Community Experience Distilled

Learning C++ by Creating Games with UE4

Learn C++ programming with a fun, real-world application that allows you to create your own games!

William Sherif

www.it-ebooks.info

Learning C++ by Creating Games with UE4

Learn C++ programming with a fun, real-world application that allows you to create your own games!

William Sherif

BIRMINGHAM - MUMBAI

www.it-ebooks.info

Learning C++ by Creating Games with UE4

Copyright © 2015 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: February 2015

Production reference: 1180215

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

ISBN 978-1-78439-657-2

www.packtpub.com

www.it-ebooks.info

Credits

Author
William Sherif

Reviewers
Brandon Mann
Matt Sutherlin
Alan Wolfe

Commissioning Editor
Edward Bowkett

Acquisition Editor
Subho Gupta

Content Development Editor
Anand Singh

Technical Editor
Saurabh Malhotra

Copy Editors
Dipti Kapadia
Deepa Nambiar

Project Coordinator
Rashi Khivansara

Proofreaders
Martin Diver
Lawrence A. Herman
Paul Hindle

Indexer
Hemangini Bari

Graphics
Sheetal Aute

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

www.it-ebooks.info

About the Author

William Sherif is a C++ programmer with more than 8 years' programming experience. He has a wide range of experience in the programming world, from game programming to web programming. He has also worked as a university course instructor (sessional) for 7 years.

He has released several apps on to the iTunes store, including strum and MARSHALL OF THE ELITE SQUADRON.

In the past, he has won acclaim for delivering course material in an easy-to-understand manner.

I'd like to thank my family, for their support when I was writing this book; Mostafa and Fatima, for their hospitality; as well as Ryan and Reda, for letting me write.

www.it-ebooks.info

About the Reviewers

Brandon Mann is a well-rounded game developer with over 7 years of professional game-development experience. He has worked on a wide range of titles, from Indie Games to AAA titles, and at companies such as Warner Bros., Midway, and 343 Industries. He has worked on several titles, including *Blacklight*, *Tango Down*, *Gotham City Impostors*, and *Battle Nations*.

Matt Sutherlin has been working in the games industry for over a decade now, where he's held job titles ranging from QA and scripter to engine programmer, and technical artist. Most recently, he has been heavily focusing on graphics technology, working on engine renderers, art pipelines, and shaders for AAA titles, such as *Heroes of the Storm* and *Halo 5: Guardians*.

I would like to thank my beautiful wife, Megan, for years of support and understanding and Matthew Phillips for giving me my debut in the industry.

www.it-ebooks.info

Alan Wolfe is a self-taught game and engine programmer who has worked at companies such as inXile Entertainment, Midway, Warner Bros., and Blizzard Entertainment. He has worked on titles including *Line Rider 2*, *Unbound*, *Gotham City Impostors*, *Battle Nations*, *Insanely Twisted Shadow Planet*, and *StarCraft II*: *Heart of the Swarm*. Alan is currently a senior engine programmer at Blizzard Entertainment, where he works on *StarCraft II* and *Heroes of the Storm*.

I'd like to thank Packt Publishing and the author for allowing me to contribute to this book and to help budding game programmers learn the same way I did. If you want to succeed as a game programmer, practice implementing everything you learn, hang out with like-minded individuals, who want to achieve the same things you do, and never stop learning new things.

www.it-ebooks.info

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.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.

https://www2.packtpub.com/books/subscription/packtlib

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

www.it-ebooks.info

www.it-ebooks.info

Table of Contents

Preface **1**

Chapter 1: Coding with C++ **9**

Setting up our project **9**

Using Microsoft Visual C++ on Windows 10

Using XCode on a Mac 14

Creating your first C++ program **17**

Semicolons 21

Handling errors 21

Warnings 22

What is building and compiling? **23**

Scripting 23

Exercise – ASCII art 23

Summary **24**

Chapter 2: Variables and Memory **25**

Variables **26**

Declaring variables – touching the silicon 26

Reading and writing to your reserved spot in memory 27

Numbers are everything 28

More on variables 29

Math in C++ 31

Exercises 32

Generalized variable syntax 33

Primitive types 33

Object types 34

Exercise – Player 36

Pointers 37

What can pointers do? 38

Address of operator & 39

The Null pointers 40

www.it-ebooks.info

cin 41
printf() 41
Exercise 42
Solution 42
Summary 43
Chapter 3: If, Else, and Switch 45
Branching 46
Controlling the flow of your program 46
The == operator 47
Coding if statements 47
Coding else statements 49
Testing for inequalities using other comparison operators (>, >=, <, <=, and !=) 50
Using logical operators 51
The Not (!) operator 51
Exercises 52
Solution 52
The And (&&) operator 52
The Or (||) operator 53
Our first example with Unreal Engine 53
Exercise 60
Solution 60
Branching code in more than two ways 61
The else if statement 61
Exercise 62
Solution 63
The switch statement 64
Switch versus if 66
Exercise 67
Solution 68
Summary 69
Chapter 4: Looping 71
The while loop 71
Infinite loops 73
Exercises 74
Solutions 74
The do/while loop 75
The for loop 76
Exercises 77
Solutions 78
Looping with Unreal Engine 78
Summary 80

www.it-ebooks.info

Chapter 5: Functions and Macros **81**
Functions **81**
An example of a <cmath> library function – sqrt() **83**
Writing our own functions **84**
A sample program trace 85
Exercise 88
Solution 88
Functions with arguments **88**
Functions that return values **89**
Exercises 91
Solutions 91
Variables, revisited **92**
Global variables 92
Local variables 93
The scope of a variable 94
Static local variables 96
Const variables 97
Function prototypes 97
.h and .cpp files 97
prototypes.h contains 98
funcs.cpp contains 99
main.cpp contains 99
Extern variables 100
Macros **100**
Advice – try to use const variables where possible 101
Macros with arguments **101**
Advice – use inline functions instead of macros with arguments 102
Summary **103**
Chapter 6: Objects, Classes, and Inheritance **105**
struct objects **106**
Member functions 107
The this keyword 107
Strings are objects? 108
Invoking a member function 109
Exercises 110
Solutions 110
Privates and encapsulation 111
Some people like it public 112
class versus struct **113**

www.it-ebooks.info

Getters and setters **113**
Getters 114
Setters 115
But what's the point of get/set operations? 116
Constructors and destructors **117**
Class inheritance **118**
Derived classes 118
Syntax of inheritance 122
What does inheritance do? 123
is-a relationship 123
protected variables 124
Virtual functions 124
Purely virtual functions (and abstract classes) 125
Multiple inheritance **125**
private inheritance 126
Putting your classes into headers **127**
.h and .cpp 129
Exercise 131
Summary **131**
Chapter 7: Dynamic Memory Allocation **133**
Dynamic memory allocation **135**
The delete keyword 135
Memory leaks 136
Regular arrays **137**
The array syntax 137
Exercise 138
Solutions 139
C++ style dynamic size arrays (new[] and delete[]) **139**
Dynamic C-style arrays **141**
Summary **142**
Chapter 8: Actors and Pawns **143**
Actors versus pawns **143**
Creating a world to put your actors in **144**
The UE4 editor **147**
Editor controls 147
Play mode controls 148
Adding objects to the scene 148
Starting from scratch **151**
Adding light sources 152
Collision volumes 154
Adding collision detection for the objects editor 154

www.it-ebooks.info

Adding an actor to the scene 156
Creating a player entity 156
Inheriting from UE4 GameFramework classes 156
Associating a model with the Avatar class 159
Loading the mesh 161
Creating a blueprint from our C++ class 161
Writing C++ code that controls the game's character 169
Making the player an instance of the Avatar class 169
Setting up controller inputs 172
Exercise 174
Solution 174
Yaw and pitch 176
Creating non-player character entities 177
Displaying a quote from each NPC dialog box 181
Displaying messages on the HUD 181
Using TArray<Message> 184
Exercise 186
Solution 187
Triggering an event when it is near an NPC 187
Make the NPC display something to the HUD when something is nearby 189
Exercises 190
Solutions 191
Summary 192
Chapter 9: Templates and Commonly Used Containers 193
Debugging the output in UE4 194
UE4's TArray<T> 194
An example that uses TArray<T> 195
Iterating a TArray 196
Finding whether an element is in the TArray 198
TSet<T> 199
Iterating a TSet 199
Intersecting TSet 200
Unioning TSet 200
Finding TSet 200
TMap<T, S> 200
A list of items for the player's inventory 201
Iterating a TMap 201
C++ STL versions of commonly used containers 202
C++ STL set 203
Finding an element in a <set> 204
Exercise 204
Solution 204

www.it-ebooks.info

C++ STL map 205
Finding an element in a <map> 206
Exercise 206
Solution 206
Summary **207**
Chapter 10: Inventory System and Pickup Items **209**
Declaring the backpack **209**
Forward declaration 210
Importing assets 211
Attaching an action mapping to a key 216
Base class PickupItem **217**
The root component 220
Getting the avatar 222
Getting the player controller 222
Getting the HUD 222
Drawing the player inventory **223**
Using HUD::DrawTexture() 224
Exercise 227
Detecting inventory item clicks 227
Dragging elements 228
Exercises 231
Summary **232**
Chapter 11: Monsters **233**
Landscape **234**
Sculpting the landscape 237
Monsters **238**
Basic monster intelligence 243
Moving the monster – steering behavior 243
The discrete nature of monster motion 246
Monster SightSphere 248
Monster attacks on the player **250**
Melee attacks 251
Defining a melee weapon 251
Sockets 257
Creating a skeletal mesh socket in the monster's hand 258
Attaching the sword to the model 260
Code to equip the player with a sword 261
Triggering the attack animation 263
Projectile or ranged attacks 277
Bullet physics 279
Adding bullets to the monster class 281
Player knockback 285
Summary **286**

www.it-ebooks.info

Chapter 12: Spell Book 287
The particle systems 289
Changing particle properties 291
Settings for the blizzard spell 294
Spell class actor 300
Blueprinting our spells 303
Picking up spells 305
Creating blueprints for PickupItems that cast spells 306
Attaching right mouse click to cast spell 308
Writing the avatar's CastSpell function 309
Instantiating the spell – GetWorld()->SpawnActor() 309
if(spell) 310
spell->SetCaster(this) 310
Writing AMyHUD::MouseRightClicked() 310
Activating right mouse button clicks 312
Creating other spells 314
The fire spell 314
Exercises 315
Summary 315
Index 317

www.it-ebooks.info

www.it-ebooks.info

Preface

So, you want to program your own games using Unreal Engine 4 (UE4). You have a great number of reasons to do so:

- UE4 is powerful: UE4 provides some of the most state-of-the-art, beautiful, realistic lighting and physics effects, of the kind used by AAA Studios.
- UE4 is device-agnostic: Code written for UE4 will work on Windows desktop machines, Mac desktop machines, Android devices, and iOS devices (at the time of writing this book—even more devices may be supported in the future).

So, you can use UE4 to write the main parts of your game once, and after that, deploy to iOS and Android Marketplaces without a hitch. (Of course, there will be a few hitches: iOS and Android in app purchases will have to be programmed separately.)

What is a game engine anyway?

A game engine is analogous to a car engine: the game engine is what drives the game. You will tell the engine what you want, and (using C++ code and the UE4 editor) the engine will be responsible for actually making that happen.

You will build your game around the UE4 game engine, similar to how the body and wheels are built around an actual car engine. When you ship a game with UE4, you are basically customizing the UE4 engine and retrofitting it with your own game's graphics, sounds, and code.

www.it-ebooks.info

What will using UE4 cost me?

The answer, in short, is $19 and 5 percent of sales.

"What?" you say. $19?

That's right. For only $19, you get full access to a world class AAA Engine, complete with a source. This is a great bargain, considering the fact that other engines can cost anywhere from $500 to $1,000 for just a single license.

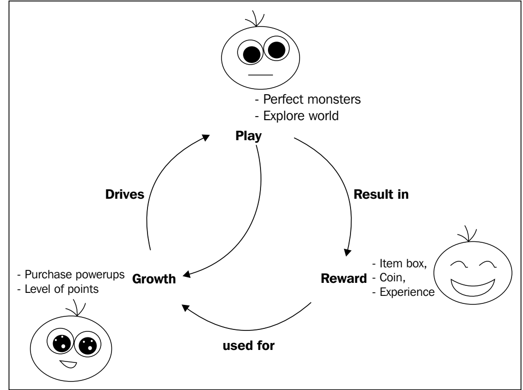
Why don't I just program my own engine and save the 5 percent?

Take it from me, if you want to create games within a reasonable time frame and you don't have a large team of dedicated engine programmers to help you, you'll want to focus your efforts on what you sell (your game).

Not having to focus on programming a game engine gives you the freedom to think only about how to make the actual game. Not having to maintain and bug-fix your own engine is a load off your mind too.

A game's overview – the Play-Reward-Growth loop

I want to show you this diagram now because it contains a core concept that many novice developers might miss when writing their first games. A game can be complete with sound effects, graphics, realistic physics, and yet, still not feel like a game. Why is that?



www.it-ebooks.info

Starting at the top of the loop, Play actions committed during the game (such as defeating a monster) result in rewards for the player (such as gold or experience). These rewards, in turn, can be used for in-game Growth (such as stats increases or new worlds to explore). This Growth then drives the gameplay in new and interesting ways. For example, a new weapon can change the basic mechanics of fighting, new spells let you take on groups of monsters with a completely different approach, or new modes of transportation can let you reach areas that were previously inaccessible.

This is the basic core loop that creates interesting gameplay. The key is that Play must result in some kind of Reward—think of glittering gold pieces popping out of nasty baddies. For rewards to have a point, it must result in some kind of Growth in the gameplay. Think about how many new locations were unlocked with the hook shot in *The Legend of Zelda*.

A game that is only Play (without Rewards or Growth) won't feel like a game: it will feel only like a really basic prototype of a game. For example, imagine a flight simulator with just an open world and no goals or objectives as well as without the ability to upgrade your plane or weapons. It wouldn't be much of a game.

A game with only Play and Rewards (but no Growth) will feel primitive and simple. The rewards will not satisfy the player if they cannot be used for anything.

A game with only Play and Growth (without Rewards) will just be seen as a mindless increasing challenge, without giving the player a sense of gratification for his achievements.

A game with all three elements will keep the player engaged with an entertaining Play. The Play has a rewarding result (loot drops and story progression), which results in the Growth of the game world. Keeping this loop in mind while you are devising your game will really help you to design a complete game.

A prototype is the proof of concept of a game. Say, you want to create your own unique version of *Blackjack*. The first thing you might do is program a prototype to show how the game will be played.

Monetization

Something you need to think about early in your game's development is your monetization strategy. How will your game make money? If you are trying to start a company, you have to think of what will be your sources of revenue from early on.

www.it-ebooks.info

Are you going to try to make money from the purchase price, such as *Jamestown*, *The Banner Saga*, *Castle Crashers*, or *Crypt of the Necrodancer*? Or, will you focus on distributing a free game with in-app purchases, such as *Clash of Clans*, *Candy Crush Saga*, or *Subway Surfers*?

A class of games for mobile devices (for example, builder games on iOS) make lots of money by allowing the user to pay in order to skip Play and jump straight to the rewards and Growth parts of the loop. The pull to do this can be very powerful; many people spend hundreds of dollars on a single game.

Why C++

UE4 is programmed in C++. To write code for UE4, you must know C++.

C++ is a common choice for game programmers because it offers very good performance combined with object-oriented programming features. It's a very powerful and flexible language.

What this book covers

Chapter 1, *Coding with C++*, talks about getting up and running with your first C++ program.

Chapter 2, *Variables and Memory*, talks about how to create, read, and write variables from computer memory.

Chapter 3, *If, Else, and Switch*, talks about branching the code: that is, allowing different sections of the code to execute, depending on program conditions.

Chapter 4, *Looping*, discusses how we repeat a specific section of code as many times as needed.

Chapter 5, *Functions and Macros*, talks about functions, which are bundles of code that can get called any number of times, as often you wish.

Chapter 6, *Objects, Classes, and Inheritance*, talks about class definitions and instantiating some objects based on a class definition.

Chapter 7, *Dynamic Memory Allocation*, discusses heap-allocated objects as well as low-level C and C++ style arrays.

Chapter 8, *Actors and Pawns*, is the first chapter where we actually delve into UE4 code. We begin by creating a game world to put actors in, and derive an `Avatar` class from a customized actor.

www.it-ebooks.info

Chapter 9, *Templates and Commonly Used Containers*, explores UE4 and the C++ STL family of collections of data, called containers. Often, a programming problem can be simplified many times by selecting the right type of container.

Chapter 10, *Inventory System and Pickup Items*, discusses the creation of an inventory system with the ability to pick up new items.

Chapter 11, *Monsters*, teaches how to create monsters that give chase to the player and attack it with weapons.

Chapter 12, *Spell Book*, teaches how to create and cast spells in our game.

What you need for this book

To work with this text, you will need two programs. The first is your integrated development environment, or IDE. The second piece of software is, of course, the Unreal Engine itself.

If you are using Microsoft Windows, then you will need Microsoft Visual Studio 2013 Express Edition for Windows Desktop. If you are using a Mac, then you will need Xcode. Unreal Engine can be downloaded from `https://www.unrealengine.com/`.

Who this book is for

This book is for anyone who wants to write an Unreal Engine application. The text begins by telling you how to compile and run your first C++ application, followed by chapters that describe the rules of the C++ programming language. After the introductory C++ chapters, you can start to build your own game application in C++.

Conventions

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

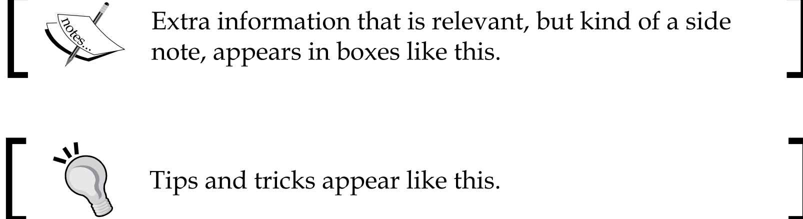
Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `variableType` is going to tell you what type of data we are going to store in our variable. The `variableName` is the symbol we'll use to read or write that piece of memory".

www.it-ebooks.info

A block of code is set as follows:

```
struct Player
{
  string name;
  int hp;
  // A member function that reduces player hp by some amount
  void damage( int amount ) {
    hp -= amount;
  }
  void recover( int amount ) {
    hp += amount;
  }
};
```

New terms and **important words** are shown in bold. Text that appears on the screen appears like this: From the **File** menu, select **New Project...**



Extra information that is relevant, but kind of a side note, appears in boxes like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

www.it-ebooks.info

Downloading the example code

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/6572OT_ColoredImages.pdf.

Errata

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

Piracy

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

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

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

www.it-ebooks.info

Questions

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

www.it-ebooks.info

1
Coding with C++

You're a first-time programmer. You have a lot to learn!

Academics often describe programming concepts in theory but like to leave implementation to someone else, preferably someone from the industry. We don't do that in this book—in this book, we will describe the theory behind C++ concepts and implement our own game as well.

The first thing I will recommend is that you do the exercises. You cannot learn to program simply by reading. You must work with the theory with the exercises.

We are going to get started by programming very simple programs in C++. I know that you want to start playing your finished game right now. However, you have to start at the beginning to get to that end (if you really want to, skip over to *Chapter 12, Spell Book,* or open some of the samples to get a feel for where we are going).

In this chapter, we will cover the following topics:

- Setting up a new project (in Visual Studio and Xcode)
- Your first C++ project
- How to handle errors
- What are building and compiling?

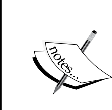
Setting up our project

Our first C++ program will be written outside of UE4. To start with, I will provide steps for both Xcode and Visual Studio 2013, but after this chapter, I will try to talk about just the C++ code without reference to whether you're using Microsoft Windows or Mac OS.

www.it-ebooks.info

Using Microsoft Visual C++ on Windows

In this section, we will install a code editor for Windows, Microsoft's Visual Studio. Please skip to the next section if you are using a Mac.



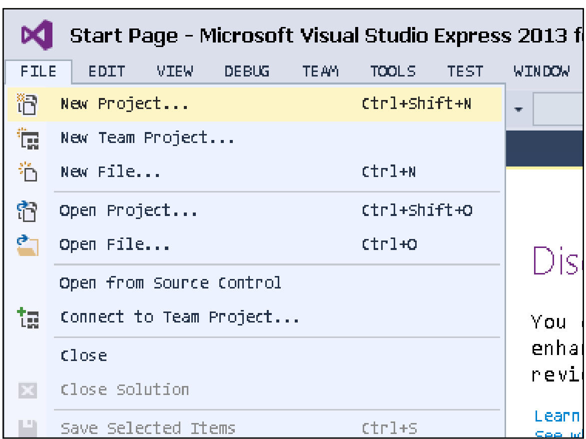
The Express edition of Visual Studio is the free version of Visual Studio that Microsoft provides on their website. Go to `http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx` to start the installation process.

To start, you have to download and install **Microsoft Visual Studio Express 2013 for Windows Desktop**. This is how the icon for the software looks:

Do not install **Express 2013 for Windows**. This is a different package and it is used for different things than what we are doing here.

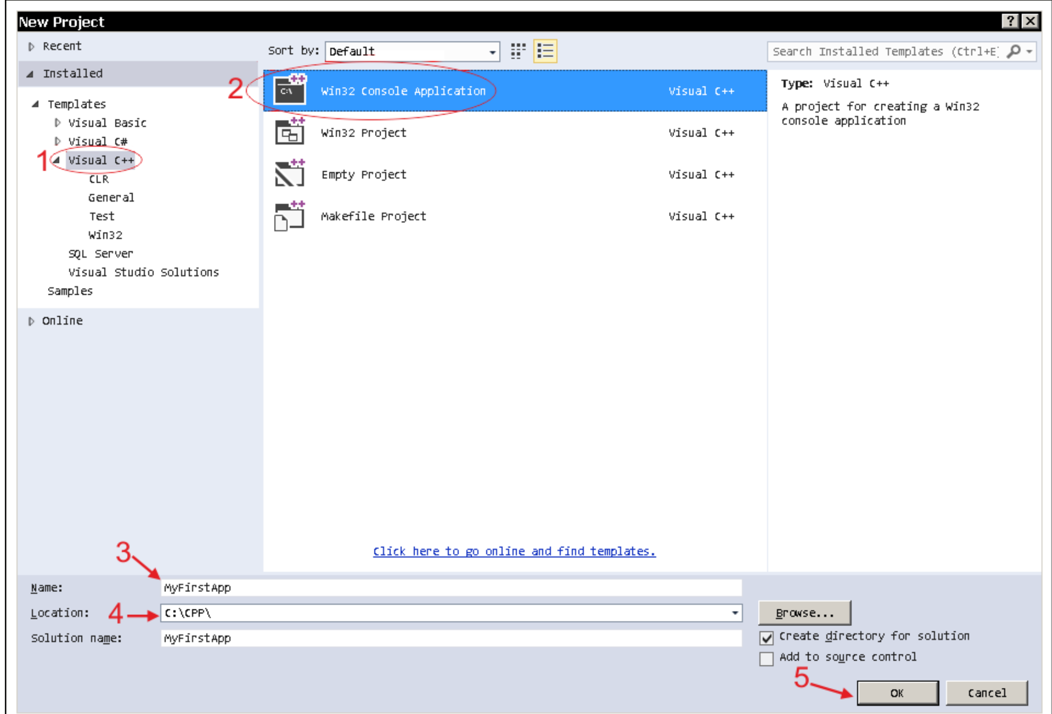
Once you have Visual Studio 2013 Express installed, open it. Work through the following steps to get to a point where you can actually type in the code:

1. From the **File** menu, select **New Project...**, as shown in the following screenshot:



www.it-ebooks.info

2. You will get the following dialog:

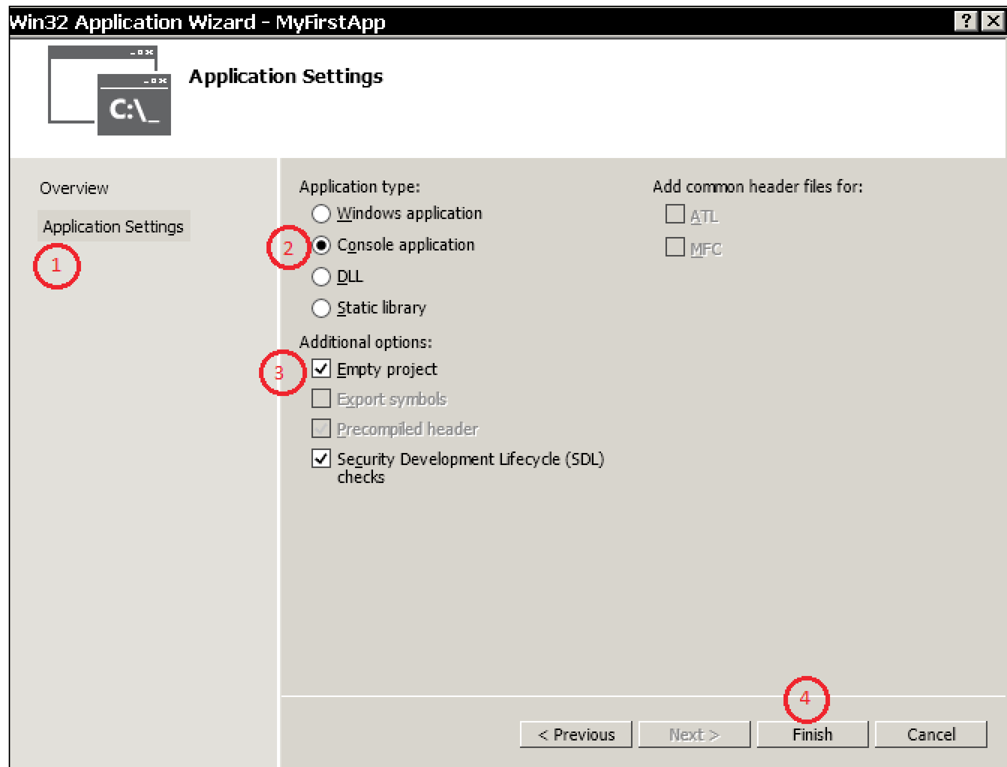


[Note that there is a small box at the bottom with the text **Solution name**. In general, **Visual Studio Solutions** might contain many projects. However, this book only works with a single project, but at times, you might find it useful to integrate many projects into the same solution.]

3. There are five things to take care of now, as follows:
   1. Select **Visual C++** from the left-hand side panel.
   2. Select **Win32 Console Application** from the right-hand side panel.
   3. Name your app (I used `MyFirstApp`).
   4. Select a folder to save your code.
   5. Click on the **OK** button.

www.it-ebooks.info

4. After this an **Application Wizard** dialog box opens up, as shown in the following screenshot:

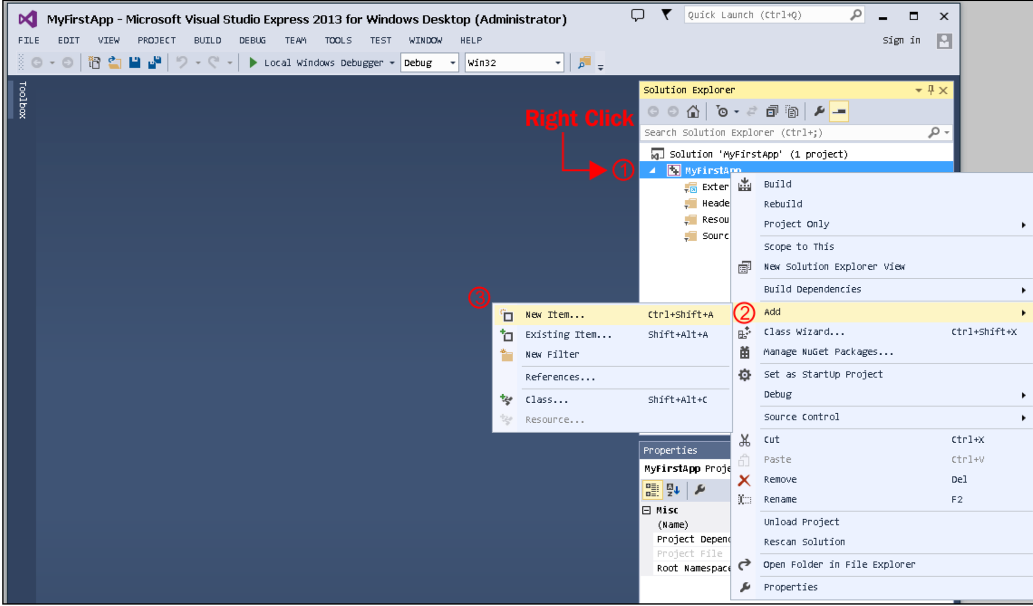


5. We have four things to take care of in this dialog box, as follows:
   1. Click on **Application Settings** in the left-hand side panel.
   2. Ensure that **Console application** is selected.
   3. Select **Empty project**.
   4. Click on **Finish**.

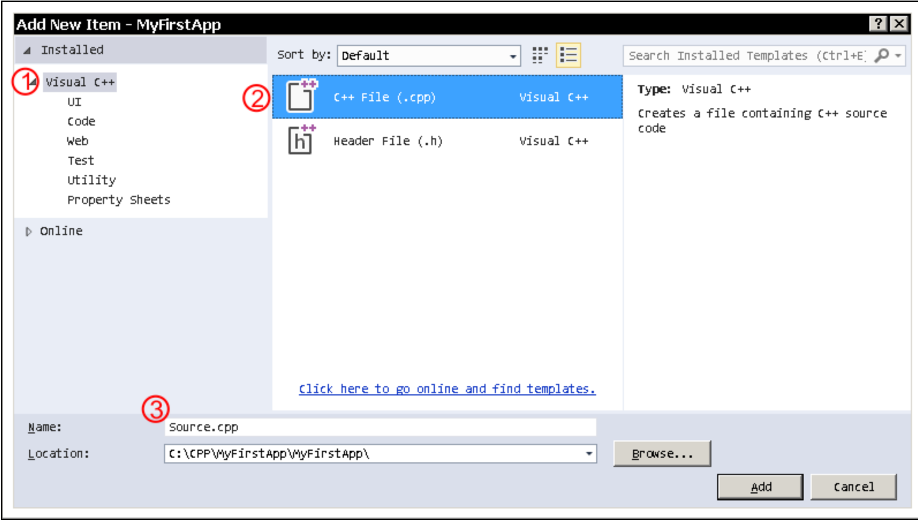
Now you are in the Visual Studio 2013 environment. This is the place where you will do all your work and code.

www.it-ebooks.info

However, we need a file to write our code into. So, we will add a C++ code file to our project, as shown in the following screenshot:



Add your new source code file as shown in the following screenshot:



You will now edit `Source.cpp`. Skip to the Your First C++ Program section and type in your code.

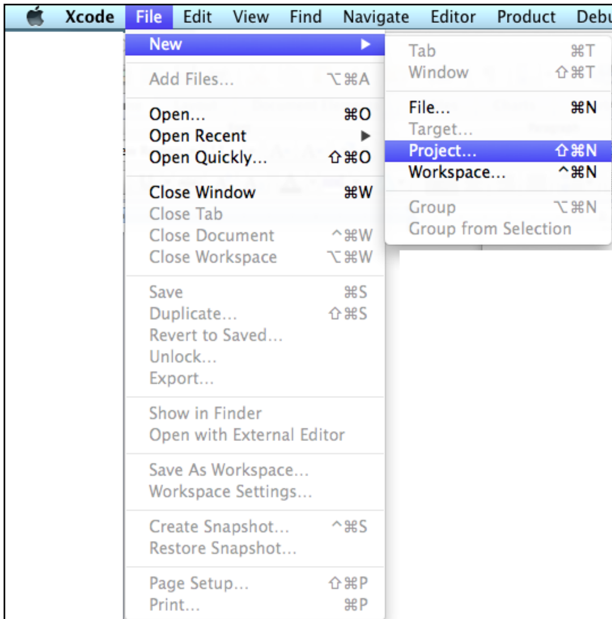
www.it-ebooks.info

Using XCode on a Mac

In this section, we will talk about how to install Xcode on a Mac. Please skip to the next section if you are using Windows.

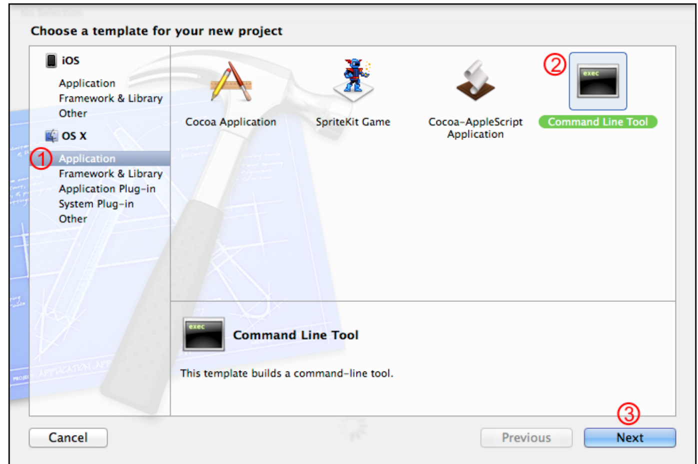
Xcode is available on all Mac machines. You can get Xcode using the Apple App Store (it's free), as shown here:

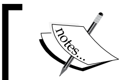
1. Once you have Xcode installed, open it. Then, navigate to **File** | **New** | **Project...** from the system's menu bar at the top of your screen, as shown in the following screenshot:



www.it-ebooks.info

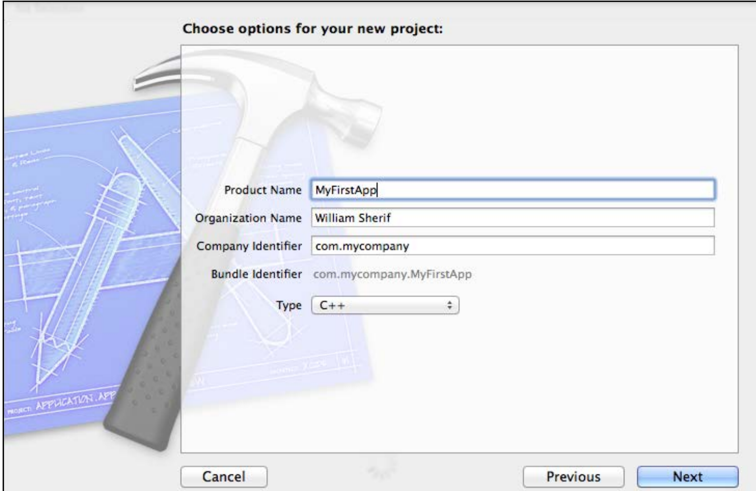
2. In the New Project dialog, select **Application** under **OS X** on the left-hand side of the screen, and select **Command Line Tool** from the right-hand side pane. Then, click on **Next**:





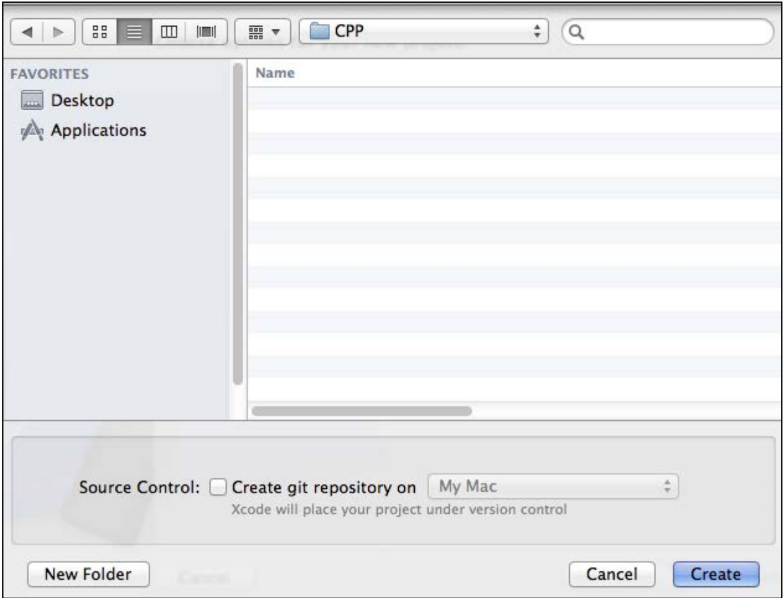
You might be tempted to click on the **SpriteKit Game** icon, but don't click on it.

3. In the next dialog, name your project. Be sure to fill in all the fields or Xcode won't let you proceed. Make sure that the project's **Type** is set to **C++** and then click on the **Next** button, as shown here:



www.it-ebooks.info

4. The next popup will ask you to choose a location in order to save your project. Pick a spot on your hard drive and save it there. Xcode, by default, creates a Git repository for every project you create. You can uncheck **Create git repository** —we won't cover Git in this chapter—as shown in the following screenshot:



Git is a **Version control system**. This basically means that Git keeps the snapshots of all the code in your project every so often (every time you *commit* to the repository). Other popular **source control management** tools (**scm**) are Mercurial, Perforce, and Subversion. When multiple people are collaborating on the same project, the scm tool has the ability to automatically merge and copy other people's changes from the repository to your local code base.

www.it-ebooks.info

Okay! You are all set up. Click on the **main.cpp** file in the left-hand side panel of Xcode. If the file doesn't appear, ensure that the folder icon at the top of the left-hand side panel is selected first, as shown in the following screenshot:

Creating your first C++ program

We are now going to write some C++ source code. There is a very good reason why we are calling it the source code: it is the source from which we will build our binary executable code. The same C++ source code can be built on different platforms such as Mac, Windows, and iOS, and in theory, an executable code doing the exact same things on each respective platform should result.

In the not-so-distant past, before the introduction of C and C++, programmers wrote code for each specific machine they were targeting individually. They wrote code in a language called assembly language. But now, with C and C++ available, a programmer only has to write code once, and it can be deployed to a number of different machines simply by sending the same code through different compilers.

In practice, there are some differences between Visual Studio's flavor of C++ and Xcode's flavor of C++, but these differences mostly come up when working with advanced C++ concepts, such as templates.

One of the main reasons why using UE4 is so helpful is that UE4 will erase a lot of the differences between Windows and Mac. The UE4 team did a lot of magic in order to get the same code to work on both Windows and Mac.

www.it-ebooks.info

A real-world tip

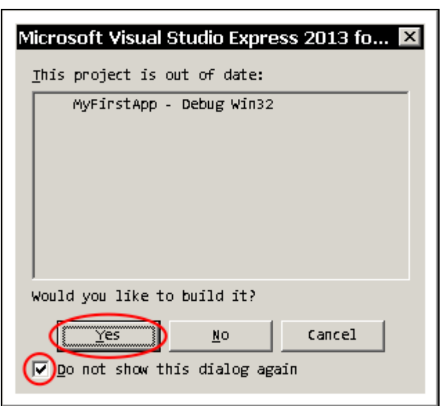
It is important for the code to run in the same way on all machines, especially for networked games or games that allow things such as shareable replays. This can be achieved using standards. For example, the IEEE floating-point standard is used to implement decimal math on all C++ compilers. This means that the result of computations such as `200 * 3.14159` should be the same on all the machines.

Write the following code in Microsoft Visual Studio or in Xcode:

```
#include <iostream>  // Import the input-output library
using namespace std; // allows us to write cout
                     // instead of std::cout
int main()
{
  cout << "Hello, world" << endl;
  cout << "I am now a C++ programmer." << endl;
  return 0;      // "return" to the operating sys
}
```

Press *Ctrl* + *F5* to run the preceding code in Visual Studio, or press ⌘ + *R* to run in Xcode.

The first time you press *Ctrl* + *F5* in Visual Studio, you will see this dialog:



Select **Yes** and **Do not show this dialog again**—trust me, this will avoid future problems.

www.it-ebooks.info

The first thing that might come to your mind is, "My! A whole lot of gibberish!"

Indeed, you rarely see the use of the hash (#) symbol (unless you use Twitter) and curly brace pairs { } in normal English texts. However, in C++ code, these strange symbols abound. You just have to get used to them.

So, let's interpret this program, starting from the first line.

This is the first line of the program:

```
#include <iostream>  // Import the input-output library
```

This line has two important points to be noted:

1. The first thing we see is an `#include` statement. We are asking C++ to copy and paste the contents of another C++ source file, called `<iostream>`, directly into our code file. The `<iostream>` is a standard C++ library that handles all the sticky code that lets us print text to the screen.
2. The second thing we notice is a `//` comment. C++ ignores any text after a double slash (`//`) until the end of that line. Comments are very useful to add in plain text explanations of what some code does. You might also see `/* */` C-style comments in the source. Surrounding any text in C or C++ with slash-star `/*` and star-slash `*/` gives an instruction to have that code removed by the compiler.

This is the next line of code:

```
using namespace std; // allows us to write cout
                     // instead of std::cout
```

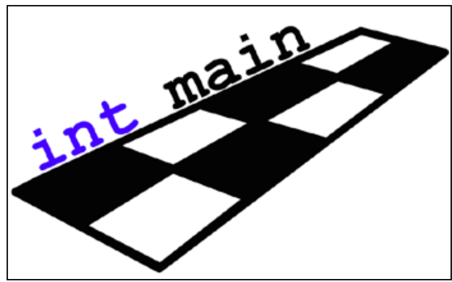
The comments beside this line explain what the `using` statement does: it just lets you use a shorthand (for example, `cout`) instead of the fully qualified name (which, in this case, would be `std::cout`) for a lot of our C++ code commands. Some people don't like a `using namespace std;` statement; they prefer to write the `std::cout` longhand every time they want to use `cout`. You can get into long arguments over things like this. In this section of the text, we prefer the brevity that we get with the `using namespace std;` statement.

This is the next line:

```
int main()
```


www.it-ebooks.info

This is the application's starting point. You can think of `main` as the start line in a race. The `int main()` statement is how your C++ program knows where to start; take a look at the following figure:



If you don't have an `int main()` program marker or if `main` is spelled incorrectly, then your program just won't work because the program won't know where to start.

The next line is a character you don't see often:

```
{
```

This { character is not a sideways mustache. It is called a curly brace, and it denotes the starting point of your program.

The next two lines print text to the screen:

```
cout << "Hello, world" << endl;
cout << "I am now a C++ programmer." << endl;
```

The `cout` statement stands for console output. Text between double quotes will get an output to the console exactly as it appears between the quotes. You can write anything you want between double quotes except a double quote and it will still be valid code.

> To enter a double quote between double quotes, you need to stick a backslash (\) in front of the double quote character that you want inside the string, as shown here:
>
> ```
> cout << "John shouted into the cave \"Hello!\" The
> cave echoed."
> ```
>
> The \ symbol is an example of an escape sequence. There are other escape sequences that you can use; the most common escape sequence you will find is `\n`, which is used to jump the text output to the next line.

www.it-ebooks.info

The last line of the program is the `return` statement:

```
return 0;
```

This line of code indicates that the C++ program is quitting. You can think of the `return` statement as returning to the operating system.

Finally, the end of your program is denoted by the closing curly brace, which is an opposite-facing sideways mustache:

```
}
```

Semicolons

Semicolons (;) are important in C++ programming. Notice in the preceding code example that most lines of code end in a semicolon. If you don't end each line with a semicolon, your code will not compile, and if that happens, you can be fired from your job.

Handling errors

If you make a mistake while entering code, then you will have a syntax error. In the face of syntax errors, C++ will scream murder and your program will not even compile; also, it will not run.

Let's try to insert a couple of errors into our C++ code from earlier:

```
#include·<iostreams>¶
using·namespace·std;¶
int·maind(¶
{¶
··count·<<·"Hello,·world·<<·ender:¶
··count·<<·"I·am·nowb·a·C++·programmer."·<<·ender;¶

}¶
```

Warning! This code listing contains errors. It is a good exercise to find all the errors and fix them!

As an exercise, try to find and fix all the errors in this program.

Note that if you are extremely new to C++, this might be a hard exercise. However, this will show you how careful you need to be when writing C++ code.

www.it-ebooks.info

Fixing compilation errors can be a nasty business. However, if you input the text of this program into your code editor and try to compile it, it will cause the compiler to report all the errors to you. Fix the errors, one at a time, and then try to recompile. A new error will pop up or the program will just work, as shown in the following screenshot:

```
#include <iostreams>                    'iostreams' file not found
using namespace std;

int maind(
{
   count << "Hello, world << ender:
   count << "I am nowb a C++ programmer." << ender;
}
```

Xcode shows you the errors in your code when you try to compile it

The reason I am showing you this sample program is to encourage the following workflow as long as you are new to C++:

1. Always start with a working C++ code example. You can fork off a bunch of new C++ programs from the *Your First C++ Program* section.
2. Make your code modifications in small steps. When you are new, compile after writing each new line of code. Do not code for one to two hours and then compile all that new code at once.
3. You can expect it to be a couple of months before you can write code that performs as expected the first time you write it. Don't get discouraged. Learning to code is fun.

Warnings

The compiler will flag things that it thinks might be mistakes. These are another class of compiler notices known as warnings. Warnings are problems in your code that you do not have to fix for your code to run but are simply recommended to be fixed by the compiler. Warnings are often indications of code that is not quite perfect, and fixing warnings in code is generally considered good practice.

However, not all warnings are going to cause problems in your code. Some programmers prefer to disable the warnings that they do not consider to be an issue (for example, warning 4018 warns against signed/unsigned mismatch, which you will most likely see later).

www.it-ebooks.info

What is building and compiling?

You might have heard of a computer process term called compiling. Compiling is the process of converting your C++ program into code that can run on a CPU. Building your source code means the same thing as compiling it.

See, your source `code.cpp` file will not actually run on a computer. It has to be compiled first for it to run.

This is the whole point of using Microsoft Visual Studio Express or Xcode. Visual Studio and Xcode are both compilers. You can write C++ source code in any text-editing program—even in Notepad. But you need a compiler to run it on your machine.

Every operating system typically has one or more C++ compilers that can compile C++ code to run on that platform. On Windows, you have Visual Studio and Intel C++ Studio compiler. On Mac, there is Xcode, and on all of Windows, Mac, and Linux, there is the **GNU Compiler Collection** (**GCC**).

The same C++ code that we write (Source) can be compiled using different compilers for different operating systems, and in theory, they should produce the same result. The ability to compile the same code on different platforms is called portability. In general, portability is a good thing.

Scripting

There is another class of programming languages called scripting languages. These include languages such as PHP, Python, and ActionScript. Scripted languages are not compiled—for JavaScript, PHP, and ActionScript, there is no compilation step. Rather, they are interpreted from the source as the program is run. The good thing about scripting languages is that they are usually platform-independent from the first go, because interpreters are very carefully designed to be platform-independent.

Exercise – ASCII art

Game programmers love ASCII art. You can draw a picture using only characters. Here's an example of an ASCII art maze:

```
cout << "****************" << endl;
cout << "*............*.*" << endl;
cout << "*.*.*******..*.*" << endl;
cout << "*.*.*..........*" << endl;
cout << "*.*.*.**********" << endl;
cout << "***.***........*" << endl;
```

Construct your own maze in C++ code or draw a picture using characters.

www.it-ebooks.info

Summary

To sum it up, we learned how to write our first program in the C++ programming language in our integrated development environment (IDE, Visual Studio, or Xcode). This was a simple program, but you should count getting your first program to compile and run as your first victory. In the upcoming chapters, we'll put together more complex programs and start using Unreal Engine for our games.

```
1  #include <iostream>  // Import the input-output library
2  using namespace std; // allows us to write cout
3                       // instead of std::cout
4
5  int main()
6  {
7      cout << "Hello, world" << endl;
8      cout << "I am now a C++ programmer." << endl;
9      return 0;
10 }
11
```

The preceding screenshot is of your first C++ program and the following screenshot is of its output, your first victory:

```
C:\Windows\system32\cm
Hello, world
I am now a C++ programmer.
Press any key to continue . . .
```


www.it-ebooks.info

2
Variables and Memory

To write your C++ game program, you will need your computer to remember a lot of things. Things such as where in the world is the player, how many hit points he has, how much ammunition he has left, where the items are in the world, what power-ups they provide, and the letters that make up the player's screen name.

The computer that you have actually has a sort of electronic sketchpad inside it called *memory,* or RAM. Physically, computer memory is made out of silicon and it looks something similar to what is shown in the following screenshot:

Does this RAM look like a parking garage? Because that's the metaphor we're going to use

RAM is short for Random Access Memory. It is called random access because you can access any part of it at any time. If you still have some CDs lying around, they are an example of non-random access. CDs are meant to be read and played back in order. I still remember jumping tracks on Michael Jackson's *Dangerous* album way back when switching tracks on a CD took a lot of time! Hopping around and accessing different cells of RAM, however, doesn't take much time at all. RAM is a type of fast memory access known as flash memory.

RAM is called volatile flash memory because when the computer is shut down, RAM's contents are cleared and the old contents of RAM are lost unless they were saved to the hard disk first.

www.it-ebooks.info

For permanent storage, you have to save your data into a hard disk. There are two main types of hard disks, platter-based **Hard Disk Drives** (**HDDs**) and **Solid-state Drives** (**SSDs**). SSDs are more modern than platter-based HDDs, since they use RAM's fast-access (Flash) memory principle. Unlike RAM, however, the data on an SSD persists after the computer is shut down. If you can get an SSD, I'd highly recommend that you use it! Platter-based drives are outdated. We need a way to reserve a space on the RAM and read and write from it. Fortunately, C++ makes this easy.

Variables

A saved location in computer memory that we can read or write to is called a *variable*.

A variable is a component whose value can vary. In a computer program, you can think of a variable as a container, into which you can store some data. In C++, these data containers (variables) have types. You have to use the right type of data container to save your data in your program.

If you want to save an integer, such as 1, 0, or 20, you will use an `int` type container. You can use float-type containers to carry around floating-point (decimal) values, such as 38.87, and you can use string variables to carry around strings of letters (think of it as a "string of pearls", where each letter is a pearl).

You can think of your reserved spot in RAM like reserving parking space in a parking garage: once we declare our variable and get a spot for it, no one else (not even other programs running on the same machine) will be given that piece of RAM by the operating system. The RAM beside your variable might be unused or it might be used by other programs.

The operating system exists to keep programs from stepping on each other's toes and accessing the same bits of computer hardware at the same time. In general, civil computer programs should not read or write to each other's memory. However, some types of cheat programs (for example, maphacks) secretly access your program's memory. Programs such as PunkBuster were introduced to prevent cheating in online games.

Declaring variables – touching the silicon

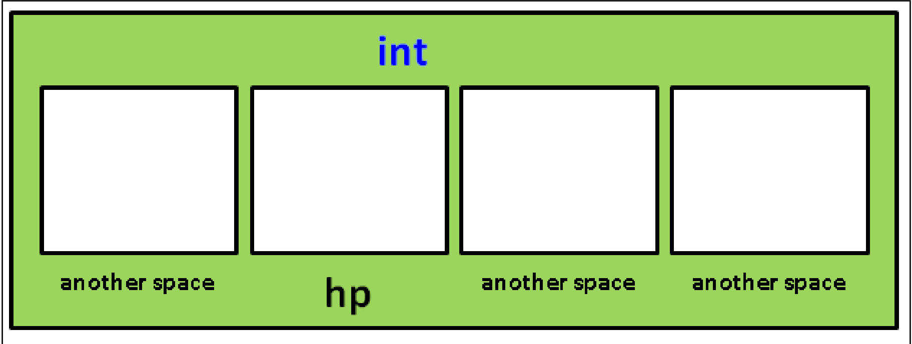
Reserving a spot on computer memory using C++ is easy. We'll want to name our chunk of memory that we will store our data in with a good, descriptive name.

www.it-ebooks.info

For example, say, we know that player **hit points** (**hp**) will be an integer (whole) number, such as 1, 2, 3, or 100. To get a piece of silicon to store the player's hp in memory, we will declare the following line of code:

```
int hp;     // declare variable to store the player's hp
```

This line of code reserves a small chunk of RAM to store an integer (`int` is short for integer), called hp. The following is an example of our chunk of RAM used to store the player's hp. This reserves a parking space for us in memory (among all the other parking spaces), and we can refer to this space in memory by its label (hp).



Among all the other spaces in memory, we get one spot to store our hp data

Notice how the variable space is type-marked in this diagram as **int**: if it is a space for a double or a different type of variable. C++ remembers the spaces that you reserve for your program in memory not only by name but by the type of variable it is as well.

Notice that we haven't put anything in hp's box yet! We'll do that later—right now, the value of the hp variable is not set, so it will have the value that was left in that parking space by the previous occupant (the value left behind by another program, perhaps). Telling C++ the type of the variable is important! Later, we will declare a variable to store decimal values, such as 3.75.

Reading and writing to your reserved spot in memory

Writing a value into memory is easy! Once you have an `hp` variable, you just write to it using the `=` sign:

```
hp = 500;
```


www.it-ebooks.info

Voila! The player has 500 hp.

Reading the variable is equally simple. To print out the value of the variable, simply put this:

```
cout << hp << endl;
```

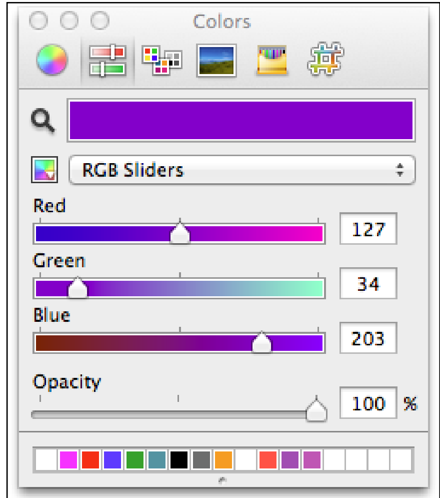
This will print the value stored inside the hp variable. If you change the value of hp, and then use `cout` again, the most up-to-date value will be printed, as shown here:

```
hp = 1200;
cout << hp << endl; // now shows 1200
```

Numbers are everything

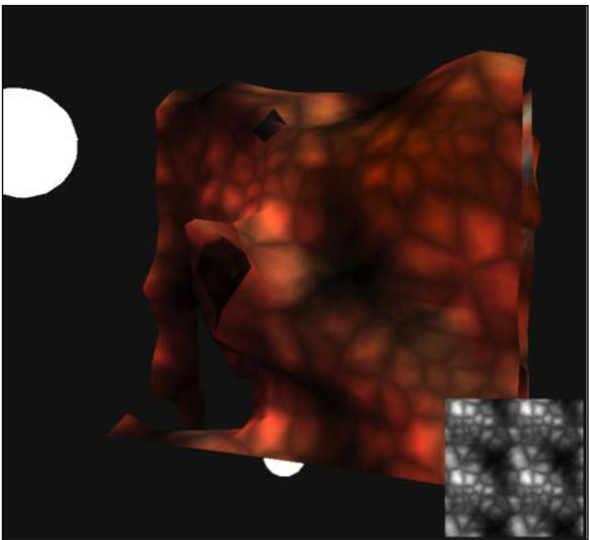
Something that you need to get used to when you start computer programming is that a surprising number of things can be stored in computer memory as just numbers. A player's hp? As we just saw in the previous section, hp can just be an integer number. If the player gets hurt, we reduce this number. If the player gains health, we increase the number.

Colors can be stored as numbers too! If you've used standard image editing programs, there are usually sliders that indicate color as how much red, green, and blue are being used, such as Pixelmator's color sliders. A color is then represented by three numbers. The purple color shown in the following screenshot is (R=127, G=34, B=203):



www.it-ebooks.info

What about world geometry? These are also just numbers: all we have to do is store a list of 3D space points (x, y, and z coordinates) and then store another list of points that explain how those points can be connected to form triangles. In the following screenshot, we can see how 3D space points are used to represent world geometry:



The combination of numbers for colors and numbers for 3D space points will let you draw large and colored landscapes in your game world.

The trick with the preceding examples is how we interpret the stored numbers so that we can make them mean what we want them to mean.

More on variables

You can think of variables as animal-carrying cases. A cat carrier can be used to carry a cat, but not a dog. Similarly, you should use a float-type variable to carry decimal-valued numbers. If you store a decimal value inside an `int` variable, it will not fit:

```
int x = 38.87f;
cout << x << endl; // prints 38, not 38.87
```

What's really happening here is that C++ does an automatic type conversion on 38.87, *transmogrifying* it to an integer to fit in the `int` carrying case. It drops the decimal to convert 38.87 into the integer value 38.

www.it-ebooks.info

So, for example, we can modify the code to include the use of three types of variables, as shown in the following code:

```
#include <iostream>
#include <string>  // need this to use string variables!
using namespace std;
int main()
{
  string name;
  int goldPieces;
  float hp;
  name = "William"; // That's my name
  goldPieces = 322; // start with this much gold
  hp = 75.5f;       // hit points are decimal valued
  cout << "Character " << name << " has "
           << hp << " hp and "
           << goldPieces << " gold.";
}
```

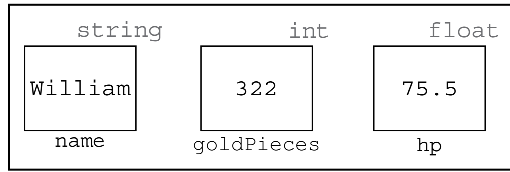
In the first three lines, we declare three boxes to store our data parts into, as shown here:

```
string name;
int goldPieces;
float hp;
```

These three lines reserve three spots in memory (like parking spaces). The next three lines fill the variables with the values we desire, as follows:

```
name = "William";
goldPieces = 322;
hp = 75.5f;
```

In computer memory, this will look as shown in the following figure:



You can change the contents of a variable at any time. You can write a variable using the `=` assignment operator, as follows:

```
goldPieces = 522;// = is called the "assignment operator"
```


www.it-ebooks.info

You can also read the contents of a variable at any time. That's what the next three lines of code do, as shown here:

```
cout << "Character " << name << " has "
     << hp << " hp and "
     << goldPieces << " gold.";
```

Take a look at this line:

```
cout << "I have " << hp << " hp." << endl;
```

There are two uses of the word `hp` in this line. One is between double quotes, while the other is not. Words between double quotes are always output exactly as you typed them. When double quotes are not used (for example, `<< hp <`), a variable lookup is performed. If the variable does not exist, then you will get a compiler error (undeclared identifier).

There is a space in memory that is allocated for the name, a space for how many `goldPieces` the player has, and a space for the hp of the player.

[In general, you should always try to store the right type of data inside the right type of variable. If you happen to store the wrong type of data, your code may misbehave.]

Math in C++

Math in C++ is easy to do; + (plus), - (minus), * (times), / (divide by) are all common C++ operations, and proper BEDMAS order will be followed (Brackets, Exponents, Division, Multiplication, Addition, and Subtraction). For example, we can do as shown in the following code:

```
int answer = 277 + 5 * 4 / 2 + 20;
```

Another operator that you might not be familiar with yet is % (modulus). Modulus (for example, 10 % 3) finds the remainder of when `x` is divided by `y`. See the following table for examples:

Operator (name)	**Example**	**Answer**
+ (plus)	7 + 3	10
- (minus)	8 - 5	3
* (times)	5*6	30
/ (division)	12/6	2
% (modulus)	10 % 3	1 (because 10/3 is 3 the remainder = 1).

www.it-ebooks.info

However, we often don't want to do math in this manner. Instead, we usually want to change the value of a variable by a certain computed amount. This is a concept that is harder to understand. Say the player encounters an imp and is dealt 15 damage.

The following line of code will be used to reduce the player's hp by 15 (believe it or not):

```
hp = hp - 15;                    // probably confusing :)
```

You might ask why. Because on the right-hand side, we are computing a new value for hp (`hp-15`). After the new value for hp is found (15 less than what it was before), the new value is written into the hp variable.

Pitfall

An uninitialized variable has the bit pattern that was held in memory for it before. Declaring a variable does not clear the memory. So, say we used the following line of code:

```
int hp;
hp = hp - 15;
```

The second line of code reduces the hp by 15 from its previous value. What was its previous value if we never set hp = 100 or so? It could be 0, but not always.

One of the most common errors is to proceed with using a variable without initializing it first.

The following is a shorthand syntax for doing this:

```
hp -= 15;
```

Besides `-=`, you can use += to add some amount to a variable, *= to multiply a variable by an amount, and /= to divide a variable by some amount.

Exercises

Write down the value of `x` after performing the following operations; then, check with your compiler:

Exercises	Solutions
`int x = 4; x += 4;`	8
`int x = 9; x-=2;`	7
`int x = 900; x/=2;`	450
`int x = 50; x*=2;`	100

www.it-ebooks.info

Exercises	Solutions
`int x = 1; x += 1;`	2
`int x = 2; x -= 200;`	-198
`int x = 5; x*=5;`	25

Generalized variable syntax

In the previous section, you learned that every piece of data that you save in C++ has a type. All variables are created in the same way; in C++, variable declarations are of the form:

```
variableType variableName;
```

The `variableType` tells you what type of data we are going to store in our variable. The `variableName` is the symbol we'll use to read or write to that piece of memory.

Primitive types

We previously talked about how all the data inside a computer will at some point be a number. Your computer code is responsible for interpreting that number correctly.

It is said that C++ only defines a few basic data types, as shown in the following table:

`Char`	A single letter, such as 'a', 'b', or '+'
`Short`	An integer from -32,767 to +32,768
`Int`	An integer from -2,147,483,647 to +2,147,483,648
`Float`	Any decimal value from approx. -1x1038 to 1x1038
`Double`	Any decimal value from approx. -1x10308 to 1x10308
`Bool`	true or false

There are unsigned versions of each of the variable types mentioned in the preceding table. An unsigned variable can contain natural numbers, including 0 (x >= 0). An unsigned `short`, for example, might have a value between 0 and 65535.

If you're further interested in the difference between float and double, please feel free to look it up on the Internet. I will keep my explanations only to the most important C++ concepts used for games. If you are curious about something that's covered by this text, feel free to look it up.

www.it-ebooks.info

It turns out that these simple data types alone can be used to construct arbitrarily complex programs. "How?" you ask. Isn't it hard to build a 3D game using just floats and integers?

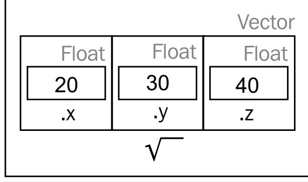
It is not really difficult to build a game from float and int, but more complex data types help. It will be tedious and messy to program if we used loose floats for the player's position.

Object types

C++ gives you structures to group variables together, which will make your life a lot easier. Take an example of the following block of code:

```
#include <iostream>
using namespace std;
struct Vector        // BEGIN Vector OBJECT DEFINITION
{
  float x, y, z;     // x, y and z positions all floats
};                   // END Vector OBJECT DEFINITION.
// The computer now knows what a Vector is
// So we can create one.
int main()
{
  Vector v; // Create a Vector instance called v
  v.x=20, v.y=30, v.z=40; // assign some values
  cout << "A 3-space vector at " << v.x << ", " << v.y << ", " <<
  v.z << endl;
}
```

The way this looks in memory is pretty intuitive; a Vector is just a chunk of memory with three floats, as shown in the following figure:



Don't confuse the struct Vector in the preceding screenshot with the `std::vector` of the STL. The Vector object above is meant to represent a three-space vector, while STL's `std::vector` type represents a sized collection of values.

www.it-ebooks.info

Here are a couple of review notes about the preceding code listing:

First, even before we use our Vector object type, we have to define it. C++ does not come with built-in types for math vectors (it only supports scalar numbers, and they thought that was enough!). So, C++ lets you build your own object constructions to make your life easier. We first had the following definition:

```
struct Vector          // BEGIN Vector OBJECT DEFINITION
{
  float x, y, z;       // x, y, and z positions all floats
};                     // END Vector OBJECT DEFINITION.
```

This it tells the computer what a Vector is (it's 3 floats, all of which are declared to be sitting next to each other in the memory). The way a Vector will look in the memory is shown in preceding figure.

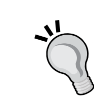
Next, we use our Vector object definition to create a Vector instance called `v`:

```
Vector v; // Create a Vector instance called v
```

The `struct` Vector definition doesn't actually create a Vector object. You can't do `Vector.x = 1`. "Which object instance are you talking about?" the C++ compiler will ask. You need to create a Vector instance first, such as Vector v1; then, you can do assignments on the v1 instance, such as v1.x = 0.

We then use this instance to write values into `v`:

```
v.x=20, v.y=30, v.z=40; // assign some values
```



We used commas in the preceding code to initialize a bunch of variables on the same line. This is okay in C++. Although you can do each variable on its own line, the approach shown here is okay too.

This makes `v` look as in the preceding screenshot. Then, we print them out:

```
cout << "A 3-space vector at " << v.x << ", " << v.y << ", " <<
  v.z << endl;
```

In both the lines of code here, we access the individual data members inside the object by simply using a dot (`.`). `v.x` refers to the `x` member inside the object `v`. Each Vector object will have exactly three floats inside it: one called `x`, one called `y`, and one called `z`.

www.it-ebooks.info

Exercise – Player

Define a C++ data struct for a Player object. Then, create an instance of your Player class and fill each of the data members with values.

Solution

Let's declare our Player object. We want to group together everything to do with the player into the Player object. We do this so that the code is neat and tidy. The code you read in Unreal Engine will use objects such as these everywhere; so, pay attention:

```
struct Player
{
  string name;
  int hp;
  Vector position;
}; // Don't forget this semicolon at the end!
int main()
{
  // create an object of type Player,
  Player me; // instance named 'me'
  me.name = "William";
  me.hp = 100.0f;
  me.position.x = me.position.y = me.position.z=0;
}
```

The struct Player definition is what tells the computer how a Player object is laid out in memory.

I hope you noticed the mandatory semicolon at the end of the struct declaration. struct object declarations need to have a semicolon at the end, but functions do not. This is just a C++ rule that you must remember.

Inside a Player object, we declared a string for the player's name, a float for his hp, and a Vector object for his complete xyz position.

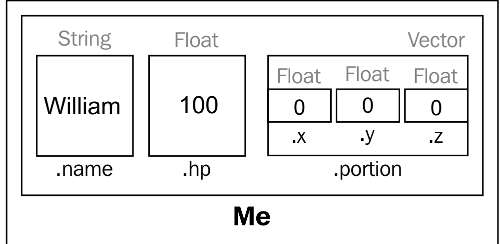
When I say object, I mean a C++ struct (or later, we will introduce the term class).

Wait! We put a Vector object inside a Player object! Yes, you can do that.

After the definition of what a Player object has inside it, we actually create a Player object instance called me and assign it some values.

www.it-ebooks.info

After the assignment, the me object looks as shown in the following figure:



Pointers

A particularly tricky concept to grasp is the concept of pointers. Pointers aren't that hard to understand but can take a while to get a firm handle on.

Say we have, as before, declared a variable of the type Player in memory:

```
Player me;
me.name = "William";
me.hp = 100.0f;
```

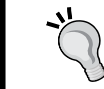
We now declare a pointer to the Player:

```
Player* ptrMe;              // Declaring a pointer
```

The `*` characters usually make things special. In this case, the `*` makes `ptrMe` special. The `*` is what makes `ptrMe` a pointer type.

We now want to link `ptrMe` to me:

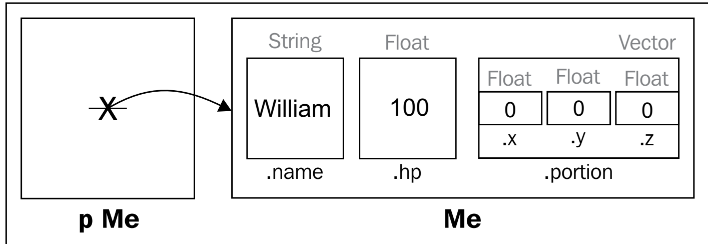
```
ptrMe = &me;                // LINKAGE
```



> This linkage step is very important. If you don't link the pointer to an object before you use the pointer, you will get a memory access violation.

www.it-ebooks.info

`ptrMe` now refers to the same object as me. Changing `ptrMe` will change me, as shown in the following figure:



What can pointers do?

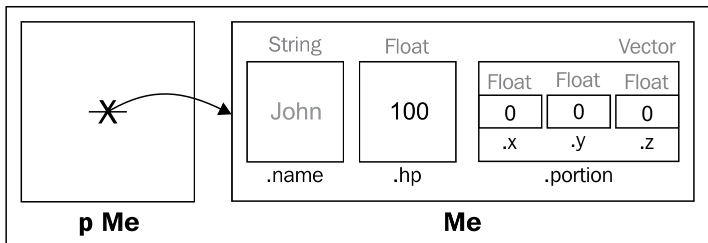
When we set up the linkage between the pointer variable and what it is pointing to, we can manipulate the variable that is pointed to through the pointer.

One use of pointers is to refer to the same object from several different locations of the code. The Player object is a good candidate for being pointed to. You can create as many pointers as you wish to the same object. Objects that are pointed to do not necessarily know that they are being pointed at, but changes can be made to the object through the pointers.

For instance, say the player got attacked. A reduction in his hp will result, and this reduction will be done using the pointer, as shown in the following code:

```
ptrMe->hp -= 33;      // reduced the player's hp by 33
ptrMe->name = "John";// changed his name to John
```

Here's how the Player object looks now:



www.it-ebooks.info

So, we changed `me.name` by changing `ptrMe->name`. Because `ptrMe` points to me, changes through `ptrMe` affect me directly.

Besides the funky arrow syntax (use `->` when the variable is a pointer), this concept isn't all that hard to understand.

Address of operator &

Notice the use of the `&` symbol in the preceding code example. The `&` operator gets the memory address of a variable. A variable's memory address is where it lives in the computer memory space. C++ is able to get the memory address of any object in your program's memory. The address of a variable is unique and also, kind of, random.

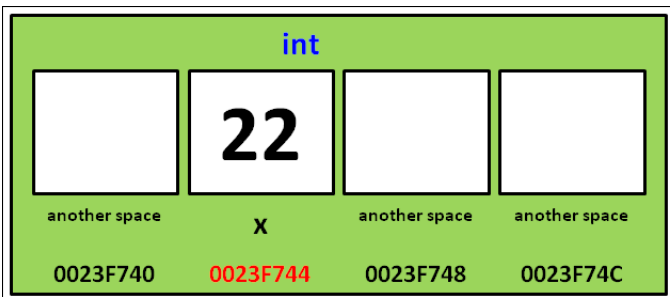
Say, we print the address of an integer variable `x`, as follows:

```
int x = 22;
cout << &x << endl; // print the address of x
```

On the first run of the program, my computer prints the following:

```
0023F744
```

This number (the value of `&x`) is just the memory cell where the variable `x` is stored. What this means is that in this particular launch of the program, the variable `x` is located at memory cell number 0023F744, as shown in the following figure:

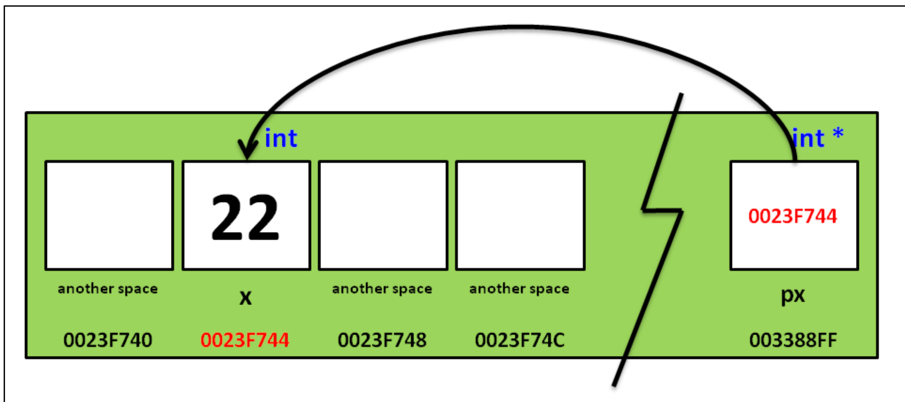


Now, create and assign a pointer variable to the address of `x`:

```
int *px;
px = &x;
```


www.it-ebooks.info

What we're doing here is storing the memory address of `x` inside the variable `px`. So, we are metaphorically pointing to the variable `x` using another different variable called `px`. This might look something similar to what is shown in the following figure:

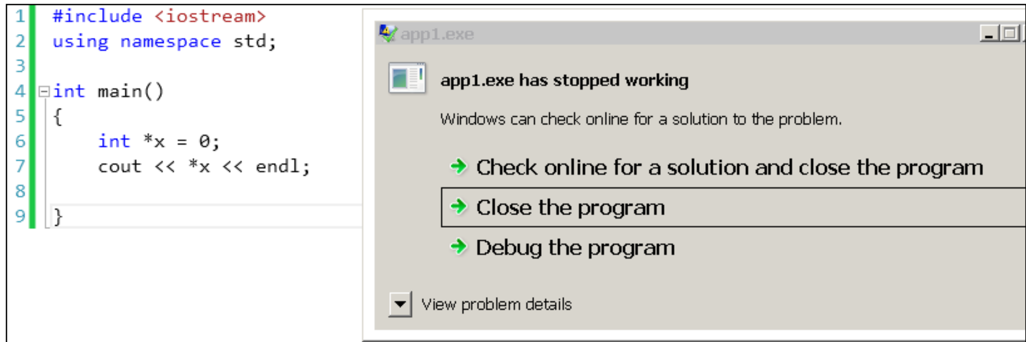


Here, the variable `px` has the address of the variable `x` inside it. In other words, the variable `px` is a reference to another variable. Differencing `px` means to access the variable that `px` is referencing. Differencing is done using the `*` symbol:

```
cout << *px << endl;
```

The Null pointers

A null pointer is a pointer variable with the value `0`. In general, most programmers like to initialize pointers to Null (`0`) on the creation of new pointer variables. Computer programs, in general, can't access the memory address `0` (it is reserved), so if you try to reference a Null pointer, your program will crash, as shown in the following screenshot:



www.it-ebooks.info

Pointer Fun with Binky is a fun video about pointers. Take a look at http://www.youtube.com/watch?v=i49_SNt4yfk.

cin

cin is the way C++ traditionally takes input from the user into the program. cin is easy to use, because it looks at the type of variable it will put the value into as it puts it in. For example, say we want to ask the user his age and store it in an int variable. We can do that as follows:

```
cout << "What is your age?" << endl;
int age;
cin >> age;
```

printf()

Although we have used cout to print out variables so far, you need to know about another common function that is used to print to the console. This function is called the printf function. The printf function is included in the <iostream> library, so you don't have to #include anything extra to use it. Some people in the gaming industry prefer printf to cout (I know I do), so let's introduce it.

Let's proceed to how printf() works, as shown in the following code:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  char character = 'A';
  int integer = 1;
  printf( "integer %d, character %c\n", integer, character );
}
```

Downloading the example code

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

www.it-ebooks.info

We start with a format string. The format string is like a picture frame, and the variables will get plugged in at the locations of the `%` in the format string. Then, the entire thing gets dumped out to the console. In the preceding example, the integer variable will be plugged into the location of the first `%` (`%d`), and the character will be plugged into the location of the second `%` (`%c`), as shown in the following screenshot:

```
printf( "integer %d, character %c\n",
        integer, character );
```

You have to use the right format code to get the output to format correctly; take a look at the following table:

Data type	Format code
Int	%d
Char	%c
String	%s

To print a C++ string, you must use the `string.c_str()` function:

```
string s = "Hello";
printf( "string %s\n", s.c_str() );
```

The `s.c_str()` function accesses the C pointer to the string, which `printf` needs.

If you use the wrong format code, the output won't appear correctly or the program might crash.

Exercise

Ask the user his name and age and take them in using `cin`. Then, issue a greeting for him at the console using `printf()` (not `cout`).

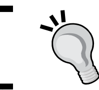
Solution

This is how the program will look:

```
#include <iostream>
#include <string>
using namespace std;
int main()
```


www.it-ebooks.info

```
{
  cout << "Name?" << endl;
  string name;
  cin >> name;
  cout << "Age?" << endl;
  int age;
  cin >> age;
  cout << "Hello " << name << " I see you have attained " << age
  << " years. Congratulations." << endl;
}
```



A string is actually an object type. Inside it is just a bunch of chars!

Summary

In this chapter, we spoke about variables and memory. We talked about mathematical operations on variables and how simple they were in C++.

We also discussed how arbitrarily complex data types can be built using a combination of these simpler data types, such as floats, integers, and characters. Constructions such as this are called objects.

www.it-ebooks.info

www.it-ebooks.info

3
If, Else, and Switch

In the previous chapter, we discussed the importance of memory and how it can be used to store data inside a computer. We spoke about how memory is reserved for your program using variables, and how we can include different types of information in our variables.

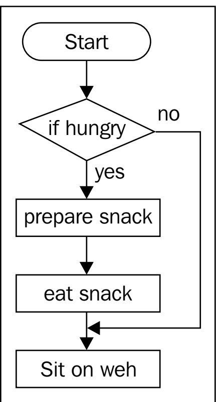
In this chapter, we will talk about how to control the flow of our program and how we can change what code gets executed by branching the code using control flow statements. Here, we'll discuss the different types of control flow, as follows:

- If statements
- How to check whether things are equal using the `==` operator
- Else statements
- How to test for inequalities (that is, how to check whether one number is greater or smaller than another using the operators >, >=, <, <=, and !=)
- Using logical operators (such as not (!), and (&&), or (||))
- Our first example project with Unreal Engine
- Branching in more than two ways:
    - The else if statement
    - The switch statement

www.it-ebooks.info

Branching

The computer code we wrote in *Chapter 2*, *Variables and Memory* went in one direction: straight down. Sometimes, we might want to be able to skip parts of the code. We might want the code to be able to branch in more than one direction. Schematically, we can represent this in the following manner:



A flowchart

In other words, we want the option to not run certain lines of code under certain conditions. The preceding figure is called a flowchart. According to this flowchart, if and only if we are hungry, then we will go prepare a sandwich, eat it, and then go and rest on the couch. If we are not hungry, then there is no need to make a sandwich, so we will simply rest on the couch.

We'll use flowcharts in this book only sometimes, but in UE4, you can even use flowcharts to program your game (using something called blueprints).

This book is about C++ code, so we will always transform our flowcharts into actual C++ code in this book.

Controlling the flow of your program

Ultimately, what we want is the code to branch in one way under certain conditions. Code commands that change which line of code gets executed next are called control flow statements. The most basic control flow statement is the `if` statement. To be able to code `if` statements, we first need a way to check the value of a variable.

So, to start, let's introduce the `==` symbol, which is used to check the value of a variable.

www.it-ebooks.info

The == operator

In order to check whether two things are equal in C++, we need to use not one but two equal signs (`==`) one after the other, as shown here:

```
int x = 5; // as you know, we use one equals sign
int y = 4; // for assignment..
// but we need to use two equals signs
// to check if variables are equal to each other
cout << "Is x equal to y? C++ says: " << (x == y) << endl;
```

If you run the preceding code, you will notice that the output is this:

```
Is x equal to y? C++ says: 0
```

In C++, 1 means true, and 0 means false. If you want the words true or false to appear instead of 1 and 0, you can use the `boolalpha` stream manipulator in the `cout` line of code, as shown here:

```
cout << "Is x equal to y? C++ says: " << boolalpha <<
        (x == y) << endl;
```

The `==` operator is a type of comparison operator. The reason why C++ uses `==` to check for equality and not just `=` is that we already used up the `=` symbol for the assignment operator! (see the *More on variables* section in *Chapter 2*, *Variables and Memory*). If we use a single `=` sign, C++ will assume that we want to overwrite `x` with `y`, not compare them.

Coding if statements

Now that we have the double equals sign under our belt, let's code the flowchart. The code for the preceding flowchart figure is as follows:

```
bool isHungry = true;  // can set this to false if not
                       // hungry!
if( isHungry == true ) // only go inside { when isHungry is true
{
  cout << "Preparing snack.." << endl;
  cout << "Eating .. " << endl;
}
cout << "Sitting on the couch.." << endl;
}
```


www.it-ebooks.info

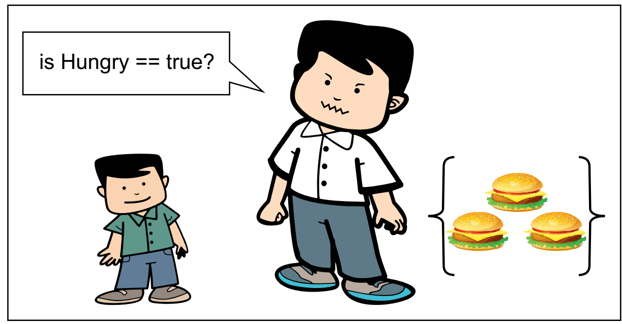
This is the first time we are using a `bool` variable! A `bool` variable either holds the value `true` or the value `false`.

First, we start with a `bool` variable called `isHungry` and just set it to `true`.

Then, we use an `if` statement, as follows:

```
if( isHungry == true )
```

The if statement acts like a guard on the block of code below it. (Remember that a block of code is a group of code encased within { and }.)



You can only read the code between { and } if `isHungry==true`

You can only get at the code inside the curly braces when `isHungry == true`. Otherwise, you will be denied access and forced to skip over that entire block of code.

We an achieve the same effect by simply writing the following line of code:

```
if( isHungry )      // only go here if isHungry is
true
```

This can be used as an alternative for the following:

```
if( isHungry == true )
```

The reason people might use the `if( isHungry )` form is to avoid the possibility of making mistakes. Writing `if( isHungry = true )` by accident will set `isHungry` to true every time the `if` statement is hit! To avoid this possibility, we can just write `if( isHungry )` instead. Alternatively, some (wise) people use what are called Yoda conditions to check an if statement: `if( true == isHungry )`. The reason we write the `if` statement in this way is that, if we accidentally write `if( true = isHungry )`, this will generate a compiler error, catching the mistake.

www.it-ebooks.info

Try to run this code segment to see what I mean:

```
int x = 4, y = 5;
cout << "Is x equal to y? C++ says: " << (x = y) << endl; //bad!
// above line overwrote value in x with what was in y,
// since the above line contains the assignment x = y
// we should have used (x == y) instead.
cout << "x = " << x << ", y = " << y << endl;
```

The following lines show the output of the preceding lines of code:

```
Is x equal to y? C++ says: 5
x = 5, y = 5
```

The line of code that has `(x = y)` overwrites the previous value of `x` (which was 4) with the value of `y` (which is 5). Although we were trying to check whether `x` equals `y`, what happened in the previous statement was that `x` was assigned the value of `y`.

Coding else statements

The `else` statement is used to have our code do something in the case that the `if` portion of the code does not run.

For example, say we have something else that we'd like to do in case we are not hungry, as shown in the following code snippet:

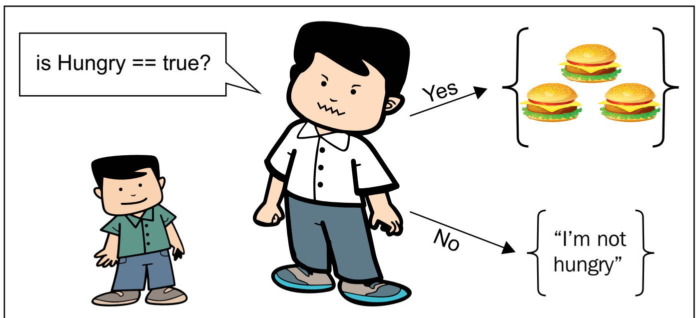
```
bool isHungry = true;
if( isHungry )      // notice == true is implied!
{
  cout << "Preparing snack.." << endl;
  cout << "Eating .. " << endl;
}
else                // we go here if isHungry is FALSE
{
  cout << "I'm not hungry" << endl;
}

cout << "Sitting on the couch.." << endl;
}
```

There are a few important things that you need to remember about the `else` keyword, as follows:

- An `else` statement must always immediately follow after an `if` statement. You can't have any extra lines of code between the end of the if block and the corresponding else block.

www.it-ebooks.info

- You can never go into both the if and the corresponding else blocks. It's always one or the other.



The else statement is the way you will go if `isHungry` is not equal to true

You can think of the `if/else` statements as a guard diverting people to either the left or the right. Each person will either go towards the food (when `isHungry==true`), or they will go away from the food (when `isHungry==false`).

Testing for inequalities using other comparison operators (>, >=, <, <=, and !=)

Other logical comparisons can be easily done in C++. The > and < symbols mean just what they do in math. They are the greater than (>) and less than (<) symbols, respectively. >= has the same meaning as the ≥ symbol in math. <= is the C++ code for ≤. Since there isn't a ≤ symbol on the keyboard, we have to write it using two characters in C++. `!=` is how we say "not equal to" in C++. So, for example, say we have the following lines of code:

```
int x = 9;
int y = 7;
```

We can ask the computer whether `x > y` or `x < y` as shown here:

```
cout << "Is x greater than y? " << (x > y) << endl;
cout << "Is x greater than OR EQUAL to y? " << (x >= y) << endl;
cout << "Is x less than y? " << (x < y) << endl;
cout << "Is x less than OR EQUAL to y? " << (x <= y) << endl;
cout << "Is x not equal to y? " << (x != y) << endl;
```


www.it-ebooks.info

> We need the brackets around the comparisons of x and y because of something known as operator precedence. If we don't have the brackets, C++ will get confused between the << and < operators. It's weird and you will better understand this later, but you need C++ to evaluate the (x < y) comparison before you output the result (<<). There is an excellent table available for reference at `http://en.cppreference.com/w/cpp/language/operator_precedence`.

Using logical operators

Logical operators allow you to do more complex checks, rather than checking for a simple equality or inequality. Say, for example, the condition to gain entry into a special room requires the player to have both the red and green keycards. We want to check whether two conditions hold true at the same time. To do this type of complex logic statement checks, there are three additional constructs that we need to learn: the *not* (`!`), *and* (`&&`), and *or* (`||`) operators.

The Not (!) operator

The `!` operator is handy to reverse the value of a `boolean` variable. Take an example of the following code:

```
bool wearingSocks = true;
if( !wearingSocks ) // same as if( false == wearingSocks )
{
    cout << "Get some socks on!" << endl;
}
else
{
    cout << "You already have socks" << endl;
}
```

The `if` statement here checks whether or not you are wearing socks. Then, you are issued a command to get some socks on. The `!` operator reverses the value of whatever is in the `boolean` variable to be the opposite value.

We use something called a truth table to show all the possible results of using the `!` operator on a `boolean` variable, as follows:

wearingSocks	**!wearingSocks**
true	false
false	true

www.it-ebooks.info

So, when `wearingSocks` has the value true, `!wearingSocks` has the value `false` and vice versa.

Exercises

1. What do you think will be the value of `!!wearingSocks` when the value of `wearingSocks` is true?
2. What is the value of `isVisible` after the following code is run?

```
bool hidden = true;
bool isVisible = !hidden;
```

Solution

1. If `wearingSocks` is true, then `!wearingSocks` is false. Therefore, `!!wearingSocks` becomes true again. It's like saying *I am not not hungry*. Not not is a double negative, so this sentence means that I am actually hungry.
2. The answer to the second question is false. `hidden` was true, so `!hidden` is false. false then gets saved into the `isVisible` variable.

The `!` operator is sometimes colloquially known as bang. The preceding bang bang operation (`!!`) is a double negative and a double logical inversion. If you bang-bang a `bool` variable, there is no net change to the variable. If you bang-bang an `int` variable, it becomes a simple `bool` variable(`true` or `false`). If the `int` value is greater than zero, it is reduced to a simple `true`. If the `int` value is 0 already, it is reduced to a simple `false`.

The And (&&) operator

Say, we only want to run a section of the code if two conditions are true. For example, we are only dressed if we are wearing both socks and clothes. You can use the following code to checks this:

```
bool wearingSocks = true;
bool wearingClothes = false;
if( wearingSocks && wearingClothes )// && requires BOTH to be true
{
   cout << "You are dressed!" << endl;
}
else
{
   cout << "You are not dressed yet" << endl;
}
```


www.it-ebooks.info

The Or (||) operator

We sometimes want to run a section of the code if either one of the variables is `true`.

So, for example, say the player wins a certain bonus if he finds either a special star in the level or the time that he takes to complete the level is less than 60 seconds, in which case you can use the following code:

```
bool foundStar = true;
float levelCompleteTime = 25.f;
float maxTimeForBonus = 60.f;
// || requires EITHER to be true to get in the { below
if( foundStar || levelCompleteTime < maxTimeForBonus )
{
   cout << "Bonus awarded!" << endl;
}
else
{
   cout << "No bonus." << endl;
}
```

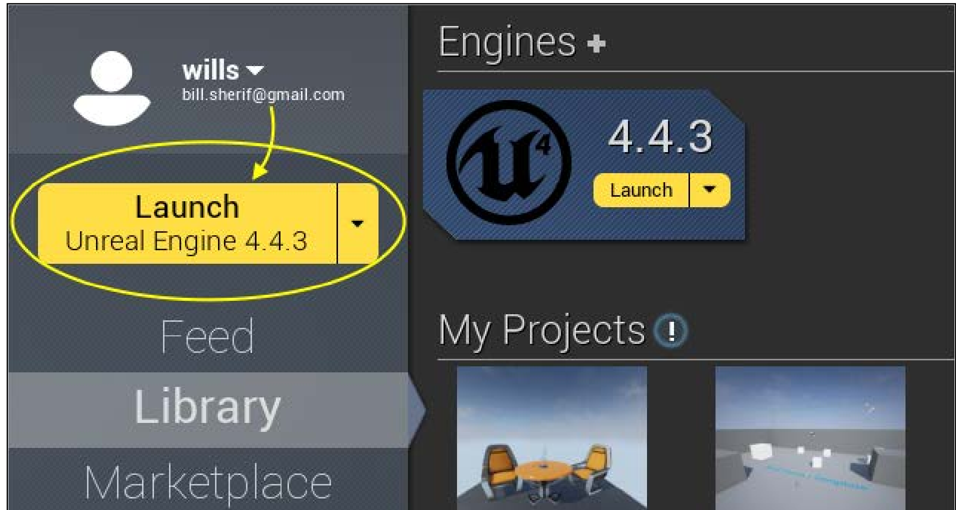
Our first example with Unreal Engine

We need to get started with Unreal Engine.

A word of warning: when you open your first Unreal project, you will find that the code looks very complicated. Don't get discouraged. Simply focus on the highlighted parts. Throughout your career as a programmer, you will often have to deal with very large code bases containing sections that you do not understand. However, focusing on the parts that you do understand will make this section productive.

www.it-ebooks.info

Open the **Unreal Engine Launcher** app (which has the blue-colored UE4 icon). Select **Launch Unreal Engine 4.4.3**, as shown in the following screenshot:

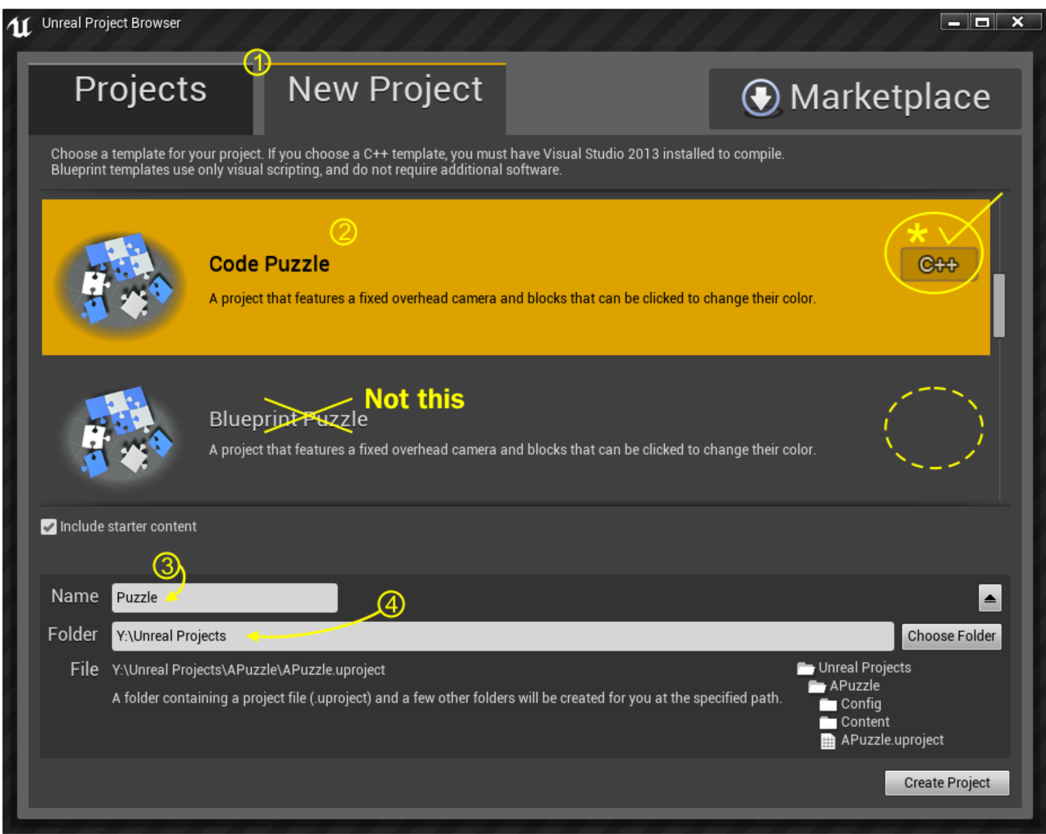


If the **Launch** button is grayed out, you need to go to the **Library** tab and download an engine (~3 GB).

Once the engine is launched (which might take a few seconds), you will be in the **Unreal Project Browser** screen (black-colored UE4 icon), as shown in the following screenshot.

Now, select the **New Project** tab in the UE4 project browser. Scroll down until you reach **Code Puzzle**. This is one of the simpler projects that doesn't have too much code, so it's good to start with. We'll go to the 3D projects later.

www.it-ebooks.info



Here are a few things to make a note of in this screen:

- Be sure you're in the **New Project** tab
- When you click on **Code Puzzle**, make sure that it is the one with the **C++** icon at the right, not **Blueprint Puzzle**
- Enter a name for your project, `Puzzle`, in the **Name** box (this is important for the example code I will give you to work on later)
- If you want to change the storage folder (to a different drive), click the down arrow so that the folder appears. Then, name the directory where you want to store your project.

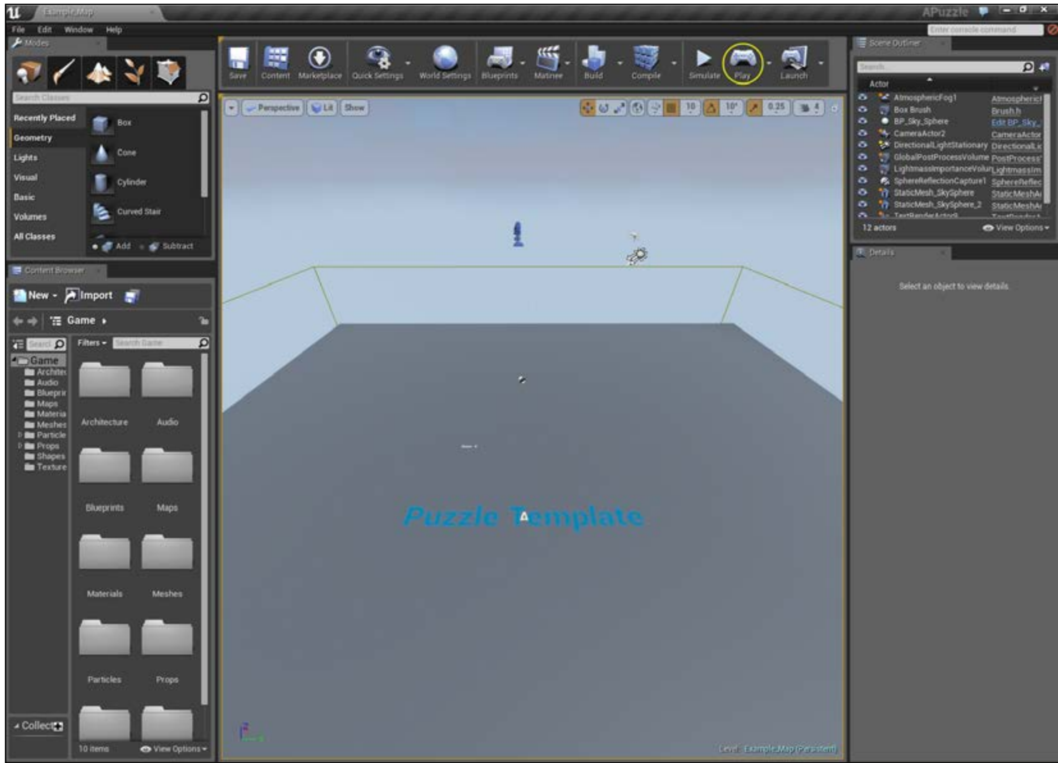
After you've done all this, select **Create Project**.

Visual Studio 2013 will open with the code of your project.

Press *Ctrl+F5* to build and launch the project.

www.it-ebooks.info

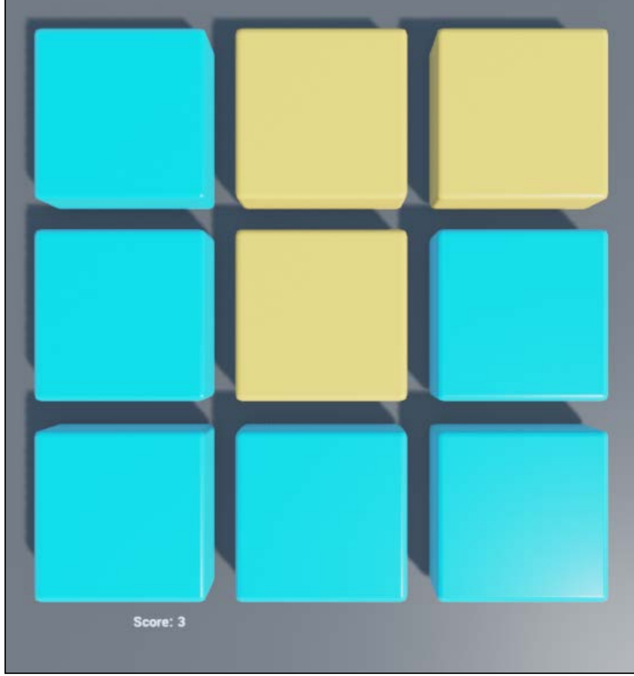
Once the project compiles and runs, you should see the Unreal Editor, as shown in the following screenshot:



Looks complicated? Oh boy, it sure is! We'll explore some of the functionality in the toolbars at the side later. For now, just select **Play** (marked in yellow), as shown in the preceding screenshot.

www.it-ebooks.info

This launches the game. This is how it should look:



Now, try clicking on the blocks. As soon as you click on a block, it turns orange, and this increases your score.

What we're going to do is find the section that does this and change the behavior a little.

Find and open the `PuzzleBlock.cpp` file.

> In Visual Studio, the list of files in the project is located inside the **Solution Explorer**. If your **Solution Explorer** is hidden, simply click on **View/Solution Explorer** from the menu at the top.

www.it-ebooks.info

Inside this file, scroll down to the bottom, where you'll find a section that begins with the following words:

```
void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp)
```

`APuzzleBlock` is the class name, and `BlockClicked` is the function name. Whenever a puzzle block gets clicked on, the section of code from the starting { to the ending } is run. Hopefully, exactly how this happens will make more sense later.

It's kind of like an `if` statement in a way. If a puzzle piece is clicked on, then this group of the code is run for that puzzle piece.

We're going to walk through the steps to make the blocks flip colors when they are clicked on (so, a second click will change the color of the block from orange back to blue).

Perform the following steps with the utmost care:

1. Open `PuzzleBlock.h` file. After line 25 (which has this code):

   ```
   /** Pointer to orange material used on active blocks */
   UPROPERTY()
   class UMaterialInstance* OrangeMaterial;
   ```

   Insert the following code after the preceding lines of code:

   ```
   UPROPERTY()
   class UMaterialInstance* BlueMaterial;
   ```

2. Now, open `PuzzleBlock.cpp` file. After line 40 (which has this code):

   ```
   // Save a pointer to the orange material
   OrangeMaterial = ConstructorStatics.OrangeMaterial.Get();
   ```


www.it-ebooks.info

Insert the following code after the preceding lines:

```
BlueMaterial = ConstructorStatics.BlueMaterial.Get();
```

3. Finally, in `PuzzleBlock.cpp`, replace the contents of the `void APuzzleBlock::BlockClicked` section of code (line 44) with the following code:

```
void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp)
{
  // --REPLACE FROM HERE--
  bIsActive = !bIsActive; // flip the value of bIsActive
  // (if it was true, it becomes false, or vice versa)
  if ( bIsActive )
  {
    BlockMesh->SetMaterial(0, OrangeMaterial);
  }
  else
  {
    BlockMesh->SetMaterial(0, BlueMaterial);
  }
  // Tell the Grid
  if(OwningGrid != NULL)
  {
    OwningGrid->AddScore();
  }
  // --TO HERE--
}
```

Only replace inside the `void APuzzleBlock::BlockClicked (UPrimitiveComponent* ClickedComp)` statement.

Do not replace the line that starts with `void APuzzleBlock::BlockClicked`. You might get an error (if you haven't named your project Puzzle). You've been warned.

So, let's analyze this. This is the first line of code:

```
bIsActive = !bIsActive; // flip the value of bIsActive
```

This line of code simply flips the value of `bIsActive`. `bIsActive` is a `bool` variable (it is created in `APuzzleBlock.h`). If `bIsActive` is true, `!bIsActive` will be false. So, whenever this line of code is hit (which happens with a click on any block), the `bIsActive` value is reversed (from `true` to `false` or from `false` to `true`).

www.it-ebooks.info

Let's consider the next block of code:

```
if ( bIsActive )
  {
    BlockMesh->SetMaterial(0, OrangeMaterial);
  }
  else
  {
    BlockMesh->SetMaterial(0, BlueMaterial);
  }
```

We are simply changing the block color. If `bIsActive` is true, then the block becomes orange. Otherwise, the block turns blue.

Exercise

By now, you should notice that the best way to get better at programming is by doing it. You have to practice programming a lot to get significantly better at it.

Create two integer variables, called x and y, and read them in from the user. Write an `if/else` statement pair that prints the name of the bigger-valued variable.

Solution

The solution of the preceding exercise is shown in the following block of code:

```
int x, y;
cout << "Enter two numbers integers, separated by a space " << endl;
cin >> x >> y;
if( x < y )
{
  cout << "x is less than y" << endl;
}
else
{
  cout << "x is greater than y" << endl;
}
```

Don't type a letter when `cin` expects a number. `cin` can fail and give a bad value to your variable if that happens.

www.it-ebooks.info

Branching code in more than two ways

In the previous sections, we were only able to make the code branch in one of the two ways. In pseudocode, we had the following code:

```
if( some condition is true )
{
  execute this;
}
else // otherwise
{
  execute that;
}
```

Pseudocode is *fake code*. Writing pseudocode is a great way to brainstorm and plan out your code, especially if you are not quite used to C++.

This code is a little bit like a metaphorical fork in the road, with only one of two directions to choose from.

Sometimes, we might want to branch the code in more than just two directions. We might want the code to branch in three ways, or even more. For example, say the direction in which the code goes depends on what item the player is currently holding. The player can be holding one of three different items: a coin, key, or sand dollar. And C++ allows that! In fact, in C++, you can branch in any number of directions as you wish.

The else if statement

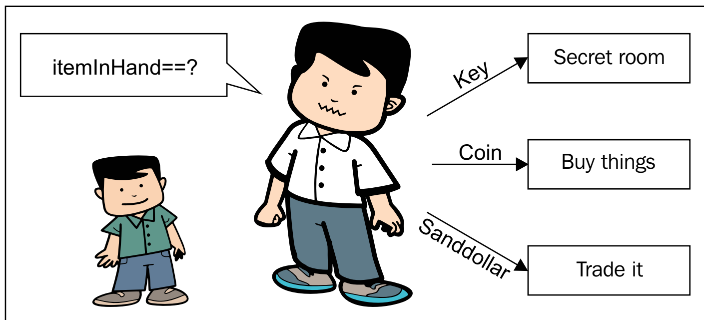
The `else if` statement is a way to code in more than just two possible branch directions. In the following code example, the code will go in one of the three different ways, depending on whether the player is holding the `Coin`, `Key`, or `Sanddollar` objects:

```
#include <iostream>
using namespace std;
int main()
{
  enum Item  // enums define a new type of variable!
  {
    Coin, Key, Sanddollar // variables of type Item can have
    // any one of these 3 values
  }
```


www.it-ebooks.info

```
    Item itemInHand = Key;  // Try changing this value to Coin,
                            // Sanddollar
    if( itemInHand == Key )
    {
      cout << "The key has a lionshead on the handle." << endl;
      cout << "You got into a secret room using the Key!" << endl;
    }
    else if( itemInHand == Coin )
    {
      cout << "The coin is a rusted brassy color. It has a picture
      of a lady with a skirt." << endl;
      cout << "Using this coin you could buy a few things" << endl;
    }
    else if( itemInHand == Sanddollar )
    {
      cout << "The sanddollar has a little star on it." << endl;
      cout << "You might be able to trade it for something." <<
      endl;
    }
    return 0;
  }
```

Note that the preceding code only goes in one of the three separate ways! In an `if`, `else if`, and `else if` series of checks, we will only ever go into one of the blocks of code.



Exercise

Use C++ program to answer the questions that follow. Be sure to try these exercises in order to gain fluency with these equality operators.

```
#include <iostream>
using namespace std;
```


www.it-ebooks.info

```
int main()
{
  int x;
  int y;
  cout << "Enter an integer value for x:" << endl;
  cin >> x; // This will read in a value from the console
  // The read in value will be stored in the integer
  // variable x, so the typed value better be an integer!
  cout << "Enter an integer value for y:" << endl;
  cin >> y;
  cout << "x = " << x << ", y = " << y << endl;
  // *** Write new lines of code here
}
```

Write some new lines of code at the spot that says (`// *** Write new...`):

1. Check whether `x` and `y` are equal. If they are equal, print `x and y are equal`. Otherwise, print `x and y are not equal`.
2. An exercise on inequalities: check whether `x` is greater than `y`. If it is, print `x is greater than y`. Otherwise, print `y is greater than x`.

Solution

To evaluate equality, insert the following code:

```
if( x == y )
{
  cout << "x and y are equal" << endl;
}
else
{
  cout << "x and y are not equal" << endl;
}
```

To check which value is greater insert the following code:

```
if( x > y )
{
  cout << "x is greater than y" << endl;
}
else if( x < y )
{
  cout << "y is greater than x" << endl;
}
else // in this case neither x > y nor y > x
{
  cout << "x and y are equal" << endl;
}
```


www.it-ebooks.info

The switch statement

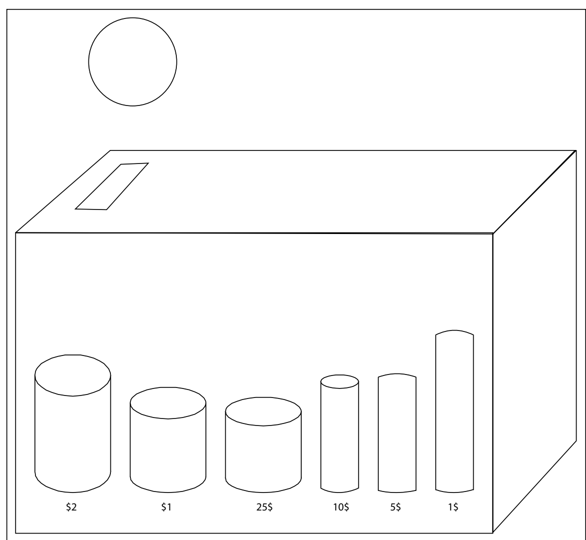
The `switch` statement allows your code to branch in multiple ways. What the `switch` statement will do is look at the value of a variable, and depending on its value, the code will go in a different direction.

We'll also introduce the `enum` construct here:

```
#include <iostream>
using namespace std;
enum Food  // enums define a new type of variable!
{
  // a variable of type Food can have any of these values
  Fish,
  Bread,
  Apple,
  Orange
};
int main()
{
  Food food = Bread; // Change the food here
  switch( food )
  {
    case Fish:
      cout << "Here fishy fishy fishy" << endl;
      break;
    case Bread:
      cout << "Chomp! Delicious bread!" << endl;
      break;
    case Apple:
      cout << "Mm fruits are good for you" << endl;
      break;
    case Orange:
      cout << "Orange you glad I didn't say banana" << endl;
      break;
    default:  // This is where you go in case none
              // of the cases above caught
      cout << "Invalid food" << endl;
      break;
  }
  return 0;
}
```


www.it-ebooks.info

Switches are like coin sorters. When you drop 25 cent into the coin sorter, it finds its way into the 25 cent pile. Similarly, a `switch` statement will simply allow the code to jump down to the appropriate section. The example of sorting the coins is shown in the following figure:



The code inside the `switch` statement will continue to run (line by line) until the `break;` statement is hit. The `break` statement jumps you out of the `switch` statement. Take a look at the following diagram to understand how the switch works:

```
   Food food = Fish; // Change the food here
(1)switch( food )
   {
   (2)case Fish:
       (3)cout << "Here fishy fishy fishy" << endl;
       (4)break;
      case Bread:
          cout << "Chomp! Delicious bread!" << endl;
          break;
   }
   cout << "End of switch" << endl;
```

1. First, the `Food` variable is inspected. What value does it have? In this case, it has `Fish` inside it.
2. The `switch` command jumps down to the correct case label. (If there is no matching case label, the switch will just be skipped).

www.it-ebooks.info

3. The `cout` statement is run, and `Here fishy fishy fishy` appears on the console.
4. After inspecting the variable and printing the user response, the break statement is hit. This makes us stop running lines of code in the switch and exit the switch. The next line of code that is run is just what would otherwise have been the next line of code in the program if the switch had not been there at all (after the closing curly brace of the switch statement). It is the print statement at the bottom, which says "End of switch".

Switch versus if

Switches are like the `if` / `else if` / `else` chains from earlier. However, switches can generate code faster than `if` / `else if` / `else if` / `else` chains. Intuitively, switches only jump to the appropriate section of the code to execute. If / else if / else chains might involve more complicated comparisons (including logical comparisons), which might take more CPU time. The main reason you will use the `if` statements is to do more with your own custom comparisons inside the brackets.

An enum is really an int. To verify this, print the following code:

```
cout <<  "Fish=" << Fish <<
         " Bread=" << Bread <<
         " Apple=" << Apple <<
         " Orange=" << Orange << endl;
```

You will see the integer values of the enum—just so you know.

Sometimes, programmers want to group multiple values under the same switch `case` label. Say, we have an `enum`, object as follows:

```
enum Vegetables { Potato, Cabbage, Broccoli, Zucchini };
```

A programmer wants to group all the greens together, so he writes a `switch` statement as follows:

```
switch( veg )
{
case Zucchini:   // zucchini falls through because no break
case Broccoli:   // was written here
  cout << "Greens!" << endl;
  break;
default:
  cout << "Not greens!" << endl;
  break;
}
```


www.it-ebooks.info

In this case, `Zucchini` falls through and executes the same code as `Broccoli`. The non-green vegetables are in the `default` case label. To prevent a fall through, you have to remember to insert an explicit `break` statement after each `case` label.

We can write another version of the same switch that does not let Zucchini fall through, by the explicit use of the keyword `break` in the switch:

```
switch( veg )
{
case Zucchini:   // zucchini no longer falls due to break
  cout << "Zucchini is a green" << endl;
  break;// stops case zucchini from falling through
case Broccoli:   // was written here
  cout << "Broccoli is a green" << endl;
  break;
default:
  cout << "Not greens!" << endl;
  break;
}
```

Note that it is good programming practice to `break` the `default` case as well, even though it is the last `case` listed.

Exercise

Complete the following program, which has an `enum` object with a series of mounts to choose from. Write a `switch` statement that prints the following messages for the mount selected:

Horse	The steed is valiant and mighty
Mare	This mare is white and beautiful
Mule	You are given a mule to ride. You resent that.
Sheep	Baa! The sheep can barely support your weight.
Chocobo	Chocobo!

Remember, an `enum` object is really an `int` statement. The first entry in an `enum` object is by default 0, but you can give the `enum` object any starting value you wish using the = operator. Subsequent values in the `enum` object are `ints` arranged in order.

www.it-ebooks.info

Bit-shifted enum

A common thing to do in an `enum` object is to assign a bit-shifted value to each entry:

```
enum WindowProperties
{
  Bordered    = 1 << 0, // binary 001
  Transparent = 1 << 1, // binary 010
  Modal       = 1 << 2  // binary 100
};
```

The bit-shifted values should be able to combine the window properties. This is how the assignment will look:

```
// bitwise OR combines properties
WindowProperties wp = Bordered | Modal;
```

Checking which `WindowProperties` have been set involves a check using `bitwise AND`:

```
// bitwise AND checks to see if wp is Modal
if( wp & Modal )
{
cout << "You are looking at a modal window" << endl;
}
```

Bit shifting is a technique that is slightly beyond the scope of this text, but I've included this tip just so you know about it.

Solution

The solution of the preceding exercise is shown in the following code:

```
#include <iostream>
using namespace std;
enum Mount
{
  Horse=1, Mare, Mule, Sheep, Chocobo
  // Since Horse=1, Mare=2, Mule=3, Sheep=4, and Chocobo=5.
};
int main()
{
  int mount;  // We'll use an int variable for mount
              // so cin works
```


www.it-ebooks.info

```
    cout << "Choose your mount:" << endl;
    cout << Horse << " Horse" << endl;
    cout << Mare << " Mare" << endl;
    cout << Mule << " Mule" << endl;
    cout << Sheep << " Sheep" << endl;
    cout << Chocobo << " Chocobo" << endl;
    cout << "Enter a number from 1 to 5 to choose a mount" << endl;
    cin >> mount;
      // Write your switch here. Describe what happens
      // when you mount each animal in the switch below
    switch( mount )
    {
      default:
        cout << "Invalid mount" << endl;
        break;
    }
  return 0;
  }
```

Summary

In this chapter, you learned how to branch the code. Branching makes it possible for the code to go in a different direction instead of going straight down.

In the next chapter, we will move on to a different kind of control flow statement that will allow you to go back and repeat a line of code a certain number of times. The sections of code that repeat will be called loops.

www.it-ebooks.info

www.it-ebooks.info

4
Looping

In the previous chapter, we discussed the `if` statement. The `if` statement enables you to put a condition on the execution of a block of code.

In this chapter, we will explore loops, which are code structures that enable you to repeat a block of code under certain conditions. We stop repeating that block of code once the condition becomes false.

In this chapter, we will explore the following topics:

- While loops
- Do/while loops
- For loops
- A simple example of a practical loop within Unreal Engine

The while loop

The `while` loop is used to run a section of the code repeatedly. This is useful if you have a set of actions that must be done repeatedly to accomplish some goal. For example, the `while` loop in the following code repeatedly prints the value of the variable `x` as it is incremented from 1 to 5:

```
int x = 1;
while( x <= 5 ) // may only enter the body of the while when x<=5
{
  cout << "x is " << x << endl;
  x++;
}
cout << "Finished" << endl;
```

www.it-ebooks.info

This is the output of the preceding program:

```
x is 1
x is 2
x is 3
x is 4
x is 5
Finished
```

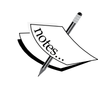
In the first line of code, an integer variable `x` is created and set to 1. Then, we go the `while` condition. The `while` condition says that while `x` is less than or equal to 5, you must stay in the block of code that follows.

Each iteration of the loop (an iteration means going once around the loop) gets a little more done from the task (of printing the numbers 1 to 5). We program the loop to automatically exit once the task is done (when `x <= 5` is no longer true).

Similar to the `if` statement of the previous chapter, entry into the block below the `while` loop is only allowed if you meet the condition within the brackets of the `while` loop (in the preceding example, `x <= 5`). You can try mentally subbing an `if` loop in the place of the `while` loop, as shown in the following code:

```
int x = 1;
if( x <= 5 ) // you may only enter the block below when x<=5
{
  cout << "x is " << x << endl;
  x++;
}
cout << "End of program" << endl;
```

The preceding code sample will only print `x is 1`. So, a `while` loop is exactly like an `if` statement, only it has this special property of automatically repeating itself until the condition between the brackets of the `while` loop becomes `false`.



I'd like to explain the repetition of the `while` loop using a video game. If you don't know Valve's *Portal*, you should play it, if only to understand loops. Check out `https://www.youtube.com/watch?v=TluRVBhmf8w` for a demo video.

www.it-ebooks.info

The `while` loops have a kind of magic *portal* at the bottom, which cause the loop to repeat. The following screenshot illustrates what I mean:

```
int x = 1;
B
while( x <= 5 )
{
    cout << "x is " << x << endl;
    x++;
    O
}
```

There is a portal at the end of the while loop that takes you back to the beginning

In the preceding screenshot, we loop back from the orange portal (marked **O**) to the blue portal (marked **B**). This is our first time of being able to go back in the code. It is like time travel, only for the code. How exciting!

The only way past a `while` loop block is to not meet the entry condition. In the preceding example, once the value of `x` becomes 6 (so, `x <= 5` becomes false), we will not enter the `while` loop again. Since the orange portal is inside the loop, we'll be able to get to finished once `x` becomes 6.

Infinite loops

You can get stuck inside the same loop forever. Consider the modified program in the following block of code (what do you think will be the output?):

```
int x = 1;
while( x <= 5 ) // may only enter the body of the while when x<=5
{
  cout << "x is " << x << endl;
}
cout << "End of program" << endl;
```

This is how the output will look:

```
x is 1
x is 1
x is 1
.
.
.
(repeats forever)
```


www.it-ebooks.info

The loop repeats forever because we removed the line of code that changed the value of `x`. If the value of `x` stays the same and is not allowed to increase, we will be stuck inside the body of the `while` loop. This is because the loop's exit condition (the value of `x` becomes 6) cannot be met if `x` does not change inside the loop body.

The following exercises will use all the concepts from the previous chapters, such as the `+=` and decrement operations. If you've forgotten something, go back and reread the previous sections.

Exercises

1. Write a `while` loop that will print the numbers from 1 to 10.
2. Write a `while` loop that will print the numbers from 10 to 1 (backwards).
3. Write a `while` loop that will print numbers 2 to 20, incrementing by 2 (for example 2, 4, 6, and 8).
4. Write a `while` loop that will print the numbers 1 to 16 and their squares beside them.

The following is an example program output of the exercise 4:

1	1
2	4
3	9
4	16
5	25

Solutions

The code solutions of the preceding exercises are as follows:

1. The solution of the `while` loop that prints the numbers from 1 to 10 is as follows:

```
int x = 1;
while( x <= 10 )
{
  cout << x << endl;
  x++;
}
```


www.it-ebooks.info

2. The solution of the `while` loop that prints the numbers from 10 to 1 in backwards is as follows:

   ```
   int x = 10; // start x high
   while( x >= 1 ) // go until x becomes 0 or less
   {
     cout << x << endl;
     x--; // take x down by 1
   }
   ```

3. The solution of the `while` loop that prints the numbers from 2 to 20 incrementing by 2 is as follows:

   ```
   int x = 2;
   while( x <= 20 )
   {
     cout << x << endl;
     x+=2; // increase x by 2's
   }
   ```

4. The solution of the `while` loop that prints the numbers from 1 to 16 with their squares is as follows:

   ```
   int x = 1;
   while( x <= 16 )
   {
     cout << x << "    " << x*x << endl; // print x and it's
     square
     x++;
   }
   ```

The do/while loop

The `do/while` loop is almost identical to the `while` loop. Here's an example of a `do/while` loop that is equivalent to the first `while` loop that we examined:

```
int x = 1;
do
{
  cout << "x is " << x << endl;
  x++;
} while( x <= 5 ); // may only loop back when x<=5
cout << "End of program" << endl;
```


www.it-ebooks.info

The only difference here is that we don't have to check the `while` condition on our first entry into the loop. This means that the `do/while` loop's body is always executed at least once (where a `while` loop can be skipped entirely if the condition to enter the `while` loop is `false` when you hit it for the first time).

The for loop

The `for` loop has a slightly different anatomy than the `while` loop, but both are very similar.

Let's examine the anatomy of a `for` loop as compared to an equivalent `while` loop. Take an example of the following code snippets:

The for loop	An equivalent while loop

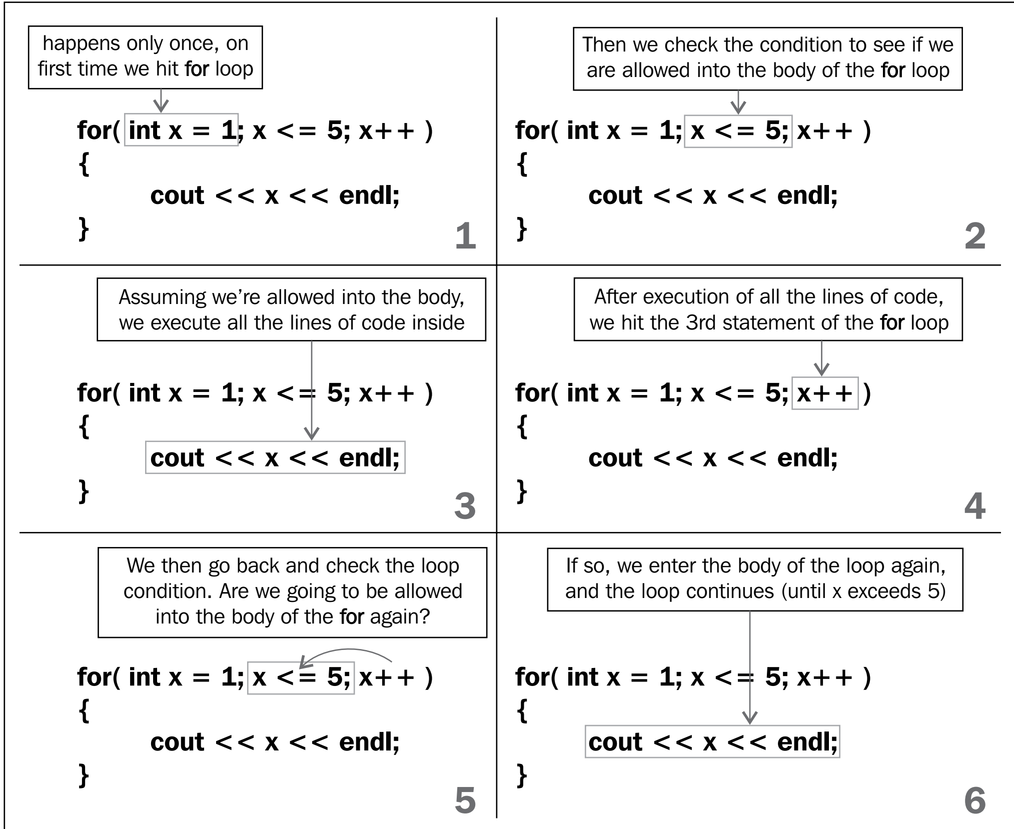
`for( int x = 1; x <= 5; x++ )`
`{`
`  cout << x << endl;`
`}`
	`int x = 1;`
	`while( x <= 5 )`
	`{`
	`  cout << x << endl;`
	`  x++;`
	`}`

The `for` loop has three statements inside its brackets. Let's examine them in order.

The first statement of the `for` loop (`int x = 1;`) only gets executed once, when we first enter the body of the `for` loop. It is typically used to initialize the value of the loop's counter variable (in this case, the variable `x`). The second statement inside the `for` loop (`x <= 5;`) is the loop's repeat condition. As long as `x <= 5`, we must continue to stay inside the body of the `for` loop. The last statement inside the brackets of the `for` loop (`x++;`) gets executed after we complete the body of the `for` loop each time.

www.it-ebooks.info

The following sequence of diagrams explain the progression of the `for` loop:



Exercises

1. Write a `for` loop that will gather the sum of the numbers from 1 to 10.
2. Write a `for` loop that will print the multiples of 6, from 6 to 30 (6, 12, 18, 24, and 30).
3. Write a `for` loop that will print numbers 2 to 100 in multiples of 2 (for example, 2, 4, 6, 8, and so on).
4. Write a `for` loop that will print numbers 1 to 16 and their squares beside them.

www.it-ebooks.info

Solutions

Here are the solutions for the preceding exercises:

1. The solution for the `for` loop for printing the sum of the numbers from 1 to 10 is as follows:

```
int sum = 0;
for( int x = 1; x <= 10; x++ )
{
  sum += x;
  cout << x << endl;
}
```

2. The solution for the `for` loop for printing multiples of 6 from 30 is as follows:

```
for( int x = 6; x <= 30; x += 6 )
{
  cout << x << endl;
}
```

3. The solution for the `for` loop for printing numbers from 2 to 100 in multiples of 2 is as follows:

```
for( int x = 2; x <= 100; x += 2 )
{
  cout << x << endl;
}
```

4. The solution for the `for` loop that prints numbers from 1 to 16 and their squares is as follows:

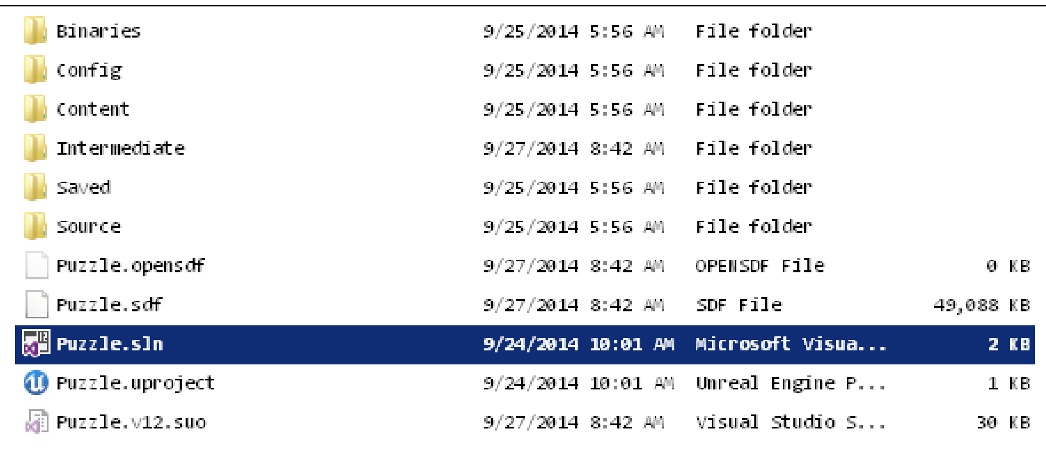
```
for( int x = 1; x <= 16; x++ )
{
  cout << x << " " << x*x << endl;
}
```

Looping with Unreal Engine

In your code editor, open your Unreal Puzzle project from *Chapter 3, If, Else, and Switch*.

www.it-ebooks.info

There are several ways to open your Unreal project. The simplest way is probably to navigate to the `Unreal Projects` folder (which is present in your user's `Documents` folder on Windows by default) and double-click on the `.sln` file in **Windows Explorer**, as shown in the following screenshot:



On Windows, open the .sln file to edit the project code

Now, open the `PuzzleBlockGrid.cpp` file. Inside this file, scroll down to the section that begins with the following statement:

```
void APuzzleBlockGrid::BeginPlay()
```

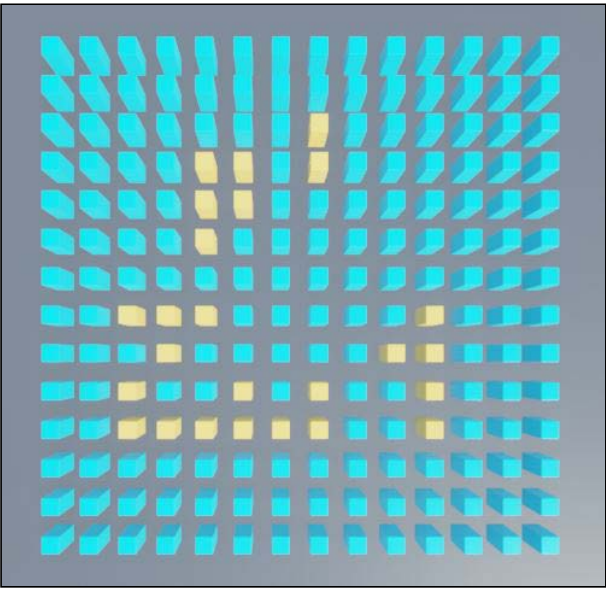
Notice that there is a `for` loop here to spawn the initial nine blocks, as shown in the following code:

```
// Loop to spawn each block
for( int32 BlockIndex=0; BlockIndex < NumBlocks; BlockIndex++ )
{
  // ...
}
```

Since `NumBlocks` (which is used to determine when to stop the loop) gets computed as `Size*Size`, we can easily change the number of blocks that get spawned by altering the value of the `Size` variable. Go to line 23 of `PuzzleBlockGrid.cpp` and change the value of the `Size` variable to four or five. Then, run the code again.

www.it-ebooks.info

You should see the number of blocks on the screen increase, as shown in the following screenshot:



Setting the size to 14 creates many more blocks

Summary

In this chapter, you learned how to repeat lines of code by looping the code, which allowed you to go back into it. This can be used to repeatedly use the same line of code in order to achieve a task. Imagine printing the numbers from 1 to 10 without using a loop.

In the next chapter, we will explore functions, which are the basic units of reusable code.

www.it-ebooks.info

5
Functions and Macros

Functions

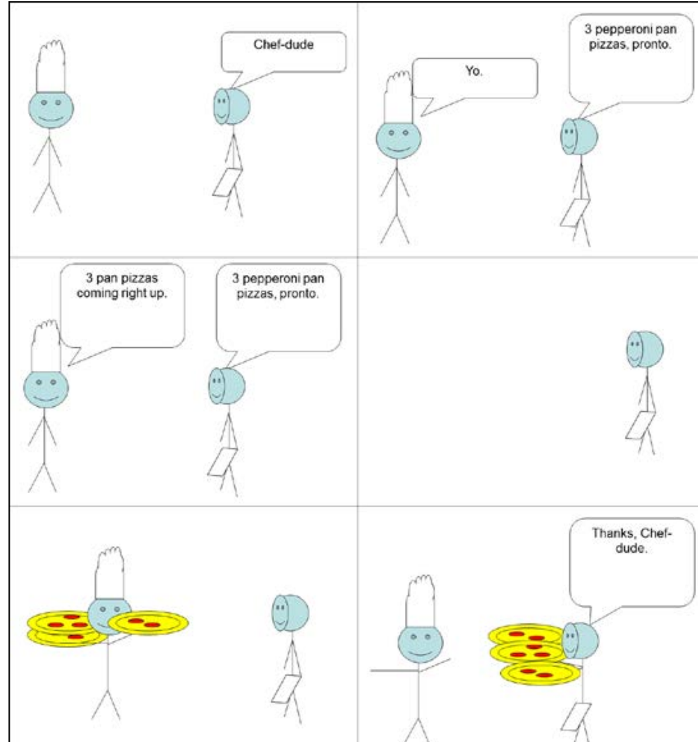
Some things needs to be repeated. Code is not one of them. A function is a bundle of code that can get called any number of times, as often you wish.

Analogies are good. Let's explore an analogy that deals with waiters, chefs, pizzas, and functions. In English, when we say a person has a function, we mean that the person performs some very specific (usually, very important) task. They can do this task again and again and whenever they are called upon to do so.

The following comic strip shows the interaction between a waiter (caller) and a chef (callee). The waiter wants food for his table, so he calls upon the chef to prepare the food required by the waiting table.

www.it-ebooks.info

The chef prepares the food and then returns the result to the waiter.



Here, the chef performs his function of cooking food. The chef accepted the parameters about what type of food to cook (three pepperoni pan pizzas). The chef then went away, did some work, and returned with three pizzas. Note that the waiter does not know and does not care about how the chef goes about cooking the pizzas. The chef abstracts away the process of cooking pizzas for the waiter, so cooking a pizza is just a simple, single-line command for the waiter. The waiter just wants his request to be completed and the pizzas returned to him.

When a function (chef) gets called with some arguments (types of pizzas to be prepared), the function performs some actions (preparing the pizzas) and optionally returns a result (the actual finished pizzas).

www.it-ebooks.info

An example of a <cmath> library function – sqrt()

Now, let's talk about a more practical example and relate it to the pizza example.

There is a function in the `<cmath>` library called the `sqrt()` function. Let me quickly illustrate its use, as shown in the following code:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double rootOf5 = sqrt( 5 ); // function call to the sqrt
  function
  cout << rootOf5  << endl;
}
```

So, `sqrt()` can find the mathematical square root of any number given to it.

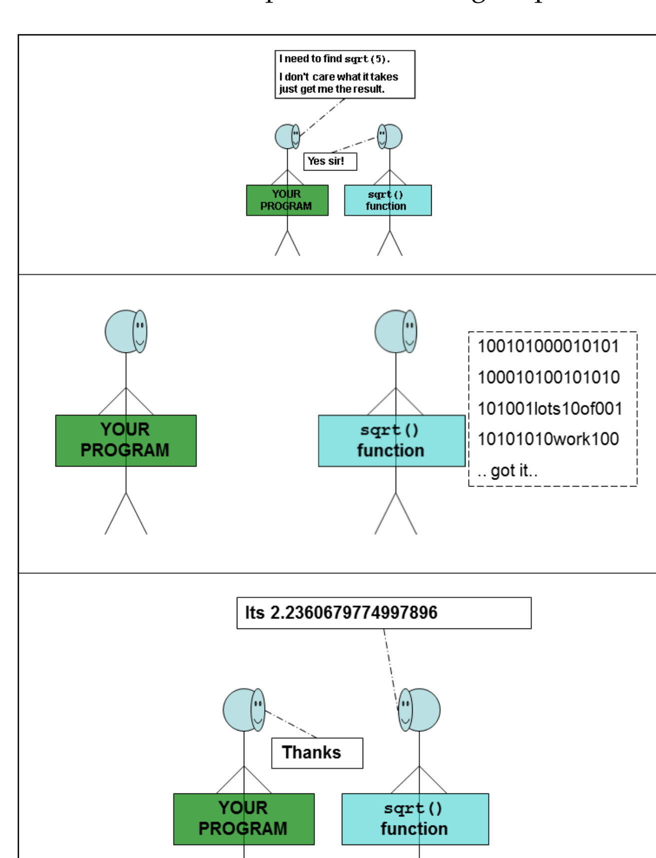
Do you know how to find the square root of a tough number such as 5? It's not simple. A clever soul sat down and wrote a function that can find the square roots of all types of numbers. Do you have to understand the math behind how the square root of 5 is found to use the `sqrt(5)` function call? Heck, no! So, just as the waiter didn't have to understand how to cook a pizza in order to get a pizza as the result, the caller of a C++ library function does not have to fully understand how that library function works internally to use it effectively.

The following are the advantages of using functions:

1. Functions abstract away a complex task into a simple, callable routine. This makes the code required to *cook a pizza*, for instance, just a single-line command for the caller (the caller is typically your program).
2. Functions avoid the repetition of code where it is not necessary. Say we have 20 or so lines of code that can find the square root of a double value. We wrap these lines of code into a callable function; instead of repeatedly copying and pasting these 20 lines of code, we simply call the `sqrt` function (with the number to root) function whenever we need a root.

www.it-ebooks.info

The following illustration shows the process of finding a square root:



Writing our own functions

Say, we want to write some code that prints out a strip of road, as shown here:

```
cout << "*   *" << endl;
cout << "* | *" << endl;
cout << "* | *" << endl;
cout << "*   *" << endl;
```


www.it-ebooks.info

Now, say we want to print two strips of road, in a row, or three strips of road. Or, say we want to print any number of strips of road. We will have to repeat the four lines of code that produce the first strip of road once per strip of road we're trying to print.

What if we introduced our own C++ command that allowed us to print a strip of road on being called the command. Here's how that will look:

```
void printRoad()
{
  cout << "*   *" << endl;
  cout << "* | *" << endl;
  cout << "* | *" << endl;
  cout << "*   *" << endl;
}
```

This is the definition of a function. A C++ function has the following anatomy:

function name

```
void printRoad()
{
  cout << "*   *" << endl;
  cout << "* | *" << endl;
  cout << "* | *" << endl;
  cout << "*   *" << endl;
}
```

function body

Using a function is simple: we simply invoke the function we want to execute by name, followed by two round brackets (). For example, calling the `printRoad()` function will cause the `printRoad()` function to run. Let's trace an example program to fully understand what this means.

A sample program trace

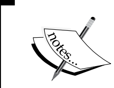
Here's a complete example of how a function call works:

```
#include <iostream>
using namespace std;
void printRoad()
{
```


www.it-ebooks.info

```
    cout << "*   *" << endl;
    cout << "* | *" << endl;
    cout << "* | *" << endl;
    cout << "*   *" << endl;
  }
  int main()
  {
    cout << "Program begin!" << endl;
    printRoad();
    cout << "Program end" << endl;
    return 0;
  }
```

Let's trace the program's execution from beginning to end. Remember that for all C++ programs, execution begins on the first line of `main()`.



> `main()` is also a function. It oversees the execution of the whole program. Once `main()` executes the `return` statement, your program ends.

When the last line of the `main()` function is reached, the program ends.

A line-by-line trace of the execution of the preceding program is shown as follows:

```
void printRoad()
{
  cout << "*   *" << endl;          // 3: then we jump up here
  cout << "* | *" << endl;          // 4: run this
  cout << "* | *" << endl;          // 5: and this
  cout << "*   *" << endl;          // 6: and this
}
int main()
{
  cout << "Program begin!" << endl; // 1: first line to execute
  printRoad();                      // 2: second line..
  cout << "Program end" << endl;    // 7: finally, last line
  return 0;                         // 8: and return to o/s
}
```

This is how the output of this program will look:

```
Program begin!
*   *
* | *
* | *
*   *
Program end
```


www.it-ebooks.info

Here's an explanation of the preceding code, line by line:

1. The program's execution begins on the first line of `main()`, which outputs `program begin!`.
2. The next line of code that is run is the call to `printRoad()`. What this does is it jumps the program counter to the first line of `printRoad()`. All the lines of `printRoad()` then execute in order (steps 3–6).
3. Finally, after the function call to `printRoad()` is complete, control returns to the `main()` statement. We then see `Program end` printed.

Don't forget the brackets after the function call to `printRoad()`. A function call must always be followed by round brackets (), otherwise the function call will not work and you will get a compiler error.

The following code is used to print four strips of road:

```
int main()
{
   printRoad();
   printRoad();
   printRoad();
   printRoad();
}
```

Alternatively, you can also use the following code:

```
for( int i = 0; i < 4; i++ )
printRoad();
```

So, instead of repeating the four lines of `cout` every time a box is printed, we simply call the `printRoad()` function to make it print. Also, if we want to change how a printed road looks, we have to simply modify the implementation of the `printRoad()` function.

Calling a function entails running the entire body of that function, line by line. After the function call is complete, the control of the program then resumes at the point of the function call.

www.it-ebooks.info

Exercise

As an exercise, find out what is wrong with the following code:

```
#include <iostream>
using namespace std;
void myFunction()
{
   cout << "You called?" << endl;
}
int main()
{
   cout << "I'm going to call myFunction now." << endl;
   myFunction;
}
```

Solution

The correct answer to this problem is that the call to `myFunction` (in the last line of `main()`) is not followed by round brackets. All function calls must be followed by round brackets. The last line of `main()` should read `myFunction();`, not just `myFunction`.

Functions with arguments

How can we extend the `printRoad()` function to print a road with a certain number of segments? The answer is simple. We can let the `printRoad()` function accept a parameter, called `numSegments`, to print a certain number of road segments.

The following code snippet shows how that will look:

```
void printRoad(int numSegments)
{
  // use a for loop to print numSegments road segments
  for( int i = 0; i < numSegments; i++)
  {
    cout << "*   *" << endl;
    cout << "* | *" << endl;
    cout << "* | *" << endl;
    cout << "*   *" << endl;
  }
}
```


www.it-ebooks.info

The following screenshot shows the anatomy of a function that accepts an argument:

parameter

```
void printRoad(int num)
{
```

Call this new version of `printRoad()`, asking it to print four segments, as follows:

```
printRoad( 4 );    // function call
```

The 4 between the brackets of the `function call` in the preceding statement gets assigned to the `numSegments` variable of the `printRoad(int numSegments)` function. This is how the value 4 gets passed to `numSegments`:

```
void printRoad(int numSegments)
{
```

= 4

The assignment of **numSegments** happens at function invokation time

An illustration of how printRoad(4) will assign the value 4 to the numSegments variable

So, `numSegments` gets assigned the value passed between the brackets in the call to `printRoad()`.

Functions that return values

An example of a function that returns a value is the `sqrt()` function. The `sqrt()` function accepts a single parameter between its brackets (the number to root) and returns the actual root of the number.

Here's an example usage of the `sqrt` function:

```
cout << sqrt( 4 ) << endl;
```

The `sqrt()` function does something analogous to what the chef did when preparing the pizzas.

As a caller of the function, you do not care about what goes on inside the body of the `sqrt()` function; that information is irrelevant since all you want is the result of the square root of the number that you are passing.

www.it-ebooks.info

Let's declare our own simple function that returns a value, as shown in the following code:

```
int sum(int a, int b)
{
  return a + b;
}
```

The following screenshot shows the anatomy of a function with parameters and a returned value:

return type   parameters

```
int sum( int a, int b )
{
        return a + b;
}
```

The `sum` function is very basic. All it does is take two `int` numbers `a` and `b`, sums them up together, and returns a result. You might say that we don't even need an entire function just to add two numbers. You're right, but bear with me for a moment. We will use this simple function to explain the concept of returned values.

You will use the `sum` function in this way (from `main()`):

```
int sum( int a, int b )
{
  return a + b;
}
int main()
{
  cout << "The sum of 5 and 6 is " << sum( 5,6 ) << endl;
}
```

For the `cout` command to complete, the `sum( 5,6 )` function call must be evaluated. At the point where the `sum( 5,6 )` function call occurs, the returned value from `sum( 5,6 )` is put right there.

www.it-ebooks.info

In other words, this is the line of code that `cout` actually sees after evaluating the `sum( 5,6 )` function call:

```
cout << "The sum of 5 and 6 is " << 11 << endl;
```

The returned value from `sum( 5,6 )` is effectively cut and pasted at the point of the function call.

A value must always be returned by a function that promises to do so (if the return type of the function is anything other than `void`).

Exercises

1. Write an `isPositive` function that returns `true` when the double parameter passed to it is indeed positive.
2. Complete the following function definition:

   ```
   // function returns true when the magnitude of 'a'
   // is equal to the magnitude of 'b' (absolute value)
   bool absEqual(int a, int b)
   {
       // to complete this exercise, try to not use
       // cmath library functions
   }
   ```

3. Write a `getGrade()` function that accepts an integer value (marks out of 100) and returns the grade (either A, B, C, D, or F).
4. A mathematical function is of the form `f(x) = 3x + 4`. Write a C++ function that returns values for `f(x)`.

Solutions

1. The `isPositive` function accepts a double parameter and returns a boolean value:

   ```
   bool isPositive( double value )
   {
     return value > 0;
   }
   ```

2. The following is the completed `absEqual` function:

   ```
   bool absEqual( int a, int b )
   {
     // Make a and b positive
   if( a < 0 )
   ```


www.it-ebooks.info

```
    a = -a;
  if( b < 0 )
    b = -b;
  // now since they're both +ve,
  // we just have to compare equality of a and b together
  return a == b;
}
```

3. The `getGrade()` function is given in the following code:

   ```
   char getGrade( int grade )
   {
     if( grade >= 80 )
       return 'A';
     else if( grade >= 70 )
       return 'B';
     else if( grade >= 60 )
       return 'C';
     else if( grade >= 50 )
       return 'D';
     else
       return 'F';
   }
   ```

4. This program is a simple one that should entertain you. The origin of the `name` function in C++ actually came from the math world, as shown in the following code:

   ```
   double f( double x )
   {
     return 3*x + 4;
   }
   ```

Variables, revisited

It's always nice to revisit a topic you've covered before, now that you understand C++ coding much more in depth.

Global variables

Now that we've introduced the concept of functions, the concept of a global variable can be introduced.

www.it-ebooks.info

What is a global variable? A global variable is any variable that is made accessible to all the functions of the program. How can we make a variable that is accessible to all the functions of the program? We simply declare the global variable at the top of the code file, usually after or near the `#include` statements.

Here's an example program with some global variables:

```
#include <iostream>
#include <string>
using namespace std;

string g_string; // global string variable,
// accessible to all functions within the program
// (because it is declared before any of the functions
// below!)

void addA(){ g_string += "A"; }
void addB(){ g_string += "B"; }
void addC(){ g_string += "C"; }

int main()
{
  addA();
  addB();
  cout << g_string << endl;
  addC();
  cout << g_string << endl;
}
```

Here, the same `g_string` global variable is accessible to all the four functions in the program (`addA()`, `addB()`, `addC()`, and `main()`). Global variables live for the duration of the program.

People sometimes prefer to prefix global variables with `g_`, but prefixing the variable name with `g_` is not a requirement for a variable to be global.

Local variables

A local variable is a variable that is defined within a block of code. Local variables go out of the scope at the end of the block in which they are declared. Some examples will follow in the next section, *The scope of a variable*.

www.it-ebooks.info

The scope of a variable

The scope of a variable is the area of code where that variable can be used. The scope of any variable is basically the block in which it is defined. We can demonstrate a variable's scope using an example, as shown in the following code:

```
int g_int; // global int, has scope until end of file
void func( int arg )
{
  int fx;
} // </fx> dies, </arg> dies

int main()
{
  int x; // variable <x> has scope starting here..
         // until the end of main()
  if( x == 0 )
  {
    int y;  // variable <y> has scope starting here,
            // until closing brace below
  } // </y> dies
  if( int x2 = x ) // variable <x2> created and set equal to <x>
  {
    // enter here if x2 was nonzero
  } // </x2> dies

for( int c = 0; c < 5; c++ ) // c is created and has
  { // scope inside the curly braces of the for loop
    cout << c << endl;
  } // </c> dies only when we exit the loop
} // </x> dies
```

The main thing that defines a variable's scope is a block. Let's discuss the scope of a couple of the variables defined in the preceding code example:

- `g_int`: This is a global integer with a scope that ranges from the point it was declared until the end of the code file. That is to say, `g_int` can be used inside `func()` and `main()`, but it cannot be used in other code files. To have a single global variable that is used across multiple code files, you will need an external variable.
- `arg` (the argument of `func()`): This can be used from the first line of `func()` (after the opening curly brace {) to the last line of `func()` (until the closing curly brace }).

www.it-ebooks.info

- `fx`: This can be used anywhere inside `func()` until the closing curly brace of `func()`}.
- `main()`(variables inside `main()`): This can be used as marked in the comments.

Notice how variables declared inside the brackets of a function's argument list can only be used inside the block below that function's declaration. For example, the `arg` variable passed to `func()`:

```
void func( int arg )
{
  int fx;
} // </fx> dies, </arg> dies
```

The `arg` variable will die after the closing curly brace (}) of the `func()` function. This is counterintuitive as the round brackets are technically outside the curly braces that define the { block }.

The same goes for variables declared inside the round brackets of a `for` loop. Take as an example the following `for` loop:

```
for( int c = 0; c < 5; c++ )
{
  cout << c << endl;
} // c dies here
```

The `int c` variable can be used inside the round brackets of the `for` loop declaration or inside the block below the `for` loop declaration. The `c` variable will die after the closing of the curly brace of the `for` loop it is declared in. If you want the `c` variable to live on after the brace brackets of the `for` loop, you need to declare the `c` variable before the `for` loop, as shown here:

```
int c;
for( c = 0; c < 5; c++ )
{
  cout << c << endl;
} // c does not die here
```


www.it-ebooks.info

Static local variables

The `static` local variables are exactly like global variables, only they have a local scope, as shown in the following code:

```
void testFunc()
{
  static int runCount = 0; // this only runs ONCE, even on
  // subsequent calls to testFunc()!
  cout << "Ran this function " << ++runCount << " times" << endl;
} // runCount stops being in scope, but does not die here

int main()
{
  testFunc();  // says 1 time
  testFunc();  // says 2 times!
}
```

With the use of the `static` keyword inside `testFunc()`, the `runCount` variable remembers its value between calls of `testFunc()`. So, the output of the two separate preceding runs of `testFunc()` is:

```
Ran this function 1 times
Ran this function 2 times
```

That's because static variables are only created and initialized once (the first time when the function they are declared in runs), and after that, the static variable retains its old value. Say, we declare `runCount` as a regular, local, nonstatic variable:

```
int runCount = 0; // if declared this way, runCount is local
```

Then, this is how the output will look:

```
Ran this function 1 times
Ran this function 1 times
```

Here, we see `testFunc` saying `Ran this function 1 time` both the times.
As a local variable, the value of `runCount` is not retained between function calls.

You should not overuse static local variables. In general, you should only use a static local variable when it is absolutely necessary.

www.it-ebooks.info

Const variables

A `const` variable is a variable whose value you promise the compiler not to change after the first initialization. We can declare one simply, for example, for the value of `pi`:

```
const double pi = 3.14159;
```

Since `pi` is a universal constant (one of the few things you can rely on to be the same), there should be no need to change `pi` after initialization. In fact, changes to `pi` should be forbidden by the compiler. Try, for example, to assign `pi` a new value:

```
pi *= 2;
```

We will get the following compiler error:

```
error C3892: 'pi' : you cannot assign to a variable that is const
```

This error makes perfect sense, because besides the initial initialization, we should not be able to change the value of `pi`— a variable that is constant.

Function prototypes

A function prototype is the signature of the function without the body. For example, let's prototype the `isPositive`, `absEqual`, and `getGrade` functions from the following exercises:

```
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

Notice how the function prototypes are just the return type, function name, and argument list that the function requires. Function prototypes don't get a body. The body of the function is typically put in the `.cpp` file.

.h and .cpp files

It is typical to put your function prototypes in an `.h` file and the bodies of the functions in a `.cpp` file. The reason for this is you can include your `.h` file in a bunch of `.cpp` files and not get multiple definition errors.

www.it-ebooks.info

The following screenshot gives you a clear picture of `.h` and `.cpp` files:.

main.cpp

```
#include <iostream>
using namespace std;

#include "prototypes.h"

int main()
{
    cout << boolalpha
<< isPositive( 4 )
<< endl;
    cout
<< absEqual( 4, -4 )
<< endl;
}
```

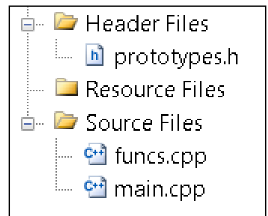
prototypes.h

```
// Make sure these prototypes
// only included in compilatio
#ifndef PROTOTYPES_H // includ
#define PROTOTYPES_H

bool isPositive( double value
bool absEqual( int a, int b );
char getGrade( int grade );

#endif
```

funcs.cpp

```
#include "prototypes.h"

bool isPositive( double value )
{
    return value > 0;
}

bool absEqual( int a, int b )
{
    // Make a and b positive
    if( a < 0 )
        a = -a;
    if( b < 0 )
        b = -b;
    // now since they're both +ve,
    // we just have to compare equality of
    return a == b;
}

char getGrade( int grade )
{
    if( grade >= 80 )
        return 'A';
```

Here, we have three files in this Visual C++ project:



prototypes.h contains

```
// Make sure these prototypes are
// only included in compilation ONCE
#pragma once
extern int superglobal; // extern: variable "prototype"
// function prototypes
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

The `prototypes.h` file contains function prototypes. We will explain what the `extern` keyword does in a few paragraphs.

www.it-ebooks.info

funcs.cpp contains

```
#include "prototypes.h" // every file that uses isPositive,
// absEqual or getGrade must #include "prototypes.h"
int superglobal; // variable "implementation"
// The actual function definitions are here, in the .cpp file
bool isPositive( double value )
{
  return value > 0;
}
bool absEqual( int a, int b )
{
  // Make a and b positive
  if( a < 0 )
    a = -a;
  if( b < 0 )
    b = -b;
  // now since they're both +ve,
  // we just have to compare equality of a and b together
  return a == b;
}
char getGrade( int grade )
{
  if( grade >= 80 )
    return 'A';
  else if( grade >= 70 )
    return 'B';
  else if( grade >= 60 )
    return 'C';
  else if( grade >= 50 )
    return 'D';
  else
    return 'F';
}
```

main.cpp contains

```
#include <iostream>
using namespace std;
#include "prototypes.h" // for use of isPositive, absEqual
// functions
int main()
{
  cout << boolalpha << isPositive( 4 ) << endl;
  cout << absEqual( 4, -4 ) << endl;
}
```


www.it-ebooks.info

When you split up the code into .h and .cpp files, the .h file (the header file) is called the interface and the .cpp file (the one with the actual functions in it) is called the implementation.

The puzzling part at first for some programmers is how does C++ know where the isPositive and getGrade function bodies are, if we only #include the prototypes? Shouldn't we #include the funcs.cpp file into main.cpp too?

The answer is *magic*. You only need to #include the prototypes.h header file in both main.cpp and funcs.cpp. As long as both .cpp files are included in your C++ **Integrated Development Environment** (**IDE**) project (that is, they appear in the **Solution Explorer** tree view at the left-hand side), the linkup of the prototypes to the function bodies is done automatically by the compiler.

Extern variables

An extern declaration is similar to a function prototype, only it is used on a variable. You can put an extern global variable declaration in an .h file, and include this .h file in a whole bunch of other files. This way, you can have a single global variable that gets shared across multiple source files, without getting the multiply defined symbols found linker error. You'd put the actual variable declaration in a .cpp file so that the variable only gets declared once. There is an extern variable in the prototypes.h file in the previous example.

Macros

C++ macros are from a class of C++ commands called preprocessor directives. A preprocessor directive is executed before compilation takes place.

Macros start with #define. For example, say we have the following macro:

```
#define PI 3.14159
```

At the lowest level, macros are simply copy and paste operations that occur just before compile time. In the preceding macro statement, the 3.14159 literal will be copied and pasted everywhere the symbol PI occurs in the program.

Take an example of the following code:

```
#include <iostream>
using namespace std;
#define PI 3.14159
int main()
{
```


www.it-ebooks.info

```
    double r = 4;
    cout << "Circumference is " << 2*PI*r << endl;
  }
```

What the C++ preprocessor will do is first go through the code and look for any usage of the PI symbol. It will find one such usage on this line:

```
cout << "Circumference is " << 2*PI*r << endl;
```

The preceding line will convert to the following just before compilation:

```
cout << "Circumference is " << 2*3.14159*r << endl;
```

So, all that happens with a #define statement is that all the occurrences of the symbol used (example, PI) are replaced by the literal number 3.14159 even before compilation occurs. The point of using macros in this way is to avoid hardcoding numbers into the code. Symbols are typically easier to read than big, long numbers.

Advice – try to use const variables where possible

You can use macros to define constant variables. You can also use const variable expressions instead. So, say we have the following line of code:

```
#define PI 3.14159
```

We will be encouraged to use the following instead:

```
const double PI = 3.14159;
```

Using a const variable will be encouraged because it stores your value inside an actual variable. The variable is typed, and typed data is a good thing.

Macros with arguments

We can also write macros that accept arguments. Here's an example of a macro with an argument:

```
#define println(X) cout << X << endl;
```

What this macro will do is every time println("Some value") is encountered in the code, the code on the right-hand side (cout << "Some value" << endl) will be copied and pasted on the console. Notice how the argument between the brackets is copied in the place of X. Say we had the following line of code:

```
println( "Hello there" )
```


www.it-ebooks.info

This will be replaced by the following statement:

```
cout << "Hello there" << endl;
```

Macros with arguments are exactly like very short functions. Macros cannot contain any newline characters in them.

Advice – use inline functions instead of macros with arguments

You have to know about how macros with arguments work because you will encounter them in C++ code a lot. Whenever possible, however, many C++ programmers prefer to use inline functions over macros with arguments.

A normal function call execution involves a `jump` instruction to the function and then the execution of the function. An inline function is one whose lines of code are copied to the point of the function call and no jump is issued. Using inline functions usually makes sense for very small, simple functions that don't have a lot of lines of code. For example, we might inline a simple function `max` that finds the larger of two values:

```
inline int max( int a, int b )
{
  if( a > b ) return a;
  else return b;
}
```

Everywhere this `max` function is used, the code for the function body will be copied and pasted at the point of the function call. Not having to `jump` to the function saves execution time, making inline functions effectively similar to macros.

There is a catch to using inline functions. Inline functions must have their bodies completely contained in the `.h` header file. This is so that the compiler can make optimizations and actually inline the function wherever it is used. Functions are made inline typically for speed (since you don't have to jump to another body of the code to execute the function) but at the cost of code bloat.

The following are the reasons why inline functions are preferred over macros:

1. Macros are error prone: the argument to the macro is not typed.
2. Macros have to be written in one line or you will see them using use escaped
   ```
   \
   newline characters \
   like this \
   which is hard to read
   ```


www.it-ebooks.info

3. If the macro is not carefully written, it will result in difficult-to-fix compiler errors. For example, if you do not bracket your argument properly, your code will just be wrong.
4. Large macros are hard to debug.

It should be said that macros do allow you to perform some preprocessor compiler magic. UE4 makes a lot of use of macros with arguments, as you will see later.

Summary

Function calls allow you to reuse basic code. Code reuse is important for a number of reasons: mainly because programming is hard and duplicating effort should be avoided wherever possible. The efforts of the programmer that wrote the `sqrt()` function do not need to be repeated by other programmers who want to solve the same problem.

www.it-ebooks.info

www.it-ebooks.info

6
Objects, Classes, and Inheritance

In the previous chapter, we discussed functions as a way to bundle up a bunch of lines of related code. We talked about how functions abstracted away implementation details and how the `sqrt()` function does not require you to understand how it works internally to use it to find roots. This was a good thing, primarily because it saved the programmer time and effort, while making the actual work of finding square roots easier. This principle of *abstraction* will come up again here when we discuss objects.

In a nutshell, objects tie together methods and their related data into a single structure. This structure is called a *class*. The main idea of using objects is to create a code representation for every thing inside your game. Every object represented in the code will have data and associated functions that operate on that data. So you'd have an *object* to represent your player instance and related functions that make the player `jump()`, `shoot()`, and `pickupItem()` functions. You'd also have an object to represent every monster instance and related functions such as `growl()`, `attack()`, and possibly `follow()`.

Objects are types of variables, though, and objects will stay in memory as long as you keep them there. You create an object instance once when the thing in your game it represents is created, and you destroy the object instance when the thing in your game it represents dies.

Objects can be used to represent in-game things, but they can also be used to represent any other type of thing. For example, you can store an image as an object. The data fields will be the image's width of the image, its height, and the collection of pixels inside it. C++ strings are also objects.

www.it-ebooks.info

This chapter contains a lot of keywords that might be difficult to grasp at first, including `virtual` and `abstract`.

Don't let the more difficult sections of this chapter bog you down. I included descriptions of many advanced concepts for completeness. However, bear in mind that you don't need to completely understand everything in this chapter to write working C++ code in UE4. It helps to understand it, but if something doesn't make sense, don't get stuck. Give it a read and then move on. Probably what will happen is you will not get it at first, but remember a reference to the concept in question when you're coding. Then, when you open this book up again, "voilà!" It will make sense.

struct objects

An object in C++ is basically any variable type that is made up of a conglomerate of simpler types. The most basic object in C++ is `struct`. We use the `struct` keyword to glue together a bunch of smaller variables into one big variable. If you recall, we did introduce `struct` briefly in *Chapter 2*, *Variables and Memory*. Let's revise that simple example:

```
struct Player
{
  string name;
  int hp;
};
```

This is the structure definition for what makes a `Player` object. The player has a `string` for his `name` and an integer for his `hp` value.

If you'll recall from *Chapter 2*, *Variables and Memory*, the way we make an instance of the `Player` object is like this:

```
Player me;    // create an instance of Player, called me
```

From here, we can access the fields of the `me` object like so:

```
me.name = "Tom";
me.hp = 100;
```


www.it-ebooks.info

Member functions

Now, here's the exciting part. We can attach member functions to the `struct` definition simply by writing these functions inside the `struct Player` definition.

```
struct Player
{
  string name;
  int hp;
  // A member function that reduces player hp by some amount
  void damage( int amount )
  {
    hp -= amount;
  }
  void recover( int amount )
  {
    hp += amount;
}
};
```

A member function is just a C++ function that is declared inside a `struct` or `class` definition. Isn't that a great idea?

There is a bit of a funny idea here, so I'll just come out and say it. The variables of `struct Player` are accessible to all the functions inside `struct Player`. Inside each of the member functions of `struct Player`, we can actually access the `name` and `hp` variables as if they were local to the function. In other words, the `name` and `hp` variables of `struct Player` are shared between all the member functions of `struct Player`.

The this keyword

In some C++ code (in later chapters), you will see more references to the `this` keyword. The `this` keyword is a pointer that refers to the current object. Inside the `Player::damage()` function, for example, we can write our reference to `this` explicitly:

```
void damage( int amount )
{
  this->hp -= amount;
}
```

The `this` keyword only makes sense inside a member function. We could explicitly include use of keyword `this` inside member functions, but without writing `this`, it is implied that we are talking about the `hp` of the current object.

www.it-ebooks.info

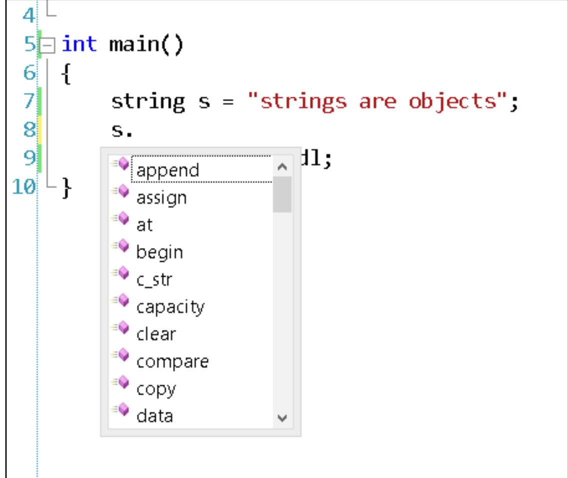
Strings are objects?

Yes! Every time you've used a string variable in the past, you were using an object. Let's try out some of the member functions of the `string` class.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string s = "strings are objects";
  s.append( "!!" ); // add on "!!" to end of the string!
  cout << s << endl;
}
```

What we've done here is use the `append()` member function to add on two extra characters to the end of the string (`!!`). Member functions always apply to the object that calls the member function (the object to the left of the dot).

> To see the listing of members and member functions available on an object, type the object's variable name in Visual Studio, then a dot (`.`), then press *Ctrl* and spacebar. A member listing will pop up.



Pressing *Ctrl* and spacebar will make the member listing appear

www.it-ebooks.info

Invoking a member function

Member functions can be invoked with the following syntax:

```
objectName.memberFunction();
```

The object invoking the member function is on the left of the dot. The member function to call is on the right of the dot. A member function invocation is always followed by round brackets `()`, even when no arguments are passed to the brackets.

So, in the part of the program where the monster attacks, we can reduce the player's `hp` value like so:

```
player.damage( 15 );  // player takes 15 damage
```

Which isn't that more readable than the following:

```
player.hp -= 15;      // player takes 15 damage
```

When member functions and objects are used effectively, your code will read more like prose or poetry than a bunch of operator symbols slammed together.

Besides beauty and readability, what is the point of writing member functions? Outside the `Player` object, we can now do more with a single line of code than just reduce the `hp` member by `15`. We can also do other things as we're reducing the player's `hp`, such as take into account the player's armor, check whether the player is invulnerable, or have other effects occur when the player is damaged. What happens when the player is damaged should be abstracted away by the `damage()` function.

Now think if the player had an armor class. Let's add a field to `struct Player` for armor class:

```
struct Player
{
  string name;
  int hp;
  int armorClass;
};
```

We'd need to reduce the damage received by the player by the armor class of the player. So we'd type a formula now to reduce `hp`. We can do it the non-object-oriented way by accessing the data fields of the `player` object directly:

```
player.hp -= 15 - player.armorClass; // non OOP
```


www.it-ebooks.info

Otherwise, we can do it the object-oriented way by writing a member function that changes the data members of the `player` object as needed. Inside the `Player` object, we can write a member function `damage()`:

```
struct Player
{
  string name;
  int hp;
  int armorClass;
void damage( int dmgAmount )
  {
    hp -= dmgAmount - armorClass;
  }
};
```

Exercises

1. There is a subtle bug in the player's `damage` function in the preceding code. Can you find and fix it? Hint: What happens if the damage dealt is less than `armorClass` of the player?
2. Having only a number for armor class doesn't give enough information about the armor! What is the armor's name? What does it look like? Devise a `struct` function for the Player's armor with fields for name, armor class, and durability rating.

Solutions

The solution is in the `struct` player code listed in the next section, *Privates and encapsulation*.

How about using the following code:

```
struct Armor
{
  string name;
  int armorClass;
  double durability;
};
```

An instance of `Armor` will then be placed inside `struct Player`:

```
struct Player
{
  string name;
  int hp;
  Armor armor; // Player has-an Armor
};
```


www.it-ebooks.info

This means the player has an armor. Keep this in mind—we'll explore `has-a` versus `is-a` relationships later.

Privates and encapsulation

So now we've defined a couple of member functions, whose purpose it is to modify and maintain the data members of our `Player` object, but some people have come up with an argument.

The argument is as follows:

- An object's data members should only ever be accessed only through its member functions, never directly.

This means that you should never access an object's data members from outside the object directly, in other words, modify the player's `hp` directly:

```
player.hp -= 15 - player.armorClass; // bad: direct member access
```

This should be forbidden, and users of the class should be forced to use the proper member functions instead to change the values of data members:

```
player.damage( 15 );    // right: access thru member function
```

This principle is called *encapsulation*. Encapsulation is the concept that every object should be interacted via its member functions only. Encapsulation says that raw data members should never be accessed directly.

The reasons behind encapsulation are:

- **To make the class self contained**: The primary idea behind encapsulation is that objects work best when they are programmed such that they manage and maintain their own internal state variables without a need for code outside the class to examine that class' private data. When objects are coded this way, it makes the object much easier to work with, that is, easier to read and maintain. To make the player object jump, you should just have to call `player.jump();` let the player object manage state changes to its `y-height` position (making the player jump!). When an object's internal members are not exposed, interacting with that object is much easier and more efficient. Interact only with an object's public member functions; let the object manage its internal state (we will explain the keywords `private` and `public` in a moment).

www.it-ebooks.info

- **To avoid breaking code**: When code outside of a class interacts with that class' public member functions only (the class' public interface), then an object's internal state management is free to change, without breaking any of the calling code. This way, if an object's internal data members change for any reason, all code using the object still remains valid as long as the member functions remain the same.

So how can we prevent the programmer from doing the wrong thing and accessing data members directly? C++ introduces the concept of *access modifiers* to prevent access of an object's internal data.

Here is how we'd use access modifiers to forbid access to certain sections of `struct Player` from outside of `struct Player`.

The first thing you'd do is decide which sections of the `struct` definition you want to be accessible outside of the class. These section will be labelled `public`. All other regions that will not be accessible outside of `struct` will be labelled `private`, as follows:

```
struct Player
{
private:        // begins private section.. cannot be accessed
                // outside the class until
  string name;
  int hp;
  int armorClass;
public:         //  until HERE. This begins the public section
  // This member function is accessible outside the struct
  // because it is in the section marked public:
  void damage( int amount )
  {
    int reduction = amount - armorClass;
    if( reduction < 0 ) // make sure non-negative!
      reduction = 0;
    hp -= reduction;
  }
};
```

Some people like it public

Some people do unabashedly use `public` data members and do not encapsulate their objects. This is a matter of preference, though considered as bad object-oriented programming practice.

www.it-ebooks.info

However, classes in UE4 do use `public` members sometimes. It's a judgment call; whether a data member should be `public` or `private` is really up to the programmer.

With experience, you will find that sometimes you get into a situation that requires quite a bit of refactoring when you make a data member `public` that should have been `private`.

class versus struct

You might have seen a different way of declaring an object, using the `class` keyword, instead of `struct`, as shown in the following code:

```
class Player // we used class here instead of struct!
{
  string name;
  //
};
```

The `class` and `struct` keywords in C++ are almost identical. There is only one difference between `class` and `struct`, and it is that the data members inside a `struct` keyword will be declared `public` by default, while in a `class` keyword the data members inside the class will be declared `private` by default. (This is why I introduced objects using `struct`; I didn't want to put `public` inexplicably as the first line of `class`.)

In general, `struct` is preferred for simple types that don't use encapsulation, don't have many member functions, and must be backward compatible with C. Classes are used almost everywhere else.

From now on, let's use the `class` keyword instead of `struct`.

Getters and setters

You might have noticed that once we slap `private` onto the `Player` class definition, we can no longer read or write the name of the player from outside the `Player` class.

If we try and read the name with the following code:

```
Player me;
cout << me.name << endl;
```

Or write to the name, as follows:

```
me.name = "William";
```


www.it-ebooks.info

Using the `struct Player` definition with `private` members, we will get the following error:

```
main.cpp(24) : error C2248: 'Player::name' : cannot access private
member declared in class 'Player'
```

This is just what we asked for when we labeled the `name` field `private`. We made it completely inaccessible outside the `Player` class.

Getters

A getter (also known as an accessor function) is used to pass back copies of internal data members to the caller. To read the player's name, we'd deck out the `Player` class with a member function specifically to retrieve a copy of that `private` data member:

```
class Player
{
private:
  string name;  // inaccessible outside this class!
                //  rest of class as before
public:
  // A getter function retrieves a copy of a variable for you
  string getName()
{
  return name;
}
};
```

So now it is possible to read the player's name information. We can do this by using the following code statement:

```
cout << player.getName() << endl;
```

Getters are used to retrieve `private` members that would otherwise be inaccessible to you from outside the class.

www.it-ebooks.info

Real world tip–Keyword const

Inside a class, you can add the `const` keyword to a member function declaration. What the `const` keyword does is promises to the compiler that the internal state of the object will not change as a result of running this function. Attaching the `const` keyword will look something like this:

```
string getName() const
{
  return name;
}
```

No assignments to data members can happen inside a member function that is marked `const`. As the internal state of the object is guaranteed not to change as a result of running a `const` function, the compiler can make some optimizations around function calls to `const` member functions.

Setters

A setter (also known as a modifier function or mutator function) is a member function whose sole purpose is to change the value of an internal variable inside the class, as shown in the following code:

```
class Player
{
private:
  string name;  // inaccessible outside this class!
                //  rest of class as before
public:
  // A getter function retrieves a copy of a variable for you
  string getName()
{
  return name;
}
void setName( string newName )
{
  name = newName;
}
};
```

So we can still change the `private` function of a `class` from outside the `class` function, but only if we do so through a setter function.

www.it-ebooks.info

But what's the point of get/set operations?

So the first question that crosses a newbie programmer's mind when he first encounters get/set operations on `private` members is, isn't get/set self-defeating? I mean, what's the point in hiding access to data members when we're just going to expose that same data again in another way? It's like saying, "You can't have any chocolates because they are private, unless you say please `getMeTheChocolate()`. Then, you can have the chocolates."

Some expert programmers even shorten the get/set functions to one liners, like this:

```
string getName(){ return name; }
void setName( string newName ){ name = newName; }
```

Let's answer the question. Doesn't a get/set pair break encapsulation by exposing the data completely?

The answer is twofold. First, get member functions typically only return a copy of the data member being accessed. This means that the original data member's value remains protected and is not modifiable through a `get()` operation.

`Set()` (mutator method) operations are a little bit counterintuitive though. If the setter is a `passthru` operation, such as `void setName( string newName ) { name=newName; }`, then having the setter might seem pointless. What is the advantage of using a mutator method instead of overwriting the variable directly?

The argument for using mutator methods is to write additional code before the assignment of a variable to guard the variable from taking on incorrect values. Say, for example, we have a setter for the `hp` data member, which will look like this:

```
void setHp( int newHp )
{
  // guard the hp variable from taking on negative values
  if( newHp < 0 )
  {
    cout << "Error, player hp cannot be less than 0" << endl;
    newHp = 0;
  }
  hp = newHp;
}
```

The mutator method is supposed to prevent the internal `hp` data member from taking on negative values. You might consider mutator methods a bit retroactive. Should the responsibility lie with the calling code to check the value it is setting before calling `setHp( -2 )`, and not let that only get caught in the mutator method? Can't you use a `public` member variable and put the responsibility for making sure the variable doesn't take on invalid values in the calling code, instead of in the setter? You can.

www.it-ebooks.info

However, this is the core of the reason behind using mutator methods. The idea behind mutator methods is so that the calling code can pass any value it wants to the `setHp` function (for example, `setHp( -2 )`), without having to worry whether the value it is passing to the function is valid or not. The `setHp` function then takes the responsibility of ensuring that the value is valid for the `hp` variable.

Some programmers consider direct mutator functions such as `getHp()`/`setHp()` a code smell. A code smell is in general a bad programming practice that people don't overtly take notice of, except for a niggling feeling that something is being done suboptimally. They argue that higher-level member functions can be written instead of mutators. For example, instead of a `setHp()` member function, we should have `public` member functions such as `heal()` and `damage()` instead. An article on this topic is available at `http://c2.com/cgi/wiki?AccessorsAreEvil`.

Constructors and destructors

The constructor in your C++ code is a simple little function that runs once when the C++ object is first created. The destructor runs once when the C++ object is destroyed. Say we have the following program:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
private:
  string name;  // inaccessible outside this class!
public:
  string getName(){ return name; }
// The constructor!
  Player()
  {
    cout << "Player object constructed" << endl;
    name = "Diplo";
  }
  // ~Destructor (~ is not a typo!)
  ~Player()
  {
    cout << "Player object destroyed" << endl;
  }
};

int main()
  {
```


www.it-ebooks.info

```
    Player player;
    cout << "Player named '" << player.getName() << "'" << endl;
  }
  // player object destroyed here
```

So here we have created a `Player` object. The output of this code will be as follows:

```
Player object constructed
Player named 'Diplo'
Player object destroyed
```

The first thing that happens during object construction is that the constructor actually runs. This prints the line `Player object constructed`. Following this, the line with the player's name gets printed: `Player named 'Diplo'`. Why is the player named *Diplo*? Because that is the name assigned in the `Player()` constructor.

Finally, at the end of the program, the player destructor gets called, and we see `Player object destroyed`. The player object gets destroyed when it goes out of scope at the end of `main()` (at } of `main`).

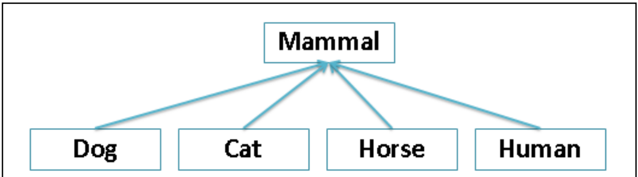
So what are constructors and destructors good for? Exactly what they appear to be for: setting up and tearing down of an object. The constructor can be used for initialization of data fields and the destructor to call delete on any dynamically allocated resources (we haven't covered dynamically allocated resources yet, so don't worry about this last point yet).

Class inheritance

You use inheritance when you want to create a new, more functional class of code, based on some existing class of code. Inheritance is a tricky topic to cover. Let's start with the concept of a *derived class* (or subclass).

Derived classes

The most natural way to consider inheritance is by analogy with the animal kingdom. The classification of living things is shown in the following screenshot:



www.it-ebooks.info

What this diagram means is that **Dog**, **Cat**, **Horse,** and **Human** are all **Mammals**. What that means is that dog, cat, horse, and human all share some common characteristics, such as having common organs (brain with neocortex, lungs, liver, and uterus in females), while being completely different in other regard. How each walks is different. How each talks is also different.

What does that mean if you were coding creatures? You would only have to program the common functionality once. Then, you would implement the code for the different parts specifically for each of the dog, cat, horse and human classes.

A concrete example of the preceding figure is as follows:

```
#include <iostream>
using namespace std;
class Mammal
{
protected:
  // protected variables are like privates: they are
  // accessible in this class but not outside the class.
  // the difference between protected and private is
  // protected means accessible in derived subclasses also
int hp;
  double speed;

public:
  // Mammal constructor - runs FIRST before derived class ctors!
Mammal()
{
  hp = 100;
  speed = 1.0;
  cout << "A mammal is created!" << endl;
}
~Mammal()
{
  cout << "A mammal has fallen!" << endl;
}
// Common function to all Mammals and derivatives
  void breathe()
  {
    cout << "Breathe in.. breathe out" << endl;
  }
  virtual void talk()
  {
    cout << "Mammal talk.. override this function!" << endl;
  }
```


www.it-ebooks.info

```
  // pure virtual function, (explained below)
  virtual void walk() = 0;
};

// This next line says "class Dog inherits from class Mammal"
class Dog : public Mammal // : is used for inheritance
{
public:
  Dog()
  {
cout << "A dog is born!" << endl;
}
~Dog()
{
  cout << "The dog died" << endl;
}
  virtual void talk() override
  {
    cout << "Woof!" << endl; // dogs only say woof!
  }
  // implements walking for a dog
  virtual void walk() override
  {
    cout << "Left front paw & back right paw, right front paw &
    back left paw.. at the speed of " << speed << endl;
  }
};

class Cat : public Mammal
{
public:
  Cat()
  {
    cout << "A cat is born" << endl;
  }
  ~Cat()
  {
    cout << "The cat has died" << endl;
  }
virtual void talk() override
  {
    cout << "Meow!" << endl;
  }
// implements walking for a cat.. same as dog!
```


www.it-ebooks.info

```
  virtual void walk() override
  {
    cout << "Left front paw & back right paw, right front paw &
    back left paw.. at the speed of " << speed << endl;
  }
};

class Human : public Mammal
{
// Data member unique to Human (not found in other Mammals)
  bool civilized;
public:
  Human()
  {
    cout << "A new human is born" << endl;
    speed = 2.0; // change speed. Since derived class ctor
    // (ctor is short for constructor!) runs after base
    // class ctor, initialization sticks initialize member
    // variables specific to this class
    civilized = true;
  }
  ~Human()
  {
    cout << "The human has died" << endl;
  }
  virtual void talk() override
  {
    cout << "I'm good looking for a .. human" << endl;
  }
// implements walking for a human..
  virtual void walk() override
  {
    cout << "Left, right, left, right at the speed of " << speed
    << endl;
  }
  // member function unique to human derivative
  void attack( Human & other )
  {
    // Human refuses to attack if civilized
    if( civilized )
      cout << "Why would a human attack another? Je refuse" <<
      endl;
    else
      cout << "A human attacks another!" << endl;
```

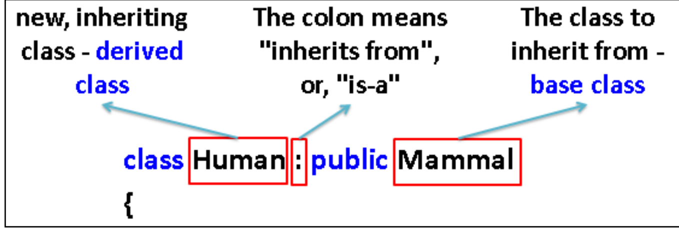

www.it-ebooks.info

```
    }
  };

  int main()
  {
    Human human;
    human.breathe(); // breathe using Mammal base class
    functionality
    human.talk();
    human.walk();

    Cat cat;
    cat.breathe(); // breathe using Mammal base class functionality
    cat.talk();
    cat.walk();

    Dog dog;
    dog.breathe();
    dog.talk();
    dog.walk();
  }
```

All of `Dog`, `Cat`, and `Human` inherit from `class Mammal`. This means that dog, cat, and human are mammals, and many more.

Syntax of inheritance

The syntax of inheritance is quite simple. Let's take the `Human` class definition as an example. The following screenshot is a typical inheritance statement:



The class on the left of the colon (**:**) is the new, derived class, and the class on the right of the colon is the base class.

www.it-ebooks.info

What does inheritance do?

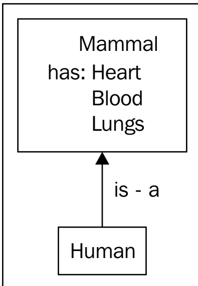
The point of inheritance is for the derived class to take on all the characteristics (data members, member functions) of the base class, and then to extend it with even more functionality. For instance, all mammals have a `breathe()` function. By inheriting from the `Mammal` class, the `Dog`, `Cat`, and `Human` classes all automatically gain the ability to `breathe()`.

Inheritance reduces replication of code since we don't have to re-implement common functionalities (such as `.breathe()`) for `Dog`, `Cat`, and `Human`. Instead, each of these derived classes enjoys the reuse of the `breathe()` function defined in `class Mammal`.

However, only the `Human` class has the `attack()` member function. This would mean that, in our code, only the `Human` class attacks. The `cat.attack()` function will introduce a compiler error, unless you write a member function `attack()` inside `class Cat` (or in `class Mammal`).

is-a relationship

Inheritance is often said to be an `is-a` relationship. When a `Human` class inherits from `Mammal` class, then we say that human *is-a* mammal.



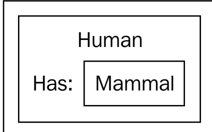
The Human inherits all the traits a Mammal has

For example, a `Human` object contains a `Mammal` function inside it, as follows:

```
class Human
{
  Mammal mammal;
};
```


www.it-ebooks.info

In this example, we would say the human *has-a* `Mammal` on it somewhere (which would make sense if the human were pregnant, or somehow carrying a mammal).



This Human class instance has some kind of mammal attached in it

Remember that we previously gave `Player` an `Armor` object inside it. It wouldn't make sense for the `Player` object to inherit from the `Armor` class, because it wouldn't make sense to say *the Player is-an Armor*. When deciding whether one class inherits from another or not in code design (for example, the Human class inherits from the Mammal class), you must always be able to comfortably say something like the Human class *is-a* Mammal. If the *is-a* statement sounds wrong, then it is likely that inheritance is the wrong relationship for that pair of objects.

In the preceding example, we're introducing a few new C++ keywords here. The first is `protected`.

protected variables

A `protected` member variable is different from a `public` or `private` variable. All three classes of variables are accessible inside the class in which they are defined. The difference between them is in regard to accessibility outside the class. A `public` variable is accessible anywhere inside the class and outside the class. A `private` variable is accessible inside the class but not outside the class. A `protected` variable is accessible inside the class, and inside of derived subclasses, but is not accessible outside the class. So, the `hp` and `speed` members of `class Mammal` will be accessible in the derived classes Dog, Cat, Horse, and Human, but not outside of these classes (in `main()` for instance).

Virtual functions

A virtual function is a member function whose implementation can be overridden in a derived class. In this example, the `talk()` member function (defined in `class Mammal`) is marked `virtual`. This means that the derived classes might or might not choose to implement their own version of what the `talk()` member function means.

www.it-ebooks.info

Purely virtual functions (and abstract classes)

A purely virtual function is one whose implementation you are required to override in the derived class. The `walk()` function in `class Mammal` is purely virtual; it was declared like this:

```
virtual void walk() = 0;
```

The `= 0` part at the end of the preceding code is what makes the function purely `virtual`.

The `walk()` function in `class Mammal` is purely `virtual` and this makes the Mammal class abstract. An abstract class in C++ is any class that has at least one purely virtual function.

If a class contains a purely virtual function and is abstract, then that class cannot be instantiated directly. That is, you cannot create a `Mammal` object now, on account of the purely virtual function `walk()`. If you tried to do the following code, you would get an error:

```
int main()
{
  Mammal mammal;
}
```

If you try to create a `Mammal` object, you will get the following error:

```
error C2259: 'Mammal' : cannot instantiate abstract class
```

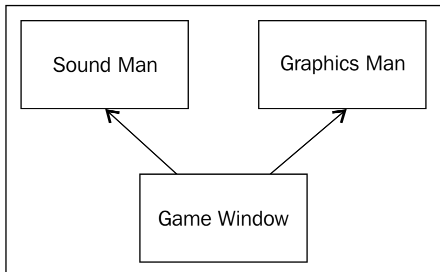
You can, however, create instances of derivatives of `class Mammal`, as long as the derived classes have all of the purely virtual member functions implemented.

Multiple inheritance

Not everything multiple is as good as it sounds. Multiple inheritance is when a derived class inherits from more than one base class. Usually, this works without a hitch if the multiple base classes we are inheriting from are completely unrelated.

www.it-ebooks.info

For example, we can have a class `Window` that inherits from the `SoundManager` and `GraphicsManager` base classes. If `SoundManager` provides a member function `playSound()` and `GraphicsManager` provides a member function `drawSprite()`, then the `Window` class will be able to use those additional capabilities without a hitch.



Game Window inheriting from Sound Man and Graphics Man means Game Window will have both sets of capabilities

However, multiple inheritance can have negative consequences. Say we want to create a class `Mule` that derives from both the `Donkey` and `Horse` classes. The `Donkey` and `Horse` classes, however, both inherit from the base class `Mammal`. We instantly have an issue! If we were to call `mule.talk()`, but `mule` does not override the `talk()` function, which member function should be invoked, that of `Horse` or `Donkey`? It's ambiguous.

private inheritance

A less talked about feature of C++ is `private` inheritance. Whenever a class inherits from another class publicly, it is known to all code whose parent class it belongs to. For example:

```
class Cat : public Mammal
```

This means that all code will know that `Cat` is an object of `Mammal`, and it will be possible to point to a `Cat*` instance using a base class `Mammal*` pointer. For example, the following code will be valid:

```
Cat cat;
Mammal* mammalPtr = &cat; // Point to the Cat as if it were a
                          // Mammal
```

The preceding code is fine if `Cat` inherits from `Mammal` publicly. Private inheritance is where code outside the `Cat` class is not allowed to know the parent class:

```
class Cat : private Mammal
```


www.it-ebooks.info

Here, externally calling code will not "know" that the `Cat` class derives from the `Mammal` class. Casting a `Cat` instance to the `Mammal` base class is not allowed by the compiler when inheritance is `private`. Use `private` inheritance when you need to hide the fact that a certain class derives from a certain parent class.

However, private inheritance is rarely used in practice. Most classes just use `public` inheritance. If you want to know more about private inheritance, see `http://stackoverflow.com/questions/406081/why-should-i-avoid-multiple-inheritance-in-c`.

Putting your classes into headers

So far, our classes have just been pasted before `main()`. If you continue to program that way, your code will all be in one file and appear as one big disorganized mess.

Therefore, it is a good programming practice to organize your classes into separate files. This makes editing each class's code individually much easier when there are multiple classes inside the project.

Take `class Mammal` and its derived classes from earlier. We will properly organize that example into separate files. Let's do it in steps:

1. Create a new file in your C++ project called `Mammal.h`. Cut and paste the entire `Mammal` class into that file. Notice that since the `Mammal` class included the use of `cout`, we write a `#include <iostream>` statement in that file as well.
2. Write a " `#include Mammal.h`" statement at the top of your `Source.cpp` file.

An example of what this looks like is shown in the following screenshot:

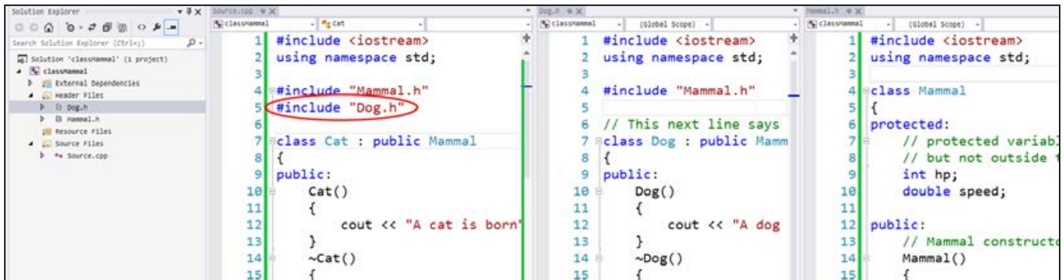
www.it-ebooks.info

What's happening here when the code is compiled is that the entire `Mammal` class is copied and pasted (#include) into the `Source.cpp` file, which contains the `main()` function, and the rest of the classes are derived from `Mammal`. Since `#include` is a copy and paste function, the code will function exactly the same as it did before; the only difference is that it will be much better organized and easier to look at. Compile and run your code at this step to make sure it still works.

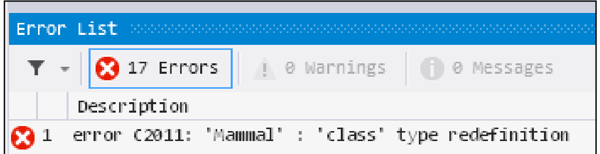
Check that your code compiles and runs often, especially when refactoring. When you don't know the rules, you're bound to make a lot of mistakes. This is why you should do your refactoring only in small steps. Refactoring is the name for the activity we are doing now—we are reorganizing the source to make better sense to other readers of our codebase. Refactoring usually does not involve rewriting too much of it.

The next thing you need to do is isolate the Dog, Cat, and Human classes into their own files. To do so, create the `Dog.h`, `Cat.h`, and `Human.h` files and add them to your project.

Let's start with the Dog class, as shown in the following screenshot:



If you use exactly this setup and try to compile and run your project, you will see the **'Mammal' : 'class' type redefinition** error, as shown in the following screenshot:



What this error means is that `Mammal.h` has been included twice in your project, once in `Source.cpp` and then again in `Dog.h`. This means effectively two versions of the Mammal class got added to the compiling code, and C++ is unsure which version to use.

www.it-ebooks.info

There are a few ways to fix this issue, but the easiest (and the one that Unreal Engine uses) is the `#pragma once` macro, as shown in the following screenshot:

```
1  #include <iostream>
2  using namespace std;
3
4  #include "Mammal.h"
5  #include "Dog.h"
6
7  class Cat : public Mammal
8  {
9  public:
10     Cat()
11     {
12         cout << "A cat is born"
13     }
```

```
1  #pragma once
2
3  #include <iostream>
4  using namespace std;
5
6  #include "Mammal.h"
7
8  // This next line says
9  class Dog : public Mamm
10 {
11 public:
12     Dog()
13     {
```

```
1  #pragma once
2
3  #include <iostream>
4  using namespace std;
5
6  class Mammal
7  {
8  protected:
9      // protected variab
10     // but not outside
11     int hp;
12     double speed;
13
```

We write `#pragma once` at the top of each header file. This way, the second time `Mammal.h` is included, the compiler doesn't copy and paste its contents again, since it already has been included before, and its content is actually already in the compiling group of files.

Do the same thing for `Cat.h` and `Human.h`, then `include` them both into your `Source.cpp` file where your `main()` function resides.

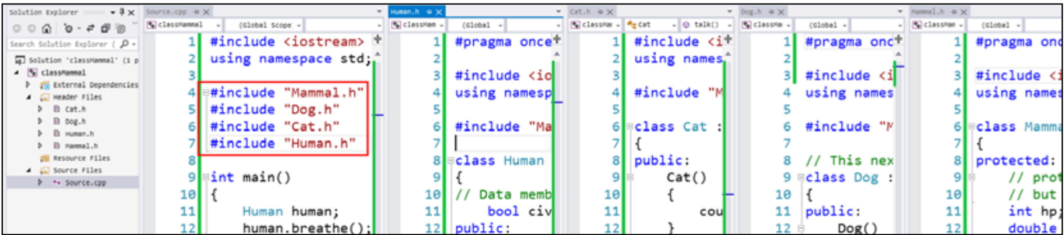


Diagram with all classes included

Now that we've included all classes into your project, the code should compile and run.

.h and .cpp

The next level of organization is to leave the class declarations in the header files (`.h`) and put the actual function implementation bodies inside some new `.cpp` files. Also, leave existing members inside the `class Mammal` declaration.

www.it-ebooks.info

For each class, perform the following operations:

1. Delete all function bodies (code between { and }) and replace them with just a semicolon. For the `Mammal` class, this would look as follows:

```
// Mammal.h
#pragma once
class Mammal
{
protected:
  int hp;
  double speed;

public:
  Mammal();
  ~Mammal();
  void breathe();
  virtual void talk();
  // pure virtual function,
  virtual void walk() = 0;
};
```

2. Create a new `.cpp` file called `Mammal.cpp`. Then simply put the member function bodies inside this file:

```
// Mammal.cpp
#include <iostream>
using namespace std;

#include "Mammal.h"
Mammal::Mammal() // Notice use of :: (scope resolution operator)
{
  hp = 100;
  speed = 1.0;
  cout << "A mammal is created!" << endl;
}
Mammal::~Mammal()
{
  cout << "A mammal has fallen!" << endl;
}
void Mammal::breathe()
{
  cout << "Breathe in.. breathe out" << endl;
}
void Mammal::talk()
{
  cout << "Mammal talk.. override this function!" << endl;
}
```


www.it-ebooks.info

It is important to note the use of the class name and scope resolution operator (double colon) when declaring the member function bodies. We prefix all member functions belonging to the `Mammal` class with `Mammal::`.

Notice how the purely virtual function does not have a body; it's not supposed to! Purely virtual functions are simply declared (and initialized to 0) in the base class, but implemented later in derived classes.

Exercise

Complete the separation of the different creature classes above into class header (`.h`) and class definition files (`.cpp`)

Summary

You learned about objects in C++; they are pieces of code that tie data members and member functions together into a bundle of code called `class` or `struct`. Object-oriented programming means that your code will be filled with things instead of just `int`, `float`, and `char` variables. You will have a variable that represents `Barrel`, another variable that represents `Player`, and so on, that is, a variable to represent every entity in your game. You will be able to reuse code by using inheritance; if you had to code implementations of `Cat` and `Dog`, you can code a common functionality in the base class `Mammal`. We also discussed encapsulation and how it is easier and more efficient to program objects such that they maintain their own internal state.

www.it-ebooks.info

www.it-ebooks.info

7
Dynamic Memory Allocation

In the previous chapter, we talked about class definitions and how to devise your own custom class. We discussed how by devising our own custom classes, we can construct variables that represented entities within your game or program.

In this chapter, we will talk about dynamic memory allocations and how to create space in memory for groups of objects.

Assume that we have a simplified version of `class Player`, as before, with only a constructor and a destructor:

```
class Player
{
  string name;
  int hp;
public:
  Player(){ cout << "Player born" << endl; }
  ~Player(){ cout << "Player died" << endl; }
};
```

www.it-ebooks.info

We talked earlier about the *scope* of a variable in C++; to recap, the scope of a variable is the section of the program where that variable can be used. The scope of a variable is generally inside the block in which it was declared. A block is just any section of code contained between { and }. Here is a sample program that illustrates variable scope:

```
int main()
{ ──────────────→ begin main() block
    int x; ─────→ x's scope is from here to the end of main()

    if( some_condition )
    { ──────────→ begin if block
        int y;  y's scope is from here to the end of the if
    } ──────────→ end if block [y destroyed]
}──────────→ end main() block, [x destroyed]
```

In this sample program, the x variable has scope through all of main(). The y variable's scope is only inside the if block

We mentioned previously that in general variables are destroyed when they go out of scope. Let's test this idea out with instances of `class Player`:

```
int main()
{
  Player player; // "Player born"
}                // "Player died" - player object destroyed here
```

The output of this program is as follows:

```
Player born
Player died
```

The destructor for the player object is called at the end of the player object's scope. Since the scope of a variable is the block within which it is defined in the three lines of code, the `Player` object would be destroyed immediately at the end of `main()`, when it goes out of scope.

www.it-ebooks.info

Dynamic memory allocation

Now, let's try allocating a `Player` object dynamically. What does that mean?

We use the `new` keyword to allocate it!

```
int main()
{
  // "dynamic allocation" - using keyword new!
  // this style of allocation means that the player object will
  // NOT be deleted automatically at the end of the block where
  // it was declared!
Player *player = new Player();
} // NO automatic deletion!
```

The output of this program is as follows:

```
Player born
```

The player does not die! How do we kill the player? We must explicitly call `delete` on the `player` pointer.

The delete keyword

The `delete` operator invokes the destructor on the object being deleted, as shown in the following code:

```
int main()
{
  // "dynamic allocation" - using keyword new!
  Player *player = new Player();
  delete player; // deletion invokes dtor
}
```

The output of the program is as follows:

```
Player born
Player died
```

So, only "normal" (or "automatic" also called as non-pointer type) variable types get destroyed at the end of the block in which they were declared. Pointer types (variables declared with `*` and `new`) are not automatically destroyed even when they go out of scope.

What is the use of this? Dynamic allocations let you control when an object is created and destroyed. This will come in handy later.

www.it-ebooks.info

Memory leaks

So dynamically allocated objects created with `new` are not automatically deleted, unless you explicitly call `delete` on them. There is a risk here! It is called a *memory leak*. Memory leaks happen when an object allocated with `new` is not ever deleted. What can happen is that if a lot of objects in your program are allocated with `new` and then you stop using them, your computer will run out of memory eventually due to memory leakage.

Here is a ridiculous sample program to illustrate the problem:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
  string name;
  int hp;
public:
  Player(){ cout << "Player born" << endl; }
  ~Player(){ cout << "Player died" << endl; }
};

int main()
{
  while( true ) // keep going forever,
  {
    // alloc..
    Player *player = new Player();
    // without delete == Memory Leak!
  }
}
```

This program, if left to run long enough, will eventually gobble the computer's memory, as shown in the following screenshot:

2 GB of RAM used for Player objects!

Note that no one ever intends to write a program with this type of problem in it! Memory leak problems happen accidentally. You must take care of your memory allocations and `delete` objects that are no longer in use.

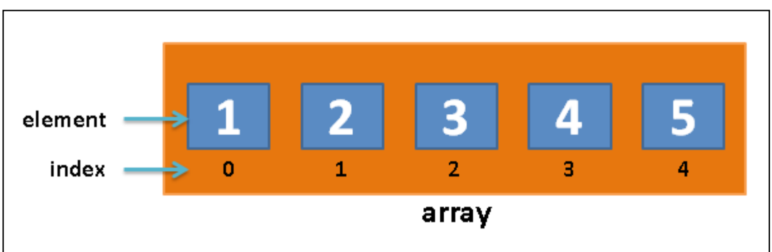
www.it-ebooks.info

Regular arrays

An array in C++ can be declared as follows:

```
#include <iostream>
using namespace std;
int main()
{
  int array[ 5 ];  // declare an "array" of 5 integers
                   // fill slots 0-4 with values
array[ 0 ] = 1;
array[ 1 ] = 2;
array[ 2 ] = 3;
array[ 3 ] = 4;
array[ 4 ] = 5;
  // print out the contents
  for( int index = 0; index < 5; index++ )
    cout << array[ index ] << endl;
}
```

The way this looks in memory is something like this:



That is, inside the `array` variable are five slots or elements. Inside each of the slots is a regular `int` variable.

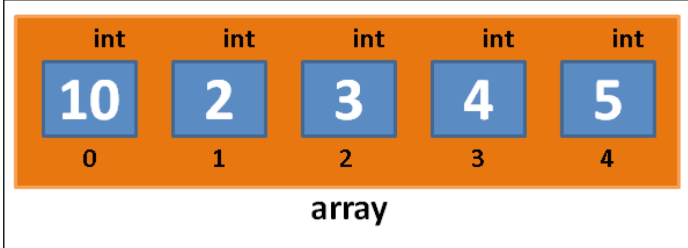
The array syntax

So, how do you access one of the `int` values in the array? To access the individual elements of an array, we use square brackets, as shown in the following line of code:

```
array[ 0 ] = 10;
```


www.it-ebooks.info

The preceding line of code would change the element at slot 0 of the array to a 10:



In general, to get to a particular slot of an array, you will write the following:

```
array[ slotNumber ] = value to put into array;
```

Keep in mind that array slots are always indexed starting from 0. To get into the first slot of the array, use `array[0]`. The second slot of the array is `array[1]` (not `array[2]`). The final slot of the array above is `array[4]` (not `array[5]`). The `array[5]` data type is out of bounds of the array! (There is no slot with index 5 in the preceding diagram. The highest index is 4.)

Don't go out of bounds of the array! It might work some times, but other times your program will crash with a **memory access violation** (accessing memory that doesn't belong to your program). In general, accessing memory that does not belong to your program is going to cause your app to crash, and if it doesn't do so immediately, there will be a hidden bug in your program that only causes problems once in a while. You must always be careful when indexing an array.

Arrays are built into C++, that is, you don't need to include anything special to have immediate use of arrays. You can have arrays of any type of data that you want, for example, arrays of `int`, `double`, `string`, and even your own custom object types (`Player`).

Exercise

1. Create an array of five strings and put inside it some names (made up or random, it doesn't matter).
2. Create an array of doubles called `temps` with three elements and store the temperature for the last three days in it.

www.it-ebooks.info

Solutions

1. The following is a sample program with an array of five strings:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string array[ 5 ];  // declare an "array" of 5 strings
                      // fill slots 0-4 with values
array[ 0 ] = "Mariam McGonical";
array[ 1 ] = "Wesley Snice";
array[ 2 ] = "Kate Winslett";
array[ 3 ] = "Erika Badu";
array[ 4 ] = "Mohammad";
  // print out the contents
  for( int index = 0; index < 5; index++ )
    cout << array[ index ] << endl;
}
```

2. The following is just the array:

```
double temps[ 3 ];
// fill slots 0-2 with values
temps[ 0 ] = 0;
temps[ 1 ] = 4.5;
temps[ 2 ] = 11;
```

C++ style dynamic size arrays (new[] and delete[])

It probably occurred to you that we won't always know the size of an array at the start of a program. We would need to allocate the array's size dynamically.

However, if you've tried it, you might have noticed that this doesn't work!

Let's try and use the `cin` command to take in an array size from the user. Let's ask the user how big he wants his array and try to create one for him of that size:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "How big?" << endl;
```


www.it-ebooks.info

```
    int size;       // try and use a variable for size..
    cin >> size;    // get size from user
    int array[ size ];  // get error: "unknown size"
  }
```

We get the following error:

```
error C2133: 'array' : unknown size
```

The problem is that the compiler wants to allocate the size of the array. However, unless the variable size is marked `const`, the compiler will not be sure of its value at compile time. The C++ compiler cannot size the array at compile time, so it generates a compile time error.

To fix this, we have to allocate the array dynamically (on the "heap"):

```
#include <iostream>
using namespace std;
int main()
{
  cout << "How big?" << endl;
  int size;       // try and use a variable for size..
  cin >> size;
  int *array = new int[ size ];  // this works
  // fill the array and print
for( int index = 0; index < size; index++ )
{
  array[ index ] = index * 2;
  cout << array[ index ] << endl;
}
delete[] array; // must call delete[] on array allocated with
                // new[]!
}
```

So the lessons here are as follows:

- To allocate an array of some type (for example, `int`) dynamically, you must use new `int[numberOfElementsInArray]`.
- Arrays allocated with `new[]` must be later deleted with `delete[]`, otherwise you'll get a memory leak! (that's `delete[]` with square brackets! Not regular delete).

www.it-ebooks.info

Dynamic C-style arrays

C-style arrays are a legacy topic, but they are still worth discussing since even though they are old, you might still see them used sometimes.

The way we declare a C-style array is as follows:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "How big?" << endl;
  int size;       // try and use a variable for size..
  cin >> size;
  // the next line will look weird..
  int *array = (int*)malloc( size*sizeof(int) ); // C-style
  // fill the array and print
for( int index = 0; index < size; index++ )
  {
    array[ index ] = index * 2;
    cout << array[ index ] << endl;
  }
free( array ); // must call free() on array allocated with
               // malloc() (not delete[]!)
}
```

The differences here are highlighted.

A C-style array is created using the `malloc()` function. The word malloc stands for "memory allocate". This function requires you to pass in the size of the array in bytes to create and not just the number of elements you want in the array. For this reason, we multiply the number of elements requested (size) by `sizeof` of the type inside the array. The size in bytes of a few typical C++ types is listed in the following table:

C++ primitive type	sizeof (size in bytes)
int	4
float	4
double	8
long long	8

Memory allocated with the `malloc()` function must later be released using `free()`.

www.it-ebooks.info

Summary

This chapter introduced you to C and C++ style arrays. In most of the UE4 code, you will use the UE4 editor built in collection classes (TArray<T>). However, you need familiarity with the basic C and C++ style arrays to be a very good C++ programmer.

www.it-ebooks.info

8
Actors and Pawns

Now we will really delve into UE4 code. At first, it is going to look daunting. The UE4 class framework is massive, but don't worry. The framework is massive, so your code doesn't have to be. You will find that you can get a lot done and a lot onto the screen using relatively less code. This is because the UE4 engine code is so extensive and well programmed that they have made it possible to get almost any game-related task done easily. Just call the right functions, and voila, what you want to see will appear on the screen. The entire notion of a framework is that it is designed to let you get the gameplay you want, without having to spend a lot of time in sweating out the details.

Actors versus pawns

In this chapter, we will discuss actors and pawns. Although it sounds as if pawns will be a more basic class than actors, it is actually the other way around. A UE4 actor (the `Actor` class) object is the basic type of the things that can be placed in the UE4 game world. In order to place anything in the UE4 world, you must derive from the `Actor` class.

A `Pawn` is an object that represents something that you or the computer's **Artificial Intelligence** (**AI**) can control on the screen. The `Pawn` class derives from the `Actor` class, with the additional ability to be controlled by the player directly or by an AI script. When a pawn or actor is controlled by a controller or AI, it is said to be possessed by that controller or AI.

Think of the `Actor` class as a character in a play. Your game world is going to be composed of a bunch of *actors*, all acting together to make the gameplay work. The game characters, **Non-player Characters** (**NPC**s), and even treasure chests will be actors.

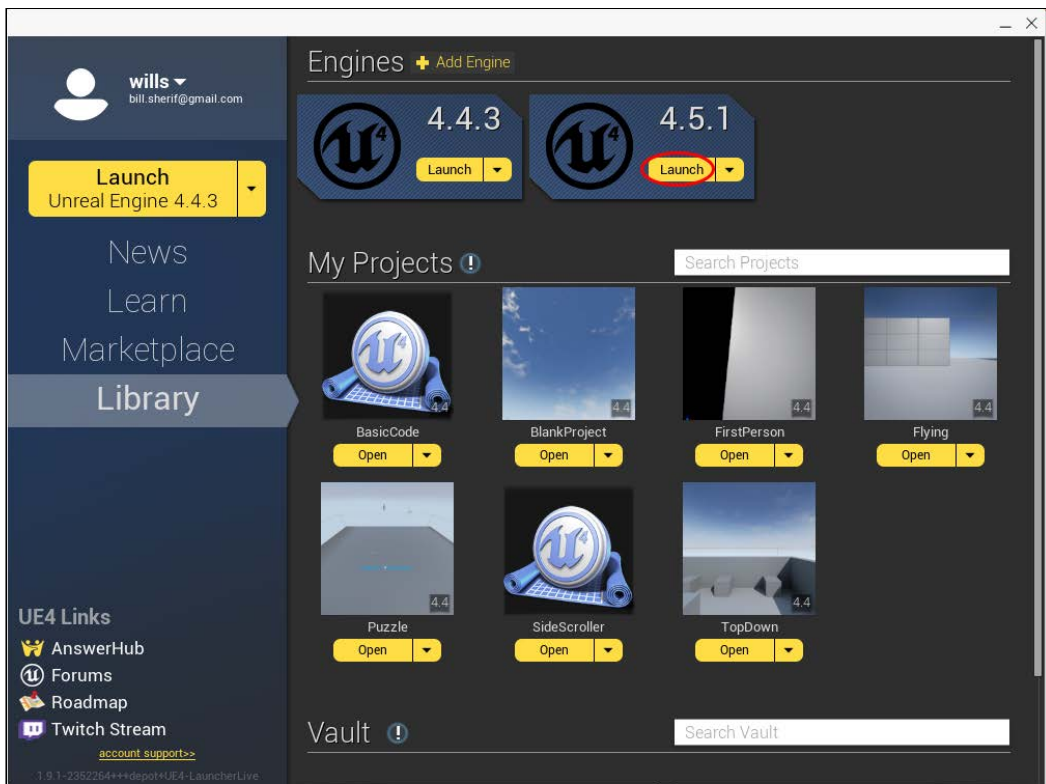
www.it-ebooks.info

Creating a world to put your actors in

Here, we will start from scratch and create a basic level into which we can put our game characters.

The UE4 team has already done a great job of presenting how the world editor can be used to create a world in UE4. I want you to take a moment to create your own world.

First, create a new, blank UE4 project to get started. To do this, in the Unreal Launcher, click on the **Launch** button beside your most recent engine installation, as shown in the following screenshot:



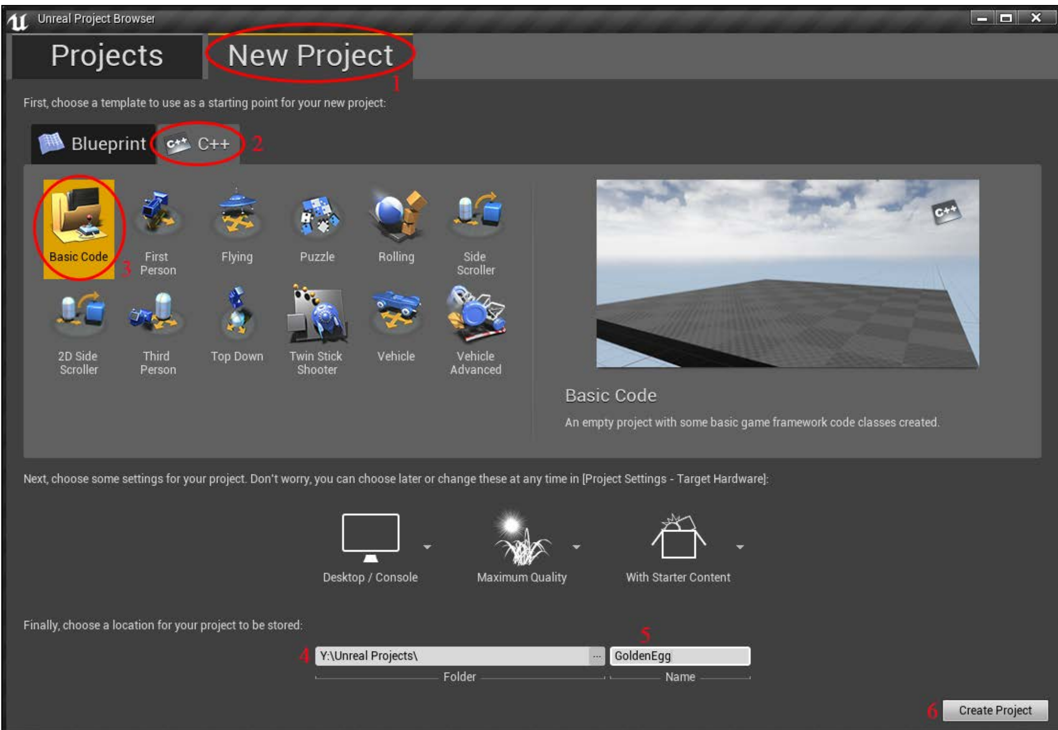
That will launch the Unreal Editor. The Unreal Editor is used to visually edit your game world. You're going to spend a lot of time in the Unreal Editor, so please take your time to experiment and play around with it.

www.it-ebooks.info

I will only cover the basics of how to work with the UE4 editor. You will need to let your creative juices flow, however, and invest some time in order to become familiar with the editor.

To learn more about the UE4 editor, take a look at the *Getting Started: Introduction to the UE4 Editor* playlist, which is available at `https://www.youtube.com/playlist?list=PLZlv_N0_O1gasd4IcOe9Cx9wHoBB7rxFl`.

Once you've launched the UE4 editor, you will be presented with the **Projects** dialog. The following screenshot shows the steps to be performed with numbers corresponding to the order in which they need to be performed:



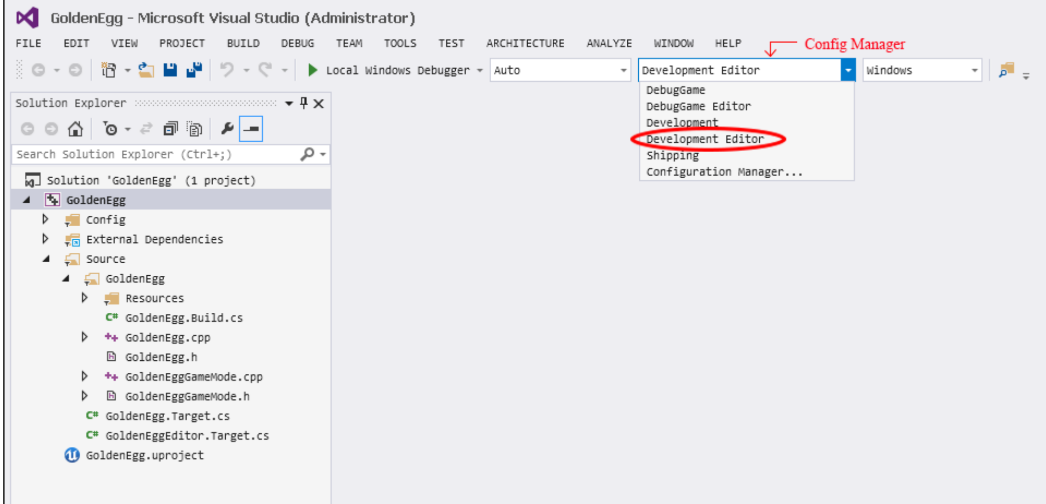
Perform the following steps to create a project:

1. Select the **New Project** tab at the top of the screen.
2. Click on the **C++** tab (the second subtab).
3. Then select **Basic Code** from the available projects listing.

www.it-ebooks.info

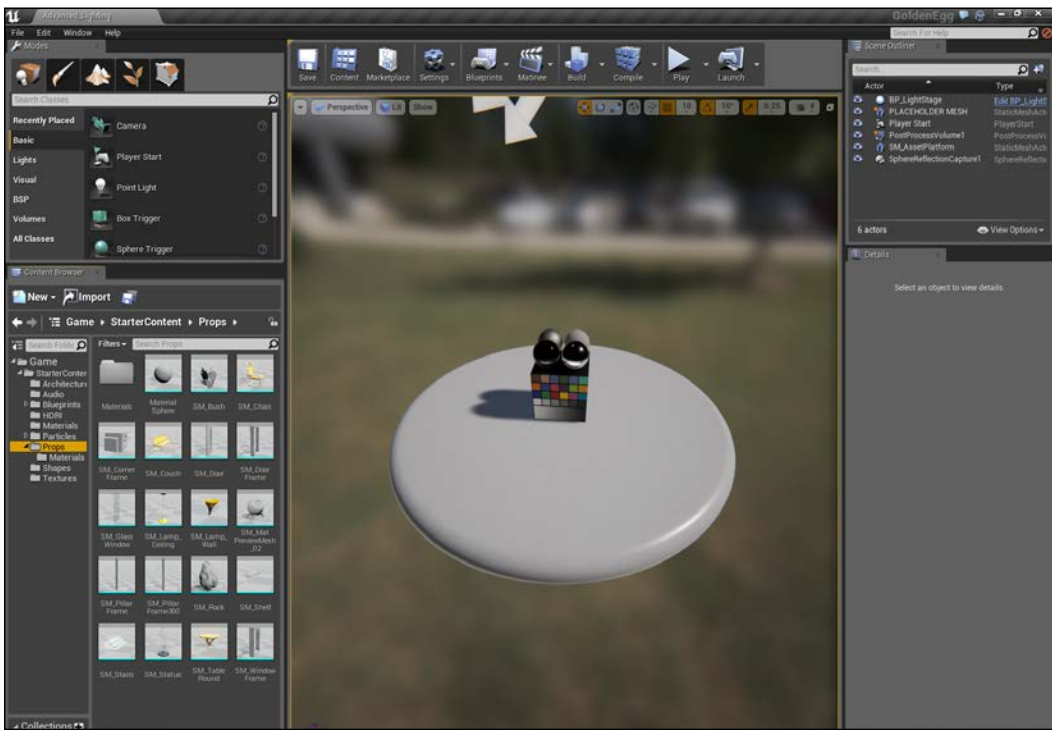
4. Set the directory where your project is located (mine is **Y:\Unreal Projects**). Choose a hard disk location with a lot of space (the final project will be around 1.5 GB).
5. Name your project. I called mine **GoldenEgg**.
6. Click on **Create Project** to finalize project creation.

Once you've done this, the UE4 launcher will launch Visual Studio. There will only be a couple of source files in Visual Studio, but we're not going to touch those now. Make sure that **Development Editor** is selected from the **Configuration Manager** dropdown at the top of the screen, as shown in the following screenshot:



www.it-ebooks.info

Now launch your project by pressing *Ctrl* + *F5* in Visual Studio. You will find yourself in the Unreal Engine 4 editor, as shown in the following screenshot:



The UE4 editor

We will explore the UE4 editor here. We'll start with the controls since it is important to know how to navigate in Unreal.

Editor controls

If you've never used a 3D editor before, the controls can be quite hard to learn. These are the basic navigation controls while in edit mode:

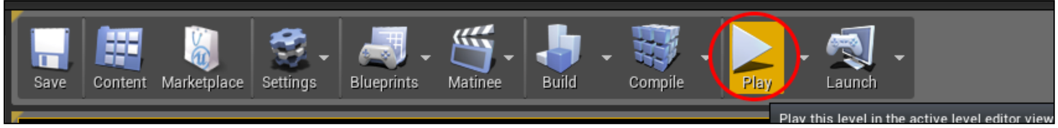
- Use the arrow keys to move around in the scene
- Press *Page Up* or *Page Down* to go up and down vertically
- Left mouse click + drag it left or right to change the direction you are facing
- Left mouse click + drag it up or down to *dolly* (move the camera forward and backward, same as pressing up/down arrow keys)

www.it-ebooks.info

- Right mouse click + drag to change the direction you are facing
- Middle mouse click + drag to pan the view
- Right mouse click and the *W*, *A*, *S*, and *D* keys to move around the scene

Play mode controls

Click on the **Play** button in the bar at the top, as shown in the following screenshot. This will launch the play mode.



Once you click on the **Play** button, the controls change. In play mode, the controls are as follows:

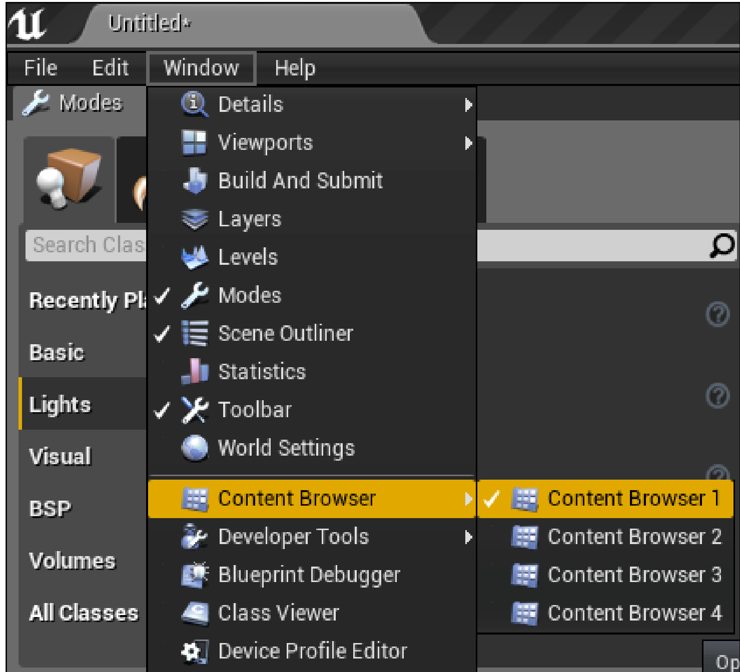
- The *W*, *A*, *S*, and *D* keys for movement
- The left or right arrow keys to look toward the left and right, respectively
- The mouse's motion to change the direction in which you look
- The *Esc* key to exit play mode and return to edit mode

What I suggest you do at this point is try to add a bunch of shapes and objects into the scene and try to color them with different *materials*.

Adding objects to the scene

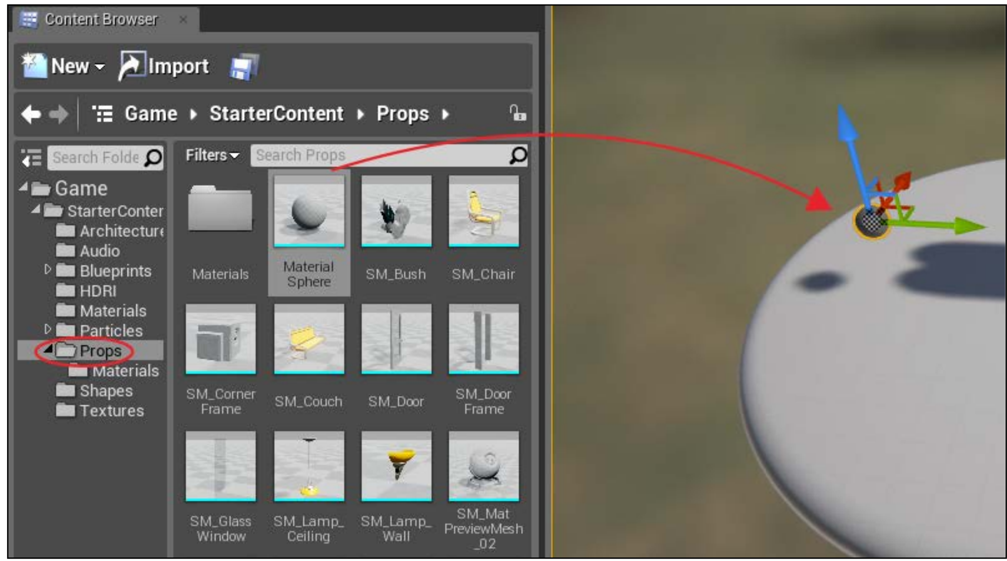
Adding objects to the scene is as easy as dragging and dropping them in from the **Content Browser** tab. The **Content Browser** tab appears, by default, docked at the left-hand side of the window. If it isn't seen, simply select **Window** and navigate to **Content Browser** in order to make it appear.

www.it-ebooks.info



Make sure that the Content Browser is visible in order to add objects to your level

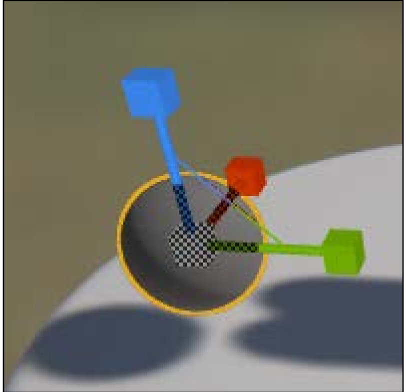
Next, select the **Props** folder on the left-hand side of the **Content Browser**.



Drag and drop things from the Content Browser into your game world

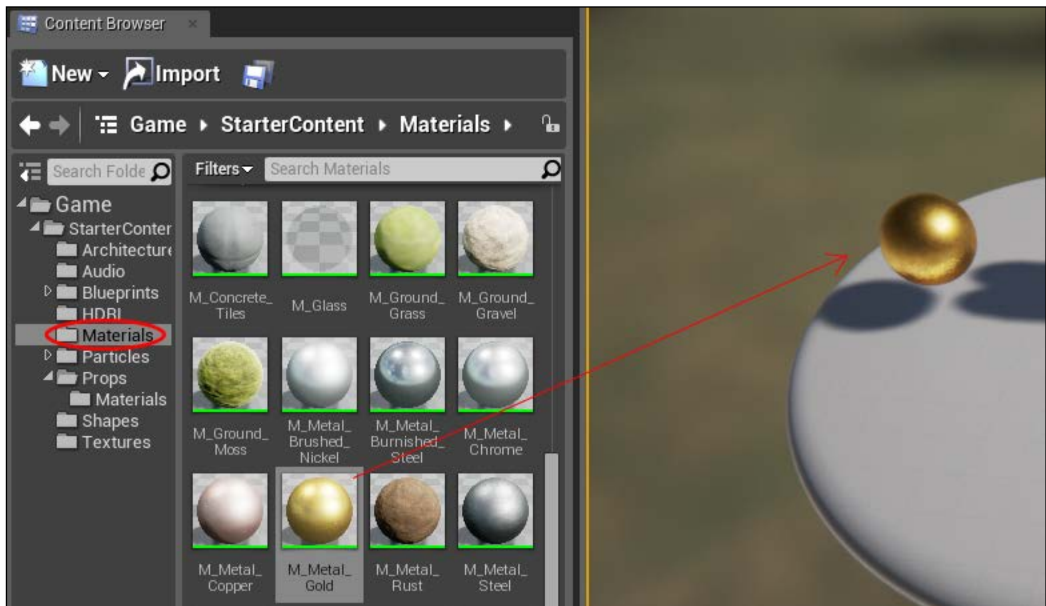
www.it-ebooks.info

To resize an object, press *R* on your keyboard. The manipulators around the object will appear as boxes, which denotes resize mode.



Press *R* on your keyboard to resize an object

In order to change the material that is used to paint the object, simply drag and drop a new material from the **Content Browser** window inside the **Materials** folder.



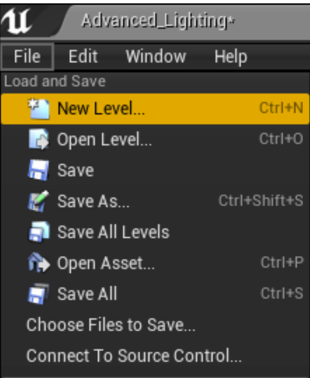
Drag and drop a material from the Content Browser's Materials folder to color things with a new color

www.it-ebooks.info

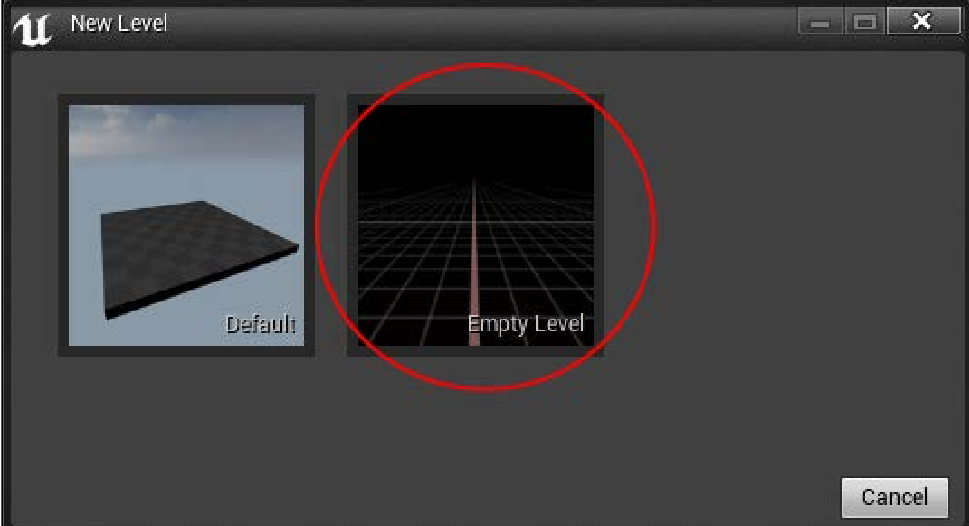
Materials are like paints. You can coat an object with any material you want by simply dragging and dropping the material you desire onto the object you desire it to be coated on. Materials are only skin-deep: they don't change the other properties of an object (such as weight).

Starting from scratch

If you want to start creating a level from scratch, simply click on **File** and navigate to **New Level...**, as shown here:



You can then select between **Default** and **Empty Level**. I think selecting **Empty Level** is a good idea, for the reasons that are mentioned later.



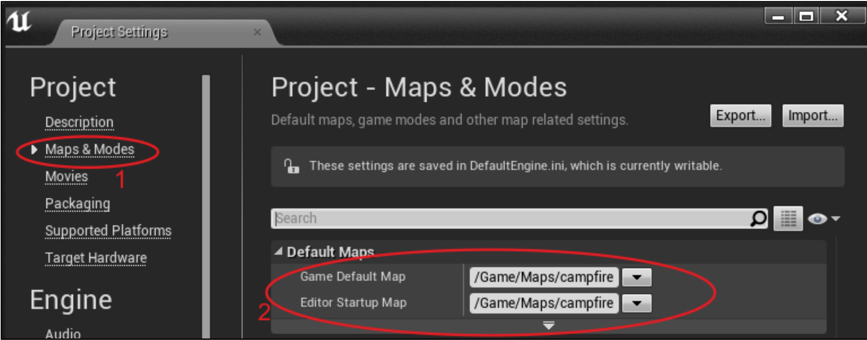
www.it-ebooks.info

The new level will be completely black in color to start with. Try dragging and dropping some objects from the **Content Browser** tab again.

This time, I added a resized shapes / box for the ground plane and textured it with moss, a couple of **Props / SM_Rocks**, **Particles / P_Fire**, and most importantly, a light source.

Be sure to save your map. Here's a snapshot of my map (how does yours look?):

If you want to change the default level that opens when you launch the editor, go to **Project Settings | Maps & Modes**; then you will see a **Game Default Map** and **Editor Startup Map** setting, as shown in the following screenshot:



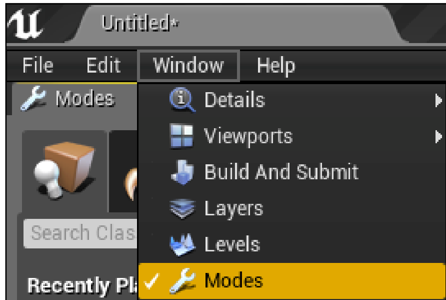
Adding light sources

Note that if your scene appears completely black, it is possible that you forgot to put a light source into it.

www.it-ebooks.info

In the previous scene, the **P_Fire** particle emitter acts as a light source, but it only emits a small amount of light. To make sure that everything appears well-lit in your scene, you should add a light source, as follows:

1. Go to **Window** and then click on **Modes** to ensure that the light sources panel is shown:



2. Then, from the **Modes** panel, drag one of the **Lights** object into the scene:

3. Select the lightbulb and box icon (it looks like a mushroom, but it isn't).
4. Click on **Lights** in the left-hand side panel.
5. Select the type of light you want and just pull it into your scene.

If you don't have a light source, your scene will appear completely black.

www.it-ebooks.info

Collision volumes

You might have noticed that, so far, the camera just passes through all the scene geometry, even in play mode. That's not good. Let's make it such that the player can't just walk through the rocks in our scene.

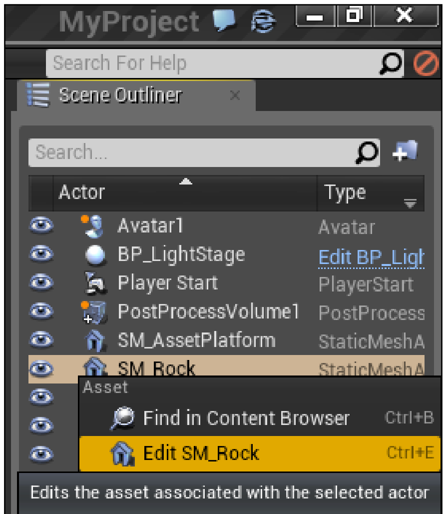
There are a few different types of collision volumes. Generally, perfect mesh-mesh collisions are way too expensive to do at runtime. Instead, we use an approximation (a bounding volume) to guess the collision volume.

Adding collision detection for the objects editor

The first thing we have to do is associate a collision volume with each of the rocks in the scene.

We can do this from the UE4 editor as follows:

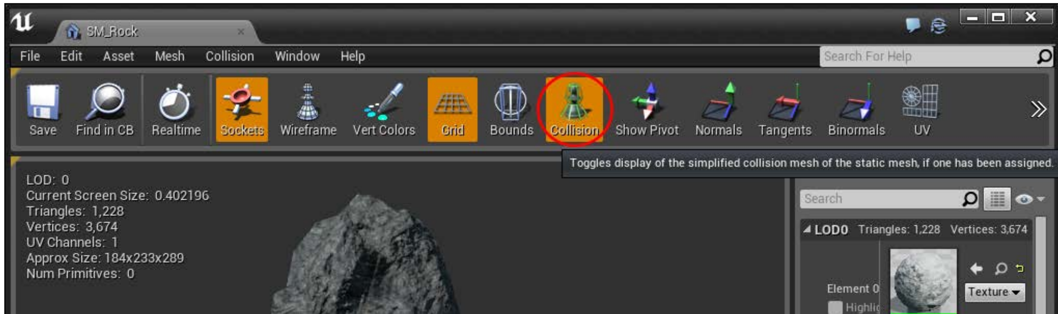
1. Click on an object in the scene for which you want to add a collision volume.
2. Right-click on this object in the **Scene Outliner** tab (the default appears on the right-hand side of the screen) and select edit, as shown in the following screenshot:



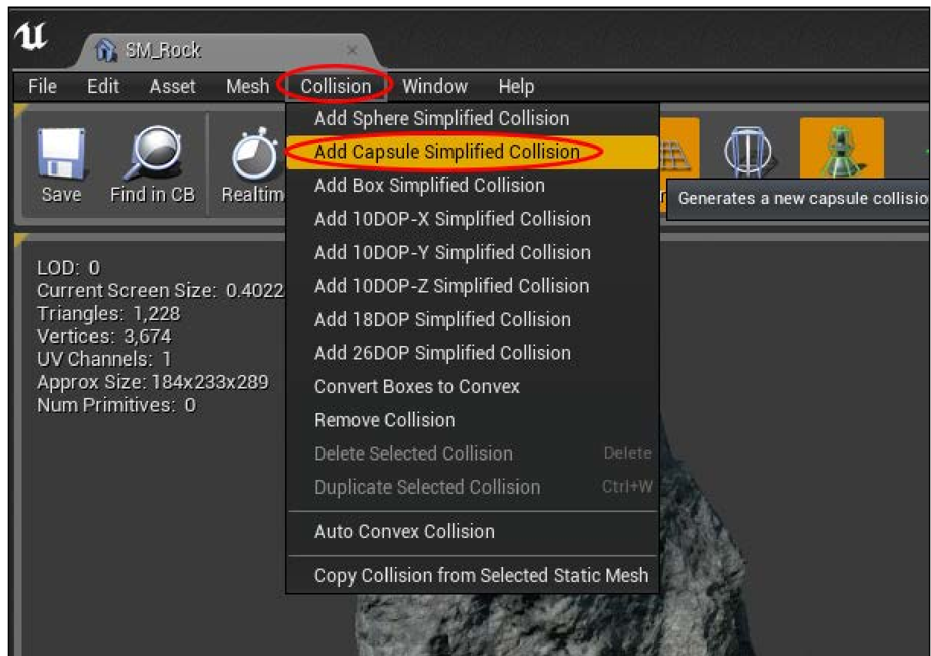
You will find yourself in the mesh editor.

www.it-ebooks.info

3. Ensure that the collision volume is highlighted, at the top of the screen:

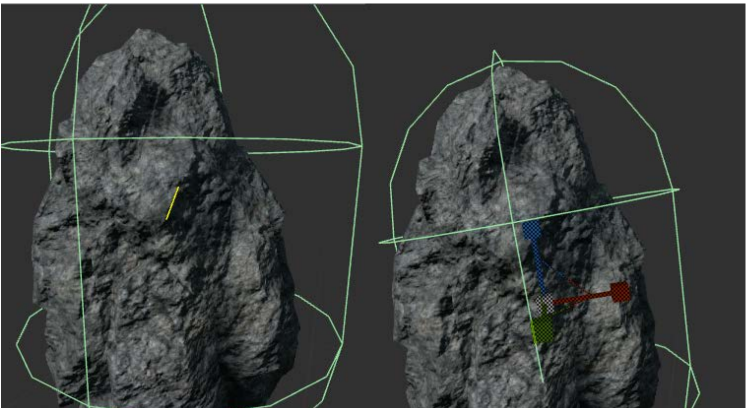


4. Go to the **Collision** menu and then click on **Add Capsule Simplified Collision**:



www.it-ebooks.info

5. The collision volume, when added successfully, will appear as a bunch of lines surrounding the object, as shown in the following images:



The default collision capsule (left) and manually resized versions (right)

6. You can resize (*R*), rotate (*E*), move (*W*), and change the collision volume as you wish, the same way you would manipulate an object in the UE4 editor.
7. When you're done with adding collision meshes, try to click on **Play**; you will notice that you can no longer pass through your collidable objects.

Adding an actor to the scene

Now that we have a scene up and running, we need to add an actor to the scene. Let's first add an avatar for the player, complete with a collision volume. To do this, we'll have to inherit from a UE4 `GameFramework` class.

Creating a player entity

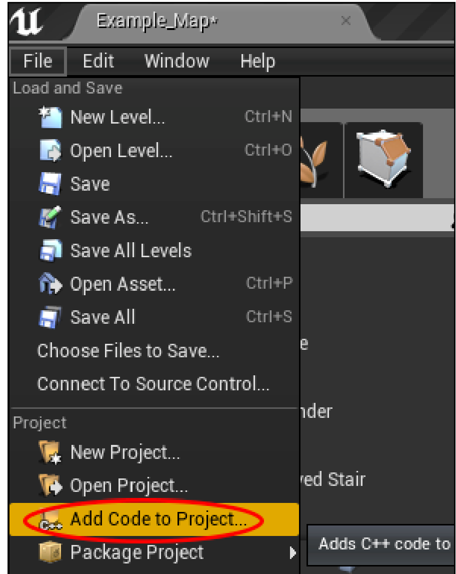
In order to create an onscreen representation of the player, we'll need to derive from the `Character` class in Unreal.

Inheriting from UE4 GameFramework classes

UE4 makes it easy to inherit from the base framework classes. All you have to do is perform the following steps:

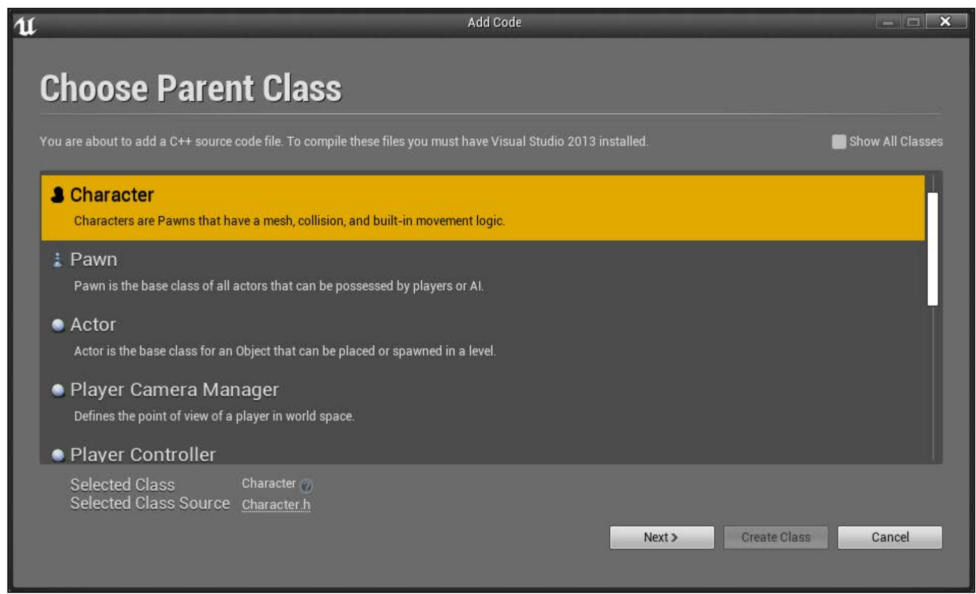
1. Open your project in the UE4 editor.
2. Go to **File** and then select **Add Code to Project...**.

www.it-ebooks.info



Navigating to File | Add Code To Project... will allow you to derive from any of the UE4 GameFramework classes

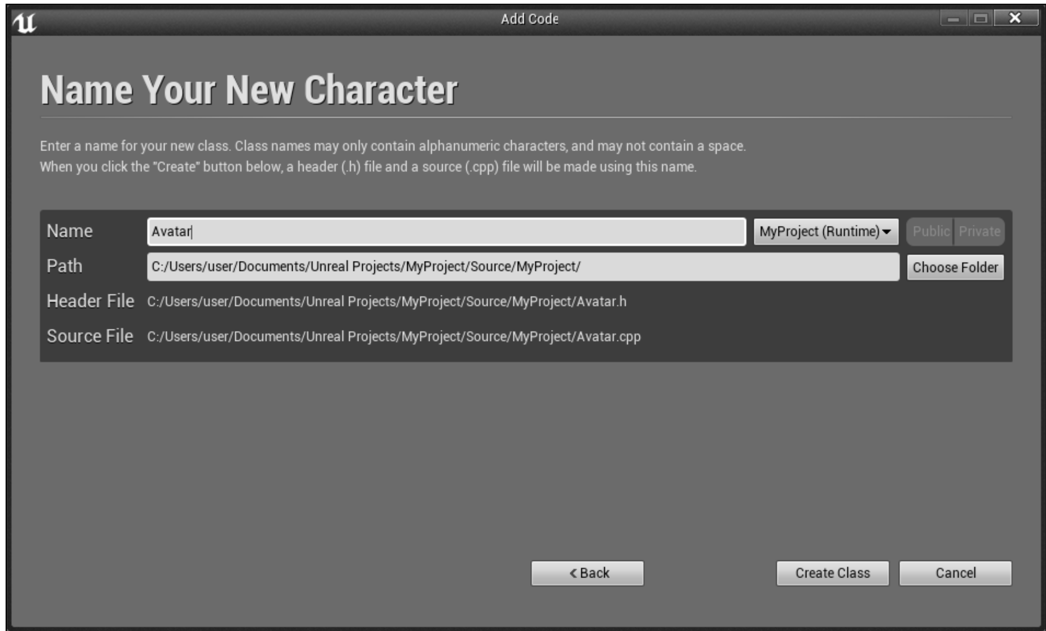
3. From here, choose the base class you want to derive from. You have **Character**, **Pawn**, **Actor**, and so on, but for now, we will derive from **Character**:



Select the UE4 class you want to derive from

www.it-ebooks.info

4. Click on **Next >** to get this dialog box, where you name the class. I named my player's class `Avatar`.



5. Finally, click on **Create Class** to create the class in code, as shown in the preceding screenshot.

Let UE4 refresh your Visual Studio project when it asks you. Open the new `Avatar.h` file from the **Solution Explorer**.

The code that UE4 generates will look a little weird. Remember the macros that I suggested you avoid in *Chapter 5*, *Functions and Macros*. The UE4 code uses macros extensively. These macros are used to copy and paste boilerplate starter code that lets your code integrate with the UE4 editor.

The contents of the `Avatar.h` file are shown in the following code:

```
#pragma once
// Avatar.h code file
#include "GameFramework/Character.h"
#include "Avatar.generated.h"
UCLASS()
class MYPROJECT_API AAvatar : public ACharacter
{
  GENERATED_UCLASS_BODY()
};
```


www.it-ebooks.info

Let's talk about macros for a moment.

The `UCLASS()` macro basically makes your C++ code class available in the UE4 editor. The `GENERATED_UCLASS_BODY()` macro copies and pastes code that UE4 needs to make your class function properly as a UE4 class.

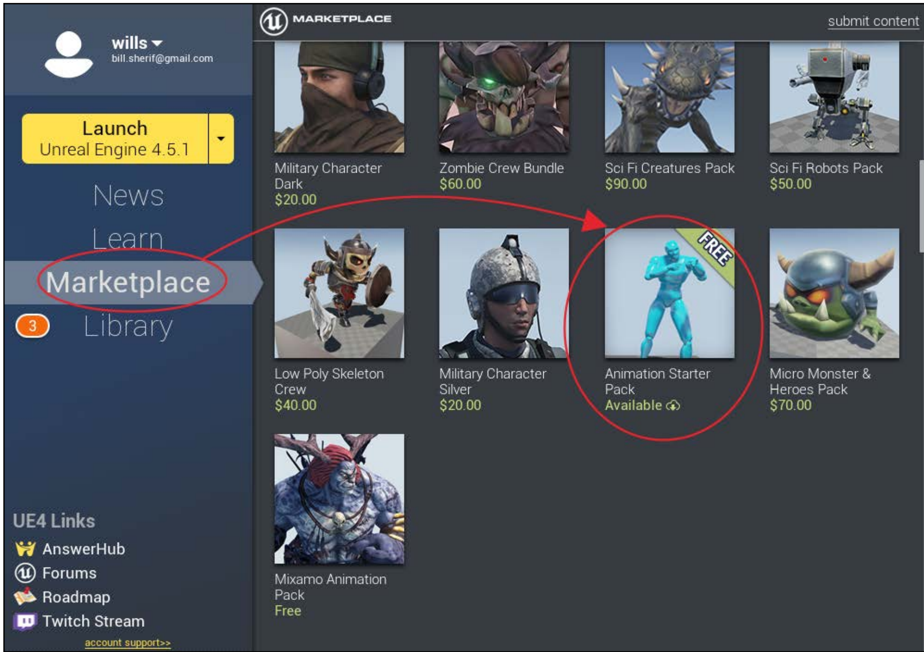
For `UCLASS()` and `GENERATED_UCLASS_BODY()`, you don't truly need to understand how UE4 works its magic. You just need to make sure that they are present at the right spot (where they were when you generated the class).

Associating a model with the Avatar class

Now we need to associate a model with our character object. In order to do this, we need a model to play with. Fortunately, there is a whole pack of sample models available from the UE4 marketplace for free.

Downloading free models

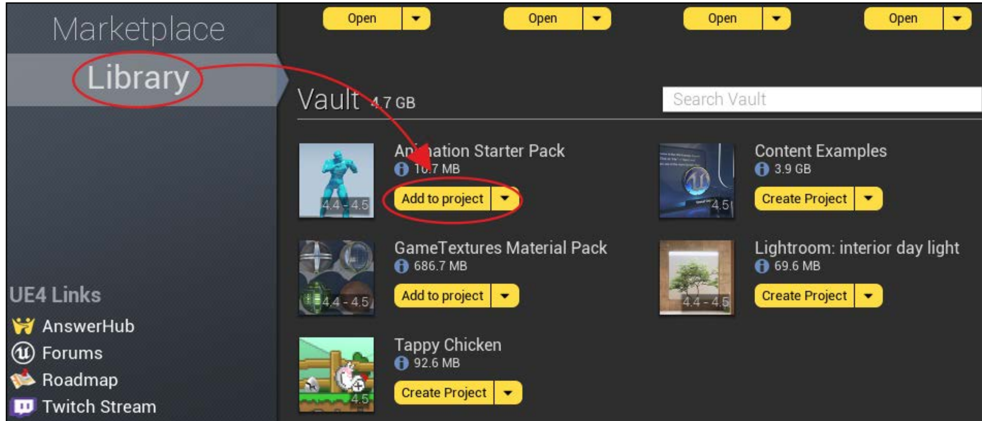
To create the player object, we'll download the **Animation Starter Pack** file (which is free) from the **Marketplace** tab.



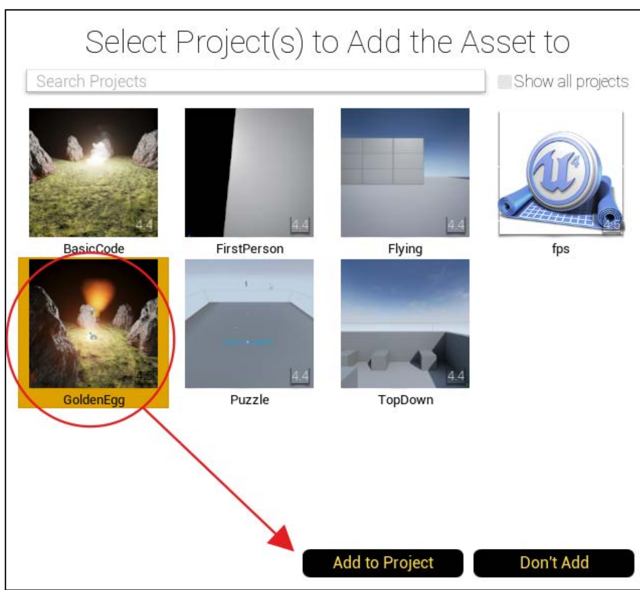
From the Unreal Launcher, click on Marketplace and search for Animation Starter Pack, which is free at the time of writing this book

www.it-ebooks.info

After you've downloaded the **Animation Starter Pack** file, you will be able to add it to any of the projects you've previously created, as shown in the following screenshot:



When you click on **Add to project** under **Animation Starter Pack**, you will get this pop up, asking which project to add the pack to:



Simply select your project and the new artwork will be available in your **Content Browser**.

www.it-ebooks.info

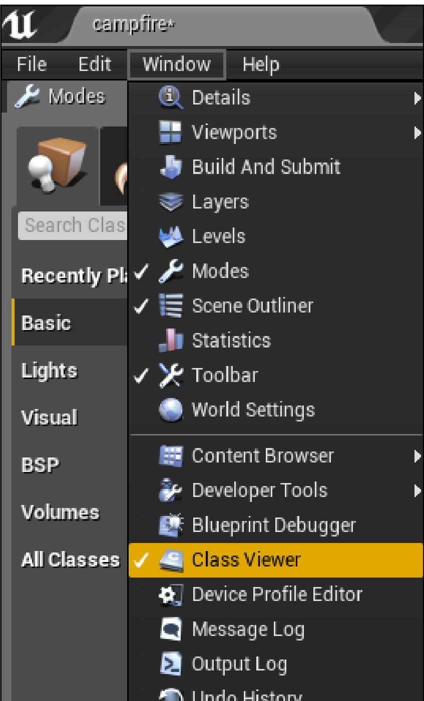
Loading the mesh

In general, it is considered a bad practice to hardcode your assets into the game. Hardcoding means that you write C++ code that specifies the asset to load. However, hardcoding means the loaded asset is part of the final executable, which will mean that changing the asset that is loaded wouldn't be modifiable at runtime. This is a bad practice. It is much better to be able to change the asset loaded during runtime.

For this reason, we're going to use the UE4 blueprints feature to set up the model mesh and collision capsule of our `Avatar` class.

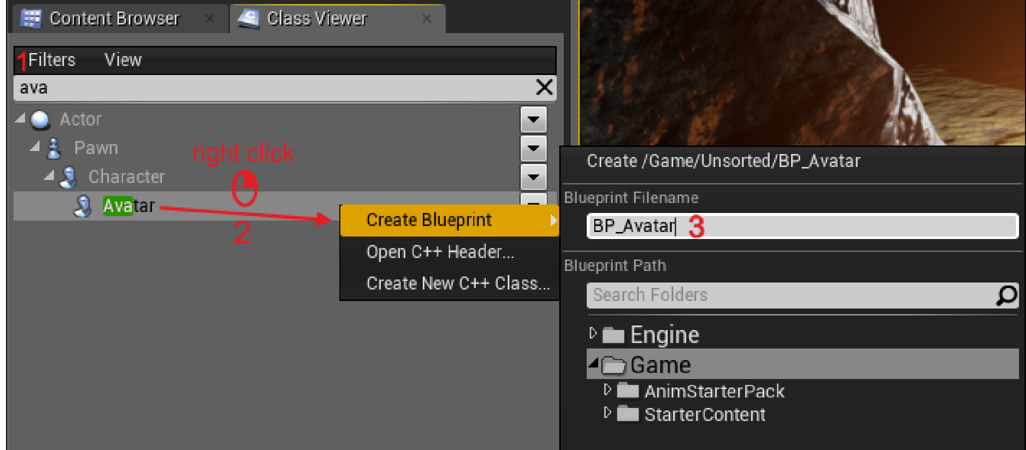
Creating a blueprint from our C++ class

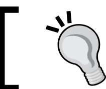
1. This is really easy. Open the **Class Viewer** tab by navigating to **Window** and then clicking on **Class Viewer**, as shown here:



www.it-ebooks.info

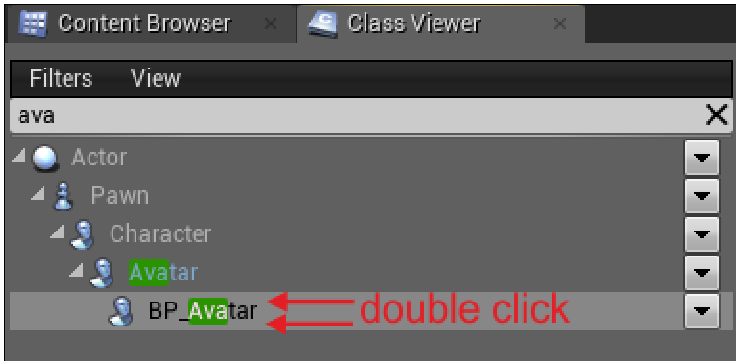
2. In the **Class Viewer** dialog, start typing in the name of your C++ class. If you have properly created and exported the class from your C++ code, it will appear, as shown in the following screenshot:





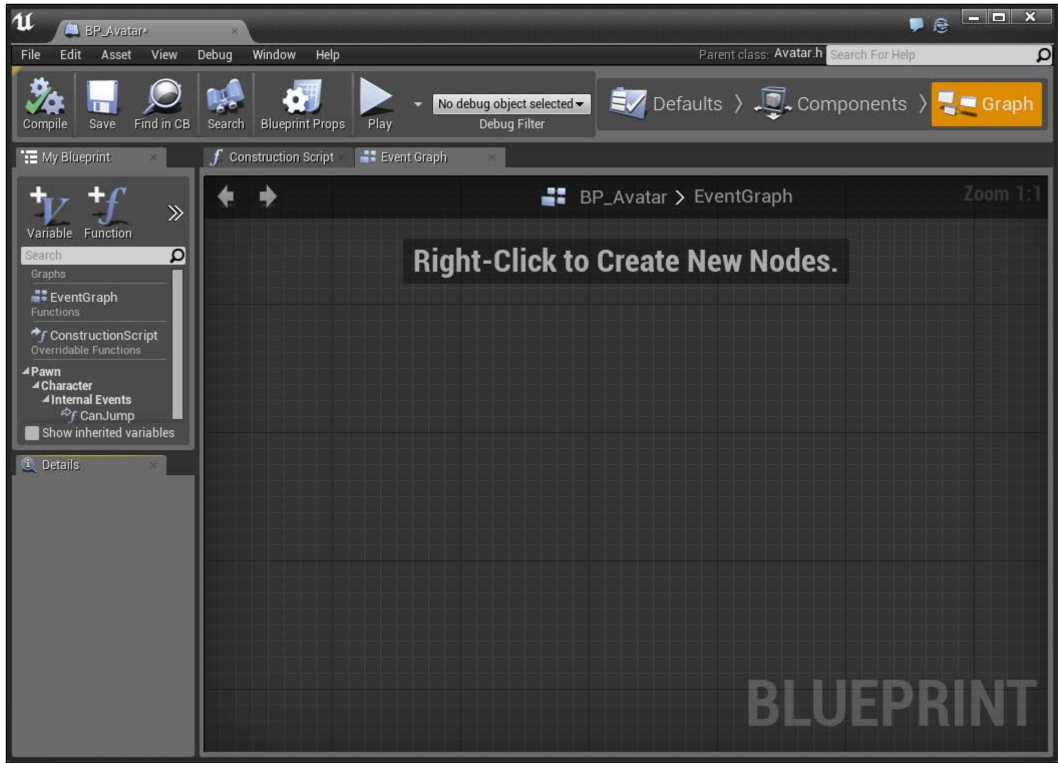
If your `Avatar` class does not show up, close the editor and compile/run the C++ project again.

3. Right-click on the class that you want to create a blueprint of (in my case, it's my **Avatar** class).
4. Name your blueprint something unique. I called my blueprint **BP_Avatar**.
5. Now, open this blueprint for editing, by double-clicking on **BP_Avatar** (it will appear in the **Class Viewer** tab after you add it, just under **Avatar**), as shown in the following screenshot:



www.it-ebooks.info

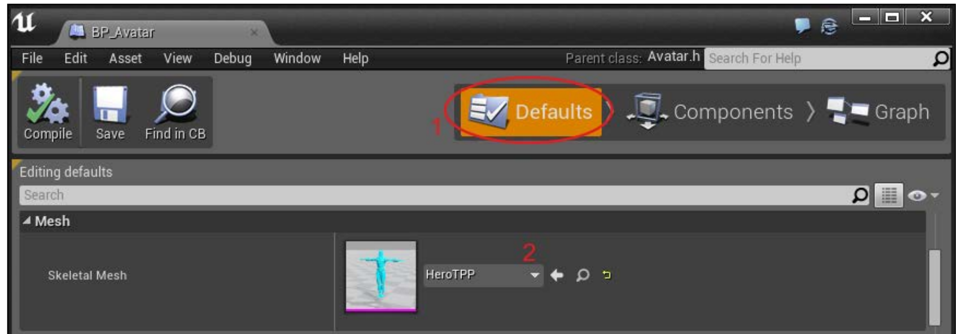
6. You will be presented with the blueprints window for your new **BP_Avatar** object, as shown here:



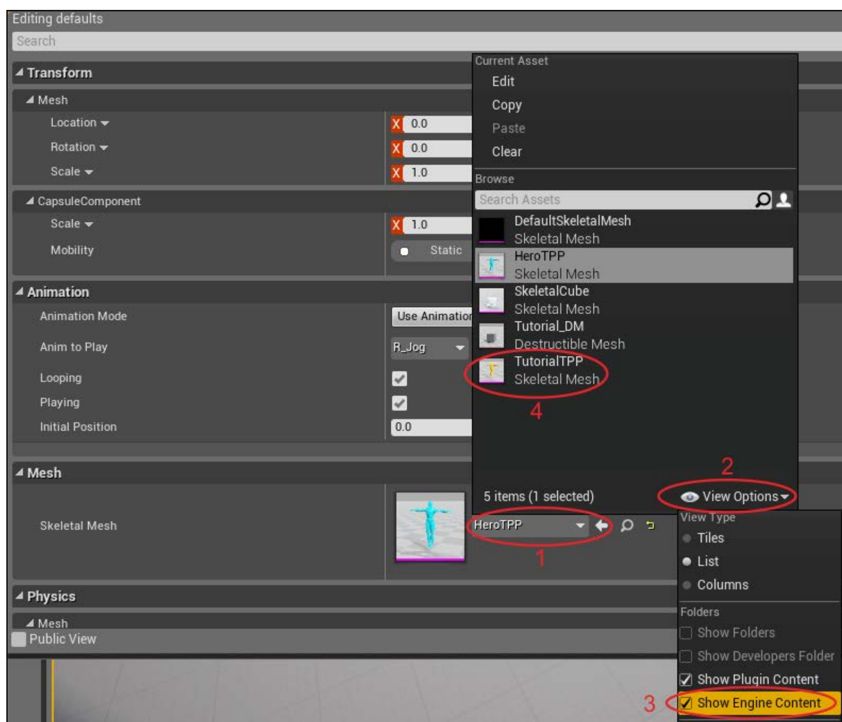
From this window, you can attach a model to the `Avatar` class visually. Again, this is the recommended pattern since artists will typically be the ones setting up their assets for game designers to play with.

www.it-ebooks.info

7. To set up the default mesh, click on the **Defaults** button at the top. Scroll down through the properties until you come across **Mesh**.

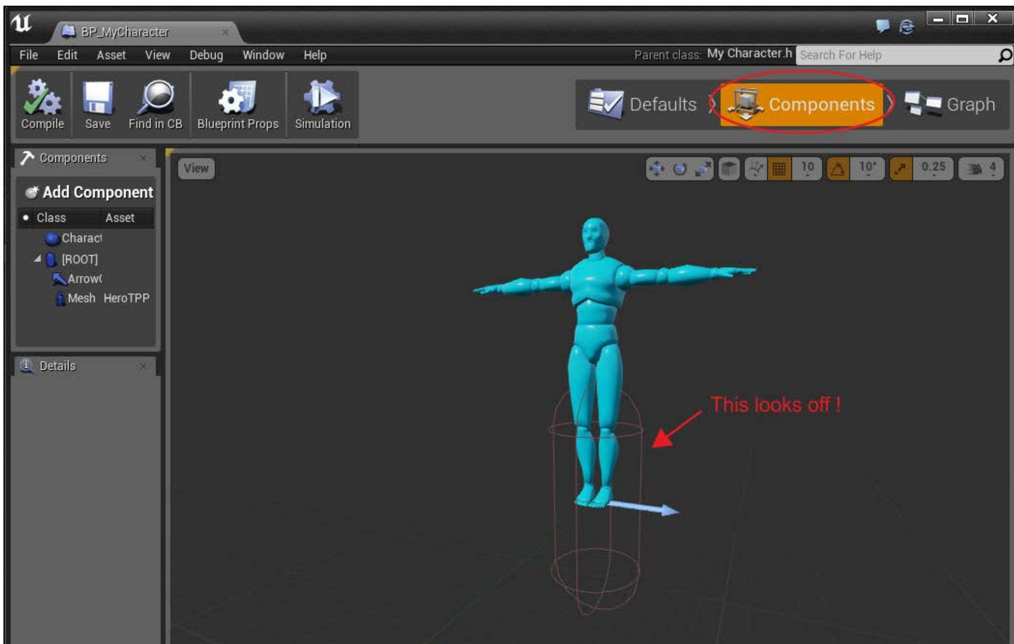


8. Click on the dropdown and select **HeroTPP** for your mesh, as shown in the preceding screenshot.
9. If **HeroTPP** doesn't appear in the dropdown, make sure that you download and add the **Animation Starter Pack** to your project. Alternatively, you can add the yellow **TutorialTPP** model to your project if you select **Show Engine Content** under **View Options**:



www.it-ebooks.info

10. What about the collision volume? Click on the **Components** tab in the blueprint editor for your avatar:

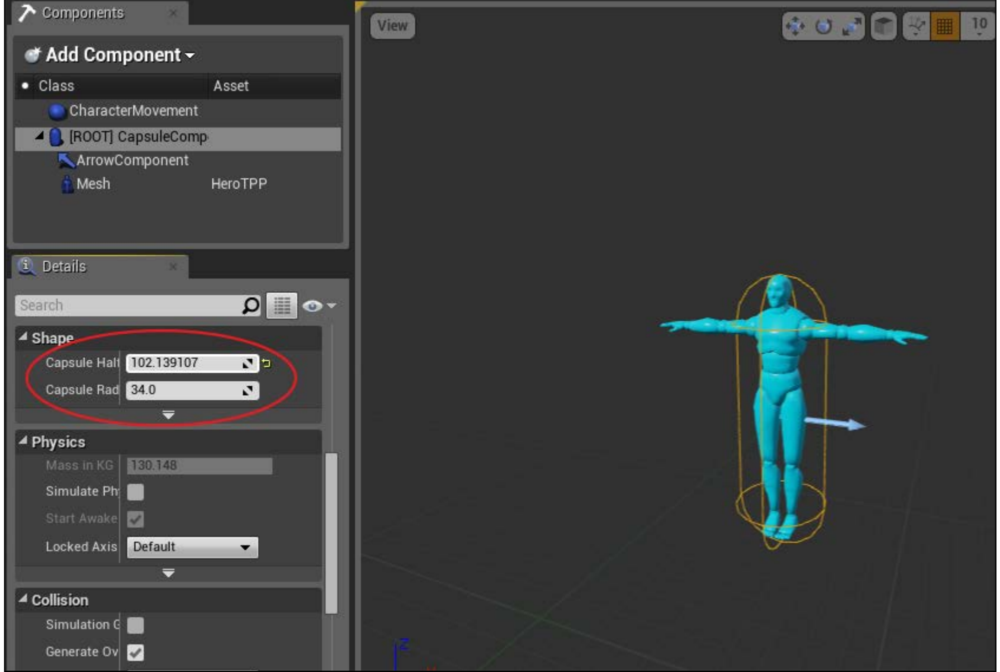


If your capsule doesn't encapsulate your model, adjust the model so that it fits

If your model ended up like mine, the capsule is off the mark! We need to adjust it.

www.it-ebooks.info

11. Click on the blue Avatar model and press the *W* key. Move him down until he fits inside the capsule. If the capsule isn't big enough, you can adjust its size in the **Details** tab under **Capsule Height** and **Capsule Radius**, as shown in the following screenshot:



You can stretch your capsule by adjusting the Capsule Height property

12. Now, we're ready to add this avatar to the game world. Click and drag your **BP_Avatar** model from the **Class Viewer** tab to your scene in the UE4 editor.

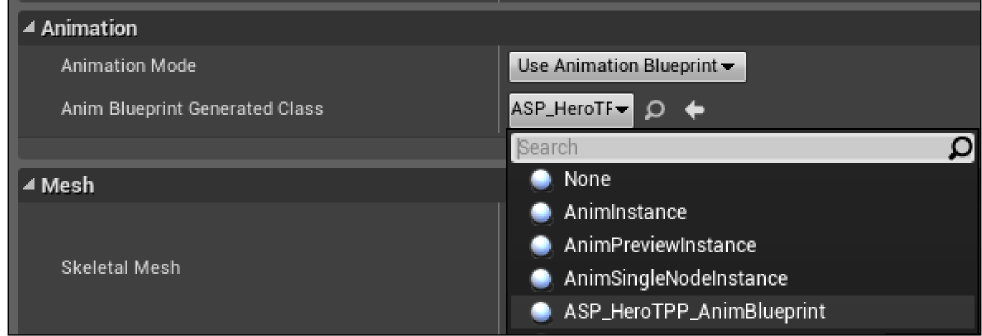
www.it-ebooks.info

Our Avatar class added to the scene, in a T-pose

The pose of Avatar is called the T-pose. Animators often leave their characters in this default pose. Animations can be applied to the character in order to make them change this default pose to something more interesting. You want him animated, you say! Well, that's easy.

Under the **Defaults** tab in the blueprint editor, just above **Mesh**, there is an **Animation** section where you can select the active animation on your **Mesh**. If you wish to use a certain animation asset, simply click on the drop-down menu and choose the animation you desire to show.

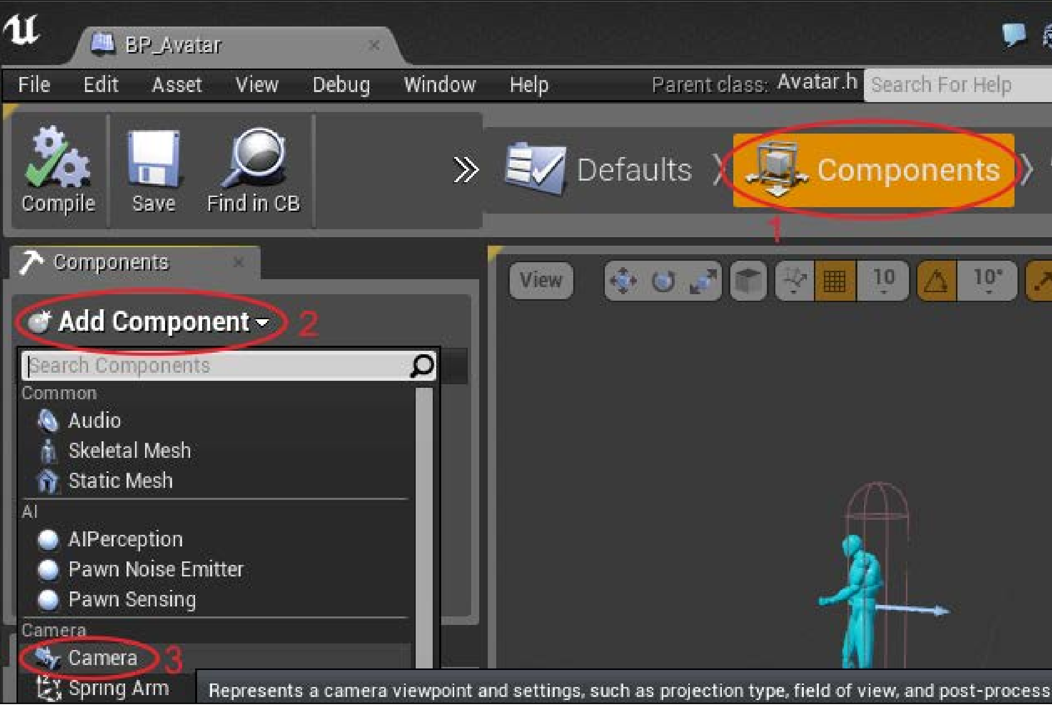
A better thing to do, however, is to use a blueprint for the animation. This way, an artist can properly set the animation based on what the character is doing. If you select **Use Animation Blueprint** from **Animation Mode** and then select **ASP_HeroTPP_AnimBlueprint** from the drop-down menu, the character will appear to behave much better in the game, because the animation will be adjusted by the blueprint (which would have been done by an artist) as the character moves.



www.it-ebooks.info

We can't cover everything here. Animation blueprints are covered in *Chapter 11, Monsters*. If you're really interested in animation, it also wouldn't be a bad idea to sit through a couple of Gnomon Workshop tutorials on IK, animation, and rigging, such as Alex Alvarez's *Rigging 101* class at `http://www.thegnomonworkshop.com/store/product/768/Rigging-101`.

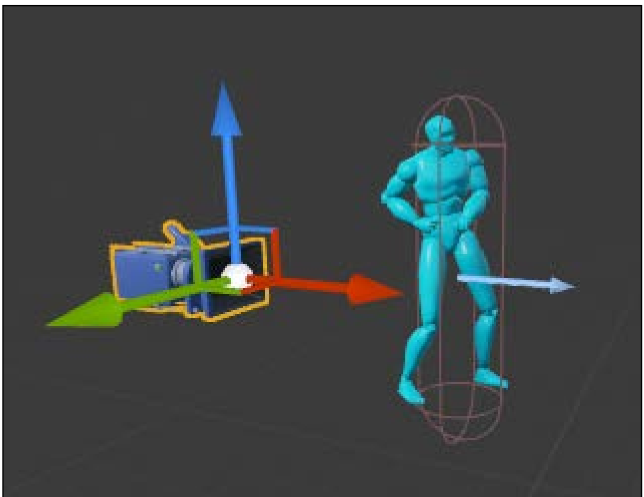
One more thing: let's make the camera for the Avatar appear behind it. This will give you a third person's point-of-view, which will allow you to see the whole character, as shown in the following screenshot with the corresponding steps:



1. In the **BP_Avatar** blueprint editor, click on the **Components** tab.
2. Click on **Add Component**.
3. Choose to add a **Camera**.

A camera will appear in the viewport. You can click on the camera and move it around. Position the camera so that it is somewhere behind the player. Make sure that the blue arrow on the player is facing the same direction as the camera. If it isn't, rotate the Avatar model mesh so that it faces the same direction as its blue-colored arrow.

www.it-ebooks.info



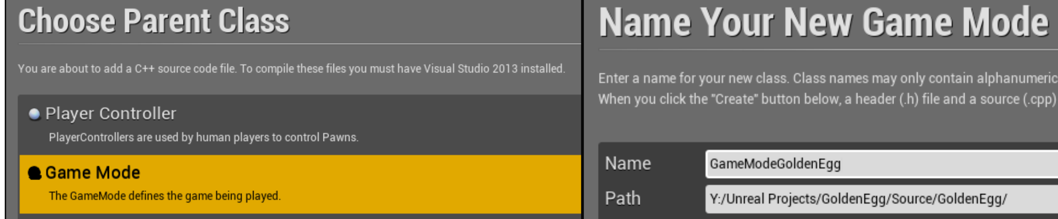
The blue-colored arrow on your model mesh indicates the forward direction for the model mesh. Make sure that the camera's opening faces the same direction as the character's forward vector

Writing C++ code that controls the game's character

When you launch your UE4 game, you might notice that the camera is a default, free-flying camera. What we will do now is make the starting character an instance of our `Avatar` class and control our character using the keyboard.

Making the player an instance of the Avatar class

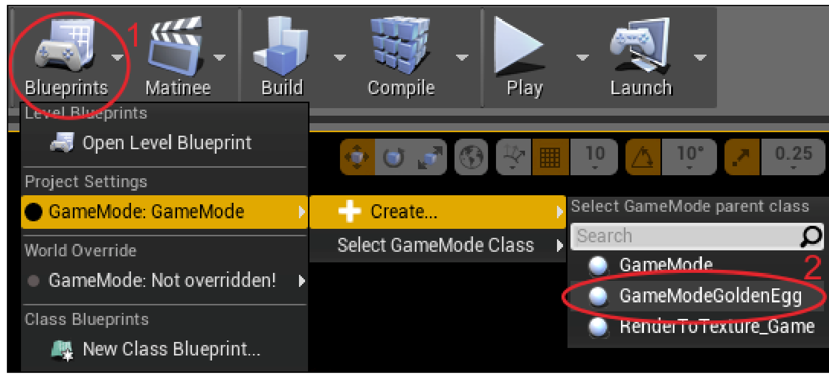
In the Unreal Editor, create a subclass of **Game Mode** by navigating to **File** | **Add Code To Project...** and selecting **Game Mode**. I named mine **GameModeGoldenEgg**.



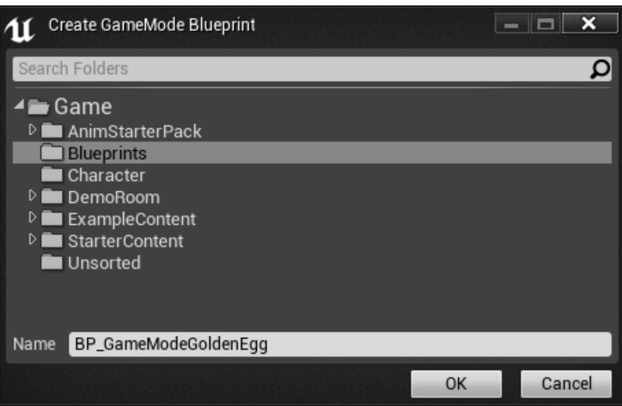
www.it-ebooks.info

The UE4 **GameMode** contains the rules of the game and describes how the game is played to the engine. We will work more with our `GameMode` class later. For now, we need to subclass it.

Recompile your project from Visual Studio, so you can create a **GameModeGoldenEgg** blueprint. Create the **GameMode** blueprint by going to the **Blueprints** icon in the menu bar at the top, clicking on **GameMode**, and then selecting **+ Create** | **GameModeGoldenEgg** (or whatever you named your **GameMode** subclass in step 1).



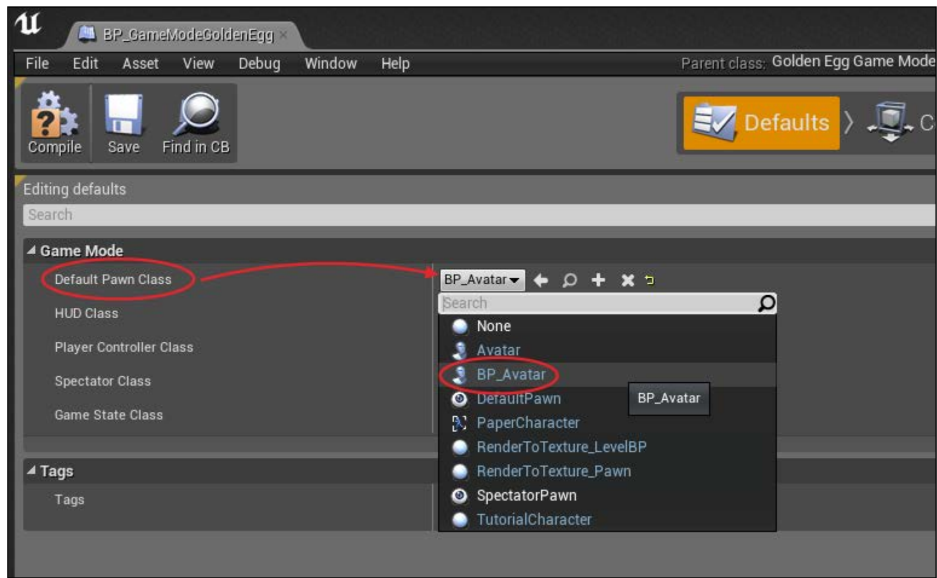
1. Name your blueprint; I called mine **BP_GameModeGoldenEgg**, as shown in the following screenshot:



2. Your newly created blueprint will open in the blueprint editor. If it doesn't, you can open the **BP_GameModeGoldenEgg** class from the **Class Viewer** tab.

www.it-ebooks.info

3. Select your **BP_Avatar** class from the **Default Pawn Class** panel, as shown in the following screenshot. The **Default Pawn Class** panel is the type of object that will be used for the player.



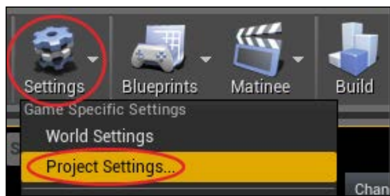
4. Now, launch your game. You can see a back view as the camera is placed behind the place, as shown here:

You'll notice that you can't move. Why is that? The answer is because we haven't set up the controller inputs yet.

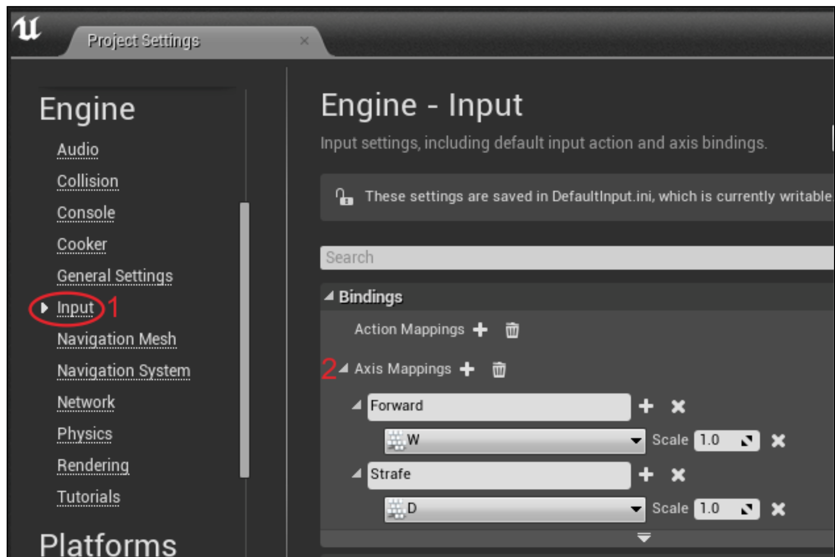
www.it-ebooks.info

Setting up controller inputs

1. To set up controller inputs, go to **Settings** | **Project Settings...**:



2. Next, in the left-hand side panel, scroll down until you see **Input** under **Engine**.



3. On the right-hand side, you can set up some **Bindings**. Click on the small arrow next to **Axis Mappings** in order to expand it. Add just two axis mappings to start, one called **Forward** (connected to the keyboard letter *W*) and one called **Strafe** (connected to the keyboard letter *D*). Remember the names that you set; we will look them up in C++ code in just a moment.
4. Close the **Project Settings** dialog. Now, open your C++ code.

www.it-ebooks.info

In the `Avatar.h` constructor, you need to add three member function declarations, as shown here:

```
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
  GENERATED_UCLASS_BODY()

  // New! These 3 new member function declarations
  // they will be used to move our player around!
  void SetupPlayerInputComponent(class UInputComponent*
  InputComponent) override;
  void MoveForward( float amount );
  void MoveRight( float amount );
};
```

Notice how the first member function we're adding (`SetupPlayerInputComponent`) is an override of a virtual function. `SetupPlayerInputComponent` is a virtual function in the `APawn` base class.

In the `Avatar.cpp` file, you need to put the function bodies. Add the following member function definitions:

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
InputComponent)
{
  check(InputComponent);
  InputComponent->BindAxis("Forward", this,
  &AAvatar::MoveForward);
  InputComponent->BindAxis("Strafe", this, &AAvatar::MoveRight);
}
```

This member function looks up the **Forward** and **Strafe** axis bindings that we just created in Unreal Editor and connects them to the member functions inside the `this` class. Which member functions should we connect to? Why, we should connect to `AAvatar::MoveForward` and `AAvatar::MoveRight`. Here are the member function definitions for these two functions:

```
void AAvatar::MoveForward( float amount )
{
  // Don't enter the body of this function if Controller is
  // not set up yet, or if the amount to move is equal to 0
  if( Controller && amount )
  {
    FVector fwd = GetActorForwardVector();
    // we call AddMovementInput to actually move the
```


www.it-ebooks.info

```
      // player by `amount` in the `fwd` direction
      AddMovementInput(fwd, amount);
    }
  }

  void AAvatar::MoveRight( float amount )
  {
    if( Controller && amount )
    {
      FVector right = GetActorRightVector();
      AddMovementInput(right, amount);
    }
  }
```

The `Controller` object and the `AddMovementInput` functions are defined in the `APawn` base class. Since the `Avatar` class derives from `ACharacter`, which in turn derives from `APawn`, we get free use of all the member functions in the base class `APawn`. Now do you see the beauty of inheritance and code reuse?

Exercise

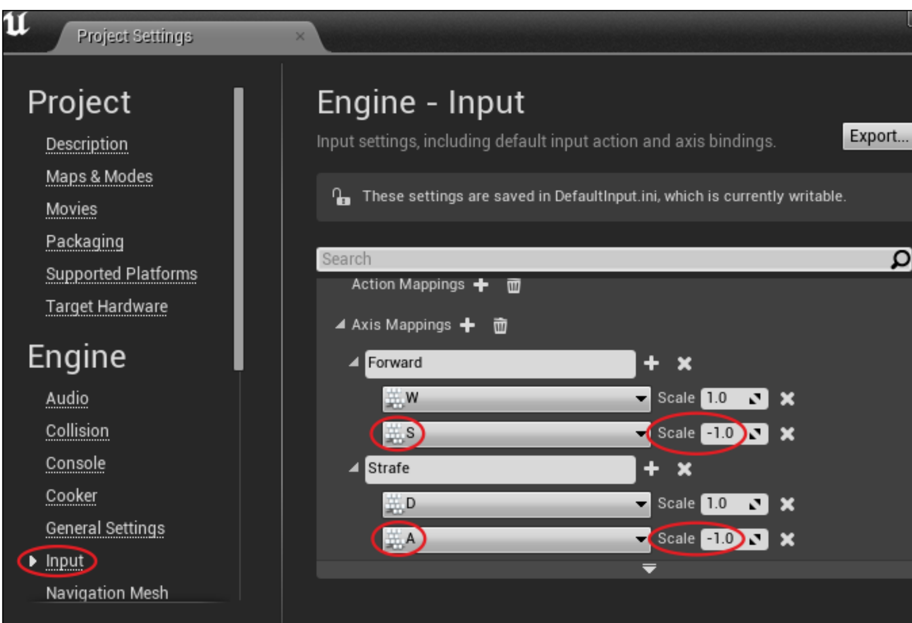
Add axis bindings and C++ functions to move the player to the left and back.

Here' a hint: you only need to add axis bindings if you realize going backwards is only the negative of going forward.

Solution

Enter two extra axis bindings by navigating to **Settings** | **Project Settings...** | **Input**, as shown here:

www.it-ebooks.info



Scale the **S** and **A** inputs by -1.0. This will invert the axis. So pressing the *S* key in the game will move the player forward. Try it!

Alternatively, you can define two completely separate member functions in your `AAvatar` class, as follows, and bind the *A* and *S* keys to `AAvatar::MoveLeft` and `AAvatar::MoveBack`, respectively:

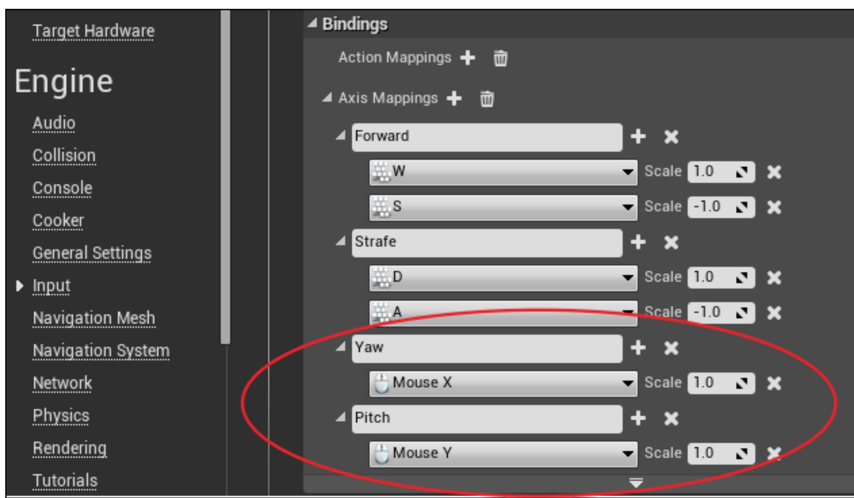
```
void AAvatar::MoveLeft( float amount )
{
  if( Controller && amount )
  {
    FVector left = -GetActorRightVector();
    AddMovementInput(left, amount);
  }
}
void AAvatar::MoveBack( float amount )
{
  if( Controller && amount )
  {
    FVector back = -GetActorForwardVector();
    AddMovementInput(back, amount);
  }
}
```


www.it-ebooks.info

Yaw and pitch

We can change the direction in which the player looks by setting the yaw and pitch of the controller.

All we have to do here is add in new axis bindings for the mouse, as shown in the following screenshot:



From C++, you need to add in two new member function declarations to `AAvatar.h`:

```
void Yaw( float amount );
void Pitch( float amount );
```

The bodies of these member functions will go in the `AAvatar.cpp` file:

```
void AAvatar::Yaw( float amount )
{
  AddControllerYawInput(200.f * amount * GetWorld()-
  >GetDeltaSeconds());
}
void AAvatar::Pitch( float amount )
{
  AddControllerPitchInput(200.f * amount * GetWorld()-
  >GetDeltaSeconds());
}
```

Then, add two lines to `SetupPlayerInputComponent`:

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
  InputComponent)
```


www.it-ebooks.info

```
{
  // .. as before, plus:
  InputComponent->BindAxis("Yaw", this, &AAvatar::Yaw);
  InputComponent->BindAxis("Pitch", this, &AAvatar::Pitch);
}
```

Here, notice how I've multiplied the `amount` values in the `Yaw` and `Pitch` functions by 200. This number represents the mouse's sensitivity. You can (should) add a `float` member to the `AAvatar` class in order to avoid hardcoding this sensitivity number.

`GetWorld()->GetDeltaSeconds()` gives you the amount of time that passed between the last frame and this frame. It isn't a lot: `GetDeltaSeconds()` should be around 16 milliseconds (0.016 s) most of the time (if your game is running at 60 fps).

So, now we have player input and control. To add new functionality to your Avatar, this is all that you have to do:

1. Bind your key or mouse actions by going to **Settings** | **Project Settings** | **Input**.
2. Add a member function to run when that key is pressed.
3. Add a line to `SetupPlayerInputComponent`, connecting the name of the bound input to the member function we want to run when that key is pushed.

Creating non-player character entities

So, we need to create a few **NPC**s (**non-playable characters**). NPCs are characters within the game that help the player. Some offer special items, some are shop vendors, and some have information to give to the player. In this game, they will react to the player as he gets near. Let's program in some of this behavior.

First, create another subclass of **Character**. In the UE4 Editor, go to **File** | **Add Code To Project...** and choose the **Character** class from which you can make a subclass. Name your subclass `NPC`.

Now, edit your code in Visual Studio. Each NPC will have a message to tell the player, so we add in a `UPROPERTY() FString` property to the `NPC` class.

`FStrings` are UE4's version of C++'s `<string>` type. When programming in UE4, you should use the `FString` objects over C++ STL's `string` objects. In general, you should preferably use UE4's built-in types, as they guarantee cross-platform compatibility.

www.it-ebooks.info

How to add the `UPROPERTY()` `FString` property to the `NPC` class is shown in the following code:

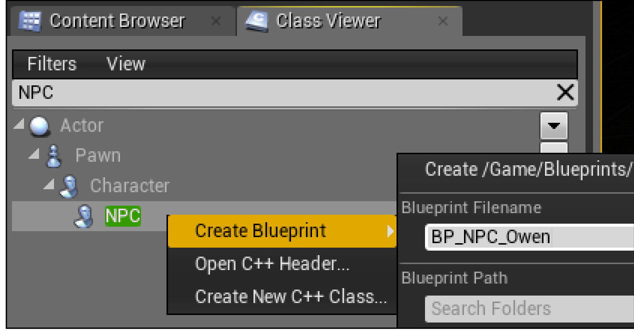
```
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
  GENERATED_UCLASS_BODY()
  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  Collision)
  TSubobjectPtr<class USphereComponent> ProxSphere;
  // This is the NPC's message that he has to tell us.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  NPCMessage)
  FString NpcMessage;
  // When you create a blueprint from this class, you want to be
  // able to edit that message in blueprints,
  // that's why we have the EditAnywhere and BlueprintReadWrite
  // properties.
}
```

Notice that we put the `EditAnywhere` and `BlueprintReadWrite` properties into the `UPROPERTY` macro. This will make the `NpcMessage` editable in blueprints.

Full descriptions of all the UE4 property specifiers are available at `https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/Properties/index.html`.

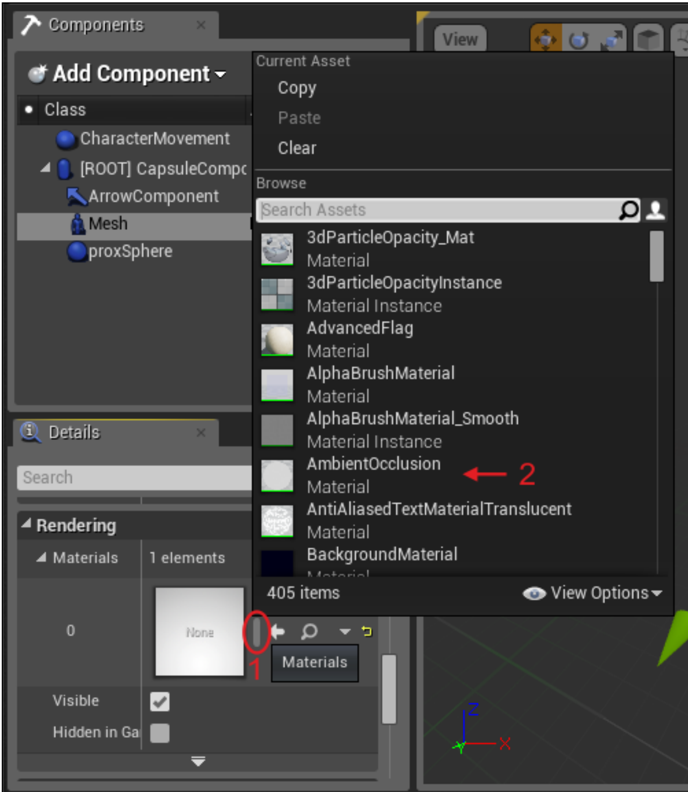
Recompile your project (as we did for the `Avatar` class). Then, go to the **Class Viewer**, right click on your `NPC` class, and create a blueprint from it.

Each NPC character you want to create can be a blueprint based off of the `NPC` class. Name each blueprint something unique, as we'll be selecting a different model mesh and message for each NPC that appears, as shown in the following screenshot:



www.it-ebooks.info

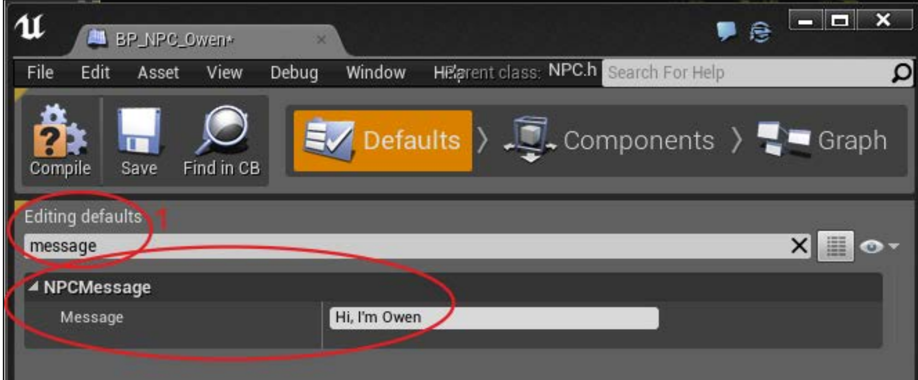
Now, open the blueprint, select skeletal **mesh** from the **Add Components**, and adjust the capsule (as we did for **BP_Avatar**). You can also change the material of your new character so that he looks different from the player.



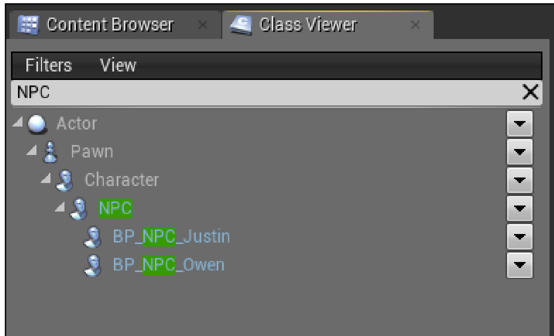
Change the material of your character in your mesh's properties. Under the Rendering tab, click on the + icon to add a new material. Then, click on the small capsule-shaped item to select a material to render with.

www.it-ebooks.info

In the **Defaults** tab, search for the `NpcMessage` property. This is our connection between C++ code and blueprints: because we entered a `UPROPERTY()` function on the `FString NpcMessage` variable, that property appears editable within UE4, as shown in the following screenshot:

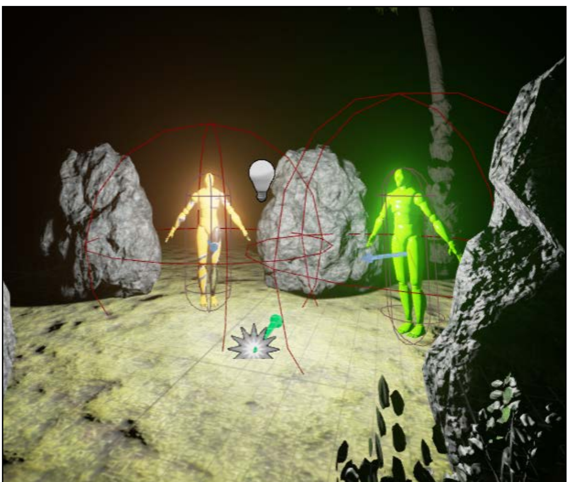


Now, drag **BP_NPC_Owen** into the scene. You can create a second or third character as well, and be sure to give them unique names, appearances, and messages.



I've created two blueprints for NPCs based on the NPC base classes, BP_NPC_Justin and BP_NPC_Owen. They have different appearances and different messages for the player.

www.it-ebooks.info



Justin and Owen in the scene

Displaying a quote from each NPC dialog box

To display a dialog box, we need a custom (heads-up display) **HUD**. In the UE4 editor, go to **File | Add Code To Project...** and choose the `HUD` class from which the subclass is created. Name your subclass as you wish; I've named mine `MyHUD`.

After you have created the `MyHUD` class, let Visual Studio reload. We will make some code edits.

Displaying messages on the HUD

Inside the `AMyHUD` class, we need to implement the `DrawHUD()` function in order to draw our messages to the HUD and to initialize a font to draw to the HUD with, as shown in the following code:

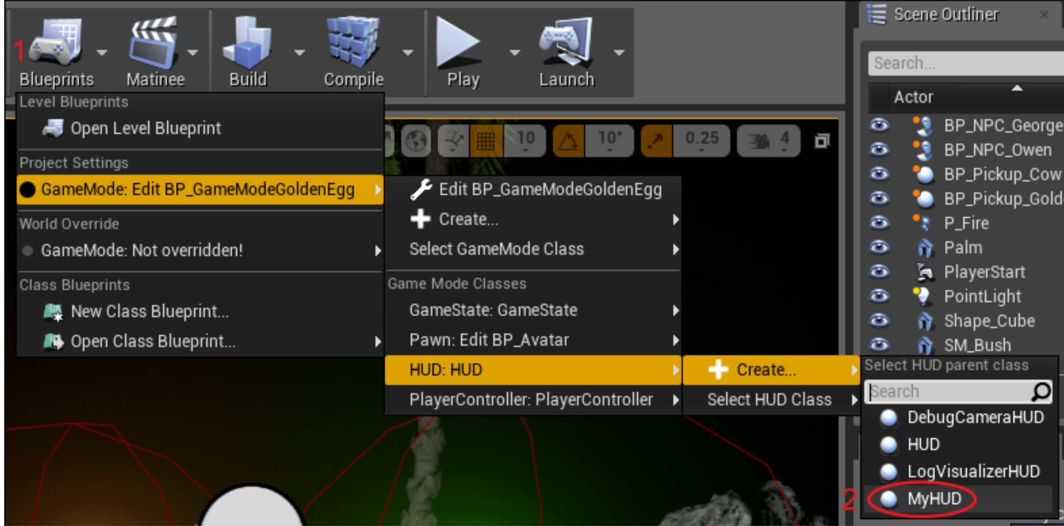
```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
  GENERATED_UCLASS_BODY()
  // The font used to render the text in the HUD.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
  UFont* hudFont;
  // Add this function to be able to draw to the HUD!
  virtual void DrawHUD() override;
};
```


www.it-ebooks.info

The HUD font will be set in a blueprinted version of the `AMyHUD` class. The `DrawHUD()` function runs once per frame. In order to draw within the frame, add a function to the `AMyHUD.cpp` file:

```
void AMyHUD::DrawHUD()
{
  // call superclass DrawHUD() function first
  Super::DrawHUD();
  // then proceed to draw your stuff.
  // we can draw lines..
  DrawLine( 200, 300, 400, 500, FLinearColor::Blue );
  // and we can draw text!
  DrawText( "Greetings from Unreal!", FVector2D( 0, 0 ), hudFont,
  FVector2D( 1, 1 ), FColor::White );
}
```

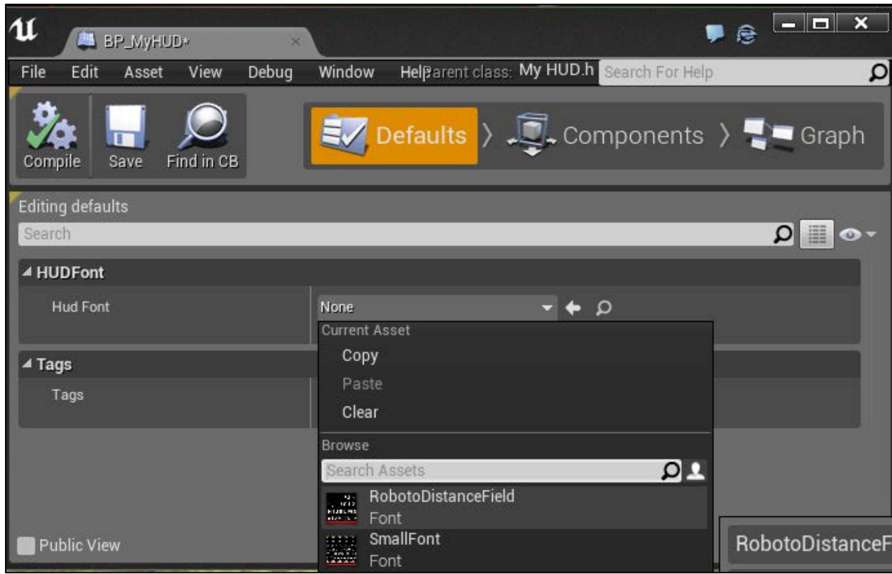
Wait! We haven't initialized our font yet. To do this, we need to set it up in blueprints. Compile and run your Visual Studio project. Once you are in the editor, go to the **Blueprints** menu at the top and navigate to **GameMode** | **HUD** | **+ Create** | **MyHUD**.



Creating a blueprint of the MyHUD class

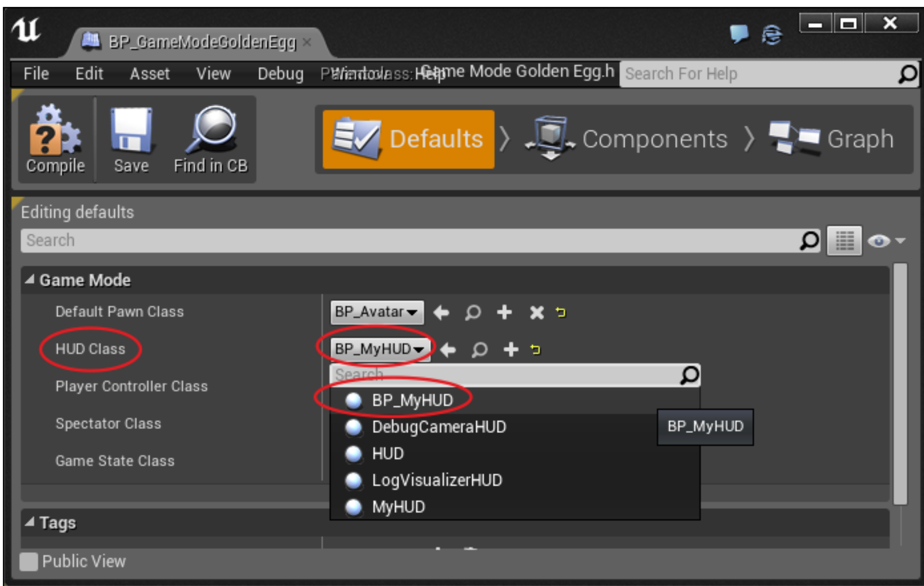
www.it-ebooks.info

I called mine `BP_MyHUD`. Edit `BP_MyHUD` and select a font from the drop-down menu under **HUDFont**:



I selected RobotoDistanceField for my HUD font

Next, edit your **Game Mode** blueprint (**BP_GameModeGoldenEgg**) and select your new `BP_MyHUD` (not `MyHUD` class) for the **HUD Class** panel:



www.it-ebooks.info

Test your program by running it! You should see text printed on the screen.

Using TArray<Message>

Each message we want to display for the player will have a few properties:

- An FString variable for the message
- A float variable for the time to display it
- An FColor variable for the color of the message

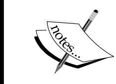
So it makes sense for us to write a little struct function to contain all this information.

At the top of MyHUD.h, insert the following struct declaration:

```
struct Message
{
  FString message;
  float time;
  FColor color;
  Message()
  {
    // Set the default time.
    time = 5.f;
    color = FColor::White;
  }
  Message( FString iMessage, float iTime, FColor iColor )
  {
```


www.it-ebooks.info

```
        message = iMessage;
        time = iTime;
        color = iColor;
    }
};
```



An enhanced version of the `Message` structure (with a background color) is in the code package for this chapter. We used simpler code here so that it'd be easier to understand the chapter.

Now, inside the `AMyHUD` class, we want to add a `TArray` of these messages. A `TArray` is a UE4-defined special type of dynamically growable C++ array. We will cover the detailed use of `TArray` in the next chapter, but this simple usage of `TArray` should be a nice introduction to garner your interest in the usefulness of arrays in games. This will be declared as `TArray<Message>`:

```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
  GENERATED_UCLASS_BODY()
  // The font used to render the text in the HUD.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
  UFont* hudFont;
  // New! An array of messages for display
  TArray<Message> messages;
  virtual void DrawHUD() override;
  // New! A function to be able to add a message to display
  void addMessage( Message msg );
};
```

Now, whenever the NPC has a message to display, we're just need to call `AMyHud::addMessage()` with our message. The message will be added to `TArray` of the messages to be displayed. When a message expires (after a certain amount of time), it will be removed from the HUD.

Inside the `AMyHUD.cpp` file, add the following code:

```
void AMyHUD::DrawHUD()
{
  Super::DrawHUD();
  // iterate from back to front thru the list, so if we remove
  // an item while iterating, there won't be any problems
  for( int c = messages.Num() - 1; c >= 0; c-- )
  {
    // draw the background box the right size
```


www.it-ebooks.info

```
      // for the message
      float outputWidth, outputHeight, pad=10.f;
      GetTextSize( messages[c].message, outputWidth, outputHeight,
      hudFont, 1.f );

      float messageH = outputHeight + 2.f*pad;
      float x = 0.f, y = c*messageH;

      // black backing
      DrawRect( FLinearColor::Black, x, y, Canvas->SizeX, messageH
      );
      // draw our message using the hudFont
      DrawText( messages[c].message, messages[c].color, x + pad, y +
      pad, hudFont );

      // reduce lifetime by the time that passed since last
      // frame.
      messages[c].time -= GetWorld()->GetDeltaSeconds();

      // if the message's time is up, remove it
      if( messages[c].time < 0 )
      {
        messages.RemoveAt( c );
      }
    }
  }

  void AMyHUD::addMessage( Message msg )
  {
    messages.Add( msg );
  }
```

The `AMyHUD::DrawHUD()` function now draws all the messages in the `messages` array, and arranges each message in the `messages` array by the amount of time that passed since the last frame. Expired messages are removed from the `messages` collection once their `time` value drops below 0.

Exercise

Refactor the `DrawHUD()` function so that the code that draws the messages to the screen is in a separate function called `DrawMessages()`.

The `Canvas` variable is only available in `DrawHUD()`, so you will have to save `Canvas->SizeX` and `Canvas->SizeY` in class-level variables.

www.it-ebooks.info

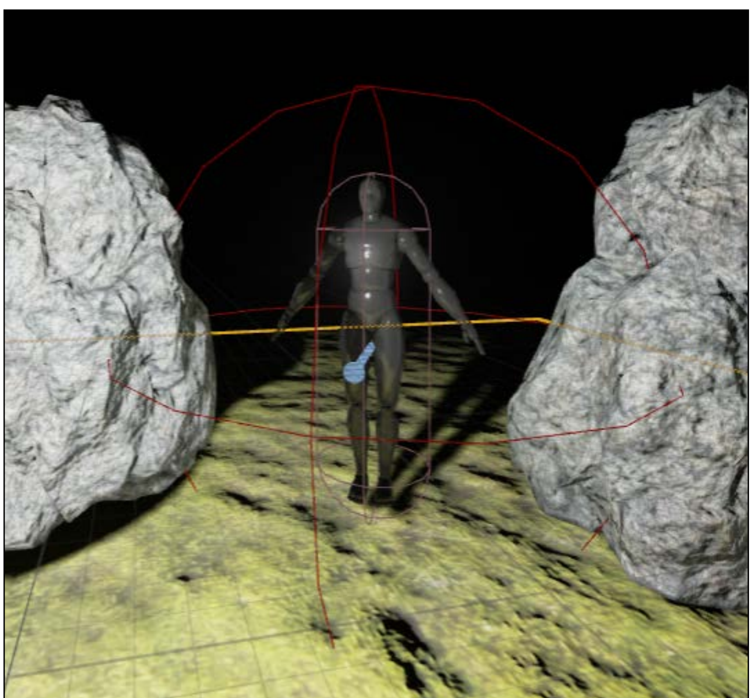
Refactoring means to change the way code works internally so that it is more organized or easier to read but still has the same apparent result to the user running the program. Refactoring often is a good practice. The reason why refactoring occurs is because nobody knows exactly what the final code should look like once they start writing it.

Solution

See the `AMyHUD::DrawMessages()` function in the code package for this chapter.

Triggering an event when it is near an NPC

To trigger an event near the NPC, we need to set an additional collision detection volume that is a bit wider than the default capsule shape. The additional collision detection volume will be a sphere around each NPC. When the player steps into the NPC sphere, the NPC reacts and displays a message.



We're going to add the dark red sphere to the NPC so that he can tell when the player is nearby

www.it-ebooks.info

Inside your `NPC.h` class file, add the following code in order to declare `ProxSphere` and `UFUNCTION` called `Prox`:

```
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
  GENERATED_UCLASS_BODY()
  // This is the NPC's message that he has to tell us.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  NPCMessage)
  FString NpcMessage;
  // The sphere that the player can collide with to get item
  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  Collision)
  TSubobjectPtr<class USphereComponent> ProxSphere;
  // The corresponding body of this function is
  // ANPC::Prox_Implementation, __not__ ANPC::Prox()!
  // This is a bit weird and not what you'd expect,
  // but it happens because this is a BlueprintNativeEvent
  UFUNCTION(BlueprintNativeEvent, Category = "Collision")
  void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp,
  int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
  SweepResult );
};
```

This looks a bit messy, but it is actually not that complicated. Here, we declare an extra bounding sphere volume called `ProxSphere`, which detects when the player is near the NPC.

In the `NPC.cpp` file, we need to add the following code in order to complete the proximity detection:

```
ANPC::ANPC(const class FPostConstructInitializeProperties& PCIP) :
Super(PCIP)
{
  ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this,
  TEXT("Proximity Sphere"));
  ProxSphere->AttachTo( RootComponent );
  ProxSphere->SetSphereRadius( 32.f );
  // Code to make ANPC::Prox() run when this proximity sphere
  // overlaps another actor.
  ProxSphere->OnComponentBeginOverlap.AddDynamic( this,
  &ANPC::Prox );
  NpcMessage = "Hi, I'm Owen";//default message, can be edited
  // in blueprints
}
```


www.it-ebooks.info

```
// Note! Although this was declared ANPC::Prox() in the header,
// it is now ANPC::Prox_Implementation here.
void ANPC::Prox_Implementation( AActor* OtherActor,
  UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
  bFromSweep, const FHitResult & SweepResult )
{
  // This is where our code will go for what happens
  // when there is an intersection
}
```

Make the NPC display something to the HUD when something is nearby

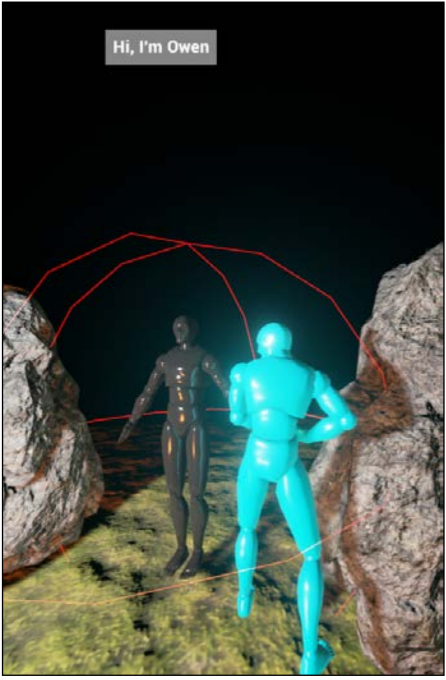
When the player is near the NPC sphere collision volume, display a message to the HUD that alerts the player about what the NPC is saying.

This is the complete implementation of `ANPC::Prox_Implementation`:

```
void ANPC::Prox_Implementation( AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult & SweepResult )
{
  // if the overlapped actor is not the player,
  // you should just simply return from the function
  if( Cast<AAvatar>( OtherActor ) == nullptr )
  {
    return;
  }
  APlayerController* PController = GetWorld()-
  >GetFirstPlayerController();
  if( PController )
  {
    AMyHUD * hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->addMessage( Message( NpcMessage, 5.f, FColor::White ) );
  }
}
```


www.it-ebooks.info

The first thing we do in this function is to cast `OtherActor` (the thing that came near the NPC) to `AAvatar`. The cast succeeds (and is not `nullptr`) when `OtherActor` is an `AAvatar` object. We get the HUD object (which happens to be attached to the player controller) and pass a message from the NPC to the HUD. The message is displayed whenever the player is within the red bounding sphere surrounding the NPC.



Owen's greeting

Exercises

1. Add a `UPROPERTY` function name for the NPC's name so that the name of the NPC is editable in blueprints, similar to the message that the NPC has for the player. Show the NPC's name in the output.
2. Add a `UPROPERTY` function (type `UTexture2D*`) for the NPC's face texture. Draw the NPC's face beside its message in the output.
3. Render the player's HP as a bar (filled rectangle).

www.it-ebooks.info

Solutions

Add this property to the `ANPC` class:

```
// This is the NPC's name
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
FString name;
```

Then, in `ANPC::Prox_Implementation`, change the string passed to the HUD to:

```
name + FString(": ") + message
```

This way, the NPC's name will be attached to the message.

Add `this` property to the `ANPC` class:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
UTexture2D* Face;
```

Then you can select face icons to be attached to the NPC's face in blueprints.

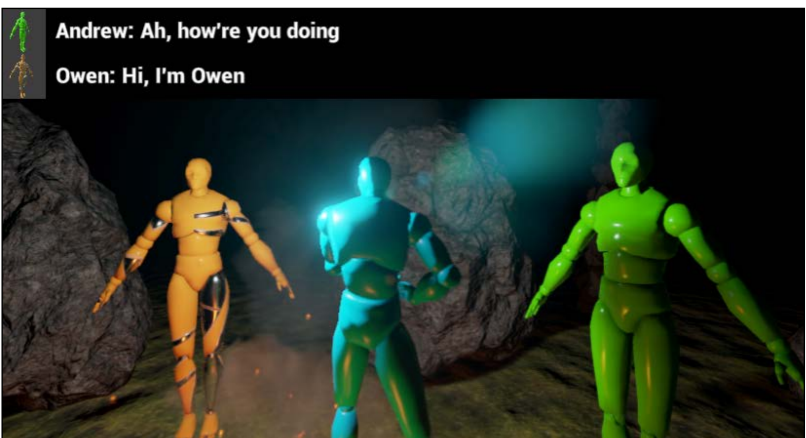
Attach a texture to your `struct Message`:

```
UTexture2D* tex;
```

To render these icons, you need to add a call to `DrawTexture()` with the right texture passed in to it:

```
DrawTexture( messages[c].tex, x, y, messageH, messageH, 0, 0, 1, 1
  );
```

Be sure to check whether the texture is valid before you render it. The icons should look similar to what is shown here, at the top of the screen:



www.it-ebooks.info

This is how a function to draw the player's remaining health in a bar will look:

```
void AMyHUD::DrawHealthbar()
{
  // Draw the healthbar.
  AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  float barWidth=200, barHeight=50, barPad=12, barMargin=50;
  float percHp = avatar->Hp / avatar->MaxHp;
  DrawRect( FLinearColor( 0, 0, 0, 1 ), Canvas->SizeX - barWidth -
  barPad - barMargin, Canvas->SizeY - barHeight - barPad -
  barMargin, barWidth + 2*barPad, barHeight + 2*barPad );
  DrawRect( FLinearColor( 1-percHp, percHp, 0, 1 ), Canvas->SizeX
  - barWidth - barMargin, Canvas->SizeY - barHeight - barMargin,
  barWidth*percHp, barHeight );
}
```

Summary

In this chapter, we went through a lot of material. We showed you how to create a character and display it on the screen, control your character with axis bindings, and create and display NPCs that can post messages to the HUD.

In the upcoming chapters, we will develop our game further by adding an *Inventory System and Pickup Items* in *Chapter 10*, as well as the code and the concept to account for what the player is carrying. Before we do that, though, we will do an in-depth exploration of some of the UE4 container types in *Chapter 9*, *Templates and Commonly Used Containers*.

www.it-ebooks.info

9
Templates and Commonly Used Containers

In *Chapter 7*, *Dynamic Memory Allocation*, we spoke about how you will use dynamic memory allocation if you want to create a new array whose size isn't known at compile time. Dynamic memory allocations are of the form `int * array = new int[ number_of_elements ]`.

You also saw that dynamic allocations using the `new[]` keyword require you to call `delete[]` on the array later, otherwise you'd have a memory leak. Having to manage memory this way is hard work.

Is there a way to create an array of dynamic size and have the memory automatically managed for you by C++? The answer is yes. There are C++ object types (commonly called containers) that handle dynamic memory allocations and deallocations automatically. UE4 provides a couple of container types to store your data in dynamically resizable collections.

There are two different groups of template containers. There is the UE4 family of containers (beginning with `T*`) and the C++ **Standard Template Library** (**STL**) family of containers. There are some differences between the UE4 containers and the C++ STL containers, but the differences are not major. UE4 containers sets are written with game performance in mind. C++ STL containers also perform well, and their interfaces are a little more consistent (consistency in an API is something that you'd prefer). Which container set you use is up to you. However, it is recommended that you use the UE4 container set since it guarantees that you won't have cross-platform issues when you try to compile your code.

www.it-ebooks.info

Debugging the output in UE4

All of the code in this chapter (as well as in the later chapters) will require you to work in a UE4 project. For the purpose of testing TArray, I created a basic code project called TArrays. In the ATArraysGameMode::ATArraysGameMode constructor, I am using the debug output feature to print text to the console.

Here's how the code will look:

```
ATArraysGameMode::ATArraysGameMode(const class
FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
  if( GEngine )
  {
    GEngine->AddOnScreenDebugMessage( 0, 30.f, FColor::Red,
    "Hello!" );
  }
}
```

If you compile and run this project, you will see the debug text in the top-left corner of your game window when you start the game. You can use a debug output to see the internals of your program at any time. Just make sure that the GEngine object exists at the time of debugging the output. The output of the preceding code is shown in the following screenshot:

UE4's TArray<T>

TArrays are UE4's version of a dynamic array. To understand what a TArray<T> variable is, you first have to know what the <T> option between angle brackets stands for. The <T> option means that the type of data stored in the array is a variable. Do you want an array of int? Then create a TArray<int> variable. A TArray variable of double? Create a TArray<double> variable.

So, in general, wherever a <T> appears, you can plug in a C++ type of your choice. Let's move on and show this with an example.

www.it-ebooks.info

An example that uses TArray<T>

A `TArray<int>` variable is just an array of `int`. A `TArray<Player*>` variable will be an array of `Player*` pointers. An array is dynamically resizable, and elements can be added at the end of the array after its creation.

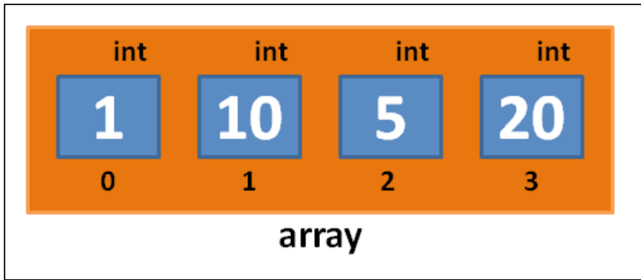
To create a `TArray<int>` variable, all you have to do is use the normal variable allocation syntax:

```
TArray<int> array;
```

Changes to the `TArray` variable are done using member functions. There are a couple of member functions that you can use on a `TArray` variable. The first member function that you need to know about is the way you add a value to the array, as shown in the following code:

```
array.Add( 1 );
array.Add( 10 );
array.Add( 5 );
array.Add( 20 );
```

These four lines of code will produce the array value in memory, as shown in the following figure:



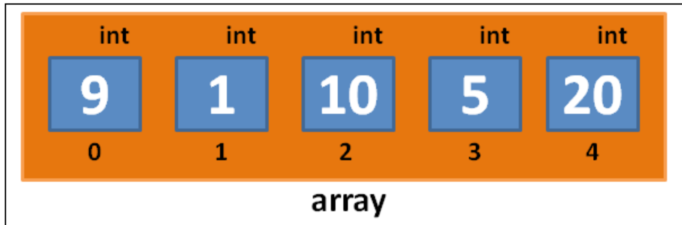
When you call `array.Add( number )`, the new number goes to the end of the array. Since we added the numbers **1**, **10**, **5**, and **20** to the array, in this order, that is the order in which they will go into the array.

If you want to insert a number in the front or middle of the array, it is also possible. All you have to do is use the `array.Insert(value, index)` function, as shown in the following line of code:

```
array.Insert( 9, 0 );
```


www.it-ebooks.info

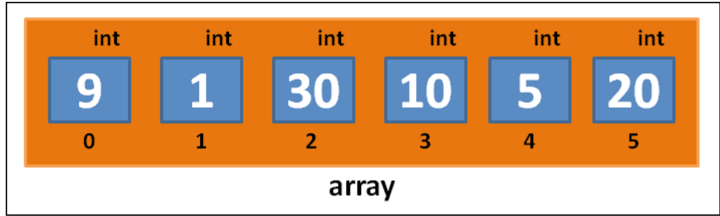
This function will push the number **9** into the position **0** of the array (at the front). This means that the rest of the array elements will be offset to the right, as shown in the following figure:



We can insert another element into position **2** of the array using the following line of code:

```
array.Insert( 30, 2 );
```

This function will rearrange the array as shown in the following figure:



If you insert a number into a position in the array that is out of bounds, UE4 will crash. So be careful not to do that.

Iterating a TArray

You can iterate (walk over) the elements of a `TArray` variable in two ways: either using integer-based indexing or using an iterator. I will show you both the ways here.

The vanilla for loop and square brackets notation

Using integers to index the elements of an array is sometimes called a "vanilla" `for` loop. The elements of the array can be accessed using `array[ index ]`, where `index` is the numerical position of the element in the array:

```
for( int index = 0; index < array.Num(); index++ )
{
  // print the array element to the screen using debug message
```


www.it-ebooks.info

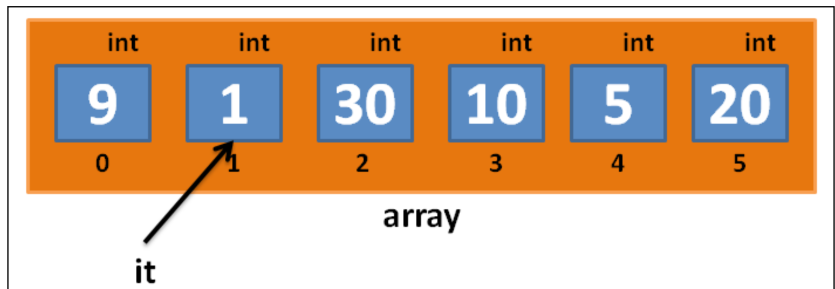
```
    GEngine->AddOnScreenDebugMessage( index, 30.f, FColor::Red,
    FString::FromInt( array[ index ] ) );
  }
```

Iterators

You can also use an iterator to walk over the elements of the array one by one, as shown in the following code:

```
int count = 0;   // keep track of numerical index in array
for( TArray<int>::TIterator it = array.CreateIterator(); it; ++it
  )
{
  GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red,
  FString::FromInt( *it ) );
}
```

Iterators are pointers into the array. Iterators can be used to inspect or change values inside the array. An example of an iterator is shown in the following figure:



The concept of an iterator: it is an external object that can look into and inspect the values of an array. Doing ++ it moves the iterator to examine the next element.

An iterator must be suitable for the collection it is walking through. To walk through a `TArray<int>` variable, you need a `TArray<int>::TIterator` type iterator.

We use `*` to look at the value behind an iterator. In the preceding code, we used `(*it)` to get the integer value from the iterator. This is called dereferencing. To dereference an iterator means to look at its value.

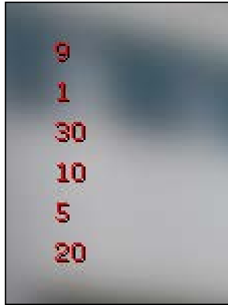
The `++it` operation that happens at the end of each iteration of the `for` loop increments the iterator, moving it on to point to the next element in the list.

www.it-ebooks.info

Insert the code into the program and try it out now. Here's the example program we have created so far using `TArray` (all in the `ATArraysGameMode::ATArraysGameMode()` constructor):

```
ATArraysGameMode::ATArraysGameMode(const class
FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
  TArray<int> array;
  array.Add( 1 );
  array.Add( 10 );
  array.Add( 5 );
  array.Add( 20 );
  array.Insert( 9, 0 );// put a 9 in the front
  array.Insert( 30, 2 );// put a 30 at index 2
  if( GEngine )
  {
    for( int index = 0; index < array.Num(); index++ )
    {
      GEngine->AddOnScreenDebugMessage( index, 30.f, FColor::Red,
      FString::FromInt( array[ index ] ) );
    }
  }
}
```

The output of the preceding code is shown in the following screenshot:



Finding whether an element is in the TArray

Searching out UE4 containers is easy. It is commonly done using the `Find` member function. Using the array we created previously, we can find the index of the value `10` by typing the following line of code:

```
int index = array.Find( 10 ); // would be index 3 in image above
```


www.it-ebooks.info

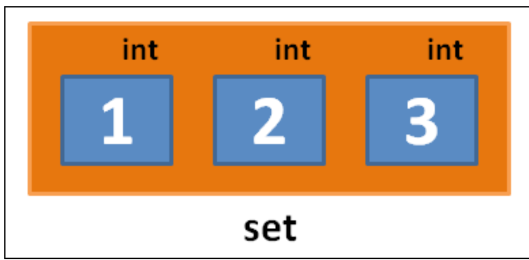
TSet<T>

A `TSet<int>` variable stores a set of integers. A `TSet<FString>` variable. stores a set of strings. The main difference between `TSet` and `TArray` is that `TSet` does not allow duplicates—all the elements inside a `TSet` are guaranteed to be unique. A `TArray` variable does not mind duplicates of the same elements.

To add numbers to `TSet`, simply call `Add`. Take an example of the following declaration:

```
TSet<int> set;
set.Add( 1 );
set.Add( 2 );
set.Add( 3 );
set.Add( 1 );// duplicate! won't be added
set.Add( 1 );// duplicate! won't be added
```

This is how `TSet` will look, as shown in the following figure:



Duplicate entries of the same value in the `TSet` will not be allowed. Notice how the entries in a `TSet` aren't numbered, as they were in a `TArray`: you can't use square brackets to access an entry in `TSet` arrays.

Iterating a TSet

In order to look into a `TSet` array, you must use an iterator. You can't use square brackets notation to access the elements of a `TSet`:

```
int count = 0;   // keep track of numerical index in set
for( TSet<int>::TIterator it = set.CreateIterator(); it; ++it )
{
  GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red,
  FString::FromInt( *it ) );
}
```


www.it-ebooks.info

Intersecting TSet

The `TSet`array has two special functions that the `TArray` variable does not. The intersection of two `TSet` arrays is basically the elements they have in common. If we have two `TSet` arrays such as `X` and `Y` and we intersect them, the result will be a third, new `TSet` array that contains only the elements common between them. Look at the following example:

```
TSet<int> X;
X.Add( 1 );
X.Add( 2 );
X.Add( 3 );
TSet<int> Y;
Y.Add( 2 );
Y.Add( 4 );
Y.Add( 8 );
TSet<int> common = X.Intersect(Y); // 2
```

The common elements between `X` and `Y` will then just be the element `2`.

Unioning TSet

Mathematically, the union of two sets is when you basically insert all the elements into the same set. Since we are talking about sets here, there won't be any duplicates.

If we take the `X` and `Y` sets from the previous example and create a union, we will get a new set, as follows:

```
TSet<int> uni = X.Union(Y); // 1, 2, 3, 4, 8
```

Finding TSet

You can determine whether an element is inside a `TSet` or not by using the `Find()` member function on the set. The `TSet` will return a pointer to the entry in the `TSet` that matches your query if the element exists in the `TSet`, or it will return `NULL` if the element you're asking for does not exist in the `TSet`.

TMap<T, S>

A `TMap<T, S>` creates a table of sorts in the RAM. A `TMap` represents a mapping of the keys at the left to the values on the right-hand side. You can visualize a `TMap` as a two-column table, with keys in the left column and values in the right column.

www.it-ebooks.info

A list of items for the player's inventory

For example, say we wanted to create a C++ data structure in order to store a list of items for the player's inventory. On the left-hand side of the table (the keys), we'd have an `FString` for the item's name. On the right-hand side (the values), we'd have an `int` for the quantity of that item.

Item (Key)	Quantity (Value)
apples	4
donuts	12
swords	1
shields	2

To do this in code, we'd simply use the following:

```
TMap<FString, int> items;
items.Add( "apples", 4 );
items.Add( "donuts", 12 );
items.Add( "swords", 1 );
items.Add( "shields", 2 );
```

Once you have created your `TMap`, you can access values inside the `TMap` using square brackets and by passing a key between the brackets. For example, in the `items` map in the preceding code, `items[ "apples" ]` is 4.

UE4 will crash if you use square brackets to access a key that doesn't exist in the map yet, so be careful! The C++ STL does not crash if you do this.

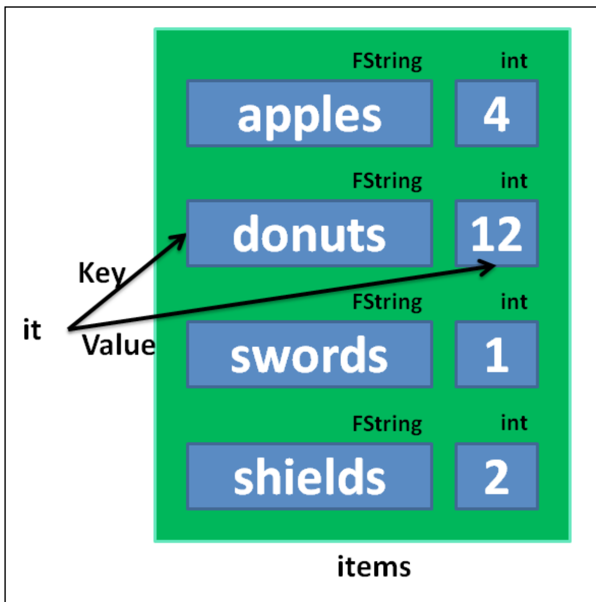
Iterating a TMap

In order to iterate a `TMap`, you use an iterator as well:

```
for( TMap<FString, int>::TIterator it = items.CreateIterator(); it;
++it )
{
  GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red,
  it->Key + FString(": ") + FString::FromInt( it->Value ) );
}
```


www.it-ebooks.info

`TMap` iterators are slightly different from `TArray` or `TSet` iterators. A `TMap` iterator contains both a `Key` and a `Value`. We can access the key inside with `it->Key` and the value inside the `TMap` with `it->Value`.



C++ STL versions of commonly used containers

I want to cover the C++ STL versions of a couple of containers. STL is the standard template library, which is shipped with most C++ compilers. The reason why I want to cover these STL versions is that they behave somewhat differently than the UE4 versions of the same containers. In some ways, their behavior is very good, but game programmers often complain of STL having performance issues. In particular, I want to cover STL's `set` and `map` containers.

If you like STL's interface but want better performance, there is a well-known reimplementation of the STL library by Electronic Arts called EASTL, which you can use. It provides the same functionality as STL but is implemented with better performance (basically by doing things such as eliminating bounds checking). It is available on GitHub at `https://github.com/paulhodge/EASTL`.

www.it-ebooks.info

C++ STL set

A C++ set is a bunch of items that are unique and sorted. The good feature about the STL `set` is that it keeps the set elements sorted. A quick and dirty way to sort a bunch of values is actually to just shove them into the same `set`. The `set` will take care of the sorting for you.

We can return to a simple C++ console application for the usage of sets. To use the C++ STL set you need to include `<set>`, as shown here:

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
  set<int> intSet;
  intSet.insert( 7 );
  intSet.insert( 7 );
  intSet.insert( 8 );
  intSet.insert( 1 );

  for( set<int>::iterator it = intSet.begin(); it != intSet.end();
  ++it )
  {
    cout << *it << endl;
  }
}
```

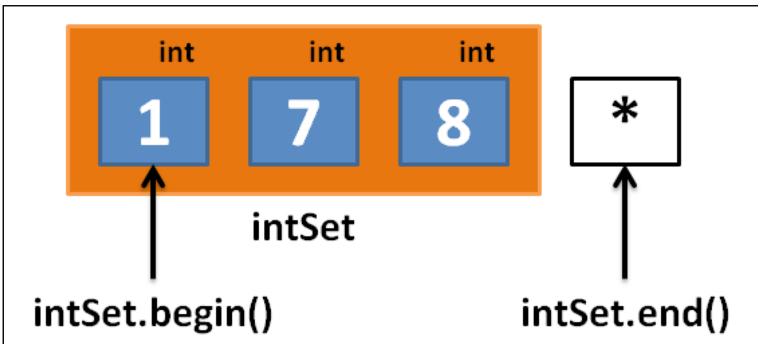
The following is the output of the preceding code:

```
1
7
8
```

The duplicate `7` is filtered out, and the elements are kept in increasing order inside the `set`. The way we iterate over the elements of an STL container is similar to UE4's `TSet` array. The `intSet.begin()` function returns an iterator that points to the head of the `intSet`.

www.it-ebooks.info

The condition to stop iterating is when `it` becomes `intSet.end()`. `intSet.end()` is actually one position past the end of the `set`, as shown in the following figure:



Finding an element in a <set>

To find an element inside an STL `set`, we can use the `find()` member function. If the item we're looking for turns up in the `set`, we get an iterator that points to the element we were searching for. If the item that we were looking for is not in the `set`, we get back `set.end()` instead, as shown here:

```
set<int>::iterator it = intSet.find( 7 );
if( it != intSet.end() )
{
  //  7  was inside intSet, and *it has its value
  cout << "Found " << *it << endl;
}
```

Exercise

Ask the user for a set of three unique names. Take each name in, one by one, and then print them in a sorted order. If the user repeats a name, then ask them for another one until you get to three.

Solution

The solution of the preceding exercise can be found using the following code:

```
#include <iostream>
#include <string>
#include <set>
using namespace std;
int main()
{
```


www.it-ebooks.info

```
    set<string> names;
    // so long as we don't have 3 names yet, keep looping,
    while( names.size() < 3 )
    {
      cout << names.size() << " names so far. Enter a name" << endl;
      string name;
      cin >> name;
      names.insert( name ); // won't insert if already there,
    }
    // now print the names. the set will have kept order
    for( set<string>::iterator it = names.begin(); it !=
    names.end(); ++it )
    {
      cout << *it << endl;
    }
  }
```

C++ STL map

The C++ STL `map` object is a lot like UE4's `TMap` object. The one thing it does that `TMap` does not is to maintain a sorted order inside the map as well. Sorting introduces an additional cost, but if you want your map to be sorted, opting for the STL version might be a good choice.

To use the C++ STL `map` object, we include `<map>`. In the following example program, we populate a map of items with some key-value pairs:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
  map<string, int> items;
  items.insert( make_pair( "apple", 12 ) );
  items.insert( make_pair( "orange", 1 ) );
  items.insert( make_pair( "banana", 3 ) );
  // can also use square brackets to insert into an STL map
  items[ "kiwis" ] = 44;

  for( map<string, int>::iterator it = items.begin(); it !=
  items.end(); ++it )
  {
    cout << "items[ " << it->first << " ] = " << it->second <<
    endl;
  }
}
```


www.it-ebooks.info

This is the output of the preceding program:

```
items[ apple ] = 12
items[ banana ] = 3
items[ kiwis ] = 44
items[ orange ] = 1
```

Notice how the iterator's syntax for an STL map is slightly different than that of `TMap`: we access the key using `it->first` and the value using `it->second`.

Notice how C++ STL also offers a bit of syntactic sugar over `TMap`; you can use square brackets to insert into the C++ STL `map`. You cannot use square brackets to insert into a `TMap`.

Finding an element in a <map>

You can search a map for a <`key`, `value`> pair using the STL map's `find` member function.

Exercise

Ask the user to enter five items and their quantities into an empty `map`. Print the results in sorted order.

Solution

The solution of the preceding exercise uses the following code:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
  map<string, int> items;
  cout << "Enter 5 items, and their quantities" << endl;
  while( items.size() < 5 )
  {
    cout << "Enter item" << endl;
    string item;
    cin >> item;
    cout << "Enter quantity" << endl;
    int qty;
    cin >> qty;
    items[ item ] = qty; // save in map, square brackets
```


www.it-ebooks.info

```
    // notation
  }

  for( map<string, int>::iterator it = items.begin(); it !=
  items.end(); ++it )
  {
    cout << "items[ " << it->first << " ] = " << it->second <<
    endl;
  }
}
```

In this solution code, we start by creating `map<string, int> items` to store all the items we're going to take in. Ask the user for an item and a quantity; then we save the `item` in the `items` map using square brackets notation.

Summary

UE4's containers and the C++ STL family of containers are both excellent for storing game data. Often, a programming problem can be simplified many times by selecting the right type of data container.

In the next chapter, we will actually get to programming the beginning of our game by keeping track of what the player is carrying and storing that information in a `TMap` object.

www.it-ebooks.info

www.it-ebooks.info

10
Inventory System and Pickup Items

We want our player to be able to pick up items from the game world. In this chapter, we will code and design a backpack for our player to store items. We will display what the player is carrying in the pack when the user presses the *I* key.

As a data representation, we can use the `TMap<FString, int>` items covered in the previous chapter to store our items. When the player picks up an item, we add it to the map. If the item is already in the map, we just increase its value by the quantity of the new items picked up.

Declaring the backpack

We can represent the player's backpack as a simple `TMap<FString, int>` item. To allow your player to gather items from the world, open the `Avatar.h` file and add the following `TMap` declaration:

```
class APickupItem; //  forward declare the APickupItem class,
                   // since it will be "mentioned" in a member
                   function decl below
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
  GENERATED_UCLASS_BODY()

  // A map for the player's backpack
  TMap<FString, int> Backpack;

  // The icons for the items in the backpack, lookup by string
  TMap<FString, UTexture2D*> Icons;
```

www.it-ebooks.info

```
    // A flag alerting us the UI is showing
    bool inventoryShowing;
    // member function for letting the avatar have an item
    void Pickup( APickupItem *item );
    // ... rest of Avatar.h same as before
};
```

Forward declaration

Before `AAvatar` class, notice that we have a `class APickupItem` forward declaration. Forward declarations are needed in a code file when a class is mentioned (such as the `APickupItem::Pickup( APickupItem *item );` function prototype), but there is no code in the file actually using an object of that type inside the file. Since the `Avatar.h` header file does not contain executable code that uses an object of the type `APickupItem`, a forward declaration is what we need.

The absence of a forward declaration will give a compiler error, since the compiler won't have heard of `class APickupItem` before compiling the code in `class AAvatar`. The compiler error will come at the declaration of the `APickupItem::Pickup( APickupItem *item );` function prototype declaration.

We declared two `TMap` objects inside the `AAvatar` class. This is how the objects will look, as shown in the following table:

`FString` **(name)**	`int` **(quantity)**	`UTexture2D*` **(im)**
GoldenEgg	2	
MetalDonut	1	
Cow	2	

In the `TMap` backpack, we store the `FString` variable of the item that the player is holding. In the `Icons` map, we store a single reference to the image of the item the player is holding.

www.it-ebooks.info

At render time, we can use the two maps working together to look up both the quantity of an item that the player has (in his `Backpack` mapping) and the texture asset reference of that item (in the `Icons` map). The following screenshot shows how the rendering of the HUD will look:

Note that we can also use an array of `struct` with an `FString` variable and `UTexture2D*` in it instead of using two maps.

For example, we can keep `TArray<Item> Backpack;` with a `struct` variable, as shown in the following code:

```
struct Item
{
  FString name;
  int qty;
  UTexture2D* tex;
};
```

Then, as we pick up items, they will be added to the linear array. However, counting the number of each item we have in the backpack will require constant reevaluation by iterating through the array of items each time we want to see the count. For example, to see how many hairbrushes you have, you will need to make a pass through the whole array. This is not as efficient as using a map.

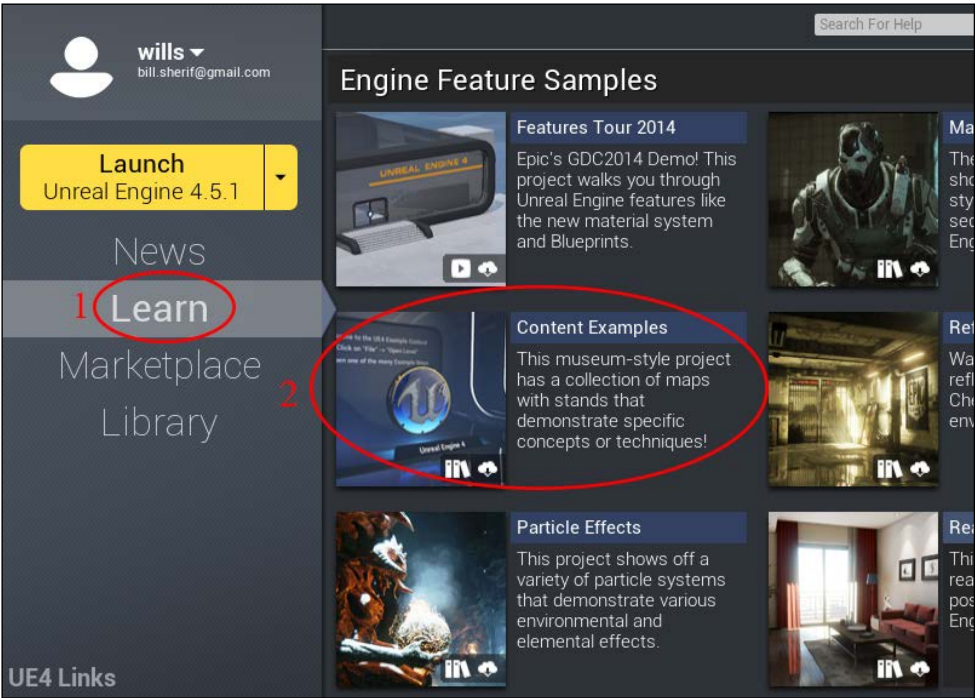
Importing assets

You might have noticed the **Cow** asset in the preceding screenshot, which is not a part of the standard set of assets that UE4 provides in a new project. In order to use the **Cow** asset, you need to import the cow from the **Content Examples** project. There is a standard importing procedure that UE4 uses.

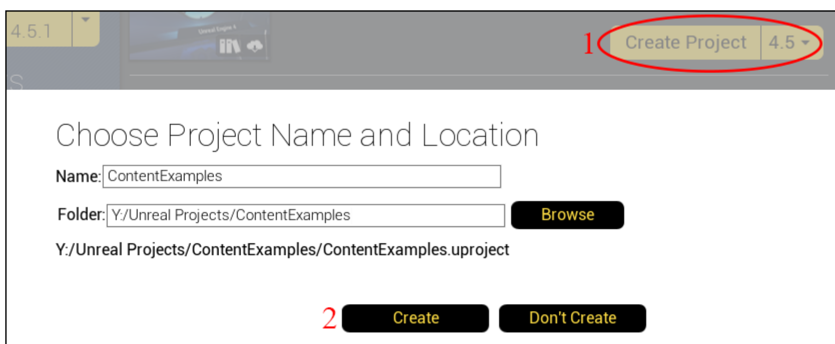
www.it-ebooks.info

In the following screenshot, I have outlined the procedure for importing the **Cow** asset. Other assets will be imported from other projects in UE4 using the same method. Perform the following steps to import the **Cow** asset:

1. Download and open UE4's **Content Examples** project:

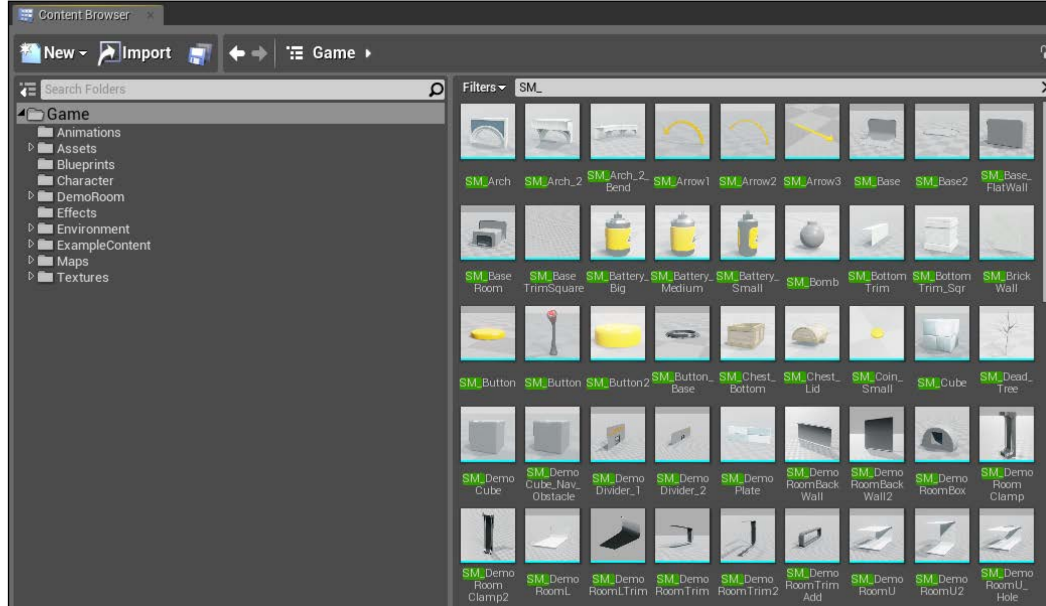


2. After you have downloaded **Content Examples**, open it and click on **Create Project**:



www.it-ebooks.info

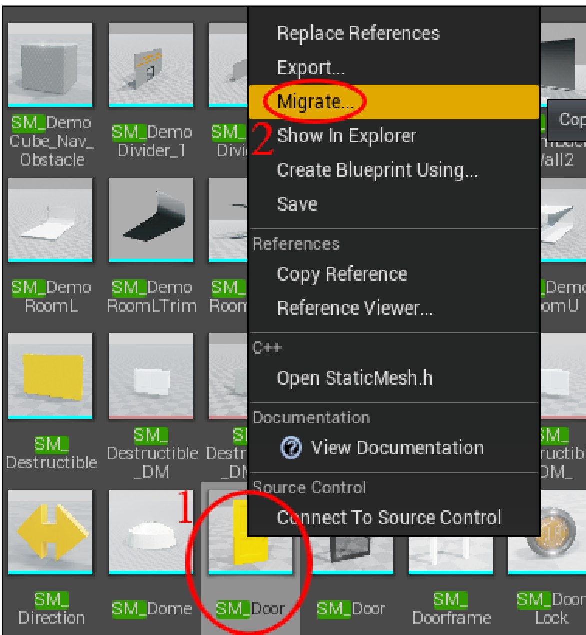
3. Next, name the folder in which you will put your `ContentExamples` and click on **Create**.
4. Open your `ContentExamples` project from the library. Browse the assets available in the project until you find one that you like. Searching for `SM_` will help since all static meshes usually begin with `SM_` by convention.



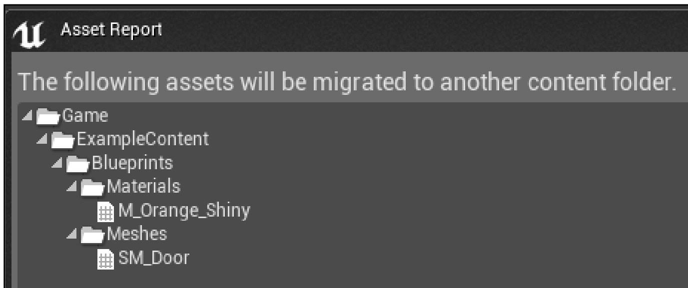
Lists of static meshes, all beginning with SM_

www.it-ebooks.info

5. When you find an asset that you like, import it into your project by right-clicking on the asset and then clicking on **Migrate...**:

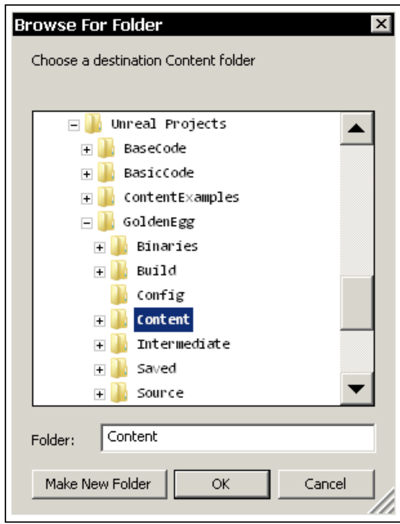


6. Click on **OK** in the **Asset Report** dialog:

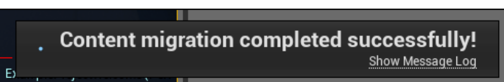


www.it-ebooks.info

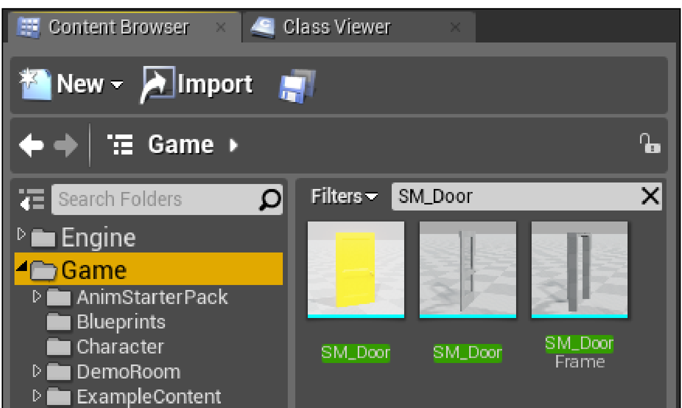
7. Select the **Content** folder from your project that you want to add the **SM_Door** file to. For me, I want to add it to `Y:/Unreal Projects/GoldenEgg/Content`, as shown in the following screenshot:



8. If the import was completed successfully, you will see a message as follows:



9. Once you import your asset, you will see it show up in your asset browser inside your project:

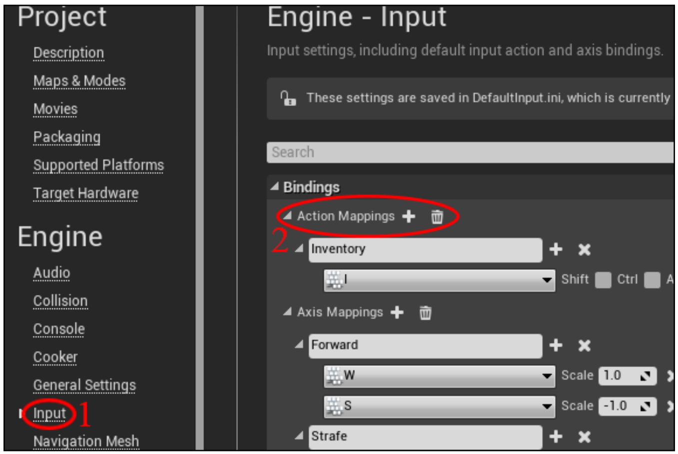


You can then use the asset inside your project normally.

www.it-ebooks.info

Attaching an action mapping to a key

We need to attach a key to activate the display of the player's inventory. Inside the UE4 editor, add an **Action Mappings +** called `Inventory` and assign it to the keyboard key *I*:



In the `Avatar.h` file, add a member function to be run when the player's inventory needs to be displayed:

```
void ToggleInventory();
```

In the `Avatar.cpp` file, implement the `ToggleInventory()` function, as shown in the following code:

```
void AAvatar::ToggleInventory()
{
  if( GEngine )
  {
    GEngine->AddOnScreenDebugMessage( 0, 5.f, FColor::Red,
    "Showing inventory..." );
  }
}
```

Then, connect the `"Inventory"` action to `AAvatar::ToggleInventory()` in `SetupPlayerInputComponent()`:

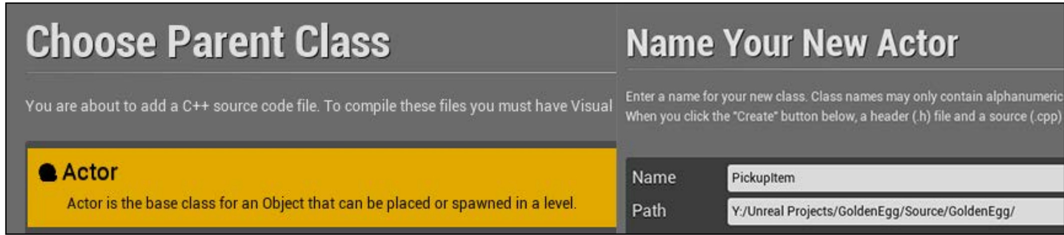
```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
  InputComponent)
{
  InputComponent->BindAction( "Inventory", IE_Pressed, this,
  &AAvatar::ToggleInventory );
  // rest of SetupPlayerInputComponent same as before
}
```


www.it-ebooks.info

Base class PickupItem

We need to define how a pickup item looks in code. Each pickup item will derive from a common base class. Let's construct the base class for a `PickupItem` class now.

The `PickupItem` base class should inherit from the `AActor` class. Similar to how we created multiple NPC blueprints from the base NPC class, we can create multiple `PickupItem` blueprints from a single `PickupItem` base class, as shown in the following screenshot:



Once you have created the `PickupItem` class, open its code in Visual Studio.

The `APickupItem` class will need quite a few members, as follows:

- An `FString` variable for the name of the item being picked up
- An `int32` variable for the quantity of the item being picked up
- A `USphereComponent` variable for the sphere that you will collide with for the item to be picked up
- A `UStaticMeshComponent` variable to hold the actual `Mesh`
- A `UTexture2D` variable for the icon that represents the item
- A pointer for the HUD (which we will initialize later)

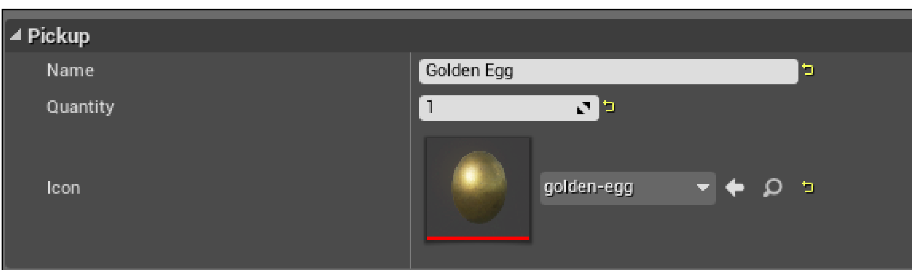
This is how the code in `PickupItem.h` looks:

```
UCLASS()
class GOLDENEGG_API APickupItem : public AActor
{
  GENERATED_UCLASS_BODY()

  // The name of the item you are getting
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
  FString Name;

  // How much you are getting
```


www.it-ebooks.info

```
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
    int32 Quantity;

    // the sphere you collide with to pick item up
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    Item)
    TSubobjectPtr<USphereComponent> ProxSphere;

    // The mesh of the item
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    Item)
    TSubobjectPtr<UStaticMeshComponent> Mesh;

    // The icon that represents the object in UI/canvas
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
    UTexture2D* Icon;

    // When something comes inside ProxSphere, this function runs
    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
    SweepResult );
  };
```

The point of all these `UPROPERTY()` declarations is to make `APickupItem` completely configurable by blueprints. For example, the items in the **Pickup** category will be displayed as follows in the blueprints editor:



www.it-ebooks.info

In the `PickupItem.cpp` file, we complete the constructor for the `APickupItem` class, as shown in the following code:

```
APickupItem::APickupItem(const class
FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
  Name = "UNKNOWN ITEM";
  Quantity = 0;

  // initialize the unreal objects
  ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this,
  TEXT("ProxSphere"));
  Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
  TEXT("Mesh"));

  // make the root object the Mesh
  RootComponent = Mesh;
  Mesh->SetSimulatePhysics(true);

  // Code to make APickupItem::Prox() run when this
  // object's proximity sphere overlaps another actor.
  ProxSphere->OnComponentBeginOverlap.AddDynamic(this,
  &APickupItem::Prox);
  ProxSphere->AttachTo( Mesh ); // very important!
}
```

In the first two lines, we perform an initialization of `Name` and `Quantity` to values that should stand out to the game designer as being uninitialized. I used block capitals so that the designer can clearly see that the variable has never been initialized before.

We then initialize the `ProxSphere` and `Mesh` components using `PCIP.CreateDefaultSubobject`. The freshly initialized objects might have some of their default values initialized, but `Mesh` will start out empty. You will have to load the actual mesh later, inside blueprints.

For the mesh, we set it to simulate realistic physics so that pickup items will bounce and roll around if they are dropped or moved. Pay special attention to the line `ProxSphere->AttachTo( Mesh )`. This line tells you to make sure the pickup item's `ProxSphere` component is attached to the `Mesh` root component. This means that when the mesh moves in the level, `ProxSphere` follows. If you forget this step (or if you did it the other way around), then `ProxSphere` will not follow the mesh when it bounces.

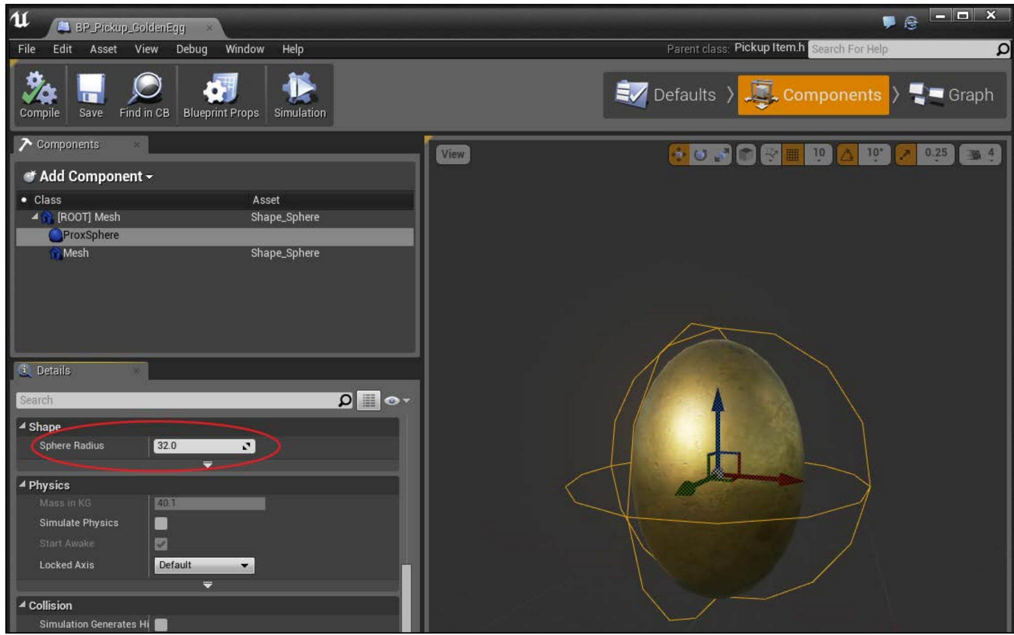
www.it-ebooks.info

The root component

In the preceding code, we assigned `RootComponent` of `APickupItem` to the `Mesh` object. The `RootComponent` member is a part of the `AActor` base class, so every `AActor` and its derivatives has a root component. The root component is basically meant to be the core of the object, and also defines how you collide with the object. The `RootComponent` object is defined in the `Actor.h` file, as shown in the following code:

```
/**
 * Collision primitive that defines the transform (location,
  rotation, scale) of this Actor.
 */
UPROPERTY()
class USceneComponent* RootComponent;
```

So the UE4 creators intended `RootComponent` to always be a reference to the collision primitive. Sometimes the collision primitive can be capsule shaped, other times it can be spherical or even box shaped, or it can be arbitrarily shaped, as in our case, with the mesh. It's rare that a character should have a box-shaped root component, however, because the corners of the box can get caught on walls. Round shapes are usually preferred. The `RootComponent` property shows up in the blueprints, where you can see and manipulate it.



You can edit the ProxSphere root component from its blueprints once you create a blueprint based on the PickupItem class

www.it-ebooks.info

Finally, the `Prox_Implementation` function gets implemented, as follows:

```
void APickupItem::Prox_Implementation( AActor* OtherActor,
  UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
  bFromSweep, const FHitResult & SweepResult )
{
  // if the overlapped actor is NOT the player,
  // you simply should return
  if( Cast<AAvatar>( OtherActor ) == nullptr )
  {
    return;
  }

  // Get a reference to the player avatar, to give him
  // the item
  AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn( GetWorld(), 0 ) );

  // Let the player pick up item
  // Notice use of keyword this!
  // That is how _this_ Pickup can refer to itself.
  avatar->Pickup( this );

  // Get a reference to the controller
  APlayerController* PController = GetWorld()-
  >GetFirstPlayerController();

  // Get a reference to the HUD from the controller
  AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
  hud->addMessage( Message( Icon, FString("Picked up ") +
  FString::FromInt(Quantity) + FString(" ") + Name,
  5.f, FColor::White, FColor::Black ) );

  Destroy();
}
```

A couple of tips here that are pretty important: first, we have to access a couple of *globals* to get the objects we need. There are three main objects we'll be accessing through these functions that manipulate the HUD: the controller (`APlayerController`), the HUD (`AMyHUD`), and the player himself (`AAvatar`). There is only one of each of these three types of objects in the game instance. UE4 has made finding them easy.

www.it-ebooks.info

Getting the avatar

The `player` class object can be found at any time from any place in the code by simply calling the following code:

```
AAvatar *avatar = Cast<AAvatar>(

  UGameplayStatics::GetPlayerPawn( GetWorld(), 0 ) );
```

We then pass him the item by calling the `AAvatar::Pickup()` function defined earlier.

Because the `PlayerPawn` object is really an `AAvatar` instance, we cast the result to the `AAvatar` class, using the `Cast<AAvatar>` command. The `UGameplayStatics` family of functions are accessible anywhere in your code—they are global functions.

Getting the player controller

Retrieving the player controller is from a *superglobal* as well:

```
APlayerController* PController =
  GetWorld()->GetFirstPlayerController();
```

The `GetWorld()` function is actually defined in the `UObject` base class. Since all UE4 objects derive from `UObject`, any object in the game actually has access to the `world` object.

Getting the HUD

Although this organization might seem strange at first, the HUD is actually attached to the player's controller. You can retrieve the HUD as follows:

```
AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
```

We cast the HUD object since we previously set the HUD to being an `AMyHUD` instance in blueprints. Since we will be using the HUD often, we can actually store a permanent pointer to the HUD inside our `APickupItem` class. We will discuss this point later.

Next, we implement `AAvatar::Pickup`, which adds an object of the type `APickupItem` to Avatar's backpack:

```
void AAvatar::Pickup( APickupItem *item )
{
  if( Backpack.Find( item->Name ) )
  {
    // the item was already in the pack.. increase qty of it
    Backpack[ item->Name ] += item->Quantity;
```


www.it-ebooks.info

```
      }
      else
      {
        // the item wasn't in the pack before, add it in now
        Backpack.Add(item->Name, item->Quantity);
        // record ref to the tex the first time it is picked up
        Icons.Add(item->Name, item->Icon);
      }
    }
```

In the preceding code, we check whether the pickup item that the player just got is already in his pack. If it is, we increase its quantity. If it is not in his pack, we add it to both his pack and the `Icons` mapping.

To add the pickup items to the pack, use the following line of code:

```
avatar->Pickup( this );
```

The `APickupItem::Prox_Implementation` is the way this member function will get called.

Now, we need to display the contents of our backpack in the HUD when the player presses *I*.

Drawing the player inventory

An inventory screen in a game such as *Diablo* features a pop-up window, with the icons of the items you've picked up in the past arranged in a grid. We can achieve this type of behavior in UE4.

There are a number of approaches to drawing a UI in UE4. The most basic way is to simply use the `HUD::DrawTexture()` calls. Another way is to use Slate. Another way still is to use the newest UE4 UI functionality: **Unreal Motion Graphics** (**UMG**) Designer.

Slate uses a declarative syntax to lay out UI elements in C++. Slate is best suited for menus and the like. UMG is new in UE 4.5 and uses a heavily blueprint-based workflow. Since our focus here is on exercises that use C++ code, we will stick to a `HUD::DrawTexture()` implementation. This means that we will have to manage all the data that deals with the inventory in our code.

www.it-ebooks.info

Using HUD::DrawTexture()

We will achieve this in two steps. The first step is to push the contents of our inventory to the HUD when the user presses the *I* key. The second step is to actually render the icons into the HUD in a grid-like fashion.

To keep all the information about how a widget can be rendered, we declare a simple structure to keep the information concerning what icon it uses, its current position, and current size.

This is how the `Icon` and `Widget` structures look:

```
struct Icon
{
  FString name;
  UTexture2D* tex;
  Icon(){ name = "UNKNOWN ICON"; tex = 0; }
  Icon( FString& iName, UTexture2D* iTex )
  {
    name = iName;
    tex = iTex;
  }
};

struct Widget
{
  Icon icon;
  FVector2D pos, size;
  Widget(Icon iicon)
  {
    icon = iicon;
  }
  float left(){ return pos.X; }
  float right(){ return pos.X + size.X; }
  float top(){ return pos.Y; }
  float bottom(){ return pos.Y + size.Y; }
};
```

You can add these structure declarations to the top of `MyHUD.h`, or you can add them to a separate file and include that file everywhere those structures are used.

Notice the four member functions on the `Widget` structure to get to the `left()`, `right()`, `top()`, and `bottom()` functions of the widget. We will use these later to determine whether a click point is inside the box.

www.it-ebooks.info

Next, we declare the function that will render the widgets out on the screen in the `AMyHUD` class:

```
void AMyHUD::DrawWidgets()
{
  for( int c = 0; c < widgets.Num(); c++ )
  {
    DrawTexture( widgets[c].icon.tex, widgets[c].pos.X,
    widgets[c].pos.Y, widgets[c].size.X, widgets[c].size.Y, 0, 0,
    1, 1 );
    DrawText( widgets[c].icon.name, FLinearColor::Yellow,
    widgets[c].pos.X, widgets[c].pos.Y, hudFont, .6f, false );
  }
}
```

A call to the `DrawWidgets()` function should be added to the `DrawHUD()` function:

```
void AMyHUD::DrawHUD()
{
  Super::DrawHUD();
  // dims only exist here in stock variable Canvas
  // Update them so use in addWidget()
  dims.X = Canvas->SizeX;
  dims.Y = Canvas->SizeY;
  DrawMessages();
  DrawWidgets();
}
```

Next, we will fill the `ToggleInventory()` function. This is the function that runs when the user presses *I*:

```
void AAvatar::ToggleInventory()
{
  // Get the controller & hud
  APlayerController* PController = GetWorld()-
  >GetFirstPlayerController();
  AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );

  // If inventory is displayed, undisplay it.
  if( inventoryShowing )
  {
    hud->clearWidgets();
    inventoryShowing = false;
    PController->bShowMouseCursor = false;
    return;
  }
```


www.it-ebooks.info

```
    // Otherwise, display the player's inventory
    inventoryShowing = true;
    PController->bShowMouseCursor = true;
    for( TMap<FString,int>::TIterator it =
    Backpack.CreateIterator(); it; ++it )
    {
      // Combine string name of the item, with qty eg Cow x 5
      FString fs = it->Key + FString::Printf( TEXT(" x %d"), it-
      >Value );
      UTexture2D* tex;
      if( Icons.Find( it->Key ) )
        tex = Icons[it->Key];
      hud->addWidget( Widget( Icon( fs, tex ) ) );
    }
  }
```

For the preceding code to compile, we need to add a function to `AMyHUD`:

```
void AMyHUD::addWidget( Widget widget )
{
  // find the pos of the widget based on the grid.
  // draw the icons..
  FVector2D start( 200, 200 ), pad( 12, 12 );
  widget.size = FVector2D( 100, 100 );
  widget.pos = start;
  // compute the position here
  for( int c = 0; c < widgets.Num(); c++ )
  {
    // Move the position to the right a bit.
    widget.pos.X += widget.size.X + pad.X;
    // If there is no more room to the right then
    // jump to the next line
    if( widget.pos.X + widget.size.X > dims.X )
    {
      widget.pos.X = start.X;
      widget.pos.Y += widget.size.Y + pad.Y;
    }
  }
  widgets.Add( widget );
}
```


www.it-ebooks.info

We keep using the `Boolean` variable in `inventoryShowing` to tell us whether the inventory is currently displayed or not. When the inventory is shown, we also show the mouse so that the user knows what he's clicking on. Also, when the inventory is displayed, free motion of the player is disabled. The easiest way to disable a player's free motion is by simply returning from the movement functions before actually moving. The following code is an example:

```
void AAvatar::Yaw( float amount )
{
  if( inventoryShowing )
  {
    return; // when my inventory is showing,
    // player can't move
  }
  AddControllerYawInput(200.f*amount * GetWorld()-
  >GetDeltaSeconds());
}
```

Exercise

Check out each of the movement functions with the `if( inventoryShowing ) { return; }` short circuit return.

Detecting inventory item clicks

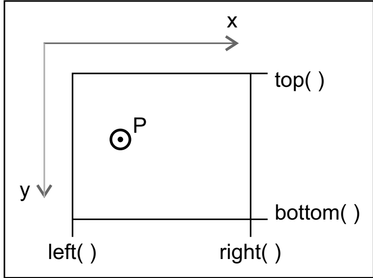
We can detect whether someone is clicking on one of our inventory items by doing a simple hit point-in-box hit. A point-in-box test is done by checking the point of the click against the contents of the box.

Add the following member function to `struct Widget`:

```
struct Widget
{
  // .. rest of struct same as before ..
  bool hit( FVector2D p )
  {
    // +---+ top (0)
    // |   |
    // +---+ bottom (2) (bottom > top)
    // L   R
    return p.X > left() && p.X < right() && p.Y > top() && p.Y <
    bottom();
  }
};
```


www.it-ebooks.info

The point-in-box test is as follows:



So, it is a hit if `p.X` is all of:

- Right of `left() (p.X > left())`
- Left of `right() (p.X < right())`
- Below the `top() (p.Y > top())`
- Above the `bottom() (p.Y < bottom())`

Remember that in UE4 (and UI rendering in general) the *y* axis is inverted. In other words, y goes down in UE4. This means that `top()` is less than `bottom()` since the origin (the `(0, 0)` point) is at the top-left corner of the screen.

Dragging elements

We can drag elements easily. The first step to enable dragging is to respond to the left mouse button click. First, we'll write the function to execute when the left mouse button is clicked. In the `Avatar.h` file, add the following prototype to the class declaration:

```
void MouseClicked();
```

In the `Avatar.cpp` file, we can attach a function to execute on a mouse click and pass the click request to the HUD, as follows:

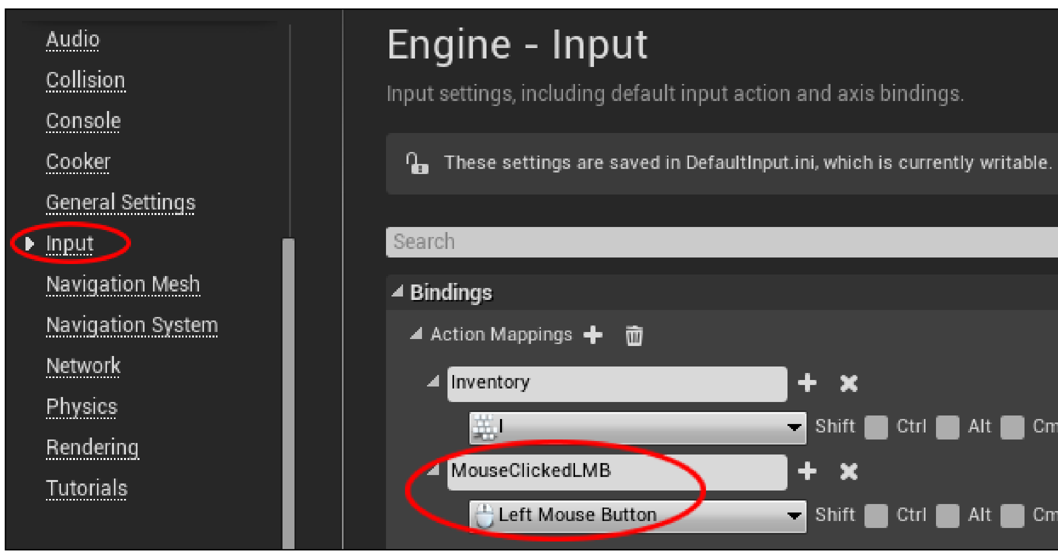
```
void AAvatar::MouseClicked()
{
  APlayerController* PController = GetWorld()-
  >GetFirstPlayerController();
  AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
  hud->MouseClicked();
}
```


www.it-ebooks.info

Then in `AAvatar::SetupPlayerInputComponent`, we have to attach our responder:

```
InputComponent->BindAction( "MouseClickedLMB", IE_Pressed, this,
  &AAvatar::MouseClicked );
```

The following screenshot shows how you can attach a render:



Add a member to the `AMyHUD` class:

```
Widget* heldWidget;  // hold the last touched Widget in memory
```

Next, in `AMyHUD::MouseClicked()`, we start searching for the `Widget` hit:

```
void AMyHUD::MouseClicked()
{
  FVector2D mouse;
  PController->GetMousePosition( mouse.X, mouse.Y );
  heldWidget = NULL; // clear handle on last held widget
  // go and see if mouse xy click pos hits any widgets
  for( int c = 0; c < widgets.Num(); c++ )
  {
    if( widgets[c].hit( mouse ) )
    {
      heldWidget = &widgets[c];// save widget
      return;                  // stop checking
    }
  }
}
```


www.it-ebooks.info

In the `AMyHUD::MouseClicked` function, we loop through all the widgets that are on the screen and check for a hit with the current mouse position. You can get the current mouse position from the controller at any time by simply looking up `PController->GetMousePosition()`.

Each widget is checked against the current mouse position, and the widget that got hit by the mouse click will be moved once a mouse is dragged. Once we have determined which widget got hit, we can stop checking, so we have a `return` value from the `MouseClicked()` function.

Hitting widget is not enough, though. We need to drag the widget that got hit when the mouse moves. For this, we need to implement a `MouseMoved()` function in `AMyHUD`:

```
void AMyHUD::MouseMoved()
{
  static FVector2D lastMouse;
  FVector2D thisMouse, dMouse;
  PController->GetMousePosition( thisMouse.X, thisMouse.Y );
  dMouse = thisMouse - lastMouse;
  // See if the left mouse has been held down for
  // more than 0 seconds. if it has been held down,
  // then the drag can commence.
  float time = PController->GetInputKeyTimeDown(
  EKeys::LeftMouseButton );
  if( time > 0.f && heldWidget )
  {
    // the mouse is being held down.
    // move the widget by displacement amt
    heldWidget->pos.X += dMouse.X;
    heldWidget->pos.Y += dMouse.Y; // y inverted
  }
  lastMouse = thisMouse;
}
```

Don't forget to include a declaration in the `MyHUD.h` file.

The drag function looks at the difference in the mouse position between the last frame and this frame and moves the selected widget by that amount. A `static` variable (global with local scope) is used to remember the `lastMouse` position between the calls for the `MouseMoved()` function.

www.it-ebooks.info

How can we link the mouse's motion to running the `MouseMoved()` function in `AMyHUD`? If you remember, we have already connected the mouse motion in the `Avatar` class. The two functions that we used were `AAvatar::Pitch()` (the y axis) and `AAvatar::Yaw()` (the x axis). Extending these functions will enable you to pass mouse inputs to the HUD. I will show you the `Yaw` function now, and you can extrapolate how `Pitch` will work from there:

```
void AAvatar::Yaw( float amount )
{
  //x axis
  if( inventoryShowing )
  {
    // When the inventory is showing,
    // pass the input to the HUD
    APlayerController* PController = GetWorld()-
    >GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->MouseMoved();
    return;
  }
  else
  {
    AddControllerYawInput(200.f*amount * GetWorld()-
    >GetDeltaSeconds());
  }
}
```

The `AAvatar::Yaw()` function first checks whether the inventory is showing or not. If it is showing, inputs are routed straight to the HUD, without affecting `Avatar`. If the HUD is not showing, inputs just go to `Avatar`.

Exercises

1. Complete the `AAvatar::Pitch()` function (y axis) to route inputs to the HUD instead of to `Avatar`.
2. Make the NPC characters from *Chapter 8*, *Actors and Pawns*, give the player an item (such as `GoldenEgg`) when he goes near them.

www.it-ebooks.info

Summary

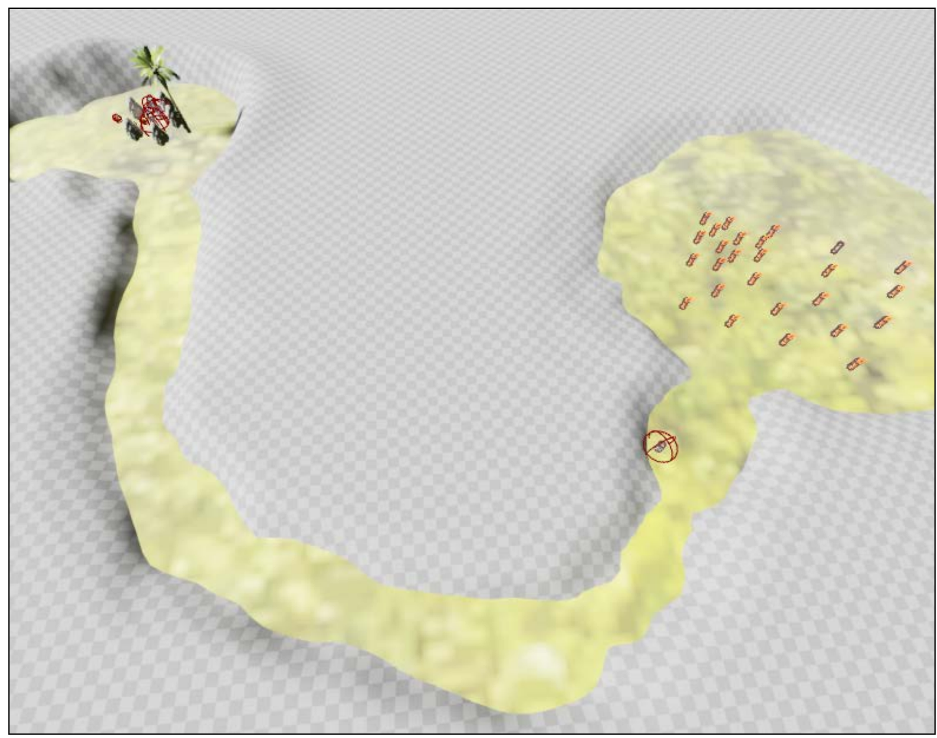
In this chapter, we covered how to set up multiple pickup items for the player to see displayed in the level and also pick up. In the next chapter, we will introduce *Monsters* and the player will be able to defend himself against the monsters using magic spells.

www.it-ebooks.info

11
Monsters

We'll add in a bunch of opponents for the player.

What I've done in this chapter is added a landscape to the example. The player will walk along the path sculpted out for him and then he will encounter an army. There is an NPC before he reaches the army that will offer advice.



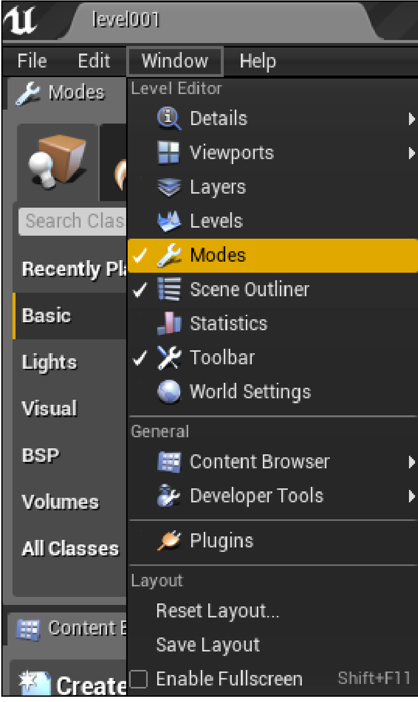
The scene: starting to look like a game

www.it-ebooks.info

Landscape

We haven't covered how to sculpt the landscape in this book yet, but we'll do that here. First, you must have a landscape to work with. Start a new file by navigating to **File** | **New**. You can choose an empty level or a level with a sky. I chose the one without the sky in this example.

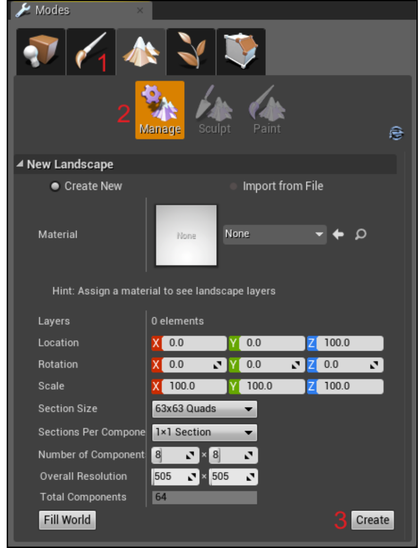
To create a landscape, we have to work from the **Modes** panel. Make sure that the **Modes** panel is displayed by navigating to **Window** | **Modes**:



Displaying the modes panel

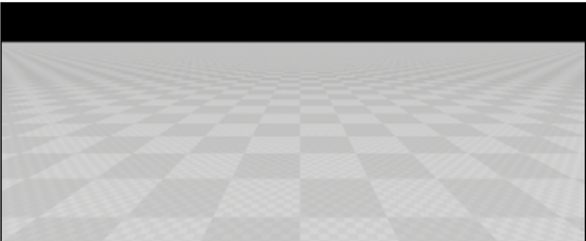
www.it-ebooks.info

A landscape can be created in three steps, which are shown in the following screenshot, followed by the corresponding steps:



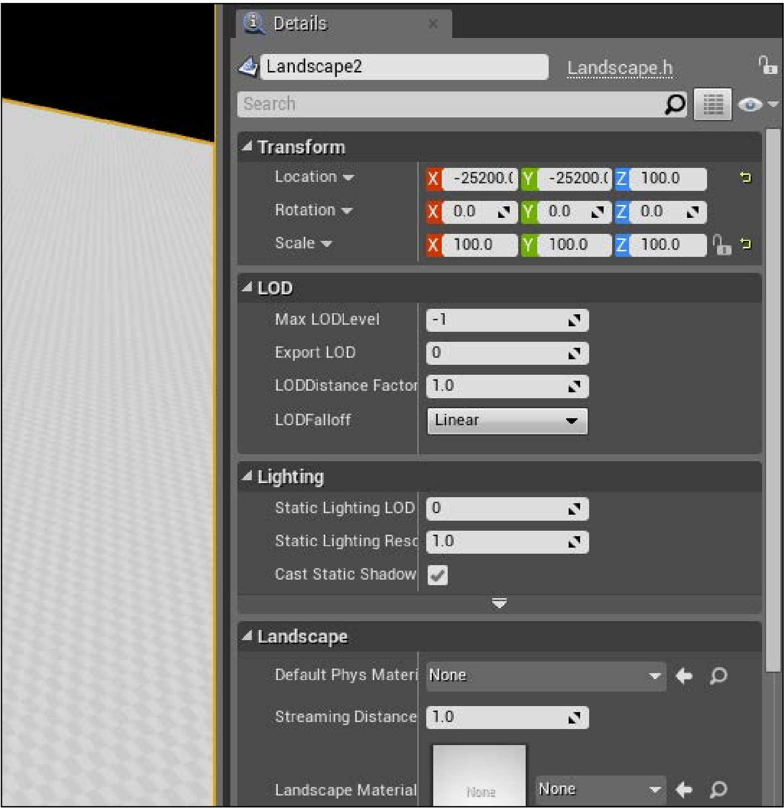
1. Click on the landscape icon (the picture of the mountains) in the **Modes** panel.
2. Click on the **Manage** button.
3. Next, click on the **Create** button in the lower right-hand corner of the screen.

You should now have a landscape to work with. It will appear as a gray, tiled area in the main window:



www.it-ebooks.info

The first thing you will want to do with your landscape scene is add some color to it. What's a landscape without colors? Right-click anywhere on your gray, tiled landscape object. In the **Details** panel at the right, you will see that it is populated with information, as shown in the following screenshot:



Scroll down until you see the **Landscape Material** property. You can select the **M_Ground_Grass** material for a realistic-looking ground.

Next, add a light to the scene. You should probably use a directional light so that all of the ground has some light on it.

www.it-ebooks.info

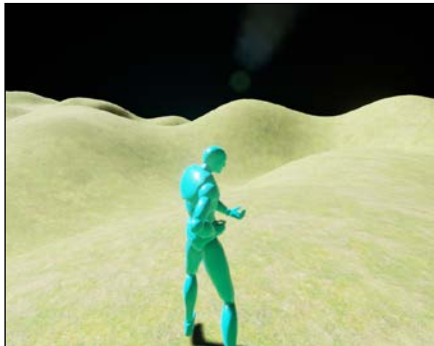
Sculpting the landscape

A flat landscape can be boring. We will at least add some curves and hills to the place. To do so, click on the **Sculpt** button in the **Modes** panel:

To change the landscape, click on the Sculpt button

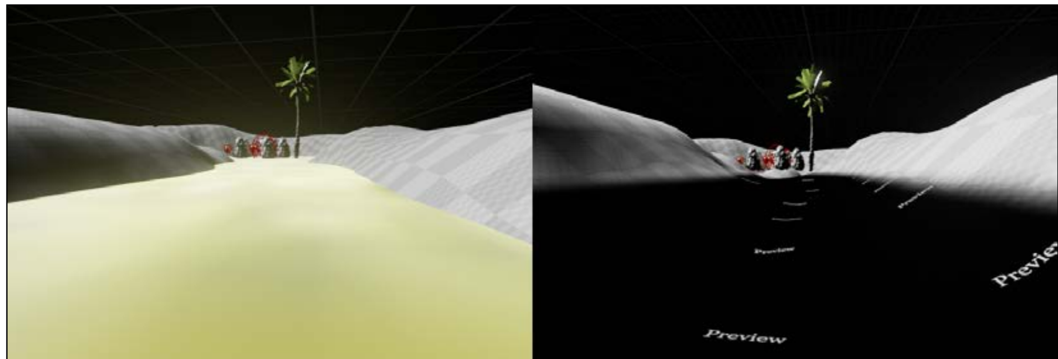
The strength and size of your brush are determined by the **Brush Size** and **Tool Strength** parameters in the **Modes** window.

Click on your landscape and drag the mouse to change the height of the turf. Once you're happy with what you've got, click on the **Play** button to try it out. The resultant output can be seen in the following screenshot:



www.it-ebooks.info

Play around with your landscape and create a scene. What I did was lower the landscape around a flat ground plane, so the player has a well-defined flat area to walk on, as shown in the following screenshot:



Feel free to do whatever you like with your landscape. You can use what I'm doing here as inspiration, if you like. I will recommend that you import assets from **ContentExamples** or from **StrategyGame** in order to use them inside your game. To do this, refer to the *Importing assets* section in *Chapter 10*, *Inventory System and Pickup Items*. When you're done importing assets, we can proceed to bring monsters into your world.

Monsters

We'll start programming monsters in the same way we programmed NPCs and `PickupItem`. First, we'll write a base class (by deriving from character) to represent the `Monster` class. Then, we'll derive a bunch of blueprints for each monster type. Every monster will have a couple of properties in common that determine its behavior. These are the common properties:

- A `float` variable for speed.
- A `float` variable for the `HitPoints` value (I usually use floats for HP, so we can easily model HP leeching effects such as walking through a pool of lava).
- An `int32` variable for the experience gained in defeating the monster.
- A `UClass` function for the loot dropped by the monster.
- A `float` variable for `BaseAttackDamage` done on each attack.
- A `float` variable for `AttackTimeout`, which is the amount of time for which the monster rests between attacking.

www.it-ebooks.info

- Two `USphereComponents` object: One of them is `SightSphere`—how far he can see. The other is `AttackRangeSphere`, which is how far his attack reaches. The `AttackRangeSphere` object is always smaller than `SightSphere`.

Derive from the `Character` class to create your class for `Monster`. You can do this in UE4 by going to **File | Add Code To Project...** and then selecting the **Character** option from the menu for your base class.

Fill out the `Monster` class with the base properties. Make sure that you declare `UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)` so that the properties of the monsters can be changed in the blueprints:

```
UCLASS()
class GOLDENEGG_API AMonster : public ACharacter
{
  GENERATED_UCLASS_BODY()

  // How fast he is
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  float Speed;

  // The hitpoints the monster has
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  float HitPoints;

  // Experience gained for defeating
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  int32 Experience;

  // Blueprint of the type of item dropped by the monster
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  UClass* BPLoot;

  // The amount of damage attacks do
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  float BaseAttackDamage;

  // Amount of time the monster needs to rest in seconds
  // between attacking
```


www.it-ebooks.info

```
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
  float AttackTimeout;

  // Time since monster's last strike, readable in blueprints
  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  MonsterProperties)
  float TimeSinceLastStrike;

  // Range for his sight
  UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
  Collision)
  USphereComponent* SightSphere;

  // Range for his attack. Visualizes as a sphere in editor,
  UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
  Collision)
  USphereComponent* AttackRangeSphere;
};
```

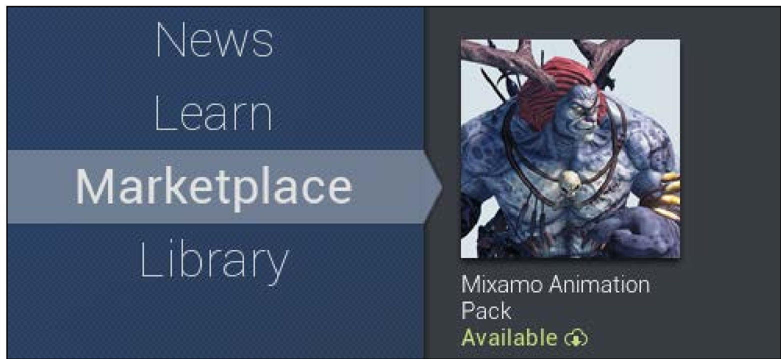
You will need some bare minimum code in your `Monster` constructor to get the monster's properties initialized. Use the following code in the `Monster.cpp` file:

```
AMonster::AMonster(const class FObjectInitializer& PCIP) : Super(PCIP)
{
  Speed = 20;
  HitPoints = 20;
  Experience = 0;
  BPLoot = NULL;
  BaseAttackDamage = 1;
  AttackTimeout = 1.5f;
  TimeSinceLastStrike = 0;

  SightSphere = PCIP.CreateDefaultSubobject<USphereComponent>
  (this, TEXT("SightSphere"));
  SightSphere->AttachTo( RootComponent );

  AttackRangeSphere = PCIP.CreateDefaultSubobject
  <USphereComponent>(this, TEXT("AttackRangeSphere"));
  AttackRangeSphere->AttachTo( RootComponent );
}
```


www.it-ebooks.info

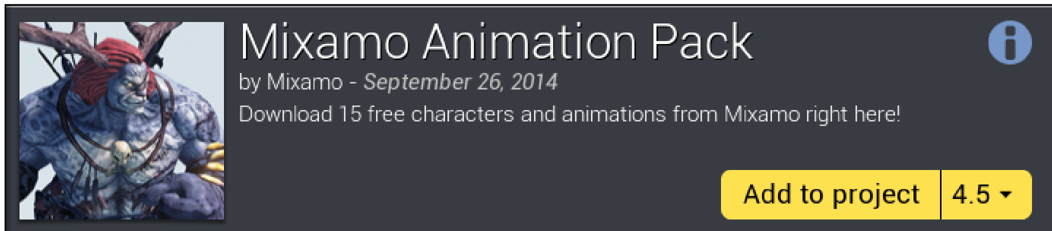
Compile and run the code. Open Unreal Editor and derive a blueprint based on your `Monster` class (call it `BP_Monster`). Now we can start configuring your monster's `Monster` properties.

For the skeletal mesh, we won't use the `HeroTPP` model for the monster because we need the monster to be able to do melee attacks and the `HeroTPP` model does not come with a melee attack. However, some of the models in the **Mixamo Animation Pack** file have melee attack animations. So download the **Mixamo Animation Pack** file from the UE4 marketplace (free).



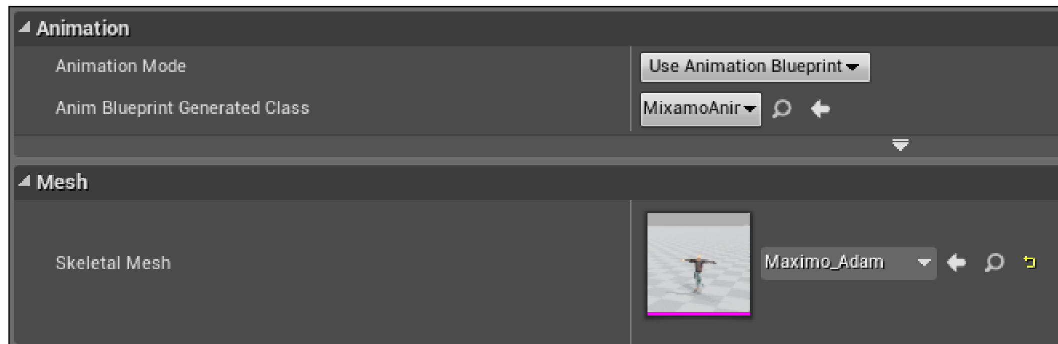
Inside the pack are some pretty gross models that I'd avoid, but others are quite good

Next, you should add the **Mixamo Animation Pack** file to your project, as shown in the following screenshot:



www.it-ebooks.info

Now, create a blueprint called `BP_Monster` based on your `Monster` class. Edit the blueprint's class properties and select **Mixamo_Adam** (it is actually typed as **Maximo_Adam** in the current issue of the package) as the skeletal mesh. Also, select **MixamoAnimBP_Adam** as the animation blueprint.



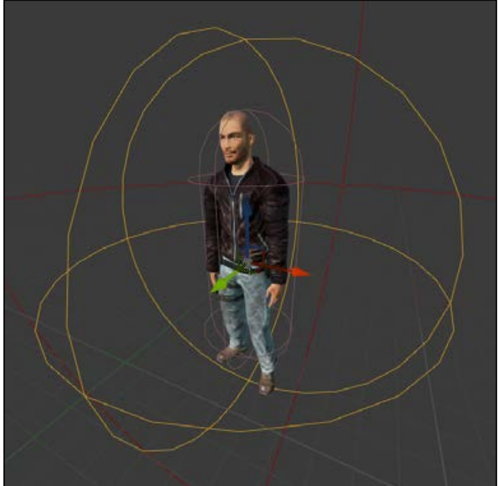
Select the Maximo_Adam Skeletal Mesh and MixamoAnimBP_Adam for Anim Blueprint Generated Class

We will modify the animation blueprint to correctly incorporate the melee attack animation later.

While you're editing your `BP_Monster` blueprint, change the sizes of the `SightSphere` and `AttackRangeSphere` objects to values that make sense to you. I made my monster's `AttackRangeSphere` object just big enough to be about an arm's reach (60 units) and his `SightSphere` object to be 25 times bigger than that (about 1,500 units).

Remember that the monster will start moving towards the player once the player enters the monster's `SightSphere`, and the monster will start attacking the player once the player is inside the monster's `AttackRangeSphere` object.

www.it-ebooks.info



Mixamo Adam with his AttackRangeSphere object highlighted in orange

Place a few of your **BP_Monster** instances inside your game; compile and run. Without any code to drive the `Monster` character to move, your monsters should just stand there idly.

Basic monster intelligence

In our game, we will add only a basic intelligence to the `Monster` characters. The monsters will know how to do two basic things:

- Track the player and follow him
- Attack the player

The monster will not do anything else. You can have the monster taunt the player when the player is first seen as well, but we'll leave that as an exercise for you.

Moving the monster – steering behavior

Monsters in very basic games don't usually have complex motion behaviors. Usually they just walk towards the target and attack it. We'll program that type of monster in this game, but mind you, you can get more interesting play with monsters that position themselves advantageously on the terrain to perform ranged attacks and so on. We're not going to program that here, but it's something to think about.

www.it-ebooks.info

In order to get the `Monster` character to move towards the player, we need to dynamically update the direction of the `Monster` character moving in each frame. To update the direction that the monster is facing, we write code in the `Monster::Tick()` method.

The `Tick` function runs in every frame of the game. The signature of the `Tick` function is:

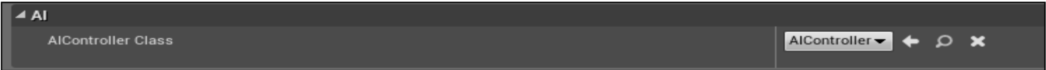
```
virtual void Tick(float DeltaSeconds) override;
```

You need to add this function's prototype to your `AMonster` class in your `Monster.h` file. If we override `Tick`, we can place our own custom behavior that the `Monster` character should do in each frame. Here's some basic code that will move the monster toward the player during each frame:

```
void AMonster::Tick(float DeltaSeconds)
{
  Super::Tick( DeltaSeconds );

  // basic intel: move the monster towards the player
  AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  if( !avatar ) return;

  FVector toPlayer = avatar->GetActorLocation() -
  GetActorLocation();
  toPlayer.Normalize(); // reduce to unit vector

  // Actually move the monster towards the player a bit
  AddMovementInput(toPlayer, Speed*DeltaSeconds);

  // At least face the target
  // Gets you the rotator to turn something
  // that looks in the `toPlayer` direction
  FRotator toPlayerRotation = toPlayer.Rotation();
  toPlayerRotation.Pitch = 0; // 0 off the pitch
  RootComponent->SetWorldRotation( toPlayerRotation );
}
```

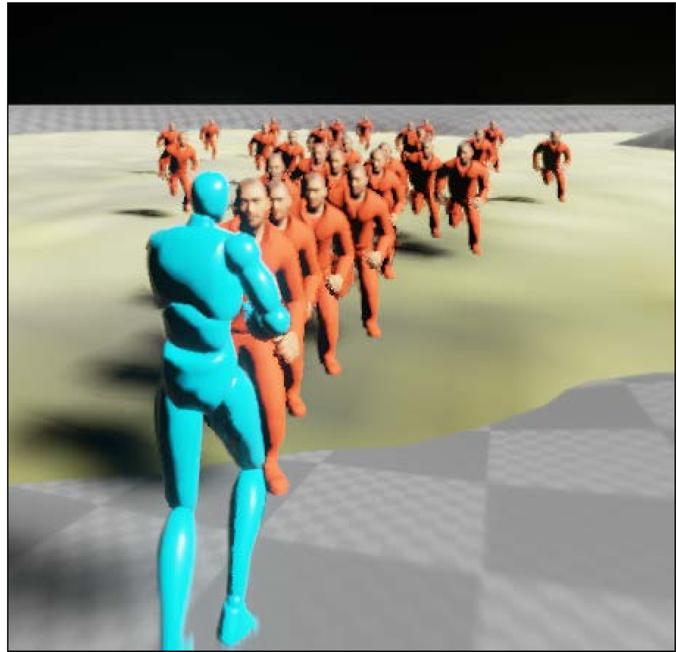

www.it-ebooks.info

For `AddMovementInput` to work, you must have a controller selected under the **AIController Class** panel in your blueprint, as shown in the following screenshot:



If you have selected `None`, calls to `AddMovementInput` won't have any effect. To prevent this, select either the `AIController` class or the `PlayerController` class as your **AIController Class**.

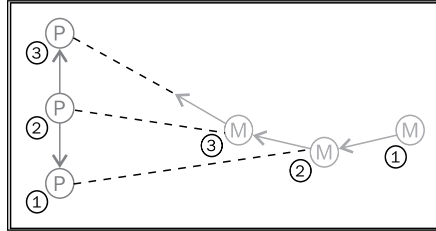
The preceding code is very simple. It comprises the most basic form of enemy intelligence: simply move toward the player by an incrementally small amount in each frame.



Our not-so-intelligent army of monsters chasing the player

www.it-ebooks.info

The result in a series of frames will be that the monster tracks and follows the player around the level. To understand how this works, you must remember that the `Tick` function is called on average about 60 times per second. What this means is that in each frame, the monster moves a tiny bit closer to the player. Since the monster moves in very small steps, his action looks smooth and continuous (while in reality, he is making small jumps and leaps in each frame).



Discrete nature of tracking: a monster's motion over three superimposed frames

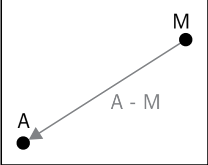
The reason why the monster moves about 60 times a second is because of a hardware constraint. The refresh rate of a typical monitor is 60 Hz, so it acts as a practical limiter on how many updates per second are useful. Updating at a frame rate faster than the refresh rate is possible, but it is not necessarily useful for games since you will only see a new picture once every 1/60 of a second on most hardware. Some advanced physics modeling simulations do almost 1,000 updates a second, but arguably, you don't need that kind of resolution for a game and you should reserve the extra CPU time for something that the player will enjoy instead, such as better AI algorithms. Some newer hardware boasts of a refresh rate up to 120 Hz (look up gaming monitors, but don't tell your parents I asked you to blow all your money on one).

The discrete nature of monster motion

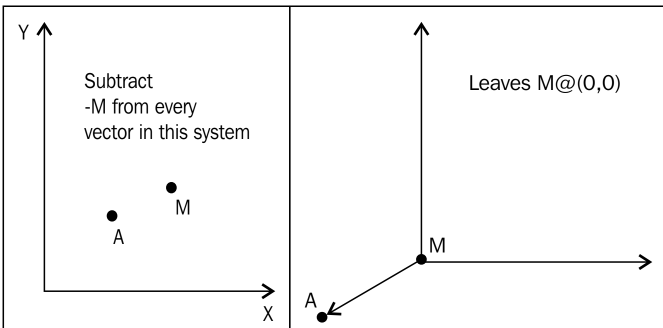
Computer games are discrete in nature. In the preceding screenshot of superimposed sequences of frames, the player is seen as moving straight up the screen, in tiny steps. The motion of the monster is also in small steps. In each frame, the monster takes one small discrete step towards the player. The monster is following an apparently curved path as he moves directly toward where the player is in each frame.

www.it-ebooks.info

To move the monster toward the player, we first have to get the player's position. Since the player is accessible in a global function, `UGameplayStatics::GetPlayerPawn`, we simply retrieve our pointer to the player using this function. Next we find the vector pointing from the `Monster` (`GetActorLocation()`) function that points to the player (`avatar->GetActorLocation()`). We need to find the vector that points from the monster to the avatar. To do this, you have to subtract the location of the monster from the location of the avatar, as shown in the following screenshot:



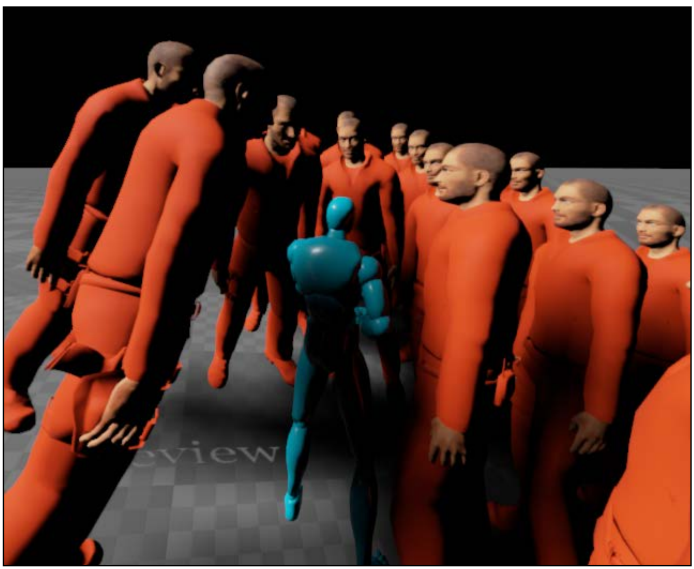
It's a simple math rule to remember but often easy to get wrong. To get the right vector, always subtract the source (the starting point) vector from the target (the terminal point) vector. In our system, we have to subtract the `Monster` vector from the `Avatar` vector. This works because subtracting the `Monster` vector from the system moves the `Monster` vector to the origin and the `Avatar` vector will be to the lower left-hand side of the `Monster` vector:



Subtracting the Monster vector from the system moves the Monster vector to (0,0)

www.it-ebooks.info

Be sure to try out your code. At this point, the monsters will be running toward your player and crowding around him. With the preceding code that is outlined, they won't attack; they'll just follow him around, as shown in the following screenshot:



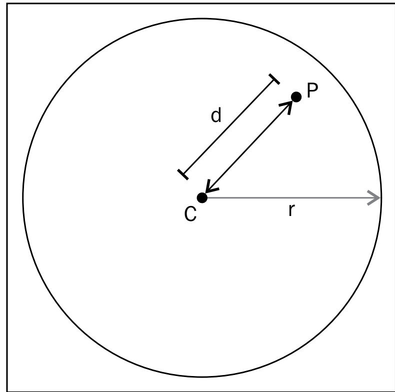
Monster SightSphere

Right now, the monsters are not paying attention to the `SightSphere` component. That is, wherever the player is in the world, the monsters will move toward him in the current setup. We want to change that now.

To do so, all we have to do is have `Monster` respect the `SightSphere` restriction. If the player is inside the monster's `SightSphere` object, the monster will give chase. Otherwise, the monsters will be oblivious to the player's location and not chase the player.

www.it-ebooks.info

Checking to see if an object is inside a sphere is simple. In the following screenshot, the point **p** is inside the sphere if the distance **d** between **p** and the centroid **c** is less than the sphere radius, **r**:



P is inside the sphere when d is less than r

So, in our code, the preceding screenshot translates to the following code:

```
void AMonster::Tick(float DeltaSeconds)
{
  Super::Tick( DeltaSeconds );
  AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  if( !avatar ) return;
    FVector toPlayer = avatar->GetActorLocation() -
    GetActorLocation();
  float distanceToPlayer = toPlayer.Size();
  // If the player is not in the SightSphere of the monster,
  // go back
  if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
  {
    // If the player is out of sight,
    // then the enemy cannot chase
    return;
  }

  toPlayer /= distanceToPlayer;  // normalizes the vector
  // Actually move the monster towards the player a bit
  AddMovementInput(toPlayer, Speed*DeltaSeconds);
  // (rest of function same as before (rotation))
}
```


www.it-ebooks.info

The preceding code adds additional intelligence to the `Monster` character. The `Monster` character can now stop chasing the player if the player is outside the monster's `SightSphere` object. This is how the result will look:

A good thing to do here will be to wrap up the distance comparison into a simple inline function. We can provide these two inline member functions in the `Monster` header as follows:

```
inline bool isInSightRange( float d )
{ return d < SightSphere->GetScaledSphereRadius(); }
inline bool isInAttackRange( float d )
{ return d < AttackRangeSphere->GetScaledSphereRadius(); }
```

These functions return the value `true` when the passed parameter, `d`, is inside the spheres in question.

An `inline` function means that the function is more like a macro than a function. Macros are copied and pasted to the calling location, while functions are jumped to by C++ and executed at their location. Inline functions are good because they give good performance while keeping the code easy to read and they are reusable.

Monster attacks on the player

There are a few different types of attacks that monsters can do. Depending on the type of the `Monster` character, a monster's attack might be melee (close range) or ranged (projectile weapon).

www.it-ebooks.info

The `Monster` character will attack the player whenever the player is in his `AttackRangeSphere`. If the player is out of the range of the monster's `AttackRangeSphere` but the player is in the `SightSphere` object of the monster, then the monster will move closer to the player until the player is in the monster's `AttackRangeSphere`.

Melee attacks

The dictionary definition of *melee* is a confused mass of people. A melee attack is one that is done at a close range. Picture a bunch of *zerglings* battling it out with a bunch of *ultralisks* (if you're a *StarCraft* player, you'll know that both zerglings and ultralisks are melee units). Melee attacks are basically close range, hand-to-hand combat. To do a melee attack, you need a melee attack animation that turns on when the monster begins his melee attack. To do this, you need to edit the animation blueprint in *Persona*, UE4's animation editor.

[Zak Parrish's *Persona* series is an excellent place to get started with in order to program animations in blueprints: `https://www.youtube.com/watch?v=AqYmC2wn7Cg&list=PL6VDVOqa_mdNW6JEu9UAS_s40OCD_u6yp&index=8`.]

For now, we will just program the melee attack and then worry about modifying the animation in blueprints later.

Defining a melee weapon

There are going to be three parts to defining our melee weapon. The first part is the C++ code that represents it. The second is the model, and the third is to connect the code and model together using a UE4 blueprint.

Coding for a melee weapon in C++

We will define a new class, `AMeleeWeapon` (derived from `AActor`), to represent hand-held combat weapons. I will attach a couple of blueprint-editable properties to the `AMeleeWeapon` class, and the `AMeleeWeapon` class will look as shown in the following code:

```
class AMonster;

UCLASS()
class GOLDENEGG_API AMeleeWeapon : public AActor
{
  GENERATED_UCLASS_BODY()
```


www.it-ebooks.info

```
    // The amount of damage attacks by this weapon do
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    MeleeWeapon)
    float AttackDamage;

    // A list of things the melee weapon already hit this swing
    // Ensures each thing sword passes thru only gets hit once
    TArray<AActor*> ThingsHit;

    // prevents damage from occurring in frames where
    // the sword is not swinging
    bool Swinging;

    // "Stop hitting yourself" - used to check if the
    // actor holding the weapon is hitting himself
    AMonster *WeaponHolder;

    // bounding box that determines when melee weapon hit
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    MeleeWeapon)
    UBoxComponent* ProxBox;

    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    MeleeWeapon)
    UStaticMeshComponent* Mesh;

    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
    SweepResult );
    void Swing();
    void Rest();
  };
```

Notice how I used a bounding box for `ProxBox`, and not a bounding sphere. This is because swords and axes will be better approximated by boxes rather than spheres. There are two member functions, `Rest()` and `Swing()`, which let `MeleeWeapon` know what state the actor is in (resting or swinging). There's also a `TArray<AActor*> ThingsHit` property inside this class that keeps track of the actors hit by this melee weapon on each swing. We are programming it so that the weapon can only hit each thing once per swing.

www.it-ebooks.info

The `AMeleeWeapon.cpp` file will contain just a basic constructor and some simple code to send damages to `OtherActor` when our sword hits him. We will also implement the `Rest()` and `Swing()` functions to clear the list of things hit.
The `MeleeWeapon.cpp` file has the following code:

```
AMeleeWeapon::AMeleeWeapon(const class FObjectInitializer& PCIP) :
  Super(PCIP)
{
  AttackDamage = 1;
  Swinging = false;
  WeaponHolder = NULL;

  Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
  TEXT("Mesh"));
  RootComponent = Mesh;

  ProxBox = PCIP.CreateDefaultSubobject<UBoxComponent>(this,
  TEXT("ProxBox"));
  ProxBox->OnComponentBeginOverlap.AddDynamic( this,
  &AMeleeWeapon::Prox );
  ProxBox->AttachTo( RootComponent );
}

void AMeleeWeapon::Prox_Implementation( AActor* OtherActor,
  UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
  bFromSweep, const FHitResult & SweepResult )
{
  // don't hit non root components
  if( OtherComp != OtherActor->GetRootComponent() )
  {
    return;
  }

  // avoid hitting things while sword isn't swinging,
  // avoid hitting yourself, and
  // avoid hitting the same OtherActor twice
  if( Swinging && OtherActor != WeaponHolder &&
  !ThingsHit.Contains(OtherActor) )
  {
    OtherActor->TakeDamage( AttackDamage + WeaponHolder-
    >BaseAttackDamage, FDamageEvent(), NULL, this );
    ThingsHit.Add( OtherActor );
  }
}
void AMeleeWeapon::Swing()
```


www.it-ebooks.info

```
{
  ThingsHit.Empty();  // empty the list
  Swinging = true;
}
void AMeleeWeapon::Rest()
{
  ThingsHit.Empty();
  Swinging = false;
}
```

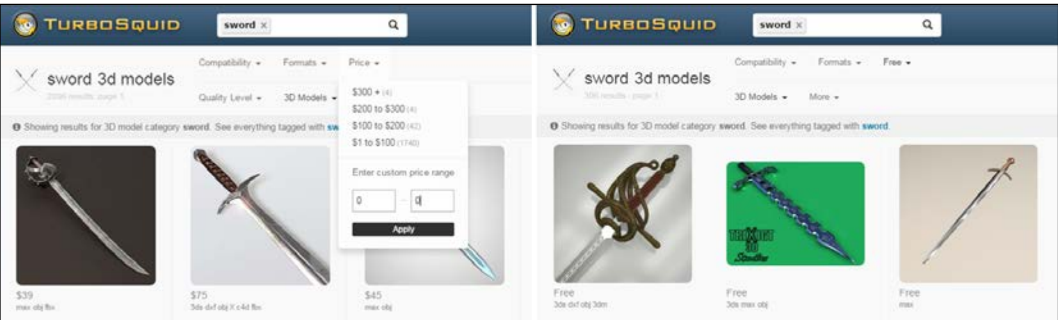
Downloading a sword

To complete this exercise, we need a sword to put into the model's hand. I added a sword to the project called *Kilic* from `http://tf3dm.com/3d-model/sword-95782.html` by Kaan Gülhan. The following is a list of other places where you will get free models:

- `http://www.turbosquid.com/`
- `http://tf3dm.com/`
- `http://archive3d.net/`
- `http://www.3dtotal.com/`

Secret tip

It might appear at first on `TurboSquid.com` that there are no free models. In fact, the secret is that you have to search in the price range $0-$0 to find them. $0 means free.

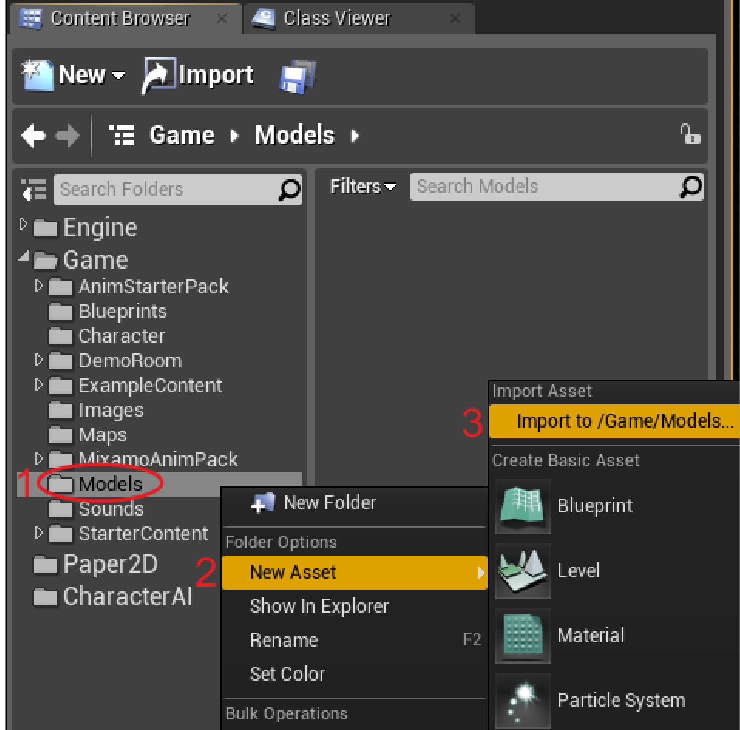


TurboSquid's search for free swords

I had to edit the *kilic* sword mesh slightly to fix the initial sizing and rotation. You can import any mesh in the **Filmbox** (**FBX**) format into your game. The kilic sword model is in the sample code package for *Chapter 11*, *Monsters*.

www.it-ebooks.info

To import your sword into the UE4 editor, right-click on any folder you want to add the model to. Navigate to **New Asset** | **Import to** | **Game** | **Models...**, and from the file explorer that pops up, select the new asset you want to import. If a **Models** folder doesn't exist, you can create one by simply right-clicking on the tree view at the left and selecting **New Folder** in the pane on the left-hand side of the **Content Browser** tab. I selected the `kilic.fbx` asset from my desktop.

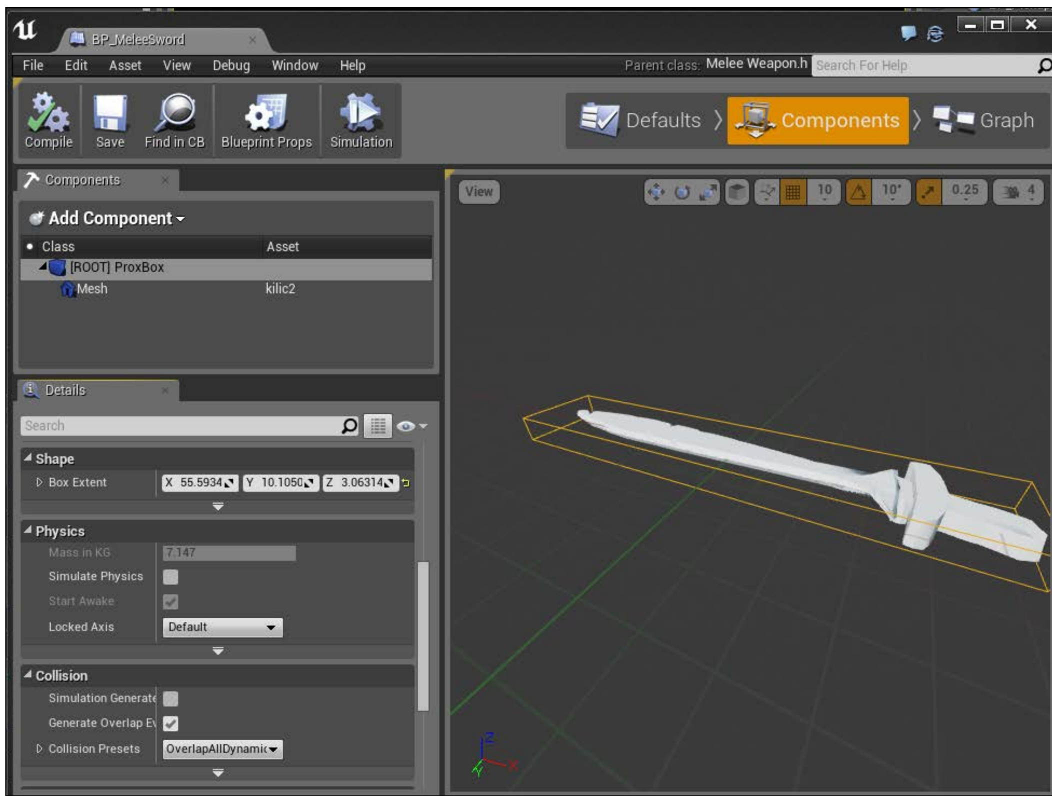


Importing to your project

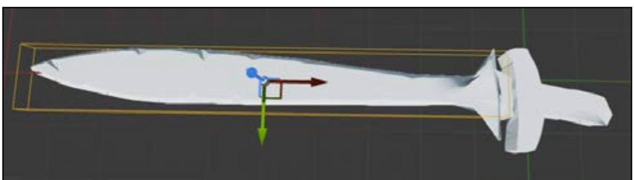
www.it-ebooks.info

Creating a blueprint for your melee weapon

Inside the UE4 editor, create a blueprint based off of `AMeleeWeapon` called `BP_MeleeSword`. Configure `BP_MeleeSword` to use the *kilic* blade model (or any blade model you choose), as shown in the following screenshot:

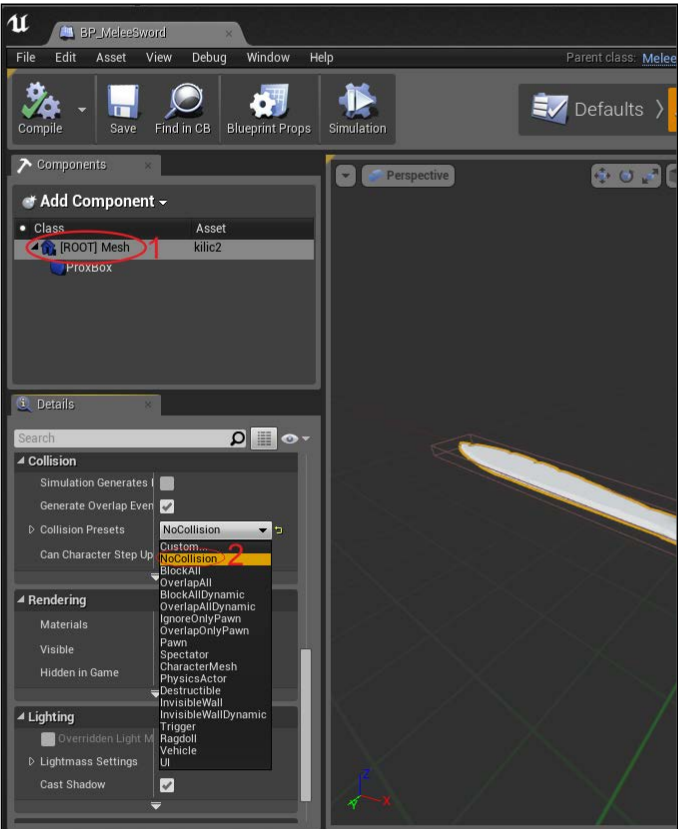


The `ProxBox` class will determine whether something was hit by the weapon, so we will modify the `ProxBox` class such that it just encloses the blade of the sword, as shown in the following screenshot:



www.it-ebooks.info

Also, under the **Collision Presets** panel, it is important to select the **NoCollision** option for the mesh (not **BlockAll**). This is illustrated in the following screenshot:



If you select **BlockAll**, then the game engine will automatically resolve all the interpenetration between the sword and the characters by pushing away things that the sword touches whenever it is swung. The result is that your characters will appear to go flying whenever a sword is swung.

Sockets

A socket in UE4 is a receptacle on one skeletal mesh for another `Actor`. You can place a socket anywhere on a skeletal mesh body. After you have correctly placed the socket, you can attach another `Actor` to this socket in UE4 code.

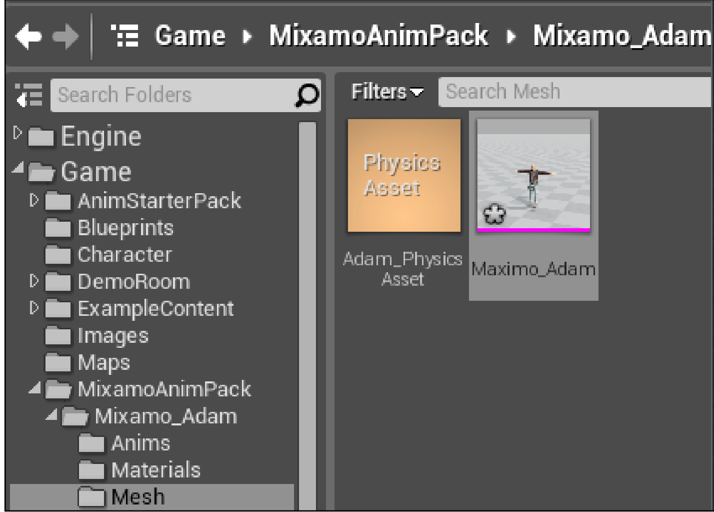
For example, if we want to put a sword in our monster's hand, we'd just have to create a socket in our monster's hand. We can attach a helmet to the player by creating a socket on his head.

www.it-ebooks.info

Creating a skeletal mesh socket in the monster's hand

To attach a socket to the monster's hand, we have to edit the skeletal mesh that the monster is using. Since we used the `Mixamo_Adam` skeletal mesh for the monster, we have to open and edit this skeletal mesh.

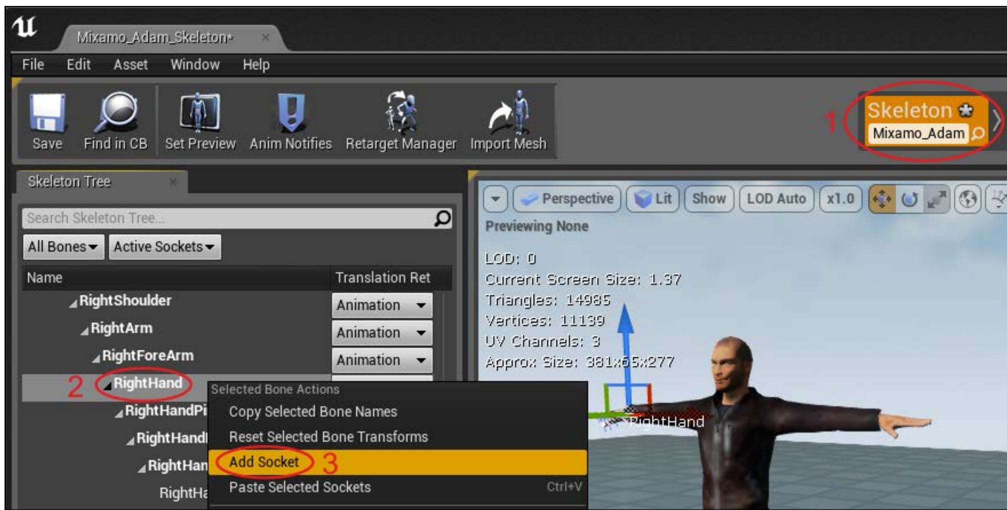
To do so, double-click on the **Mixamo_Adam** skeletal mesh in the **Content Browser** tab (this will appear as the T-pose) to open the skeletal mesh editor. If you don't see **Mixamo Adam** in your **Content Browser** tab, make sure that you have imported the **Mixamo Animation Pack** file into your project from the Unreal Launcher app.



Edit the Maximo_Adam mesh by double-clicking on the Maximo_Adam skeletal mesh object

www.it-ebooks.info

Click on **Skeleton** at the top-right corner of the screen. Scroll down the tree of bones in the left-hand side panel until you find the **RightHand** bone. We will attach a socket to this bone. Right-click on the **RightHand** bone and select **Add Socket**, as shown in the following screenshot:



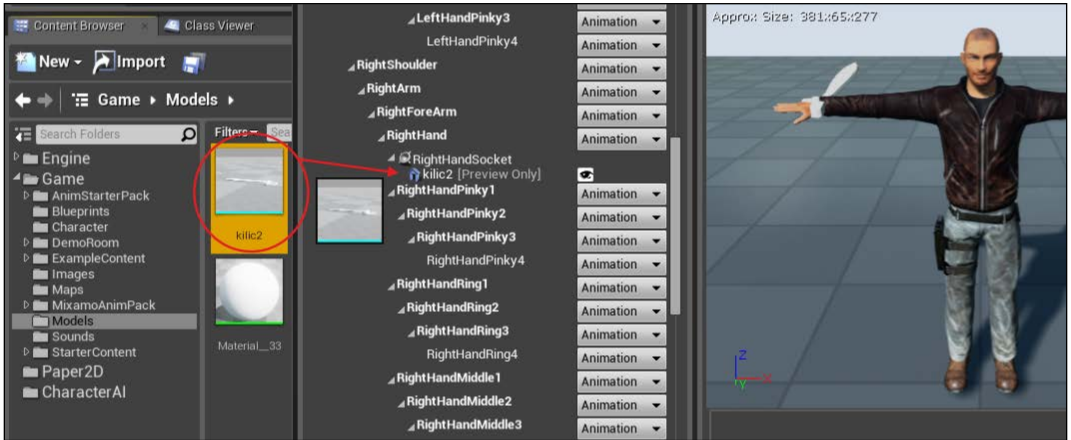
You can leave the default name (**RightHandSocket**) or rename the socket if you like, as shown in the following screenshot:

Next, we need to add a sword to the actor's hand.

www.it-ebooks.info

Attaching the sword to the model

With the Adam skeletal mesh open, find the **RightHandSocket** option in the tree view. Since Adam swings with his right hand, you should attach the sword to his right hand. Drag and drop your sword model into the **RightHandSocket** option. You should see Adam grip the sword in the image of the model at the right-hand side of the following screenshot:



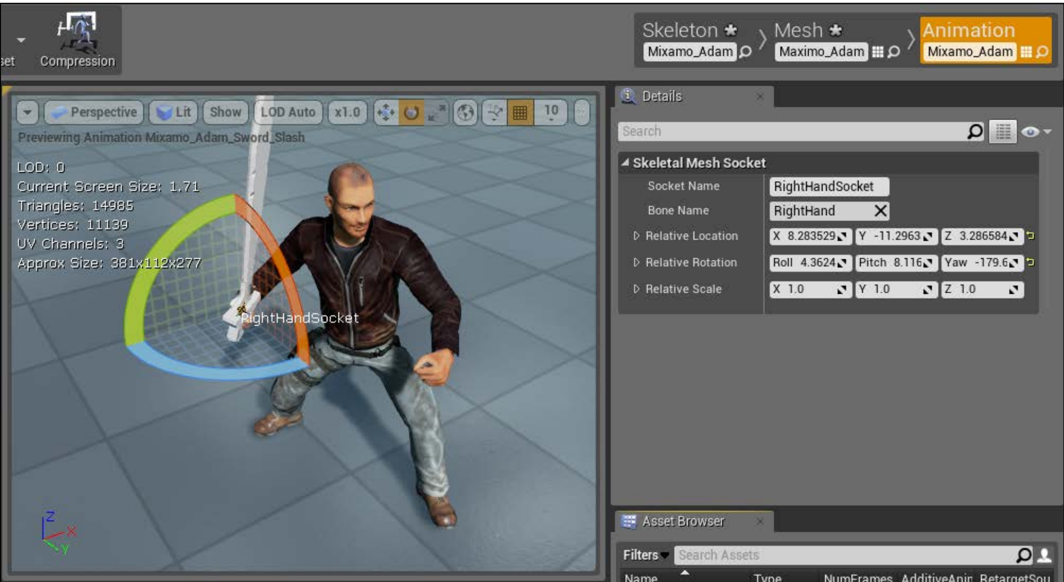
Now, click on **RightHandSocket** and zoom in on Adam's hand. We need to adjust the positioning of the socket in the preview so that the sword fits in it correctly. Use the move and rotate manipulators to line the sword up so that it fits in his hand correctly.

Positioning the socket in the right hand so that the sword rests correctly

www.it-ebooks.info

> **A real-world tip**
> If you have several sword models that you want to switch in and out of the same **RightHandSocket**, you will need to ensure quite a bit of uniformity (lack of anomalies) between the different swords that are supposed to go in that same socket.

You can preview your animations with the sword in the hand by going to the **Animation** tab in the top-right corner of the screen.



Equipping the model with a sword

However, if you launch your game, Adam won't be holding a sword. That's because adding the sword to the socket in *Persona* is for preview purposes only.

Code to equip the player with a sword

To equip your player with a sword from the code and permanently bind it to the actor, instantiate an `AMeleeWeapon` instance and attach it to `RightHandSocket` after the monster instance is initialized. We do this in `PostInitializeComponents()` since in this function the `Mesh` object will have been fully initialized already.

www.it-ebooks.info

In the Monster.h file, add a hook to select the **Blueprint** class name (UClass) of a melee weapon to use. Also add a hook for a variable to actually store the MeleeWeapon instance using the following code:

```
// The MeleeWeapon class the monster uses
// If this is not set, he uses a melee attack
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
UClass* BPMeleeWeapon;

// The MeleeWeapon instance (set if the character is using
// a melee weapon)
AActor* MeleeWeapon;
```

Now, select the BP_MeleeSword blueprint in your monster's blueprint class.

In the C++ code, you need to instantiate the weapon. To do so, we need to declare and implement a PostInitializeComponents function for the Monster class. In Monster.h, add a prototype declaration:

```
virtual void PostInitializeComponents() override;
```

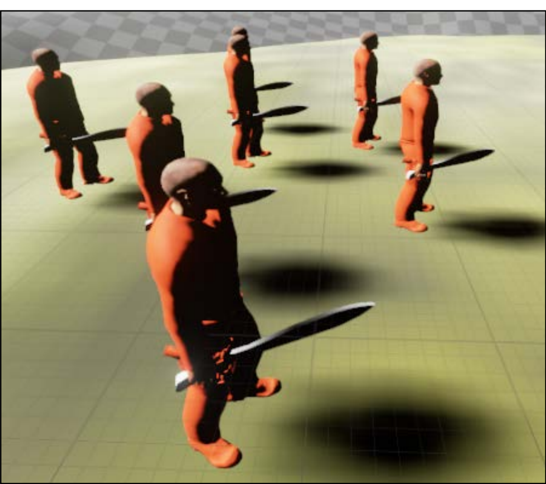
PostInitializeComponents runs after the monster object's constructor has completed and all the components of the object are initialized (including the blueprint construction). So it is the perfect time to check whether the monster has a MeleeWeapon blueprint attached to it or not and to instantiate this weapon if it does. The following code is added to instantiate the weapon in the Monster.cpp implementation of AMonster::PostInitializeComponents():

```
void AMonster::PostInitializeComponents()
{
  Super::PostInitializeComponents();

  // instantiate the melee weapon if a bp was selected
  if( BPMeleeWeapon )
  {
    MeleeWeapon = GetWorld()->SpawnActor<AMeleeWeapon>(
      BPMeleeWeapon, FVector(), FRotator() );

    if( MeleeWeapon )
    {
      const USkeletalMeshSocket *socket = Mesh->GetSocketByName(
      "RightHandSocket" ); // be sure to use correct
                           // socket name!
      socket->AttachActor( MeleeWeapon, Mesh );
    }
```


www.it-ebooks.info

```
    }
  }
```

The monsters will now start with swords in hand if `BPMeleeWeapon` is selected for that monster's blueprint.



Monsters holding weapons

Triggering the attack animation

By default, there is no connection between our C++ `Monster` class and triggering the attack animation; in other words, the `MixamoAnimBP_Adam` class has no way of knowing when the monster is in the attack state.

Therefore, we need to update the animation blueprint of the Adam skeleton (`MixamoAnimBP_Adam`) to include a query in the `Monster` class variable listing and check whether the monster is in an attacking state. We haven't worked with animation blueprints (or blueprints in general) in this book before, but follow it step by step and you should see it come together.

I will introduce blueprints terminology gently here, but I will encourage you to have a look at Zak Parrish's tutorial series at `https://www.youtube.com/playlist?list=PLZlv_N0_O1gbYMYfhhdzfW1tUV4jU0YxH` for your first introduction to blueprints.

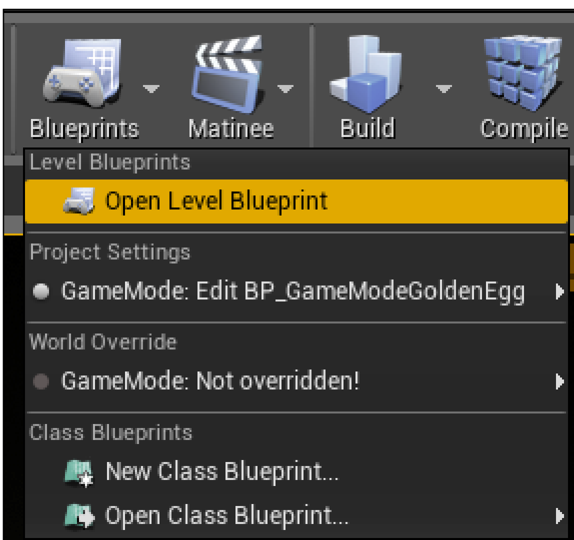
www.it-ebooks.info

Blueprint basics

A UE4 blueprint is a visual realization of the code (not to be confused with how sometimes people say that a C++ class is a metaphorical blueprint of a class instance). In UE4 blueprints, instead of actually writing code, you drag and drop elements onto a graph and connect them to achieve desired play. By connecting the right nodes to the right elements, you can program anything you want in your game.

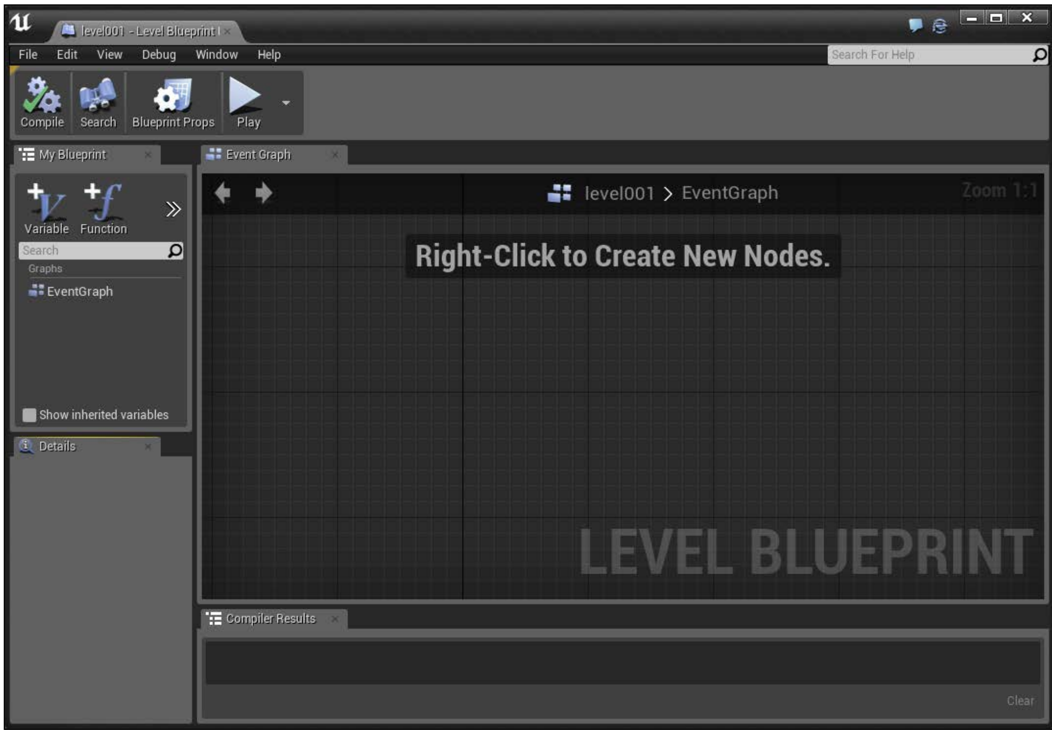
This book does not encourage the use of blueprints since we are trying to encourage you to write your own code instead. Animations, however, are best worked with blueprints, because that is what artists and designers will know.

Let's start writing a sample blueprint to get a feel how they work. First, click on the blueprint menu bar at the top and select **Open Level Blueprint**, as shown in the following screenshot:

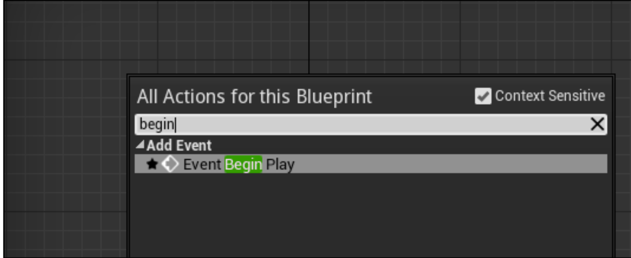


www.it-ebooks.info

The **Level Blueprint** option executes automatically when you begin the level. Once you open this window, you should see a blank slate to create your gameplay on, as shown here:

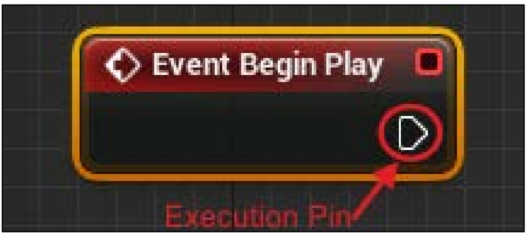


Right-click anywhere on the graph paper. Start typing `begin` and click on the **Event Begin Play** option from the drop-down list that appears. Ensure that the **Context Sensitive** checkbox is checked, as shown in the following screenshot:

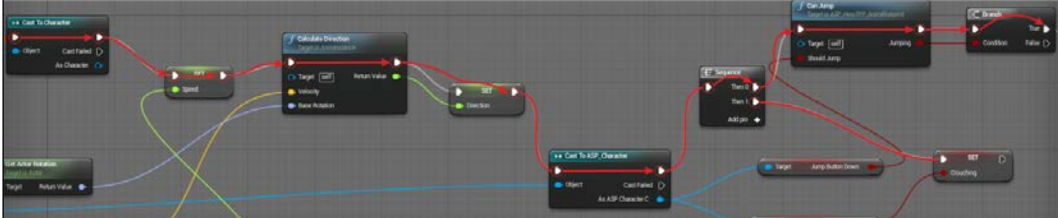


www.it-ebooks.info

Immediately after you click on the **Event Begin Play** option, a red box will appear on your screen. It has a single white pin at the right-hand side. This is called the execution pin, as shown here:



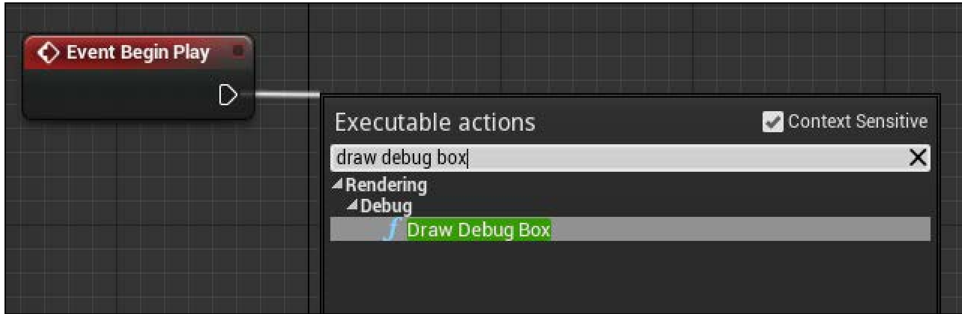
The first thing that you need to know about animation blueprints is the white pin execution path (the white line). If you've seen a blueprint graph before, you must have noticed a white line going through the graph, as shown in the following diagram:



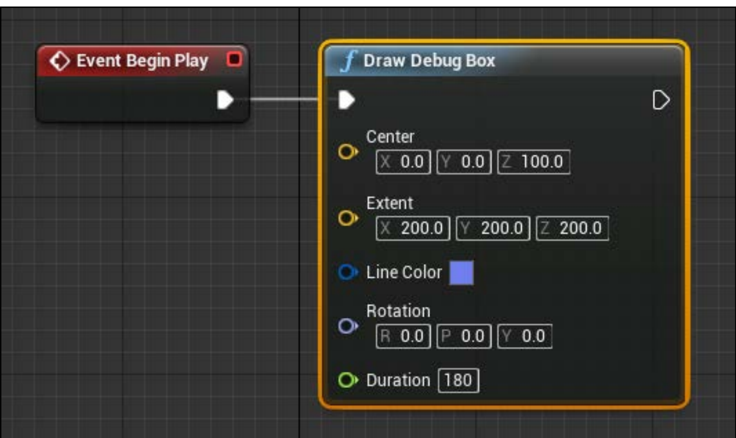
The white pin execution path is pretty much equivalent to having lines of code lined up and run one after the other. The white line determines which nodes will get executed and in what order. If a node does not have a white execution pin attached to it, then that node will not get executed at all.

www.it-ebooks.info

Drag off the white execution pin from **Event Begin Play**. Start by typing `draw debug box` in the **Executable actions** dialog. Select the first thing that pops up (**f Draw Debug Box**), as shown here:



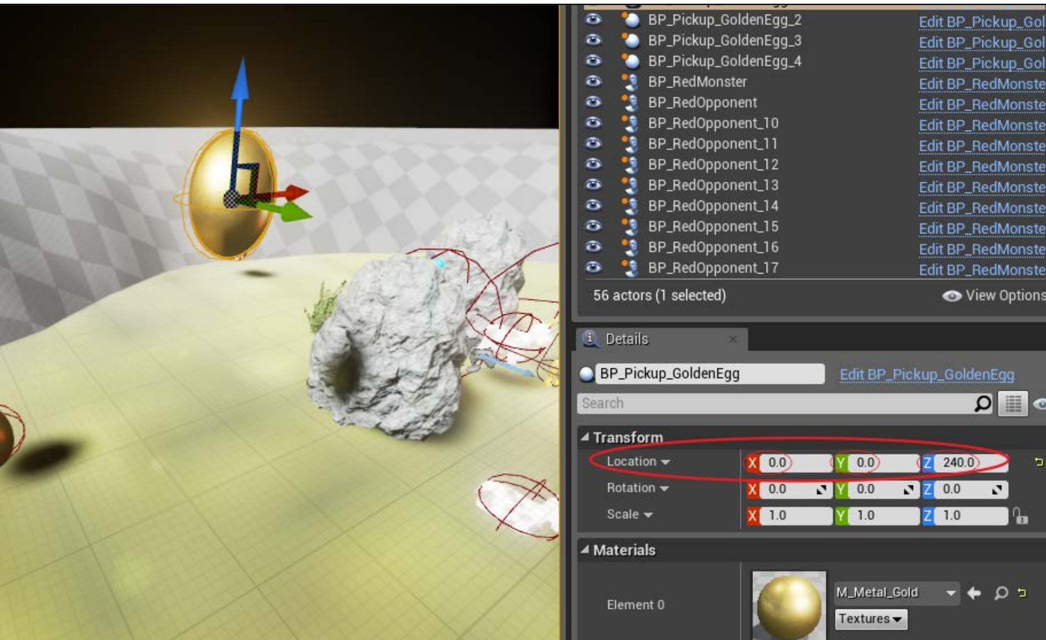
Fill in some details for how you want the box to look. Here, I selected the color blue for the box, the center of the box at (0, 0, 100), the size of the box to be (200, 200, 200), and a duration of 180 seconds (be sure to enter a duration that is long enough to see the result), as shown in the following screenshot:



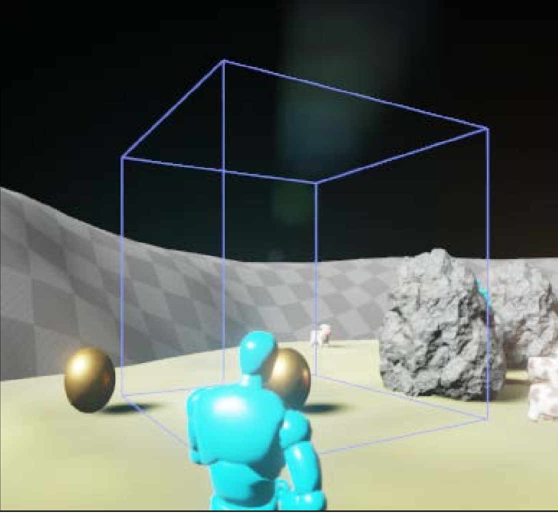
Now click on the **Play** button to realize the graph. Remember that you have to find the world's origin to see the debug box.

www.it-ebooks.info

Find the world's origin by placing a golden egg at (0, 0, (some z value)), as shown in the following screenshot:



This is how the box will look in the level:



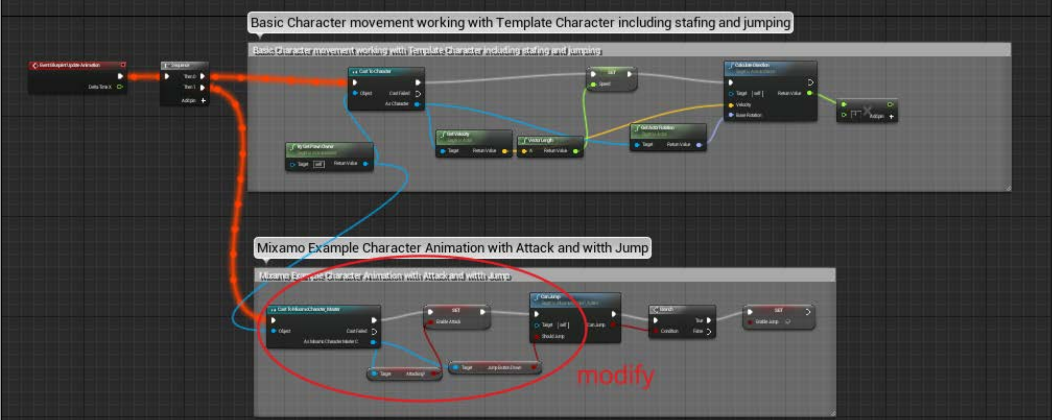
A debug box rendered at the origin

www.it-ebooks.info

Modifying the animation blueprint for Mixamo Adam

To integrate our attack animation, we have to modify the blueprint. Under **Content Browser**, open up `MixamoAnimBP_Adam`.

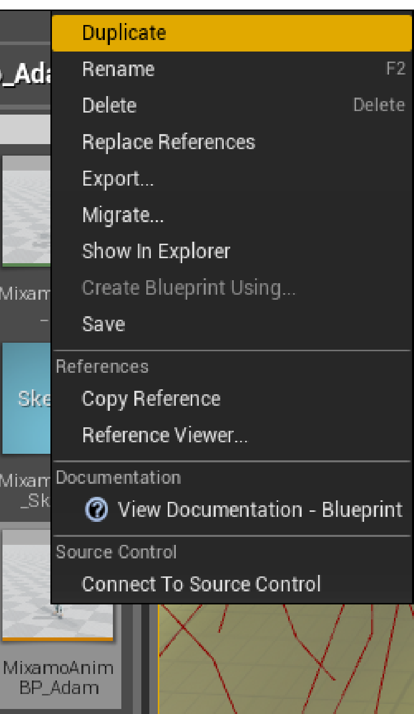
The first thing you'll notice is that the graph has two sections: a top section and a bottom section. The top section is marked "**Basic Character movement**...," while the bottom section says "**Mixamo Example Character Animation**...." Basic character movement is in charge of the walking and running movements of the model. We will be working in the **Mixamo Example Character Animation with Attack and Jump** section, which is responsible for the attack animation. We will be working in the latter section of the graph, shown in the following screenshot:



When you first open the graph, it starts out by zooming in on a section near the bottom. To scroll up, right-click the mouse and drag it upwards. You can also zoom out using the mouse wheel or by holding down the *Alt* key and the right mouse button while moving the mouse up.

www.it-ebooks.info

Before proceeding, you might want to duplicate the **MixamoAnimBP_Adam** resource so that you don't damage the original, in case you need to go back and change something later. This allows you to easily go back and correct things if you find that you've made a mistake in one of your modifications, without having to reinstall a fresh copy of the whole animation package into your project.



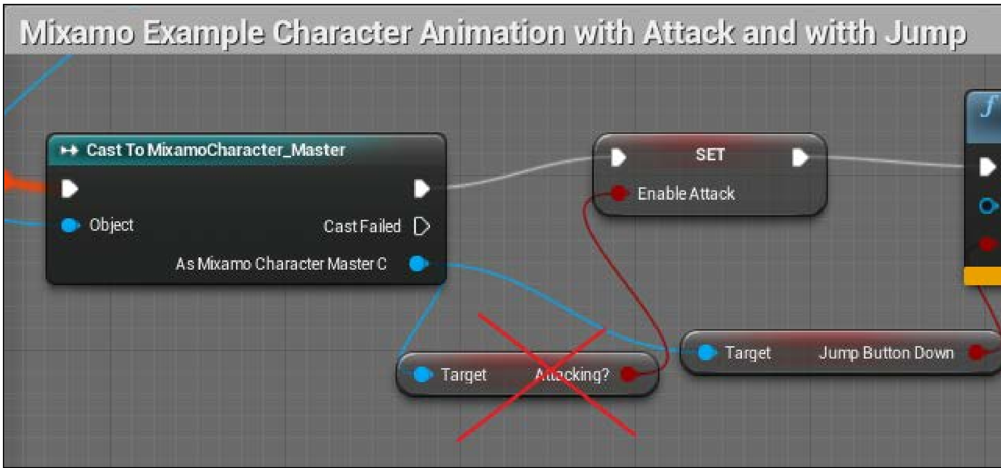
Making a duplicate of the MixamoAnimBP_Adam resource to avoid damaging the original asset

When assets are added to a project from the Unreal Launcher, a copy of the original asset is made, so you can modify **MixamoAnimBP_Adam** in your project now and get a fresh copy of the original assets in a new project later.

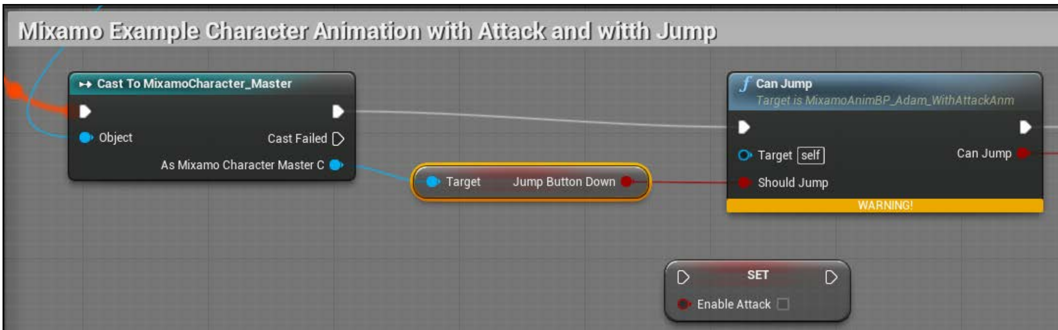
www.it-ebooks.info

We're going to do only a few things to make Adam swing the sword when he is attacking. Let's do it in order.

1. Deleting the node that says **Attacking?**:

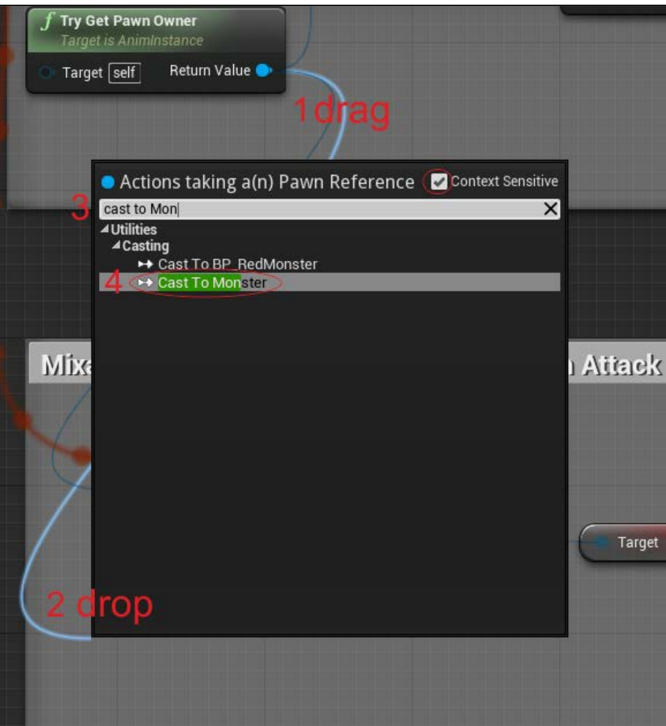


2. Rearrange the nodes, as follows, with the **Enable Attack** node by itself at the bottom:



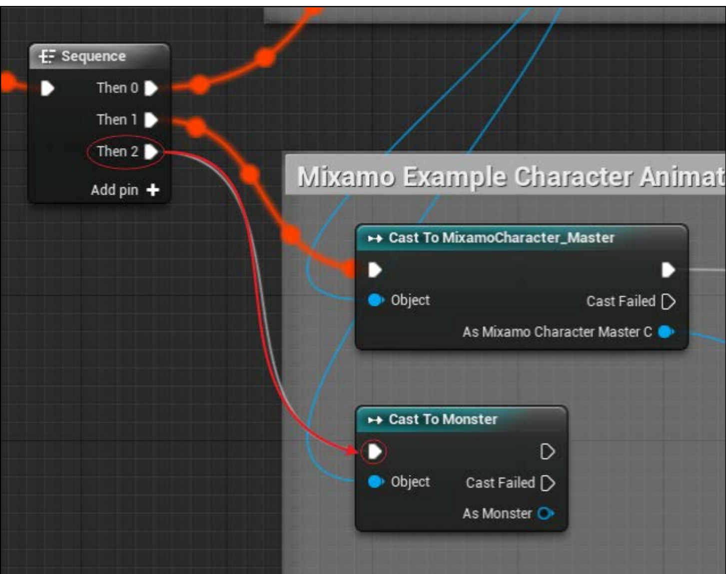
www.it-ebooks.info

3. Next we're going to handle the monster that this animation is animating. Scroll up the graph a bit and drag the blue dot marked as **Return Value** in the **Try Get Pawn Owner** dialog. Drop it into your graph, and when the pop-up menu appears, select **Cast to Monster** (ensure that **Context Sensitive** is checked, or the **Cast to Monster** option will not appear). The **Try Get Pawn Owner** option gets the `Monster` instance that owns the animation, which is just the `AMonster` class object, as shown in the following screenshot:

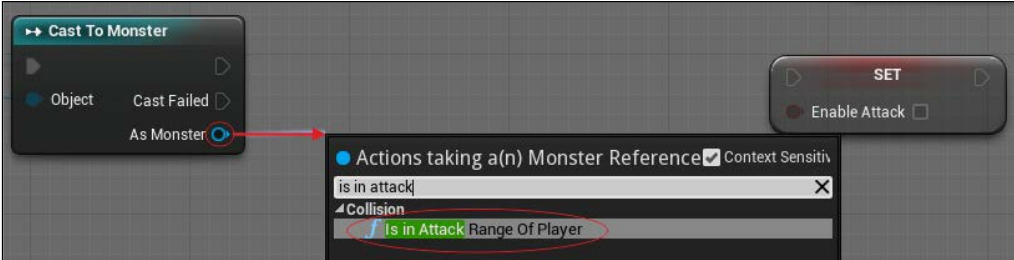


4. Click on **+** in the **Sequence** dialog and drag another execution pin from the **Sequence** group to the **Cast to Monster** node instance, as shown in the following screenshot. This ensures that the **Cast to Monster** instance actually gets executed.

www.it-ebooks.info

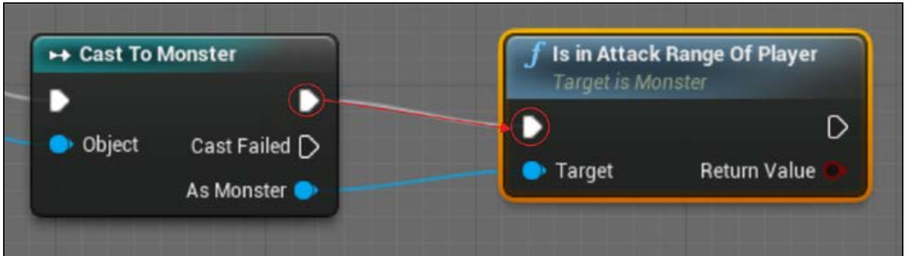


5. The next step is to pull out the pin from the **As Monster** terminal of the **Cast to Monster** node and look for the **Is in Attack Range Of Player** property:



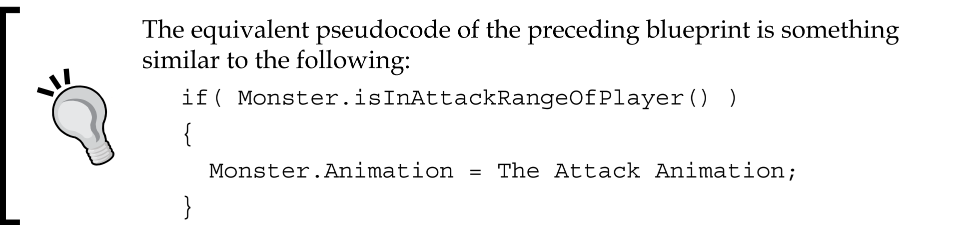
www.it-ebooks.info

6. Take the white execution pin from the **Cast to Monster** node at the left-hand side and drop it into the **Is in Attack Range Of Player** node at the right-hand side:



This ensures a transfer of control from the **Cast to Monster** operation to the **Is in Attack Range Of Player** node.

7. Pull the white and red pins over to the **SET** node, as shown here:



The equivalent pseudocode of the preceding blueprint is something similar to the following:

```
if( Monster.isInAttackRangeOfPlayer() )
{
  Monster.Animation = The Attack Animation;
}
```

Test your animation. The monster should swing only when he is within the player's range.

Code to swing the sword

We want to add an animation notify event when the sword is swung. First, declare and add a blueprint callable C++ function to your `Monster` class:

```
// in Monster.h:
UFUNCTION( BlueprintCallable, Category = Collision )
void SwordSwung();
```


www.it-ebooks.info

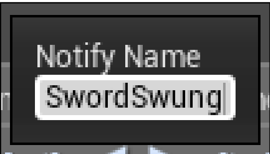
The `BlueprintCallable` statement means that it will be possible to call this function from blueprints. In other words, `SwordSwung()` will be a C++ function that we can invoke from a blueprints node, as shown here:

```
// in Monster.cpp
void AMonster::SwordSwung()
{
  if( MeleeWeapon )
  {
    MeleeWeapon->Swing();
  }
}
```

Next open the **Mixamo_Adam_Sword_Slash** animation by double-clicking on it from your **Content Browser** (it should be in **MixamoAnimPack/Mixamo_Adam/Anims/Mixamo_Adam_Sword_Slash**). Scrub the animation to the point where Adam starts swinging his sword. Right-click on the animation bar and select **New Notify** under **Add Notify...** , as shown in the following screenshot:

www.it-ebooks.info

Name the notification `SwordSwung`:



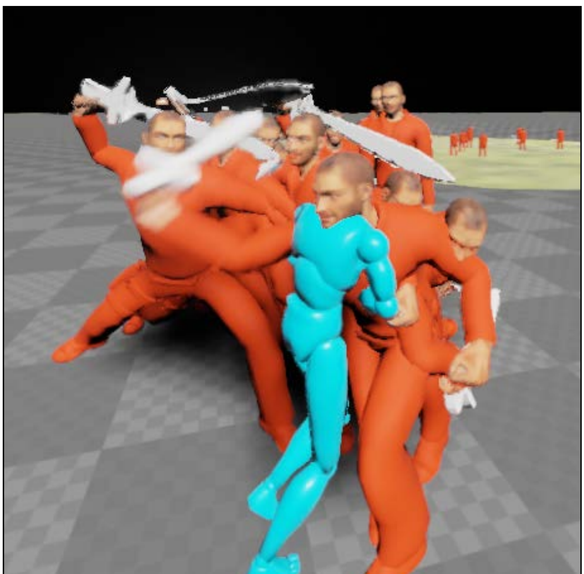
The notification name should appear in your animation's timeline, shown as follows:

Save the animation and then open up your version of **MixamoAnimBP_Adam** again. Underneath the **SET** group of nodes, create the following graph:

The **AnimNotify_SwordSwung** node appears when you right-click in the graph (with **Context Sensitive** turned on) and start typing `SwordSwung`. The **Cast To Monster** node is again fed in from the **Try Get Pawn Owner** node as in step 2 of the *Modifying the animation blueprint for Mixamo Adam* section. Finally, **Sword Swung** is our blueprint-callable C++ function in the `AMonster` class.

If you start the game now, your monsters will execute their attack animation whenever they actually attack. When the sword's bounding box comes in contact with you, you should see your HP bar go down a bit (recall that the HP bar was added at the end of *Chapter 8*, *Actors and Pawns*, as an exercise).

www.it-ebooks.info



Monsters attacking the player

Projectile or ranged attacks

Ranged attacks usually involve a projectile of some sort. Projectiles are things such as bullets, but they can also include things such as lightning magic attacks or fireball attacks. To program a projectile attack, you should spawn a new object and only apply the damage to the player if the projectile reaches the player.

To implement a basic bullet in UE4, we should derive a new object type. I derived a `ABullet` class from the `AActor` class, as shown in the following code:

```
UCLASS()
class GOLDENEGG_API ABullet : public AActor
{
  GENERATED_UCLASS_BODY()

  // How much damage the bullet does.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  Properties)
  float Damage;

  // The visible Mesh for the component, so we can see
  // the shooting object
  UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
  Collision)
```


www.it-ebooks.info

```
    UStaticMeshComponent* Mesh;

    // the sphere you collide with to do impact damage
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    Collision)
    USphereComponent* ProxSphere;

    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
    SweepResult );
};
```

The `ABullet` class has a couple of important members in it, as follows:

- A `float` variable for the damage that a bullet does on contact
- A `Mesh` variable for the body of the bullet
- A `ProxSphere` variable to detect when the bullet finally hits something
- A function to be run when `Prox` near an object is detected

The constructor for the `ABullet` class should have the initialization of the `Mesh` and `ProxSphere` variables. In the constructor, we set `RootComponent` to being the `Mesh` variable and then attach the `ProxSphere` variable to the `Mesh` variable. The `ProxSphere` variable will be used for collision checking, and collision checking for the `Mesh` variable should be turned off, as shown in the following code:

```
ABullet::ABullet(const class FObjectInitializer& PCIP) : Super(PCIP)
{
  Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
  TEXT("Mesh"));
  RootComponent = Mesh;

  ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this,
  TEXT("ProxSphere"));
  ProxSphere->AttachTo( RootComponent );

  ProxSphere->OnComponentBeginOverlap.AddDynamic( this,
  &ABullet::Prox );
  Damage = 1;
}
```


www.it-ebooks.info

We initialized the `Damage` variable to `1` in the constructor, but this can be changed in the UE4 editor once we create a blueprint out of the `ABullet` class. Next, the `ABullet::Prox_Implementation()` function should deal damages to the actor hit if we collide with the other actor's `RootComponent`, using the following code:

```
void ABullet::Prox_Implementation( AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult & SweepResult )
{
  if( OtherComp != OtherActor->GetRootComponent() )
  {
    // don't collide w/ anything other than
    // the actor's root component
    return;
  }

  OtherActor->TakeDamage( Damage, FDamageEvent(), NULL, this );
  Destroy();
}
```

Bullet physics

To make bullets fly through the level, you can use UE4's physics engine.

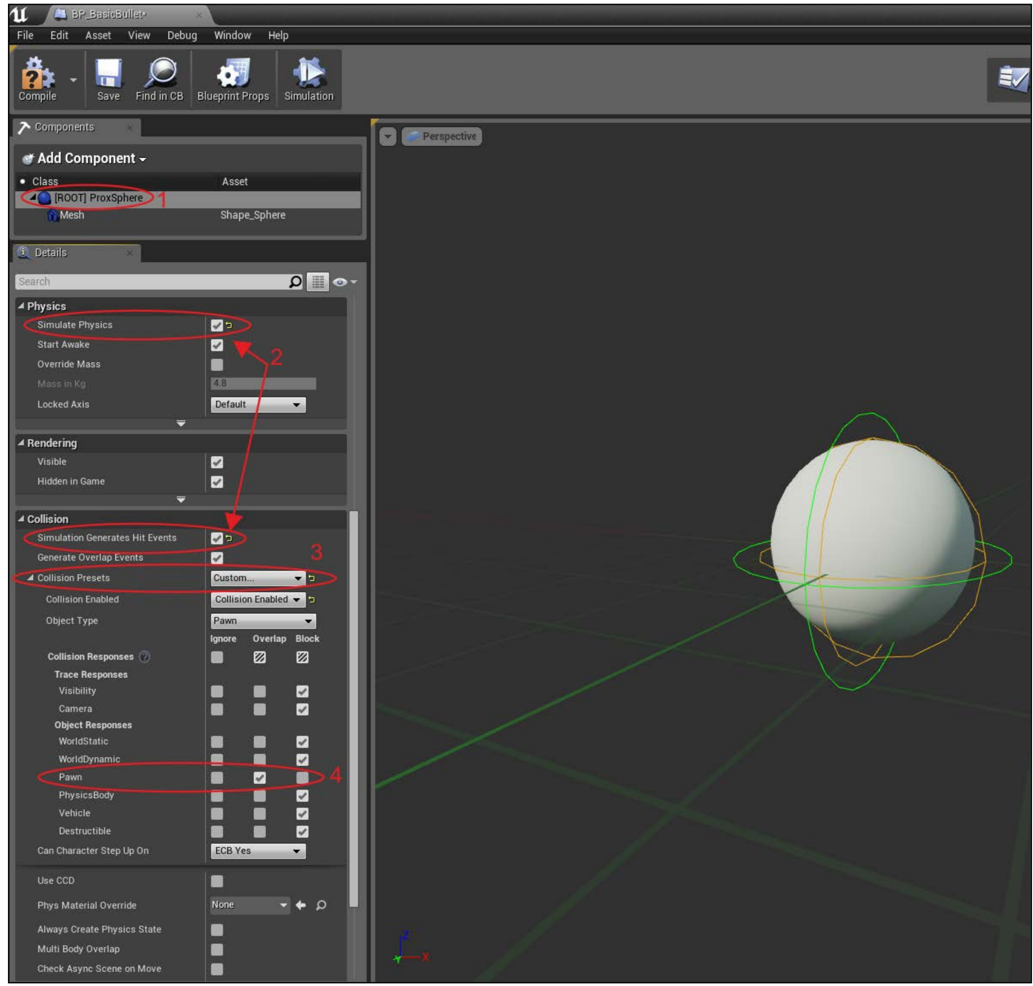
Create a blueprint based on the `ABullet` class. I selected **Shape_Sphere** for the mesh. The bullet's mesh should not have collision physics enabled; instead we'll enable physics on the bullet's bounding sphere.

Configuring the bullet to behave properly is mildly tricky, so we'll cover this in four steps, as follows:

1. Select **[ROOT] ProxSphere** in the **Components** tab. The `ProxSphere` variable should be the root component and should appear at the top of the hierarchy.
2. In the **Details** tab, check both **Simulate Physics** and **Simulation Generates Hit Events**.
3. From the **Collision Presets** dropdown, select **Custom…**.

www.it-ebooks.info

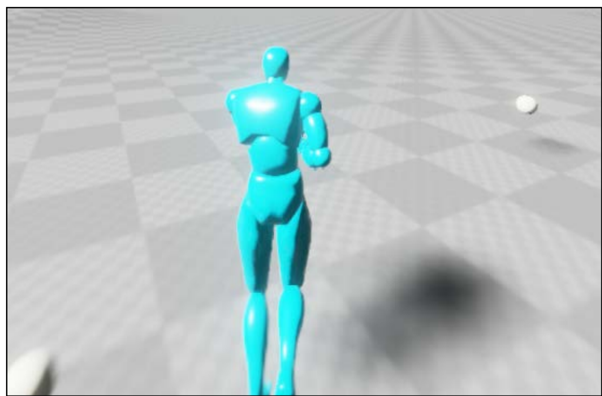
4. Check the **Collision Responses** boxes as shown; check **Block** for most types (**WorldStatic**, **WorldDynamic**, and so on) and check **Overlap** only for **Pawn**:



The **Simulate Physics** checkbox makes the `ProxSphere` property experience gravity and the impulse forces exerted on it. An impulse is a momentary thrust of force, which we'll use to drive the shot of the bullet. If you do not check the **Simulation Generate Hit Events** checkbox, then the ball will drop on the floor. What **BlockAll Collision Preset** does is ensure that the ball can't pass through anything.

www.it-ebooks.info

If you drag and drop a couple of these `BP_Bullet` objects from the **Content Browser** tab directly into the world now, they will simply fall to the floor. You can kick them once they are on the the floor. The following screenshot shows the ball object on the floor:



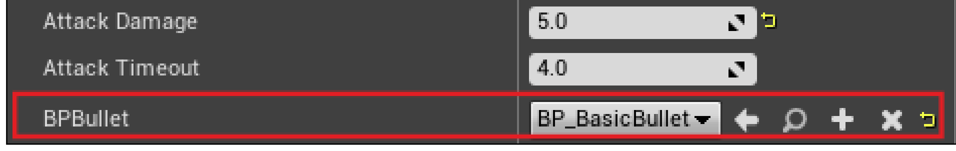
However, we don't want our bullets falling on the floor. We want them to be shot. So let's put our bullets in the `Monster` class.

Adding bullets to the monster class

Add a member to the `Monster` class that receives a blueprint instance reference. That's what the `UClass` object type is for. Also, add a blueprint configurable float property to adjust the force that shoots the bullet, as shown in the following code:

```
// The blueprint of the bullet class the monster uses
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
UClass* BPBullet;
// Thrust behind bullet launches
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
  MonsterProperties)
float BulletLaunchImpulse;
```

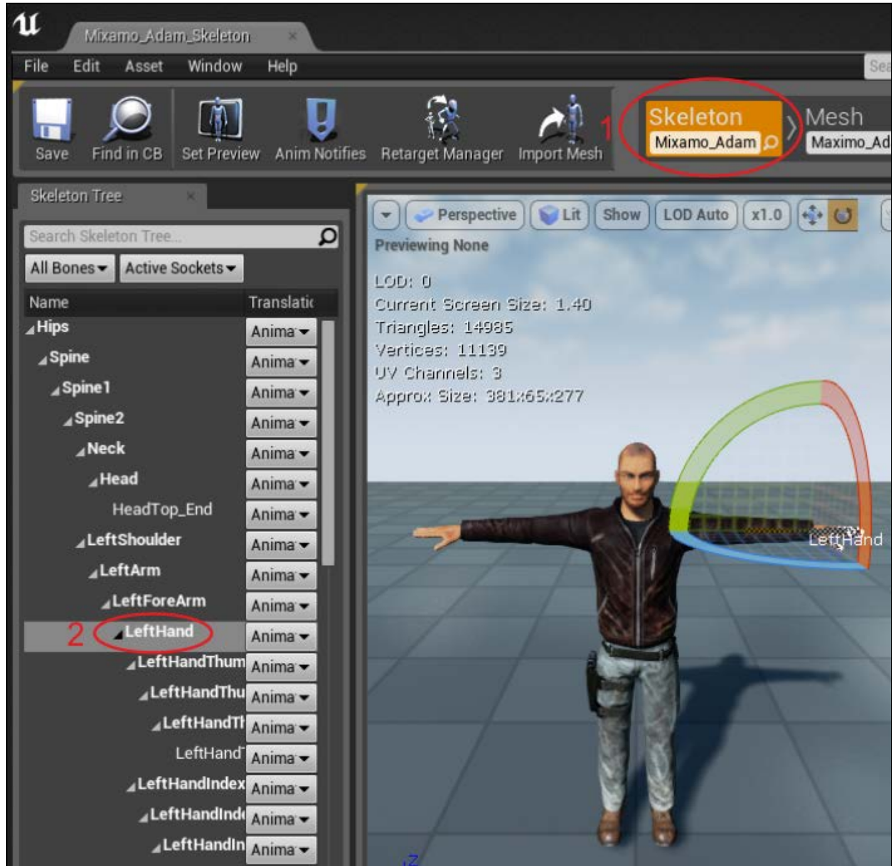
Compile and run the C++ project and open your `BP_Monster` blueprint. You can now select a blueprint class under `BPBullet`, as shown in the following screenshot:



www.it-ebooks.info

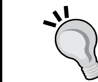
Once you've selected a blueprint class type to instantiate when the monster shoots, you have to program the monster to shoot when the player is in his range.

Where does the monster shoot from? Actually, he should shoot from a bone. If you're not familiar with the terminology, bones are just reference points in the model mesh. A model mesh is usually made up of many "bones." To see some bones, open up the **Mixamo_Adam** mesh by double-clicking on the asset in the **Content Browser** tab, as shown in the following screenshot:



Go to the **Skeleton** tab and you will see all the monster's bones in a tree view list in the left-hand side. What we want to do is select a bone from which bullets will be emitted. Here I've selected the `LeftHand` option.

www.it-ebooks.info



An artist will normally insert an additional bone into the model mesh to emit the particle, which is likely to be on the tip of the nozzle of a gun.

Working from the base model mesh, we can get the `Mesh` bone's location and have the monster emit the `Bullet` instances from that bone in the code.

The complete monster `Tick` and `Attack` functions can be obtained using the following code:

```
void AMonster::Tick(float DeltaSeconds)
{
  Super::Tick( DeltaSeconds );

  // move the monster towards the player
  AAvatar *avatar = Cast<AAvatar>(
  UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
  if( !avatar ) return;

  FVector playerPos = avatar->GetActorLocation();
  FVector toPlayer = playerPos - GetActorLocation();
  float distanceToPlayer = toPlayer.Size();

  // If the player is not the SightSphere of the monster,
  // go back
  if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
  {
    // If the player is OS, then the enemy cannot chase
    return;
  }

  toPlayer /= distanceToPlayer;  // normalizes the vector

  // At least face the target
  // Gets you the rotator to turn something
  // that looks in the `toPlayer` direction
  FRotator toPlayerRotation = toPlayer.Rotation();
  toPlayerRotation.Pitch = 0; // 0 off the pitch
  RootComponent->SetWorldRotation( toPlayerRotation );

  if( isInAttackRange(distanceToPlayer) )
  {
    // Perform the attack
    if( !TimeSinceLastStrike )
```


www.it-ebooks.info

```
        {
          Attack(avatar);
        }

        TimeSinceLastStrike += DeltaSeconds;
        if( TimeSinceLastStrike > AttackTimeout )
        {
          TimeSinceLastStrike = 0;
        }

        return;  // nothing else to do
      }
      else
      {
        // not in attack range, so walk towards player
        AddMovementInput(toPlayer, Speed*DeltaSeconds);
      }
    }
```

The `AMonster::Attack` function is relatively simple. Of course, we first need to add a prototype declaration in the `Monster.h` file in order to write our function in the `.cpp` file:

```
void AMonster::Attack(AActor* thing);
```

In `Monster.cpp`, we implement the `Attack` function, as follows:

```
void AMonster::Attack(AActor* thing)
{
  if( MeleeWeapon )
  {
    // code for the melee weapon swing, if
    // a melee weapon is used
    MeleeWeapon->Swing();
  }
  else if( BPBullet )
  {
    // If a blueprint for a bullet to use was assigned,
    // then use that. Note we wouldn't execute this code
    // bullet firing code if a MeleeWeapon was equipped
    FVector fwd = GetActorForwardVector();
    FVector nozzle = GetMesh()->GetBoneLocation( "RightHand" );
    nozzle += fwd * 155;// move it fwd of the monster so it
    doesn't
    // collide with the monster model
```


www.it-ebooks.info

```
    FVector toOpponent = thing->GetActorLocation() - nozzle;
    toOpponent.Normalize();
    ABullet *bullet = GetWorld()->SpawnActor<ABullet>(
    BPBullet, nozzle, RootComponent->GetComponentRotation());

    if( bullet )
    {
      bullet->Firer = this;
      bullet->ProxSphere->AddImpulse(
        fwd*BulletLaunchImpulse );
    }
    else
    {
      GEngine->AddOnScreenDebugMessage( 0, 5.f,
      FColor::Yellow, "monster: no bullet actor could be spawned.
      is the bullet overlapping something?" );
    }
  }
}
```

We leave the code that implements the melee attack as it is. Assuming that the monster is not holding a melee weapon, we then check whether the `BPBullet` member is set. If the `BPBullet` member is set, it means that the monster will create and fire an instance of the `BPBullet` blueprinted class.

Pay special attention to the following line:

```
ABullet *bullet = GetWorld()->SpawnActor<ABullet>(BPBullet,
  nozzle, RootComponent->GetComponentRotation() );
```

This is how we add a new actor to the world. The `SpawnActor()` function puts an instance of `UCLASS` that you pass, at `spawnLoc`, with some initial orientation.

After we spawn the bullet, we call the `AddImpulse()` function on its `ProxSphere` variable to rocket it forward.

Player knockback

To add a knockback to the player, I added a member variable to the `Avatar` class called `knockback`. A knockback happens whenever the avatar gets hurt:

```
FVector knockback; // in class AAvatar
```


www.it-ebooks.info

In order to figure out the direction to knock the player back when he gets hit, we need to add some code to `AAvatar::TakeDamage`. Compute the direction vector from the attacker towards the player and store this vector in the `knockback` variable:

```
float AAvatar::TakeDamage(float Damage, struct FDamageEvent const&
  DamageEvent, AController* EventInstigator, AActor* DamageCauser)
{
  // add some knockback that gets applied over a few frames
  knockback = GetActorLocation() - DamageCauser-
  >GetActorLocation();
  knockback.Normalize();
  knockback *= Damage * 500; // knockback proportional to damage
}
```

In `AAvatar::Tick`, we apply the knockback to the avatar's position:

```
void AAvatar::Tick( float DeltaSeconds )
{
  Super::Tick( DeltaSeconds );

  // apply knockback vector
  AddMovementInput( knockback, 1.f );

  // half the size of the knockback each frame
  knockback *= 0.5f;
}
```

Since the knockback vector reduces in size with each frame, it becomes weaker over time, unless the knockback vector gets renewed with another hit.

Summary

In this chapter, we explored how to instantiate monsters on the screen that run after the player and attack him. In the next chapter, we will give the player the ability to defend himself by allowing him to cast spells that damage the monsters.

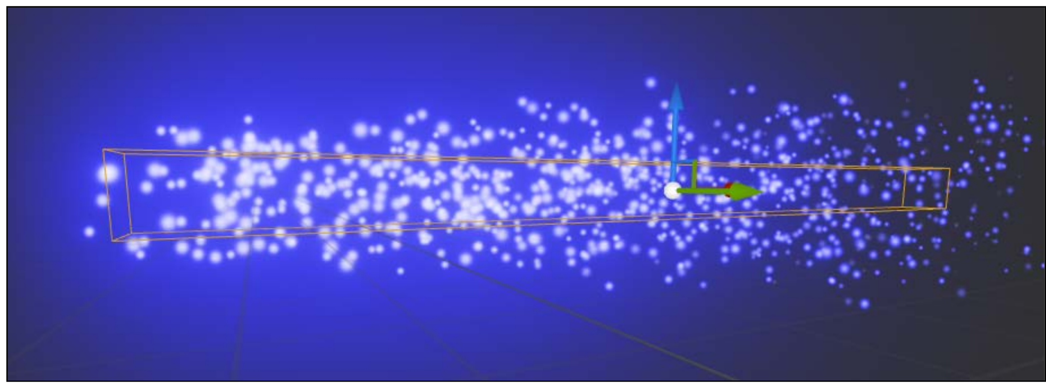
www.it-ebooks.info

12
Spell Book

The player does not yet have a means to defend himself. We will equip the player with a very useful and interesting way, of doing so called magic spells. Magic spells will be used by the player to affect monsters nearby.

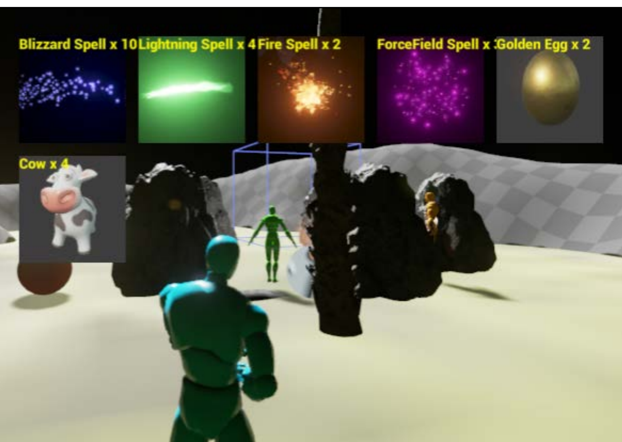
Practically, spells will be a combination of a particle system with an area of effect represented by a bounding volume. The bounding volume is checked for contained actors in each frame. When an actor is within the bounding volume of a spell, then that actor is affected by that spell.

The following is a screenshot of the blizzard and force field spells, with their bounding volumes highlighted in orange:



Visualization of the blizzard spell can be seen at the right, with a long, box-shaped bounding volume. Visualization of the force field spell, with a spherical bounding volume, for pushing monsters away, is shown in the following screenshot:

In each frame, the bounding volume is checked for contained actors. Any actor contained in the spell's bounding volume is going to be affected by that spell for that frame only. If the actor moves outside the spell's bounding volume, the actor will no longer be affected by that spell. Remember, the spell's particle system is a visualization only; the particles themselves are not what will affect game actors. The `PickupItem` class we created in *Chapter 8*, *Actors and Pawns* can be used to allow the player to pick up items representing the spells. We will extend the `PickupItem` class and attach the blueprint of a spell to cast each `PickupItem`. Clicking on a spell's widget from the HUD will cast it. The interface will look something like this:



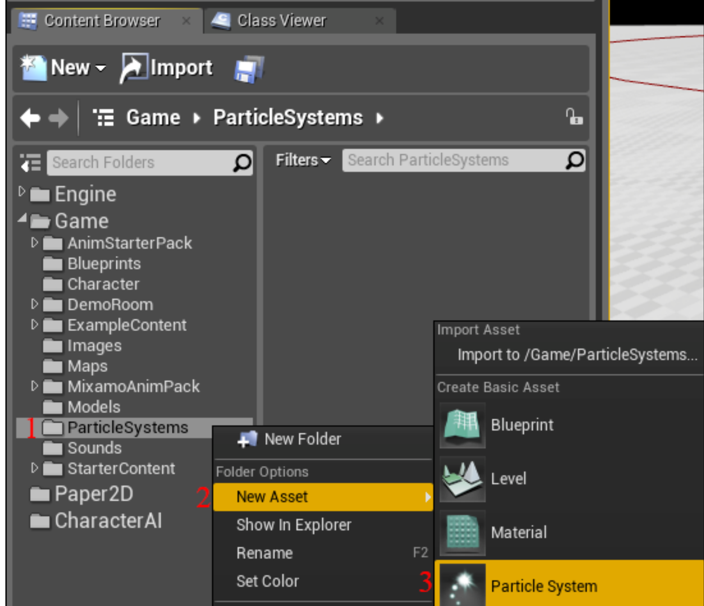
Items the player has picked up, including four different spells

www.it-ebooks.info

We'll begin the chapter by describing how to create our own particle systems. We'll then move on to wrap up the particle emitter into a `Spell` class, and write a `CastSpell()` function for the avatar to be able to actually cast spells.

The particle systems

First, we need a place to put all our snazzy effects. In your **Content Browser** tab, right-click on the **Game** root and create a new folder called **ParticleSystems**. Right-click on that new folder, and select **New Asset | Particle System**, as shown in the following screenshot:

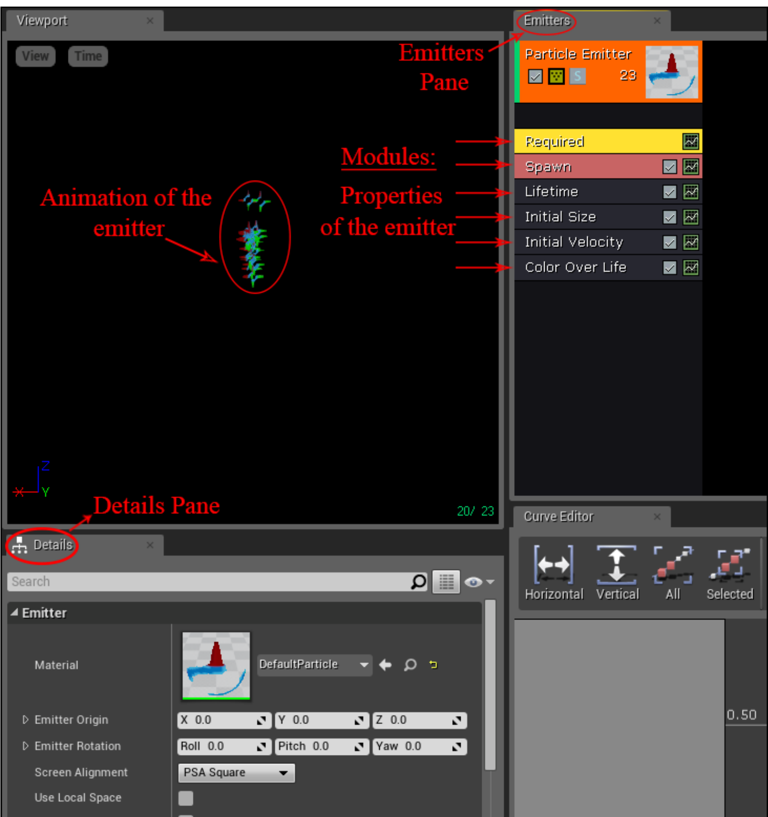


See this Unreal Engine 4 Particle Systems guide for information on how unreal particle emitters work: `https://www.youtube.com/watch?v=OXK2Xbd7D9w&index=1&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t`.

www.it-ebooks.info

Double-click on the **NewParticleSystem** icon that appears, as shown in the following screenshot:

You will be in Cascade, the particle editor. A description of the environment is shown in the following screenshot:



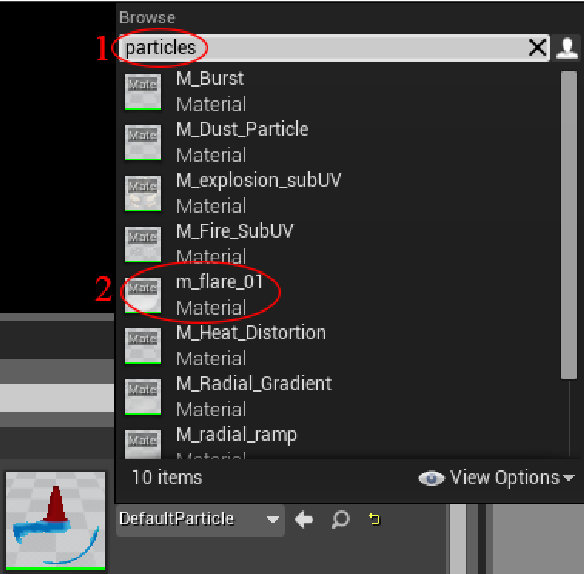
www.it-ebooks.info

There are several different panes here, each of which shows different information. They are as follows:

- At the top left is the **Viewport** pane. This shows you an animation of the current emitter as its currently working.
- At the right is the **Emitters** pane. Inside it, you can see a single object called **Particle Emitter** (you can have more than one emitter in your particle system, but we don't want that now). The listing of modules of **Particle Emitter** appears listed under it. From the preceding screenshot, we have the **Required**, **Spawn**, **Lifetime**, **Initial Size**, **Initial Velocity**, and **Color Over Life** modules.

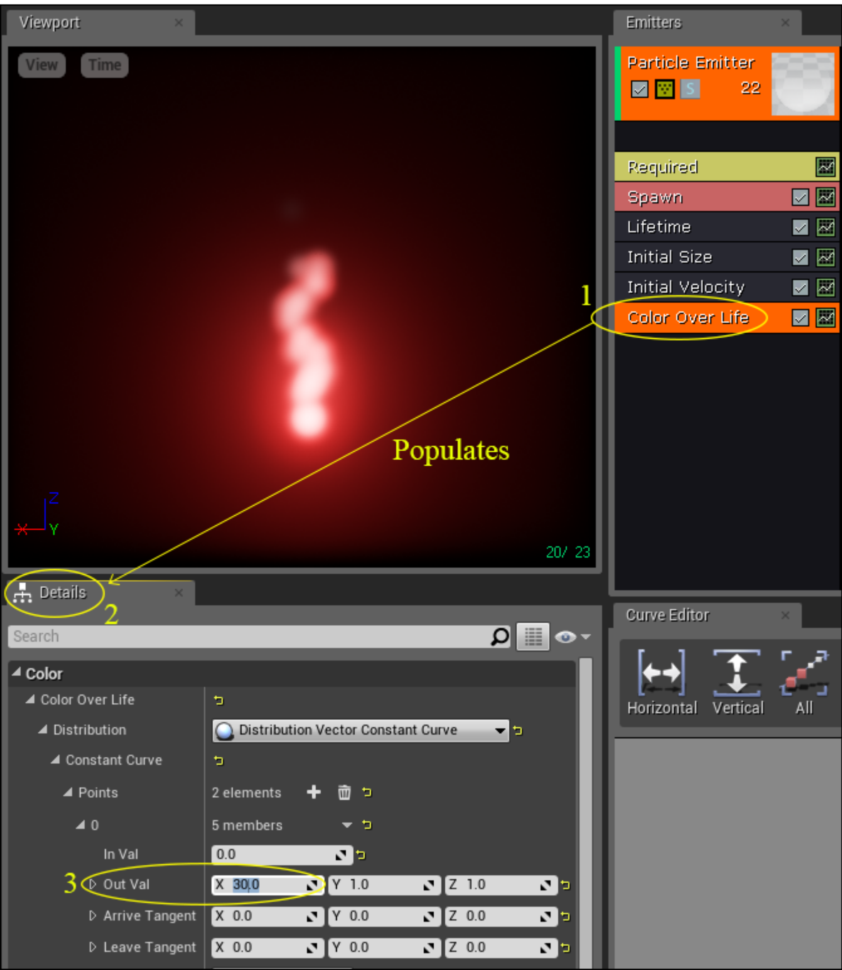
Changing particle properties

The default particle emitter emits crosshair-like shapes. We want to change that to something more interesting. Click on the yellow **Required** box under **Emitters** panel, then under **Material** in the **Details** panel, type `particles`. A list of all the available particle materials will pop up. Choose **m_flare_01** option to create our first particle system, as shown in the following screenshot:



www.it-ebooks.info

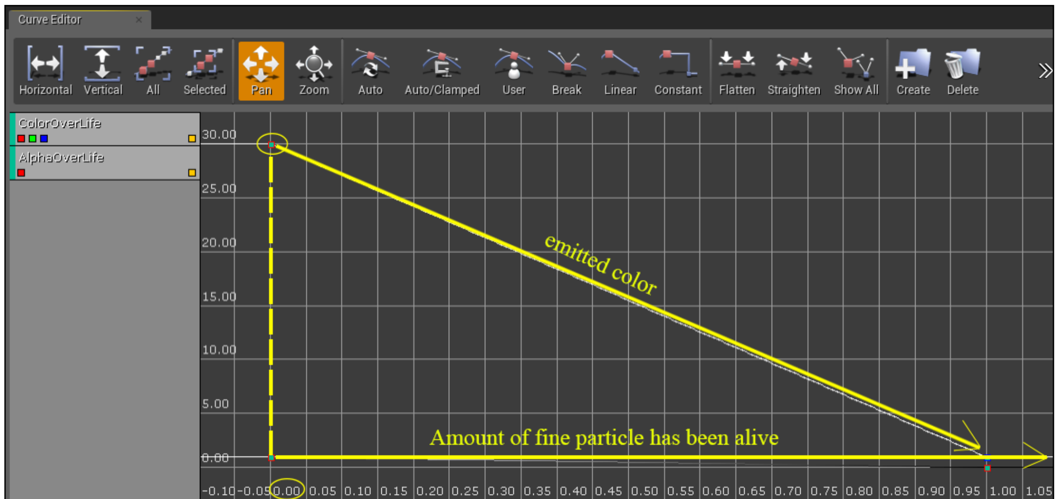
Now, let's change the behavior of the particle system. Click on the **Color Over Life** entry under the **Emitters** pane. The **Details** pane at the bottom shows the information about the different parameters, as shown in the following screenshot:



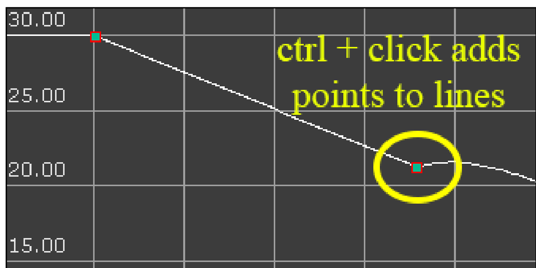
In the **Details** pane of **Color Over Life** entry, I increased **X**, but not **Y** and not **Z**. This gives the particle system a reddish glow. (**X** is red, **Y** is green, and **Z** is blue).

www.it-ebooks.info

Instead of editing the raw numbers, however, you can actually change the particle color more visually. If you click on the greenish zigzag button beside the **Color Over Life** entry, you will see the graph for **Color Over Life** displayed in the **Curve Editor** tab, as shown in the following screenshot:



We can now change the **Color Over Life** parameters. The graph in the **Curve Editor** tab displays the emitted color versus the amount of time the particle has been alive. You can adjust the values by dragging the points around. Pressing *Ctrl* + left mouse button adds a new point to a line:



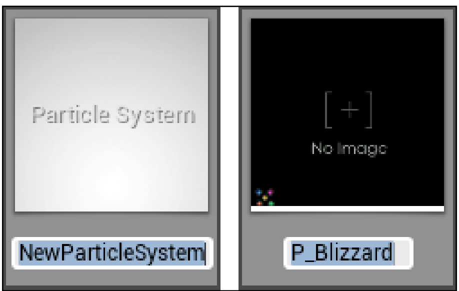
Ctrl + click adds points to lines.

You can play around with the particle emitter settings to create your own spell visualizations.

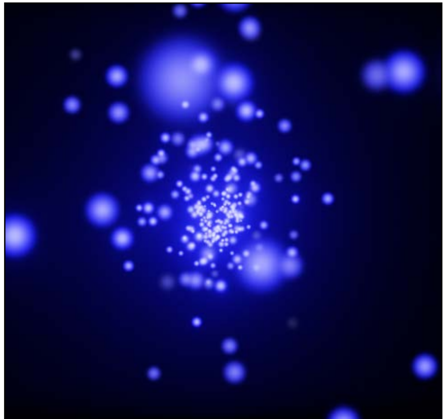
www.it-ebooks.info

Settings for the blizzard spell

At this point, we should rename our particle system, from **NewParticle System** to something more descriptive. Let's rename it **P_Blizzard**. You can rename your particle system by simply clicking on it and pressing *F2*.



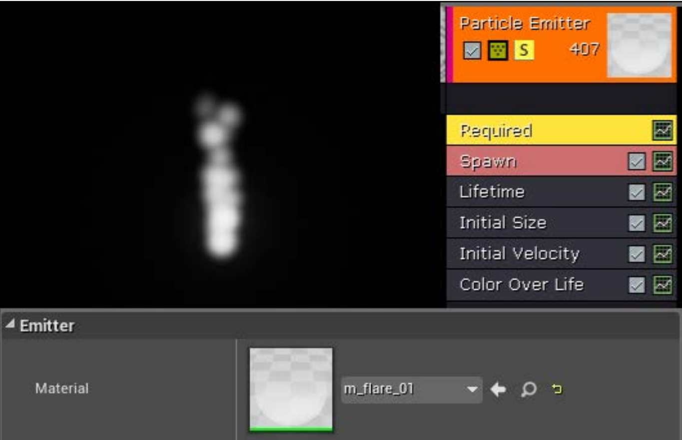
Press *F2* on an object in the Content Browser to rename it



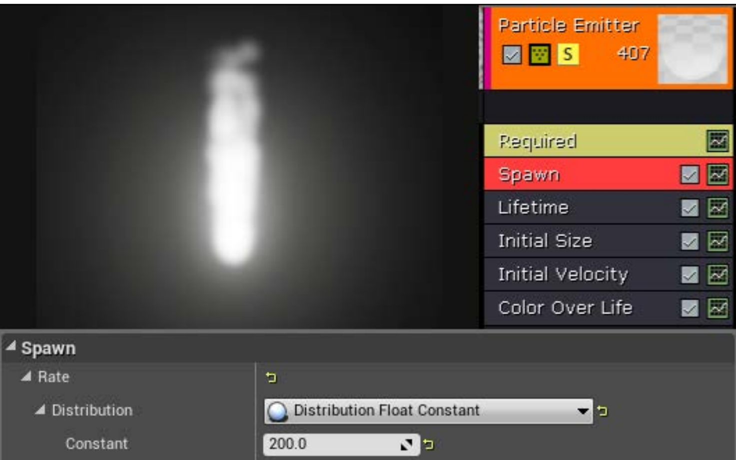
We will tweak some of the settings to get a blizzard particle effect spell. Perform the following steps:

1. Under the **Emitters** tab, click on the **Required** box. In the **Details** pane, change the **Emitter** material to **m_flare_01** as shown:

www.it-ebooks.info

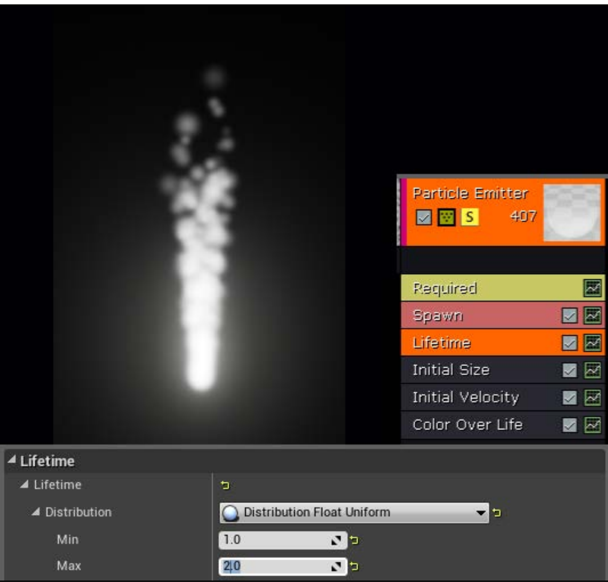


2. Under the **Spawn** module, change the spawn rate to 200. This increases the density of the visualization, as shown:

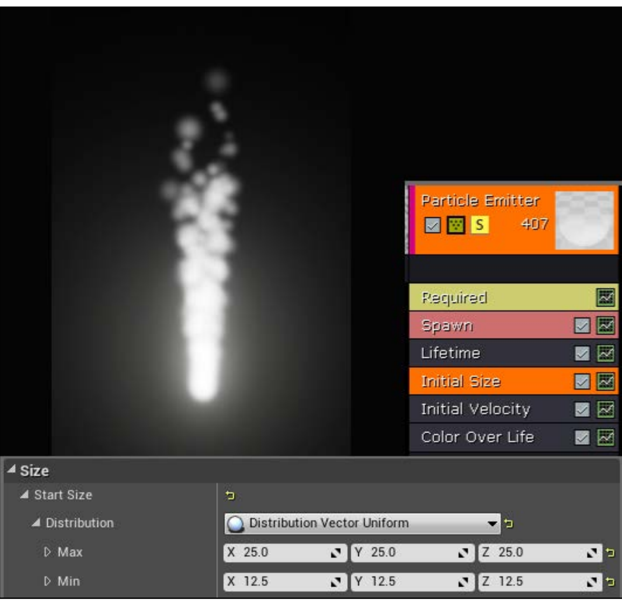


www.it-ebooks.info

3. Under the **Lifetime** module, increase the **Max** property from 1.0 to 2.0. This introduces some variation to the length of time a particle will live, with some of the emitted particles living longer than others.

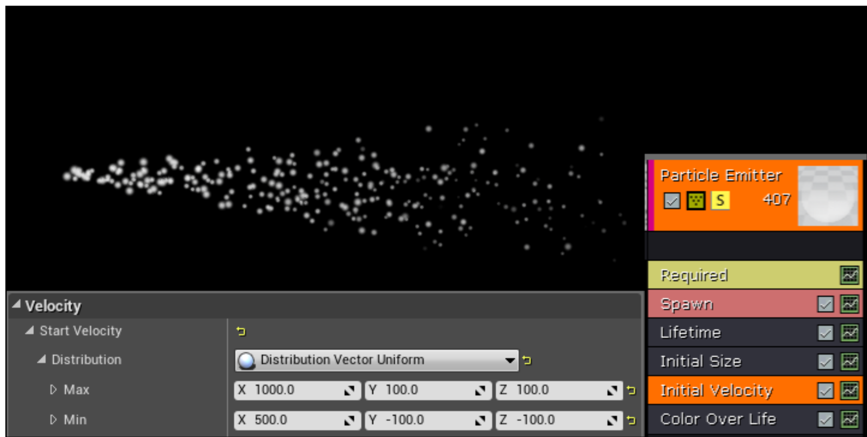


4. Under the **Initial Size** module, change the **Min** property size to 12.5 in **X**, **Y**, and **Z**:

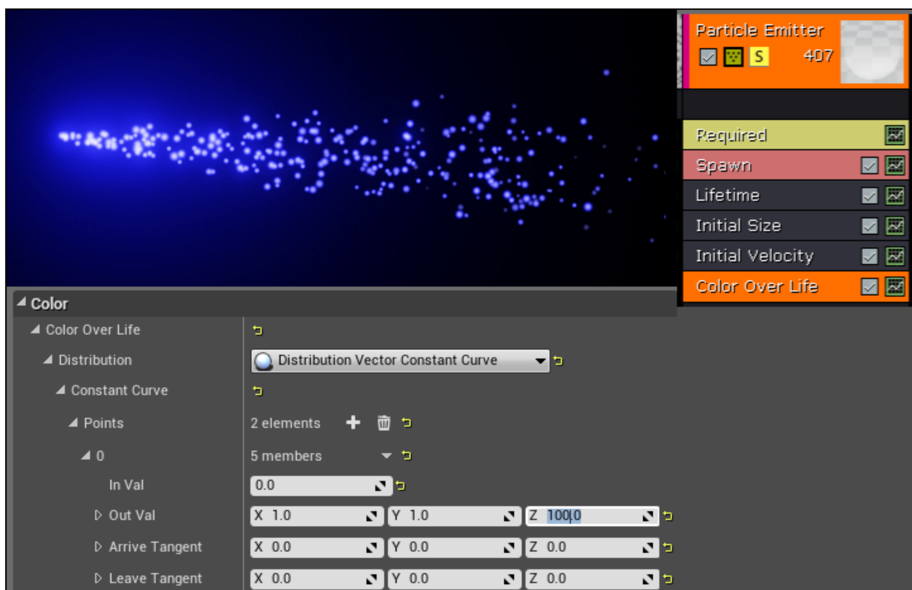


www.it-ebooks.info

5. Under the **Initial Velocity** module, change the **Min**/**Max** values to the values shown:



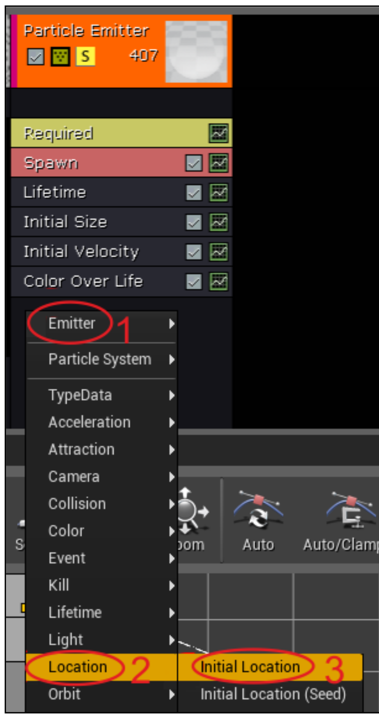
6. The reason we're having the blizzard blow in +X is because the player's forward direction starts out in +X. Since the spell will come from the player's hands, we want the spell to point in the same direction as the player.
7. Under the **Color Over Life** menu, change the blue (**Z**) value to 100.0. You will see an instant change to a blue glow:



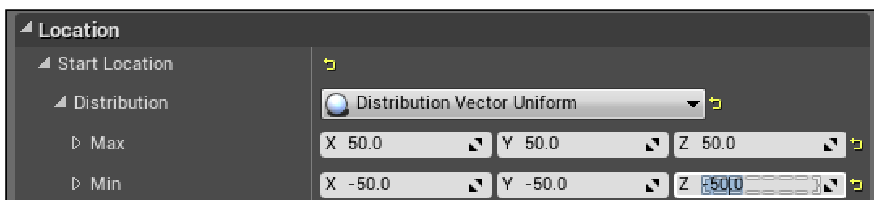
Now it's starting to look magical!

www.it-ebooks.info

8. Right-click on the blackish area below the **Color Over Life** module. Choose **Location** | **Initial Location**:



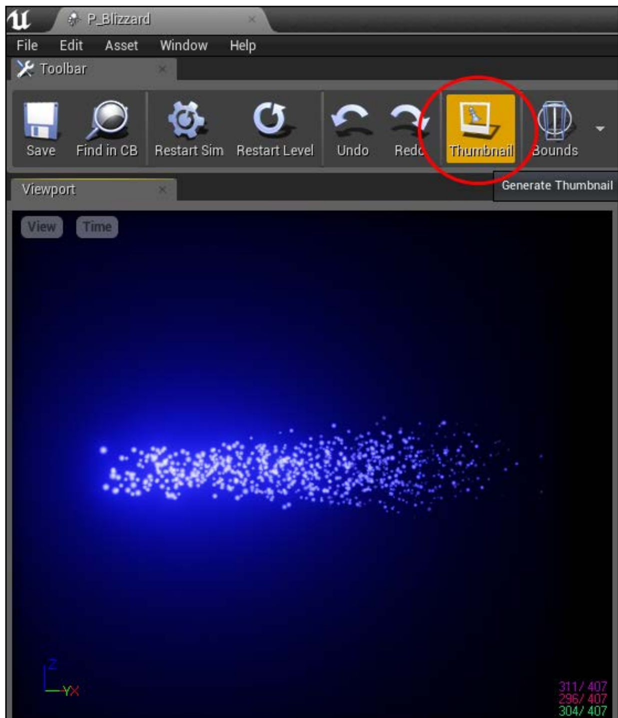
9. Enter values under **Start Location** | **Distribution** as shown below:



www.it-ebooks.info

10. You should have a blizzard that looks like this:

11. Move the camera to a position you like, then click on the **Thumbnail** option in the top menu bar. This will generate a thumbnail icon for your particle system in the **Content Browser** tab.



Clicking Thumbnail at the top menu bar will generate a mini icon for your particle system

www.it-ebooks.info

Spell class actor

The `Spell` class will ultimately do damage to all the monsters. Towards that end, we need to contain both a particle system and a bounding box inside the `Spell` class actor. When a `Spell` class is cast by the avatar, the `Spell` object will be instantiated into the level and start `Tick()` functioning. On every `Tick()` of the `Spell` object, any monster contained inside the spell's bounding volume will be affected by that `Spell`.

The `Spell` class should look something like the following code:

```
UCLASS()
class GOLDENEGG_API ASpell : public AActor
{
  GENERATED_UCLASS_BODY()

  // box defining volume of damage
  UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
  Spell)
  TSubobjectPtr<UBoxComponent> ProxBox;

  // the particle visualization of the spell
  UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
  Spell)
  TSubobjectPtr<UParticleSystemComponent> Particles;

  // How much damage the spell does per second
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
  float DamagePerSecond;

  // How long the spell lasts
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
  float Duration;

  // Length of time the spell has been alive in the level
  float TimeAlive;

  // The original caster of the spell (so player doesn't
  // hit self)
  AActor* Caster;

  // Parents this spell to a caster actor
  void SetCaster( AActor* caster );

  // Runs each frame. override the Tick function to deal damage
  // to anything in ProxBox each frame.
  virtual void Tick( float DeltaSeconds ) override;
};
```


www.it-ebooks.info

There are only three functions we need to worry about implementing, namely the `ASpell::ASpell()` constructor, the `ASpell::SetCaster()` function, and the `ASpell::Tick()` function.

Open the `Spell.cpp` file. Add a line to include the `Monster.h` file, so we can access the definition of `Monster` objects inside the `Spell.cpp` file, as shown in the following line of code:

```
#include "Monster.h"
```

First, the constructor, which sets up the spell and initializes all components is shown in the following code:

```
ASpell::ASpell(const class FPostConstructInitializeProperties&
  PCIP) : Super(PCIP)
{
  ProxBox = PCIP.CreateDefaultSubobject<UBoxComponent>(this,
  TEXT("ProxBox"));
  Particles = PCIP.CreateDefaultSubobject<UParticleSystemComponent>(t
his,
  TEXT("ParticleSystem"));

  // The Particles are the root component, and the ProxBox
  // is a child of the Particle system.
  // If it were the other way around, scaling the ProxBox
  // would also scale the Particles, which we don't want
  RootComponent = Particles;
  ProxBox->AttachTo( RootComponent );

  Duration = 3;
  DamagePerSecond = 1;
  TimeAlive = 0;

  PrimaryActorTick.bCanEverTick = true;//required for spells to
  // tick!
}
```

Of particular importance is the last line here, `PrimaryActorTick.bCanEverTick = true`. If you don't set that, your `Spell` objects won't ever have `Tick()` called.

www.it-ebooks.info

Next, we have the `SetCaster()` method. This is called so that the person who casts the spell is known to the `Spell` object. We can ensure that the caster can't hurt himself with his own spells by using the following code:

```
void ASpell::SetCaster( AActor *caster )
{
  Caster = caster;
  AttachRootComponentTo( caster->GetRootComponent() );
}
```

Finally, we have the `ASpell::Tick()` method, which actually deals damage to all contained actors, as shown in the following code:

```
void ASpell::Tick( float DeltaSeconds )
{
  Super::Tick( DeltaSeconds );

  // search the proxbox for all actors in the volume.
  TArray<AActor*> actors;
  ProxBox->GetOverlappingActors( actors );

  // damage each actor the box overlaps
  for( int c = 0; c < actors.Num(); c++ )
  {
    // don't damage the spell caster
    if( actors[ c ] != Caster )
    {
      // Only apply the damage if the box is overlapping
      // the actors ROOT component.
      // This way damage doesn't get applied for simply
      // overlapping the SightSphere of a monster
      AMonster *monster = Cast<AMonster>( actors[c] );

      if( monster && ProxBox->IsOverlappingComponent( monster-
      >CapsuleComponent ) )
      {
        monster->TakeDamage( DamagePerSecond*DeltaSeconds,
        FDamageEvent(), 0, this );
      }

      // to damage other class types, try a checked cast
      // here..
    }
  }
```


www.it-ebooks.info

```
    TimeAlive += DeltaSeconds;
    if( TimeAlive > Duration )
    {
      Destroy();
    }
  }
```

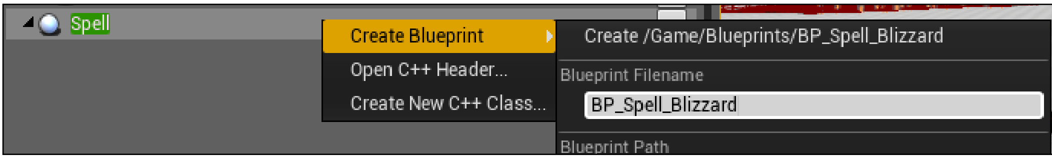
The `ASpell::Tick()` function does a number of things, as follows:

- Gets all actors overlapping `ProxBox`. Any actor that is not the caster gets damaged if the component overlapped is the root component of that object. The reason we have to check for overlapping with the root component is because if we don't, the spell might overlap the monster's `SightSphere`, which means we will get hits from very far away, which we don't want.
- Notices that if we had another class of thing that should get damaged, we would have to attempt a cast to each object type specifically. Each class type might have a different type of bounding volume that should be collided with, other types might not even have `CapsuleComponent` (they might have `ProxBox` or `ProxSphere`).
- Increases the amount of time the spell has been alive for. If the spell exceeds the duration it is allotted to be cast for, it is removed from the level.

Now, let's focus on how the player can acquire spells, by creating an individual `PickupItem` for each spell object that the player can pick up.

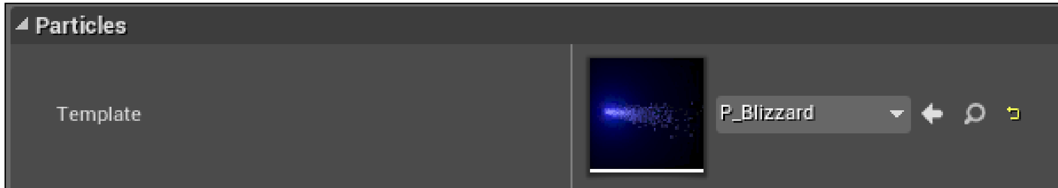
Blueprinting our spells

Compile and run your C++ project with the `Spell` class that we just added. We need to create blueprints for each of the spells we want to be able to cast. In the **Class Viewer** tab, start to type `Spell`, and you should see your `Spell` class appear. Right-click on **Spell**, and create a blueprint called **BP_Spell_Blizzard**, and then double-click to open it, as shown in the following screenshot:

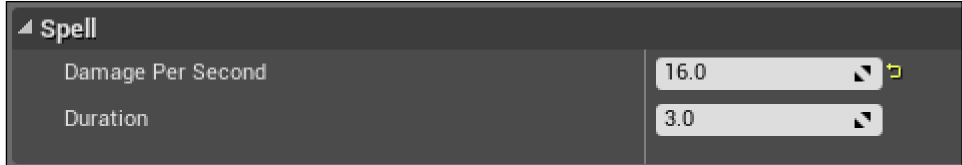


www.it-ebooks.info

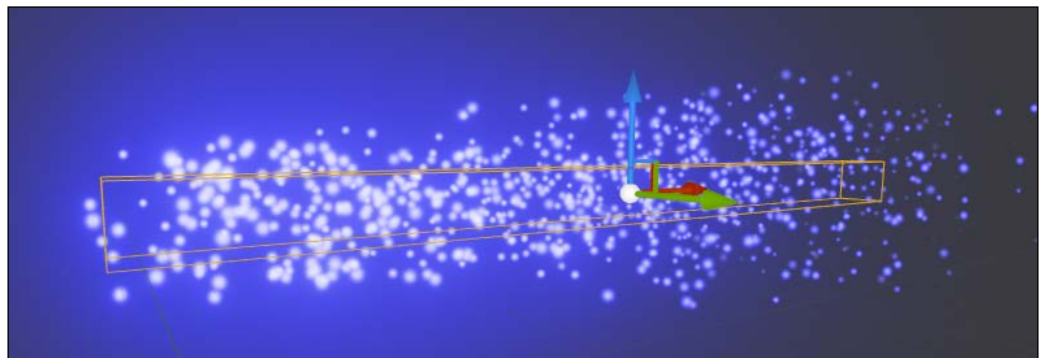
Inside the spell's properties, choose the **P_Blizzard** spell for the particle emitter, as shown in the following screenshot:



Scroll down until you reach the **Spell** category, and update the **Damage Per Second** and **Duration** parameters to values you like. Here, the blizzard spell will last 3.0 seconds, and do 16.0 damage total per second. After three seconds, the blizzard will disappear.



After you have configured the **Default** properties, switch over to the **Components** tab to make some further modifications. Click on and change the shape of `ProxBox` so that its shape makes sense. The box should wrap the most intense part of the particle system, but don't get carried away in expanding its size. The `ProxBox` object shouldn't be too big, because then your blizzard spell would affect things that aren't even being touched by the blizzard. As shown in the following screenshot, a couple of outliers are ok.

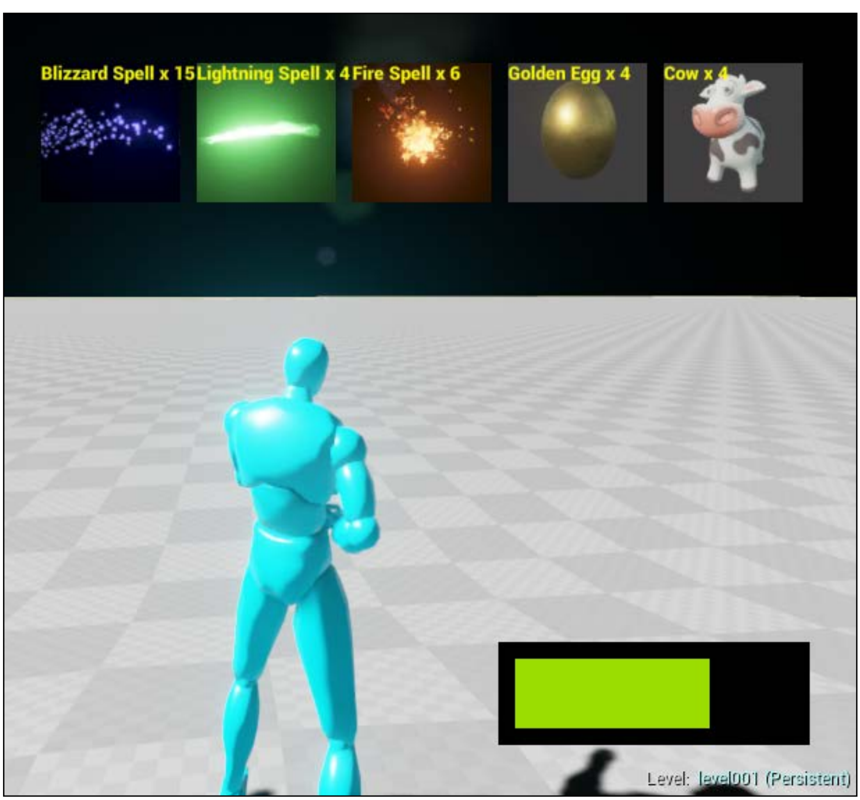


Your blizzard spell is now blueprinted and ready to be used by the player.

www.it-ebooks.info

Picking up spells

Recall that we previously programmed our inventory to display the number of pickup items the player has when the user presses *I*. We want to do more than that, however.



Items displayed when the user presses *I*

To allow the player to pick up spells, we'll modify the `PickupItem` class to include a slot for a blueprint of the spell the player casts by using the following code:

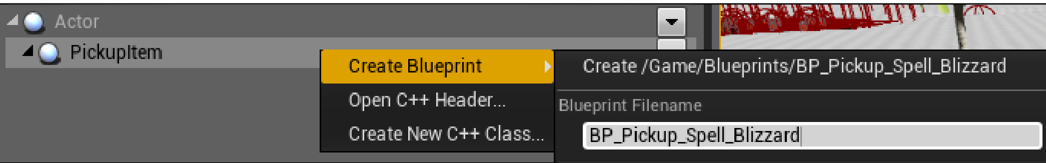
```
// inside class APickupItem:
// If this item casts a spell when used, set it here
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
UClass* Spell;
```

Once you've added the `UClass* Spell` property to the `APickupItem` class, recompile and rerun your C++ project. Now, you can proceed to make blueprints of `PickupItem` instances for your `Spell` objects.

www.it-ebooks.info

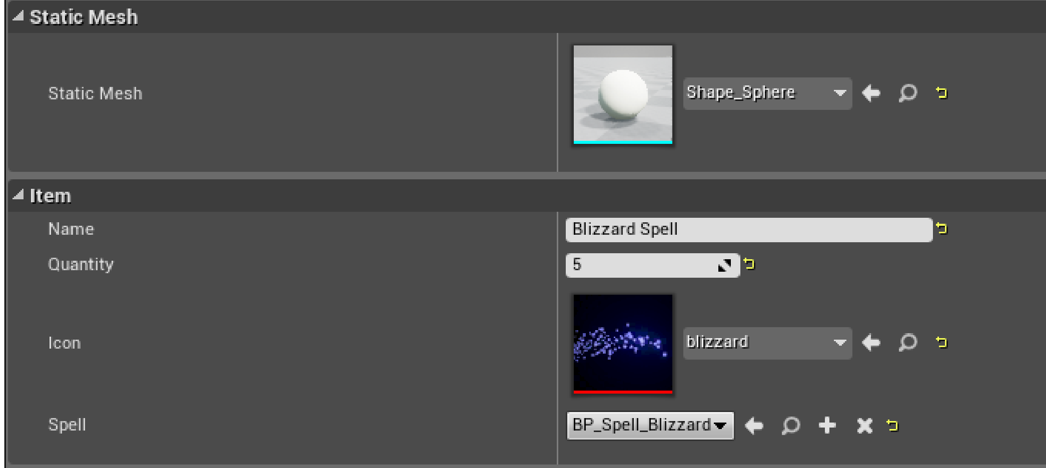
Creating blueprints for PickupItems that cast spells

Create a **PickupItem** blueprint called **BP_Pickup_Spell_Blizzard**. Double-click on it to edit its properties, as shown in the following screenshot:



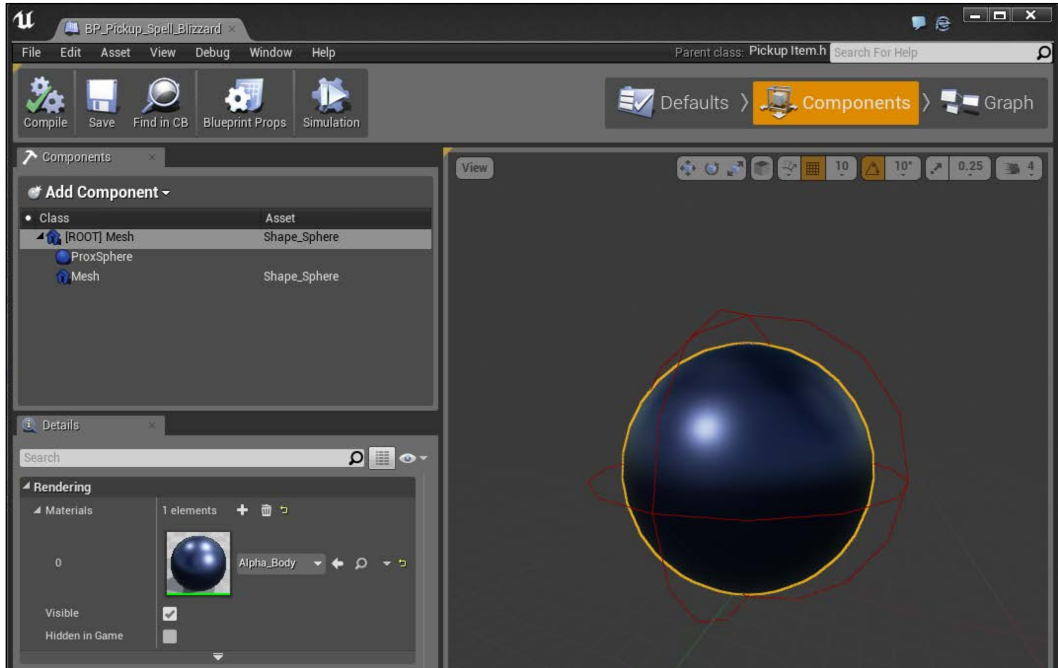
I set the blizzard item's pickup properties as follows:

The name of the item is **Blizzard Spell**, and five are in each package. I took a screenshot of the blizzard particle system and imported it to the project, so the **Icon** is selected as that image. Under spell, I selected **BP_Spell_Blizzard** as the name of the spell to be cast (not **BP_Pickup_Spell_Blizzard**), as shown in the following screenshot:

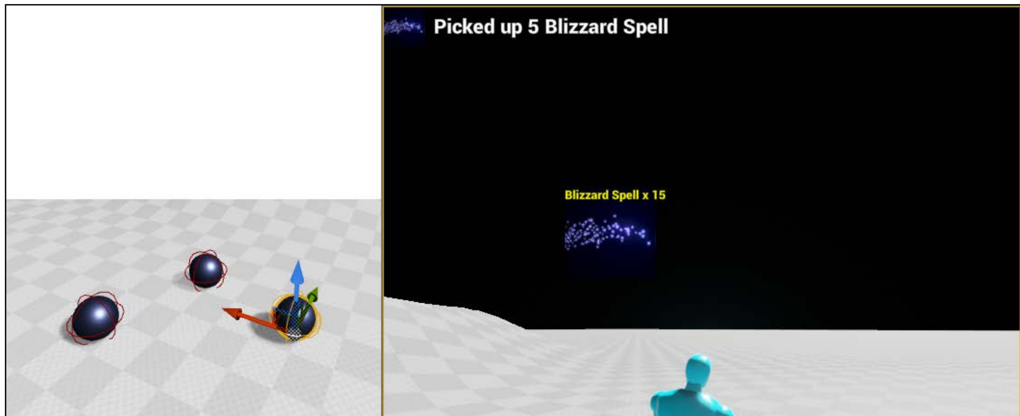


www.it-ebooks.info

I selected a blue sphere for the `Mesh` class of the `PickupItem` class. For **Icon**, I took a screenshot of the blizzard spell in the particle viewer preview, saved it to disk, and imported that image to the project (see the images folder in the **Content Browser** tab of the sample project).



Place a few of these `PickupItem` in your level. If we pick them up, we will have some blizzard spells in our inventory.



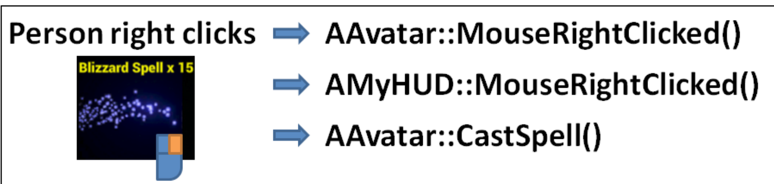
Left: Blizzard spell pickup items in game world. Right: Blizzard spell pickup item in inventory.

www.it-ebooks.info

Now we need to activate the blizzard. Since we already attached the left mouse click in *Chapter 10, Inventory System and Pickup Items* to dragging the icons around, let's attach the right mouse click to casting the spell.

Attaching right mouse click to cast spell

The right mouse click will have to go through quite a few function calls before calling the avatar's `CastSpell` method. The call graph would look something like the following screenshot:



A few things happen between right click and spell cast. They are as follows:

- As we saw before, all user mouse and keyboard interactions are routed through the `Avatar` object. When the `Avatar` object detects a right-click, it will pass the click event to `HUD` through `AAvatar::MouseRightClicked()`.
- Recall from *Chapter 10, Inventory System and Pickup Items* where we used a `struct Widget` class to keep track of the items the player had picked up. `struct Widget` only had three members:

  ```
  struct Widget
  {
    Icon icon;
    FVector2D pos, size;
    ///.. and some member functions
  };
  ```

  We will need to add an extra property for `struct Widget` class to remember the spell it casts.

  The `HUD` will determine if the click event was inside `Widget` in `AMyHUD::MouseRightClicked()`.

- If the click was on the `Widget` that casts a spell, the `HUD` then calls the avatar back with the request to cast that spell, by calling `AAvatar::CastSpell()`.

www.it-ebooks.info

Writing the avatar's CastSpell function

We will implement the preceding call graph in reverse. We will start by writing the function that actually casts spells in the game, `AAvatar::CastSpell()`, as shown in the following code:

```
void AAvatar::CastSpell( UClass* bpSpell )
{
  // instantiate the spell and attach to character
  ASpell *spell = GetWorld()->SpawnActor<ASpell>(bpSpell,
  FVector(0), FRotator(0) );

  if( spell )
  {
    spell->SetCaster( this );
  }
  else
  {
    GEngine->AddOnScreenDebugMessage( 1, 5.f, FColor::Yellow,
    FString("can't cast ") + bpSpell->GetName() );
  }
}
```

You might find that actually calling a spell is remarkably simple. There are two basic steps to casting the spell:

- Instantiate the spell object using the world object's `SpawnActor` function
- Attach it to the avatar

Once the `Spell` object is instantiated, its `Tick()` function will run each frame when that spell is in the level. On each `Tick()`, the `Spell` object will automatically feel out monsters within the level and damage them. A lot happens with each line of code mentioned previously, so let's discuss each line separately.

Instantiating the spell – GetWorld()->SpawnActor()

To create the `Spell` object from the blueprint, we need to call the `SpawnActor()` function from the `World` object. The `SpawnActor()` function can take any blueprint and instantiate it within the level. Fortunately, the `Avatar` object (and indeed any `Actor` object) can get a handle to the `World` object at any time by simply calling the `GetWorld()` member function.

The line of code that brings the `Spell` object into the level is as follows:

```
ASpell *spell = GetWorld()->SpawnActor<ASpell>( bpSpell,
  FVector(0), FRotator(0) );
```


www.it-ebooks.info

There are a couple of things to note about the preceding line of code:

- `bpSpell` must be the blueprint of a `Spell` object to create. The `<ASpell>` object in angle brackets indicates that expectation.
- The new `Spell` object starts out at the origin (0, 0, 0), and with no additional rotation applied to it. This is because we will attach the `Spell` object to the `Avatar` object, which will supply translation and direction components for the `Spell` object.

if(spell)

We always test if the call to `SpawnActor<ASpell>()` succeeds by checking `if( spell )`. If the blueprint passed to the `CastSpell` object is not actually a blueprint based on the `ASpell` class, then the `SpawnActor()` function returns a `NULL` pointer instead of a `Spell` object. If that happens, we print an error message to the screen indicating that something went wrong during spell casting.

spell->SetCaster(this)

When instantiating, if the spell does succeed, we attach the spell to the `Avatar` object by calling `spell->SetCaster( this )`. Remember, in the context of programming within the `Avatar` class, the `this` method is a reference to the `Avatar` object.

Now, how do we actually connect spell casting from UI inputs, to call `AAvatar::CastSpell()` function in the first place? We need to do some `HUD` programming again.

Writing AMyHUD::MouseRightClicked()

The spell cast commands will ultimately come from the HUD. We need to write a C++ function that will walk through all the HUD widgets and test to see if a click is on any one of them. If the click is on a `widget` object, then that `widget` object should respond by casting its spell, if it has one assigned.

We have to extend our `Widget` object to have a variable to hold the blueprint of the spell to cast. Add a member to your `struct Widget` object by using the following code:

```
struct Widget
{
  Icon icon;
  // bpSpell is the blueprint of the spell this widget casts
  UClass *bpSpell;
```


www.it-ebooks.info

```
  FVector2D pos, size;
  Widget(Icon iicon, UClass *iClassName)
}
```

Now, recall that our `PickupItem` had the blueprint of the spell it casts attached to it previously. However, when the `PickupItem` class is picked up from the level by the player, then the `PickupItem` class is destroyed.

```
// From APickupItem::Prox_Implementation():
avatar->Pickup( this ); // give this item to the avatar
// delete the pickup item from the level once it is picked up
Destroy();
```

So, we need to retain the information of what spell each `PickupItem` casts. We can do that when that `PickupItem` is first picked up.

Inside the `AAvatar` class, add an extra map to remember the blueprint of the spell that an item casts, by item name:

```
// Put this in Avatar.h
TMap<FString, UClass*> Spells;
```

Now in `AAvatar::Pickup()`, remember the class of spell the `PickupItem` class instantiates with the following line of code:

```
// the spell associated with the item
Spells.Add(item->Name, item->Spell);
```

Now, in `AAvatar::ToggleInventory()`, we can have the `Widget` object that displays on the screen. Remember what spell it is supposed to cast by looking up the `Spells` map.

Find the line where we create the widget, and just under it, add assignment of the `bpSpell` objects that the `Widget` casts:

```
// In AAvatar::ToggleInventory()
Widget w( Icon( fs, tex ) );
w.bpSpell = Spells[it->Key];
```

Add the following function to `AMyHUD`, which we will set to run whenever the right mouse button is clicked on the icon:

```
void AMyHUD::MouseRightClicked()
{
  FVector2D mouse;
  APlayerController *PController = GetWorld()-
  >GetFirstPlayerController();
  PController->GetMousePosition( mouse.X, mouse.Y );
```

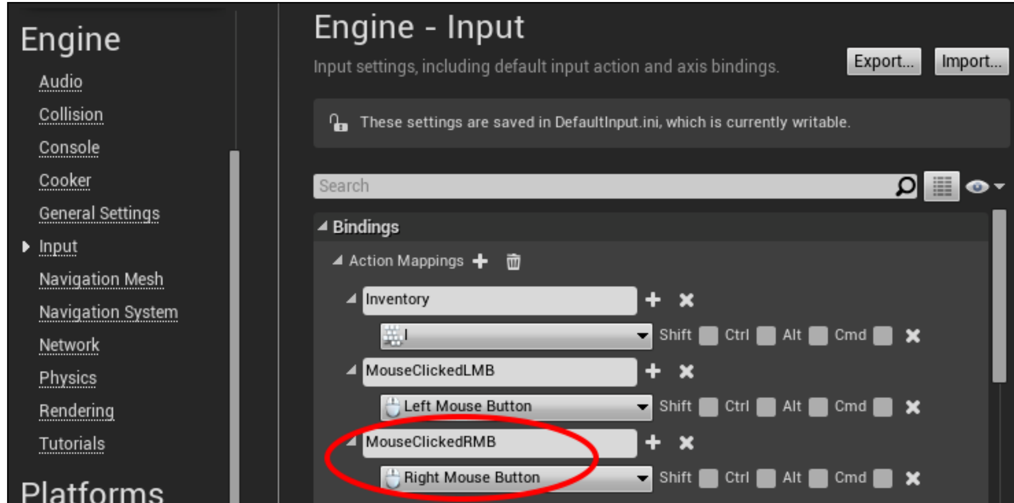

www.it-ebooks.info

```
    for( int c = 0; c < widgets.Num(); c++ )
    {
      if( widgets[c].hit( mouse ) )
      {
        AAvatar *avatar = Cast<AAvatar>(
        UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
        if( widgets[c].spellName )
          avatar->CastSpell( widgets[c].spellName );
      }
    }
  }
```

This is very similar to our left mouse click function. We simply check the click position against all the widgets. If any `Widget` was hit by the right-click, and that `Widget` has a `Spell` object associated with it, then a spell will be cast by calling the avatar's `CastSpell()` method.

Activating right mouse button clicks

To connect this HUD function to run, we need to attach an event handler to the mouse right-click. We can do so by going to **Settings** | **Project Settings**, and from the dialog that pops up, adding an **Input** option for **Right Mouse Button**, as shown in the following screenshot:



www.it-ebooks.info

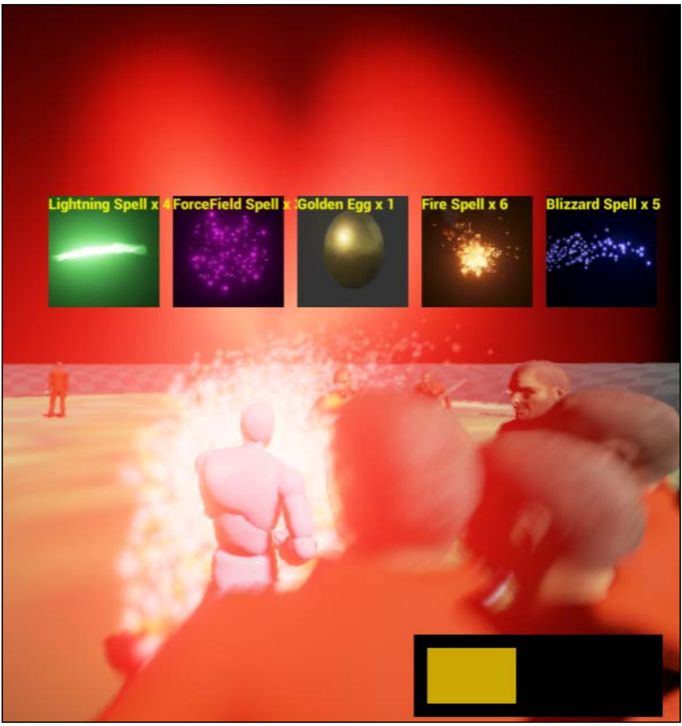
Declare a function in `Avatar.h`/`Avatar.cpp` called `MouseRightClicked()` with the following code:

```
void AAvatar::MouseRightClicked()
{
  if( inventoryShowing )
  {
    APlayerController* PController = GetWorld()-
    >GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->MouseRightClicked();
  }
}
```

Then, in `AAvatar::SetupPlayerInputComponent()`, we should attach `MouseClickedRMB` event to that `MouseRightClicked()` function:

```
// In AAvatar::SetupPlayerInputComponent():
InputComponent->BindAction( "MouseClickedRMB", IE_Pressed, this,
  &AAvatar::MouseRightClicked );
```

We have finally hooked up spell casting. Try it out, the gameplay is pretty cool, as shown in the following screenshot:



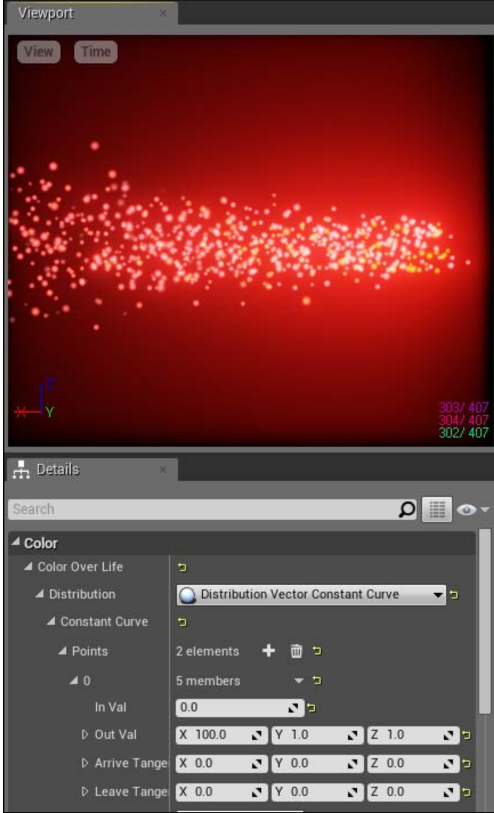
www.it-ebooks.info

Creating other spells

By playing around with particle systems, you can create a variety of different spells that do different effects.

The fire spell

You can easily create a fire variant of our blizzard spell by changing the color of the particle system to red:



The out val of the color changed to red

www.it-ebooks.info

Exercises

Try the following exercises:

1. **Lightning spell**: Create a lightning spell by using the beam particle. Follow Zak's tutorial for an example of how beams are created and shot in a direction, at `https://www.youtube.com/watch?v=ywd3lFOuMV8&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t&index=7`.
2. **Forcefield spell**: A forcefield will deflect attacks. It is essential for any player. Suggested implementation: Derive a subclass of `ASpell` called `ASpellForceField`. Add a bounding sphere to the class, and use that in the `ASpellForceField::Tick()` function to push the monsters out.

What's next? I would highly recommend that you expand on our little game here. Here are some ideas for expansion:

- Create more environments, expand the terrain, add in more houses and buildings
- Add quests that come from NPCs
- Define more melee weapons such as, swords
- Define armor for the player, such as shields
- Add shops that sell weapons to the player
- Add more monster types
- Implement loot drops for monsters

You have literally thousands of hours of work ahead of you. If you happen to be a solo programmer, form working relationships with other souls. You cannot survive in the game marketplace on your own.

It's dangerous to go alone—Take a friend.

Summary

This concludes this chapter. You have come a long way. From not knowing anything about C++ programming at all, to hopefully being able to string together a basic game program in UE4.

www.it-ebooks.info

www.it-ebooks.info

Index

Symbols

<cmath> library function
sqrt() function 83
.cpp files 97, 129-131
.h files 97, 129-131
& operator 39, 40
== operator 47

A

actors
about 143
adding, to scene 156
versus pawns 143
AMyHUD::MouseRightClicked()
creating 310-312
And (&&) operator 52
APickupItem class
FString variable 217
int32 variable 217
USphereComponent variable 217
UStaticMeshComponent variable 217
UTexture2D variable 217
APuzzleBlock class 58
arrays
about 137
array syntax 137, 138
exercise 138
solutions, of exercise 139
Artificial Intelligence (AI) 143
attack animation, triggering
about 263
animation blueprint, modifying for Mixamo Adam 269-274
blueprint basics 264-268
code, for swinging sword 274-276

B

basic monster intelligence
about 243
discrete nature, of monster motion 246-248
monster, moving 243-246
Monster SightSphere 248-250
blizzard spell
settings 294-299
BlockClicked function 58
branching 46
bullet physics 279, 280
bullets
adding, to monster class 281-285

C

C++
math 31
cast spell
right mouse click, attaching to 308
CastSpell function
writing 309
C++ code, for controlling game character
controller inputs, setting up 172-174
player, making instance of Avatar class 169-171
writing 169
yaw and pitch, setting 176, 177
cin command
about 41
using 139

class
about 113
putting into headers 127-129
versus struct 113
class inheritance
about 118, 123
derived classes 118-122
is-a relationship 123, 124
protected variables 124
purely virtual function 125
syntax 122
virtual function 124
comparison operators
used, for testing inequalities 50
compiling 23
constructors 117, 118
const variables 97
control flow statements 46
controller inputs
exercise 174
setting up 172-174
solution, of exercise 174, 175
C++ primitive type
double 141
float 141
int 141
long long 141
C++ program
ASCII art maze example 23
compiling 23
creating 17-21
errors, handling 21, 22
flow, controlling 46
Microsoft Visual C++, using on Windows 10-13
scripting 23
semicolons 21
setting up 9
source code, building 23
variables 26
warnings 22
XCode, using on Mac 14-17
C++ STL map
about 205, 206
element, finding 206
exercise 206
solution, of exercise 207
C++ STL set
about 203
element, finding 204
exercise 204
solution, of exercise 204
C++ STL versions
about 202
C++ STL map 205, 206
C++ STL set 203
C-style arrays 139-141

D

delete[] array 140
destructors 117, 118
do/while loop 75
dynamic memory allocations
about 133-135
delete keyword 135
memory leaks 136

E

EASTL
URL 202
else statements
coding 49
encapsulation
about 111
reasons 111
event, triggering
exercises 190
solutions, of exercises 191, 192
extern variables 100

F

fire spell 314
for loop
about 76
exercises 77
solutions, of exercises 78
forward declaration, players backpack 210, 211

www.it-ebooks.info

function prototypes 97
functions
<cmath> library function 83
about 81, 82
advantages 83
custom function, writing 84, 85
exercise 88
main() function 86
printRoad() function 85
sample program trace 85-87
solution, of exercise 88
values, returning 89-91
with arguments 88, 89
functions, returning values
exercises 91
solutions, of exercises 91, 92

G

GENERATED_UCLASS_BODY() macro 159
get() operation 116, 117
getters 113, 114
GetWorld()->SpawnActor()
instantiating 309, 310
Git 16
global variables 92, 93
GNU Compiler Collection (GCC) 23

H

hardcoding 161
Hard Disk Drives (HDDs) 26
HUD 181
HUD::DrawTexture()
exercise 227
using 224-227

I

if(spell) function 310
if statement
about 71
coding 47, 48
Integrated Development Environment (IDE) project 100
inventory item clicks
detecting 227, 228
elements, dragging 228-231
exercises 231
is-a relationship 123
iterating ways, TArray
about 196
iterators 197, 198
vanilla for loop and square brackets notation 196

K

knockback
adding, to player 285, 286

L

landscape
about 234
creating 234-236
sculpting 237, 238
level
collision detection, adding for objects editor 154-156
collision volumes 154
creating 151, 152
light sources, adding 152, 153
local variables 93
logical operators
about 51
And (&&) operator 52
Not (!) operator 51
Or (| |) operator 53
using 51

M

macros
about 100
const variables, using 101
macros, with arguments
about 101
inline functions, using 102
main.cpp file 100

www.it-ebooks.info

malloc() function
  using 141
math, in C++
  about 31, 32
  exercises 32
melee attacks
  about 251
  melee weapon, defining 251
member functions
  about 107
  encapsulation 111
  exercises 110
  invoking 109, 110
  public data members 112
  solutions, of exercises 110, 111
  strings, using as objects 108
  this keyword 107
memory
  about 25
  numbers 28, 29
memory access violation 138
memory leak 136
Microsoft Visual C++
  using, on Windows 10-13
monster attacks, on player
  about 250
  knockback, adding to player 285, 286
  melee attacks 251
  projectile attack 277-279
  ranged attack 277-279
  sockets 257
monster class
  bullets, adding to 281-285
monsters
  programming 238-243
Monster SightSphere 248-250
multiple inheritance
  about 125, 126
  private inheritance 126

N

new[] array 140
non-player character entities
  creating 177-180
Not (!) operator 51
null pointer 40

O

object types
  about 34, 35
  Player object exercise 36
Or (| |) operator 53

P

particle properties
  modifying 291-293
particle systems 289-291
pawns 143
PickupItem base class
  about 217-219
  root component 220, 221
PickupItem blueprint
  creating 306, 307
PickupItem.h 217
player
  knockback, adding to 285, 286
player entity
  blueprint, creating from
    C++ class 161-168
  creating 156
  free models, downloading 159, 160
  inheriting, from UE4 GameFramework
    classes 156, 158
  mesh, loading 161
  model, associating with Avatar class 159
player inventory
  drawing 223
  HUD::DrawTexture(), using 224-226
  inventory item clicks, detecting 227, 228
player's backpack declaration
  about 209
  action mapping, attaching to key 216
  assets, importing 211-215
  forward declaration 210, 211
pointers
  about 37, 38
  uses 38, 39

www.it-ebooks.info

primitive types 33, 34
printf() function 41
program flow, C++
  == operator 47
  controlling 46
  else statements, coding 49
  if statements, coding 47, 48
  inequalities, testing with comparison operators 50
projectile attacks 277-279
prototypes.h file 98

Q

quote, displaying from NPC dialog box
  about 181
  event, triggering near NPC 187, 188
  messages, displaying on HUD 181-184
  NPC, making display to HUD 189
  TArray<Message>, using 184-186

R

RAM 25
ranged attacks 277-279
Rigging 101 class
  URL 168
right mouse button clicks
  activating 312, 313
  attaching, to cast spell 308
root component, PickupItem base class
  about 220, 221
  avatar, obtaining 222
  HUD, obtaining 222, 223
  player controller, obtaining 222
RootComponent property 220

S

scene
  actor, adding 156
scope of variable
  about 94, 95
  arg() 94
  fx 95
  g_int 94
  main() 95
scripting languages 23
s.c_str() function 42
set() operation 116, 117
setters 115
sockets
  about 257
  attack animation, triggering 263
  player, equipping with sword 261, 262
  skeletal mesh socket, creating in monster's hand 258, 259
  sword, attaching to model 260, 261
Solid-state Drives (SSDs) 26
source control management tools (scm) 16
Spell class actor 300-303
spells
  about 287
  blueprinting 303, 304
  creating 314
  instantiating 309, 310
  picking up 305
spell->SetCaster(this)
  calling 310
sqrt() function 83, 89
static local variables 96
struct 113
struct objects 106

T

TArray<int> variable 194-196
TArray<Message>
  exercise 186
  solution, of exercise 187
  using 184, 185
TArray<T>
  about 194
  iterating 196
  uses 195, 196
TMap<T, S>
  about 200
  items list 201
  iterating 201, 202

www.it-ebooks.info

TSet<T>
about 199
finding 200
interating 199
intersecting 200
unioning 200

U

UCLASS() macro 159
UE4 blueprint 264
UE4 editor
about 147
editor controls 147
objects, adding to scene 148-150
play mode controls 148
URL 145
UE4 project
output, debugging 194
TArray<T> 194
TMap<T, S> 200
TSet<T> 199
world, creating 144-146
Unreal Engine example
about 53-60
code, branching 61
else if statement 61, 62
exercise 60
exercise, else if statement 62, 63
exercise, switch statement 67, 68
looping with 78, 79
switch statement 64, 65
switch, versus if 66, 67
Unreal Engine 4 Particle Systems
URL, for video 289

V

variables
.cpp file 97, 98
.h file 97, 98
& operator 39, 40
about 26, 92
cin 41
const variables 97
declaring 26
extern variables 100
funcs.cpp file 99
function prototypes 97
generalized variable syntax 33
global variables 92, 93
local variables 93
main.cpp file 100
null pointer 40
object types 34, 35
pointers 37
primitive types 33, 34
printf() function 41
prototypes.h files 98
reading 27
scope of variable 94, 95
static local variables 96
using 29-31
writing 28
Visual Studio
URL 10

W

weapon, melee attacks
blueprint, creating 256, 257
coding, in C++ 251-253
defining 251
sword, downloading 254, 255
while loop
about 71-73
exercises 74
infinite loops 73
solutions, of exercises 74

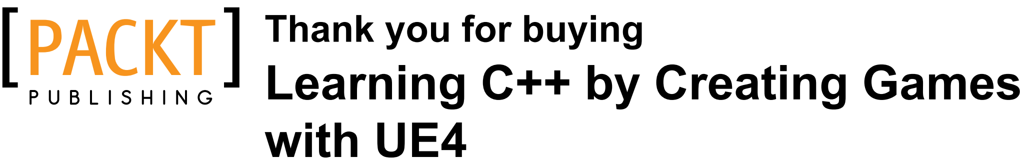
X

XCode
about 14
using, on Mac 14-17

Y

yaw and pitch
setting 176, 177

www.it-ebooks.info



About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

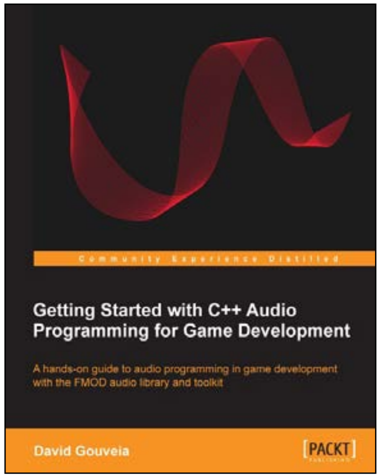
Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

www.it-ebooks.info

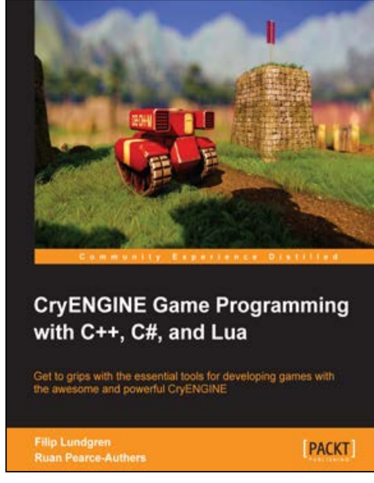


Getting Started with C++ Audio Programming for Game Development

ISBN: 978-1-84969-909-9 Paperback: 116 pages

A hands-on guide to audio programming in game development with the FMOD audio library and toolkit

1. Add audio to your game using FMOD and wrap it in your own code.
2. Understand the core concepts of audio programming and work with audio at different levels of abstraction.
3. Work with a technology that is widely considered to be the industry standard in audio middleware.



CryENGINE Game Programming with C++, C#, and Lua

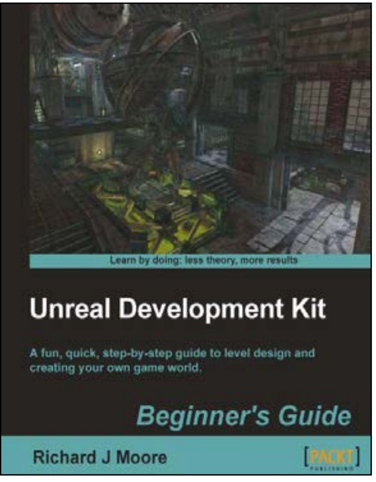
ISBN: 978-1-84969-590-9 Paperback: 276 pages

Get to grips with the essential tools for developing games with the awesome and powerful CryENGINE

1. Dive into the various CryENGINE subsystems to quickly learn how to master the engine.
2. Create your very own game using C++, C#, or Lua in CryENGINE.
3. Understand the structure and design of the engine.

Please check **www.PacktPub.com** for information on our titles

www.it-ebooks.info

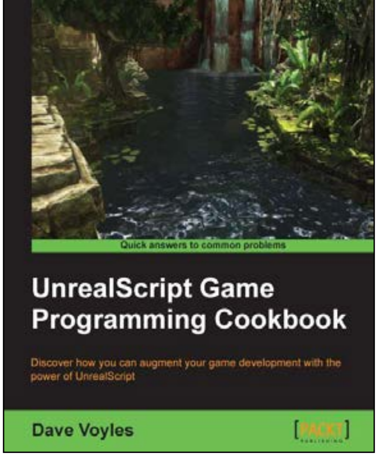


Unreal Development Kit Beginner's Guide

ISBN: 978-1-84969-052-2 Paperback: 244 pages

A fun, quick, step-by-step guide to level design and creating your own game world

1. Full of illustrations, diagrams, and tips for creating your first level and game environment.
2. Clear step-by-step instructions and fun practical examples.
3. Master the essentials of level design and environment creation.



UnrealScript Game Programming Cookbook

ISBN: 978-1-84969-556-5 Paperback: 272 pages

Discover how you can augment your game development with the power of UnrealScript

1. Create a truly unique experience within UDK using a series of powerful recipes to augment your content.
2. Discover how you can utilize the advanced functionality offered by the Unreal Engine with UnrealScript.
3. Learn how to harness the built-in AI in UDK to its full potential.

Please check **www.PacktPub.com** for information on our titles

www.it-ebooks.info