nature Starter Kit 2 41

Negotiated Contest 31, 32

network callbacks

OnCheckObserver() 301

OnDeSerialize() 301

OnNetworkDestroy() 301

OnRebuildObservers() 301

OnSerialize() 301

OnSetLocalVisibility() 301

OnStartClient() 301

OnStartLocalPlayer() 301

OnStartServer() 301

network programming 340

network-enabled RPG characters

about 318

items, spawning 333

items, synchronizing 332, 333

networked non-player character, creating 326,
327, 330, 331

networked player character, creating 321, 323,
325, 326

NPCs, spawning 333

player customization, synchronizing 332, 333

scene, creating 318, 320

testing 334, 335, 337, 338, 339

networking components

Network Identity 295

Network Lobby Manager 295

Network Manager 295

Network Manager HUD 295

Network Transform 295

non-player character

about 123, 124

AI, implementing 136, 138, 139, 140, 142, 143

Animator Controller (AC), creating 130, 132,
133

attack mode, implementing 133, 134, 135

basics 125

player character, interacting with 144, 145, 146,
148, 149, 150, 151

setting up 125, 126, 127, 129, 130

P

patterns, RPG

about 26

contest tree 29, 30

Last Man Standing 30

Negotiated Contest 31, 32

terminology 27, 28

pen-and-paper (PnP) 6

player character, multiplayer game

adding 297, 298, 299, 300

cannonball, creating for tank 304, 306

client RPC calls, creating 302, 304

commands, sending 301

network callbacks 301

network lobby manager, configuring 306, 307,
309, 310

tank prefab, creating 306, 307, 309, 310

variables, synchronizing 300

player character

class enhancements 169, 170, 171

- code, enhancing 285, 287, 288, 289
- customizable parts 113, 115
- customization class, updating 172, 174, 176, 178, 179
- customization code 118, 120, 121
- customization scene 123
- customizing 110, 111, 112
- data management 169
- interacting, with non-player character 144, 145, 146, 148, 149, 150, 151
- inventory item, applying 243, 244
- state, preserving 122
- User Interface (UI), using 115, 116, 117

R

- Raycasting 287
- Real-Time Strategy (RTS) game 247
- Role-Playing Game (RPG)
 - about 5, 21, 247
 - building 33
 - characteristics 7
 - Citadel 24
 - Computer RPG (cRPG) 6
 - Cyberpunk 2077 25
 - Live action 6
 - MU Legend 22
 - Tabletop 6
 - Titan Siege 23
- Root Motion 87
- RPG, key elements
 - character actions 13, 14
 - character attributes 13, 14
 - combat system 17
 - experience 15, 17
 - exploration 8
 - graphics 18, 19, 21
 - inventory system 10, 11, 12
 - leveling 15, 17
 - quests 8
 - setting 7
 - story 7
 - user interaction 18, 19, 21

T

- terminology, RPG games
 - attribute 27
 - character 27
 - characteristics 27
 - common characteristics 27
 - conflict 27
 - contest 27
 - derived attributes 27
 - drama 27
 - flaw 27
 - fortune 27
 - Game Master 27
 - gauge 27
 - gift 28
 - karma 28
 - Non-Player Character (NPC) 28
 - Optional Characteristics 28
 - player 28
 - Player Character (PC) 28
 - Primary Attribute 28
 - rank 28
 - Ranked Trait 28
 - Selected Characteristic 28
 - shared gauge 28
 - skill 28
 - trait 28
- Terrain Toolkit 2017, pre-defined generators
 - Fractal 51
 - Perlin 51
 - Voronoi 51
- Terrain Toolkit 2017
 - about 41
 - using 49
- Titan Siege
 - about 23
 - reference 23

U

- UI elements
 - code, integrating 261, 262, 263, 264, 265, 266, 267, 269, 270, 272, 273, 274, 275
- Unity Editor 50

W

weighted inventory 185, 186

Z

Zazar dynasty, quests

- about 34
- awakening 35
- broken forest 36
- kingdom 38

village 35

Zazar dynasty, story

- about 34
- backstory 34
- exploration 34
- quests 34

Zazar dynasty

- asset inventory 39
- GameMaster script, creating 69, 70, 72, 73
- level design 46
- main menu, creating 68, 69