Quick answers to common problems

UnrealScript Game Programming Cookbook

Discover how you can augment your game development with the power of UnrealScript

Dave Voyles

[PACKT] PUBLISHING

UnrealScript Game Programming Cookbook

Discover how you can augment your game development with the power of UnrealScript

Dave Voyles

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

UnrealScript Game Programming Cookbook

First published: February 2013

Production Reference: 1080213

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

ISBN 978-1-84969-556-5

www.packtpub.com

Cover Image by Dave Voyles (dnvoyles@gmail.com)

Credits

Author
Dave Voyles

Reviewers
Kolby Brooks
John P. Doran
William Gaul
Dan Weiss

Acquisition Editor
Erol Staveley

Lead Technical Editor
Arun Nadar

Technical Editors
Pooja Prakashan
Veronica Fernandes

Project Coordinator
Arshad Sopariwala

Proofreader
Maria Gould

Indexer
Tejal R. Soni

Graphics
Valentina D'silva
Aditi Gajjar

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Dave Voyles has worked as a coordinator for the last two Indie Games Uprisings on Xbox Live, an annual event organized to highlight the talented developers and their titles on Xbox Live Indie Games. Additionally, he has released a title of his own, Piz-ong on XBLIG, as well as projects using Unity and the Unreal Engine for game jams.

He's proficient in C# and UnrealScript, and all facets of the Unreal Engine, as well as a number of 3D modeling suites, including 3DS Max and Maya.

He has also worked as a technical reviewer on *Unreal Development Kit Game Programming with UnrealScript: Beginner's Guide*, *Packt Publishing* and *Unreal Development Kit Beginner's Guide*, *Packt Publishing*. Moreover, he works as managing editor at Armless Octopus, a site dedicated to cover Indie game development with an emphasis on XNA and XBLIG. You can find him on Twitter under the handle `@DaveVoyles` or at `www.About.me/DaveVoyles`.

I'd like to thank my mother and father, for always supporting my gaming habit as a child, despite my poor taste in Sega CD era FMV games.

About the Reviewers

Kolby Brooks took a strong interest to programming at the young age of seven, getting his start by modifying games such as Unreal, Dirt Track Racing, and Half-Life.

He now has over 14 years of growing experience in multiple game-related fields including, but not limited to, multiplayer anti-cheat solutions, game programming, and engine development. As a hobby, he develops and maintains multiple third-party solutions for games such as server tools, utilities, and modification frameworks.

You can contact Kolby by e-mail at `brooks.kolby@gmail.com`.

I would like to thank my family, especially my mother and father, for their continued support over the years. In addition, I would like to thank Jason Ismail and Draco Rat for being great friends and gaming buddies.

John P. Doran is a technical game designer who has been creating games for over 10 years. He has worked on an assortment of games in student, mod, and professional projects independently as well as in teams having up to or over 70 members.

He previously worked at LucasArts on Star Wars: 1313 as a game design intern, the only junior designer on a team of seniors. He later graduated from DigiPen Institute of Technology in Redmond, WA, with a Bachelor of Science in Game Design.

He is currently a software engineer at DigiPen's Singapore campus while at the same time tutoring and assisting students with difficulties in Computer Science concepts, programming, linear algebra, game design, and advanced usage of UDK, Flash, Unity, and Actionscript in a development environment.

He is the author of *Mastering UDK Game Development Hotshot*, *Packt Publishing* and is the co-author of *UDK iOS Game Development Beginner's Guide*, *Packt Publishing*.

He can be found online at `http://johnpdoran.com` and can be contacted at `john@johnpdoran.com`.

Thanks so much to the author for allowing me to give him my thoughts while writing the book, I hope that they helped.

I'd also like to thank my brother, Chris Doran, and my girlfriend Hannah Mai for being there for me whenever I needed them and being patient while I was working on this.

I'd also like to thank Arun Nadar, Arshad Sopariwala, and all the lovely people at Packt for all of their support and knowhow!

William Gaul is an aspiring game developer working primarily in the Unreal Development Kit (UDK). Since 2008, he has learned a wide range of skills in the industry, and maintains an active YouTube channel (`http://www.youtube.com/user/willyg302`) and blog (`http://willyg302.wordpress.com/`) dedicated to game development.

His programming knowledge includes Java, C/C++, UnrealScript, BASIC, HTML/CSS, and LaTeX. He also specializes in a variety of media solutions, with experience in the Adobe Creative Suite, Blender, and FL Studio.

Dan Weiss is currently a programmer working at Psyonix Studios in San Diego, CA. He is a 2010 graduate of DigiPen Institute of Technology, having worked on titles such as *Attack of the 50ft Robot!* during his time there. He has been working in the Unreal Engine since 2004, independently producing the mod Unreal Demolition for Unreal Tournament 2004 and Unreal Tournament 3. At Psyonix, he has been involved with Unreal Engine work on mobile devices, having released ARC Squadron for iOS devices.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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

PACKTLIB®

http://PacktLib.PacktPub.com

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

The Unreal Engine was first introduced to the gaming landscape in 1998 through Epic's first-person shooter, *Unreal*. While the core of it is written in C++, Epic managed to craft a language of their own, called UnrealScript, which is similar to Java in a number of ways.

In November 2009, Epic released the Unreal Development Kit, an SDK utilizing the Unreal Engine, which allows developers to write and release games of their own. Many of this generation's leading AAA titles utilize the Unreal Engine, including the *Mass Effect*, *BioShock*, and *Gears of War* franchises.

My plan with this book is to allow you to have the ability to craft worlds of your own, by teaching you how to program for the industry's leading 3D engine for AAA development.

What this book covers

Chapter 1, *Development Environments*, will take us through several development environments which can handle UnrealScript, define some of the perks and pitfalls of each, and highlight the benefits of understanding the source code, through UncodeX and the Dungeon Defenders Development Kit.

Chapter 2, *Archetypes and Prefabs*, will show you that as a programmer, one of your tasks is to assist the level designers. This can be done in a number of ways, but one of the most useful ways is to create what are known as prefabs and archetypes. By creating templates for objects and actors and only exposing the variables that a designer will find to be useful, you can make your work, and that of a designer, far more efficient.

Chapter 3, *Scripting a Camera System*, tells us about cameras in UDK that are an essential part of gameplay. They can simultaneously be one of the most frustrating yet rewarding things to program, as once they are working correctly they can completely change a player's experience, because you control their window to the world.

Chapter 4, *Crafting Pickups*, tells us that artificial intelligence can cover a variety of things in UDK, so we won't delve too far down that path, at least not in this chapter. Here, we'll briefly cover it, and how the AI interacts with pickups throughout the game, specifically what attracts them to certain pickups. Furthermore, we'll dive into creating our own pickups and how they interact with our pawn's inventory.

Chapter 5, *AI and Navigation*, shows us that the Unreal Engine has two ways of handling path finding. They both have their pros and cons, despite being somewhat similar. They can simply be broken down into waypoints and navigation meshes. Each offer their own sets of perks and pitfalls, so we'll explore the pros and cons of both.

Chapter 6, *Weapons*, walks you through the weapons in UDK that are inventory items, which can be handled by the player, and are generally used to fire a projectile. On the surface, the default weapon system found in Unreal Engine 3 may appear to be catered to creating various types of guns as is common in most FPS games; but it's actually pretty easy to create various sorts of weapons and usable inventory items, which may be found in other types of games such as healing projectiles, bombs, landmines, or flashlights, as in the case with *Alan Wake*.

Chapter 7, *HUD*, shows us that heads-up display, or HUD, in addition to a user interface (UI), offers a means for providing information to a player to allow them to interact with the game world. UDK offers two methods for creating a HUD. The first, and far more simple method that we'll be covering here is the canvas. The other method, which requires knowledge of the flash language and some fancy art skills, allows UDK to make use of a third-party tool called ScaleForm to draw the HUD.

Chapter 8, *Miscellaneous Recipes*, walks you through the recipes that are going to cover things which may not necessarily fit in one particular chapter, but are still very valuable in a number of applications. We'll go over a new scheme for aiming our weapons and drawing a crosshair, as well as allowing our pawn to flash continuously as its health depreciates, among other useful items.

What you need for this book

For this book you will need the following items to get started:

- The latest build of the Unreal Development Kit, which can be found at `http://www.unrealengine.com/udk/downloads/`
- An integrated development environment
- A solid understanding of object oriented programming basics

Who this book is for

This book is designed for users who have a solid understanding of object oriented programming and want to get introduced to a powerful yet unique language using a well established framework. Although the Unreal Development Kit offers an extensive schema for first-person shooters, it can be so much more if harnessed correctly. Prior experience with 3D Math or other 3D engines will certainly be helpful as well.

Conventions

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

Code words in text are shown as follows: "A player's view into the world is determined by the `Camera` class."

A block of code is set as follows:

```
classTutorialGame extends UTGame;

defaultproperties
{
  PlayerControllerClass=class'Tutorial.TutorialPlayerController'
  DefaultPawnClass=class'Tutorial.TutorialPawn'
  DefaultInventory(0)=class'UTWeap_ShockRifle'
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Scroll down to **World Properties**, and left-click on that".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/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.

Questions

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

1
Development Environments

In this chapter, we will be covering the following recipes:

- Using UnCodeX
- Dungeon Defenders to save the day
- Unreal Script IDE
- nFringe
- Unreal X-Editor
- Editing runtime values with Remote Control

Introduction

Working with UnrealScript can be a daunting task at first glance, especially because it has years' worth of extensive improvements and iterations, in spite of being a language only used for this application. To make things worse, UDK does not include a development environment out of the box, so we're forced to find one that best suits our needs. Fortunately there are several out there, each of which bears many pros and cons.

In this chapter, we will look at several development environments which can handle UnrealScript, define some of the perks and pitfalls of each, and highlight the benefits of understanding the source code, through UnCodeX and the Dungeon Defenders Development Kit.

So with that, let's talk about integrated development environments.

Integrated development environments

An **integrated development environment** (**IDE**) sounds far more complicated than it is, as it simply serves as a way for a developer to talk to the machine and write code for an application. It usually consists of three components, a source code editor, build automation tools, and a debugger. The IDEs we'll cover in this chapter provide all three.

Using UnCodeX

Now that we know what IDEs are and how they work, how do we use the code provided by Epic? It's there for the taking, but we need an easy way to sift through it. That's where UnCodeX comes in. Let's take a closer look.

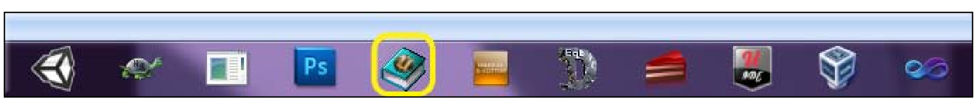
UnCodeX is an open source tool which provides an easy interface to browse through the code, analyzes UnrealScript, and is capable of producing a Javadoc like documentation of the source code.

Every good programmer knows their source material. That doesn't mean you need to know the engine inside out, but at least take some time to understand where your most important classes are extending from, and what's available to you. There's no sense in reinventing the wheel if someone has done it for you already, right?

Getting ready

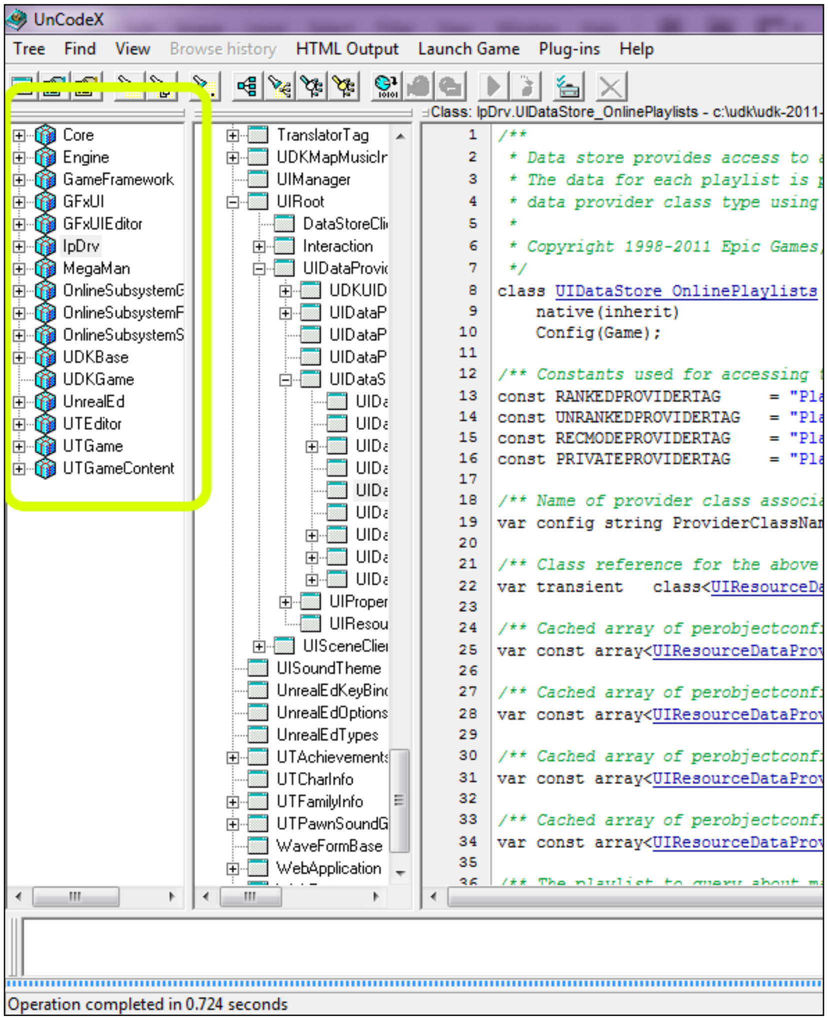
The first thing we'll need to do is acquire a copy of it for ourselves. Let's head over to `http://sourceforge.net/projects/uncodex/` and download it.

Once you have it downloaded and installed, open it up. Personally, I keep it pinned to my taskbar as it is a constant point of reference for my work that I find myself accessing day in and day out, along with my other Unreal tools.



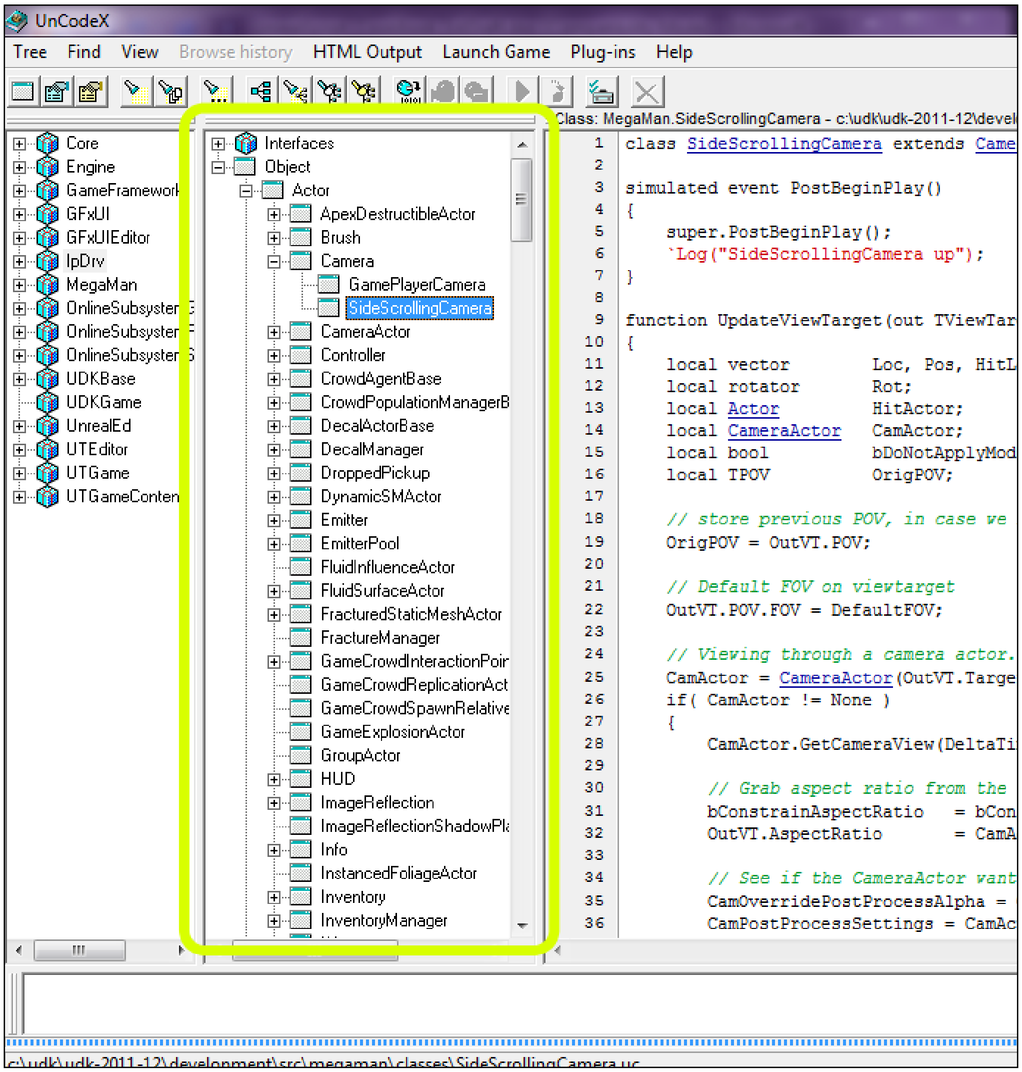
How to do it...

On the left-hand side of the screen, you'll see the **package browser**. This allows us to see all of the packages currently contained within your UDK directory of choice.



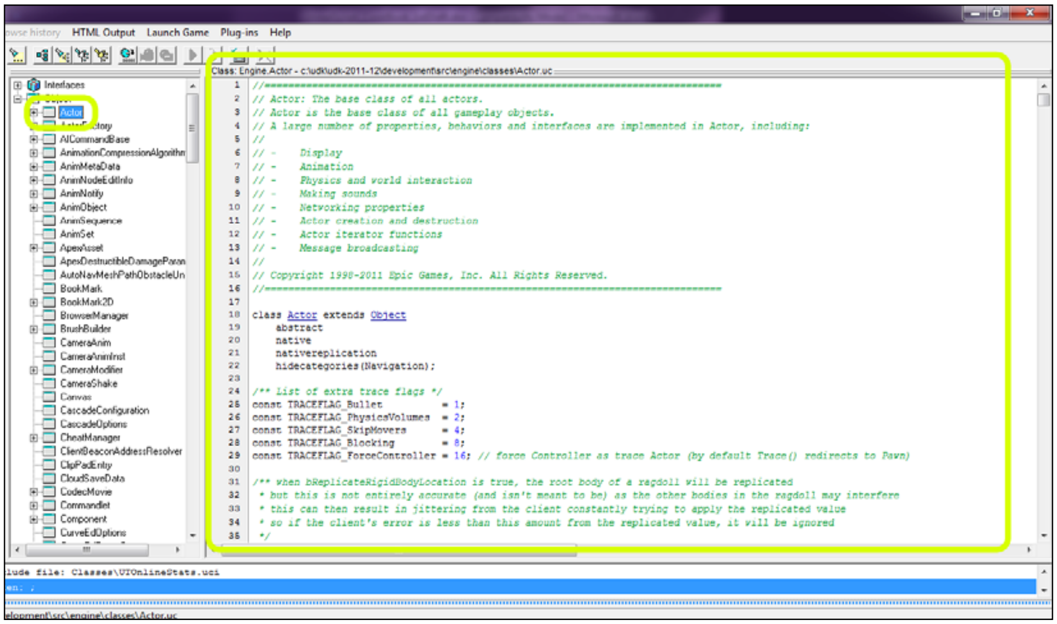
This can be updated manually by clicking on the **Tree** menu in the top-left corner of the browser, and then clicking on **Rebuild and Analyse**. It may take a few moments to rebuild the package, as it is sorting through any changes you may have made since the last time you wrote any code. You can check the current status of the operation in the bottom-left corner of the program.

Adjacent to the package browser is the **class tree browser**. Most of the IDEs we'll be working with will include a class tree as well. The tree browser allows us to dig deeper into the code by seeing exactly how each and every class in UDK are connected to one another.



You'll notice that all classes in UDK extend from the `Object` class. It's the base class for everything in the game and allows for everything in the game to share some common properties and functions. `Actor` is perhaps the class you will be most concerned with however, as it is the base class for all gameplay objects in UDK.

It can be a bit overwhelming at first, especially when you see precisely how many classes are there. Most of what is there you will never use, but it still makes for an incredible reference. It's not what you know, but more importantly it's about knowing where to find what you're looking for.



To the right of the class tree you'll see the class browser. Double-click on one of the classes from the class tree to view its contents to the right. Other classes will be underlined and colored blue, just like when you create a URL in a web browser or word document.

Hovering the mouse cursor over any of the underlined text in the class browser will draw a pop up on the screen, which illustrates the location on your hard drive where you can find that class. This is a great way to help visualize how UDK's classes are interconnected and assist you in understanding how to best utilize them to create your own.

There's more...

UnCodeX is an essential part of any UnrealScript programmer's tool belt. It can not only help you understand what is currently running under the hood, but also help you understand the best practices for extending from its base classes to create your own.

As a rule of thumb, you don't need to know the base code inside out, but it's essential that you at least have an understanding of the work Epic has laid at your fingertips. This includes the functions such as `Tick`, `PostBeginStart.PostBeginPlayt`, and the default properties block.

There are a number of other great resources to find additional content and help for UDK and UnrealScript. These include:

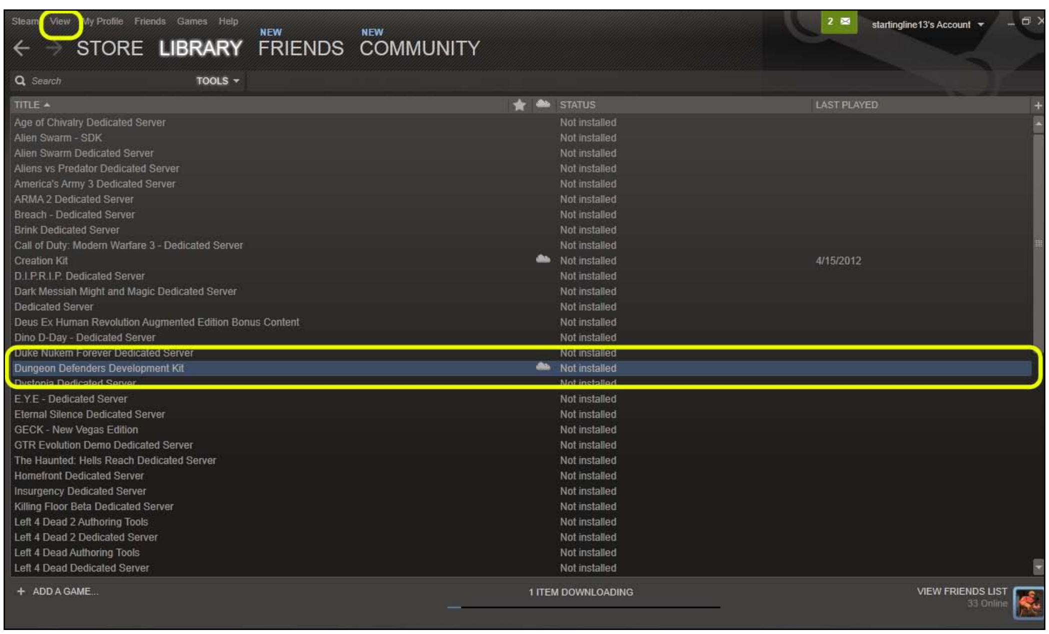
- Epic's UnrealScript forum at `http://forums.epicgames.com/forums/367-Programming-and-Unrealscript`
- Eat 3D's UnrealScript reference at `http://eat3d.com/unrealscript`
- Epic's UDK Gems at `http://udn.epicgames.com/Three/DevelopmentKitGems.html`

Dungeon Defenders to save the day

Epic isn't the only team to offer a plethora of knowledge at your fingertips. Trendy Entertainment, the development studio behind Dungeon Defenders, was also kind enough to release much of their source code and development kit, in what is known as **Dungeon Defenders Development Kit** (**DDDK**). Similar to UnCodeX, this source code allows you to have a far greater understanding of how an entire game is pieced together.

Getting ready

The DDDK can be found by downloading and installing the Steam client at `http://store.steampowered.com/about/`. Once installed, click on **View**, which is at the top of your screen, then select **Tools**. The DDDK is actually a piece of DLC, and can be found here.



How to do it...

One of the largest contrasts you'll find between the Dungeon Defenders source and that of UDK is their use of cameras and the player controller, as that game utilized a third person perspective. Take a few moments to sift through the code and understand the path that Trendy Entertainment took.

It's well documented too, so even a novice should be able to sift through the code and see how things are connected. Even better, Epic's UDK forums are filled with questions from other developers asking questions about that particular set of code, in addition to code from other resources, so there are chance that many of your questions have been previously answered there. Take a look at `http://forums.epicgames.com/forums/367-UDK-Programming-and-Unrealscript` and see for yourself. The search bar can prove to be an invaluable tool when sifting through the forums as well, considering they are rather dense and filled with subforums from a number of Epic's properties, such as *Gears of War*.

Moreover, some of their programmers are extremely active in the UDK forums and often provide a bit more introspection on the code and how it is used.

How it works...

As license holders, Trendy Entertainment utilizes a custom version of the Unreal Engine, thereby allowing the developers of the DDDK to have far more control of the engine than we could without the license. This wasn't always the case however, so much of the original UDK code is still in place. With the DDDK you can either create modifications for Dungeon Defenders, or total conversions to the entire game.

See also

So why did Trendy Entertainment release their source code for free? Well, publicity could be one reason. Epic heavily promoted Dungeon Defenders at launch and lauded it as an excellent example of how to utilize the engine to create something different from what they created it to do.

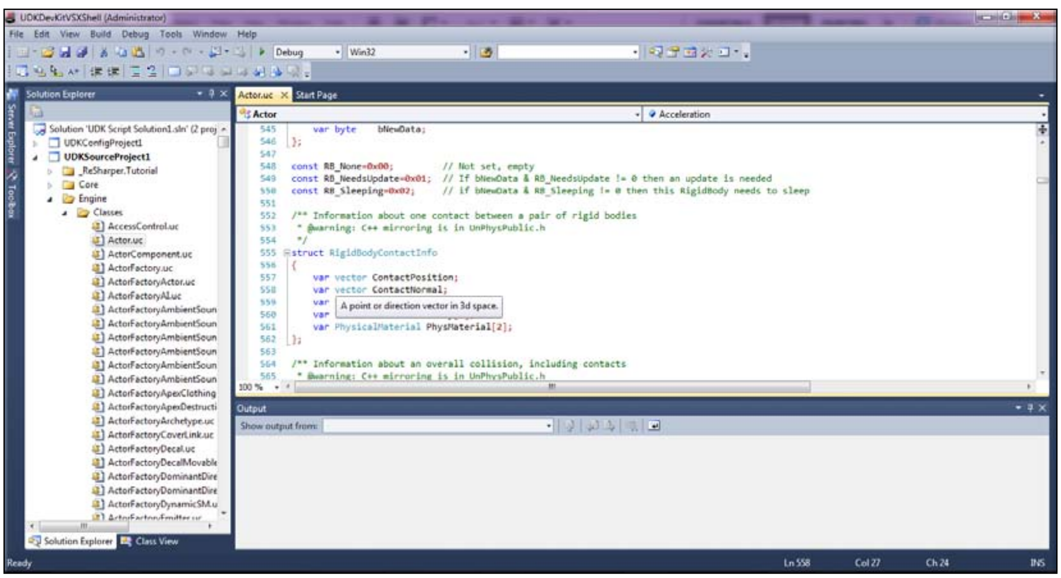
Copies of the game are very affordable nowadays, sometimes running even as low as a few dollars during a Steam sale, and the original demo is still free. So I'd suggest picking up a copy and seeing how it compliments all of that code you've just sifted through. Even if you don't own the game, the source is still available as free DLC through Steam or can be found at `http://www.unrealengine.com/showcase/udk/dungeon_defense/`.

Unreal Script IDE

The Unreal Script IDE is a professional development framework, utilizing the Visual Studio shell. If you're a .NET developer, then you should feel right at home with this IDE. It will be our tool of choice for this book for a number of reasons, but most notably for a few of the features which aren't found in any other development environment, such as the following:

- **Debugging**
- **Conditional Breakpoints**
- **Go To Declaration**
- **IntelliSense**
- **Find All References**

There are many other reasons I prefer this environment over the others, but those are just some of the highlights. **IntelliSense** in particular is extremely useful, as it auto completes your code as you write. Furthermore, it makes for easy disambiguation for functions, methods, and variables. **Find All References** is also notable, as it locates any instance of a particular variable or function within UDK, and shows you where and how it's used. This is outstanding for learning the source material.



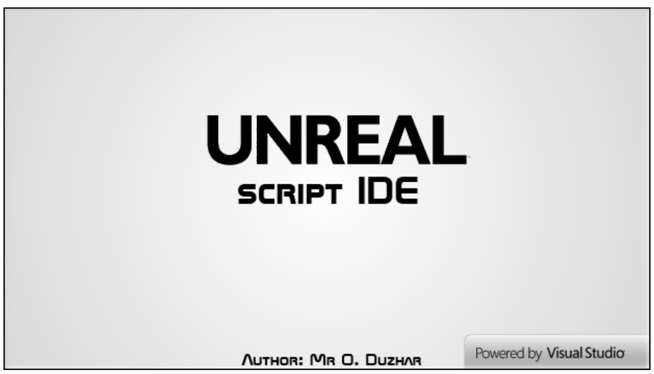
Getting ready

Head over to `http://uside.codeplex.com/` to grab a copy of this free open source tool. You won't need your own copy of Visual Studio either, as this runs in an isolated shell. Bear in mind, you must have UDK installed before installing the IDE.

How to do it...

When installing the IDE, it's important to remember to select the UDK Win32 binaries folder correctly. Additionally, the source folder must be listed as `C:\UDK\UDK-Date\Development\Src\`; otherwise your project will appear empty when you open the solution.

> This is the default installation path. Yours may be different.



From there, your Unreal Script IDE should be populated with your current project, and update itself automatically as you make changes.

There's more...

Even better, the Unreal Script IDE allows you to continue to use extensions! Head to `http://visualstudiogallery.msdn.microsoft.com/`, to find the ones which work best for you. Afterwards, extract the content into the `Extensions` folder found at `C:\Program Files (x86)\Mr O. Duzhar\Unreal Script IDE (UDKDevKit) VS 2010 Isolated Shell\`.

nFringe

Just like the Unreal Script IDE, nFringe is a complete IDE that also uses the Visual Studio shell. Similarly, it offers many of the features that Unreal Script IDE does, but many require a commercial license of the Unreal Engine, such as **Goto Native Definition** for native script functions. For this reason, I prefer to use the former tool.

Pixel Mine is the development team behind this product, which comes in both premium, indie, and commercial licenses, although it can get a bit costly, as indie licenses start at $350 a seat. A trial version is available however, so you may want to consider that before taking the step into the premium pond.

Many professional AAA studios prefer to use this IDE, so focusing your efforts here may not be a bad idea. Furthermore, nFringe includes an excellent debugger, which can prove invaluable when trying to troubleshoot critical errors in your programming.

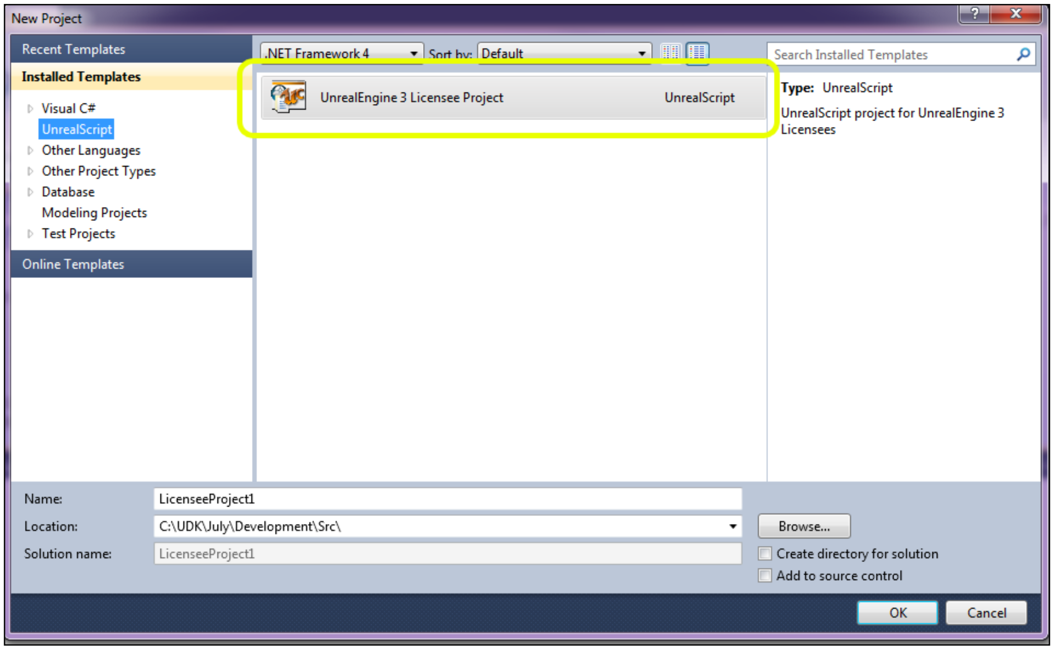
> nFringe will not work with Visual Studio 2012 right out of the box. You'll need to make some changes for it to work properly, as noted in the following site:
>
> `http://forums.epicgames.com/threads/874296-debug-Unreal-Script-error-at-vs2011`

Getting ready

Head over to `http://pixelminegames.com/nfringe/` to pick up a copy for yourself. It's one small download and a pretty straightforward installation process.

How to do it...

Just as Unreal Script IDE used a Visual Studio shell, nFringe does too, but it is based on the 2008 version. Even if you don't own Visual Studio, you're still in the clear as you can run it through the shell. For those of you who do have a copy of Visual Studio, nFringe simply installs like an extension, and allows you to create new projects from your current installation. Moreover, you can also use Visual Studio Express, which is the free version of the program.

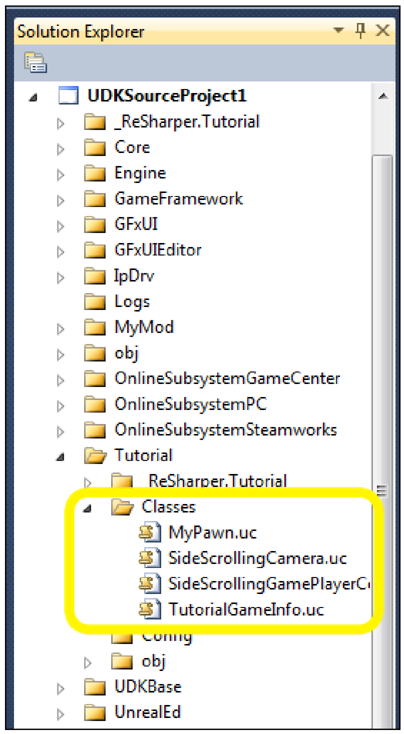


Starting a new project is as simple as opening up Visual Studio, selecting the installed **UnrealEngine3 Licensee Project** template, and you have all of the features of nFringe available at your fingertips. Setting up the new project is a bit more work however, as you'll need to carefully follow these instructions to do so.

Configuring the directories is the first step in creating your own project scripts:

1. Within your UDK install directory (that is, `C:\UDK\July`) browse to the `Development\Src` folder and create a folder of your own. In this example we'll use `Tutorial`.
2. Create a `Classes` folder inside your `Tutorial` folder.

All of your scripts will be stored in here. You cannot create more folders within your `Classes` folder for organizational purposes. You can however create various packages within UDK to neatly separate your classes.



3. Now we need to notify the engine that you'll be adding a new package, or collection of scripts, and that they should be compiled each time you build the game. The configuration files stored inside of `UDKGame\Config` are the ones which inform the engine of this package. Browse over to that folder and open the `DefaultEngine.ini` file.
4. Once inside, scroll down to `[UnrealEd.EditorEngine]` and add the name of our own game package. It should now read `+EditPackages=Tutorial`.

> The order in which the packages are loaded into UDK are directly related to the order in which they appear in this `.ini` file. If your package relies on any of the `UTGame` or `UTGameContent` scripts (and if you extend from anything within UDK, then your package certainly does), then your package must be loaded after those scripts.

```
[UnrealEd.EditorEngine]
+EditPackages=UTGame
+EditPackages=UTGameContent
+EditPackages=Tutorial
```

How it works...

That's all there is to it. Another reason I prefer Unreal Script IDE and even the forthcoming Unreal X-Editor is because all of this work is done for you during the initial installation, because you pointed the install towards your UDK directory.

There's more...

Extensions also work with nFringe, so most of the ones you already have installed should seamlessly integrate with your new UDK project.

Unreal X-Editor

A strong contender for the most intuitive IDE is the Unreal X-Editor. While the editor has come a long way since its initial release, it's still in its infancy, as the tool is less than one year old at the time of this writing.

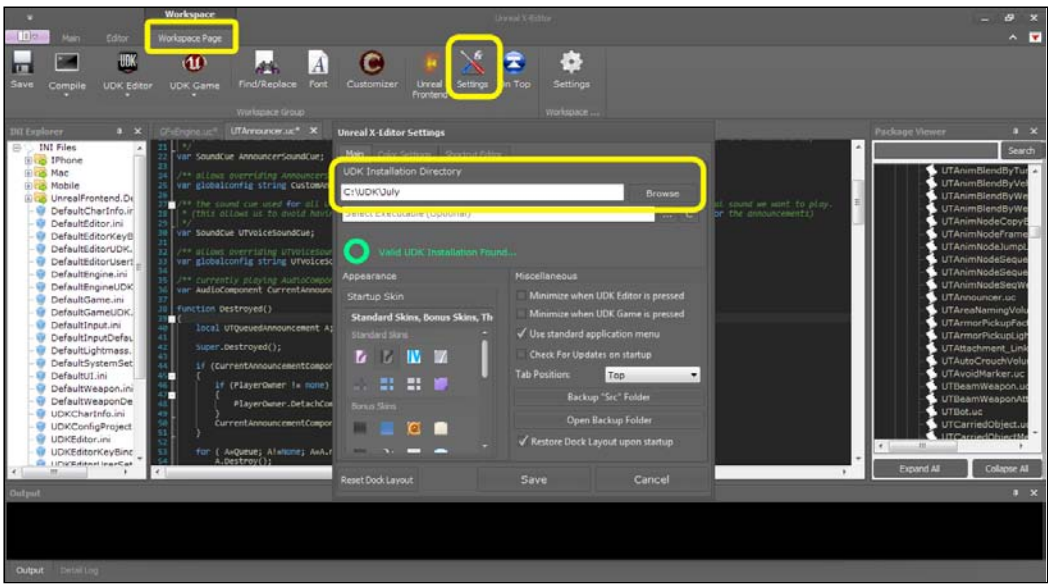
Unreal X-Editor offers a number of features that makes it stand out, including the following:

- **Class Tree Viewer**
- **Autocomplete**
- **Code Folding**
- **C# Style Commenting**
- **Syntax Highlighting**
- **Basic Preset Scripts**
- **Run UDK Editor / UDK Game**
- **Compile/Full Compile Scripts**
- **Various Skins to change the Look And Feel**

Getting ready

Head over to `http://unrealxeditor.wordpress.com/` to download the latest version of Unreal X-Editor. Setting up the IDE is pretty straightforward as well, so no explanation is needed here.

Once you have it installed, scroll over to the **Workspace Page** tab, then left-click on the **Settings** button. A pop up should appear on screen, allowing you to customize your settings. We're looking to set your UDK installation directory, so click on that and browse to the folder where you placed UDK.



Unreal X-Editor will then have access to all of the content in that folder, including the UDK executable and any folders containing the `.ini` files, the UnrealScript source code, and your new folders.

That's it! This is by far the easiest tool to get up and running.

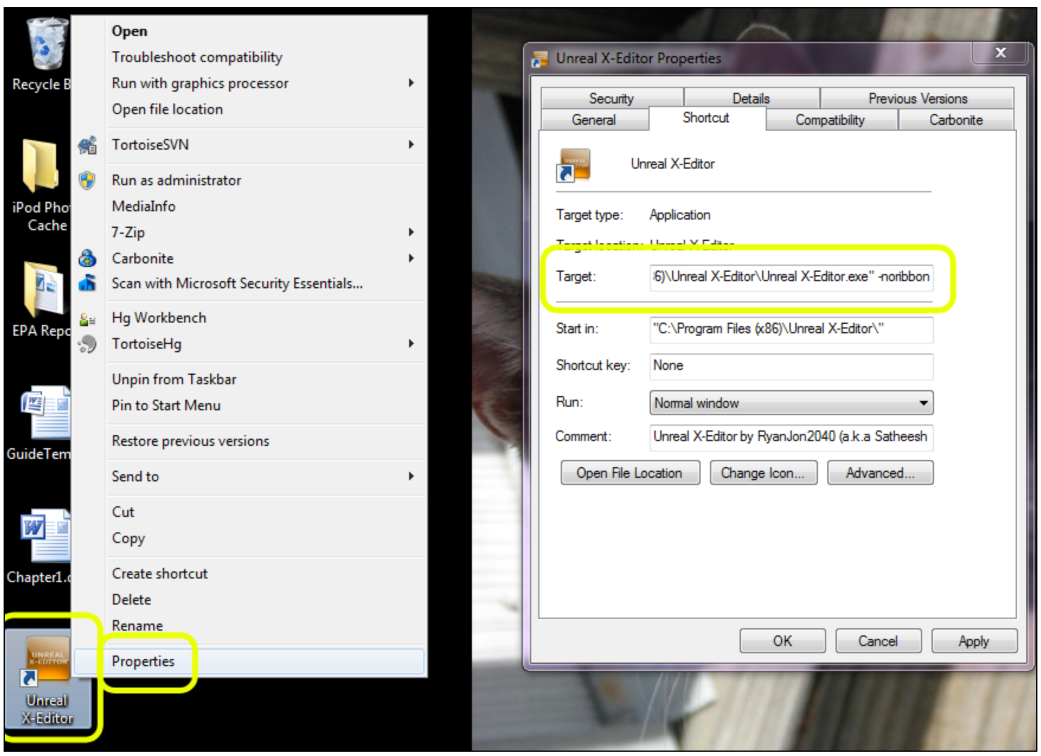
How to do it...

Unreal X-Editor offers a number of customization options, including additional skins, various font colors, and the ability to work without the GUI ribbon, thereby offering an interface more akin to our Visual Studio alternatives.

As far as simplicity goes, this IDE offers just about everything a programmer could want, and does so with a slick interface. Those of you who prefer the extra real estate and want to work without a ribbon can do that as well by doing the following:

1. Right-click on the desktop shortcut and select **Properties**.
2. In **Properties**, select the **Shortcut** tab and in the **Target** box add the command line `-noribbon`.

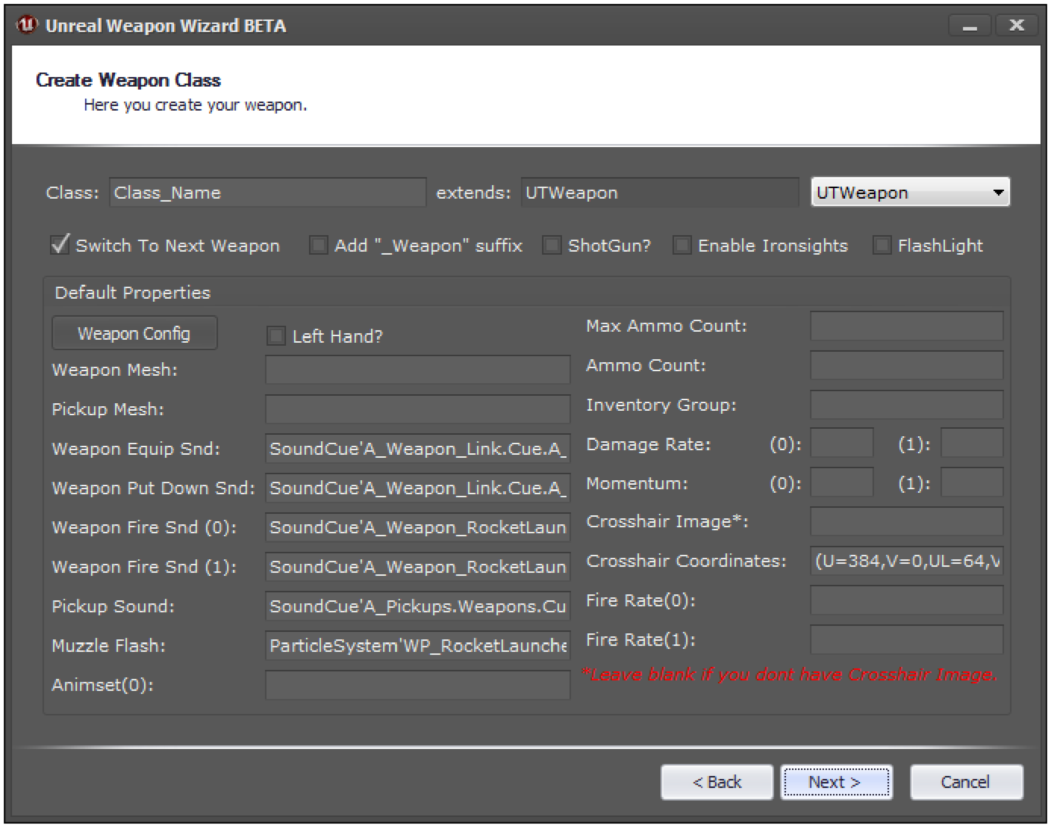
> An **Access Denied** dialog may appear, declaring that you will need to provide administrator permission to make these changes. If so, simply click on **Continue**, and then click on **Apply** and **OK**.



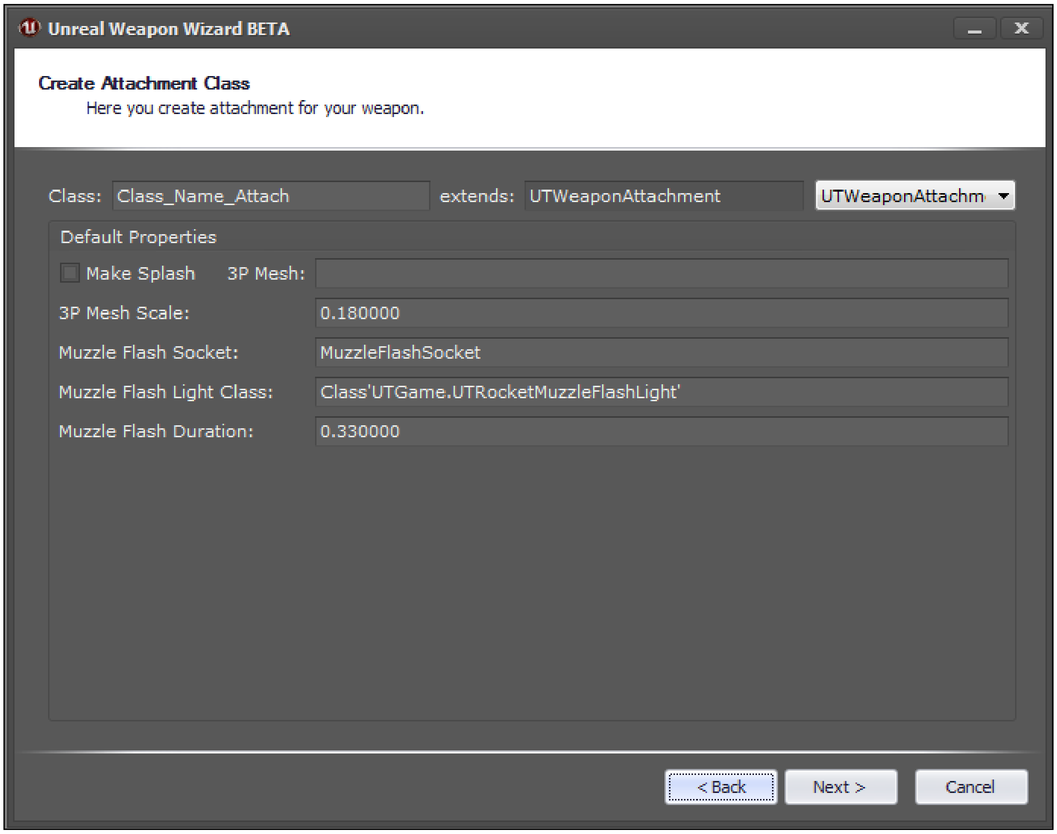
There's more...

Unreal X-Editor is perhaps the greatest IDE for those starting off with UnrealScript, and for a number of reasons. It streamlines the often intimidating and convoluted process of making changes to a game utilizing UDK.

Let's take a look at the **Unreal Weapon Wizard**. This tool allows us to create a weapon class with the help of visual editor. Currently only the weapon class is included, although other classes such as character and game are planned for the future as well.

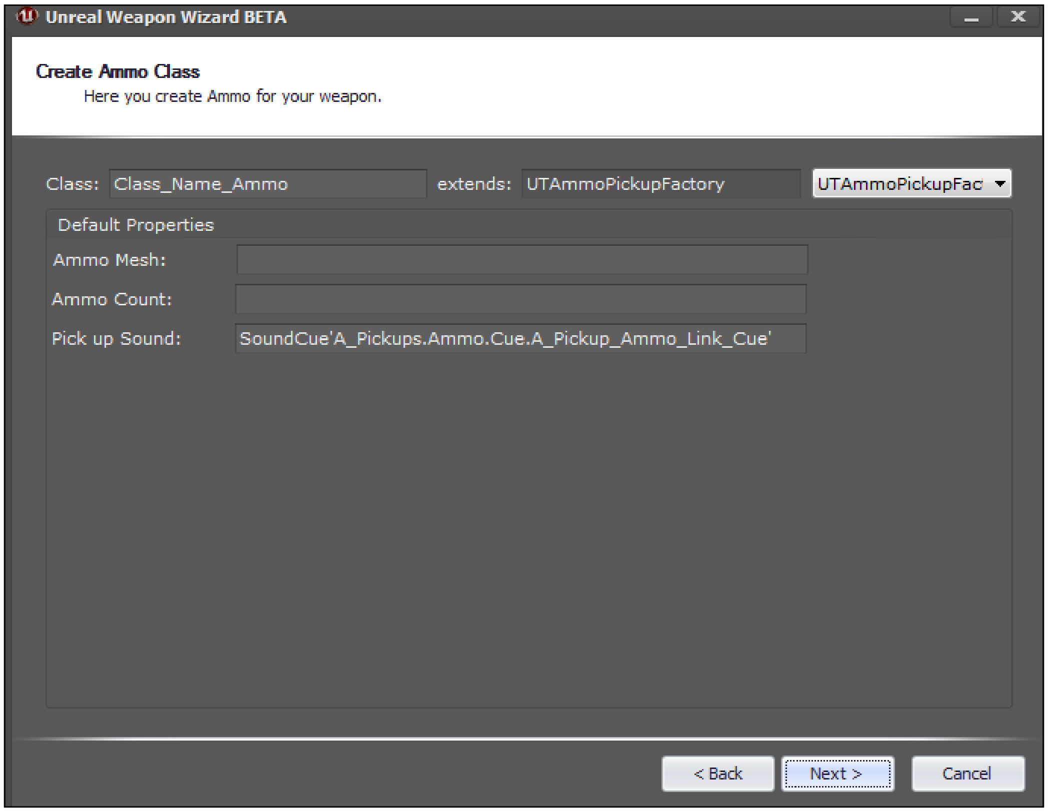


All you need to do is copy and paste the names of various weapon related properties, such as mesh, muzzle flash, and pick up sound from the UDK Content Browser to the appropriate fields in the wizard and click on **Next**.



This is a great way to get started with learning how weapons work and are constructed in UDK. Compare and contrast your new weapon with some of those which come packaged with UDK to really get a feel of how to develop some weapons of your own without the wizard.

Afterwards, you'll be greeted with another screen for constructing your attachment class, which extends `UTWeaponAttachment`. This class determines how the weapon connects with your pawn when in a third person perspective.

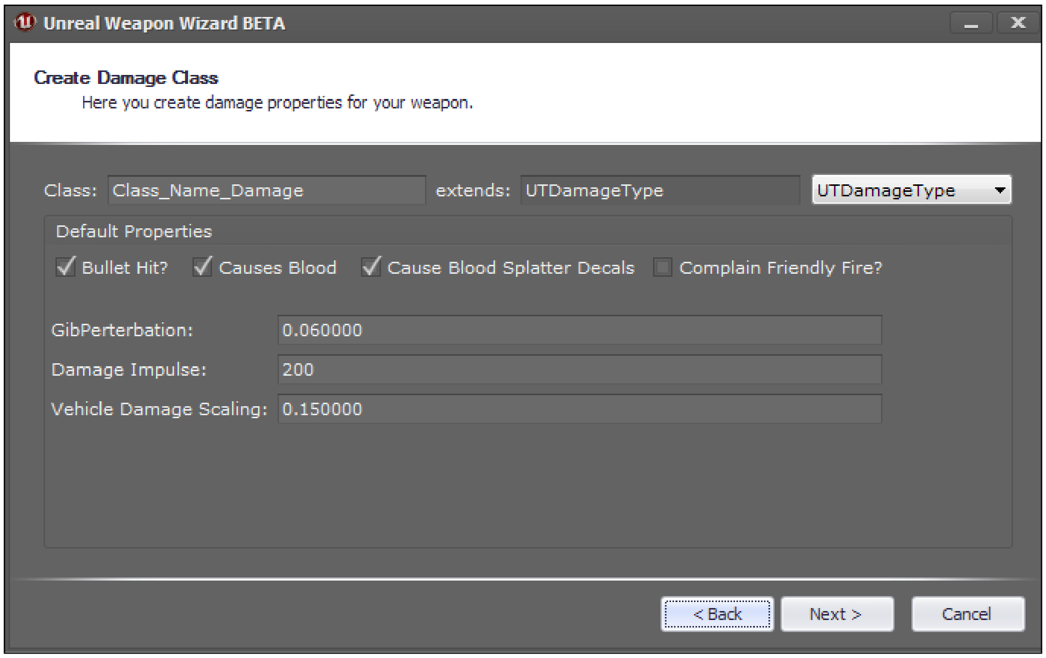


The preceding screenshot shows the wizard for your ammunition. This is the easiest class to create, as it only requires a mesh, default starting count, and sound effect for when it is picked up. It extends `UTAmmoPickupFactory`, which we'll touch on more in a later tutorial.

The definitions of some of these properties can be a bit confusing, so I'll clarify these in the following list:

- **Make Splash**: This displays a splash effect for the player when the projectile hits water.
- **3P Mesh**: This is the skeletal mesh for this socket. Generally left empty.
- **Muzzle Flash Socket**: This determines where the flash will occur in relation to the weapon when it is fired.

> Often weapons have multiple sockets (that is, rocket launchers, double-barreled shotguns), so that's why this option is there.



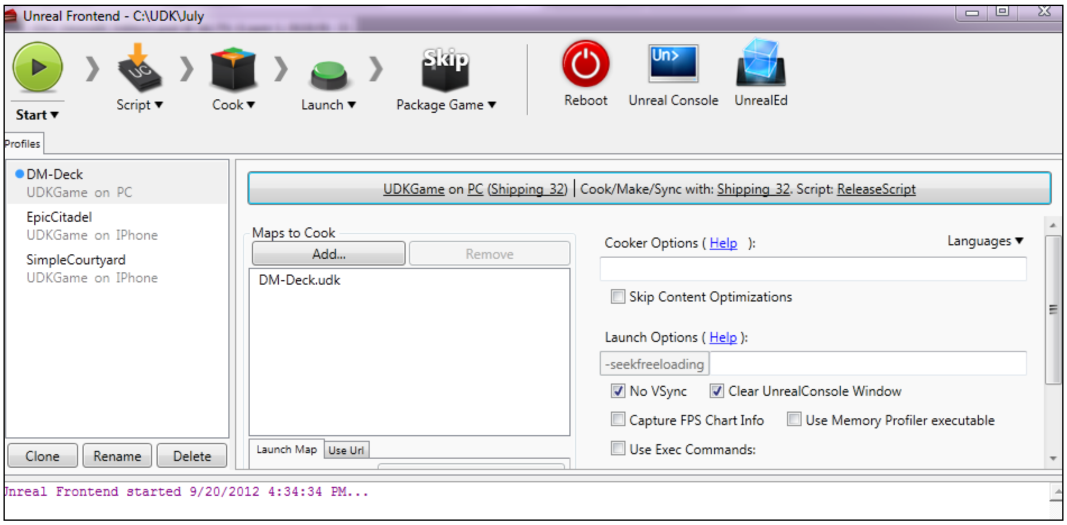
The final step of the process is to create the damage class, which covers a few properties you may not be familiar with, so let's walk through some of them:

- **GibPeterbation** is a Boolean, and means that when it is active the chunks will fly off in random directions
- **Bullet Hit?** notifies the target that it was hit by a bullet
- **Complain Friendly Fire?** determines whether teammates should complain about friendly fire with this damage type
- **Vehicle Damage Scaling** determines whether or not a weapon should do more damage to be in proportion with increased health and armor of a vehicle, as opposed to just firing at a pawn
- **Damage Impulse** determines the size of impulse to apply when doing radial damage

Once that's all finished, you have the option of opening all of your newly created classes to explore your creation. Take a look and see how they compare to the default weapons created in UDK and how they are assembled!

See also

There are a number of visual customizations that you can make to the editor as well, from fonts and backgrounds to panel layout. Find one that suits you best!



In addition to the visual configuration, the Unreal Frontend is available from within the Unreal X-Editor, which allows you to compile scripts, cook a level, and package a game, all from Epic's supplied interface.

Editing runtime values with Remote Control

As an UnrealScript programmer, one of your main tasks is often to assist level designers and artists by creating tools to streamline their work. While UDK comes with a robust system for adding and editing content in game, sometimes you just want to create a weapon or item to be used in game and only expose certain properties to the level creators.

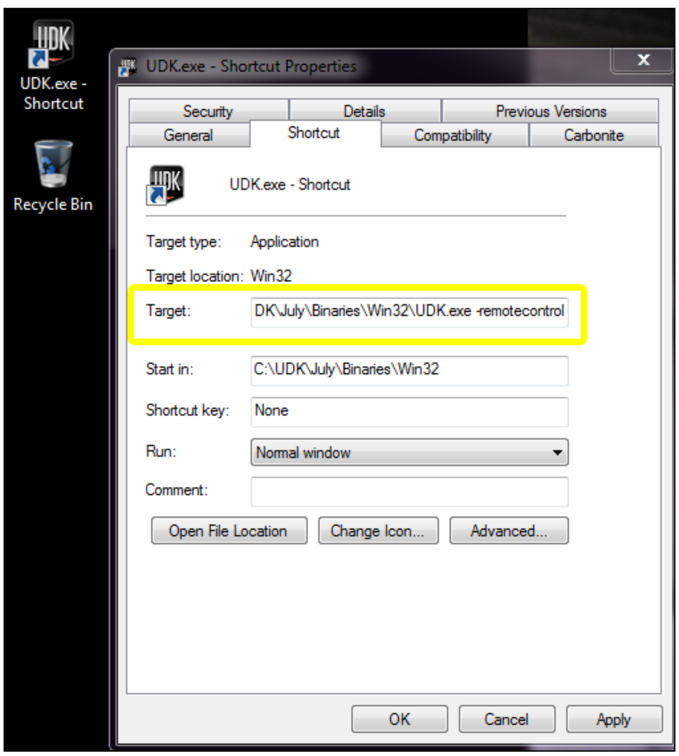
We'll use a weapon as an example. Not every property is applicable to what a level designer may want or need, therefore we'll hide some of those from the designer's editor to provide a clean interface for them to work with and serve as a means to streamline their weapon creation process through the use of **prefabs**, which we'll touch more on later. First we'll need to understand how UDK natively allows us to do this.

UDK already provides a way for developers to alter properties at runtime through a feature called **Remote Control**. This allows you to make alterations to which statistics are being captured, alterations to graphics settings, as well as to scrutinize and edit actors in a scene. A plethora of console commands that allow you to edit most instantiated objects, archetypes, or actors at runtime are also at your fingertips.

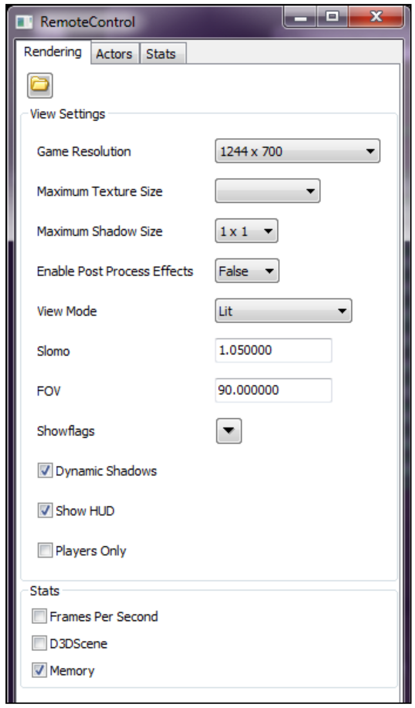
When tweaking the values of a particular object within your game, for example, a pawn, the Remote Control can prove invaluable. Rather than going through the arduous process of changing code within the IDE, compiling scripts, then launching the game, you can make these small adjustments within Remote Control while the game is running. This is perfect for those moments when you want to quickly iterate and lock down the values that make a character feel just right, for example, a particular running speed, or jumping height. Once you've found the value you're looking for, you can always go back into the UnrealScript code and permanently make those changes.

Getting ready

To launch the game with Remote Control enabled, you'll need to edit your UDK launch batch file by adding `-remotecontrol` as an argument to the end of your `UDK.exe` file, just as we did for the ribbonless version of Unreal X-Editor.



Upon launching the game, it may state that your scripts are out of date and ask you to rebuild them. Just click on **OK** and let them compile. Once that is completed, you'll start the game as you normally do, except that there will now be another panel next to your game. This is where you'll be able to make all of your changes.

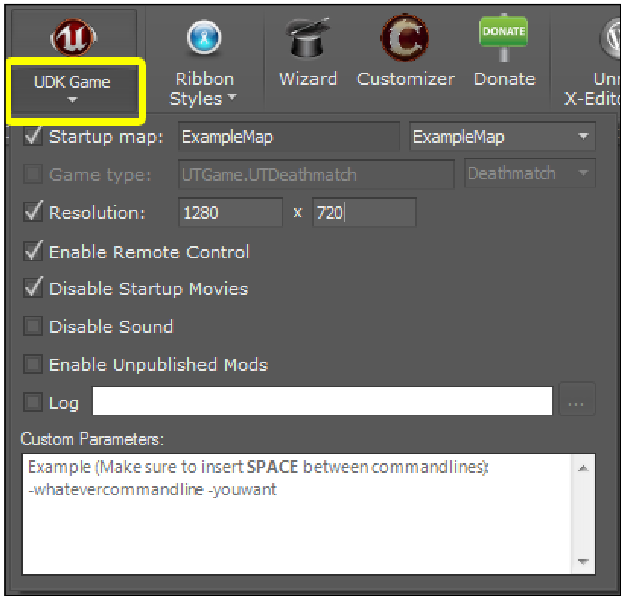


How to do it...

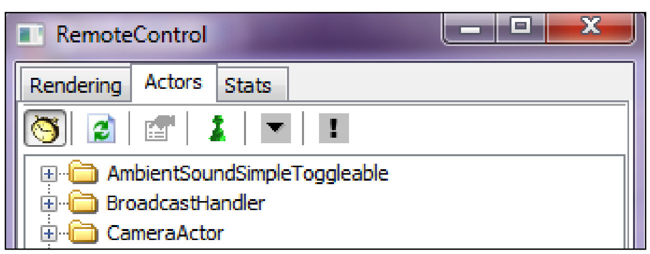
With Remote Control now available we can begin to create objects for level designers to work with. Let's start by creating a simple weapon:

1. Open Unreal X-Editor.
2. Click on the bottom half of the **UDK Game** icon to make the game's options editor appear.

3. Adjust your settings so that they match the ones shown in the following screenshot. Be sure to check the checkbox marked **Enable Remote Control**!



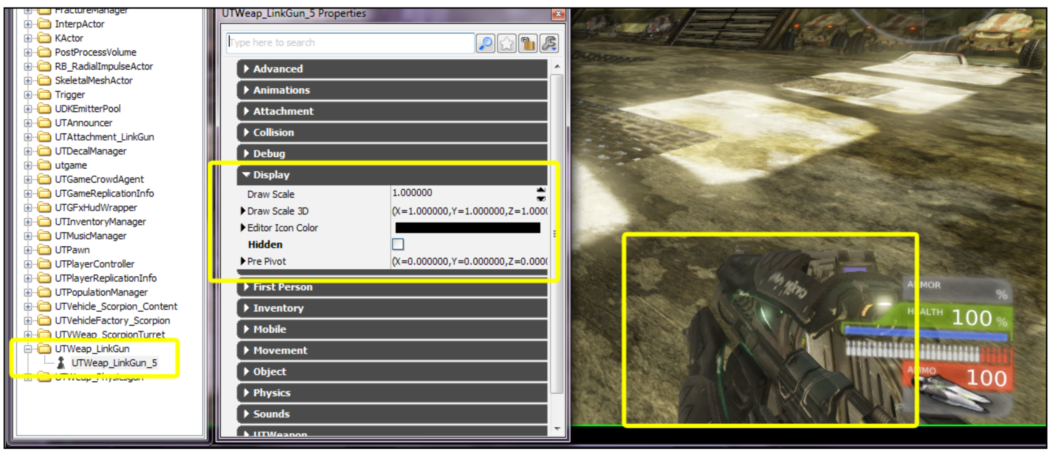
4. Click the top half of the **UDK Game** icon to launch the game.
5. When your game launches, you should see the **Remote Control** panel appear to the left of the screen. Fire your **Link Gun** and then select your **Actors** tab.
6. Click on the refresh icon to update **Remote Control** and allow it to list the latest actors into the scene.



> The **Link Gun** won't appear in your **Actors** tab, until you actually fire it and hit refresh to load everything that may have occurred since the map's initialization. That's why we're having you perform these actions.

7. Scroll down to the `UTWeap_LinkGun` folder, open up the folder, and double-click on `UTWeap_LinkGun_#`. An in-game editor window should now appear.

> # can be any number, but is generally `0`. If there are pawns that spawn into your scene, then the window may not appear immediately as that gun may be assigned to one of them. Continue to select the different `UTWeap_LinkGun` until the window appears for you.



8. Scroll down to the **Display** tab in this new editor and take a look at the properties here. This is where you'll be able to adjust the appearance of the gun from your perspective. Feel free to play with various properties and see what they do.
9. Checking the **Hidden** checkbox will hide your weapon from view.
10. Click on the **Display Scale 3D** tab to bring down the **X**, **Y**, **Z** values for where your weapon is drawn in 3D space.

> **Want to see what your weapon would look like if you were left handed?**
> We know that 0 is the center of the screen, so we'll need to negate the value to move it to the left-hand side. Change the value in the **Y** scale to a negative number to do this.

> You'll notice that when firing our weapon, the locations where the projectiles and beam start no longer line up with our weapon. That's because we haven't changed those values.
>
> I also can't tell you the exact values to change, as the appearance of the weapon in the game will depend largely on the resolution with which you are running.

11. From here you'll want to scroll down to the **Weapon** tab and look for **Fire Offset**. These values will allow you to adjust how the projectiles will appear to fire from your weapon.

There's more...

You can also call objects and actors by their names using the command line. Hitting the *Tab* key during gameplay and typing `EditObject <nameofobject>` during play will allow you to create an instance of most objects.

2

Archetypes and Prefabs

In this chapter, we will be covering the following recipes:

- Construct a leaking pipe prefab
- Adding particles to our prefab
- Adding audio effects to our prefab
- Create a PointLight archetype
- Create a subarchetype from an archetype

Introduction

As a programmer, one of your tasks is to assist the level designers. This can be done in a number of ways, but one of the most useful ways is to create what are known as prefabs and archetypes. By creating templates for objects and actors and only exposing the variables that a designer will find to be useful, you can make your work, and that of a designer, far more efficient.

So with that, let's talk about prefabs.

Prefabs

A prefab is a combination of multiple actors into one unit. This allows us to easily manipulate the properties and visuals of multiple objects on screen at once.

For example, if you wanted to use a torch in a game, you could grab a particle effect for the fire, another for the smoke, then a static mesh for the torch handle, a sound effect instance for the crackling sound, and finally a light source for the lighting. Or you could combine all of these objects into one actor and call it a torch.

Prefabs are tied to the original assets through references, therefore if you make a change to an asset, the next time you load UDK the changes will be applied to the prefab. This allows for quick and dynamic prototyping, as you can create a prefab ahead of time, using placeholder objects that you know you'll need, and adjust the aesthetics at a later time when the assets are ready from the artist or modeler on your team.

That's the purpose of a prefab, to simplify your life and streamline the development process.

Constructing a leaking pipe prefab

Prefabs are simply a combination of multiple objects and actors within a scene. We can combine virtually any item in UDK to create one.

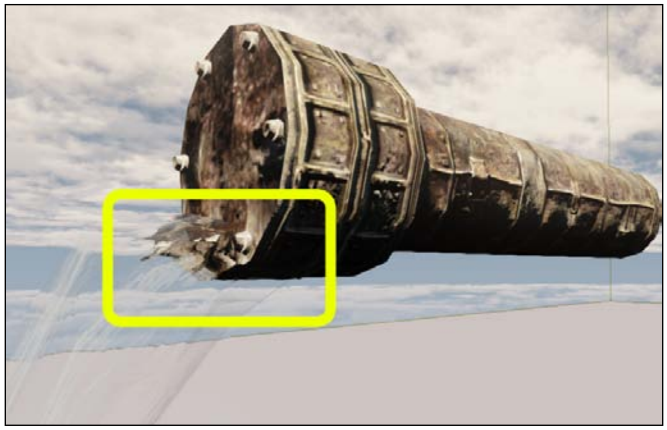
Getting ready

We'll begin by loading the map that contains our example prefab. This will allow you to see the various components used to create one, as well as teach you to create one of your own. Along the way you'll have to manipulate certain objects such as rotating a pipe or adjusting the scale of a waterfall, but this won't be anything overly complicated.

How to do it...

To get things started, let's open up a new map called `Chapter2` under the `Tutorials` folder. This will have the chapter's prefab and archetype already in place for you, so that you can see the end result of your work, and what it should look like. Fortunately, all of our in-game assets can be accessed through the ever useful content browser. Let's begin by looking in the following:

1. Open the **Content Browser** tab and drag the static mesh `S_HU_Deco_Pipes_SM_PipeSet_B01` into the scene. This will serve as the static mesh for the pipe itself.
2. Rotate the pipe by 90 degrees, so that it is horizontal, to match the following screenshot. Pressing the Space bar allows the widget to change from translation to rotation.
3. Our pipe is looking pretty bland by itself, so let's add some flowing water to it. Also under static meshes, you'll find `S_UN_Liquid_SM_Waterfall_02`. Drag that into the scene as well.
4. Line up the center of the waterfall with the center of the pipe. It's still a bit too large for the pipe, but we'll fix that in the next step.
5. Adjust the scale of the waterfall on the x plane, so that it appears small enough to actually be falling from the pipe.



So now we have all of our assets lined up and drawn to the correct scale. Our last task is to group them as a prefab, so that we can easily create more and use them throughout the level.

6. Select both of your objects and right-click on them to bring up the options menu. Scroll down and left-click on **Create prefab...**.

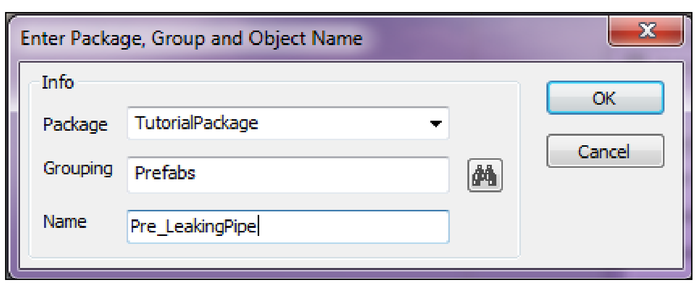
> *Ctrl* + *Alt* + left-click and drag in viewport will allow you to marquee select items, which can save a great deal of time. Alternatively, *Ctrl* + left-click will allow you to select multiple items at once.

7. When the prompt to save the prefab appears, I saved mine under the following settings:

   Package: `TutorialPackage`

   Grouping: `Prefabs`

   Name: `Pre_LeakingPipe`



If you see a white box with a red **P** in the center, then you've successfully created a prefab!

We've essentially created our first prefab. It's still a bit bland though, so we'll add more to it in the next chapter.

> When saving the prefab, it is important to center it within your viewport, as that image will be saved in the content browser for when you have to reference it later.

How it works...

A prefab is a combination of multiple actors in one unit. This allows us to easily manipulate the properties and visuals of multiple objects on screen at once.

We can combine virtually any actors within UDK and create a prefab from them. For simplicity's sake, we've only used static meshes in this example, but we'll soon include more advanced components.

There's more...

Prefabs keep a record of whenever a property is altered when used in the editor, so you can edit one prefab and not worry about how it will affect the rest of them. Alternatively, you can carry the change you just made to one prefab and apply it to all of them across the scene as well. This is both a time and labor saving process, so give it a try yourself.

Adding particles to our prefab

Our prefab only needs two items for it to actually be called prefab, but it looks a bit plain at the moment. Let's spice things up a bit by adding some subtle particle effects to it.

Getting ready

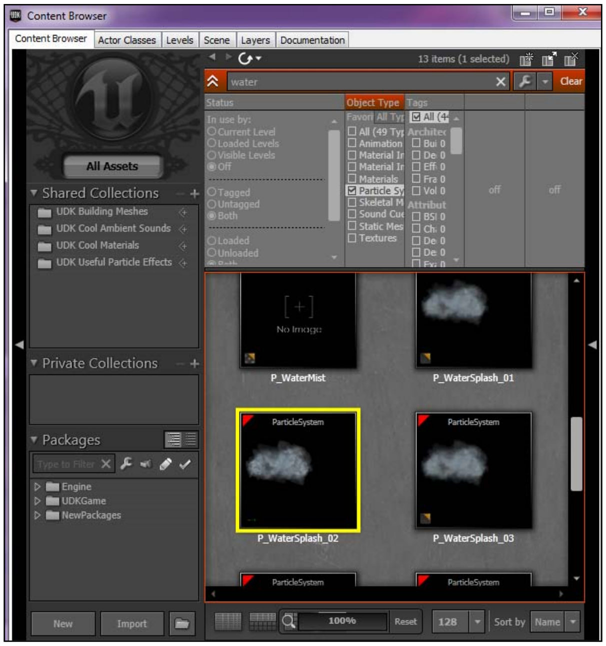
We only need to open up the prefab we just made, as we'll be building off of that.

How to do it...

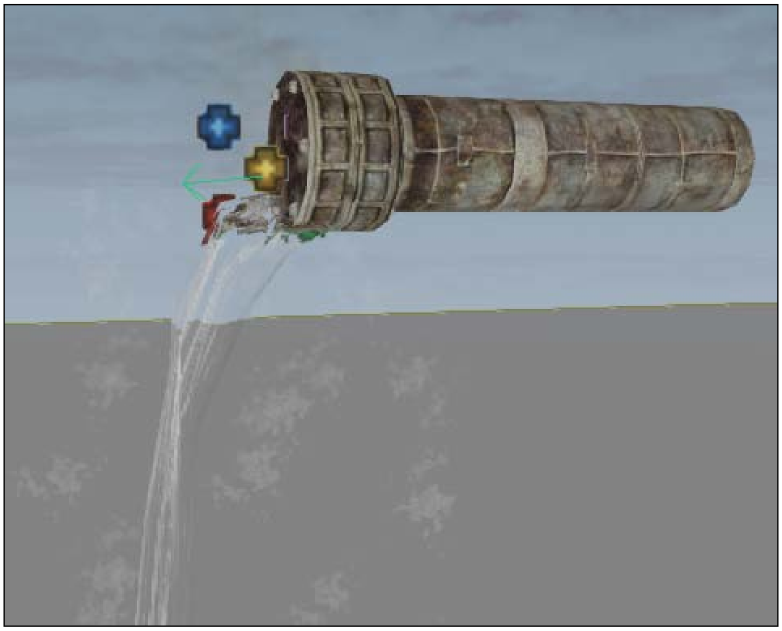
This will be an easy recipe; we're only going to drag-and-drop a particle effect onto our prefab, and combine the results. It's easy to create or edit particles on your own, but that isn't covered in the scope of this book. The content browser is where we'll begin most of our recipes in this chapter, as it offers easy access to all of our in-game assets. Let's begin by opening that.

1. Back in the **Content Browser** tab, under particles, select `P_WaterSplash_02` and drag it into the scene.

> You'll have to be careful when using particles, as they are costly in terms of computing overhead and can quickly bog down your system if too many are on screen at once.



2. Align the particles in front of the pipe, so that it is nearly overlapping with the water. We want to create an effect to appear as though there is hot steam radiating from the front of the pipe. Adjust the scale of the particle accordingly.



3. Be sure to save your changes again.

How it works...

The content browser allows us to easily add components to our prefab by simply searching for it within the browser, and dragging it into a map. Selecting our prefab and an additional component at the same time allows the two (or more) items to be connected, and in turn, part of the prefab from then on.

There's more...

Mix and match some particle effects until you find some that really work for you. Although it's beyond the stretch of this tutorial, spending some time with the particle editor can really add a sense, so that you have some trickling water to the pipe. Just be sure to take it easy when adding too many particles as overdoing it can quickly bog down a system due to the intense load on the CPU.

Adding audio effects to our prefab

Our prefab is nearly complete! We've got all of the visual aesthetics for our prefab in place, but it's still not very believable within the game because we don't hear any sound coming from it. Let's change that.

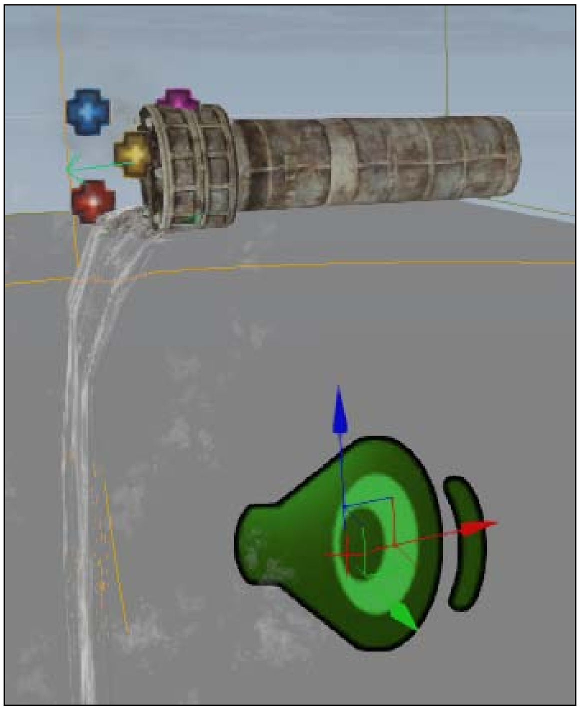
Getting ready

We only need to open up the prefab we just made, as we'll be building off of that.

How to do it...

The changes here will be minor, but similar to what we've done with the particles in the previous recipe. UDK offers extensive audio capabilities, but those are over the scope of this tutorial, so we won't cover them here, but you can find a number of resources at the Unreal Development Network site, `http://udn.epicgames.com/Three/AudioHome.html`.

1. Head back to the **Content Browser** tab and search for the final piece to make this work, that is, the `Waterfall_Medium_02_Cue` sound effect. Drag it into the scene and align it closely with the other objects for our leaking pipe.



2. Align the sound cue beneath the pipe. When you hit the **PIE** button, you'll notice that the sound gradually decreases as you move further from the pipe as well.
3. Be sure to save your changes again.

How it works...

Just as before, the content browser offers quite a bit of value to our level designer, as it allows him or her to quickly parse through our library of assets and drag content onto our map. Prefabs can be created at any point when we select two or more components at the same time.

See also

Try adding some water beneath the piping, so that your stream is flowing into a pool of water. How do you think the sound of the water would change? See if you can find an appropriate sound cue for water splashing against another water source.

Creating a PointLight archetype

Archetypes and prefabs share a number of similarities, but most noticeably in that we use them to create instances of an object. All instances of an archetype on a map will update when an archetype is stored in a package, and that's one of the convenient reasons for using them.

Furthermore, if you chose to only alter an instance of an archetype, then the rest of those on the map will remain the same! So you have both ends of the spectrum, all within one tool. Archetypes also allow us to create physical representations of our code, to be manipulated and edited within the UDK editor, and later used in the game itself.

Archetypes are different from prefabs in three distinct ways:

- Prefabs can be composed of archetypes
- Prefabs can contain sequences (Kismet)
- Prefabs preserve and translate object references within the prefab

In the editor, archetypes are represented as placeable items, much like how classes are placeable resources. Archetypes are often thought of as "script less classes", in that their purpose is to provide a way for designers to drop an actor that uses a set of default property values, which are different from the actor's class defaults. They also appear in the content browser, and provide the same functionality as any other resource type.

By default, the scripts and classes you write for UDK will appear in the **Actor Classes** tab, adjacent to the **Classes Browser** tab in the editor. By converting our classes to archetypes we can visually edit properties for these classes, thereby allowing for changes on the fly, instead of constantly having to change code in the IDE, rebuild, and then view our changes in the editor.

We'll be creating an archetype in UnrealScript to allow the level designers to use it in the editor. To have a better understanding of how archetypes work within the editor in UDK visit `http://udn.epicgames.com/Three/UsingArchetypes.html`.

To streamline the prototyping process and eliminate clutter from archetypes, we'll also be hiding certain properties from the level designer. Alternatively, we can expose new variables as well and organize them within categories. This allows us to remove properties that generally wouldn't be beneficial or of any use to a level designer. If your designer ever wanted more control over an archetype, it is as simple as changing one line of code.

Getting ready

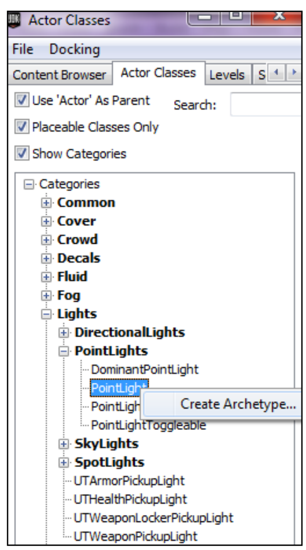
We'll begin by loading the map that contains our example archetype. Much like our prefab, this will allow you to see the various components used to create one, as well as teach you to create one of your own.

To get things started, let's open up a new `DefaultMap` under the `Tutorials` folder. Alternatively, if you want to see the end result of this chapter, load the map in the Chapter 2 folder called `Ch2_Archetypes`.

How to do it...

We're going to start by creating an archetype from a PointLight, which is already provided by UDK. Since the development kit offers a plethora of excellent and professional assets available to us, we'll use these for most of our recipes.

1. Open the **Actor Classes** tab, and left-click on **Lights** to open the pull-down list.

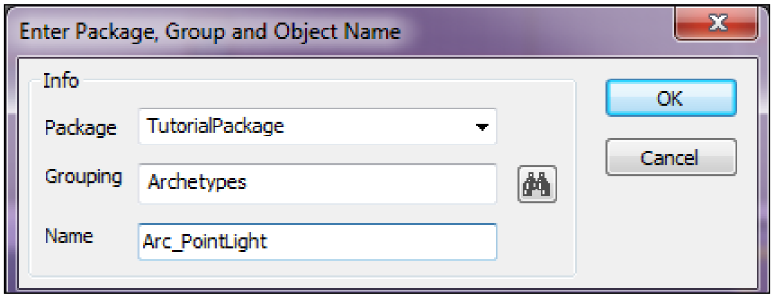


2. From there scroll to **PointLight** and left-click on the **+** to open up all of the lights under that category. Right-click on **PointLight** to bring up the **Create Archetype...** menu.
3. Left-click on **Create Archetype...** and bring up the **Enter Package Name** pop-up window.
4. I've entered the following information into the fields:

   Package: `TutorialPackage`

   Grouping: `Archetypes`

   Name: `Arc_Pointlight`



5. Now head to your content browser, and under your `TutorialPackages` folder you should see a group called `Archetypes`, and it is filled with your new `Arc_PointLight` object. With that selected, press *F4* to bring up its properties.

   You now have a new instance of a `PointLight` archetype, which you can use as a template for future lights in your game.

   We have quite a few properties exposed here though, and things look a bit cluttered. Our level designer probably won't need access to all of these things, such as the static mesh actor, collision, and debug. It is only a light, after all.

   Let's head back to our IDE and make a brief change to the class.

6. Save your package by right-clicking on your folder marked `Tutorial` in your content browser. If you see a ***** next to the package then you know that you haven't saved since you last made a change.
7. Close the editor and use your IDE to create a new class. We're going to extend from the `PointLight` class, as we've just used it in the editor, and it seemed to fit our needs. Create a new class called `Tut_PointLight`. Your code should look like the following:

```
classTut_PointLight extends PointLight;
```

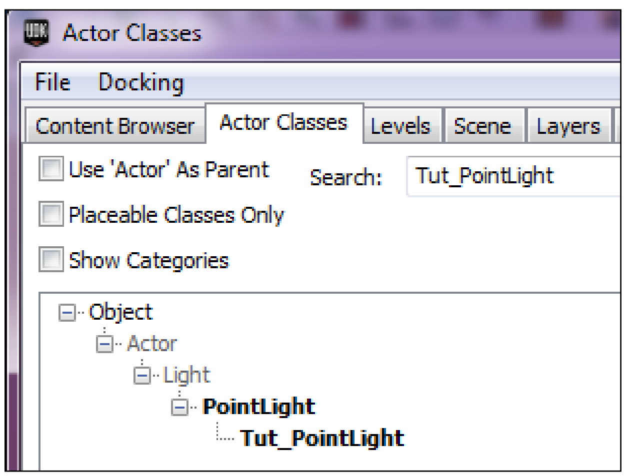
8. We've got our class created, but now we need to hide some properties from the editor. Add the following code beneath where you declared your class:

```
/** Hides categories that we won't be needing from the
archetype */

HideCategories(Object, Debug, Advanced, Mobile, Physics,
Movement, Attachment, Physics, Collision);
```

   Recompile the code. Be careful to note, however, that you must remove the semicolon after `extends PointLight`, otherwise you'll receive an error. With this done, we can now head back to the editor and create a new archetype.

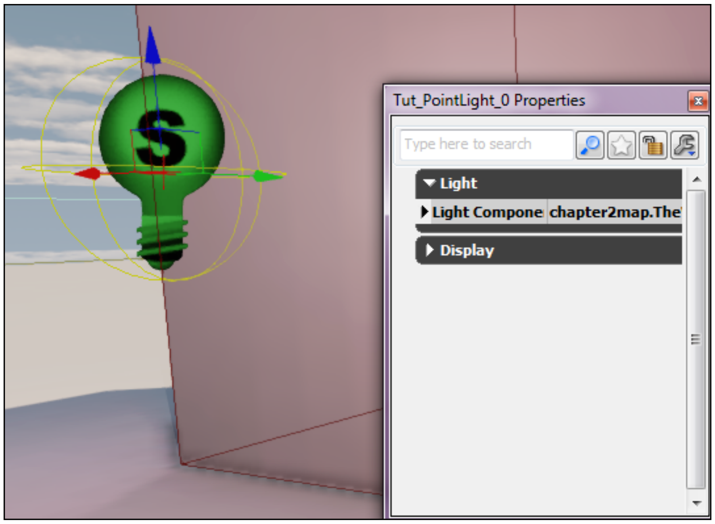
9. Save your class and open the UDK editor.
10. Create a new archetype from your new `Tut_PointLight` class. Use the same naming scheme as we had used before, so that it overwrites the one we've previously made, as we won't be needing it anymore.

> Make sure that the checkboxes for **Use Actor As Parent**, **Placeable Classes Only**, and **Show Categories** are not marked, otherwise your class will not be shown.

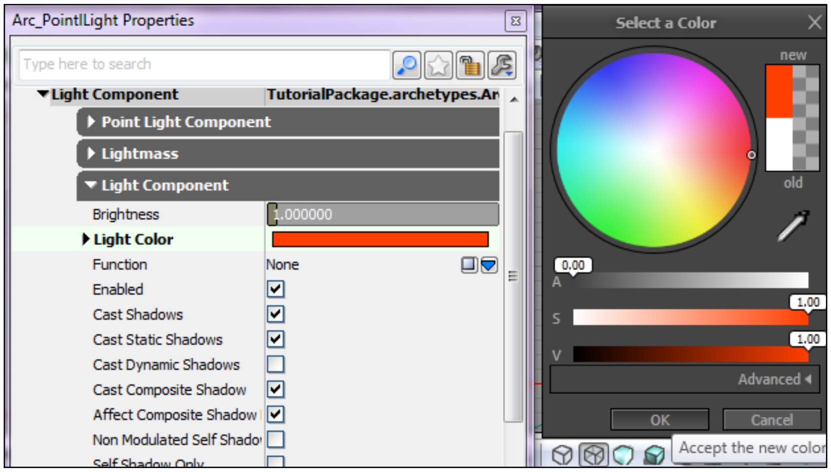


11. Drag your new archetype into the map and press *F4* to bring up its properties again.

    You'll immediately notice that many of those properties that were cluttering our screen before are now gone!

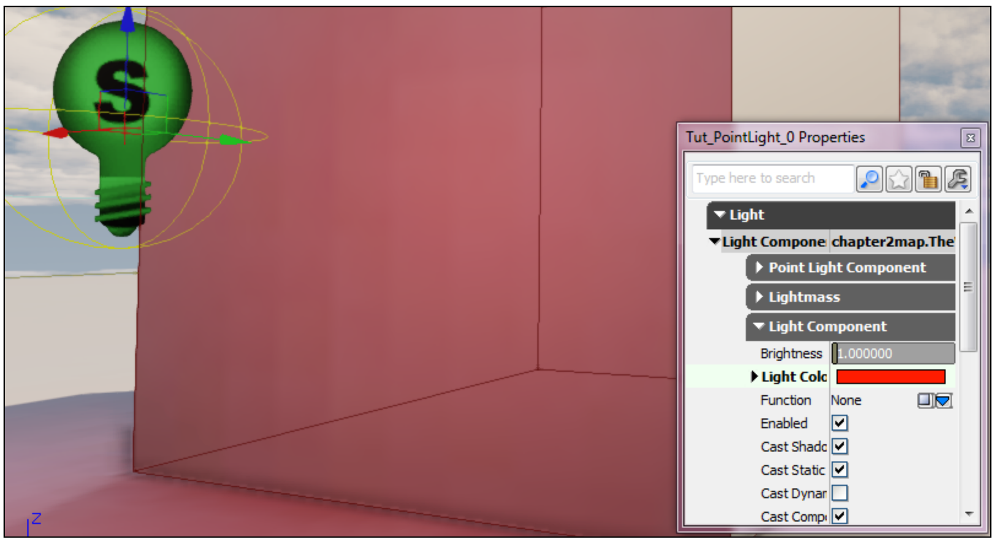


12. Let's continue by altering some values. Let's adjust our light so that it emits a red glow. Under the light gray colored **Light Component** tab, select the indented, dark gray colored **Light Component** tab.
13. In bold letters you will see **Light Color** and it is currently white. Left-click anywhere on the color bar to enable the **Select a Color** window, which is similar to the color wheel you would see in other art programs like Photoshop. Left-click on the red area and find a value that suits you. I've gone with **1.00** for **S** (Saturation) and **1.00** for **V** (Brightness).



14. Let's change the radius of our light as well. Open the indented dark gray colored tab marked **Point Light Component** to expose the **Radius** property. Let's cut it in half, from the default value of 1024 down to 512 to give us a more concentrated light.
15. We should also shrink the **Falloff Exponent** property, so that the light source has a sharper decline from its brightest point to its lowest. Cutting this in half, from its default value of two and changing it to one, will suit our purpose.

> You can see these changes in real time by dragging your archetype into the scene at any point. Notice that our adjustments to the **Radius** and **Falloff Exponent** properties shrink the light blue spherical lines around our PointLight when viewed from the wireframe mode in the editor.



> **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.

How it works...

The best way to look at archetypes is to consider them as templates. You are inheriting all of the values from one parent class and simply adjusting the default properties to what you feel suits the needs for your current application.

There are a number of occasions when you may need to make use of archetypes, so let's go through a few of them:

- Multiple deviations of an actor or object is necessary (that is, a ball which has an assortment of colors)
- Altering objects within the editor, thereby making a level designer's role easier, as they will not have to access the UnrealScript to make these changes
- Reduction of compile-time and load-time overhead, that is, you can have multiple instances of the same object and the Unreal Engine simply sees it as one object, thus lower computational overhead

Multiple deviations of an actor or object is necessary

Inheritance, or extending from a parent class, is a common practice in object-oriented programming languages. This allows us to inherit all of the properties from a parent class, as well as make changes to those properties in our new child class, while adding some new functionality of our own.

If you want your functionality to stay the same but are looking to make content or data driven differences (colors, particle effects, damage, and so on), then archetypes are an excellent solution.

A simple enemy of multiple classes is an excellent example to use. Say you want to have a scout class and a heavy weapons class in your game. Many of these changes are cosmetic, so you may just be altering the clothing color so that they appear different, but you want the scout to obviously run much faster than the heavy weapons, while also having a smaller amount of health. Simply changing the default maximum hit points and run speed of these two characters will create a great difference as well.

Altering objects within the editor

Like I mentioned before, with using Remote Control, tweaking values during runtime or while in the editor is a quick way to iterate and allow non-programmers to make adjustments without ever seeing the code.

You may find yourself in a situation where art assets for a project are not yet available, but you understand the functionality you'll want out of an object. Therefore you can use placeholder art, which will later be replaced with the final product, but in the mean time your time can be spent putting the prototype into place.

By opening the archetype properties within the level editor, a designer can quickly change how objects and actors appear and interact with a level.

Reduction of compile-time and load-time overhead

Unreal Engine 3 loads everything before it is needed, so it may take some time to build scripts and compile a project each time you make a change to the code, especially if you have a large number of references to content. There are ways around this, however, in case of dynamically loading an object, it loads everything during the runtime.

See also

Feel free to edit more parameters or even grab other actors to create archetypes from, such as weapons. With the assets UDK provides, you can create something such as a rocket launcher that fires pulse beams and has a particle emitter from the shock rifle.

Creating a subarchetype from an archetype

An archetype is a set of property values which will be assigned to a newly created object, a template. There may be a time when you want to extend from those values and create another template with similar properties. The March 2012 update of UDK Epic allowed that to happen with the addition of subarchetypes.

Getting ready

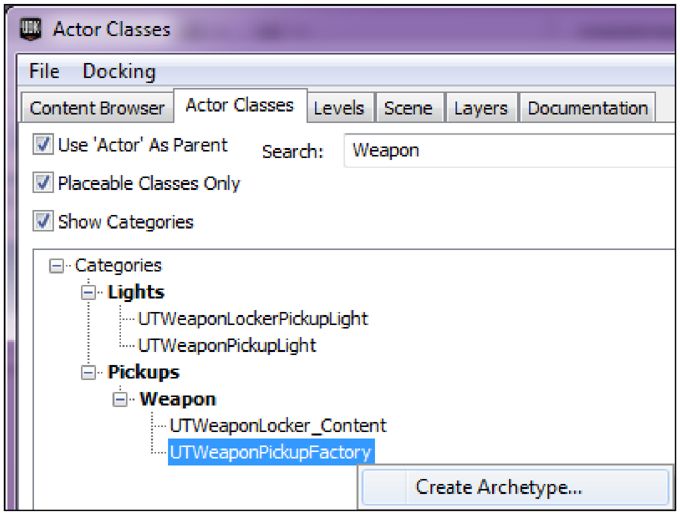
We're going to create a subarchetype from a weapon pickup factory, so the only thing you'll need to get started is to open up a fresh map within the editor.

How to do it...

Think of it as extending from a class, just as we would do with an IDE, but instead we are extending from an actor from within the editor. To do this we'll need to go back to our **Actor Classes Browser,** which is necessary whenever creating archetypes of any sort.

1. Open the **Actor Classes** tab and in the **Search** bar type `Weapon` to bring up the list of applicable actors.

2. Right-click on the actor marked `UTWeaponPickupFactory` and then left-click on **Create Archetype...** to bring up the dialog box for storing your content.

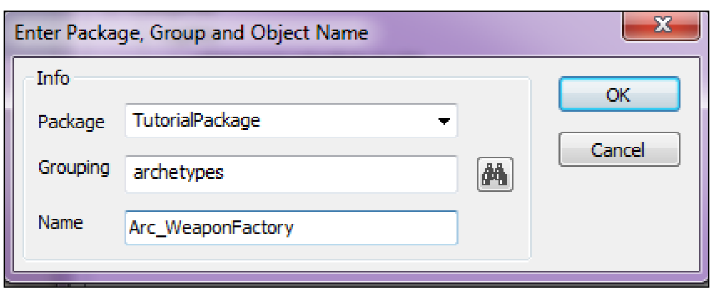


3. Enter the following information in the dialog box:

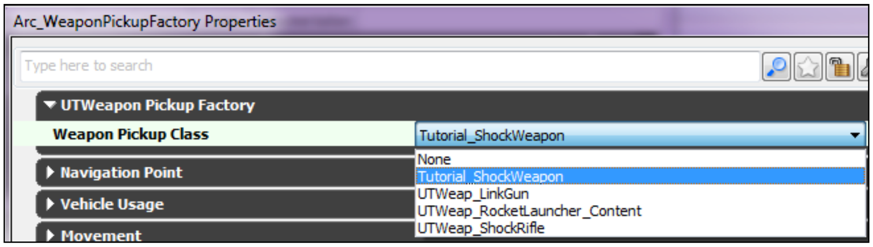
   Package: `TutorialPackage`

   Grouping: `archetypes`

   Name: `Arc_WeaponFactory`

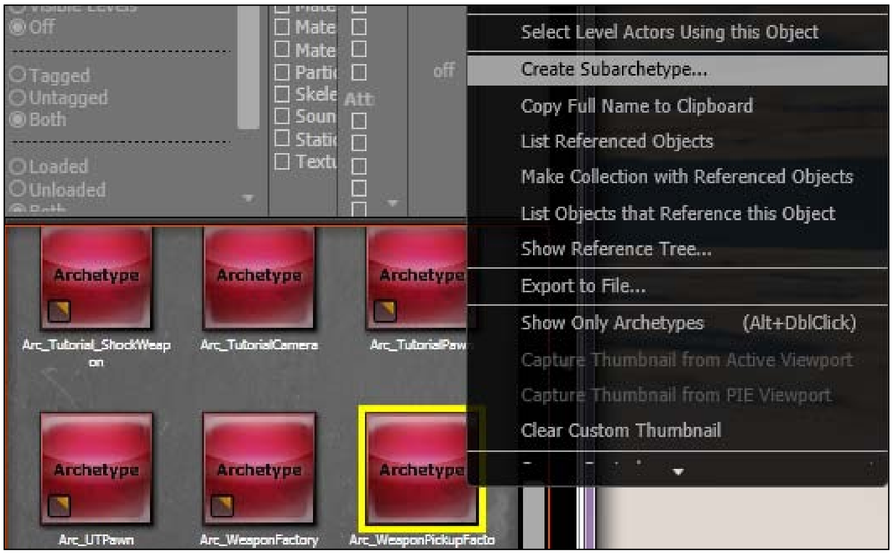


4. Go back to the content browser and left-click on the newly created archetype, then press *F4* to bring up its properties. We're going to keep things simple and only change one property here, and that is which weapon this pickup factory spawns.



5. Under **UTWeapon Pickup Factory** | **Weapon Pickup Class**, scroll down to `UTWeap_Shockrifle` and left-click the weapon to select it. This pickup factory will now spawn the shock rifle.
6. To test it out, drag-and-drop your archetype into the map and press the **PIE** button to play. When you run towards the pickup now, you'll see that it has a shock rifle floating above it. Run over it to pick it up.

7. Time to create our subarchetype. Back in the content browser, right-click on your archetype. One of the options that appears should read **Create Subarchetype...**. Left-click on that option.

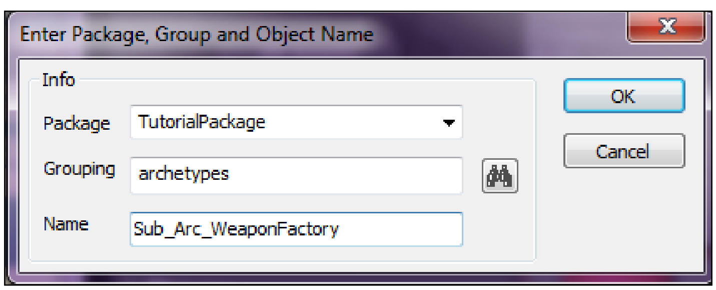


8. Another dialog box will appear. Enter the following information:

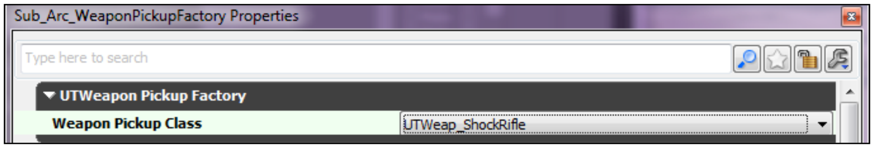
   Package: `TutorialPackage`

   Grouping: `archetypes`

   Name: `Sub_Arc_WeaponFactory`



9. Your new subarchetype should appear in the content browser. Left-click on it, then hit *F4* again to bring up its properties. You'll see that the **Weapon Pickup Class** is already set on the shock rifle!



How it works...

Subarchetypes inherit the properties and values of their parent archetype, just as classes receive all of the properties and values from their parent classes. This allows us to place a number of actors onto a map that share common characteristics.

We use archetypes instead of just code, as it allows our actors to be easily manipulated within the editor.

3
Scripting a Camera System

In this chapter, we will be covering the following recipes:

- Configuring the engine and editor for a custom camera
- Writing the `TutorialCamera` class
- Camera properties and archetypes
- Creating a first person camera
- Creating a third person camera
- Creating a side-scrolling camera
- Creating a top-down camera

Introduction

Cameras in UDK are an essential part of gameplay. They can simultaneously be one of the most frustrating yet rewarding things to program, as once they are working correctly they can completely change a player's experience, because you control their window to the world.

So with that, let's talk about cameras.

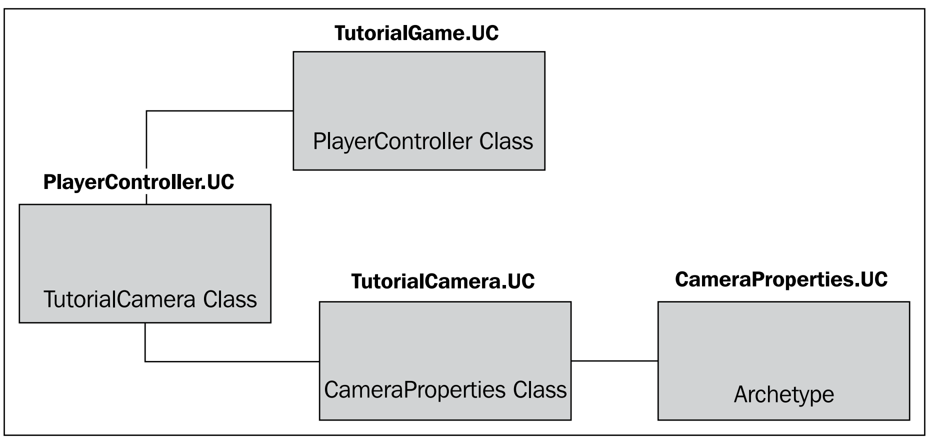
Understanding the camera

The camera system for UDK is comprised of three key classes: `Camera`, `Pawn`, and `PlayerController`. All of these classes interact to control the rotation, position, and special effects that should be applied to the player's camera during the course of a game.

A reference to the `Camera` class is being held in the `PlayerController` class, as well as the `Pawn` class being controlled. The input from the player is received from the `PlayerController` class and used to update the positions and rotation of the pawn it is controlling. The `Camera` class passes its update to the `Pawn` class, which in turn updates the rotation and position back to the camera.

By altering one or more of these classes and the way they interact, the player's camera can be set to show the world to the player using any perspective. By default, the player's camera uses a first-person perspective with the option to toggle it to a third-person over-the-shoulder perspective. We're going to create our own camera system which will allow us to do all of that, and more.

A player's view into the world is determined by the `Camera` class. The camera's rotation and position determines the viewpoint from which the scene is rendered when displayed on screen. Additionally, the `Camera` class holds properties for controlling the way the world is seen, such as setting the aspect ratio, field of view, and so on.



Cameras also have special effects that can be applied to the `Camera` class, thus allowing things such as post processing, camera animations, lens effects, and camera modifiers. While we won't discuss those special effects here, I, at least, wanted to bring them to your attention.

The PlayerController class

Responsibility for translating player input into game actions, such as moving a pawn or controlling the camera, is passed off to the `PlayerController` class. It's not unusual for the player controller's rotation to drive the camera rotation, although there are other ways to handle this, such as having our target pawn implement `CalcCamera`. We will not be taking that approach, however. There are some negatives associated with this path, including the loss of some functionality, such as camera animations and post processing effects.

When creating new camera perspectives, it may be necessary to update or override some functionality within the `PlayerController` class, as the player's input is translated into the orientation and the movement of the pawn can differ with each type of camera and perspective.

Now how exactly does this tie into the pawn? The player's physical representation in the world is not only handled by the pawn, but it can also be responsible for controlling the position and rotation of the player's camera. By overriding certain functions, you can create new camera perspectives. This is exactly what we're going to do with our `Camera` and `PlayerController` classes.

Configuring the engine and editor for a custom camera

All of our recipes will require a new custom `GametTypegametype` class to tell UDK to use our new `Pawn` and `PlayerController` classes.

Getting ready

We'll be using the same game type and player controller for all of these cameras, so we'll begin this chapter's recipes by explaining them here. Begin this lesson by extending our game from `UTGame`:

```
class TutorialGame extends UTGame;

defaultproperties
{
  PlayerControllerClass=class'Tutorial.TutorialPlayerController'
  DefaultPawnClass=class'Tutorial.TutorialPawn'
  DefaultInventory(0)=class'UTWeap_ShockRifle'
}
```

We set the default properties which include our new custom `TutorialPlayerController` class and `TutorialPawn` class. I choose to use `UTWeap_ShockRifle` as my weapon of choice, but you can place whatever you'd like here.

We'll need to modify the `DefaultGameEngine.ini` and `DefaultGame.ini` files as well, to tell the editor and engine to use the new game type as the default. These files can be found in your directory under the path, `UDKGame/Config`.

```
DefaultGameEngine.ini

[UnrealEd.EditorEngine]
+EditPackages=UTGame
+EditPackages=UTGameContent
```

```
+EditPackages=Tutorial

DefaultGame.ini

[Engine.GameInfo]
DefaultGame=Tutorial.TutorialGame
DefaultServerGame=Tutorial.TutorialGame
```

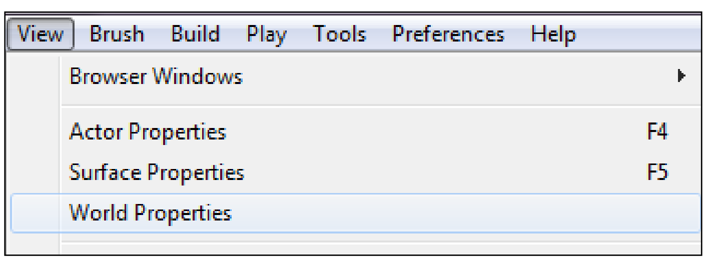
With that out of the way, we can finally get to make our prototype camera system work within the editor.

> It may be necessary to delete your `UDKGame.ini` and `UDKEngine.ini` files after modifying the default ones, as we have done here. Our game runs off of the UDK versions; if they're still there, they'll be used without our modifications.

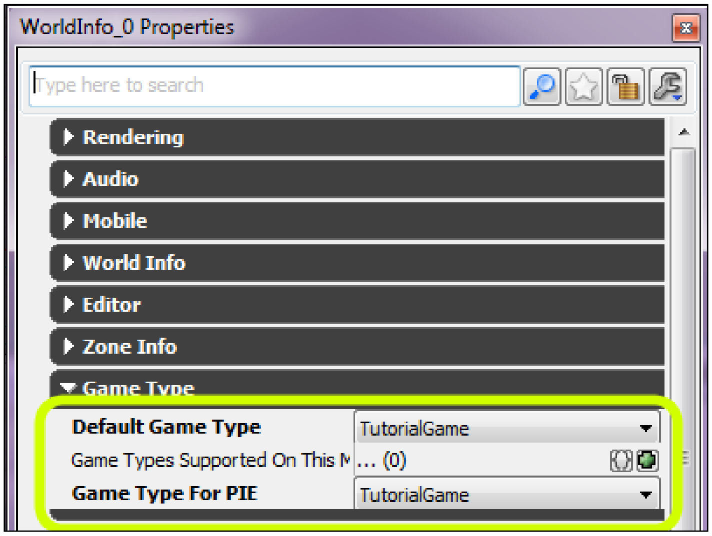
How to do it...

With our game configured correctly, we need to make sure we have the correct map and game type loaded when we start the UDK editor. We really only have to change the game type from a menu, so that UDK knows to look for our custom game, instead of its default setting of Unreal Tournament as explained in the following steps:

1. Load the UDK editor and open our `DefaultMap.udk`.
2. Afterwards, left-click on the **View** tab at the top-left corner, which will bring down its contents.
3. Scroll down to **World Properties**, and left-click on that.



4. The **WorldInfo_0 Properties** dialog box should appear. Look for the **Game Type** tab, which will contain our **Default Game Type** and **Game Type for PIE** (Play In Editor).



5. Change **Default Game Type** and **Game Type for PIE** to **TutorialGame**.

> Now when we hit the **PIE** or **Play In Viewport** buttons, our game will load our **TutorialGame**, which as we learned before, is what sets the elements we need to have loaded for our camera, pawn, and player controller in motion.

How it works...

When changing the default gametype and adding packages to UDK, we'll always need to configure the `.ini` files accordingly. These `.ini`, or standard configuration files, are what the engine checks before compiling our projects, and are the ones which instruct the engine to look for specific packages.

Once we've made those changes, so that it looks for our new game type and package of classes, we just need to change the game type for our specific map.

Writing the TutorialCamera class

With our game type in place, we now need to write the code for our `TutorialCamera` and `TutorialCameraProperties` classes. The properties class includes the variables we will expose to the UDK editor. Coming back around to the archetypes we spoke about earlier, our camera will now be made into one, which is what allows its properties (`TutorialCameraProperties.uc`) to be manipulated in the editor.

Getting ready

We'll need to launch our IDE of choice here, and create the `TutorialCamera` class by creating a file called `TutorialCamera.uc` class in our `Classes` folder (`C:\UDK\July\Development\Src\Tutorial\Classes`).

How to do it...

The first thing we'll need is a `Camera` class to store all of our variables and functions in. We'll use these throughout the rest of our tutorials, and they'll hold the values for manipulating how the camera interacts with the environment. We can do this by creating one new class.

1. We'll begin by creating our `TutorialCamera` class. As UDK's camera class already offers much of what we'll need for our own camera, we'll just extend from theirs. We'll also need to have access to our `TutorialCameraProperties` class, and we do so by declaring a variable for it.

   ```
   class TutorialCamera extends Camera;

   // Reference to the camera properties
   var const TutorialCameraProperties CameraProperties;
   ```

2. Immediately following that, we'll include some code which will be written to our log file when the camera is first initialized. This allows us to verify that our camera is actually being called.

   ```
   /** Lets us know that the class is being called, for debugging
   purposes */
   simulated event PostBeginPlay()
   {
       super.PostBeginPlay();
       'Log("Tutorial Camera up");
   }
   ```

I cannot overstate the importance of using the `'Log` function when building code. It will save you an enormous amount of frustration, especially when debugging, as you will know immediately whether or not your code you're looking at had ever been called.

We place this in our `PostBeginPlay()` method as it will be one of the first things executed before a class is fully initialized. The `super.PostBeginPlay()` method above our `'Log` function indicates that we want our parent class' `PostBeginPlay` method to be called first, then execute ours, wherein our parent class is `Camera`.

3. Now we've got to build the meat and potatoes of our class with our `UpdateViewTarget` method, as well as declare our local variables that we'll be using:

```
/*****************************************************************
Query ViewTarget and outputs Point Of View.
 *
 * @param    OutVT        ViewTarget to use.
 * @param    DeltaTime    Delta Time since last camera
                          update in seconds
*****************************************************************/
functionUpdateViewTarget
(out TViewTargetOutVT, float DeltaTime)
{
  local Pawn   Pawn;
  local Vector V, PotentialCameraLocation, HitLocation,
                HitNormal;
  local Actor HitActor;
```

4. The next `if` statement is used for blending what the camera sees as we begin to move around:

```
  /** If there is an interpolation, don't update outgoing
       viewtarget */
  if (PendingViewTarget.Target!=None&&
  OutVT==ViewTarget&&BlendParams.bLockOutgoing)
  {
  return;
   }
```

`Pawn` refers to our game pawn. We'll have a number of vectors to define, but it is simply if we break it down:

- The first one, `V`, is simply a placeholder and of no use to us. The `GetActorEyesViewPoint` method required a vector as one of its parameters and we had to put something in there, so we used `V`. When a vector's properties are not defined it simply defaults to X=0, Y=0, Z=0.

- `PotentialCameraLocation` is where we actually want the camera to be.
- The `HitLocation` and `HitNormal` variables are used for our trace. This comes into play when our camera bumps into a wall, and rather than clipping through the wall, allows the camera to be offset, so that it still displays our pawn without interrupting the gameplay experience.
- Finally, the `HitActor` variable declares what we've just hit when doing our trace.

5. If we know which pawn socket name we'd like to use, then declare it, otherwise we'll use the pawn's eyes as our default starting point for what we see (that is, we draw our viewpoint from where the weapon is pointing instead, using the `WeaponSocket` socket):

```
/** Our pawn will be where we are grabbing our
perspective from */
Pawn = Pawn(OutVT.Target);

/** If our pawn is alive*/
if (Pawn != None)
{

  /** Start the camera location from a valid socket name,
           if set correctly in the camera properties */
  // (i.e. WeaponSocket)
  if (Pawn.Mesh!=None&&Pawn.Mesh.GetSocketByName
  (CameraProperties.PawnSocketName)!=None)
  {
    Pawn.Mesh.GetSocketWorldLocationAndRotation
    (CameraProperties.PawnSocketName, OutVT.POV.Location,
    OutVT.POV.Rotation);
  }
  /** Otherwise grab it from the target eye view point */
  else
  {
    OutVT.Target.GetActorEyesViewPoint
    (OutVT.POV.Location, OutVT.POV.Rotation);
  }
```

> The weapon socket is one of the properties we'll declare in the `TutorialCameraProperties` class, so we won't have to worry about hardcoding it here just yet.

6. We generally want to use the rotation of our target, in this case our pawn, so that we view the world as it would. If that's the case, we'll want to have this turned on.

```
/** If the camera properties forces the camera to always
use the target rotation, then extract it now */
    if (CameraProperties.UseTargetRotation)
    {
      OutVT.Target.GetActorEyesViewPoint(V,
      OutVT.POV.Rotation);
    }

    //CameraProperties.UseTargetRotation = false;
```

> This is another Boolean, which we can select to have on or off in the `TutorialCameraProperties` class.

7. We will want to offset the rotation of our camera from its default socket location. This is what allows us to create first, third, and virtually any other camera we'd like to use. We'll also need to do the applicable math for the calculation.

```
/** Offset the camera */
  OutVT.POV.Rotation+=CameraProperties.CameraRotationOffset;

    /** Do the math for the potential camera location */
PotentialCameraLocation=OutVT.POV.Location+
(CameraProperties.CameraOffset>>OutVT.POV.Rotation);
```

> Technical editor William Gaul described the whole process as follows:
>
> Put simply, A >> B rotates vector A in the way described by rotator B. One Unreal rotation unit is 32,768/Pi radians, however people tend to think in degrees. For reference, check the following:
>
> - 65,536 = 360 degrees
> - 32,768 = 180 degrees
> - 16,384 = 90 degrees
> - 8,192 = 45 degrees
> - 182 = 1 degree
>
> What occurs here is a coordinate system transformation. We take the local `CameraOffset` and adjust it to global space so it can be applied to the out vector.

8. This is where our trace will come into play. We need to check and see if our potential camera location will work, meaning that it won't put us in a wall. If we do run into a collision issue, the camera will automatically offset itself by the value of the normal of what we hit. This is perhaps the most complicated part of the code, as it involves so much math:

```
/** Draw a trace to see if the potential camera location will
work*/
    HitActor=Trace(HitLocation, HitNormal,
    PotentialCameraLocation, OutVT.POV.Location, true,,,
    TRACEFLAG_BULLET);

    /** Will the trace hit world geometry? If so then use
        the hit location and offset it by the hit normal */
    if (HitActor!=None&&HitActor.bWorldGeometry)
    {
      OutVT.POV.Location=HitLocation+HitNormal*16.f;
    }
    else
    {
      OutVT.POV.Location=PotentialCameraLocation;
    }
```

9. Our last piece of code makes us declare the archetype that we will be using to access our camera's properties. We'll cover exactly where we get this file path at the end of our `TutorialCameraProperties` tutorial.

```
defaultproperties
{
  /** This sets our camera to use the settings we create in the
editor */
CameraProperties=TutorialCameraProperties'TutorialPackage.
archetypes.Arc_TutorialCamera'
}
```

How it works...

Flipping back and forth between an IDE and the editor to determine the exact numbers for variables such as offset, rotation, and distance for our camera system can be a frustrating and time consuming. By scripting one prototyping camera system from our IDE, we will be allowed to change values on the fly from within the UDK editor. This way, we can not only save valuable time and avoid frustration, but also quickly prototype new camera systems and see how our current ones work.

We've created a camera system and exposed certain properties to the UDK editor to avoid cluttering our editor, while at the same time allowing for easy manipulation of important properties, such as rotation and vector.

Camera properties and archetypes

With our `Camera` class in place, we now need to create a class for our publicly exposed properties, as well as the archetype that we'll physically interact with, inside the UDK editor.

Getting ready

Open your IDE and prepare to create a new class.

How to do it...

Exposing properties to the editor can be a far easier way to make tweaks and changes to actors like our camera. Without this archetype we are about to build, we'd have to manually change the variable within our IDE, compile the project, then view our results in the editor. We plan on making the life of your level designer much easier with this simple fix.

In the following recipe we'll need to create a new class, in addition to an archetype within the UDK editor, so that our level designers can reference our class without ever having to open up an IDE:

1. Create `TutorialCameraProperties.uc` in our `Classes` folder (`C:\UDK\July\Development\Src\Tutorial\Classes`).
2. As we're simply making an archetype, we'll be extending our class from the most basic of all UDK classes, `Object`.

   ```
   class TutorialCameraProperties extends Object
   ```

3. Moreover, we want to keep our editor as clean as possible, so we'll be hiding all of the camera's properties with this next step.

   ```
   /** We don't want to clutter up the editor, so hide Object
       categories */
   HideCategories(Object);
   ```

4. Moving on, we'll want to expose only the variables we've created in our `TutorialCamera` class. This allows us to easily and quickly make changes to nearly anything we would want to change for our camera. All of these variables were previously defined in our `TutorialCamera` class.

   ```
   /** Camera offset to apply */
   var(Camera) const Vector CameraOffset;
   /** Camera rotational offset to apply */
   var(Camera) const Rotator CameraRotationOffset;
   /** Pawn socket to attach the camera to */
   var(Camera) const Name PawnSocketName;
   /** If true, then always use the target rotation */
   var(Camera) constboolUseTargetRotation;
   ```

> The `PawnSocketName` variable is any socket located on the pawn. You can even create your own! For the most part though, we'll be sticking with either `WeaponSocket` or `HeadShotGoreSocket` (pawn's eyes).

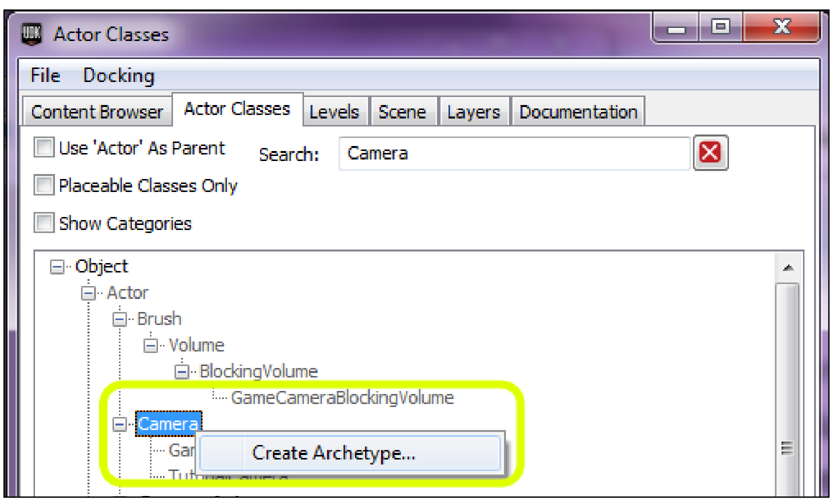
5. We don't have information to place in our default properties, but we're required to have it regardless.

```
defaultproperties
{
}
```

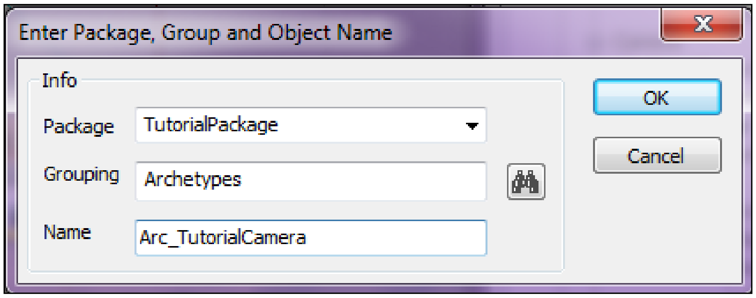
6. Our next goal is to get the archetype created within the UDK editor. Launch the editor and open your **Actor Browser**.

> I like to add the `-log` parameter to my editor, so that I can see the debug screen as I'm making changes. To do the same, have your launch path look like the following:
>
> `C:\UDK\July|Binaries\UDKLift editor -log`

7. Make sure that **Use 'Actor' As Parent**, **Placeable Classes Only**, and **Show Categories** are not checked, otherwise our camera will not appear in our search. In the search field, enter `Camera`, and you will see it appear.



8. Right-click on **Camera** to display the **Create Archetype** pop-up. Left-click to bring up the dialog box for archetype options.

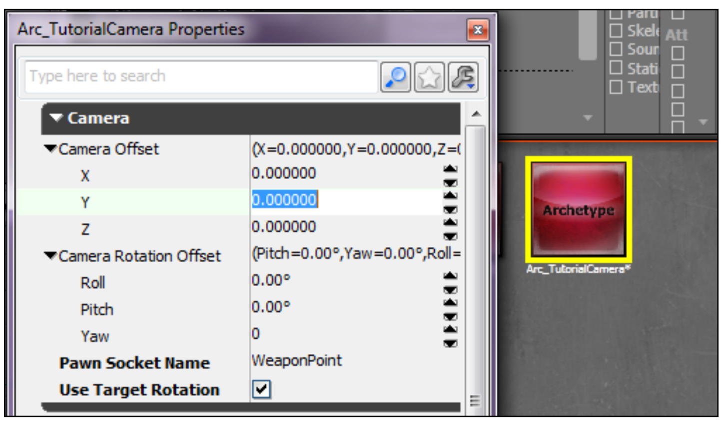


9. Yours should now read as the following:

   Package: `Tutorial Package`

   Grouping: `Archetypes`

   Name: `Arc_TutorialCamera`



10. With that completed, your newly created archetype will be available to you in the content browser. Left-clicking on it will cause the dialog box for the camera properties to display, which illustrates all of the variables we exposed in our `TutorialCameraProperty` class.
11. Hit the **PIE** or **Play in Viewport** button in your editor (*F8* works too), and the pawn will spawn. Adjusting any of the values in your archetype will now yield new results for your camera!

Once you've hardcoded some values that seem to work for a new camera system (that is, third person), you can then create a Boolean in these properties that allow you to quickly switch it on and off with the touch of a button.

Add the following to the variables section of your `TutorialCameraProperties` class:

```
/** If true, then use the third person settings */
var(Camera) constboolUseThirdPerson;

Afterward, add the following to your TutorialCamera class, just
beneath where we've stated If (Pawn != None):

  /** Be sure to set the values in your camera within the editor
      to 0! Otherwise this will not work correctly! If the
      camera properties forces the camera into third person,
      then extract it now */
    if (CameraProperties.UseThirdPerson)
    {
      OutVT.POV.Location.X += 120;
      OutVT.POV.Location.Y += 50;
      OutVT.POV.Location.Z += 35;

      OutVT.POV.Rotation.Roll += 0.0f;
      OutVT.POV.Rotation.Pitch += 0.0f;
      OutVT.POV.Rotation.Yaw += 10.0f;
            // Hide the pawn's "first person" weapon
            if(Pawn.Weapon != none)
            {
                Pawn.Weapon.SetHidden(true);
            }
    }
```

Selecting the checkbox in your editor within the game will now instantly transform your camera into a third person view!

How it works...

Our camera properties class is simply an object that attaches to our `Camera` class and allows us to alter our camera's properties from within the editor. We've created an archetype from our `CameraProperties` class, so that we can reference it from UScript and attach it to our `Camera` class through our default properties.

Attaching our `CameraProperties` archetype to our camera allows us to make changes on the fly within the editor, and remove the frustration of needing to go back and forth between our IDE, recompiling, and then the editor to see any changes take place.

There's more...

In the default properties for our `Camera` class, we listed the path to our camera archetype. Now that we've created our archetype, we can fill in our default property with the proper location:

1. Right-click on `Arc_TutorialCamera` in your **Content Browser** and scroll down to **Copy full name to clipboard**.
2. With the name copied, go to `TutorialCamera.uc` and in default properties our `TutorialCamera` will now grab its values from what we set in the editor, and your class should look like the following:

```
defaultproperties
{
/** This sets our camera to use the settings we create in
    the editor */
CameraProperties=TutorialCameraProperties'TutorialPackage.
archetypes.Arc_TutorialCamera'
}
```

See also

You can add sockets to your pawn at any time to make offsetting your camera even easier. The following is an image illustrating the five sockets available to our pawn by default:

Creating a first person camera

While a first person camera comes standard with UDK, we've crafted a camera system that is modular, and allows us to easily adjust our perspective while still avoiding clipping through objects.

We'll take all that we've learned from our previous lessons and apply it to this one. We'll be hardcoding a first person camera, based on the property values I've found to be consistent with what we are looking for.

Getting ready

Our next few lessons will all require one similar change, all of which will occur with `TutorialPlayerCotroller` class. Under `defaultproperties`, we'll need to change the following:

```
CameraClass = Class'Name_Of_Camera_Class'
```

So, for this tutorial it should read as follows:

```
CameraClass = Class'FirstPersonCam'
```

With this completed, our player controller will now ignore any change we make to the `CameraProperties` archetype we've created and instead use the values we write in `FirstPersonCam.uc`.

Moreover, our new camera system will look very similar to our tutorial camera, so I won't go over everything again in much detail.

Rather than rely on the values we've entered in our archetype, our camera will now be built around hardcoded numbers which we've deemed to best suit our needs. Of course, you could alter them at any time through code.

How to do it...

UDK comes with a first person camera right out of the box, but we want to create one that fits in with our modular camera system. We'll be creating a new class, which is very similar to our previous camera class in many aspects. Additionally, we'll need to bind this new camera class to our player controller so that we can actually utilize it, which is explained as follows:

1. We'll start by creating a new class called `FirstPersonCam`, extending from `Camera`, just as we did with our previous camera system.

   ```
   class FirstPersonCam extends Camera;
   ```

2. Follow that up by declaring the two new variables that we'll need, `CamOffset`, which is of type `Vector`, and `CamOffsetRotation`, which is of type `Rotator`.

> Instead of relying on our archetype for values, we'll write them in default properties and store them in these two variables.

```
/** Hardcoded vector offset we will use, rather than tweaking
values in the editor's CameraProperties */
varconst Vector CamOffset;
/** Hardcoded rotator offset we will use, rather than tweaking
values in the editor's CameraProperties */
varconst Rotator CamOffsetRotation;
```

3. Our `PostBeginPlay` function is the same as before. However, now that we're using a first person view, we'll want to hide the third person mesh, otherwise it will constantly be clipping into the view of our camera. Therefore, we add this bit of code to our `UpdateViewTarget` function:

```
Pawn = Pawn(OutVT.Target);
    if (Pawn != None)
    {
    /** To hide the third person mesh */
        Pawn.SetHidden(true);

/*********************************************************
 * If you know the name of the bone socket you want to use,
 * then replace 'WeaponPoint' with yours.
 * Otherwise, just use the Pawn's eye view point as your
 * starting point.
*********************************************************/
```

4. This next part is new. We now want to declare where our point of view will begin by using a socket. For the third person and over the shoulder views, the weapon works well; but for first person, the pawn's head works best. The offset values, we declare later, will all be from the socket point we declare here. If you aren't sure of which socket to use, then the camera will start from our pawn's eyes by default.

```
/** socket not found, use the other way of updating vectors */
if(Pawn.Mesh.GetSocketWorldLocationAndRotation('HeadShotGoreSock
et', OutVT.POV.Location, OutVT.POV.Rotation) == false)
{
  /** Start the camera location from the target eye
  view point */
  OutVT.Target.GetActorEyesViewPoint
  (OutVT.POV.Location, OutVT.POV.Rotation);
}
```

5. This part is slightly different from what we've seen before. Rather than offer the option of choosing to use the target's rotation or not, we're just declaring that we will indeed use the target's (pawn's) rotation.

```
/** Force the camera to use the target's rotation */
OutVT.Target.GetActorEyesViewPoint(V,
OutVT.POV.Rotation);
/** Add the camera offset */
OutVT.POV.Rotation += CamOffsetRotation;

/** Math for the potential camera location */
PotentialCameraLocation = OutVT.POV.Location +
(CamOffset>>OutVT.POV.Rotation);

/** Draw a trace to see if the potential camera location will work
*/
HitActor = Trace(HitLocation, HitNormal,
PotentialCameraLocation, OutVT.POV.Location, true,,,
TRACEFLAG_BULLET);

/** Will the trace hit world geometry? If so then use the hit
location and offset it by the hit normal */
if (HitActor != None && HitActor.bWorldGeometry)
{
  OutVT.POV.Location = HitLocation + HitNormal * 16.f;
}
else
{
  OutVT.POV.Location = PotentialCameraLocation;
}
```

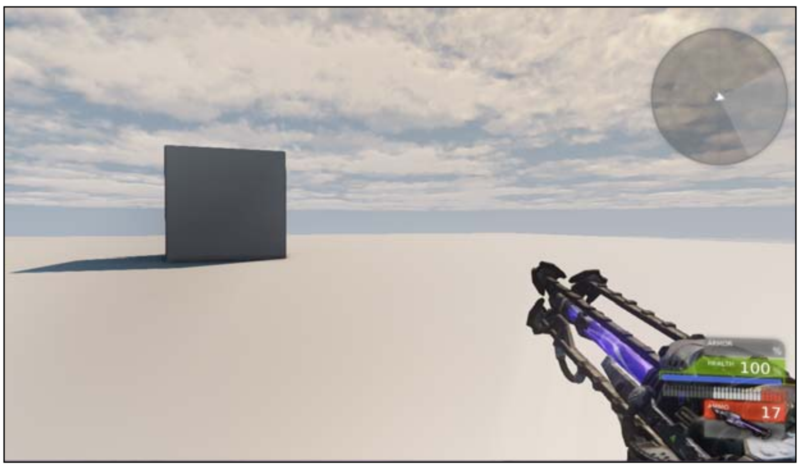
6. Our hardcoded values for our camera's offset and rotation will be defined in the class' default properties, `CamOffsetRotation=(Pitch=16384, Roll=0, Yaw=0),` which is the Unreal unity equivalent to 90 degrees.

```
/** Hardcoded vector & rotator values for our camera */
defaultproperties
{
    CamOffset=(x=+10,y=0,z=0)
    CamOffsetRotation=(Pitch=0, Roll=0, Yaw=0)
}
```

[I've included a reference as well, to give you a better understanding of how the Unreal Engine converts degree rotations into Unreal rotator units. I've also added `10` to our `x` value, because if we don't, then we're able to see the back of the gun's model.]

So, if we wanted our pitch to be turned up 90 degrees, we would write, `CamOffsetRotation=(Pitch=16384, Roll=0, Yaw=0).`

With our class completed, all you need to do is compile the project, launch the editor, and you'll have a first person camera!



How it works...

There was no need to reference our `CameraProperties` archetype in this example, as we've hardcoded our values based on what worked with our easy tutorial camera.

To start things off, we needed our player controller to use our first person camera, so we made the appropriate change in our default properties block.

Our `FirstPersonCam` extends our `Camera` class, and allows for more freedom down the road, as opposed to using `CalcCamera`. While our method requires a bit more work, we are now free to use camera animations and post processing effects.

For the most part, the code is very similar to what we had in our `TutorialCamera` class, but instead of variables for things like camera offset and rotation being read from our `CameraProperties` archetype, we are hardcoding them in the default properties block of our `FirstPersonCam` class.

Creating a third person camera

UDK now comes with a third person camera built into the kit, but we'd still prefer to use our own modular camera system. Epic's popular *Gears of War* franchise uses this camera style, and then zooms into an over-the-shoulder view when sprinting. We'll cover more of that in our next recipe.

Getting ready

As with our previous recipe, we'll require one similar change in the `TutorialPlayerCotroller` class. Under `defaultproperties`, we'll need to change it so that it reads as the following:

```
CameraClass = Class'ThirdPersonCam'
```

Our code looks nearly identical to that of our first person camera, minus a few simple changes that I'll highlight. Most notably, we're looking to hide our first person mesh, so that only our third person mesh is exposed to the camera.

In the end, you may notice that your projectiles are not firing from the correct mesh. Not to worry, this isn't an issue with the camera, but a simple change that we need to make in the weapon class, which we'll highlight in the chapter about weapons.

If your game is using our `Tut_Pawn`, `Tutorial_ShockWeapon`, and `Tut_Attachment_Shockweapon` classes, then you'll be fine, and you can see the exact functions that allow our projectiles to fire from the proper location.

How to do it...

Rather than rely on the values we've entered in our archetype, our camera will now be built around hardcoded numbers which we've deemed to best suit our needs, just as we did with the first person camera. For this recipe we'll be creating a new class for our third person camera, and then binding it to our player controller:

1. Create a new class called `ThirdPersonCam` and have it extend from the `Camera` class:

   ```
   class ThirdPersonCam extends Camera
   ```

2. Remember when we hid our pawn from view before? Well this time we're going to allow the pawn to be shown, but hide our first person weapon:

```
Pawn = Pawn(OutVT.Target);
    if (Pawn != None)
    {
      /** Hide the pawn's "extra" weapon */
      if(Pawn.Weapon != none)
      {
        Pawn.Weapon.SetHidden(true);
      }
    }
```

   There are a number of reasons as to why we do this, most notably, due to the fact that designers want to limit the detail of the weapons and pawns as the camera gets further away. There's no reason to have a high poly model present if the player never sees it. This allows better graphical efficiency as the distance between the pawn or weapon increases.

3. In our first person camera we used the pawn's eyes as the socket point from which our offset would be based. This time however, we're going to be using the weapon's socket, simply titled `WeaponSocket`.

```
/***********************************************************
* If you know the name of the bone socket you want to use,
* then replace 'WeaponPoint' with yours.
* Otherwise, just use the Pawn's eye view point as your
starting point.
**********************************************************/
/*socket not found, use the other way of updating vectors
  */
if (Pawn.Mesh.GetSocketWorldLocationAndRotation
('WeaponPoint',OutVT.POV.Location, OutVT.POV.Rotation) == false)
{
  /*Start the camera location from the target eye
  view point */
  OutVT.Target.GetActorEyesViewPoint
  (OutVT.POV.Location, OutVT.POV.Rotation);
}
```

4. Where the magic happens is in the default properties. We're shifting the camera so that it gives us the third person view we're looking for:

```
/** Hardcoded vector & rotator values for our camera */
defaultproperties
{
    CamOffset=(x=-100,y=15,z=20)
    CamOffsetRotation=(Pitch=-2048)
}
```

[You may notice that we don't have values for `Roll` and `Yaw`. That's because any values which aren't declared in a Rotator are assumed to be zero.]

How it works...

There was no need to reference our `CameraProperties` archetype in this example, as we've hardcoded our values based on what worked with our easy tutorial camera.

To start things off, we needed our player controller to use our third person camera, so we made the appropriate change in our default properties block.

For the most part, the code is nearly identical to what we had in our `FirstPersonCam` class, except for the hardcoded values declared in our default properties block, which are now adjusted to work for a third person view.

Creating a side-scrolling camera

Side-scrolling games have been a hit for decades, and some of gaming's largest franchises have taken advantage of it. While we don't often see games using UDK take advantage of the side-scrolling perspective, one high profile title published by Epic, *Shadow Complex* certainly did, although with a bit more of a pulled back, Metroidvania style about it. For those not familiar with the term Metroidvania, it is a 2D side-scroller style of game with an emphasis on a non-linear, exploratory action-adventure structure; it inherits its name from the Metroid and Castlevaniaseries.

We'll be hardcoding our values again. Much of what you'll see next is similar to the code found in the third person camera code.

We'll also have to make a few changes to the `PlayerController` class for a number of reasons. Specifically, we want our pawn to only be able to move forward and back, thereby removing the ability to strafe left and right. Moreover, our system requires that the right side of the screen is always considered forward.

For this purpose, we will want to create a new player controller class, called `TutorialPlayerControllSSC`.

Just as we did with the third person camera, we'll want the first person weapon mesh to remain hidden, and continue to expose the third person weapon and pawn meshes to the camera.

Getting ready

We're going to make a whole new player controller class for this tutorial, as we need to add a new function to it. Create a new class called `TutorialPlayerControllerSSC` (side-scrolling camera) and have it extend from the `PlayerController` class:

```
class TutorialPlayerControllerSSC extends PlayerController;
```

How to do it...

With our new `TutoriaPlayerControlerSSC` class made, we can begin filling it with the functions we need:

1. For now, add this code, which you should already be familiar with by now, as it appears in our `TutorialPlayerController.uc` class:

```
/** Lets us know that the class is being called, for
debugging purposes */
simulated event PostBeginPlay()
{
```

```
    super.PostBeginPlay();
    'Log("TutorialPlayerControllerSSC up");
}

/**********************************************************
*** FOR ALL CAMERAS ***
* Rotator where the pawn will be aiming the weapon.
* Will be different, depending on which camera system we
* are using.
**********************************************************/
function Rotator GetAdjustedAimFor
(Weapon W, vector StartFireLoc)
{
  returnPawn.GetBaseAimRotation();
}
```

2. The next function we add allows our projectiles to fire in the correct direction. We are overriding `UpdateRotation`, so that the projectiles use the pawn's rotation and not the camera's rotation when firing. Without this, you'll notice that projectiles always fire in the same direction, and towards the camera. The following is that bit of code:

```
/**************************************************************
* Forces the weapon to fire in the direction the pawn is
facing
**************************************************************/
functionUpdateRotation( float DeltaTime )
{
  local Rotator DeltaRot, newRotation, ViewRotation;

  ViewRotation = Rotation;

  /** Calculate Delta to be applied on ViewRotation */
  DeltaRot.Pitch = PlayerInput.aLookUp;
  ProcessViewRotation( DeltaTime, ViewRotation, DeltaRot );
  SetRotation(ViewRotation);
  NewRotation = ViewRotation;

  if ( Pawn != None )
    Pawn.FaceRotation(NewRotation, deltatime);
}
```

3. This next bit seems long-winded, but it is essentially one function (really, a state) that we are overriding from `Pawn.uc`. It is identical to the one found in that class, except that we are altering this bit of code:

   ```
   /** The only change for the side scrolling camera to this
   function. Update acceleration - pawn can only move forward and
   back now */
   NewAccel.Y = -1 * PlayerInput.aStrafe * DeltaTime * 100 *
   PlayerInput.MoveForwardSpeed;

   /** Set to 0 to not allow movement on the X or Z axes */
   NewAccel.X = 0;
   NewAccel.Z = 0;
   ```

   This controls the movement of our pawn, based on the direction it is facing. Hitting the left and right (or *A* and *D*) keys on the keyboard will now force your pawn to move forward and back across the screen. Previously, you would have to hit the up and down keys to make the pawn move forward and back.

   Moreover, setting `NewAccel` to `0` on both the `X` and `Z` axes prevents the pawn from strafing left and right. With our plane of movement locked, we can now create a true side-scroller.

4. In the following code, we are overriding the state, `PlayerWalking`, found in the `PlayerController.uc` class. Our goal is to update the acceleration to allow only forward/back movement.

   ```
   /*****************************************************************
   * Player movement, overriding the one found in
   PlayerController.UC
   * We are updating the acceleration to allow for only
   forward/back movement
   *****************************************************************/
   state PlayerWalking
   {
     ignores SeePlayer, HearNoise, Bump;

     function PlayerMove(floatDeltaTime)
     {
     localvector   X,Y,Z, NewAccel;
     localeDoubleClickDir DoubleClickMove;
     localrotator OldRotation;
     localbool bSaveJump;
   ```

```
if( Pawn == None )
{
  GotoState('Dead');
}
  else
  {
    GetAxes(Pawn.Rotation,X,Y,Z);

    /** The only change for the side scrolling camera to
    this function. Update acceleration - pawn can only
    move forward and back now */
    NewAccel.Y=-1*PlayerInput.aStrafe*DeltaTime*
    100*PlayerInput.MoveForwardSpeed;

    /** Set to 0 to not allow movement on the X or Z axes
    */
    NewAccel.X=0;
    NewAccel.Z=0;

if(IsLocalPlayerController())
{
  AdjustPlayerWalkingMoveAccel(NewAccel);
}

DoubleClickMove=PlayerInput.CheckForDoubleClickMove
( DeltaTime/WorldInfo.TimeDilation );

/** Update rotation. */
OldRotation = Rotation;
UpdateRotation( DeltaTime );
bDoubleJump=false;

if( bPressedJump&&Pawn.CannotJumpNow())
{
  bSaveJump=true;
  bPressedJump=false;
}
else
   {
     bSaveJump=false;
   }

if( Role < ROLE_Authority ) // then save this move and
                               replicate it
{
```

```
      ReplicateMove(DeltaTime, NewAccel, DoubleClickMove,
      OldRotation - Rotation);
    }
      else
       {
         ProcessMove(DeltaTime, NewAccel, DoubleClickMove,
         OldRotation - Rotation);
       }
        bPressedJump=bSaveJump;
      }
    }
}
```

5. All that's left now is our default property, where we set the camera we'll be using, in this case, our side-scrolling camera:

```
defaultproperties
{
  CameraClass = Class'SideScrollingCam'
}
```

With our player controller configured, we can now move on to the actual camera itself.

6. Make a new class called `SideScrollingCamand`, and have it extend from `Camera`:

```
class SideScrollingCam extends Camera;
```

7. The rest of our code will be nearly identical to that found in our other camera classes. I did change the socket that the camera is based off of, however. Previously, we were using the `WeaponSocket` socket, which is where the pawn grips the weapon. This time I prefer to use the pawn's `HeadShotGoreSocket`, as I feel it gives me a better perspective of the world.

```
/** socket not found, use the other way of updating vectors */
if (Pawn.Mesh.GetSocketWorldLocationAndRotation
('HeadShotGoreSocket', OutVT.POV.Location,
OutVT.POV.Rotation) == false)
```

8. We don't want to see our pawn's first person weapon again, now that we're using a perspective outside of the pawn's eyes, so let's hide that as well. Place this code just above where you placed the code for step 7:

```
/** Hide the pawn's third person weapaon */
if(Pawn.Weapon != none)
{
  Pawn.Weapon.SetHidden(true);
}
```

Furthermore, it allows the camera to clip beneath the ground and turn it invisible. If we continue to use the `WeaponSocket` socket, the camera pulls itself in, towards the pawn, as we get closer to the ground. Use the one that best suits your needs.

9. Finally, we change our hardcoded default properties:

```
/** Hardcoded vector & rotator values for our camera */
defaultproperties
{
    CamOffset=(x=-340,y=70,z=0)
    CamOffsetRotation=(Yaw=53000)
}
```

Again, this is purely a matter of preference. Adjust it accordingly.

10. There is one final change to make, and that's in our `TutorialPawn` class. We need to change our `GetBaseAimRotation` method. This function is called by `GetAdjustedAimFor` in the player controller, which is the rotator for where the pawn will be aiming its shots. Essentially, we are telling the game to use the direction the pawn is facing for firing shots, and not the camera's. Add the following code:

```
/*****************************************************************
** USED FOR SIDE SCROLLING **
* Forces the weapon to use the pawn's direction for aiming,
and not the camera's.
* shots will be fired in the direction the gun is pointed.
Used by PlayerController
* Comment this out if you are not using the Side Scrolling
Camera.
* @return POVRot.
*****************************************************************/
simulated singular event Rotator GetBaseAimRotation()
{
  local rotator  POVRot;

  /** We simply use our rotation */
  POVRot = Rotation;

  /** If our Pitch is 0, then use RemoveViewPitch */
  if( POVRot.Pitch == 0 )
  {
    POVRot.Pitch = RemoteViewPitch << 8;
  }
  return POVRot;
}
```

11. Compile the project and take a look at your results!

How it works...

There was no need to reference our `CameraProperties` archetype in this example, as we've hardcoded our values based on what worked with our easy tutorial camera.

To start things off, we needed our player controller to use our side-scrolling camera, so we made the appropriate change in our default properties block. We also had to make a few changes to the `PlayerController` class. Specifically, we wanted our pawn to only be able to move forward and back, thereby removing the ability to strafe left and right. Moreover, our system requires that the right side of the screen is always considered forward.

We also had to override the `GetBaseAimRotation` function in our pawn class and `GetAdjustedAimFor` function in our player controller class. These changes tell the game to use the direction the pawn is facing for firing shots, and not the camera's.

Other than the player controller though, the code is nearly identical to what we had in our `FirstPersonCam` class, except for the hardcoded values declared in our default properties block, which are now adjusted to work for a side-scrolling view.

See also

Don't forget to go back to `TutorialGame.uc` and change your default properties so that the game is using our new `TutorialPlayerControllerSSC` class, and not our old `TutorialPlayerController` class!

Creating a top-down camera

The top-down camera is popular with RTS games or shooters, as it offers a perspective that allows players to easily see approaching enemies. With our top-down camera we'll have our pitch locked, so that the player cannot look up or down, and therefore also locks the camera from being raised or lowered.

We will still allow the player to freely yaw left and right, and therefore rotate around the world, although the camera will follow the pawn from a fixed perspective.

We'll be hardcoding our values again. Much of what you'll see next is similar to the code found in the side-scrolling camera code. We'll be using another custom player controller as well, which is only marginally different from that of the side-scroller's.

For this purpose, we will want to create a new player controller class, called `TutorialPlayerControllTDC`.

Just as we did with the third person camera, we'll want the first person weapon mesh to remain hidden, and continue to expose the third person weapon and pawn meshes to the camera.

Getting ready

We're going to make a whole new `PlayerController` class for this tutorial, as we need to add a new function to it. Create a new class called `TutorialPlayerControllerTDC` (top-down camera) and have it extend from `PlayerController`. For this recipe we'll be creating a new player controller class so that our pawn's aim is not affected by the new camera system we've implemented. Otherwise, our pawn's aim would be way off target as it would not be using our player's rotation, but our camera's.

```
class TutorialPlayerControllerTDC extends PlayerController;
```

How to do it...

The first thing we'll want to do here is to adjust the aiming for our pawn. Without this, our pawn's aim would follow where our camera is pointed, and not where our pawn is facing.

1. Let's add the code for that now. This is identical to the code found in `TutorialPlayerControllerSSC.uc`.

```
/********************************************************** * USED
FOR ALL CAMERAS ***
* Rotator where the pawn will be aiming the weapon.
* Will be different, depending on which camera system we
are using
**********************************************************/
function Rotator GetAdjustedAimFor(Weapon W, vector StartFireLoc)
{
  returnPawn.GetBaseAimRotation();
}
```

2. The next part is nearly identical to that in the side-scrolling camera as well, and is the only change we've made to this class. Add the following code, then I'll explain:

```
/**********************************************************
* Forces the weapon to fire in the direction the pawn is
facing
**********************************************************/
function UpdateRotation( float DeltaTime )
{
  local Rotator DeltaRot, newRotation, ViewRotation;

  ViewRotation = Rotation;

  // Stop the player from adjusting the pitch of the camera
  DeltaRot.Pitch  = 0;
```

```
  // Allows the pawn to rotate left and right
  DeltaRot.Yaw = PlayerInput.aTurn;
  ProcessViewRotation( DeltaTime, ViewRotation, DeltaRot );
  SetRotation(ViewRotation);
  NewRotation = ViewRotation;

  if ( Pawn != None )
    Pawn.FaceRotation(NewRotation, deltatime);
}
```

Let's focus on the important change that we've made within that block of code:

```
  /** Stop the player from adjusting the pitch of the
  camera */
  DeltaRot.Pitch = 0;
  /** Allows the pawn to rotate left and right */
  DeltaRot.Yaw = PlayerInput.aTurn;
```

Our `Pitch` is set to `0`, just as it was for the side-scroller, because we don't want the camera or pawn to be able to look up or down.

At the same time, we are tying the yaw of the pawn (and by connection, the camera) to the yaw of the mouse. If the player moves the mouse left and right, the pawn and camera will follow.

3. Our default properties are left empty as shown in the following code snippet:

```
defaultproperties
{
}
```

4. Next up, we need to create our `TopDownCam` class. Have it extend from `Camera` as shown as follows:

```
class TopDownCam extends Camera;
```

5. Just as we've done with all of our other cameras, we'll add the code for this. It looks identical to the side-scrolling camera. First we'll add our variables as shown in the following code snippet:

```
/** Hardcoded vector offset we will use, rather than
tweaking values in the editor's CameraProperties */
var constVector  CamOffset;

/** Hardcoded rotator offset we will use, rather than
tweaking values in the editor's CameraProperties */
var constRotator CamOffsetRotation;
```

6. Now add our functions as shown in the following code snippet:

```
/*****************************************************************
* Query ViewTarget and outputs Point Of View.
* @paramOutVTViewTarget to use.
* @paramDeltaTime  Delta Time since last camera update (in
seconds)
*****************************************************************/
functionUpdateViewTarget
(outTViewTargetOutVT,floatDeltaTime)
{
  local Pawn Pawn;
  local Vector V, PotentialCameraLocation, HitLocation,
                HitNormal;
  localActor  HitActor;

  /** UpdateViewTarget for the camera class we're extending
      from */
  Super.UpdateViewTarget(OutVT, DeltaTime);

  /** If there is an interpolation, don't update outgoing
     viewtarget */
  if (PendingViewTarget.Target!=None&&OutVT==
     ViewTarget&&BlendParams.bLockOutgoing)
  {
    return;
  }

  Pawn = Pawn(OutVT.Target);
  if (Pawn != None)
  {
    /** Hide the pawn's "extra" weapon */
    if(Pawn.Weapon!=none)
    {
      Pawn.Weapon.SetHidden(true);
    }
/*****************************************************************
* If you know the name of the bone socket you want to use,
then
* replace 'WeaponPoint' with yours.
* Otherwise, just use the Pawn's eye view point as your
starting
* point.
*****************************************************************/
```

```
/**socket not found, use the other way of updating vectors */
    if(Pawn.Mesh.GetSocketWorldLocationAndRotation
    ('WeaponPoint', OutVT.POV.Location,
    OutVT.POV.Rotation)==false)
    {
      /** Start the cam location from the target eye view
      point */
      OutVT.Target.GetActorEyesViewPoint
      (OutVT.POV.Location, OutVT.POV.Rotation);
    }

    /** Force the camera to use the target's rotation */
    OutVT.Target.GetActorEyesViewPoint
    (V, OutVT.POV.Rotation);

    /** Add the camera offset */
    OutVT.POV.Rotation+=CamOffsetRotation;

    /** Math for the potential camera location */
    PotentialCameraLocation=OutVT.POV.Location
    +(CamOffset>>OutVT.POV.Rotation);

    /** Draw a trace to see if the potential camera
    location will work */
    HitActor=Trace(HitLocation, HitNormal,
    PotentialCameraLocation, OutVT.POV.Location, true,,,
    TRACEFLAG_BULLET);

    /**  Will the trace hit world geometry? If so then use
    the hit location and offset it by the hit normal */

    if (HitActor!=None&&HitActor.bWorldGeometry)
    {
      OutVT.POV.Location=HitLocation+HitNormal*16.f;
    }
    else
    {
      OutVT.POV.Location=PotentialCameraLocation;
    }
  }
}
```

Our default properties are the only changes to this class, when compared to the side-scroller's. I found these values by using the tutorial camera and adjusting the values with the archetype until I found what I deemed to be appropriate.

I used a pitch of -80 degrees to have the camera point down at the pawn. You'll notice that my pitch says `-14000` here though. That's because of UDK's rotation system that I mentioned earlier which is based on radians, remember? A pitch of -80 degrees is roughly equivalent to 14000 of Unreal's unit of measurement.

```
/** Hardcoded vector & rotator values for our camera */
defaultproperties
{
    CamOffset=(x=-700,y=0,z=0)
    CamOffsetRotation=(Pitch=-14000)
}
```

7. Next up, we need to make a change to our `TutorialPawn` class. There is only one small change here from the changes we made during our side-scroller tutorial. We're going to override the `GetBaseAimRotation` function found in the `Pawn.uc` class again to have it suit our needs. Add the following code:

```
/******************************************************************
** USED FOR TOP DOWN Camera**
* Forces the weapon to use the pawn's direction for aiming,
and not the camera's.
* shots will be fired in the direction the gun is pointed.
Used by PlayerController.
* Comment this out if you are not using the Side Scrolling
Camera.
* @return POVRot.
*****************************************************************/
simulated singular event Rotator GetBaseAimRotation()
{
  local rotator POVRot, tempRot;

  tempRot = Rotation;
  SetRotation(tempRot);
  POVRot = Rotation;

  /** Stops the player from being able to adjust the pitch
   of the shot, forcing the camera to always point down
   towards the pawn
  * We can still rotate left and right, however.*/
  POVRot.Pitch = 0;

  returnPOVRot;
}
```

As you can see, we've set our `POVRot` to use the rotation of our pawn. Therefore, our camera's rotation will follow the pawn's rotation. Additionally, we've set our `Pitch` to `0`, so that the player no longer has any control over the pitch of either the pawn, and by extension, the camera.

8. The final change we need to make is in the `TuorialGame` class. Be sure to change the following code in your `defaultproperties`, so that your game uses your new controller:

   ```
   PlayerControllerClass=class'Tutorial.TutorialPlayerControll
   erTDC'
   ```

9. Compile the project and take a look!

How it works...

We wanted to hardcode our values again. Much of what we saw was similar to the code found in the side-scrolling camera. We also made use of another custom player controller as well, which is only marginally different from that of the side-scroller's.

Just as we did with the third person camera, we hid the first person weapon mesh and continued to expose the third person weapon and pawn meshes to the camera.

We also had to override the `GetBaseAimRotation` function in our pawn class and `GetAdjustedAimFor` function in our player controller. These changes tell the game to use the direction the pawn is facing for firing shots, and not the camera's direction. We've also locked our pawn so that it cannot look up or down when firing shots.

Other than the player controller though, the code is very similar to what we had in our side-scrolling camera class, except for the hardcoded values declared in our `defaultproperties` block, which are now adjusted to work for a top-down view.

4
Crafting Pickups

In this chapter, we will be covering the following recipes:

- Creating our first pickup
- Creating a base for our pickup to spawn from
- Animating our pickup
- Altering what our pickup does
- Allowing vehicles to use a pickup

Introduction

Artificial intelligence can cover a variety of things in UDK, so we won't delve too far down that path, at least not in this chapter. Here, we'll briefly cover it, and how the AI interacts with pickups throughout the game, specifically what attracts them to certain pickups. Furthermore, we'll dive into creating our own pickups and how they interact with our pawn's inventory.

So with that, let's talk about **AI.**

Understanding AI

The main class that handles player actions in UDK is the `PlayerController` class. Similarly, actions are controlled by the `AIController` class. Considering that they both inherit `Controller`, they share quite a bit of functionality. For the most part, AI controllers don't necessarily need a pawn, just like a player controller.

Moving is one of the categories in which actions for AI fall into, while the other is anything that is not moving. Moving may be as simple as following another pawn, or tracking a freshly spawned health pickup, while the other category contains things such as aiming or firing a weapon.

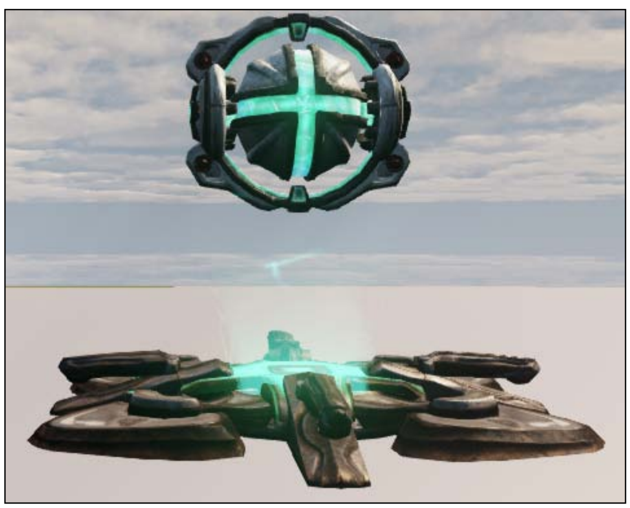
The bots, or computer controlled pawns, can be configured to have a multitude of preconfigured attitudes or reactions towards events. For example, you can write a script that instructs the bot to run for cover after taking damage, or to only fire after being fired upon.

We won't cover any of that in depth, but we will cover how to attract pawns towards our newly created pickups in this chapter, and we'll get more in depth with their intelligence in the following chapter.

Pickups

Pickups in UDK are similar to pickups in just about any other game; however, they can serve a variety of purposes. In a first person area shooter, such as the *Unreal Tournament* series or *Quake III Arena*, they can be used to adjust a pawn's properties by temporarily increasing a weapon's damage, providing invulnerability, or even invisibility.

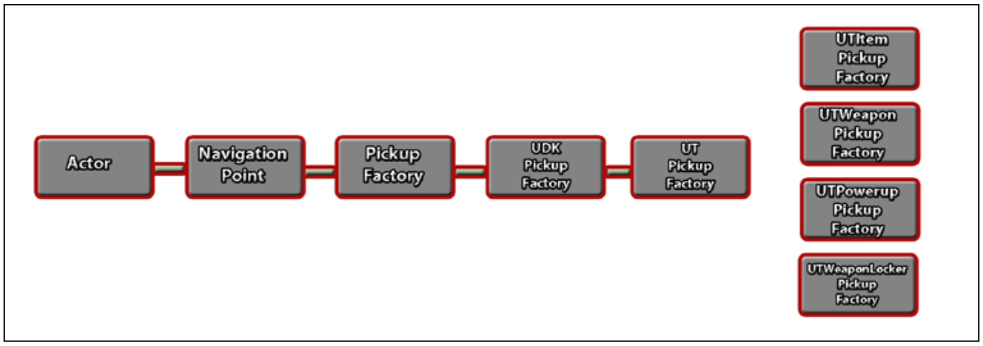
They are generally used in games to add an item to a pawn's inventory, such as ammunition for a particular weapon, additional armor for protection, or restoring a player's health, like the following example:



We can not only change what a pickup offers a player, but also how it looks, whether or not it is animated, who can/cannot acquire the item, and how attracted an AI bot is to it. We'll be covering all of those things in the following chapter.

Fortunately, UDK has provided a great template for us to work from, so we'll be using what they've provided and make additional changes to suit our needs.

Let's start by tracing the classes that our pickups will inherit from:



`Actor` is the base class of all gameplay objects. It offers functions for the animation, physics, sounds, and networking for objects in UDK, which is explained as follows:

- `NavigationPoint` is organized into a network to provide `AIControllers` the ability to determine paths to destinations throughout a level.
- `PickupFactory` is where our class finds begin to take shape. It becomes active when touched by a valid toucher, or actor that we define should be able to accept this pickup.
- `UDKPickupFactory` is largely responsible for how our pickup is perceived inside the game. Our materials are created from this class, as are other aesthetics such as how frequently the base of our pickup pulses and whether or not our pickup can rotate.
- `UTPickupFactory` provides much of what is necessary for a first person shooter, that is, it updates the player's HUD and inventory in Unreal Tournament, and an additional bot AI is illustrated here.

From those base classes the `PickupFactory` splits into four distinct classes, each of which provides unique functionality. These are given as follows:

- `UTWeaponLocker` and `UTWeaponPickupFactory`: Similar classes are used for picking up new weapons
- `UTPowerupPickupFactory`: This adds power-ups, such as improved jumping, quad damage, and temporary invulnerability
- `UTItemPickupFactory`: This contains health, ammo, and armor

The trick with creating your own pickups, or UnrealScript in general, is to find a template that best suits your need, and either extend from that and override the functions and default properties you need to change, or create your own pickup class that extends from UT or UDK pickup factories.

Creating our first pickup

We're going to create our first pickup by extending from one of the excellent ones already provided by UDK. In this case, we'll be extending from `UTAmmoPickupFactory` to create our own ammo pickup.

Getting ready

We'll need to open up our IDE and create a new class extending from our `UTAmmoPickupFactory` class.

```
class Tut_AmmoPickup extends UTAmmoPickupFactory
```

Afterwards we'll make some tweaks so that it suits our needs, and the tutorials that follow will really make this a personalized pickup, as we adjust a pawn's desire to head towards it, the animations it performs, and who can and cannot pick it up.

Now that we've got a class extending from `UTAmmoPickupFactory`, we've also inherited all of that class's properties, including its functionality. For this recipe, we won't have to make any changes to the functions, but we will be altering some of its default properties.

We won't need every category in our class to be accessible within the editor once we create an archetype for the class, so let's clean things up by adding the following code to keep things neat:

```
// Hides categories that we won't be needing from the archetype
HideCategories(Object, Debug, Advanced, Mobile, Physics);
```

> You'll also need to remove the semicolon (`;`) from your class declaration, otherwise you'll receive an error. The top of your class should now read as follows:
>
> ```
> Class Tut_AmmoPickup extends UTAmmoPickupFactory
> ```
>
> For reference, all of our classes will be saved under the following directory:
>
> `C:\UDK\July\Development\Src\Tutorial`
>
> It follows the format given next:
>
> `HardDrive\UDK\MonthOfTheBuild\Development\Src\FolderName`

How to do it...

1. In the `defaultproperties` block at the bottom of your class, add the following code:

   ```
   // How many rounds will be added to our weapon type
     AmmoAmount=10
     // The type of weapon that this pickup will supply ammo
        for
     TargetWeapon=class'UTWeap_ShockRifle'
   ```

   This simply declares the amount of ammo that will be added to the pawn's inventory, as well as for which weapon class when it is picked up.

2. Moving on, add the following code for the sound the pickup will make when it is acquired, respawn time, and the desirability for a bot to pick it up:

   ```
   // The sound effect triggered when the pickup is acquired
   PickupSound=SoundCue'A_Pickups.Ammo.Cue.
   A_Pickup_Ammo_Rocket_Cue'

   /** The value at which an AI bot desires the pickup. Higher
   value = will lean towards this pickup */
   MaxDesireability=0.3

   // Time (seconds) before the pickup respawns
   RespawnTime=10.0
   ```

> Desirability is a bot's attraction to a particular object in UDK. It can be anything from ammo or health, to a specific weapon. Bots can also be programmed to have a preferred weapon, and seek ammunition for that above all else. The higher the desirability, the more likely a pawn is to ignore other objects (and often pawns) to go after it.
>
> This is especially effective when changing an AI's routine during a particular gameplay type. In capture, if the flag matches, you can often tell your AI companions to either defend your flag, attack the enemy flag, or roam about and seek enemies.

We really could use any sound here, so for the sake of variety let's use the rocket launcher's pickup sound.

3. Next, we need to add the light environment for our pickup. Write this code in your `defaultproperties` block:

```
/** Offers a light around our texture so that it can be
seen within the game and editor*/
Begin Object Name=PickupLightEnvironment
    AmbientGlow=(R=1.0f,G=1.0f,B=1.0f,A=1.0f)
End Object
```

   Without a light environment you would have a very dark texture, and it would be nearly unrecognizable. We'll use an ambient glow and set all values (red, green, blue, and alpha (transparency)) to one. If you want more of a washed out look, feel free to increase the values across the board.

> Perhaps you'd like to bask your pickup in a glow that represents the current environment or atmosphere of a level. If you were in a stage filled with lava and fire, it may be wise to have a stronger red value than green or blue. Stages surrounded by water would be best suited to have a blue hue, so consider raising the blue value.

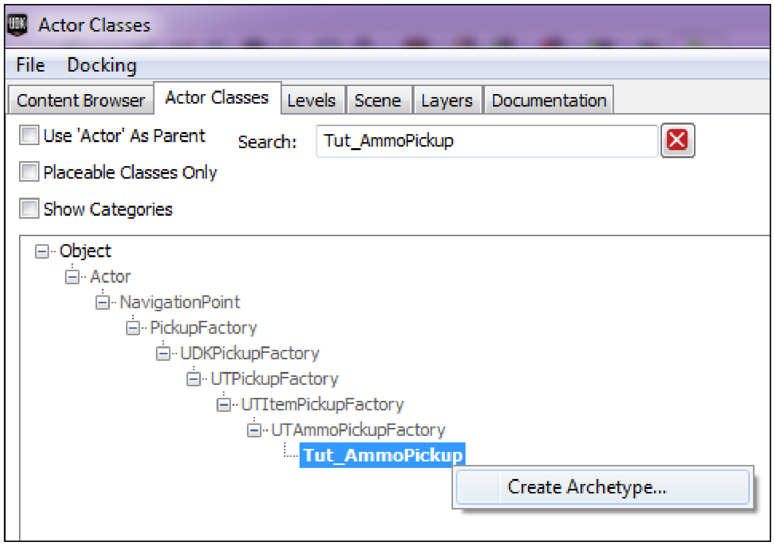
4. With our light environment taken care of, we can now add the visible mesh for our ammunition.

```
/** The static mesh, or object you physically see within
the editor and game */
Begin Object Name=AmmoMeshComp
    StaticMesh=StaticMesh'Pickups.
Ammo_Link.Mesh.S_Ammo_LinkGun'

/** Slight offset, to allow for the mesh and base to line on
center */
   Translation=(Y=-10.0)

    Rotation=(Roll=16384)
End Object
```

   I always thought that the link gun's ammo looked pretty neat, so let's add that in there. The rotation value does exactly what you would imagine; it rotates the ammo in place. If you remember from our camera tutorial, UDK uses its own system for rotation, as illustrated in the camera tutorial.

   That's all there is to scripting an ammo pickup. We'll now need to create an archetype for it in the editor, so that we can place it in a level.

5. Compile the code and launch the editor. Within the **Actor Classes** browser, make sure that none of the boxes are checked, and access our newly created class by typing `Tut_AmmoPickup` in the search field.
6. Right-click on `Tut_AmmoPickup` when it appears, and when the **Create Archetype...** dialog appears, left-click on it to create an archetype.

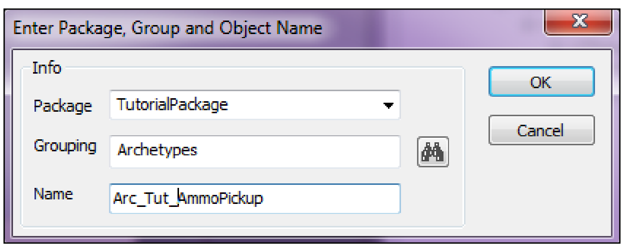


7. Enter the following information into the fields:

   Package: `TutorialPackage`

   Group: `Archetypes`

   Name: `Arc_Tut_AmmoPickup`



8. With that done we now have a pickup that we can drag-and-drop into an editor. Drag your archetype onto the screen and it should appear like the following image:

We've created our first pickup!

How it works...

By extending from a parent class, which offered a considerable amount of base functionality to build from, we were able to easily add our own properties to make a working pickup.

Rather than create our own assets, we chose to use those already packed in with UDK. We aren't limited to just using static meshes that other pickups use, however. Nothing is stopping us from shrinking a truck down and using it as an RC sized pickup to represent a larger vehicle.

We want to use the class we wrote in the UDK editor, so we've created an archetype for it, which references the `Tut_AmmoPickup` class.

Creating a base for our pickup to spawn from

Now that we have our first pickup created, we'll want to really customize it to suit our purposes. Our pickup seems kind of boring if it is just floating by itself. Arena style shooters generally have a row of four or five small health or ammo packs adjacent to one another, but what if we want to make our pickup seem more important, at least visually?

Adding a base mesh to a pickup is a great way to emphasize that a particular pickup is important. By adding a base mesh, our pickup is no longer floating from thin air, but actually appears to spawn from a device of some sort.

We may want a weapon or a power-up, such as quad damage to be highlighted on the level, so we'll add a static mesh beneath it.

Getting ready

Open up our `Tut_AmmoPickup` class in your IDE and we'll begin to make those changes.

How to do it...

Once again we will need to start by making changes within our `defaultproperties` block, as that is where most of our pickup's functionality can be adjusted easily:

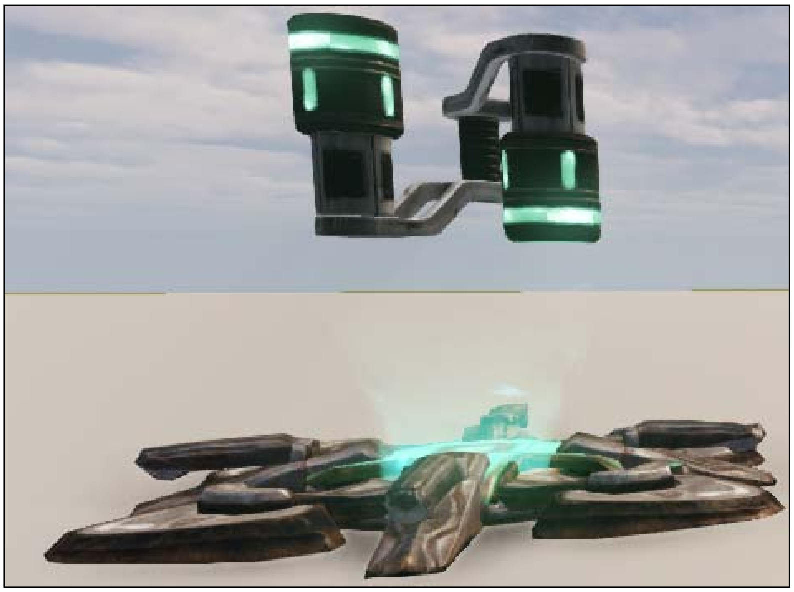
1. At the bottom of your class, inside of your `defaultproperties` block, add the following code:

   ```
   defaultproperties
   {
   ......
   /** Name of the base mesh that sits beneath our pickup */
   Begin Object Name=BaseMeshComp
   StaticMesh=StaticMesh'Pickups.
   Health_Large.Mesh.S_Pickups_Base_Health_Large'

   /** We want to drop it down a bit beneath the pickup to
   allow for a particle to rest between the pickup and the
   base */
     Translation=(Z=-44)
     Scale=.8
   End Object
   ......
   }
   ```

   We've just added a base mesh to sit beneath our pickup. The reason we translate on the z axis is because we want to have some room to add a particle effect in our next step, otherwise our base would look kind of bland. We also scale it to 0.8, as otherwise there would be a slight offset applied to the base mesh. You could have the base mesh at full size, but you would need to translate on the x axis to compensate for the offset.

2. Beneath that block, let's add some code for our particle effect:

```
defaultproperties
{
....
/** Particle class component between our base and pickup */
Begin Object Class=UTParticleSystemComponent
Name=ParticleGlow

    Template=ParticleSystem'Pickups.Health_Large.
    Effects.P_Pickups_Base_Health_Glow'

    /** Slight translation to allow for the particle to sit
    between the base mesh and the pickup */
    Translation=(Z=-50.0)

End Object
Components.Add(ParticleGlow)
Glow=ParticleGlow
....
}
```



This is added the same way that we've added the base and pickup meshes. We add a small translation on the z axis here too, because we want it to appear as though it is sprouting from the base mesh, and finishing around where our pickup will sit.

How it works...

All game-related objects are derived from the base class, `Actor`, in the Unreal Engine. `ActorComponents` define numerous lightweight components that provide an interface for modularity of actor collision and rendering. An `ActorComponent` object is an object which can be attached to an actor, and subclass can override some or all of the default properties of the component.

In this example, we're adding a particle component to our pickup, which allows for the viewing and alteration of particles when the pickup is spawned.

We've also added a component for our base mesh, which allows us to easily swap out the static resting beneath our pickup. Both of these aesthetic changes can make it easy for players to discern what purpose a pickup serves from a distance (that is, health, armor, weapons, and so on).

Animating our pickup

Our pickup is moving towards almost completed, but there are a few more additions we can make to it, to allow for a bit more life behind breathe life into our object. Let's add a rotation and bob to our pickup, so that it really grabs our eye with some animation.

Getting ready

Open up your `Tut_AmmoPickup` class in your IDE and we can begin.

How to do it...

This is very straightforward. We start off by adding a rotation to our pickup, and then add an animated bobbing motion. Just as we did with our previous tutorials, we'll need to alter our `defaultproperties` block as explained in the following steps:

1. In the `defaultproperties` block write the following code:

```
defaultproperties
{
....
/** If true, our pickup will rotation in place */
bRotatingPickup=true
/** How quickly the pickup rotates */
YawRotationRate=16384
/** if true, the pickup mesh floats (bobs) slightly */
bFloatingPickup=true
```

```
/** How fast should it bob */
BobSpeed=1.0
/** How far to bob.  It will go from +/- this number */
BobOffset=5.0
....
}
```

2. Your pickup will now rotate in place, based on the pivot point of your static mesh. Adjust the rotation rate to a number that best suits your needs by adjusting the rotation rate. This will also allow our pickup to bob up and down.

> Static meshes may not always have their pivot point centered on the object (that is, a door generally uses a corner) so you may have to offset yours a bit. You can edit it with `var vector PivotTranslation;`.

With the `bFloatingPickup` set to `true` our pickup will now bob in place, while `BobSpeed` and `BobOffset` are variables to fine-tune the animation itself. Increasing or decreasing the offset will force the pickup to drop and raise to lower and higher points respectively, as though it were riding on a wave.

How it works...

The parent classes of our pickup offer Booleans for whether or not our pickup can be animated in a number of ways. By default their values are set to `false`, or off, and we are simply turning them on. Additionally, we can easily manipulate the animation properties by adjusting the `BobSpeed` and `BobOffset` variables.

Play with some of these values to really create something completely different, like a quickly spinning pickup that spawns a particle effect when picked up.

Altering what our pickup does

Now that we have a pickup which offers ammo to the player, and know how to alter a variety of the pickup variables and aesthetics, let's take a moment to create a pickup that now offers health.

Getting ready

Open up IDE and create a new class called `Tut_HealthPickup`. Have it extend from `UTHealthPickupFactory`.

```
class Tut_HealthPickup extends UTHealthPickupFactory
```

How to do it...

This is a bit more complicated than our previous recipes. We'll have to create a new pickup class in our IDE, as well as an archetype in the editor, so that we're able to access the class and its variables within the editor. This is essential when working with level designers who are not familiar with code.

1. Firstly, we want to have a sleek interface when we play with our pickup in the editor, so add the following code, beneath your class declaration:

```
class Tut_HealthPickup extends UTHealthPickupFactory
/** Hides categories that we won't be needing from the
    archetype */
HideCategories(Object, Debug, Advanced, Mobile, Physics);
```

2. The rest of the code will be going in the `defaultproperties` block:

```
defaultproperties
{
  /** The value at which an AI bot desires the pickup.
Higher value = will lean towards this pickup */
  MaxDesireability=0.700000
  /** How much this pickup will heal the pawn for */
  HealingAmount=20

  /** sound played when the pickup becomes available */
  RespawnSound=SoundCue'A_Pickups.Health.Cue.
  A_Pickups_Health_Respawn_Cue'
  /** Time (seconds) before health pickup respawns */
  RespawnTime=10.000000

  /** Pickup will rotate */
  bRotatingPickup=true
  /** Speed of the rotation */
  YawRotationRate=16384
  /** if true, the pickup mesh floats (bobs) slightly */
  bFloatingPickup=true
  /** How fast should it bob */
  BobSpeed=7.0
  /** How far to bob. It will go from +/- this number */
  BobOffset=2.5
}
```

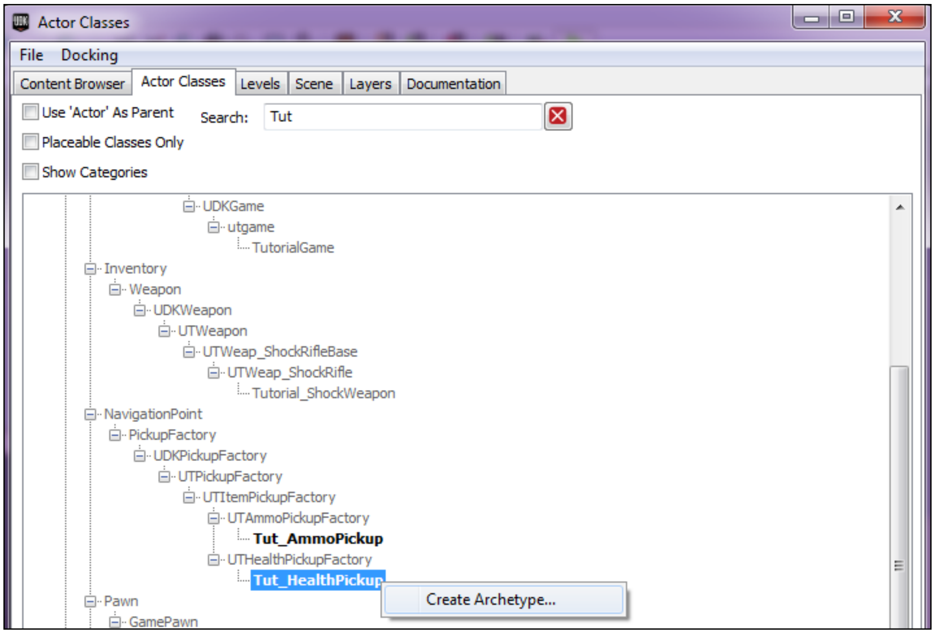
This is the same code from the previous recipe, albeit some of the values have changed and we've now added a new variable, `HealingAmount`. This does exactly what you think it does.

3. Finally, we're going to add the static mesh code for the pickup and the base, along with the particle system that rests between those two items:

```
/** Base mesh */
Begin Object Name=BaseMeshComp
  StaticMesh=StaticMesh'Pickups.WeaponBase.
  S_Pickups_WeaponBase'
  Translation=(Z=-44)
  Rotation=(Yaw=16384)
  Scale=0.8
End Object

/** Health Mesh */
Begin Object Name=HealthPickUpMesh
  StaticMesh=StaticMesh'Pickups.Ammo_Shock.Mesh.
  S_Ammo_ShockRifle'
  MaxDrawDistance=7000
  Materials(0)=Material'Pickups.Ammo_Shock.Materials.
  M_Ammo_ShockRifle'
End Object

/** Particle System */
Begin Object Class=UTParticleSystemComponent
Name=ParticleGlow
  Template=ParticleSystem'Pickups.Health_Large.Effects.
  P_Pickups_Base_Health_Glow'
  Translation=(Z=-50.0)
  SecondsBeforeInactive=1.0f
End Object
Components.Add(ParticleGlow)
Glow=ParticleGlow
```

4. Build your script, then launch the UDK editor. We're going to create an archetype for this, just as we did for the ammo pickup.
5. Open the **Actor Classes** browser, uncheck all of the boxes for **Use Actor as Parent**, **Placeable Classes Only**, and **Show Categories**.

6. Search for the name of our new pickup class, `Tut_HealthPickup`, and when it appears, right-click on it to see the **Create Archetype...** pop-up appear, and left-click to accept.

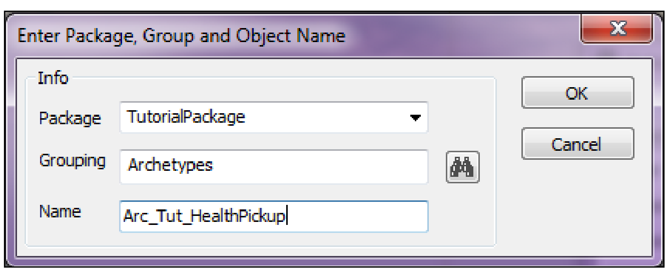


Enter the following information for the package, group, and object name:

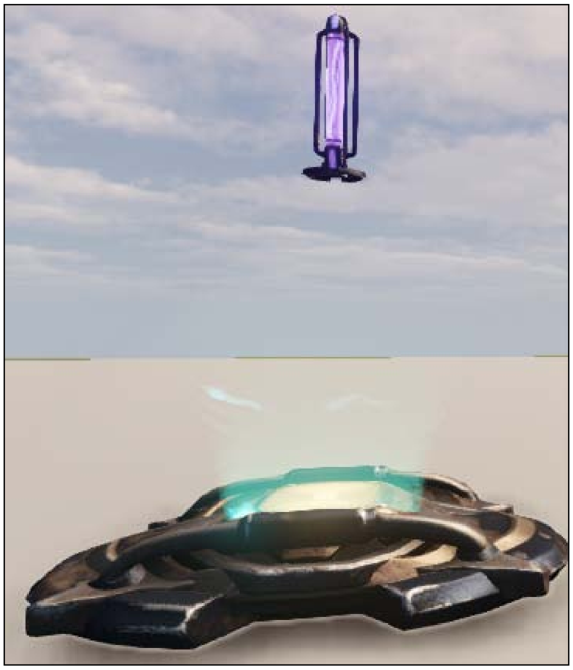
Package: `Tutorial Package`

Grouping: `Archetypes`

Name: `Arc_Tut_HealthPickup`



7. With our newly created archetype ready, we can now drag-and-drop it into the editor's window and see the reward for our hard work.



We're done! Our pickup will now offer health instead of ammo. If you were to create another health pickup which offers more health, some subtle changes can go a long way.

I'd suggest placing a new static mesh for the pickup, and consider increasing the scale of the mesh, along with increasing the health value. Furthermore, bots generally desire health pickups more than ammo, so if you do create a pickup which offers more health, be sure to increase the bot desirability as well.

How it works...

By extending from a different class, `UTHealthPickupFactory`, we are able to easily change the purpose of our pickup. Rather than have one which drops ammo, we can now have a pickup which spawns health. We can just as easily create a pickup which spawns armor as well.

Again, we've created an archetype, so that our pickup class can be referenced and used in the UDK editor and placed in our level.

Allowing vehicles to use a pickup

By default, vehicles in UDK cannot make use of pickups. If you had a game that relied heavily on vehicle use however, I'm sure you'd love to find a way to repair your vehicle's health, or increase its dwindling ammunition reserves.

Getting ready

Vehicles in UDK cannot pickup items and add them to their inventory by default. The process of allowing vehicles to make use of the pickup inventory is incredibly simple, and we'll start by creating our own vehicle class, along with its content class.

Make the first class, `Tut_Vehicle_Scorpion_Content`, and have it extend from `UTVehicle_Scorpion_Content`.

```
class Tut_Vehicle_Scorpion_Content extends UTVehicle_Scorpion_Content;
```

The only information in this class is found within the `defaultproperties` block, and should read as follows:

```
defaultproperties
{
  bCanPickupInventory=true
}
```

This is what allows our vehicle to use pickups. We use a vehicle's content class, because that's where all of the data for the aesthetics, weapons, and attributes are held, while the vehicle class generally holds the gameplay functions and mechanics for the vehicle.

Create another class, specifically for the vehicle, and call it `Tut_Vehicle_Scorpion`. It should extend from `UTVehicleFactory`. Now let's begin.

How to do it...

This is perhaps the most daunting of our recipes so far. We'll have to create a few classes, in addition to archetypes, and place them throughout the level for our use. This will include creating a new vehicle class, as well as a content class for the vehicle, which is where it inherits most of its default properties from, such as the aesthetics, sounds, and even part of the physics behavior. Additionally, we'll be creating a pickup class that allows our vehicle to drive over and acquire the pickup as explained in the following steps:

1. In the `defaultproperties` block for `Tut_Vehicle_Scorpion`, add the following code:

   ```
   defaultproperties
   {
   ```

```
  /** Vehicle's skeletal mesh. What you actually see in the
  game and editor */
  Begin Object Name=SVehicleMesh
    SkeletalMesh=SkeletalMesh'VH_Scorpion.Mesh.
    SK_VH_Scorpion_001'
    Translation=(X=0.0,Y=0.0,Z=-70.0)
  End Object


  /** Removes the sprite from the in game editor. We see
  the actual skeletal mesh instead of a sprite */
  Components.Remove(Sprite)

  Begin Object Name=CollisionCylinder
    CollisionHeight=+80.0
    CollisionRadius=+120.0
    Translation=(X=-45.0,Y=0.0,Z=-10.0)
  End Object

  /** Path to our custom made scorpion content, which can
  now pickup items */
  VehicleClassPath="Tutorial.Tut_Vehicle_Scorpion_Content"
  /** Default scale for the object to appear in game */
  DrawScale=1.2
}
```

This is almost identical to the scorpion used in UDK. The only changes I've made were the comments and the `VehicleClassPath` variable. This now leads to the custom scorpion content that we made at the beginning of this recipe, as it allows our scorpion to use pickups.

Our vehicle now has the ability to gather pickups throughout a level. The problem now however, is that UDK doesn't come with any vehicle pickups! So we must create one ourselves.

2. Now that we've made several pickups in the previous recipes, this part should be easy. We'll just be editing the functional part of the pickup, and leave the aesthetics as they are for now. For this, create a new class called `Tut_Vehicle_Health_Pickup` and have it extend from `UTHealthPickupFactory` as shown in the following code:

```
class Tut_Vehicle_Health_Pickup extends
UTHealthPickupFactory

// Hides categories that we won't be needing from the archetype
HideCategories(Object, Debug, Advanced, Mobile, Physics);
In the default properties block, add the following code:
Defaultproperties
{
  /** The value at which an AI bot desires the pickup. Higher
value = will lean towards this pickup */
```

```
  MaxDesireability=0.700000
  /** How much this pickup will heal the vehicle for */
  HealingAmount=20
  /** The sound effect triggered when the pickup is
  acquired */
  PickupSound=SoundCue'A_Pickups.Ammo.Cue.
  A_Pickup_Ammo_Rocket_Cue'

  /** Time (seconds) before health pickup respawns */
  RespawnTime=10.000000

  /** Pickup will rotate */
  bRotatingPickup=true
  /** Speed of the rotation */
  YawRotationRate=16384
  /** if true, the pickup mesh floats (bobs) slightly */
  bFloatingPickup=true
  /** How fast should it bob */
  BobSpeed=1.0
  /** How far to bob. It will go from +/- this number */
  BobOffset=5.0
}
```

These are all the same properties and values that we used for our pawn's pickup as well.

3. Now we'll add the appearance of the pickup:

```
/** Base Mesh */
  Begin Object Name=BaseMeshComp
    StaticMesh=StaticMesh'Pickups.WeaponBase.
    S_Pickups_WeaponBase'
    Translation=(Z=-44)
    Rotation=(Yaw=16384)
    Scale=0.8
  End Object

/** Health mesh */
  Begin Object Name=HealthPickUpMesh
    StaticMesh=StaticMesh'Pickups.Health_Medium.Mesh.
    S_Pickups_Health_Medium'
    MaxDrawDistance=7000
  End Object

/** Particle */
  Begin Object Class=UTParticleSystemComponent
  Name=ParticleGlow
    Template=ParticleSystem'Pickups.Health_Large.Effects.
    P_Pickups_Base_Health_Glow'
    Translation=(Z=-50.0)
    SecondsBeforeInactive=1.0f
```

```
    End Object
    Components.Add(ParticleGlow)
    Glow=ParticleGlow
}
```

Again, this is identical to our pawn's health pickup, except that we'll change the mesh for the pickup, so that it stands out from the pawn's health. I've swapped out the shock rifle ammo's static mesh (which was serving as a placeholder) with the medium health pickup static mesh, as seen in the following line:

```
'Pickups.Health_Medium.Mesh.S_Pickups_Health_Medium'
```

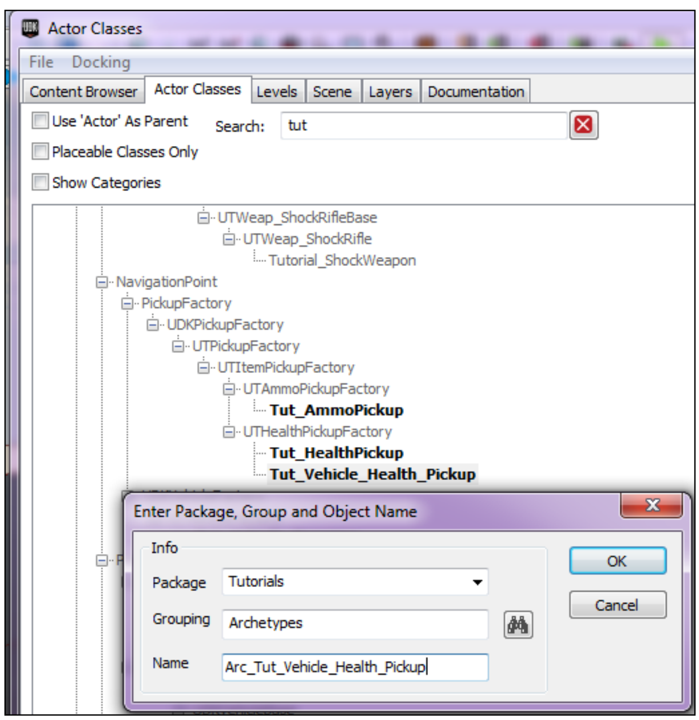
4. We've got our vehicle class, vehicle content class, and health pickup scripts all written, but now we need to get the vehicle and pickup within the UDK editor. As we've been doing with all of our other classes in this book, we'll be using archetypes.

   Open up the UDK editor and scroll over to your **Actor Classes** browser. Create an archetype for your vehicle (NOT the content class) and another archetype for your health pickup. For the vehicle enter the following information:

   Package: `TutorialPackage`

   Grouping: `Archetypes`

   Name: `Arc_Tut_Vehicle_HealthPickup`



5. With our archetypes created we can now drag-and-drop them into the map and use them from there.

> Included with the chapter are all of the archetypes, scripts, and a map with all of these objects placed in them. Feel free to peruse through the map if you are confused at any point.

6. Drag-and-drop your two new classes into the map, then press the **PIE** button to spawn your character into the map.

7. Hop into the scorpion, then exit it, and shoot it for a bit to damage it.
8. Hop back into the scorpion, and drive over the health pickup. Notice how your health is suddenly increased by 20 points.

> If you're using the Scaleform version of the HUD, you'll immediately notice here that the damage applied to the vehicle doesn't display correctly on the HUD. For some reason your health percentage will remain at 100, but your ammo will drop as you take damage. At least it is displayed that way.
>
> The reason you need to hop into your scorpion for the first time seems silly as well, but it's necessary as the vehicles are invulnerable to damage until a pawn enters it for the first time!

With that in place, we've completed our first pickup for our vehicles!

How it works...

This process required a few steps. First, we had to create `Tut_Vehicle_Scorpion_Content` and have it extend from `UTVehicle_Scorpion_Content`. This allowed us to create a copy of the scorpion content class, without worrying about our changes affecting the original. Within this new class, we only changed the default property of `bCanPickupInventory` and set it to `true`, so that our vehicle can now pickup items throughout a map.

Once we've told the game's scorpion to use our new scorpion content class, we were able to move onto the pickup, which is made in the exact same manner as we made the pickups for our pawn. Again, an archetype was created for our pickup, so that the class can be placed in the editor and used on the map.

5
AI and Navigation

In this chapter, we will be covering the following recipes:

- Laying PathNodes on a map
- Laying NavMeshes on a map
- Adding a scout to create NavMesh properties
- Adding an AI Pawn via Kismet
- Allowing a pawn to wander randomly around a map
- Making a pawn patrol PathNodes on a map
- Making a pawn randomly patrol PathNodes on a map
- Allowing a pawn to randomly patrol a map with NavMeshes
- Making a pawn follow us around the map with NavMeshes

Introduction

The Unreal Engine has two ways of handling pathfinding. They both have their pros and cons, despite being somewhat similar. Quite simply, they can be broken down into **WayPoints** and **navigation meshes**.

WayPoints

Pathfinding in the Unreal Engine is based on a pre-generated path network, which is laid down by the developer. The path network doesn't cover 100 percent of the area the AI may navigate. Therefore the AI also needs to be able to perform localized evaluation and routing of the environment. The AI does this by using collision (ray) traces to determine how far objects are and whether or not they are passable.

NavigationPoints are laid on a map by the designer, and are used to illustrate where they do and do not wish for pawns to be able to move to. When the level is built, the NavigationPoints will produce **ReachSpecs** between them. The ReachSpec data is then used by the game's pathfinding, which in turn allows AI to pathfind from one point to the next and judge the most efficient way of maneuvering about it. Path searches are initialized from **Anchors**, which are navigation points that the AI can reach directly, without having to perform pathfinding. Pathfinding starts by searching for a start and end point and then calculating the most efficient way to navigate the points between the two to reach the end goal.

WayPoints make use of the `FindPathToward` function, which will determine the path network route from the anchor to goal. First, it checks to see if the pawn is valid and then takes a glance at all of the nodes across a path. This information is then stored in the **RouteCache** of the **AI Controller**.

PathNodes can be anything from a **PlayerStart** (subclass of NavigationPoint), as well as any of those pickups we've created in the previous tutorials. As a general rule, PathNodes should be less than 1200 Unreal units apart to prevent issues with the AI not being able to find the path, although this can be modified in `MAXPATHDIST`, which is an internal variable of NavigationPoint. Essentially, level designers want to place PathNodes throughout an entire map in order to allow the AI to better navigate the playable area and not waste any time creating art and assets that players will never see or use.

Benefits of WayPoints

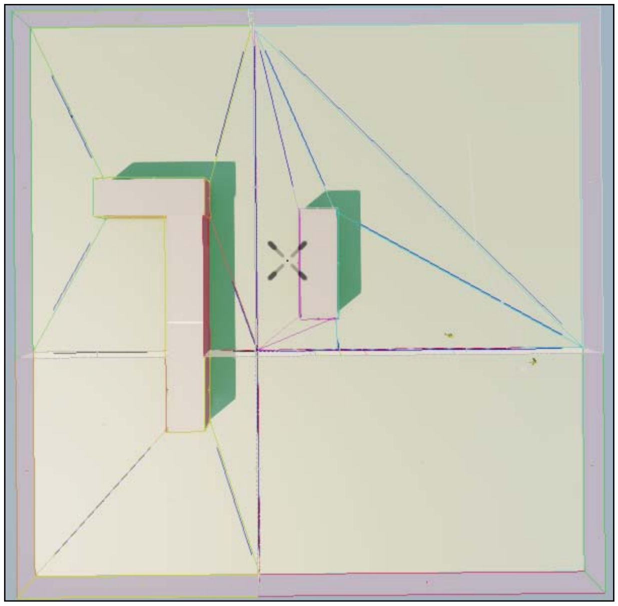
While WayPoints may be a bit more work and cause a bit more clutter on a map, they are great when you want to tell the AI precisely where to go. For example, if you wanted to create a scenario where you have AI pawns flanking on either side of you, this could easily be arranged by assigning them to make use of pre-set WayPoints when you run over a trigger.

Moreover, WayPoints are great for when you have actors of different sizes making use of a tight environment. Perhaps you only want pawns to be able to access an area and not a vehicle. Well then, WayPoints are the way to go for relaying this information to the game.

Above all else, WayPoints are all about precision and allow you to have complete control over how pawns can traverse an environment. You tell them exactly where they can and cannot explore.

NavMeshes

NavMeshes take an alternative approach to pathfinding. NavMeshes attempt a more accurate representation of an AI's configuration space via a connected graph of convex polygons, as opposed to representing the world as a series of connected points. Through the use of a node, which is essentially a polygon, the AI can maneuver from any point in that node, or any other point in any connected node, due to its convexity.



A path represented by a series of polygons illustrates not only the direction your pawn can travel, but also the space the pawn can occupy. Rather than needing to touch each point precisely along a waypoint-graph generated path as we would have had to do with the WayPoints, all of the information associated with the interface between nodes of the navigation mesh is provided to the AI. This benefits us twofold: it provides a more accurate and natural looking movement, in addition to eliminating corner cutting.

Benefits of NavMeshes

The overall graph density goes down as we can now use polygons instead of numerous nodes in a given area to represent a movable area. These are just a few of the benefits of this approach:

- Memory footprint reduction coincides with the decrease in nodes being stored
- As the density of the graph being indexed decreases, pathfinding time goes down
- Fewer nodes means less time wasted correcting cross-level pathing information

Additional benefits are listed in the next section.

Improved representation of the environment

A constant representation of the walkable area is advantageous for many other types of space searches an AI might do.

If you were to build a game that used squads, the process of determining a location to remain in squad formation is improved immensely, as you can actually check to see if the desired formation location is in the mesh and thus walkable or not. Previously, developers relied on the expensive work of finding the nearest path node to the formation location. Finally, the location's nearest path node is not necessarily very near the formation position, and often looks unrealistic.

Additionally, if your game uses walls which have the ability to be mantled over, the AI can perform this at any point along the wall rather than having to go to a discrete PathNode which represents a "mantle-able" location.

No more raycasts

Much of the raycasting AI does can be eliminated by using the data we generate into the navigation mesh. When an AI attempts to make an initial move, performed in order to determine if the AI can go directly to its destination and avoid pathfinding on the network, a raycast is performed.

There are two reasons to why this is eliminated, both of which are cheaper than raycasting. If a point can be directly reached, in most cases it will be in the same polygon navigation mesh as the AI, then in most cases, it will be the same polygon navigation mesh the AI is currently standing in. From there it's only a matter of seeking the polygon which contains our goal, and identifying that it is in fact the same polygon and identifying that they are the same.

Moreover, the obstacle mesh serves as a backup on which we can perform a low-overhead linecheck to determine if we can directly reach an area.

The navigation mesh is a rough representation of the potential space the AI can walk on, so it would be a simple task to project onto the mesh and do a single raycast to correctly align the AI onto the visible geography, as opposed to the number of raycasts per frame `PHYS_Walking` does.

The potential to handle more crowd actors at a time by snapping them to the navigation mesh rather than doing collision checks against world geometry with WayPoints is another benefit. We can now handle more AI on screen at once, as the overhead for doing so is far lower.

Laying PathNodes on a map

WayPoints use PathNodes for navigation. We will start by creating a new simple map with PathNodes for our AI pawns to follow.

Getting ready

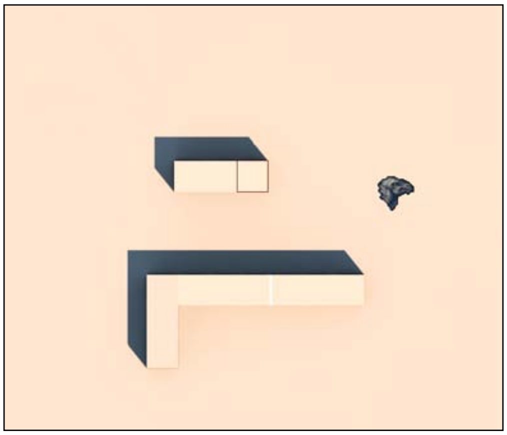
In the UDK editor, create a new map by going to **File** | **New Level**. When the pop-up for **Choose a map** template appears, select any of the lighting samples.

When your new map appears, we'll be ready to advance.

How to do it...

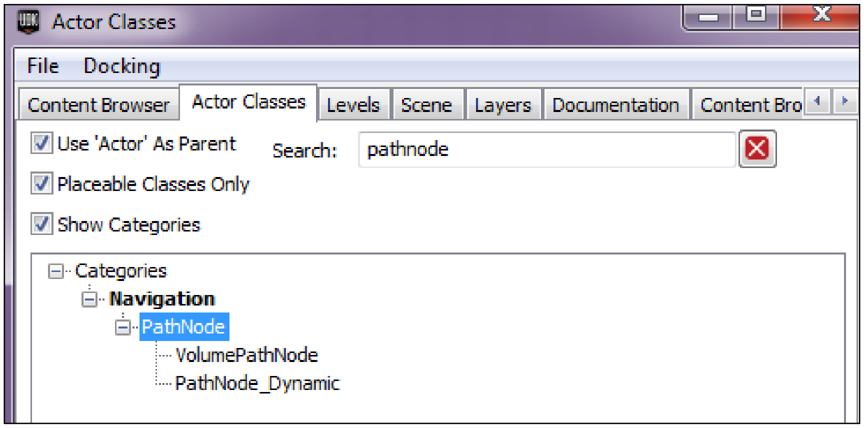
With our new map, we can begin by adding some geometry for our pawn to navigate around.

1. Create some simple geometry by adding some **Binary Space Partition** (**BSP**) brushes throughout the level. Left-click on the BSP brush in the center of your level, and hold the *Alt* key while dragging to create copies. Release the left mouse button and the *Alt* key for a copy to be made.



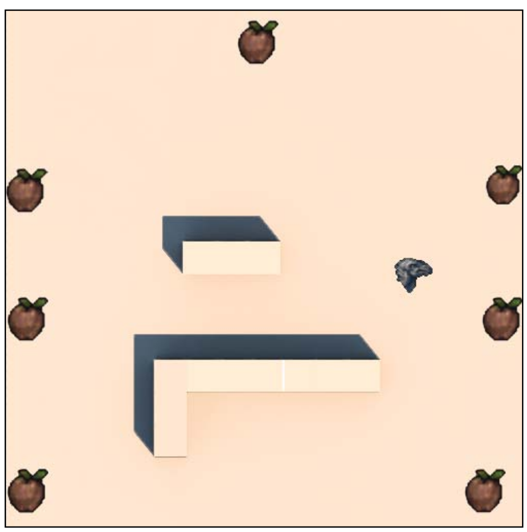
As you can see in the previous map, there is some simple geometry for the pawn to navigate through. The map offers us the ability to block line-of-sight, as we'll be needing that later on.

2. From here we can lay PathNodes, which our pawn will use to navigate. Open your **Actor Classes** browser and type in `PathNode` to bring up the available actors.



3. With the PathNode selected, left-click and drag it onto the map. It should spawn the node in the center of the map, along with the translation widget. Move it accordingly, while creating new copies along the way.

   I've aligned mine in such a way that the pawn will search the perimeter of the map, while still offering a path down the middle of the map, between the BSP.



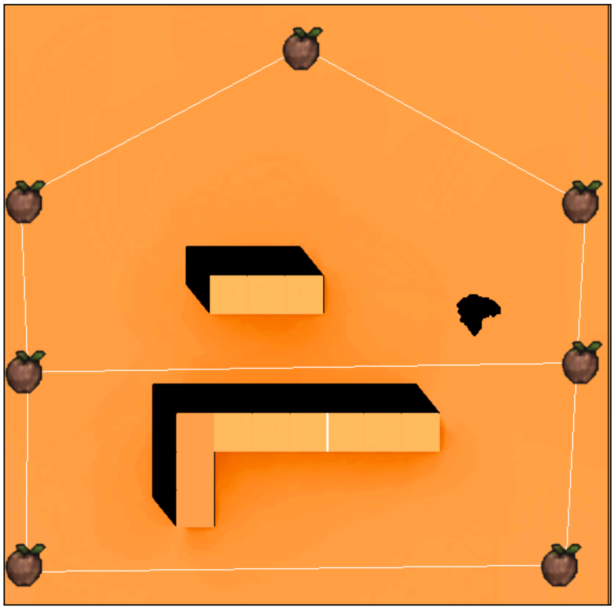
Also be sure that you have a PlayerStart navigation point on your map. There should be one in place on your map by default as UDK places it there automatically, but it's important to be aware of its exact location, as that will come into play later.

4. When we're finished laying PathNodes, we can rebuild our map. Because we have new BSP, we'll need to rebuild geometry, lighting, and PathNodes. Left-click on the **Build All** button at the top of the editor.

With everything rebuilt, you should now see a series of connected PathNodes, marked by the white lines between them. The editor may give you a warning stating that we haven't built the map with production lighting. Because this isn't a finished map, that's fine, as we're simply trying to speed up the rebuilding process by using preview lighting.

> If you don't see the lines, hit *P* on your keyboard to allow the editor to draw paths.

You should now have a map that looks like the following:



With that complete, we can move on to creating a map with our alternative navigation method, NavMeshes! You can view my copy of the map at any time, as it is included with this book and titled `Ch5_PathNodes2`.

How it works...

WayPoints are all about precision and allow a designer to have complete control over how pawns can traverse an environment. You tell them exactly where they can and cannot explore.

By dropping PathNode actors from the Actor Browser throughout the map we can create a network of paths for a pawn to traverse. Selecting the **Rebuild Paths** button from the editor connects all paths which are directly reachable, and creates a network for our pawn to travel along.

See also

- More information regarding the colored lines can be found at the Unreal Developer Network (UDN) (`http://udn.epicgames.com/Three/NavigationMeshPathDebugging.html`)

Laying NavMeshes on a map

With one means of navigation out of the way, we can work on building another map, albeit using NavMeshes for maneuvering through an environment.

Getting ready

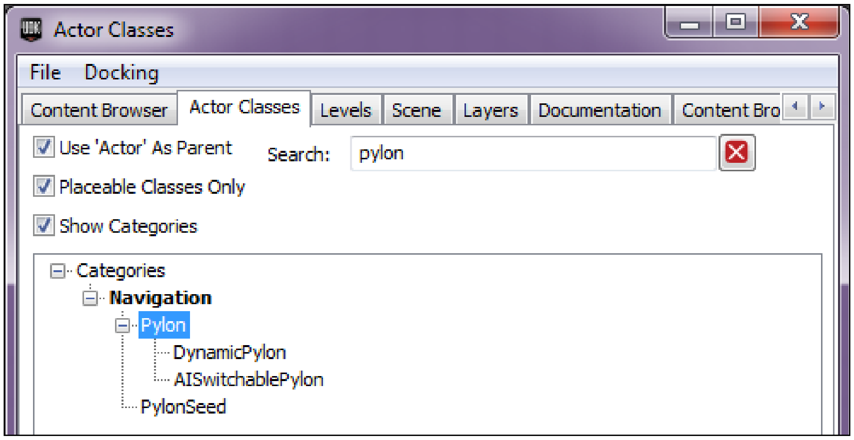
To keep things simple, we'll use our existing map, as the geometry also suits this lesson well. Start by deleting the PathNodes that you've created, so that we're starting from a clean slate.

With that done, we're ready to start laying our NavMeshes.

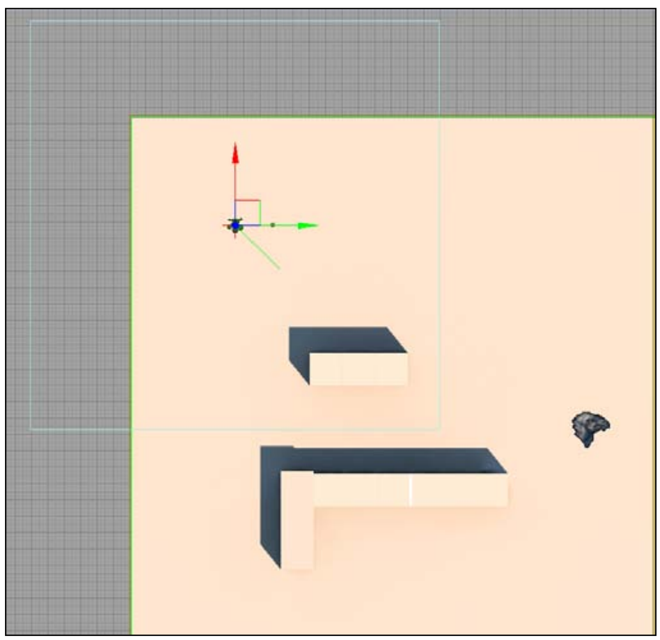
How to do it...

We've covered how NavMeshes work, along with their benefits and potential pitfalls. It's now time for us to actually implement them into our map so you can see the results for yourself!

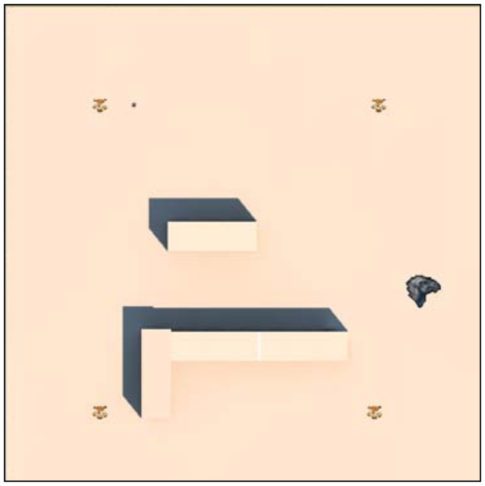
1. In our **Actor Classes** browser, search for **Pylon**. With **Pylon** highlighted, drag-and-drop it onto the map. It should appear in the center of the map, along with the translation widget, just as we saw with the PathNode.



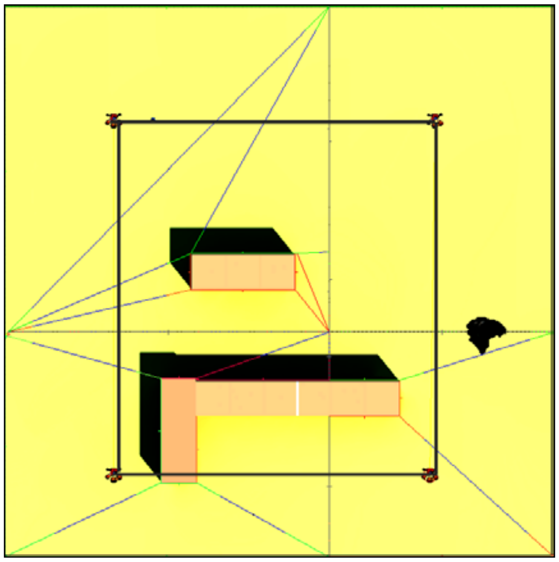
2. Drag the pylon toward the top-left corner of the map. You'll notice that the bounds for the pylon extend past the bounds of our map. That is fine.



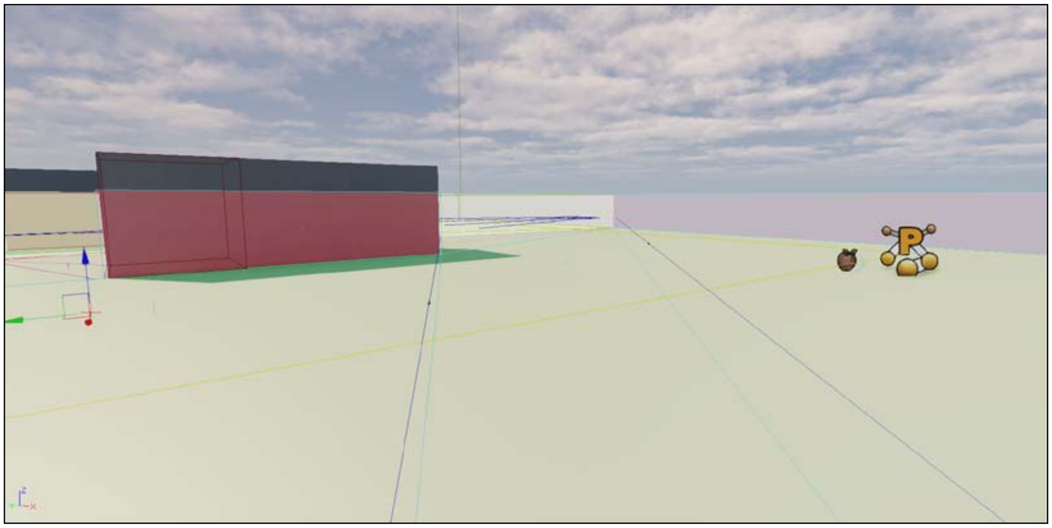
3. Create three more copies of the NavMesh, and spread them out in a similar manner, so that all corners of the map are covered. We want to make sure that none of our pylons fall into the boundary lines of another pylon, as the map will not build correctly. However, it is perfectly fine if the bounds for the pylons overlap.



4. Rebuild paths by selecting the **Rebuild Paths** button at the top of the editor. If you laid your pylons down correctly, you should see a series of polygons, illustrating the walkable area that your AI pawn may traverse, along with a yellow line (it appears as the thickest set of lines, shaped like a square in this figure) connecting the pylons, as they can be seen by line-of-sight.



If we switch to a perspective view, we can see that the height of our NavMeshes doesn't extend very far. This could potentially be an issue if we ever use a pawn that is taller than the NavMesh. Not to worry though, as we're going to correct this in our recipe!



You can find my version of this map with the included materials under the name `Ch5_NavMesh2`.

How it works...

While NavMeshes don't offer the precise control that PathNodes do, they do decrease the memory footprint, and allow for more natural movements.

We lay NavMeshes on a map in the same manner as we do with PathNodes: by dragging them onto the map from the Actor Browser. Rebuilding paths allows the mesh to create a network, so that the pawn doesn't have to calculate it at runtime and can easily avoid obstacles.

Adding a scout to create NavMesh properties

The height for a NavMesh is actually not found in the properties for the NavMesh itself, oddly enough. Unreal relies on a `Scout` class to determine this. The `Scout` class is designed to be a basic pawn with enough logic just to navigate around your map and see if it can get from one node to another, and therefore generating pathfinding information.

If we were to ever use a pawn whose height is greater than that of the scout's NavMesh, then upon spawning into a map, the pawn would instantly fall asleep and refuse to move. We're going to resolve this before it ever becomes an issue by creating our own `Scout` class.

Getting ready

We're going to extend from `Scout.uc` and create our own version of the `Scout` class. This `Scout` class is essentially what UDK uses for measuring how high the walls need to be in order to be used by the engine's navigation system. If our scout class is smaller than the size of our pawn, then the pawn will not be able to maneuver around the map correctly, as the maximum wall height recognized by our navigation image is based on the height of our scout.

Start off by creating a new class called `TutScout` and have it extend from `Scout`.

```
class TutScout extends Scout;
```

We're ready to begin working on our new class.

How to do it...

We're only going to alter the default properties for our custom `Scout` class, as all of the methods inside of it suit our needs well.

1. In your default properties, add the following code:

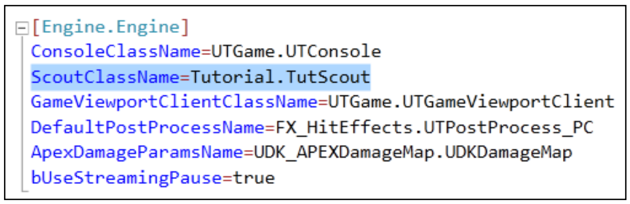
```
defaultproperties
{
  PathSizes.Empty
  /** Clears out any paths that may previously have been
      there. We will be using the size of our pawn as a
      template for how tall and wide our paths should
      be */
  PathSizes.Add((Desc=Human,Radius=180,Height=330))
  NavMeshGen_EntityHalfHeight=165

/** Subtract this from our MaxPolyHeight to get the final
  height for our NavMesh bounds */
  NavMeshGen_StartingHeightOffset=140

/** This number needs to be larger than the size of your
  default pawn */
  NavMeshGen_MaxPolyHeight=175
}
```

2. The current setting of `175` for `NavMeshGen_MaxPolyHeight` may cause some issues for us if we use a pawn larger than our default pawn. Let's increase this value to `300`.

```
/** This number needs to be larger than the size of your
  default
  pawn
NavMeshGen_MaxPolyHeight=300
```

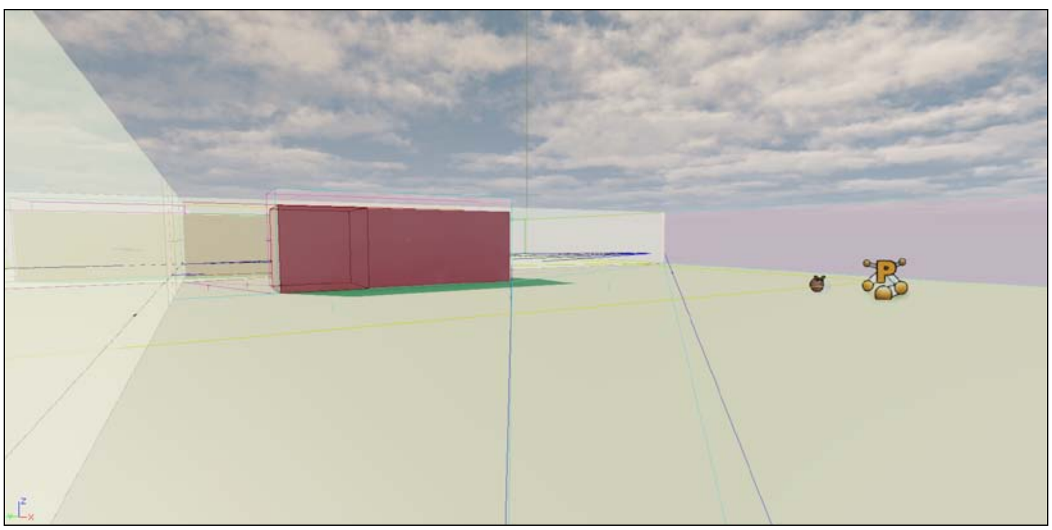
We need to tell the engine to use our `TutScout` class instead of the engine's default `Scout` class.

3. In your `DefaultEngine.ini`, change `ScoutClassName=UTGame.UTScout` to `ScoutClassName=YourGamePackage.YourScoutClassName`.



```
[Engine.Engine]
ConsoleClassName=UTGame.UTConsole
ScoutClassName=Tutorial.TutScout
GameViewportClientClassName=UTGame.UTGameViewportClient
DefaultPostProcessName=FX_HitEffects.UTPostProcess_PC
ApexDamageParamsName=UDK_APEXDamageMap.UDKDamageMap
bUseStreamingPause=true
```

4. Go back into the UDK editor, open up our map again, and rebuild paths. You should now see that our NavMesh's geometry height has increased quite a bit, extending over the top of our BSP!



How it works...

NavMeshes use a `Scout` class to determine its default properties for things such as height. If any pawn we use on our map has a larger collision cylinder than the scout, then it will not able to successfully pathfind and make use of our NavMeshes.

By creating our own `Scout` class and altering its properties so that it allows for larger pawns on the map, we can avoid potential errors in the future. To ensure that the engine makes use of our custom `Scout` class we needed to make a change to the `DefaultEngine.ini` file with the editor.

Adding an AI pawn via Kismet

With our means for navigating a level out of the way, we can finally work on adding a pawn who will take advantage of the things we've built, and allow the pawn to wander around the level. Later on we're even going to add functionality so that it follows us around the map.

Getting ready

We're going to create a new bot, which is really just an AI controller for pathfinding, by extending from UDKBot.

```
Class TutBot extends UDKBot;
```

As we move along, we'll begin to add more functionality to it such as the ability to wander using the PathNodes or NavMeshes, as well as follow our pawn.

How to do it...

One of Kismet's many useful functions is the ability to spawn bots and pawns. In *Chapter 8, Miscellaneous Recipes* we'll cover how to do spawn objects from code, but for now we'll stick to Kismet. This recipe
is one that we'll have to use frequently throughout the rest of this chapter as well.

1. As always, we'll want to add debugging information. This is more important than ever, as there are a number of things that could go wrong within this class and which may cause confusion. Start by adding a simple log function that allows us to see whether or not our pawn is ever even spawned.

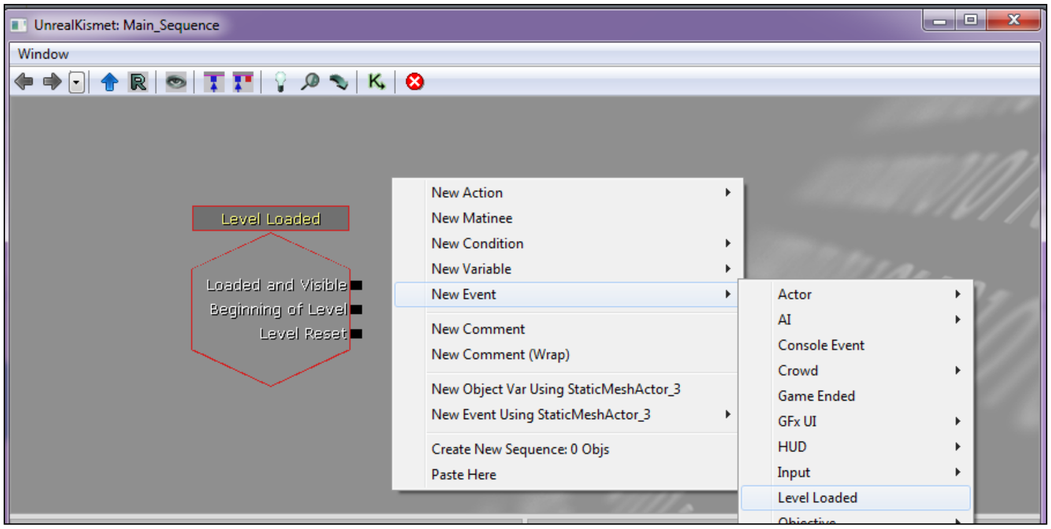
```
/** Lets us know that the class is being called, for
  debugging purposes */
simulated event PostBeginPlay()
```

```
{
  super.PostBeginPlay();
  'Log("TutBot up");
}
```

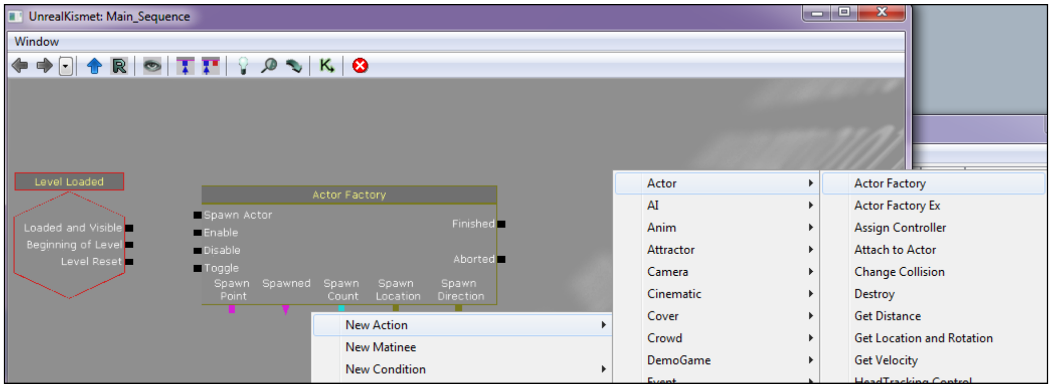
That's all we're going to add to the pawn right now. It has plenty of functionality from the `UDKBot` class, so we're going to leave it alone for now. Right now our focus is on getting this pawn to spawn within our map.

2. Open up the map you made with the PathNodes. We're going to use Kismet for the first time to create a spawn point for our custom bot.
3. Open Kismet by selecting the Kismet icon within the UDK editor.

4. When the Kismet dialog box appears, right-click anywhere in the blank space and you'll notice that the submenus appear. Scroll over to **New Event** | **Level Loaded**, and left-click to create a new Level Loaded node.

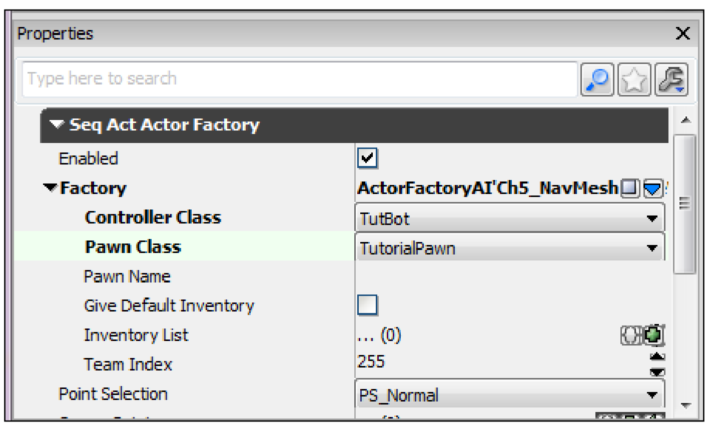


5. Follow the same procedure for creating an Actor Factory by right-clicking on a blank space on the canvas, and selecting **New Action** | **Actor** | **Actor Factory**.

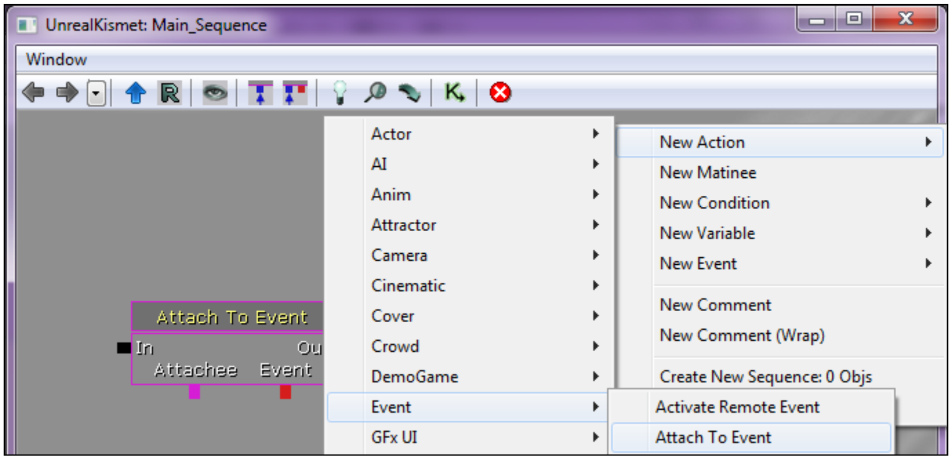


6. Right-click on your Actor Factory to bring the properties for the node. Under **Seq Act Actor Factory**, there is a dropdown for **Factory**. Left-click on the downward blue arrow marked **Create New Object** and select **ActorFactoryAI** from the top of the list.

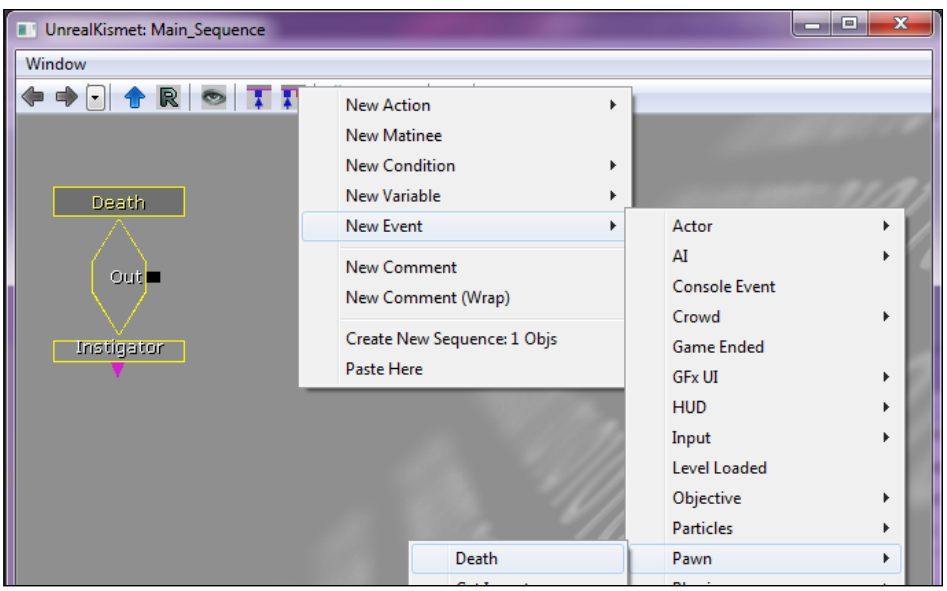
7. A new series of properties for **Factory** should appear. For **Controller Class**, select **TutBot**. For **Pawn Class**, we'll just use our **TutorialPawn**. Our Actor Factory is now told to spawn our newly created TutBot when we call it into action.



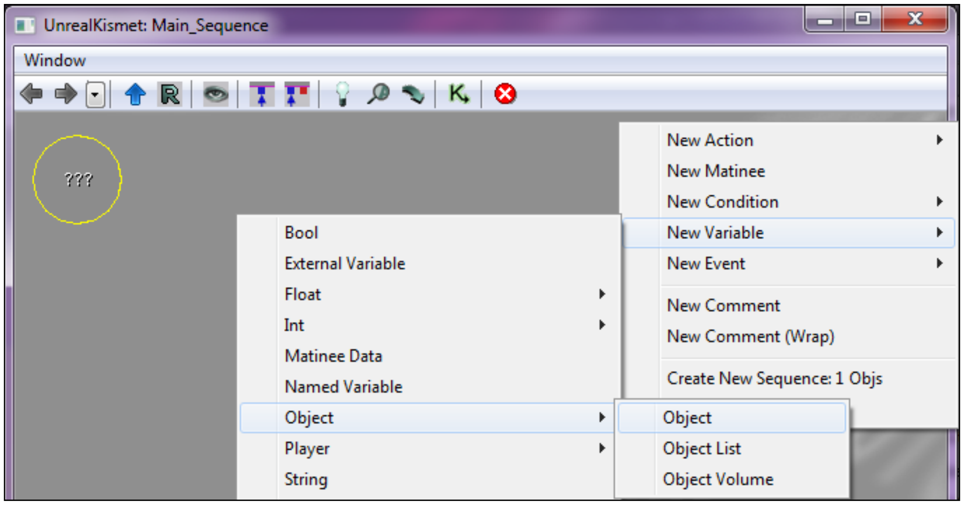
8. We now need to attach an event for our Actor Factory using Kismet. Again, right-click on a blank space on the canvas and select **New Action** | **Event** | **Attach to Event**.



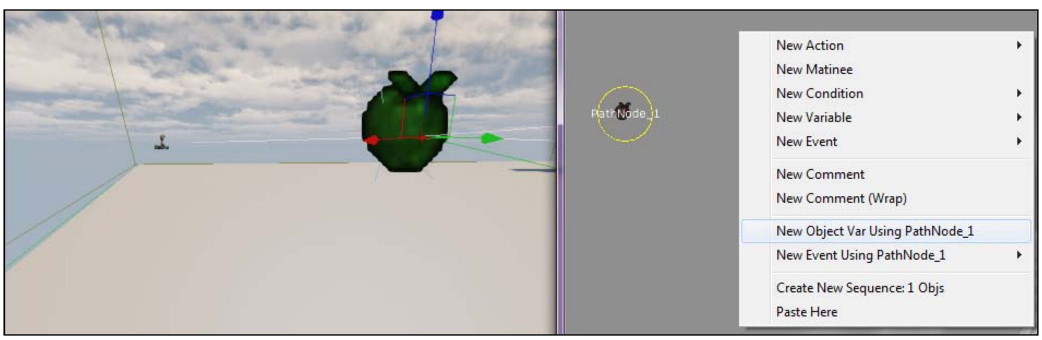
9. The event we are going to be connecting to this is the AI pawn's death. Should the AI pawn die, we'd like for it to respawn. Therefore, we need to create the event of the actual pawn dying. In the canvas, right-click and select **New Event** | **Pawn** | **Death**.



10. Only two more steps to go with Kismet. From the **Spawned** node in our **Actor Factory**, we need to attach an object. This object, quite simply, is our pawn.

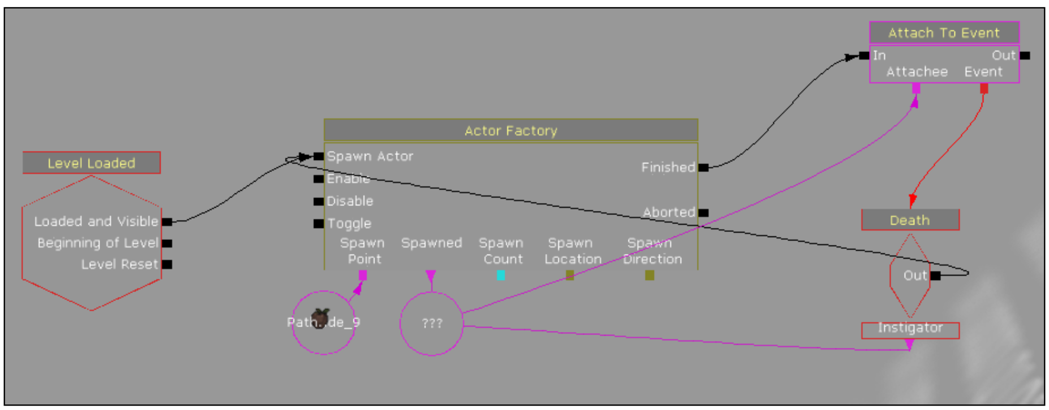


11. Now that our pawn has the ability to respawn within the map, we need to create an actual location for it to restart at. Left-click on any of the PathNodes in your map. Once it is highlighted, it can be now be referenced within Kismet. Right-click on an open space in Kismet and select select **New Object Var Using Pathnode_1** (could be 2, 3, 4, and so on).



Our pawn will now spawn from that location. This also works with PlayerStart nodes too.

12. Our final step is to connect all of the nodes appropriately. In total, it should look like this:
    - Level Loaded | Loaded and Visible is connected to Actor Factory | Spawn Actor
    - PathNode is connected to Actor Factory | Spawn Point
    - ??? is connected to Actor Factory | Spawned
    - Death | Out is connected to Actor Factory | Spawn Actor
    - Death | Instigator is connected to ???
    - Attach To Event | Event is connected to Death
    - Actor Factory | Finished is connected to Attach To Event | In
    - ??? is connected to Attach To Event | Attachee



13. Rebuild the map, and press **PIE** or **Play** | **In Editor** to play in the editor. You should now see another pawn on the map! Shoot it a bit to destroy it, and watch as it spawns back to life at the same node.

> Be sure to change **View** | **World Properties** | **Game Type** so that it uses our **TutorialGame**.
>
> Perform these same tasks in your map using NavMashes as well, otherwise your new AI pawn won't spawn.

How it works...

There are numerous ways to spawn bots on a level, including UnrealScript, making a pawn placeable, or using Kismet. To make things easy for us, we've gone the Kismet route, which allows us to quickly adjust the spawn location, pawn, and controller from within the editor.

We can spawn our bot from any PathNode or PlayerStart node on a map by using it as an object within Kismet. Moreover, the ActorFactory within Kismet allows us to select the controller and pawn, which will be spawned at that location.

While there are a number of ways to manipulate how our bots spawn, such as having multiple bots at once, creating a timer to have them spawn at a set rate, or only allowing a certain number of pawns on the map at once, we've decided to keep it simple and only have one bot spawn immediately after the map is loaded. The **Level Loaded** node makes this possible.

Essentially I've just used Kismet in this chapter, so that we can quickly iterate on what we've done and make changes to the pawn within the editor.

If we did this in code, we'd have to close the editor, go to the IDE, and change the code so that it spawns the appropriate bot, along with the correct controller (AI) that we want to use. With Kismet we can simply use the ActorFactory to see all of the bots and controllers are at our disposal, then press *Enter* to have them spawn.

Allowing a pawn to wander randomly around a map

With our PathNodes laid throughout our map and our bot now having the ability to spawn on the map, we're ready to start adding functionality to the bot so that it has artificial intelligence.

Getting ready

Load up your IDE and prepare to create a new class.

How to do it...

This recipe will be the starting point for our topic on AI. We'll start by simply creating a bot that randomly wanders around a map before moving onto more advanced things, such as creating a bot which takes advantage of our pathfinding system.

1. We're going to start by creating a new class for our bot, called simply, `WanderBot`. We don't need all of the complicated functionality behind the UDK and UT bot, so we'll be extending from GameAIController, and only add what we need.

```
class WanderBot extends GameAIController;
```

```
We only need one global variable, and that's TargetLocation.
```

```
var Vector TargetLocation;
```

2. We'll need to override the GameAIController's `PostBeginPlay()` function and add the `SetTimer()` function. This tells our pawn to run the `MoveRandom()` function every 2.5 seconds. Without this, our bot would constantly be searching for a new target location without ever taking a break.

```
/*========================================================

 * Called right after the map loads
=========================================================*/
simulated event PostBeginPlay()
{
  /** Calls all of the PostBeginPlay functions from
    parent classes */
  super.PostBeginPlay();
  /** Frequency at which the bot will begin looking for a
    new path */
  SetTimer(2.5, true,'MoveRandom');
}
```

The timer creates a more natural movement. For example, if you wanted your bot to move to a random location, and twiddle his thumbs for 5 seconds before contusing, this function is what would allow that to happen.

3. The `Possess` event tells our pawn to begin moving as soon as the map loads. `SetMovementPhysics()` is key here, as it automatically makes the pawn walk. Without this our pawn would need some sort of impulse, such as receiving damage, before moving.

```
/*========================================================

 * Forces the pawn to begin moving as soon as the map
  loads
========================================================*/
event Possess(Pawn inPawn, bool bVehicleTransition)
{
  super.Possess(inPawn, bVehicleTransition);
  Pawn.SetMovementPhysics();
}
```

4. We need a function to handle all of the math for where our bot currently is, and where we want it to be. We're using local variables for the offset, as we won't need them outside of this function.

The `OffsetX` and `OffsetY` variables tell us how far to the left and right, and then forward and backward, we want our bot to move each time the function is called. Our target location is where we want to set the bot. We're simply taking our current pawn's location, and subtracting it from the offset, that is, where we want the bot to be going. Each time `MoveRandom()` is called, it will pick a random integer between the minimum and maximum values you've set.

At the end of this function we're informing the bot to go to the **wander state**.

```
/*========================================================
* Math for our wandering state
========================================================*/
function MoveRandom()
{
  local int OffsetX;
  local int OffsetY;

  // set a random number for our X value
  OffsetX = Rand(1300)-Rand(700);

  // set a random number for our Y value
  OffsetY = Rand(1100)-Rand(1100);
  // distance left or right of the pawn
  TargetLocation.X = Pawn.Location.X + OffsetX;
  // distance in front of or behind the pawn
  TargetLocation.Y = Pawn.Location.Y + OffsetY;
  /** prevents the pawn from quickly aiming up and down
  the Z axis while moving */
  TargetLocation.Z = Pawn.Location.Z;

  // move to the random location
  GoToState('Wander');
}
```

5. The wander state only has one function, and that is `MoveTo(TargetLocation)`. The math for the target location (a vector) was previously done in our `MoveRandom()` function.

```
/*======================================================
* Tells the pawn to start wandering
========================================================*/
state Wander
{
  Begin:

  // move to a location (vector)
  MoveTo(TargetLocation);
}
```

6. Our final addition to this class is in the default properties. We'll add this to all of our AIController classes, as it tells the game that the pawn is either a player, or a player-bot.

```
defaultproperties
{
  // Pawn is a player or a player-bot
  bIsPlayer = true
}
```

We'll need to make some changes to our `TutorialPawn` class before we can advance. Because our `TutorialPawn` class was previously just controlled by us, we didn't have to worry about falling off ledges. AIControllers will now be using the pawn, so we need to add some code to instruct it not to fall off the ledges, or the edge of our map.

7. In the default properties block for our `TutorialPawn.uc` file, add these two lines:

```
defaultproperties
{
  bAvoidLedges=true // don't get too close to ledges
  bStopAtLedges=true // if bAvoidLedges and
  bStopAtLedges, Pawn doesn't try to walk
  along the edge at all
}
```

8. Rebuild scripts and start up the editor.

In our `Ch5_PathNodes2.udk` map and in Kismet, change the player controller to **WanderBot** and pawn to use our **TutorialPawn**. Press the **PIE** button to play in the editor and you should see your bot on the map. After one moment he'll begin to wander from place to place!

How it works...

This is about as simple as bot AI can ever get, and therefore, is an excellent starting point for us. We're simply taking our bot's current location and then informing the bot of where we want it to be. This math is done by taking a random integer, clamping the minimum and maximum values, and then applying it to where we want our bot to be.

After our bot reaches its target location, we then instruct it to wait for 2.5 seconds before deciding on a new location to reach.

We also found it necessary to add two booleans in the default properties block for our **TutorialPawn**, as we want to prevent our pawn from falling off ledges.

There's more...

There are two console commands which will make you understand how the bots operate and what is going through their mind.

While playing a game within the editor, press the console key (~ or *Tab*) to bring up the console. Enter the command `ViewBot` to change your perspective to that of the bot. Type `ViewSelf` to bring it back to your pawn's camera.

`ShoweDebug` will display all of the debug features for your pawn. This is extremely useful for displaying information such as which state your pawn is in, its next goal, a ray trace toward the next goal, and whether or not it has detected any enemies.

Making a pawn patrol PathNodes on a map

Our movement in the last recipe was very random, but we didn't have much control over where the bot went. With PathNodes, we can have precise control over where the bot can and cannot go.

We'll start by creating an AI Controller that allows our bot to wander from PathNode to PathNode in the order they were laid on the map.

Getting ready

Open your IDE and prepare to create a new class.

How to do it...

We've covered some simple AI up until this point, but now we want to take advantage of the PathNodes we've laid on a map. In this recipe we'll have our pawns maneuver the map using the scripts we've written.

1. Create a new class called `PatrolNodeBot` and have it extend from `UDKBot`. We're extending from UDKBot because we'll need the PathNode functionality included with it.

   ```
   class PatrolNodeBot extends UDKBot;
   ```

2. We'll need three variables to go along with it. `WayPoints` will store our PathNodes in an array, while `_PathNode` is the number (integer) of PathNodes on our map. `CloseEnough` is an integer that defines how close our pawn needs to be to a node before it starts looking for a new one.

   ```
   // PathNode Array
   var array<Pathnode>  WayPoints;
   ```

```
// declare it at the start so you can use it throughout the script
Var int _PathNode;
/** Distance our pawn needs to get to the node before it starts
    looking for a new one */
var int CloseEnough;
```

3. We need to override `PostBeginPlay()` in our class as well. All we're doing here is storing all of the PathNodes on a map to our array with the line `WayPoints.AddItem( Current );`.

```
/*=========================================================
 * Called right after the map loads
=========================================================*/
simulated function PostBeginPlay()
{
  local PathNode Current;

  /** Calls all of the PostBeginPlay functions from
  parent classes*/
  super.PostBeginPlay();
  //add the PathNodes to the array
  foreach WorldInfo.AllActors(class'Pathnode',Current)
  {
    WayPoints.AddItem( Current );
  }
}
```

4. Just as we had for our last AI Controller, we need to add `event Posses`. This tells our pawn to start moving as soon as the map loads.

```
/*==========================================================
  * Forces the pawn to begin moving as soon as the map
  loads
=========================================================*/
event Possess(Pawn inPawn, bool bVehicleTransition)
  {
    super.Possess(inPawn, bVehicleTransition);
    Pawn.SetMovementPhysics();
  }
```

5. We also need to override `Tick()`, which is called every frame. All it does is check that our pawn exists, and if it does, call our `PathFind()` function each frame.

```
/*==========================================================
 * Called each frame
=========================================================*/
simulated function Tick(float DeltaTime)
{
```

```
  // Calls all of the Tick functions from parent classes
  super.Tick(DeltaTime);

  // If there is a pawn on the map....
  if (Pawn != None)
  {
    /** Then call this method, which initializes our
      Pathfinding*/
    PathFind();
  }
}
```

The `PathFind()` function is where all of the math for our pathfinding occurs. We're making use of that `Distance` variable (integer), and defining it as the difference in distance between our bot's current location and the location of the PathNodes in our array.

If we are within a predetermined distance (the `CloseEnough` variable), then we instruct the bot to move towards the next node in the array, as noted by `_PathNode++;`. If we've iterated through each of the nodes, then start back at 0. Once that is done we move to the `PathFinding state`.

```
/*========================================================
* The meat-and-potatoes of the class.
* This is where all of the logic occurs
========================================================*/
simulated function PathFind()
{
  local int Distance;

  /** Distance between our pawn's current location and
  the next
  PathNode in our array */
  Distance = VSize2D(Pawn.Location -
    WayPoints[_PathNode].Location);

  if (Distance <= CloseEnough)
  {
    // Interate through the next node in our array
    _PathNode++;
  }

  // If we've gone through all of the nodes in our
  //array...
  if (_PathNode >= WayPoints.Length)
```

```
    {
      // Then set it back to zero and begin again
      _PathNode = 0;
    }
    // Go to the Pathfinding state
    GoToState('Pathfinding');
  }
```

6. The `PathFinding state` has the prefix `Auto` to denote that this is the default state the bot will start in. It checks if there are PathNodes in our array, and if there is one, starts moving towards it.

> We use the function `MoveToward()` as it accepts an actor as a parameter, and therefore we begin to move toward the actor. If we wanted to use a vector instead, we would use the `MoveTo()` function.

```
/*========================================================
* Auto state for our PathNode navigation.
========================================================*/
auto state Pathfinding
{
  Begin:
  // Move to PathNode if one exists
  if (WayPoints[_PathNode] != None)
  {
  MoveToward(WayPoints[_PathNode], WayPoints[_PathNode],
  128);
  }
}
```

7. The final bit to add is in the default properties block. `CloseEnough` is the integer we used earlier, and we've seen `bIsPlayer` before as well.

```
DefaultProperties
{
  // Once we're within this man UU's of our Node....
  CloseEnough= 200
  // Pawn is a player or a player-bot
  bIsplayer = True
}
```

8. Load up our `Ch5_PathNodes2.udk` map, set Kismet to use our new controller and watch as the bot wanders from PathNode to PathNode in the same order you laid the nodes on the map!

How it works...

The key component here was adding PathNodes to our array. We created a variable to store our array (`WayPoints`) and then created an integer to iterate through each one (`_PathNode`). The first thing our pawn does upon spawning is add the PathNodes to our array through `waypints.AddItems(Current);` in `PostBeginPlay()`.

Once they are in our array, we then iterate through them, one by one, until we've gone through them all and determined that one of them is close enough, based on our `CloseEnough` variable.

Making a pawn randomly patrol PathNodes on a map

PathNodes are designed so that we can have complete control over how our bots progress through a map. The problem with having a bot patrol PathNodes in a set path is that it doesn't look very realistic, unless your bot is a guard patrolling a prison.

For that reason we want to create a bot that can patrol our PathNodes at random.

Getting ready

Open up your IDE and prepare to create a new class.

How to do it...

Our goal here is similar to what we did with our first recipe in the chapter. We want to create a bot that wanders again, but this time we want it to take advantage of the PathNodes we've laid throughout the map. This gives us far more control over the landscapes the bot can traverse, as opposed to the previous manner, which gave the bot access to nearly every location on the map.

1. Start by creating a new class called `RandomNodeBot`. Have it extend from `UDKBot`.

   ```
   class RandomNodeBot extends UDKBot;
   ```

   From here on, our class is identical to the `PatrolNodeBot` class in every way except for the `PathFind()` function.

2. Rather than iterate through each node, we're going to set `_PathNode=Rand(WayPoints.length);`, which selects a random PathNode in our array.

   ```
   /*========================================================
   * The meat-and-potatoes of the class.
   ```

```
* This is where all of the logic occurs
=======================================================*/
simulated function PathFind()
{
  local int Distance;

  /** Distance between our pawn's current location and
  the next
  PathNode in our array */
  Distance = VSize2D(Pawn.Location -
    WayPoints[_PathNode].Location);

  if (Distance <= CloseEnough)
  {
    // Head towards a random PathNode in our array
    _PathNode=Rand(WayPoints.length);
  }
  GoToState('Pathfinding');
}
```

3. Load our `Ch5_PathNodes2.udk map` and in your Kismet Actor Factory, set your controller and pawn to `RandomNodeBot` and `TutorialPawn` classes, as this tells the factory to now spawn those. You can do this by selecting the pull down, which will list the various classes available to us.
4. Press the **PIE** button and play in the editor. Your bot should now patrol from node to node!

How it works...

This was a very straightforward recipe, as we were only required to make changes to one function for our new functionality. Making use of UDK's `Rand()` function allowed us to pick a random PathNode along the array we created when the bot was spawned.

Allowing a pawn to randomly patrol a map with NavMeshes

We've been working with bots that operate on maps with PathNodes lately. Sometimes we may find that NavMeshes better suit our needs. In that case, we'll need a bot who can wander around a map while pathfinding. This prevents the bot from running into walls and objects along the way.

If the bot does collide with something along its journey, it will pick a different path.

Getting ready

Open your IDE and prepare to create a new class.

How to do it...

This recipe will be slightly different from our most recent one. Again, we're trying to take advantage of all that UDK offers in terms of pathfinding, so we'll be creating a bot which randomly patrols a map, but uses NavMeshes. NavMeshes are probably more common in UDK development at this point, due to their ease of use and flexibility, so this recipe can prove invaluable in future AI endeavors.

1. Create a new class called `PatrollingNavMeshBot`, and have it extend from `AIController`.

   ```
   class PatrollingNavMeshBot extends AIController;
   ```

2. We'll only need two variables here, one to store our temporary destination, and another to store our final one.

   ```
   var Vector TempDest;
   var vector FinalDest;
   ```

3. As usual, we'll need to add the `Possess()` function.

   ```
   /*========================================================
    * Forces the pawn to begin moving as soon as the map
     loads
     ======================================================*/
   event Possess(Pawn inPawn, bool bVehicleTransition)
   {
     super.Possess(inPawn, bVehicleTransition);
     Pawn.SetMovementPhysics();
   }
   ```

4. Our `PatrolNavMesh` state is where all of the functionality occurs in this class.

   We set it to ignore `SeePlayer` so that it doesn't pay attention to any other pawns on the map. The function `FineNavMeshPath()` is where we tell the bot to clear any paths it may have previously had, along with any constraints. Afterwards, we create constraints, `EnforceTwoWayEdges` and `FindRandom`, which prevent our bot from getting stuck, and also allow it to find a random path to our goal.

   Finally, we tell it to find a path to its destination with the `FindPath()` function.

   `FindNavMeshPath` is only called once we begin our state. Once it does begin, we define a random point on the map, draw a debug line to the point, in addition to a red sphere. This allows us to easily view what's going through our bot's mind.

Again, if our bot realized that it can't directly access the point on the map, it will draw a new one and move towards it. At the end of the function we tell the bot to rest for half a second, and then start from the beginning again.

```
/*========================================================
* Patrols a map's NavMeshes using direct movetoward if
  player is reachable and pathfinding if not.
=======================================================*/

auto state PatrolNavMesh
{
  // If we see a player or pawn, ignore it
  ignores SeePlayer;

  function bool FindNavMeshPath()
  {
    // Clear cache and constraints
    NavigationHandle.PathConstraintList = none;
    NavigationHandle.PathGoalList = none;
    NavigationHandle.bDebugConstraintsAndGoalEvals =
      true;


    /** this makes sure the bot wont wander into an area
    where it will get stuck */
    class'NavMeshPath_EnforceTwoWayEdges'.static.
      EnforceTwoWayEdges(NavigationHandle);

    /** Tells the bot to set a random goal.
    There are 2 optional
    variables you can pass, a float or int representing
    the range
    to search, and an int representing how many polys
    away he can
    move to */
    class'NavMeshGoal_Random'.static.FindRandom
    (NavigationHandle);
    // set his goal.

    // Find path
    return NavigationHandle.FindPath();
  }
  Begin:
  if(FindNavMeshPath())
  {
```

```
    NavigationHandle.SetFinalDestination
      (NavigationHandle.PathCache_GetGoalPoint());

    // The random point is any area within the NavMesh
    FinalDest = NavigationHandle.
      FinalDestination.Position;

    // Draw the line to our pawn
    DrawDebugLine(Pawn.Location, FinalDest,255,0,0,true);
    /** Draw a red sphere to illustrate the next location
    the bot
    will stop at */
    DrawDebugSphere(FinalDest,16,20,255,0,0,true);
```

The debug information is now drawn on screen so that we can see where our bot will go. This visualization makes it far easier to understand if our bot is handling our code correctly or not, especially the pathfinding.

5. The following code represents what goes on behind the scenes, or within our bot's mind as it is looking for a new path. All of the pathfinding code is as follows:

```
  // While our bot hasn't reached the random point yet...
  while(!pawn.ReachedPoint(FinalDest, none))
  {
    /** If the bot realizes it can't reach this point
        directly...*/

    if(!NavigationHandle.PointReachable(FinalDest))
    {
      // Get out of here and pick another point
      break;
    }
    // Otherwise...
    else
    {
      // Move to the random point
      MoveTo(FinalDest);
    }
    // Rest for (X) seconds before picking a new point
    Sleep(0.5);
  }
  // Start from the beginning again
  goto 'Begin';
  }
}
```

6. The final piece of the puzzle is the default properties block.

```
defaultproperties
{
  // Pawn is a player or a player-bot
  bIsPlayer = true
}
```

   With all of this in place, load up the `Ch5_NavMesh2.udk` map. Open up Kismet and select the Actor Factory that we created earlier. Select the pulldown which allows you to select your `PatrollingNavMeshBot` and `TutorialPawn` as the pawn and controller to be used.

7. Press the **PIE** button and play in the editor. Your bot should now wander around the map and draw red spheres around the map while doing so!

How it works...

NavMeshes allow for a more natural movement for AI bots. If the bot finds that it will collide with something along its journey it will pick a different point to reach.

Most of our bot's logic is handled in one function, in combination with a state. We start by clearing any paths or constraints our bot may have previously had, then declare some of our own. In our case, we want our bot to steer clear of edges and find a random path to move towards.

The bot performs pathfinding on the NavMesh to see if the random path will be blocked by some sort of geometry, and if it senses that this is true, the bot will return another random path. Once the bot reaches that point, it draws another path.

Making a pawn follow us around the map with NavMeshes

We started to see the benefit of using NavMeshes with our last recipe, as it allowed our bot to determine whether or not a destination was reachable before it began to move to it.

Now we're going to take it to the next step, and have our bot follow us around the map. This time however, when the bot detects that we are too far away, or it can't reach us directly, it will create a path of its own to reach us.

Getting ready

Open your IDE and prepare to create a new class.

How to do it...

We've covered enough wandering and patrolling up to this point. Why not create an AI which is more friendly, and can follow us around a map? I'm sure you've played a game where one character follows another, whether it is the shadow ninjas in *Ninja Gaiden* on the NES, or the president's daughter following Leon in the more recent *Resident Evil* franchise. Ever wonder how the characters do that? It's time to explain how.

1. Create a new class called `FollowBot`. Have it extend from `AI Controller`.

```
class FollowBot extends AIController;
```

2. We're going to need four variables for this one. Some of them we've already seen from previous examples, but they are all pretty well explained in the comments.

```
/** Whatever target we'd like to use. In this case, our
      pawn */
var Actor target;

/** The temporary destination the pawn will be headed
    toward
  (PathFinding) */
var() Vector   TempDest;

/** PathNode Array */
var array<Pathnode> WayPoints;

/** Distance our pawn needs to get to the node before it
    starts
  looking for a new one */
var int CloseEnough;
```

3. Add the event for `Possess`, just as we've done with our other bots.

```
/*=======================================================
 * Forces the pawn to begin moving as soon as the map
  loads
=======================================================*/
event Possess(Pawn inPawn, bool bVehicleTransition)
{
  super.Possess(inPawn, bVehicleTransition);
  Pawn.SetMovementPhysics();
}
```

4. Just as we added PathNodes to our array for our bots to use, we need to add the WayPoints that NavMeshes use to our array. Add the `PostBeginPlay()` function to your class.

```
/*=========================================================
 * Called right after the map loads
=========================================================*/
simulated function PostBeginPlay()
{
  local PathNode Current;

  /** Calls all of the PostBeginPlay functions from
  parent classes */
  super.PostBeginPlay();

  // Add the PathNodes to the array
  foreach WorldInfo.AllActors(class'Pathnode',Current)
  {
    WayPoints.AddItem(Current);
  }
}
```

5. Let's add our `Idle` state, which informs our bot that we want it to remain still until it detects another pawn to follow.

   This is the first time that we use two states in our class. We use the keyword `auto` to denote that this is the state that our pawn will be in as soon as it spawns on the map. Previously, we told our bot to ignore `SeePlayer`, but now we want it to be actively aware of other players on the map. We're setting our target variable to the first pawn the bot sees. This target variable is what we'll be passing around to the other functions for finding the location we want our bot to head towards.

```
/*=========================================================
 * We want the pawn to remain still until our player
 * crosses its path
=========================================================*/
auto state Idle
{
  event SeePlayer (Pawn Seen)
  {
    super.SeePlayer(Seen);
    target = Seen;

    /** If our pawn is spotted, go to the function that
        follows him */
    GotoState('Follow');
  }
  Begin:
}
```

Now it's time to create our `follow state`. We're going to write the function for `FindNavMeshPath()` first, which is nearly identical to the one we used in our previous recipe. However, we are creating new constraints here. We're passing in our target variable, which was defined previously in our idle state, as the pawn our bot has just seen. We then tell the bot to find the most appropriate path to our pawn by returning the `NavigationHandle.FindPath` function.

```
/*=========================================================
 * Nav Mesh code for following our pan
========================================================*/
state Follow
{
  ignoresSeePlayer;
  functionboolFindNavMeshPath()
  {

    /** Clear cache and constraints (ignore recycling
        for the  moment) */
    NavigationHandle.PathConstraintList = none;
    NavigationHandle.PathGoalList = none;

    // Create constraints
    class'NavMeshPath_Toward'.static.TowardGoal
    (NavigationHandle,target);

    class'NavMeshGoal_At'.static.AtActor
    (NavigationHandle, target,32);

    // Find path to our pawn
    return NavigationHandle.FindPath();
  }
```

6. The second half of our follow state is where the state actually begins.

   If we can reach our pawn directly, then we just move toward it without needing to perform any sort of pathfinding. Otherwise, if the bot determines that it can't reach our pawn directly then it's time to perform pathfinding. The bot will now create a path to its target Actor.

   With the `DrawPathCache()` function, we draw lines to each point in our path cache, from our current location to where our bot is. Afterwards, we move to the first node on that path. The two lines of debug code that follow determine the color of our lines, as well as the color of the sphere at our next location.

   Once we've reached our pawn, go back to the beginning of the state again. If at any point we've determined that we can't reach the pawn, whether through pathfinding or not, we go back to our idle state to prevent an infinite loop, which would create a game crashing bug.

```
    Begin:

    // If we can reach our pawn....
    if(NavigationHandle.ActorReachable(target))
    {
      /** Clear any debug lines that would otherwise be
          drawn*/
      FlushPersistentDebugLines();

      /** Move directly toward our pawn */
      MoveToward(target,target);
    }
    else if( FindNavMeshPath())
    {
      // Our end goal is to reach our pawn
      NavigationHandle.SetFinalDestination
        (target.Location);

      /** Clear any debug lines that would otherwise be
          drawn */

      FlushPersistentDebugLines();
      // Draw lines for how to reach our pawn
      NavigationHandle.DrawPathCache(,TRUE);

      // Move to the first node on the path
      if(NavigationHandle.GetNextMoveLocation
        (TempDest, Pawn.GetCollisionRadius()))
      {
        // Draw the line to our pawn
        DrawDebugLine(Pawn.Location,TempDest,255,0,0,true);

        /** Draw a red sphere to illustrate the next
            location the bot will stop at */
        DrawDebugSphere(TempDest,16,20,255,0,0,true);

        /** Move directly to this red sphere, without
            pathfinding */

        MoveTo(TempDest, target);}
    }
    else
    {
      /** We can't follow the pawn, so exit this state
          otherwise we'll enter an infinite loop. */
      GotoState('Idle');
    }
    // Go back to the beginning of this state
    goto 'Begin';
  }
```

How it works...

The major benefit of using NavMeshes is that they can provide for more natural movements. Furthermore, they allow for pathfinding if our bot cannot find a direct route to its target. This is particularly useful for areas where there are a large number of narrow walls, such as a maze.

If our pawn can't find a direct route to its target, whether it is a vector when we are using the `MoveTo()` function or actor when we use the `MoveToward()` function, pathfinding can automatically generate an efficient route.

By setting a list of constraints, such as avoiding edges of walls or finding a random target, we can create natural movements and prevent our pawn from getting stuck. Along the way we have our pawn create a list of points it needs to reach before it can arrive at our target, in addition to displaying a red sphere at its next destination for debugging purposes.

See also

Additional resources for this chapter can be found at the following locations:

- `http://udn.epicgames.com/Three/AIAndNavigationHome.html`
- `http://romerounrealscript.blogspot.com/2012/04/ai-navigation-in-unrealscript.html`
- `http://x9productions.com/blog/?page_id=521`

6

Weapons

In this chapter, we will be covering the following recipes:

- Creating a gun that fires homing missiles
- Creating a gun that heals pawns
- Creating a weapon that can damage over time
- Adding a flashlight to a weapon
- Creating an explosive barrel
- Creating a landmine

Introduction

Weapons in UDK are inventory items which can be handled by the player, and are generally used to fire a projectile. On the surface, the default weapon system found in Unreal Engine 3 may appear to be catered to create various types of guns that are common in most FPS games; but it's actually pretty easy to create various sorts of weapons and usable inventory items, which may be found in other types of games, such as healing projectiles, bombs, landmines, or flashlights, as in the case with *Alan Wake*.

Rather than reinvent the wheel, we're simply going to extend from the shock rifle for many of our weapons, as its default abilities offer quite a bit of flexibility. We could dedicate an entire book to creating an excellent base weapon and archetypes to extend from, but for simplicity's sake we're going to create some practical examples in the following chapter. This knowledge will allow you to easily create other similar types of weapons and devices.

Creating a gun that fires homing missiles

UDK already has a homing rocket launcher packaged with the dev kit (`UTWeap_RocketLauncher`). The problem however, is that it isn't documented well; it has a ton of excess code only necessary for multiplayer games played over a network, and can only lock on when you have loaded three rockets.

We're going to change all of that, and allow our homing weapon to lock onto a pawn and fire any projectile of our choice. We also need to change a few functions, so that our weapon fires from the correct location and uses the pawn's rotation and not the camera's. These are the same functions which we added to our `ShockRifle` class in *Chapter 3*, *Scripting a Camera System*. We'll need to create two classes for this first, so let's get started!

Getting ready

As I mentioned earlier, our main weapon for this chapter will extend from the `UTWeap_ShockRifle`, as that gun offers a ton of great base functionality which we can build from.

Let's start by opening your IDE and creating a new weapon called `MyWeapon`, and have it extend from `UTWeap_ShockRifle` as shown as follows:

```
class MyWeapon extends UTWeap_ShockRifle;
```

How to do it...

We need to start by adding all of the variables that we'll be needing for our lock on feature. There are quite a few here, but they're all commented in pretty great detail. Much of this code is straight from UDK's rocket launcher, that is why it looks familiar. In this recipe, we'll be creating a base weapon which extends from one of the Unreal Tournament's most commonly used weapons, the shock rifle, and base all of our weapons from that.

1. I've gone ahead and removed an unnecessary information, added comments, and altered functionality so that we can lock onto pawns with any weapon, and fire only one missile while doing so.

   ```
   /*********************************************************
   * Weapon lock on support
   ********************************************************/
   /** Class of the rocket to use when seeking */
   var class<UTProjectile> SeekingRocketClass;

   /** The frequency with which we will check for a lock */
   var(Locking) float    LockCheckTime;
   ```

```
/** How far out should we be considering actors for a lock */
var float    LockRange;

/** How long does the player need to target an actor to lock on to
it*/
var(Locking) float    LockAcquireTime;

/** Once locked, how long can the player go without painting the
object before they lose the lock */
var(Locking) float    LockTolerance;

/** When true, this weapon is locked on target */
var bool          bLockedOnTarget;

/** What "target" is this weapon locked on to */
var Actor         LockedTarget;

var PlayerReplicationInfo LockedTargetPRI;

/** What "target" is current pending to be locked on to */
var Actor         PendingLockedTarget;

/** How long since the Lock Target has been valid */
var float           LastLockedOnTime;

/** When did the pending Target become valid */
var float         PendingLockedTargetTime;

/** When was the last time we had a valid target */
var float         LastValidTargetTime;

/** angle for locking for lock targets */
var float          LockAim;

/** angle for locking for lock targets when on Console */
var float          ConsoleLockAim;

/** Sound Effects to play when Locking */
var SoundCue       LockAcquiredSound;
var SoundCue       LockLostSound;

/** If true, weapon will try to lock onto targets */
var bool bTargetLockingActive;

/** Last time target lock was checked */
var float LastTargetLockCheckTime;
```

2. With our variables in place, we can now move onto the weapon's functionality. The `InstantFireStartTrace()` function is the same function we added in our weapon during our *Chapter 3, Scripting a Camera System*. It allows our weapon to start its trace from the correct location using the `GetPhysicalFireStartLoc()` function.

   As mentioned before, this simply grabs the rotation of the weapon's muzzle flash socket, and tells the weapon to fire projectiles from that location, using the socket's rotation. The same goes for `GetEffectLocation()`, which is where our muzzle flash will occur.

> The `v` in vector for the `InstantFireStartTrace()` function is not capitalized. The reason being that vector is actually of `struct` type, and not a function, and that is standard procedure in UDK.

```
/*****************************************************
* Overriden to use GetPhysicalFireStartLoc() instead of
* Instigator.GetWeaponStartTraceLocation()
* @returns position of trace start for instantfire()
*****************************************************/
simulated function vector InstantFireStartTrace()
{
  return GetPhysicalFireStartLoc();
}

/*****************************************************
* Location that projectiles will spawn from. Works for secondary
fire on
* third person mesh
*****************************************************/
simulated function vector GetPhysicalFireStartLoc(optional vector
AimDir)
{
  Local SkeletalMeshComponent AttachedMesh;
  local vector SocketLocation;
  Local TutorialPawn TutPawn;

  TutPawn = TutorialPawn(Owner);
  AttachedMesh = TutPawn.CurrentWeaponAttachment.Mesh;
/** Check to prevent log spam, and the odd situation win
    which a cast to type TutPawn can fail */
if (TutPawn != none)
{
    AttachedMesh.GetSocketWorldLocationAndRotation
    (MuzzleFlashSocket, SocketLocation);
}
```

```
    return SocketLocation;
}

/*******************************************************
* Overridden from UTWeapon.uc
* @return the location + offset from which to spawn effects
(primarily tracers)
*******************************************************/
simulated function vector GetEffectLocation()
{
  Local SkeletalMeshComponent AttachedMesh;
  local vector SocketLocation;
  Local TutorialPawn TutPawn;
  TutPawn = TutorialPawn(Owner);
  AttachedMesh = TutPawn.CurrentWeaponAttachment.Mesh;
  if (TutPawn != none)
  {
    AttachedMesh.GetSocketWorldLocationAndRotation
    (MuzzleFlashSocket, SocketLocation);
  }
  MuzzleFlashSocket, SocketLocation);
  return SocketLocation;
}
```

3. Now we're ready to dive into the parts of code that are applicable to the actual homing of the weapon. Let's start by adding our debug info, which allows us to troubleshoot any issues we may have along the way.

```
********************************************************
* Prints debug info for the weapon
*******************************************************/
simulated function GetWeaponDebug( out Array<String> DebugInfo )
{
  Super.GetWeaponDebug(DebugInfo);

  DebugInfo[DebugInfo.Length] = "Locked:
  "@bLockedOnTarget@LockedTarget@LastLockedontime@
  (WorldInfo.TimeSeconds-LastLockedOnTime);
  DebugInfo[DebugInfo.Length] =
  "Pending:"@PendingLockedTarget@PendingLockedTargetTime
  @WorldInfo.TimeSeconds;
}
```

Here we are simply stating which target our weapon is currently locked onto, in addition to the pending target. It does this by grabbing the variables we've listed before, after they've returned from their functions, which we'll add in the next part.

4. We need to have a default state for our weapon to begin with, so we mark it as inactive.

```
/*******************************************************
* Default state. Go back to prev state, and don't use our
* current tick
*******************************************************/
auto simulated state Inactive
{
  ignores Tick;

  simulated function BeginState(name PreviousStateName)
  {
    Super.BeginState(PreviousStateName);

    // not looking to lock onto a target
    bTargetLockingActive = false;

    // Don't adjust our target lock
    AdjustLockTarget(None);
}
```

We ignore the tick which tells the weapon to stop updating any of its homing functions. Additionally, we tell it not to look for an active target or adjust its current target, if we did have one at the moment.

5. While on the topic of states, if we finish our current one, then it's time to move onto the next:

```
/*******************************************************
 * Finish current state, & prepare for the next one
*******************************************************/
  simulated function EndState(Name NextStateName)

  {
    Super.EndState(NextStateName);

    // If true, weapon will try to lock onto targets
    bTargetLockingActive = true;
  }
}
```

6. If our weapon is destroyed or we are destroyed, then we want to prevent the weapon from continuing to lock onto a target.

```
/*********************************************************
 * If the weapon is destroyed, cancel any target lock
*********************************************************/
simulated event Destroyed()
{

// Used to adjust the LockTarget.
  AdjustLockTarget(none);

  //Calls the previously defined Destroyed function
  super.Destroyed();
}
```

7. Our next chunk of code is pretty large, but don't let it intimidate you. Take your time and read it through to have a thorough understanding of what is occurring. When it all boils down, the `CheckTargetLock()` function verifies that we've actually locked onto our target.

   We start by checking that we have a pawn, a player controller, and that we are using a weapon which can lock onto a target. We then check if we can lock onto the target, and if it is possible, we do it. At the moment we only have the ability to lock onto pawns.

```
/******************************************************************
* Have we locked onto our target?
*****************************************************************/
function CheckTargetLock()
{
  local Actor BestTarget, HitActor, TA;
  local UDKBot BotController;
  local vector StartTrace, EndTrace, Aim, HitLocation,
  HitNormal;
  local rotator AimRot;
  local float BestAim, BestDist;

  if((Instigator == None)||(Instigator.Controller ==
  None)||(self != Instigator.Weapon) )
  {
    return;
  }
```

```
if ( Instigator.bNoWeaponFiring)
// TRUE indicates that weapon firing is disabled for this
pawn
{
  // Used to adjust the LockTarget.
  AdjustLockTarget(None);

  // "target" is current pending to be locked on to
  PendingLockedTarget = None;
  return;
}
// We don't have a target
BestTarget = None;
BotController = UDKBot(Instigator.Controller);

// If there is BotController...
if ( BotController != None )
{
  // only try locking onto bot's target
  if((BotController.Focus != None) &&
  CanLockOnTo(BotController.Focus) )
  {
    // make sure bot can hit it
    BotController.GetPlayerViewPoint
    ( StartTrace, AimRot );
    Aim = vector(AimRot);

    if((Aim dot Normal(BotController.Focus.Location -
    StartTrace)) > LockAim )
    {
      HitActor = Trace(HitLocation, HitNormal,
      BotController.Focus.Location, StartTrace, true,,,
      TRACEFLAG_Bullet);
      if((HitActor == None)||
      (HitActor == BotController.Focus) )
      {
        // Actor being looked at
        BestTarget = BotController.Focus;
      }
    }
  }
}
```

Immediately after that, we do a trace to see if our missile can hit the target, and check for anything that may be in the way. If we determine that we can't hit our target then it's time to start looking for a new one.

```
else
{
  // Trace the shot to see if it hits anyone
  Instigator.Controller.GetPlayerViewPoint
  ( StartTrace, AimRot );
  Aim = vector(AimRot);

  // Where our trace stops
  EndTrace = StartTrace + Aim * LockRange;

  HitActor = Trace
  (HitLocation, HitNormal, EndTrace, StartTrace,
  true,,, TRACEFLAG_Bullet);

  // Check for a hit
  if((HitActor == None)||!CanLockOnTo(HitActor) )
  {
    /** We didn't hit a valid target?  Controller
     attempts to pick a good target */
    BestAim = ((UDKPlayerController
    (Instigator.Controller)!=None)&&
    UDKPlayerController(Instigator.Controller).
    bConsolePlayer) ? ConsoleLockAim : LockAim;
    BestDist = 0.0;
    TA = Instigator.Controller.PickTarget
     (class'Pawn', BestAim, BestDist, Aim, StartTrace,
    LockRange);
    if ( TA != None && CanLockOnTo(TA) )
    {
      /** Best target is the target we've locked */
      BestTarget = TA;
    }
  }

  // We hit a valid target
  else
  {
    // Best Target is the one we've done a trace on
    BestTarget = HitActor;
  }
}
```

8. If we have a possible target, then we note its time mark for locking onto it. If we can lock onto it, then start the timer. The timer can be adjusted in the default properties and determines how long we need to track our target before we have a solid lock.

```
// If we have a "possible" target, note its time mark
if ( BestTarget != None )
{
  LastValidTargetTime = WorldInfo.TimeSeconds;

  // If we're locked onto our best target
  if ( BestTarget == LockedTarget )
  {
    /** Set the LLOT to the time in seconds since
     level began play */

    LastLockedOnTime = WorldInfo.TimeSeconds;
  }
```

Once we have a good target, it should turn into our current one, and start our lock on it. If we've been tracking it for enough time with our crosshair (`PendingLockedTargetTime`), then lock onto it.

```
  else
  {
    if ( LockedTarget != None&&(
    (WorldInfo.TimeSeconds - LastLockedOnTime >
    LockTolerance)||!CanLockOnTo(LockedTarget)) )
    {
      // Invalidate the current locked Target
      AdjustLockTarget(None);
    }

    /** We have our best target, see if they should
        become our current target Check for a new
        pending lock */
    if (PendingLockedTarget != BestTarget)

    {
      PendingLockedTarget = BestTarget;
      PendingLockedTargetTime =
      ((Vehicle(PendingLockedTarget) != None)
      &&(UDKPlayerController
      (Instigator.Controller)!=None)
      &&UDKPlayerController(Instigator.Controller).
      bConsolePlayer)
        ? WorldInfo.TimeSeconds + 0.5*LockAcquireTime
        : WorldInfo.TimeSeconds + LockAcquireTime;
    }
```

```
        /** Otherwise check to see if we have been
            tracking the pending lock long enough */
        else if (PendingLockedTarget == BestTarget
        && WorldInfo.TimeSeconds = PendingLockedTargetTime )
        {
          AdjustLockTarget(PendingLockedTarget);
          LastLockedOnTime = WorldInfo.TimeSeconds;
          PendingLockedTarget = None;
          PendingLockedTargetTime = 0.0;
        }
      }
    }
```

Otherwise, if we can't lock onto our current or our pending target, then cancel our current target, along with our pending target.

```
    else
    {
      if ( LockedTarget != None&&((WorldInfo.TimeSeconds -
      LastLockedOnTime > LockTolerance)||
      !CanLockOnTo(LockedTarget)) )
      {
        // Invalidate the current locked Target
        AdjustLockTarget(None);
      }

      // Next attempt to invalidate the Pending Target
      if ( PendingLockedTarget != None&&
      ((WorldInfo.TimeSeconds - LastValidTargetTime >
      LockTolerance)||!CanLockOnTo(PendingLockedTarget)) )
      {
        // We are not pending another target to lock onto
        PendingLockedTarget = None;
      }
    }
  }
```

That was quite a bit to digest. Don't worry, because the functions from here on out are pretty simple and straightforward.

9. As with most other classes, we need a `Tick()` function to check for something in each frame. Here, we'll be checking whether or not we have a target locked in each frame, as well as setting our `LastTargetLockCheckTime` to the number of seconds passed during game-time.

```
/*********************************************************
* Check target locking with each update
*********************************************************/
event Tick( Float DeltaTime )
```

```
{
  if ( bTargetLockingActive && ( WorldInfo.TimeSeconds >
  LastTargetLockCheckTime + LockCheckTime ) )
  {
    LastTargetLockCheckTime = WorldInfo.TimeSeconds;
    // Time, in seconds, since level began play
    CheckTargetLock();
    // Checks to see if we are locked on a target
  }
}
```

10. As I mentioned earlier, we can only lock onto pawns. Therefore, we need a function to check whether or not our target is a pawn.

```
/*******************************************************
* Given an potential target TA determine if we can lock on to it.
By
* default, we can only lock on to pawns.
*******************************************************/
simulated function bool CanLockOnTo(Actor TA)
{
  if ( (TA == None) || !TA.bProjTarget || TA.bDeleteMe ||
  (Pawn(TA) == None) || (TA == Instigator) ||
  (Pawn(TA).Health <= 0) )
  {
    return false;
  }
  return ( (WorldInfo.Game == None) ||
  !WorldInfo.Game.bTeamGame || (WorldInfo.GRI == None) ||
  !WorldInfo.GRI.OnSameTeam(Instigator,TA) );
}
```

11. Once we have a locked target we need to trigger a sound, so that the player is aware of the lock. The whole first half of this function simply sets two variables to not have a target, and also plays a sound cue to notify the player that we've lost track of our target.

```
/*******************************************************
* Used to adjust the LockTarget.
*******************************************************/
function AdjustLockTarget(actor NewLockTarget)
{
  if ( LockedTarget == NewLockTarget )
  {
    // No need to update
    return;
  }
```

```
    if (NewLockTarget == None)
    {
      // Clear the lock
      if (bLockedOnTarget)
      {
        // No target
        LockedTarget = None;
        // Not locked onto a target
        bLockedOnTarget = false;
        if (LockLostSound != None && Instigator != None &&
        Instigator.IsHumanControlled() )
        {
          // Play the LockLostSound if we lost track of the
             target
          PlayerController(Instigator.Controller).
          ClientPlaySound(LockLostSound);
        }
      }
    }
    else
    {
      // Set the lock
      bLockedOnTarget = true;
      LockedTarget = NewLockTarget;
      LockedTargetPRI = (Pawn(NewLockTarget) != None) ?
      Pawn(NewLockTarget).PlayerReplicationInfo : None;
      if ( LockAcquiredSound != None && Instigator != None &&
      Instigator.IsHumanControlled() )
      {
        PlayerController(Instigator.Controller).
        ClientPlaySound(LockAcquiredSound);
      }
    }
  }
```

12. Once it looks like everything has checked out we can fire our ammo! We're just setting everything back to `0` at this point, as our projectile is seeking our target, so it's time to start over and see whether we will use the same target or find another one.

```
/********************************************************
* Everything looks good, so fire our ammo!
********************************************************/
simulated function FireAmmunition()
{
  Super.FireAmmunition();
  AdjustLockTarget(None);
```

```
  LastValidTargetTime = 0;
  PendingLockedTarget = None;
  LastLockedOnTime = 0;
  PendingLockedTargetTime = 0;
}
```

13. With all of that out of the way, we can finally work on firing our projectile, or in our case, our missile. `ProjectileFile()` tells our missile to go after our currently locked target, by setting the `SeekTarget` variable to our currently locked target.

```
/*********************************************************
* If locked on, we need to set the Seeking projectile's
* LockedTarget.
 *********************************************************/
simulated function Projectile ProjectileFire()
{
  local Projectile SpawnedProjectile;

  SpawnedProjectile = super.ProjectileFire();
  if (bLockedOnTarget &&
  UTProj_SeekingRocket(SpawnedProjectile) != None)
  {
    /** Go after the target we are currently locked
        onto */
    UTProj_SeekingRocket(SpawnedProjectile).SeekTarget =
    LockedTarget;
  }
  return SpawnedProjectile;
}
```

14. Really though, our projectile could be anything at this point. We need to tell our weapon to actually use our missile (or rocket, they are used interchangeably) which we will define in our `defaultproperties` block.

```
/*********************************************************
* We override GetProjectileClass to swap in a Seeking Rocket if we
are
* locked on.
 *********************************************************/
function class<Projectile> GetProjectileClass()
{
  // if we're locked on...
  if (bLockedOnTarget)
  {
    // use our homing rocket
    return SeekingRocketClass;
  }
```

```
    // Otherwise...
    else
    {
      // Use our default projectile
      return WeaponProjectiles[CurrentFireMode];
    }
  }
```

If we don't have a `SeekingRocketClass` class defined, then we just use the currently defined projectile from our `CurrentFireMode` array.

15. The last part of this class involves the `defaultproperties` block. This is the same thing we saw in our `Camera` class. We're setting our muzzle flash socket, which is used for not only firing effects, but also weapon traces, to actually use our muzzle flash socket.

```
defaultproperties
{
    // Forces the secondary fire projectile to fire from
    the weapon attachment */
    MuzzleFlashSocket=MuzzleFlashSocket
}
```

Our `MyWeapon` class is complete. We don't want to clog our `defaultproperties` block and we have some great base functionality, so from here on out our weapon classes will generally be only changes to the `defaultproperties` block. Simplicity!

16. Create a new class called `MyWeapon_HomingRocket`. Have it extend from `MyWeapon`.

```
class MyWeapon_HomingRocket extends MyWeapon;
```

17. In our `defaultproperties` block, let's add our skeletal and static meshes. We're just going to keep using the shock rifle mesh. Although it's not necessary to do this, as we're already a child class of (that is, inheriting from) `UTWeap_ShockRifle`, I still want you to see where you would change the mesh if you ever wanted to.

```
defaultproperties
{
  // Weapon SkeletalMesh
  Begin Object class=AnimNodeSequence Name=MeshSequenceA
  End Object

  // Weapon SkeletalMesh
  Begin Object Name=FirstPersonMesh
    SkeletalMesh=
    SkeletalMesh'WP_ShockRifle.Mesh.SK_WP_ShockRifle_1P'
    AnimSets(0)=
    AnimSet'WP_ShockRifle.Anim.K_WP_ShockRifle_1P_Base'
```

```
    Animations=MeshSequenceA
    Rotation=(Yaw=-16384)
    FOV=60.0
  End Object

  // PickupMesh
  Begin Object Name=PickupMesh
    SkeletalMesh=
    SkeletalMesh'WP_ShockRifle.Mesh.SK_WP_ShockRifle_3P'
  End Object

  // Attachment class
  AttachmentClass=
  class'UTGameContent.UTAttachment_ShockRifle'
```

18. Next, we want to declare the type of projectile, the type of damage it does, and the frequency at which it can be fired. Moreover, we want to declare that each shot fired will only deplete one round from our inventory. We can declare how much ammo the weapon starts with too.

```
// Defines the type of fire for each mode
  WeaponFireTypes(0)=EWFT_InstantHit
  WeaponFireTypes(1)=EWFT_Projectile
  WeaponProjectiles(1)=class'UTProj_Rocket'

  // Damage types
  InstantHitDamage(0)=45
  FireInterval(0)=+1.0
  FireInterval(1)=+1.3
  InstantHitDamageTypes(0)=class'UTDmgType_ShockPrimary'
  InstantHitDamageTypes(1)=None
  // Not an instant hit weapon, so set to "None"

  // How much ammo will each shot use?
  ShotCost(0)=1
  ShotCost(1)=1

  // # of ammo gun should start with
  AmmoCount=20

  // Initial ammo count if weapon is locked
  LockerAmmoCount=20

// Max ammo count
  MaxAmmoCount=40
```

19. Our weapon will use a number of sounds that we didn't previously need, such as locking onto a pawn, as well as losing lock. So let's add those now.

```
// Sound effects
WeaponFireSnd[0] =
SoundCue'A_Weapon_ShockRifle.Cue.A_Weapon_SR_FireCue'
WeaponFireSnd[1]=SoundCue'A_Weapon_RocketLauncher.Cue.
A_Weapon_RL_Fire_Cue'
WeaponEquipSnd=
SoundCue'A_Weapon_ShockRifle.Cue.A_Weapon_SR_RaiseCue'
WeaponPutDownSnd=
SoundCue'A_Weapon_ShockRifle.Cue.A_Weapon_SR_LowerCue'
PickupSound=SoundCue'A_Pickups.Weapons.Cue.
A_Pickup_Weapons_Shock_Cue'
LockAcquiredSound=SoundCue'A_Weapon_RocketLauncher.Cue.
A_Weapon_RL_SeekLock_Cue'
LockLostSound=SoundCue'A_Weapon_RocketLauncher.Cue.
A_Weapon_RL_SeekLost_Cue'
```

20. We won't be the only one to use this weapon, as bots will be picking it up during Deathmatch style games as well. Therefore, we want to declare some logic for the bots, such as how strongly they will desire it, and whether or not they can use it for things like sniping.

```
// AI logic
MaxDesireability=0.65      // Max desireability for bots
AIRating=0.65
CurrentRating=0.65
bInstantHit=false         // Is it an instant hit weapon?
bSplashJump=false

// Can a bot use this for splash damage?
bRecommendSplashDamage=true

// Could a bot snipe with this?
bSniping=false

   // Should it fire when the mouse is released?
ShouldFireOnRelease(0)=0

// Should it fire when the mouse is released?
ShouldFireOnRelease(1)=0
```

21. We need to create an offset for the camera too, otherwise the weapon wouldn't display correctly as we switch between first and third person cameras.

```
// Holds an offset for spawning projectile effects
FireOffset=(X=20,Y=5)

// Offset from view center (first person)
PlayerViewOffset=(X=17,Y=10.0,Z=-8.0)
```

22. Our homing properties section is the bread and butter of our class. This is where you'll alter the default values for anything to do with locking onto pawns.

```
  // Homing properties
  /** angle for locking for lock
      targets when on Console */
  ConsoleLockAim=0.992

/** How far out should we be before considering actors for
    a lock? */
LockRange=9000

// Angle for locking, for lockTarget
LockAim=0.997

// How often we check for lock
LockChecktime=0.1

// How long does player need to hover over actor to lock?
LockAcquireTime=.3

// How close does the trace need to be to the actual target
LockTolerance=0.8

 SeekingRocketClass=class'UTProj_SeekingRocket'
```

23. Animations are an essential part of realism, so we want the camera to shake when firing a weapon, in addition to an animation for the weapon itself.

```
// camera anim to play when firing (for camera shakes)
  FireCameraAnim(1)=CameraAnim'Camera_FX.ShockRifle.
  C_WP_ShockRifle_Alt_Fire_Shake'

  // Animation to play when the weapon is fired
  WeaponFireAnim(1)=WeaponAltFire
```

24. While we're on the topic of visuals, we may as well add the flashes at the muzzle, as well as the crosshairs for the weapon.

```
  // Muzzle flashes
  MuzzleFlashPSCTemplate=WP_ShockRifle.Particles.
  P_ShockRifle_MF_Alt

  MuzzleFlashAltPSCTemplate=WP_ShockRifle.Particles.
  P_ShockRifle_MF_Alt

  MuzzleFlashColor=(R=200,G=120,B=255,A=255)
```

```
    MuzzleFlashDuration=0.33
    MuzzleFlashLightClass=
    class'UTGame.UTShockMuzzleFlashLight'
    CrossHairCoordinates=(U=256,V=0,UL=64,VL=64)
    LockerRotation=(Pitch=32768,Roll=16384)

    // Crosshair
    IconCoordinates=(U=728,V=382,UL=162,VL=45)
    IconX=400
    IconY=129
    IconWidth=22
    IconHeight=48

    /** The Color used when drawing the Weapon's Name on the
      HUD */
    WeaponColor=(R=160,G=0,B=255,A=255)
```

25. Since weapons are part of a pawn's inventory, we need to declare which slot this weapon will fall into (from one to nine).

```
// Inventory
InventoryGroup=4    // The weapon/inventory set, 0-9
GroupWeight=0.5     // position within inventory group.
                        (used by prevweapon and nextweapon)
```

26. Our final piece of code has to do with rumble feedback with the Xbox gamepad. This is not only used on consoles, but also it is generally reserved for it.

```
  /** Manages the waveform data for a forcefeedback device,
      specifically for the xbox gamepads. */
  Begin Object Class=ForceFeedbackWaveform
  Name=ForceFeedbackWaveformShooting1
    Samples(0)=(LeftAmplitude=90,RightAmplitude=40,
    LeftFunction=WF_Constant,
    RightFunction=WF_LinearDecreasing,Duration=0.1200)
  End Object

  // controller rumble to play when firing
  WeaponFireWaveForm=ForceFeedbackWaveformShooting1
}
```

27. All that's left to do is to add the weapon to your pawn's default inventory. You can easily do this by adding the following line to your `TutorialGame` class's `defaultproperties` block:

```
defaultproperties
{
  DefaultInventory(0)=class'MyWeapon_HomingRocket'
}
```

Load up your map with a few bots on it, hold your aiming reticule over it for a brief moment and when you hear the lock sound, fire away!

How it works...

To keep things simple we extend from `UTWeap_ShockRifle`. This gave us a great bit of base functionality to work from. We created a `MyWeapon` class which offers not only everything that the shock rifle does, but also the ability to lock onto targets.

When we aim our target reticule over an enemy bot, it checks for a number of things. First, it verifies that it is an enemy and also whether or not the target can be reached. It does this by drawing a trace and returns any actors which may fall in our weapon's path. If all of these things check out, then it begins to lock onto our target after we've held the reticule over the enemy for a set period of time. We then fire our projectile, which is either the weapon's firing mode, or in our case, a rocket.

We didn't want to clutter the `defaultproperties` block for `MyWeapon`; so we create a child class called `MyWeapon_HomingRocket` that makes use of all the functionality and only changes the `defaultproperties` block, which will influence the weapon's aesthetics, sound effects, and even some functionality with the target lock.

Creating a gun that heals pawns

UDK has built-in functionality for healing players through pickups such as health packs, but there is no way for one player to heal another.

In the following recipe, we'll create an instant hit weapon that heals a target for 10 points of health each time it is shot.

Getting ready

Start by creating a new class called `MyWeapon_HealingInstantHit` and have it extend from `MyWeapon`.

```
class MyWeapon_HealingInstantHit extends MyWeapon;
```

How to do it...

1. The great thing about setting up our `MyWeapon` class is that adding additional functionality to it is a breeze. This class has only one function. Let's add the `ProcessInstantHit()` function now.

First, we define the pawn that is being hit. Then, it takes the `ProcessInstantHit()` function and rather than have it apply damage to a pawn, it applies additional health by calling our pawn's `HealDamage()` function. The first parameter used by `HealDamage()` is an integer, which declares exactly how much health each shot will heal a pawn for. We've set it to the modest value of `10`.

We use a log for debugging again and have it output our pawn's health each time it is shot, through the call to `P.Health`.

```
/********************************************************
* Heals a pawn with an instant hit weapon
* Doesn't allow pawn's health to exceed maximum (100)
********************************************************/
simulated function ProcessInstantHit(byte FiringMode, ImpactInfo
Impact, optional int NumHits)
{
  local Pawn P;

  if (Impact.HitActor != None &&
  (Impact.HitActor).IsA('Pawn'))
  {
    // Defining the pawn
    P = Pawn(Impact.HitActor);

    // Increase health by 10
    P.HealDamage(10, Instigator.Controller,
                 InstantHitDamageTypes[1]);

    // Log for debugging
    'Log("***Pawn Health:" @P.Health);
  }
}
```

2. We still need to make one alteration to our `DefaultProperties` block.

```
DefaultProperties
{
  // Do not perform any damage
  InstantHitDamageTypes(1)=None
}
```

We're telling the game that despite us using an instant hit weapon, we do not want it to perform any damage. Therefore, we set the damage type to `None`.

How it works...

By overriding `UTWeapon's ProcessInstantHit()`, we remove its default functionality of performing damage on a pawn and instead do just the opposite; heal it! Alternatively, you could set a function that causes damage to a pawn to a negative number, and that could also heal the pawn.

We simply heal the pawn, let it know which pawn is performing the heal, and finally assign a firing mode to the function. In our case, this is the secondary function for our new weapon. Don't forget, arrays in UDK start at `0`, so `0` is really the first firing mode, and `1` is the second.

> Also don't forget to change your pawn's default inventory, so that it uses your new weapon!

There's more...

See if you can make the weapon rapid fire by increasing its firing speed. This can be done in the default properties again. You may want to decrease how much health each shot is worth, however, otherwise you can increase a pawn's health to maximum capacity instantly.

Creating a weapon that can damage over time

Damage over Time (**DoT**) weapons have been a staple in gaming for decades. They can be anything from a pawn taking acid damage, falling into a pool of lava, drowning, or even being poisoned.

Our next recipe will have us creating a weapon that allows our pawn to take a set amount of damage over a brief period of time. This will require both a weapon, as well as a number of changes to our pawn.

Getting ready

Start by creating a new class called `MyWeapon_PoisonDamage` and have it extend from `MyWeapon`.

```
class MyWeapon_PoisonDamage extends MyWeapon;
```

How to do it...

We're only going to add one function to this class.

1. Just as we did with our healing weapon, we need to override the `UTWeapon` class's `ProcessInstantHit()` function. We're not going to do any sort of healing again though.

   ```
   simulated function ProcessInstantHit(byte FiringMode, ImpactInfo
   Impact, optional int NumHits)
   {
     local TutorialPawn TP;

     if (Impact.HitActor != None &&
     (Impact.HitActor).IsA('Pawn'))
     {
       // Defining the pawn
       TP = TutorialPawn(Impact.HitActor);

   /** Calls the poison function in the TutorialPawn
       class */
       TP.PoisonPlayer();

   // Log for debugging
       'Log("***Pawn Health:" @TP.Health);

     }
   }
   ```

   First we perform a check to see whether our projectile has hit an actor. Immediately after that, we check to see if that actor is a pawn. We don't want to apply our poison damage to something non-living like a vehicle or a barrel. Afterwards, we check to see if the pawn is of our `TutorialPawn` class. From there, we call the `PoisonPlayer()` function from our `TutorialPawn` class. This is what actually poisons the pawn. Finally, we add our standard log which tracks our tutorial pawn's health, to verify that our function is actually having some kind of an effect.

2. Now it's time to head to our `TutorialPawn` class. Let's start by adding a variable to hold the amount of time ticking by, as our pawn is poisoned:

   ```
   var int PoisonCountdown;
   ```

3. Our `PoisonPlayer()` function is what is actually being called by our weapon. This simple function resets our poison counter to `0`, and prevents our damage from stacking.

Next we use a standard `SetTimer()` function, which you will find is frequently used throughout UDK. We're telling the game to call the `PoisonDmg()` function every `.5` seconds. This is the actual tick that damages our pawn. We could easily increase the amount of damage done by increasing the frequency at which `PoisonDmg()` is called.

```
/*********************************************************
* Called by MyWeapon_PoisonDamge when the pawn is shot
*********************************************************/
function PoisonPlayer()
{
  // Reset Poison Counter
  PoisonCountdown = 0;

  // Every .5 seconds PoisonDmg() will be called
  SetTimer(0.5, true, 'PoisonDmg');
}
```

Let's take a look at what `PoisonDmg()` actually does:

```
/*********************************************************
* Actually does the damage to the pawn
*********************************************************/
function PoisonDmg()
{
  // Does 5 damage to the pawn of type UTDmgType_Burning
  TakeDamage( 5, None, Location, vect(0,0,0) ,
  class'UTDmgType_Burning');
  PoisonCountdown=PoisonCountdown+1;
  // Increment timer
  'Log("***Pawn Health:" @Health);
  // Log for debugging

  // clear the infinitely looping 0.5 second timer after 10
    counts of damage
  if(PoisonCountdown >= 10)
  {
    ClearTimer('PoisonDmg');
  }
}
```

We start by calling the `TakeDamage()` function and informing the pawn how much damage it is taking (`5`), where it is being hit (`Location`), any momentum to be applied (`None`, as our vector reads `0,0,0`) and finally the type of damage (`UTDmgType_Burning`). Burning is a DoT as well, so rather than create a new damage type, we just stick with what UDK provides.

Our PoisonCoundown variable, which we created in the preceding code, is now being used as well. Each time the function is called (in our case, every 0.5 seconds), we add one to the count. Next, we create an if statement that clears our timer once we've accumulated 10 seconds worth of damage.

> **Want to have the DoT effect last longer?**
> Simply set the integer in our if(PoisonCountdown >= 10) statement to be greater than 10!

4. Change your pawn's default inventory, so that it uses our new weapon and give it a spin! Once hit, you'll notice that the pawn flashes red very briefly, each time damage is received. Take a look at your log to see exactly how much health is drained with each hit.

How it works...

Because our MyWeapon class is so modular, we're able to create brand new functionality by only having to override one function within the class. When our instant hit projectile (ProcessInstantHit()) hits our TutorialPawn, it calls the PoisonPlayer() function within that class.

PoisonPlayer() then calls our PoisonDmg() function every half second. PoisonDmg() then sets a number of attributes, such as the amount of damage taken during each hit, where the pawn is hit, and how long the effect will last.

There's more...

Now that we have a projectile, which poisons our enemies, what's stopping us from creating a grenade that does the same, or even a pickup? See if you can create an object that causes poison damage when the pawn picks it up with the use function, or if a pawn runs it over (collides).

Looking further down the road, one excellent idea would be to create a grenade or explosive that heals people over time.

> Take a look at the Bump() and Touch() functions in your pawn.

Adding a flashlight to a weapon

Flashlights have been man's best friend to combat darkness since the invention of electricity. A flashlight can even be turned into a weapon, as we saw with 2010's release of Remedy's *Alan Wake*.

In the following recipe, we'll be creating a flashlight that can attach to the pawn, as well as a weapon, and be toggled on and off with any key of your choice.

Getting ready

Start by creating a new class called `WeaponFlashlight` and have it extend from `SpotLightMovable`. Also make it non placeable. Non placeable means that it cannot be dropped onto the map. Essentially it's there to keep things clean for the level designers, and avoid confusion as to how something should or should not be used.

For this recipe, we'll have to make small modifications to our `TutoialPawn` class, in addition to creating one new class, which is our actual flashlight.

```
class WeaponFlashlight extends SpotLightMovable
      notplaceable;
```

How to do it...

1. `SpotlightMoveable` already has all of the functionality that we'll need, so we're only going to adjust a few of the default properties.

   ```
   DefaultProperties
   {
     Begin Object Name=SpotLightComponent0
     // Sets the light color
       LightColor=(R=200,G=200,B=200)
       InnerConeAngle=10.0
       OuterConeAngle=20.0
     End Object
     // Cannot be deleted during play.
     bNoDelete=false
   }
   ```

   We decrease the RGB value of the light down from a full 255 to 200 so that we don't have a blinding white light. I prefer a softer white with a subtle shadow. The inner and outer cone angles will also greatly affect how the light is displayed on screen. Play with the values for a bit to really get a feel for what works best for your needs.

   The only things left to do from here are additions and alterations to our `TutorialPawn` class.

2. Start off by declaring a variable to reference our `WeaponFlashlight`.

```
var WeaponFlashlight Flashlight;
```

3. We also need to add some functionality in our `PostBeginPlay()` function:

```
simulated event PostBeginPlay()
{
  Super.PostBeginPlay();
  'Log("================");
  'Log("Tutorial Pawn up");

  //***** Used for the flashlight *****//
  // Spawns the light on the player, setting self as owner
  Flashlight = Spawn(class'WeaponFlashlight', self);

  // Sets the lights base on at the player
  Flashlight.SetBase(self);

// Light is off by default
  Flashlight.LightComponent.SetEnabled(false);

  // Starts at 75% brightness
  Flashlight.LightComponent.SetLightProperties(0.75);
}
```

We're doing a few things here. First, we're spawning our `WeaponFlashlight` and setting our pawn as the owner. Afterwards, we're attaching the light to our pawn, keeping it toggled to off by default, and dropping the brightness down to 75 percent.

4. The flashlight is attached to our pawn, but we need it to rotate when our pawn rotates, so let's do that now:

```
/*********************************************************
* Forces the flashlight to use our pawn's rotation
*********************************************************/
event UpdateEyeHeight( float DeltaTime )
{
    Super.UpdateEyeHeight(DeltaTime);

    // Flaslight will use our controller's rotation
    Flashlight.SetRotation(Controller.Rotation);

    /* Offset the light slightly, so that it looks as
       though it is coming from our pawn's eyes/helmet */
    Flashlight.SetRelativeLocation
    (Controller.RelativeLocation + vect(20, 0, 25));
}
```

`UpdateEyeHeight()` takes `Deltatime` as a parameter and updates each frame. With `Flashlight.SetRotation()` we set the rotation to use `Controller.Rotation`, and then offset it slightly so that it appears as though the light is coming from our pawn's helmet.

5. We've got our flashlight set to our pawn, but we need a way to turn it on and off now. Add the following function to your class:

```
/********************************************************
* Turns the light on and off
********************************************************/
exec function ToggleFlashlight()
{
    // If the light is off...
    if(!Flashlight.LightComponent.bEnabled)
    {
      // Then turn it on
      Flashlight.LightComponent.SetEnabled(true);
      'log("TOGGLE FLASHLIGHT ON");
    }
    else        // If it's already on
    {
      Flashlight.LightComponent.SetEnabled(false);
      // Turn it off
      'log("TOGGLE FLASHLIGHT OFF");
    }
}
```

`exec function` tells us that this function can be called by the player through a key press. We're stating that if the light is off, then when we select our `ToggleFlashlight` key that we're about to define, and then turn it on. Otherwise, if it's already on, then turn it off.

We need a way to toggle the flashlight on now though! This is a quick fix. Browse to your `.ini` files. We're going to be looking for `DefaultInput.ini`.

> This can be generally found under the file path, `UDKGame/Config`.

6. Open up that file and scroll down until you see the key configurations. Right above the text marked `BINDINGS THAT ARE REMOVED FROM BASEINPUT.INI` add the following code:

```
;--------------------------------------------------------
; CUSTOM BINDINGS FOR TUTORIALS
```

```
;-----------------------------------------------------
.Bindings=(Name="GBA_ToggleFlashlight"
,Command="ToggleFlashlight")
.Bindings=(Name="X" ,Command="GBA_ToggleFlashlight")
```

The first line is setting our `ToggleFlashlight` function to the name `GBA_ToggleFlashlight` and the next line is binding our `GBA_ToggleFlashlight` command to the *X* key.

7. Rebuild scripts and hop into the game. `Ch5_PathNodes2.udk` should work fine. Run towards the block in the center of the map, hit the *X* key on your keyboard, and watch as you have a flashlight that can now toggle on and off!

> It may be wise to delete your `UDKInput.ini` file before rebuilding, otherwise the engine may not use your changes. This way, during the next rebuild, it will grab your update file and use that as the default settings.

How it works...

UDK provides a plethora of classes that we can subclass from. As `SpotLightMovable` suits our needs well, we've just extended from that and only adjusted a few minor default properties for our flashlight.

Afterwards, we attached the light to our pawn, then slightly offset the starting point of the cone to begin from our pawn's eyes, before finally telling it to follow our controller's rotation, so that it always faces the same direction our pawn is looking at.

Finally, we set a new input command in our `DefaultInput.ini` file, and bound our *X* key to execute the function `ToggleFlashlight`.

Creating an explosive barrel

Guns aren't the only weapons we can make use of in UDK. It's time to think outside the box for a bit and come up with some other creative weapons. For this next recipe, we'll create a weapon that makes use of our environment, an explosive barrel.

Since the early days of gaming, the explosive barrel has been a primary tool in every level designer's tool belt. Granted, I've never seen an explosive barrel in my life, but games would lead you to believe otherwise.

With that in mind, let's create our barrel.

Getting ready

Open your IDE and start by creating a new class called `ExplosiveBarrel` and have it extend from `DynamicSMActor`.

Because our barrel is an item that will probably be heavily used by a level designer, let's hide many of the properties from the editor, as it will just clutter up the screen. Hide `Movement`, `Attachment`, `Debug`, `Advanced`, `Mobile`, and `Physics`.

We also want to make our barrel placeable within a level.

```
class ExplosiveBarrel extends DynamicSMActor
  HideCategories(Movement, Attachment, Debug, Advanced, Mobile,
                Physics)
      placeable;
```

How to do it...

For this recipe, we'll be creating a new class, unlike any we've encountered yet. We'll be using timers to call specific functions at set intervals, detonating and respawning explosives, and triggering particle effects. And best of all, this is done by creating only one class!

1. It's time to add our variables. We have quite a few here, but they are commented pretty clearly, so don't worry.

```
/** Explodes when damaged. */
var()  bool    bDestroyOnDmg;

/** Explodes when a player walks over it */
var()  bool    bDestroyOnPlayerTouch;

/** Explodes when a vehicle drives over it */
var()  bool    bDestroyOnVehicleTouch;

/** Mesh to swap in when destroyed. */
var()  StaticMesh    MeshOnDestroy;

/** How long the spawned physics object should last. */
var()  float    SpawnPhysMeshLifeSpan;

/** Initial linear velocity for spawned phys obj. */
var()  vector    SpawnPhysMeshLinearVel;

/** initial angular velocity for spawned physics object.
var()  vector    SpawnPhysMeshAngVel;

/** Sound to play when destroyed. */
```

```
var()  SoundCue    SoundOnDestroy;

/** Particles to play when destroyed. */
var()  ParticleSystem    ParticlesOnDestroy;

/** Allows particles to be turned on/off. */
var()  ParticleSystemComponent    PSC;


/** Static mesh to spawn as physics object when destroyed. */
var()  StaticMesh    SpawnPhysMesh;

/** Time between being destroyed & respawning. */
var()  float    RespawnTime;

/** Set the mesh back to the original upon respawning */
var StaticMesh    RespawnSM;

/** Is the barrel currently destroyed? */
var bool    bDestroyed;

/** Time before we are going to respawn. */
var float    TimeToRespawn;
```

2. Our first function is simply `PostBeginPlay()`. Here we set `RespawnSM` (respawn static mesh) to use our static mesh component.

```
/*******************************************************
* Setting respawn mesh to use our static mesh
*******************************************************/
simulated function PostBeginPlay()
{
  Super.PostBeginPlay();

// Uses this mesh when respawning
  RespawnSM = StaticMeshComponent.StaticMesh;
}
```

When our barrel explodes, we'll need a way to set it back to its original condition when it respawns. We do this with the `RespawnDestructable()` function. Here, we reset the static mesh and then reattach the static mesh component. Additionally, we turn off the particle system, so that we no longer see the smoke and fire from the previously destroyed barrel.

```
/*******************************************************
* Place destroyed item back in original condition
*******************************************************/
```

```
simulated function RespawnDestructible()
{
  // Turns off fire/smoke particles
  PSC.DeactivateSystem();

  // Reset static mesh & re-attach SM component.
  StaticMeshComponent.SetStaticMesh(RespawnSM);
  if(!StaticMeshComponent.bAttached)
  {
    AttachComponent(StaticMeshComponent);
  }
  bDestroyed = FALSE;
}
```

3. The main part of this class is the barrel exploding, which we simply name `Explode()`. Within `Explode()` you'll find the `HurtRadius()` function, to which we pass parameters for the base damage, radius, damage type, momentum applied to the explosion, location, and whether or not it can apply full damage to the pawn. Most, if not all, area-effect weapons in UDK use this function.

```
/*********************************************************
* Called when damage is taken or it is touched
*********************************************************/
simulated function Explode()
{
  local UTSD_SpawnedKActor PhysMesh;

  HurtRadius(30.0, 200.0, class'UTDamageType', 300.0,
             Location,,, True);

  // Swap or hide mesh when destroyed
  if(MeshOnDestroy != None)
  {
    StaticMeshComponent.SetStaticMesh(MeshOnDestroy);
  }
  else
  {
    StaticMeshComponent.SetStaticMesh(None);
    DetachComponent(StaticMeshComponent);
  }

  // Play sfx after object is destroyed
  if(SoundOnDestroy != None)
  {
    PlaySound(SoundOnDestroy, TRUE);
```

```
  }

  //Generate fire particle after object destruction
  if(ParticlesOnDestroy != None)
  {
    PSC = WorldInfo.MyEmitterPool.SpawnEmitter
          (ParticlesOnDestroy, Location, Rotation);
  }

  // Spawn physics mesh
  if(SpawnPhysMesh != None)
  {
    PhysMesh = spawn(class'UTSD_SpawnedKActor',,,Location,
                     Rotation);
    PhysMesh.StaticMeshComponent.SetStaticMesh
    (SpawnPhysMesh);
    PhysMesh.StaticMeshComponent.SetRBLinearVelocity
    (SpawnPhysMeshLinearVel, FALSE);
    PhysMesh.StaticMeshComponent.SetRBAngularVelocity
    (SpawnPhysMeshAngVel, FALSE);
    PhysMesh.StaticMeshComponent.WakeRigidBody();

    // Collides with the world but, not players or vehicles
    PhysMesh.SetCollision(FALSE, FALSE);
    PhysMesh.StaticMeshComponent.SetRBChannel
    (RBCC_Default);
    PhysMesh.StaticMeshComponent.SetRBCollidesWithChannel
    (RBCC_Default, TRUE);

    // Set lifespan
    PhysMesh.LifeSpan = SpawnPhysMeshLifeSpan;
  }
  bDestroyed = TRUE;
  TimeToRespawn = RespawnTime;

    // It will respawn after (X) seconds
    SetTimer(RespawnTime, FALSE, 'RespawnDestructible');
}
```

Next, if our static mesh is destroyed, we need to either hide it, or replace it with a destroyed version of our mesh. This is particularly useful when dealing with larger objects, such as vehicles. When a vehicle explodes surely there must be something left behind, right? Because we are using a static mesh natively supplied from UDK, we don't have a replacement mesh. We simply tell the mesh to disappear.

This would be an excellent time to add more functionality to this, such as having the mesh fracture into pieces. UDK offers a great tool for doing exactly this, although it's beyond the scope of this recipe, so we'll pass over it for the moment.

We then trigger our sound effect and particle effect for the explosion. The current particle effect is great, as it allows for a slow roasting flame along with smoke to continue where the barrel was for quite some time after detonation. The spawning of our physics mesh follows shortly after, and this kinetic actor is what allows the barrel to be moved and manipulated within the world.

Finally, we call `RespawnTime`, which is actually defined in the default properties. The barrel will respawn after a set number of seconds defined there.

4. What good is an explosive barrel if it doesn't explode when hit? We need to create a way for our barrel to explode when it takes damage, so the next function does exactly that.

```
/********************************************************
* Called when the object is shot or damaged
********************************************************/
simulated function TakeDamage
(int DamageAmount, Controller EventInstigator, vector
HitLocation, vector Momentum, class<DamageType> DamageType,
optional TraceHitInfo HitInfo, optional Actor DamageCauser)
{
  if(!bDestroyed && bDestroyOnDmg)
  {
    Explode();
  }
}
```

We check to see if our barrel is not destroyed and if it is to be destroyed when taking damage. If both are true, call `Explode()`!

5. That's not the only way to detonate our barrel though. We also have the option to have it explode if it is touched, either by a vehicle or a pawn. Let's add that function:

```
/********************************************************
*  Called when a pawn/vehicle touches it
********************************************************/
simulated function Touch(Actor Other, PrimitiveComponent
OtherComp, vector HitLocation, vector HitNormal)
{
  // Ignore if destroyed.
  if(bDestroyed)
  {
    return;
  }
```

```
    if( Vehicle(Other) != None )
    {
      // If a vehicle touches it...
      if(bDestroyOnVehicleTouch)
      {
        // Explode
        Explode();
      }
    }
    else
    {
      // If a player touches it...
      if(bDestroyOnPlayerTouch && Pawn(other) !=None)
      {
        // Explode
        Explode();
      }
    }
  }
```

For now, it will only detonate when run over (touched) by a vehicle. We can easily turn this on or off, as well as for a pawn, in the `defaultproperties` block.

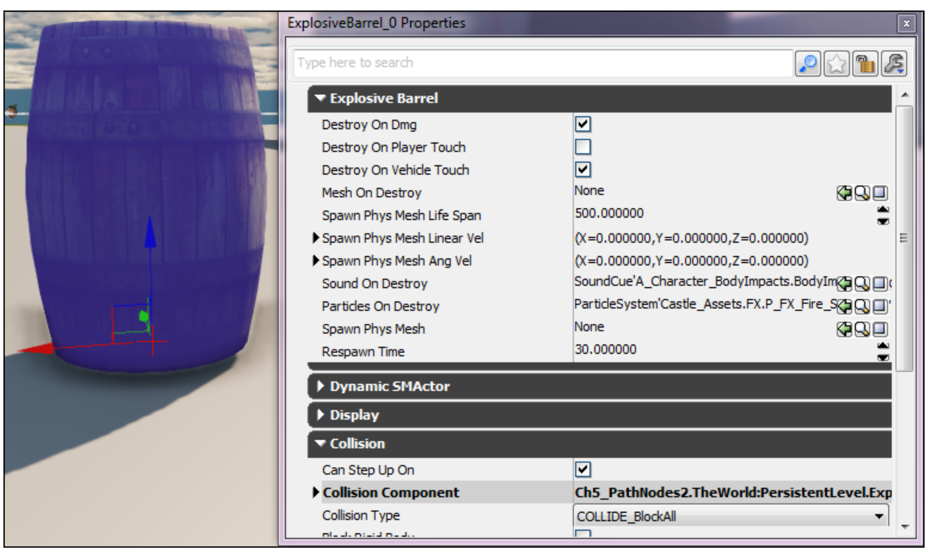
6. The last part of this class is the `defaultproperties` block. Let's define our values now:

```
defaultproperties
{
  bCollideActors=TRUE
  bProjTarget=TRUE
  bPathColliding=FALSE
  bNoDelete=TRUE
  Begin Object Name=MyLightEnvironment
      bEnabled=TRUE
      bDynamic=FALSE
  End Object

  // Mesh for the object
  Begin Object Name=StaticMeshComponent0
    StaticMesh=StaticMesh'E3_Demo.Meshes.SM_Barrel_01'
  End Object

  ParticlesOnDestroy[0]=
  ParticleSystem'Castle_Assets.FX.P_FX_Fire_SubUV_01'
```

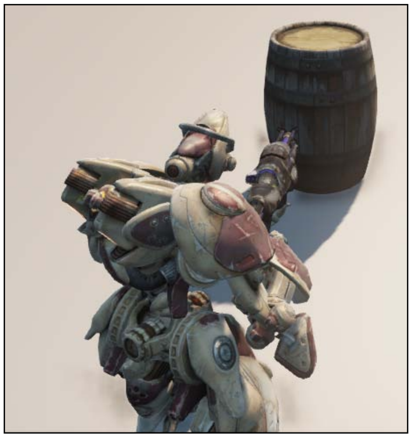
```
    SoundOnDestroy=SoundCue'A_Character_BodyImpacts.
    BodyImpacts.A_Character_RobotImpact_BodyExplosion_Cue'

    MeshOnDestroy=
    StaticMesh'Envy_Effects.VH_Deaths.S_Envy_Rocks'

    RespawnTime=30.0

    // How long the spawned physics object should last
    SpawnPhysMeshLifeSpan=500.0

    // Destroyed when damaged
    bDestroyOnDmg=TRUE

    // Destroyed when touched by player
    bDestroyOnPlayerTouch=FALSE

    // Destroyed when touched by vehicle
    bDestroyOnVehicleTouch=TRUE

    // Blocks other nonplayer actors
    bBlockActors=TRUE;
  }
```

It is all very straightforward, and most of the properties are the ones commonly used within UDK. With our explosive barrel built, it's now time to hop into our map and place the barrel.

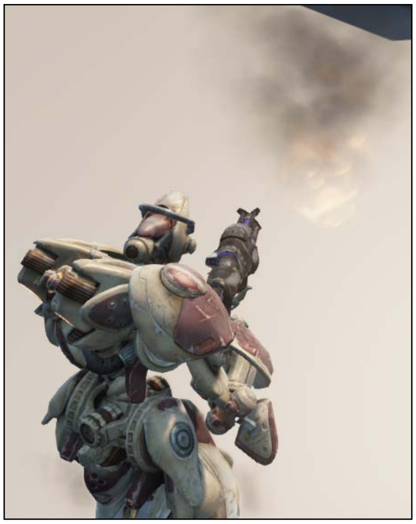
7. Start the UDK editor and head to your **Actors Browser**. Enter `Explosive Barrel` in the search bar and your barrel will be present. Drag-and-drop it into the map.



Like I mentioned earlier, we have hidden a number of properties from the barrel to make a slick interface for our level designers. There is no need for them to ever have to access code to make changes to the properties, as properties such as the static mesh, sound cue, and particle effects can be swapped in here as well.



Our barrel won't go off if we get too close to it, although it will detonate if we fire a few rounds at it!



A slick particle effect comes in place of our barrel when it is detonated. See if you can add a bit more realism to the effect by triggering another explosive to detonate just before our fire particle turns on. Also try adding some appropriate debris at the location where the particle is emitted.

How it works...

We extend our barrel from `DynamicSMActor`, because we want players to be able to interact with it. Dynamic, or kinematic actors, allow for movement and manipulation at runtime. From there we simply create a placeable dynamic static mesh, which can be destroyed when touched.

The `Touched()` function is key here, because it allows detonation to occur either when a pawn or vehicle touches it, or when it takes damage. You may have also noticed that this function is used pretty frequently within UDK, especially for taking damage.

Finally, we make it easy for our level designers to alter the properties of the barrel by exposing ones such as particles, sound cue, and static mesh within the editor.

Creating a landmine

The explosive barrel is a nice touch in any environment, but let's build something for a more specific application. What if we were to set a trap and have it spring when a character comes within a close enough proximity? Even better, a landmine in an open environment like a battlefield.

With that in mind, let's get to building a landmine that detonates when touched.

Getting ready

Open your IDE and start by creating a new class called `Landmine` and have it extend from `ExplosiveBarrel`.

Because our mine is an item that will probably be heavily used by a level designer, let's again hide many of the properties from the editor, as it will just clutter up the screen. Hide `Movement`, `Attachment`, `Debug`, `Advanced`, `Mobile`, and `Physics`.

We also want to make our `Landmine` placeable within a level.

```
class Landmine extends DynamicSMActor
  HideCategories(Movement, Attachment, Debug, Advanced, Mobile,
                 Physics)
      placeable;
```

How to do it...

Our class was made with modularity in mind, so we really only need to change a few default properties to really get a new, albeit similar, object.

```
defaultproperties
{
  bCollideActors=TRUE
  bProjTarget=TRUE
  bPathColliding=FALSE
  bNoDelete=TRUE

  Begin Object Name=MyLightEnvironment
    bEnabled=TRUE
    bDynamic=FALSE
  End Object

// Mesh for the object
  Begin Object Name=StaticMeshComponent0
    StaticMesh=StaticMesh'Pickups.WeaponBase.
    S_Pickups_WeaponBase'
    Scale=0.5
  End Object

  ParticlesOnDestroy[0]=
  ParticleSystem'Castle_Assets.FX.P_FX_Fire_SubUV_01'
  SoundOnDestroy=SoundCue'A_Vehicle_Cicada.SoundCues.
  A_Vehicle_Cicada_Explode'
  MeshOnDestroy=
  StaticMesh'Envy_Effects.VH_Deaths.S_Envy_Rocks'

  RespawnTime=30.0

// How long the spawned physics object should last
  SpawnPhysMeshLifeSpan=500.0

// Destroyed when damaged
  bDestroyOnDmg=TRUE

// Destroyed when touched by player
  bDestroyOnPlayerTouch=TRUE

// Destroyed when touched by vehicle
  bDestroyOnVehicleTouch=TRUE

// Blocks other nonplayer actors
  bBlockActors=FALSE;
}
```

We've changed a few values here from the explosive barrel. Most notably, we use a different static mesh, and instead of a collision cylinder we use a rough approximation of the mesh's size. To see the difference during runtime, enter `Show Collision` into the command line when running the editor. We've also changed the sound effect.

We didn't have our explosive barrel detonate when touched by the pawn, but we do want that to occur with the landmine, so we've set that value to `TRUE` here. We also do not want the mine to block actors like the barrel did, so we set that to `FALSE`.

Other than that, our landmine is identical to the explosive barrel. Play with some of the properties for a bit and see what you're able to create as well!

How it works...

Just as we did with our explosive barrel, we extend our landmine from `ExplosiveBarrel`, because we want players to be able to interact with it. We've also made it placeable again, so that our level editors can manipulate it within the editor.

The `Touched()` function is the key here too, because it allows detonation to occur either when a pawn or vehicle touches it, as well as when it takes damage. You may have also noticed that this function is used pretty frequently within UDK, especially for taking damage.

Finally, we make it easy for our level designers to alter the properties of the `Landmine` class by exposing properties such as particles, sound cues, and static meshes within the editor.

7
HUD

In this chapter, we will cover:

- Displaying a bar for the player's health
- Drawing text for the player's health
- Displaying a bar for the player's ammo
- Drawing text for the player's ammo
- Drawing the player's name on screen
- Creating a crosshair

Introduction

A heads-up display, or HUD, in addition to providing a **user interface** (**UI**), offers a means of providing information to a player to allow them to interact with the game world.

UDK offers two methods for creating a HUD. The first and most simple method which we'll be covering here is Canvas. The other method, which requires knowledge of the flash language ActionScript and some fancy art skills, allows UDK to make use of a third-party tool called Scaleform to draw the HUD.

Scaleform is probably what you see in most TripleA games, as it offers a far more impressive and professional aesthetic, albeit at a greater time commitment, in addition to an expensive suite of tools. There are a number of free flash editors available, however, including Ajax Animator, FlashDevelop, and Open Dialect.

We'll be sticking with Canvas for our purposes, as it suits our needs well and only requires knowledge of UnrealScript. Canvas offers a plethora of functions for depicting elements to the screen, including materials, shapes, images, and text. Combining these elements we can make an attractive, cohesive, and useful interface.

Canvas is not without its limitations though. On the PC or a console material drawing, it works fine; but it is not currently supported on mobile. Therefore, if you're considering UDK for mobile games, you may want to take a deeper look into Scaleform.

Displaying a bar for the player's health

One of the most important bits of information to display on screen is that of the player's health. Developers have also begun to think outside the box and come up with creative ways to illustrate a player's health other than just through text or a standard HUD. Capcom's *Resident Evil* franchise often has characters appear visibly different, such as characters walking with a limp and moving at a slower pace when injured. *Dead Space* uses the player's space suit to illustrate the current health by drawing a health bar on the back, although this is done with Scaleform.

Regardless, the generic HUD doesn't seem to be leaving at any time soon, so let's start by illustrating our most important information, a health bar.

Getting ready

Start the same as we always do, that is, with our IDE open and a new class created. This time we're going to create a new class called `TutorialHUD` and have it extend from `MobileHUD`.

```
class TutorialHUD extends MobileHUD;
```

How to do it...

Our goal is to use what UDK has provided for us and create a simple HUD of our own. In this recipe, we'll be adding a health bar which changes color depending on the amount of health our pawn currently has. When full, the bar will be tinted green; but when the pawn's health is critically low, it will become red.

1. With our class created, we can focus on adding the variables now.

```
/** Holds the scaled width and height of the viewport, which
adjusts with the res */
var float    ResScaleX, ResScaleY;
```

```
/** Texture for HP bar*/
var const Texture2D    BarTexture;

/** Current owner of the HUD */
var Pawn    PawnOwner;

/** Sets owner of the current TutorialPawn */
var TutorialPawn    TutPawnOwner;

/** Positoning for HP bar and text */
var vector2D    HPPosition, HPTextOffset;
var TextureCoordinates         BarCoords;
```

2. Next we need the `PostRender()` function, which is responsible for caching the value of our variables, and is part of the main draw loop.

```
/***********************************************************
* Caches values for variables. Also the main draw loop
**********************************************************/
event PostRender()
{
  Super.PostRender();

  /** Sets the pawn owner */
  PawnOwner = Pawn(PlayerOwner.ViewTarget);
  if ( PawnOwner == None )
  {
    PawnOwner = PlayerOwner.Pawn;
  }
  TutPawnOwner = TutorialPawn(PawnOwner);

  if (TutPawnOwner != None)
  {
    /** Sets the size of the screen based on resolution*/
    ResScaleX = Canvas.ClipX/1024;
    ResScaleY = Canvas.ClipY/768;
  }
}
```

We won't use `PostRender` as heavily as Unreal Tournament does, but it is important to understand that it's essential for using Scaleform as it allows for additional things to be drawn on screen, including animated crosshairs and overlays that mobile devices, such as iOS, require.

3. Adding the `DrawHUD` function is the next step. This is part of the game's main loop and is called by each frame. We'll be putting any function here for drawing the HUD, and you'll understand how heavily it is utilized in the coming recipes.

```
/**********************************************************
* Draws the HUD
**********************************************************/
function DrawHUD()
{
    super.DrawHUD();
    DrawHealthText();
}
```

4. The next function, despite being called `DrawHealthText()`, actually sets some variables that we'll need for our health bar, in addition to calling `DrawHealthbar()`.

```
/**********************************************************
* Draws the health text bar
**********************************************************/
function DrawHealthText()
{
  local Vector2D    TextSize, POS, HPTextOffsetPOS;
  local int    HPAmount, HPAmountMax;

  /** Sets the bar position */
  POS = CorrectedHudPOS
  (HPPosition,BarCoords.UL,BarCoords.VL);

  /** Offsets the text from the bar */
  HPTextOffsetPOS = HudOffset(POS, HPTextOffset,);

  /** Sets the pawn's health amount */
  HPAmount = PlayerOwner.Pawn.Health;
  HPAmountMax = PlayerOwner.Pawn.HealthMax;

  /** Draws health bar */
  DrawHealthBar(POS.X, POS.Y, HPAmount, HpAmountMax, 80,
  Canvas);
}
```

You'll also see that we're setting the bar position by using a variable called `POS` and setting it to the `CorrectedHudPos` function. This will be explained shortly. We will be creating a `DrawHealthBar()` function which takes an integer as its third and fourth parameters. We've set our `HPAmount` and `HPAmountMax` variables to grab our pawn's hit points, and we'll use that as our parameter.

5. Now we need to draw the actual bar graph to represent our health. We'll be using the same function to draw a bar for both our health and ammo.

```
/*********************************************************
* Draw bar graph for health / ammo and background HP bar.
* Called by DrawHealth()and DrawAmmo().
*********************************************************/
simulated function DrawBarGraph(float X, float Y, float
Width, float MaxWidth, float Height, Canvas DrawCanvas,
Color BarColor, Color BackColor)
{
  /** Draw the dark bar behind our current one */
  if ( MaxWidth > 24.0 )
  {
    DrawCanvas.DrawColor = BackColor;
    DrawCanvas.SetPos(X,Y);
    DrawCanvas.DrawTile(BarTexture,MaxWidth*2,Height,
    407,479,FMin(MaxWidth,118),16);
  }

  /** Draw the bar */
  DrawCanvas.DrawColor = BarColor;
  DrawCanvas.SetPos(X, Y);
  DrawCanvas.DrawTile(BarTexture,Width*2,Height,
  BarCoords.U,BarCoords.V,BarCoords.UL,BarCoords.VL);
}
```

We're really going to be drawing two bars here. The first one is going to be a light gray color and will represent our full health. This is always drawn and the value will not change.

Beneath that, we'll be drawing another bar which represents our health in its current state. If we lose health, then this bar shrinks in size as well.

6. We need a function to draw our distinct health bar now. When the time comes to draw our ammo bar, you'll see that we've created a similar function for that as well. This bar will change color too, depending on our current health values. It's always nice to have a bit of a warning when our health is getting low.

```
/*********************************************************
* Draw player's health. Adjusts the bar color based on health
*********************************************************/
simulated function DrawHealthBar(float X, float Y, float
Width, float MaxWidth, float Height,Canvas DrawCanvas,
optional byte Alpha=255)
{
  local float    HealthX;
  local color    DrawColor, BackColor;

// Color of bar relies on the player's current HP
HealthX = Width/MaxWidth;
```

```
  // Set default color to white
  DrawColor = Default.WhiteColor;
  DrawColor.B = 16;

  // If our HP is > 80%, decrease the amount of red
  if (HealthX > 0.8)
  {
    DrawColor.R = 112;
  }

// If our HP is < 40%, decrease the amount of green
  else if (HealthX < 0.4 )
  {
    DrawColor.G = 80;
  }

  DrawColor.A = Alpha;
  BackColor = default.WhiteColor;
  BackColor.A = Alpha;

  /** Health bar texture */
  DrawBarGraph(X,Y,Width,MaxWidth,Height,
  DrawCanvas,DrawColor,BackColor);
}
```

In the preceding code, you'll see that we've set our default color to white, but immediately after we've subtracted the amount of blue from 255 (full color) to 15. We could have just created our own variable for this, for example, `full color`; but this works just as well.

If our health is higher than 80 percent, then decrease the amount of red, thereby giving this a bit of an orange/yellow tint. That's what actually allows for this to look green when we have a full health bar. The next `if` statement declares that if we drop dangerously low to 40 percent of health or less, then we need to decrease the green value and paint the bar red.

7. With all of our drawing functions for the bar in place, we need a way to align it on screen.

```
/**********************************************************
* Returns corrected HUD position based on current res.
*
* @Param Position    Default position based on 1024x768 res
* @Param Width       Width of image based on 1024x768
* @Param Height      Height of image based on 1024x768
*
* @returns FinalPOS
**********************************************************/
function Vector2D CorrectedHudPOS(vector2D Position, float Width,
float Height)
{
  local vector2D  FinalPos;

  FinalPos.X = (Position.X < 0) ? Canvas.ClipX -
  (Position.X * ResScaleY) - (Width * ResScaleY)  :
                             Position.X * ResScaleY;
  FinalPos.Y = (Position.Y < 0) ? Canvas.ClipY -
  (Position.Y * ResScaleY) - (Height * ResScaleY) :
                             Position.Y * ResScaleY;

  return FinalPos;
}
```

`CorrectedHudPOS()` verifies that our HUD looks the same regardless of resolution. Now that UDK is supported on a number of mobile devices, this is more necessary than ever. Console developers generally only have a handful of resolutions to contend with, while iOS and Android developers now add a whole new set of problems into the mix.

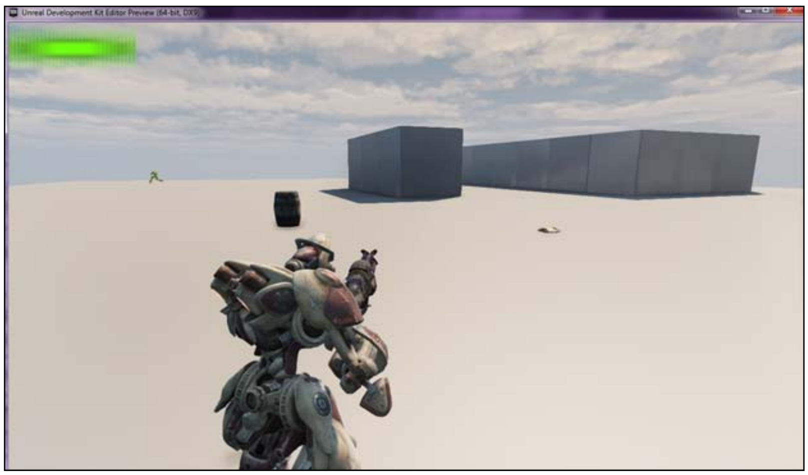
This function scales the location of our HUD based on the resolution and handles that sticky math for us.

8. With our functions out of the way, the only thing left to do in this class is to add the default properties.

```
DefaultProperties
{
  // Texture for HP bar
  BarTexture=Texture2D'UI_HUD.HUD.UI_HUD_BaseA'

  /** Hit Points */
  /** Corner Position of bar. + / - to X / Y changes which
      corner it is in */
  HPPosition=(X=0,Y=1)
  // Coords for the HP bar
  BarCoords=(U=277,V=494,UL=4,VL=13)
}
```

9. We now have a fully functioning HUD that displays our health bar in the top-left corner! There's still one last step, however. We need to tell our game to use our new HUD. In our `TutorialGame` class, add the following code in the `DefaultProperties` block:

```
DefaultProperties
{
  HUDType=class'Tutorial.TutorialHUD'
}
```



Notice how our bar changes to a red color when our health is dangerously low. You can also now see the first bar drawn, which illustrates the maximum capacity for our hit points.

How it works...

We start off by defining the size of our screen based on resolution. Our `PostRender()` function dynamically scales our HUD based on the default screen size of 1024 x 768. Our `DrawHUD()` function gets called during each frame of the main game loop, so it's constantly updating the screen every time the game itself becomes updated, based on changed information, such as the pawn's health.

We then added a function to draw our health bar. Really though, UDK calls this a tile with its `DrawTile()` function, but we prefer to call it a bar. This same function will be used for our ammo bar.

Our health bar needs a specific value to be drawn, however, to represent both the maximum and current health. We define and then use those in the `DrawHealthText()` function, and use those parameters in the `DrawHealthBar()` function.

The bar itself is drawn in the top-left corner of the screen, as defined by `CorrectedHudPos()`. It's a lot thrown at you at once, so bear with me. In `DrawHealthText()`, we define our position with the `Vector2D` variable `POS`. `POS` is actually set to the `CorrectedHudPos()`, which takes our `HPPosition` variable as a `Vector2D` parameter, then applies the math within `CorrectedHudPos()` to draw our text at the given position.

You'll see in our `DefaultParameters` block that we define what our `HPPosition` variable is. Setting the `Y` value to `1` will place the bar in the top of the screen, while setting it to `-1` will place it in the bottom. Setting the `X` value to `1` will place our bar against the top of the screen, and a negative value will do just the opposite! How convenient is that?

Drawing text for a player's health

Sure, it's great to have a bar to represent our health, but what if we want something more accurate? It's like reading the gas meter on your car, are you *really* teetering on empty, or can you push it just a tiny bit longer?

Because I don't condone living so dangerously, I suggest we draw an actual number on screen so that we know exactly how much health we are left with. We're going to take our existing functions and make only a few changes to allow this.

Getting ready

Open your IDE and have your `TutorialHUD` class available. We're going to make some additions.

How to do it...

For this recipe, we'll take what we learned in our previous recipe and add more functionality onto it. A colored health bar is great, but sometimes it's nice to know the exact amount of health remaining. Therefore we're going to add an integer next to our bar to give an exact number.

1. Let's start by adding our new variables.

```
/** Stores the HUD font*/
var Font    TutFont;

/** Stores how large the text should be displayed on screen*/
var float    TextScale;
```

2. We need to make some additions to our `DrawHealthText()` function.

```
  /** Draws text */
  Canvas.Font = TutFont;
  Canvas.SetDrawColorStruct(WhiteColor);
  Canvas.SetPos(HPTextOffsetPOS.X,HPTextOffsetPOS.Y);
  Canvas.DrawText
  (HPAmount,,TextScale * ResScaleY,TextScale * ResScaleY);
  Canvas.TextSize(HPAmount, TextSize.X, TextSize.Y);
```

We're calling a number of functions from the `Canvas` class here. We're setting the font as `TutFont`, which in our `Defaultproperties` block we will later declare to be `I_Fonts.MultiFonts.MF_HudLarge`. Then we set the position of our text, using `HPTextPoffsetPOS`, which we grab from our `HudOffset()` function. We'll cover this shortly.

We want to draw our text, so we define what our `HPAmount` and `HpAmountMax` variables are with the following bit of code:

```
HPAmount  = PlayerOwner.Pawn.Health;
HpAmountMax = PlayerOwner.Pawn.HealthMax;
```

We also needed to define a size for the text though, so we multiply our `TextScale` variable by the `ResScaleY` variable, which is the resolution size of the game.

The whole function should now look like the following code snippet:

```
/**********************************************************
* Draws the health text and bar
**********************************************************/
function DrawHealthText()
{
  local Vector2D    TextSize, POS, HPTextOffsetPOS;
  local int    HPAmount, HPAmountMax;
```

```
    /** Sets the bar position */
    POS = CorrectedHudPOS
    (HPPosition,BarCoords.UL,BarCoords.VL);

    /** Offsets the text from the bar */
    HPTextOffsetPOS = HudOffset(POS, HPTextOffset,);

    /** Sets the pawn's health amount */
    HPAmount = PlayerOwner.Pawn.Health;
    HpAmountMax = PlayerOwner.Pawn.HealthMax;

    /** Draws text */
    Canvas.Font = TutFont;
    Canvas.SetDrawColorStruct(WhiteColor);
    Canvas.SetPos(HPTextOffsetPOS.X,HPTextOffsetPOS.Y);
    Canvas.DrawText
    (HPAmount,,TextScale * ResScaleY,TextScale * ResScaleY);
    Canvas.TextSize(HPAmount, TextSize.X, TextSize.Y);

    /** Draws health bar */
    DrawHealthBar
    (POS.X, POS.Y, HPAmount, HpAmountMax, 80, Canvas);
  }
```

3. One more function is necessary to offset the text from our bar, and looks nearly identical to our `CorrectedHudPos()` function. It takes `CorrectedHudPos` and offsets it by a value defined in the `DefaultProperties` block.

```
/**********************************************************
* Offsets HUD and places the bottom/right portion of image
* at coords if. If offset is great than 0, & if width /
* height are supplied @Param Position Default position
* based on 1024x768 res
* @Param Offset Value to offset the text from the texture
* @returns FinalPOS
*********************************************************/
function Vector2D HudOffset(vector2D HUDPosition, vector2D
Offset, optional float Width, optional float Height)
{
  local vector2D FinalPos;

  FinalPos.X = (Offset.X < 0 && Width != 0) ?
  HUDPosition.X - (Width * ResScaleY) +
  (Offset.X * ResScaleY) :
  HUDPosition.X + (Offset.X * ResScaleY);
```

```
    FinalPos.Y = (Offset.Y < 0 && Height != 0) ?
    HUDPosition.Y - (Height * ResScaleY) +
    (Offset.Y * ResScaleY) :
    HUDPosition.Y + (Offset.Y * ResScaleY);

    return FinalPos;
  }
```

4. Let's define a few values in our `DefaultProperties` block.

```
DefaultProperties
{
....
  /** Hit Points */
  // Offsets text from bar
  HPTextOffset=(X=220,Y=0)

  // Text scale
  TextScale=0.25

// Font used for the text
  TutFont="UI_Fonts.MultiFonts.MF_HudLarge"
....
}
```

The `X` value in our offset is what offsets the text from our bar. With `X` set to `220`, it will move the text 220 bars over to the right of our current text position.

5. Compile the code and take a look.

Our current health is displayed on the top-right corner of our health bar. Let's take a look at how it changes as our health decreases as well:

As you can see, the integer in the corner certainly makes it a bit easier to quickly digest exactly how much health is remaining. In the next two recipes we'll cover how to do this with our ammo!

How it works...

Not much is going on here. We add a few necessary variables to our `DrawHealthText()` function to allow the text to be displayed. From there we're calling various `Canvas` functions to perform activities such as drawing the actual text, coloring, positioning on screen, and sizing.

We do add a new function to the mix here, however, with the addition of `HudOffset()`. The purpose of this is to align our text in the same corner as our health bar, and then offset it by a given value which we declare in our `DefaultProperties` block.

Displaying a bar for the player's ammo

Visualizing our player's health is essential, but their ammunition count is nearly just as important. Games in recent memory have taken alternative approaches to illustrate this kind of information, including the *Halo* series, which displays the current ammo count on the end of the rifle.

We could mimic this with Scaleform, but again, that's a discussion for another day. We're going to keep things simple again, as we aren't using Scaleform. So let's get started with displaying our ammo bar!

Getting ready

Open your IDE and have it ready to edit our `TutorialHUD` class again. We're only going to make changes to this class, all of which will be similar to what we've done already as now we've established a solid base to work from.

How to do it...

This is going to be very similar to the steps we took in the recipe for our health. The only major change we need to make here is to the properties we'll be accessing and using. Rather than displaying our pawn's health, we'll be using the ammo property for our pawn's currently equipped weapon.

1. Let's begin by adding our variables. We won't need many, as we have a solid foundation already.

```
/** Positoning for Ammo bar and text */
var vector2d    AmmoPosition, AmmoTextOffset;
var TextureCoordinates    AmmoCoords;
```

2. We need a function to draw our ammo. For now we'll just call it `DrawAmmoText()`, as it will be used to draw text in the next recipe.

```
/**********************************************************
* Draws the ammo text and bar
**********************************************************/
function DrawAmmoText()
{
  local Vector2D    POS;
  local Int     AmmoCount, MaxAmmo;

  /** Sets the variables */
  AmmoCount = UTWeapon(PawnOwner.Weapon).AmmoCount;
  MaxAmmo = UTWeapon(PawnOwner.Weapon).MaxAmmoCount;

  /** Sets the current bar position */
  POS = CorrectedHudPOS
  (AmmoPosition,AmmoCoords.UL,AmmoCoords.VL);

  /** Draws Ammo Bar */
  DrawAmmoBar
  (POS.X, POS.Y, AmmoCount, MaxAmmo, 80, Canvas);
}
```

This looks very similar to our `DrawHealthBarText()` function used in the previous chapter. We set our `AmmoCount` and `MaxAmmo` variables by pulling these values from our pawn's currently selected weapon.

We offset the bar's position again by using the `CorrectedHudPOS()` function we used in the last recipe as well. This time however, the bar will be in the top-right corner of the screen.

At the bottom of the function, we have a call to our `DrawAmmoBar()` function, which we'll get to know in the next step. It looks nearly identical to our `DrawHealthBar()` function, except that we're using our `AmmoCount` and `MaxAmmo` variables as parameters.

3. Let's create that `DrawAmmoBar()` function. It appears and operates just like our `DrawHealthBar()` function in the previous recipe.

```
/*********************************************************
* Draw player's ammo. Adjusts the bar color based on
* available ammo
*********************************************************/
simulated function DrawAmmoBar(float X, float Y, float
Width, float MaxWidth, float Height, Canvas DrawCanvas,
optional byte Alpha=255)
{
  local float AmmoX;
  local color DrawColor, BackColor;

// Color of bar relies on the player's current ammo
  AmmoX = Width/MaxWidth;

  // Set default color to white
  DrawColor = Default.WhiteColor;

  // Decrease the amount of blue
  DrawColor.B = 16;

  // If our ammo is > 80%, decrease the amount of red
  if (AmmoX > 0.8)
  {
    DrawColor.R = 112;
  }

// If our ammo is < 40%, decrease the amount of green
  else if (AmmoX < 0.4 )
  {
    DrawColor.G = 80;
  }
  DrawColor.A = Alpha;
  BackColor = default.WhiteColor;
  BackColor.A = Alpha;

  /** Ammo bar texture */
  DrawBarGraph(X,Y,Width,MaxWidth,Height,
  DrawCanvas,DrawColor,BackColor);
}
```

This should look identical to our `DrawHealthBar()` function, except we've changed the parameters for health and turned them into ammo. Our ammo bar will now change color as we continue to drain our ammo supply.

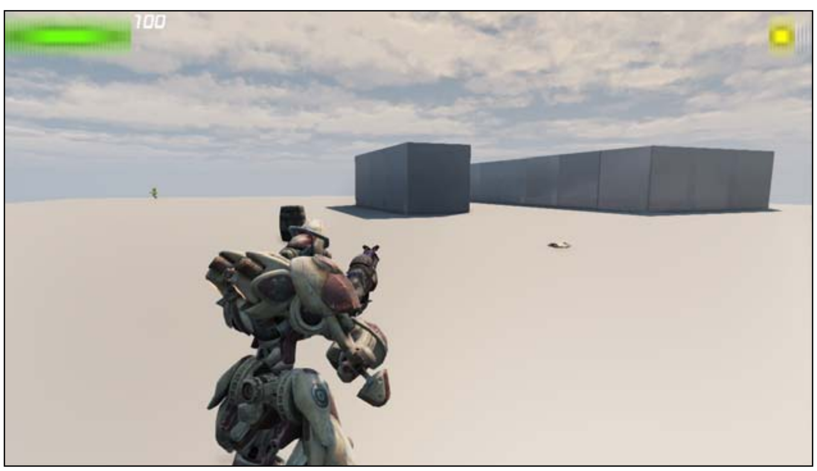
4. Let's make our `DrawHud()` function call `DrawHealthText()`, which will put everything in motion. The whole function should now look like this:

```
/*********************************************************
* Draws the HUD
*********************************************************/
function DrawHUD()
{
  super.DrawHUD();
  DrawHealthText();
  DrawAmmoText();
}
```

5. Finally, let's add the default properties as shown in the following code snippet:

```
/** Ammo */
/** Corner position of bar. + / - to X / Y changes which
 corner it appears in */
AmmoPosition=(X=-1,Y=1)
// Coordinates of ammo bar
AmmoCoords=(U=277,V=494,UL=76,VL=126)
```

6. Compile the code and test it out for yourself. You'll see that the ammo bar is located in the top-right corner. Remember, changing the `X` value for the `AmmoPosition` parameter from `-1` to `1` will adjust which corner the bar sits in.



You'll also notice that our bar starts half full and is colored yellow. That's fine. Our weapon's default ammo is half of its maximum capacity, so it will continue to look this way until we locate more ammo.

How it works...

The process is done in the same manner as our health bar. We start off by adding a function to draw our ammo bar. Really though, UDK calls this a tile with its `DrawTile()` function, but we prefer to call it a bar.

Our ammo bar needs a specific value to be drawn, however, to represent both the maximum and current ammo. We define and then use those in the `DrawAmmoText()` function, and use those parameters in the `DrawAmmoBar()` function.

The bar itself is drawn in the top-right corner of the screen, as defined by `CorrectedHudPos()`. It's a lot thrown at you at once, so bear with me. In `DrawAmmoText()`, we define our position on screen with the `Vector2D` variable `POS`. `POS` is actually set to our `CorrectedHudPos()` function which is a function that takes `AmmoPosition` as a `Vector2D` parameter and uses that information to determine where the ammo will be drawn on screen.

Drawing text for the player's ammo

We've got a bar to illustrate our current ammo count against our maximum capacity, but it's always useful to see exact values. Therefore we're going to add text for both of these values, similar to how we did it for our health.

Getting ready

Open your IDE and have your `TutorialHUD` class ready to be altered.

How to do it...

This is going to be very similar to the steps we took in the recipe for displaying an integer to represent our pawn's health. The only major change we need to make here is to the properties that we'll be accessing and using. Rather than displaying our pawn's health, we'll be using the ammo property for our pawn's currently equipped weapon.

1. Let's start by adding the only variable we'll need for this one. It stores the offset we apply to the text from our ammo bar.

```
/** Positioning for Ammo bar and text */
var vector2d    AmmoTextOffset;
```

2. In our `DrawAmmoText()` function, let's add our new local variables and `Canvas` functions to draw the text.

```
/*********************************************************
* Draws the ammo text and bar
```

```
*******************************************************/
function DrawAmmoText()
{
  local Vector2D    TextSize, AmmoTextOffsetPOS;
  local String    Text;

  /** Sets the variables */
  Text = AmmoCount @ "|" @ MaxAmmo;

  //** Offsets the text from the bar */
  AmmoTextOffsetPOS = HudOffset(POS, AmmoTextOffset,);

  //** Draws text */
  Canvas.Font = TutFont;
  Canvas.SetDrawColorStruct(WhiteColor);
  Canvas.SetPos(AmmoTextOffsetPOS.X,AmmoTextOffsetPOS.Y);
  Canvas.DrawText
  (Text,,TextScale * ResScaleY,TextScale * ResScaleY);
  Canvas.TextSize(AmmoCount, TextSize.X, TextSize.Y);
}
```

Our entire function should now look like this:

```
/*******************************************************
* Draws the ammo text and bar
*******************************************************/
function DrawAmmoText()
{
  local Vector2D    TextSize, POS, AmmoTextOffsetPOS;
  local Int    AmmoCount, MaxAmmo;
  local String    Text;

  /** Sets the variables */
  AmmoCount = UTWeapon(PawnOwner.Weapon).AmmoCount;
  MaxAmmo = UTWeapon(PawnOwner.Weapon).MaxAmmoCount;
  Text = AmmoCount @ "|" @ MaxAmmo;

  /** Sets the current bar position */
  POS = CorrectedHudPOS
  (AmmoPosition,AmmoCoords.UL,AmmoCoords.VL);

  /** Offsets the text from the bar */
  AmmoTextOffsetPOS = HudOffset(POS, AmmoTextOffset,);

  /** Draws text */
  Canvas.Font = TutFont;
  Canvas.SetDrawColorStruct(WhiteColor);
  Canvas.SetPos(AmmoTextOffsetPOS.X,AmmoTextOffsetPOS.Y);
  Canvas.DrawText
  (Text,,TextScale * ResScaleY,TextScale * ResScaleY);
  Canvas.TextSize(AmmoCount, TextSize.X, TextSize.Y);
```

```
    /** Draws Ammo Bar */
    DrawAmmoBar
    (POS.X, POS.Y, AmmoCount, MaxAmmo, 80, Canvas);
  }
```

This should look very similar to what we have for our health bar text. Again, we're using our `AmmoCount` and `MaxAmmo` count, just as we did for our ammo bar. We are offsetting the text from our ammo bar, but because the bar is on the right side of the screen we won't have any room to display the text to the right. Therefore, we're going to offset it to the left.

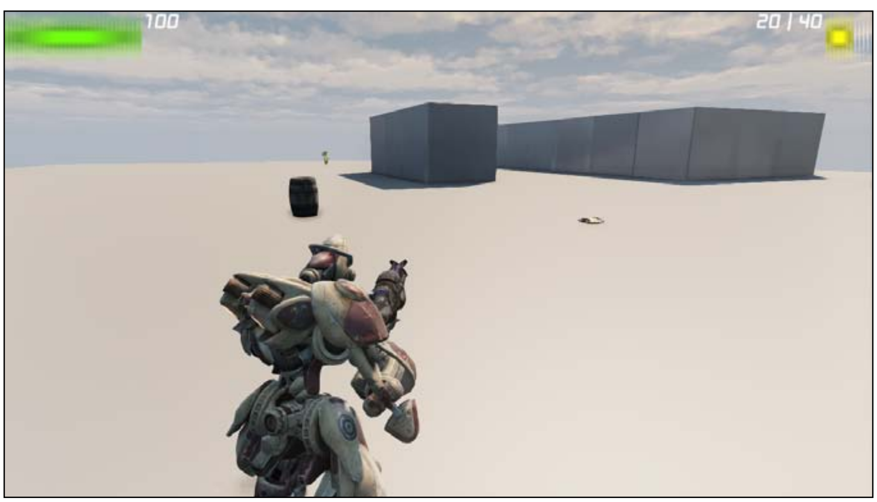
Our `Text` variable consists of our pawn's weapon's current ammo, followed by the `@` sign, which concatenates the string. I've placed a `|` character in quotes to divide the current ammo variable from the maximum ammo variable as well.

3. Of course, let's not forget to add our variable to our `DefaultProperties` block. We'll be offsetting our text `-108` pixels from the right-hand corner of our ammo bar, so that the text is placed right where the bar ends.

> -108 pixels may not always be enough space to fit your ammo count; you may want to consider offsetting it by a different value if you are using a weapon whose ammo count may be more than five characters in length.

```
// Offsets text from bar
AmmoTextOffset=(X=-108, Y=0)
```

4. Compile the code and take a look; we've got text on screen to represent our current and maximum ammo capacity!



How it works...

This is almost identical to our `DrawHealthText()` function. We add one variable to our `DrawAmmoText()` function to allow the text to be displayed. From there we're calling various `Canvas` functions to perform activities such as drawing the text, coloring, positioning on screen, and sizing.

Again, we use `HudOffset()` to align our text in the same corner as our ammo bar, and then offset it by a given value which we declare in our `defaultproperties` block.

Drawing the player's name on screen

When I think of drawing the player's name on screen the *Doom* guy always comes to my mind. The image of his mug shot centered on the bottom of my screen is perhaps burned into my mind forever. With that in mind, I thought it would be nice to display our pawn's name on screen in the same manner. Of course, we could always replace the pawn's name with something like an image of its face just as easily.

Sometimes it's nice to know exactly which player you are controlling, especially in a multiplayer game. Often you'll just want to know if you are LocalPlayer01 or LocalPlayer02. You could, of course, have a user input their name before a game starts and then grab that data and draw it on screen as well. Let's discover exactly how to draw this on screen.

Getting ready

Open your IDE and have your `TutorialHUD` class ready to be altered.

Things are going to be pretty simple in this chapter, as we've done most of this work in our previous recipes.

How to do it...

Drawing a player's name on screen isn't much different from drawing an integer to represent the pawn's health or ammo count. We're going to store texture coordinates for our name, along with a position in 2D space for where it will be displayed on screen. From there we grab the local player controller's name and draw it on screen using the coordinates we stored earlier.

1. Just as we did before, let's begin by adding the variables for our new text.

```
/** Position the name text */
var Vector2D    NamePosition;
var TextureCoordinates        NameCoords;
```

2. We need a function to draw the player's name on the screen.

```
/**********************************************************
* Draws the text for the pawn's name
**********************************************************/
function DrawPawnNameText()
{
  local Vector2D    TextSize, POS;
  local String    PlayerName;

  /** Sets the player name */
  PlayerName =
  PlayerOwner.PlayerReplicationInfo.PlayerName;

  /** Sets the name position */
  POS = CorrectedHudPOS
  (NamePosition,NameCoords.UL,NameCoords.VL);

  /** Draws the text */
  Canvas.Font = TutFont;
  Canvas.SetDrawColorStruct(WhiteColor);
  Canvas.SetPos(0.9f *(Canvas.ClipX/2), POS.Y);
  Canvas.DrawText
  (PlayerName,,TextScale / RatioX,TextScale / RatioY);
  Canvas.TextSize(PlayerName, TextSize.X, TextSize.Y);
}
```

We have a `Vector2D` variable to store the size of our text as well as the position on screen. Moreover, we have a string to store our player's name.

Our name is grabbed from our `PlayerController` class, in our case `PlayerOwner`. If you look into `HUD.uc`, which is what our `TutorialHUD` class extends from, you'll see that `PlayerOwner` is a player controller, and defined as the player controller that this HUD belongs to.

We cut the screen in half with the call to `Canvas.SetPos()`, which places our text in the middle of the screen. The problem, however, is that it centers the origin of the text, which is the top-left corner. Because of this we need to offset it slightly. Therefore, we use the float `0.9` to compensate for this difference, as seen by the line `Canvas.SetPos(0.9f *(Canvas.ClipX/2), POS.Y);`.

3. Of course, we don't want to forget to have `DrawHud()` call our function either. It should now look like this:

```
/**********************************************************
* Draws the HUD
**********************************************************/
function DrawHUD()
```

```
{
  super.DrawHUD();
  DrawHealthText();
  DrawPawnNameText();
  DrawAmmoText();
}
```

4. The final step is to add some information pertaining to our two new variables in the `defaultproperties` block as shown in the following code snippet:

```
/** Name */
/** Corner Position of name text. + / - to X / Y changes
  which corner it is in */
NamePosition=(X=0, Y=0)

// Coordinates for the player name text
NameCoords=(U=0,V=0,UL=0,VL=0)
```

   You'll see that our `X` and `Y` value for `NamePosition` are set to `0`. We do this because we want the name to be centered on the screen and flash with the health and ammo bars at the top.

5. Compile the code and yield the results!

How it works...

This is very similar to our `DrawHealthText()` and `DrawAmmoText()` functions. We add two functions to allow the text to be displayed. From there we're calling various `Canvas` functions to perform activities such as drawing the text, coloring, positioning on screen, and sizing.

Again, we use `HudOffset()` to align our text with the top of the health bars and on the center of the screen.

Creating a crosshair

One of the most valuable bits of information in a first person shooter is the crosshair. This is obviously important for knowing exactly where our shots will land and where our pawn is focusing its attention.

We're going to be drawing a crosshair directly in the center of the screen, which is precisely where our projectiles will be firing.

Getting ready

Open your IDE and have the `TutorialHUD` class ready to edit. We're going to be adding quite a few variables here.

How to do it...

To create crosshairs we'll need to declare variables which store our data, as well as a function to draw the actual crosshair. We'll then take our newly created `DrawWeaponCrosshair()` function and add it to `DrawHUD()`, so that it gets drawn on screen with the rest of our HUD.

1. Let's start by declaring our variables.

```
/***********************************************************
* Crosshairs
***********************************************************/
/** Used for scaling the size of the crosshair */
var float    ConfiguredCrosshairScaling;

/** Coordinates for crosshairs */
var UIRoot.TextureCoordinates    CrossHairCoordinates;
```

```
/** Holds the image to use for the crosshair */
var Texture2D    CrosshairImage;

/** Various colors */
var const color    BlackColor;

/** color to use when drawing the crosshair */
var config color    CrosshairColor;
```

We're going to use much of the same code that Epic uses for drawing the crosshair. Rather than go through the convoluted processes of drawing the crosshair through `PostRender()` and doing a number of checks, we're just going to draw the crosshair at all times.

2. We'll call our function, `DrawWeaponCrosshair();`, and begin by defining the local variables.

```
/**********************************************************
* Draws the crosshair
**********************************************************/
simulated function DrawWeaponCrosshair()
{
  local vector2d    CrosshairSize;
  local float    x,y,ScreenX, ScreenY;
  local MyWeapon    W;
  local float    TargetDist;

  /** Set weapon and target distance */
  W = MyWeapon(PawnOwner.Weapon);
  TargetDist = W.GetTargetDistance();

  /** Sets crosshair size */
  CrosshairSize.Y = ConfiguredCrosshairScaling *
  CrossHairCoordinates.VL * Canvas.ClipY/720;

  CrosshairSize.X = CrosshairSize.Y *
  ( CrossHairCoordinates.UL / CrossHairCoordinates.VL );

  /** Sets screen dimensions */
  X = Canvas.ClipX * 0.5;
  Y = Canvas.ClipY * 0.5;
  ScreenX = X - (CrosshairSize.X * 0.5);
  ScreenY = Y - (CrosshairSize.Y * 0.5);
```

We start by setting the weapon and the weapon's target distance. This is simply used for the `Z` value when we call the `Canvas.SetPosition()` function in our next set of code. We also set the crosshair size through a number of factors. Our crosshair scaling float is defined in our `defaultproperties` block; it's currently set to `1`. We then multiply it by the crosshair coordinates (also defined in the `defaultproperties` block), and again by the canvas size.

The screen dimensions are then set after that. The `Y` and `X` values are set to half of the screen clipping size. The screen clip essentially determines where the edges of the screen are. Cutting this in half lets us know where the middle of the screen is. Finally, we set `ScreenX` and `ScreenY` to be our previous value, minus half of the crosshair size. This finds the center point of our crosshair.

3. Let's add the second half of the function as shown in the following code:

```
if ( CrosshairImage != none )
{
  /** Draw crosshair drop shadow */
  Canvas.DrawColor = BlackColor;

  Canvas.SetPos( ScreenX+1, ScreenY+1, TargetDist );

  Canvas.DrawTile(CrosshairImage,CrosshairSize.X,
  CrosshairSize.Y, CrossHairCoordinates.U,
  CrossHairCoordinates.V, CrossHairCoordinates.UL,
  CrossHairCoordinates.VL);

  /** Draw crosshair */
  CrosshairColor = Default.CrosshairColor;

  Canvas.DrawColor = CrosshairColor;

  Canvas.SetPos(ScreenX, ScreenY, TargetDist);

  Canvas.DrawTile(CrosshairImage,CrosshairSize.X,
  CrosshairSize.Y, CrossHairCoordinates.U,
  CrossHairCoordinates.V, CrossHairCoordinates.UL,
  CrossHairCoordinates.VL);
}
```

We're going to draw our crosshair twice in this example. The first time is to add a nice drop shadow to the crosshairs by offsetting our values by one pixel and shading the crosshair black.

In the next step, we draw the actual crosshair which we will see. Our `Drawcolor` parameter is defined in our `defaultproperties` block. We are setting the position to be the center of the screen, and using the `CrosshairImage` (also defined in the properties block) as the texture to be drawn.

4. Let's add those default properties now.

```
/** Crosshairs - from UTWeapon */
// Crosshair image
  CrosshairImage=Texture2D'UI_HUD.HUD.UTCrossHairs'
// Crosshair location
  CrossHairCoordinates=(U=192,V=64,UL=64,VL=64)

/** Crosshairs - From UTHUDBase */
// Crosshair size
  ConfiguredCrosshairScaling=1.0
// Crosshair color
  BlackColor=(R=0,G=0,B=0,A=255)
```

5. Our default properties block should look like this now:

```
defaultproperties
{
  /** Textures and font */
  // Text scale
  TextScale=0.25
  // Font used for the text
  TutFont="UI_Fonts.MultiFonts.MF_HudLarge"
  // Texture for HP bar
  BarTexture=Texture2D'UI_HUD.HUD.UI_HUD_BaseA'

  /** Ammo */
  /** Corner Position of bar. + / - to X / Y changes which
  corner it is in */
  AmmoPosition=(X=-1,Y=1)
  // Offsets text from bar
  AmmoTextOffset=(X=-108, Y=0)
  // Coordinates of ammo bar
  AmmoCoords=(U=277,V=494,UL=76,VL=126)
  /** Corner Position of bar. + / - to X / Y changes which
  corner it is in */
  /** Hit Points */
  HPPosition=(X=0,Y=1)
  // Offsets text from bar
  HPTextOffset=(X=220,Y=0)
  // Coords for the HP bar
  BarCoords=(U=277,V=494,UL=4,VL=13)

  /** Name */
  /** Corner Position of name text. + / - to X / Y changes
  which corner it is in */
  NamePosition=(X=0, Y=0)
  // Coordinates for the player name text
  NameCoords=(U=0,V=0,UL=0,VL=0)

  /** Crosshairs - from UTWeapon */
```

```
    // Crosshair image
    CrosshairImage=Texture2D'UI_HUD.HUD.UTCrossHairs'
    // Crosshair location
    CrossHairCoordinates=(U=192,V=64,UL=64,VL=64)

    /** Crosshairs - From UTHUDBase */
    // Crosshair size
    ConfiguredCrosshairScaling=1.0
    // Crosshair color
    BlackColor=(R=0,G=0,B=0,A=255)
}
```

6. Don't forget to add our `DrawWeaponCrosshair()` function to our `DrawHud()` function!

```
/***********************************************************
* Draws the HUD
***********************************************************/
function DrawHUD()
{
  super.DrawHUD();

  DrawHealthText();
  DrawPawnNameText();
  DrawAmmoText();
  DrawWeaponCrosshair();
}
```

7. Compile the code and view your results!

How it works...

We start by declaring a few variables that we'll use on our crosshair. Within our `DrawWeaponCrosshair()` function we actually draw the crosshair twice. The first layer consists of a dark drop shadow which is slightly offset from the center of the screen to give our crosshair some depth.

The second crosshair is the one the end user will actually see. We paint it right in the center of the screen, which is exactly where our projectiles fire.

8
Miscellaneous Recipes

In this chapter, we will be covering the following recipes:

- Creating an army of companions
- Having enemies flash quickly as their health decreases
- Creating a crosshair that uses our weapon's trace
- Changing the crosshair color when aiming at a pawn
- Drawing a debug screen
- Drawing a bounding box around pawns

Introduction

In the previous chapters, we've covered topics that ranged from weapons and navigation, to a heads-up display and AI. In this chapter, our recipes are going to cover things that may not necessarily fit in one particular chapter, but are still very valuable in a number of applications.

We'll go over a new scheme for aiming our weapons and drawing a crosshair, as well as allowing our pawn to flash continuously as its health depreciates, among other things.

Creating an army of companions

Going through a game alone is seldom any fun. We're social creatures so we enjoy the company of others. What better way to celebrate this than by creating a group of companions to follow us along on our journey? In this chapter, we'll explore how to create a small party of companions who spawn at our location and reap the rewards of our adventure!

Getting ready

Start by having your IDE open and ready to make some changes. We won't have to create any new classes, but we will alter the behavior of our existing ones by adding some functions.

How to do it...

1. Let's begin by overriding the `SpawnDefaultFor()` function in our `TutorialGame` class. Once we spawn our default pawn, `PlayerSpawned()` in our player controller is to be called. This function spawns our companion pawns.

```
/*****************************************************************
 * Returns a pawn of the default pawn class
 * @param     NewPlayer - Controller for whom this pawn is spawned
 * @param     StartSpot - PlayerStart at which to spawn
 * pawn
 * @return      pawn
*****************************************************************/
function Pawn SpawnDefaultPawnFor
(Controller NewPlayer, NavigationPoint StartSpot)
{
  local Pawn ResultPawn;
   ResultPawn = super.SpawnDefaultPawnFor
  (NewPlayer, StartSpot);

  if(ResultPawn != none)
  {
    TutorialPlayerController(NewPlayer).PlayerSpawned
    (StartSpot);
  }

  return ResultPawn;
}
```

2. We need to tell `TutorialPawn` to spawn our default controller. This is actually defined in `UTPawn`, but we need to place it in our `PostBeginPlay()` function so that our companions spawn right as the map loads.

```
PostBeginPlay()
{
  ....
  /** called from UTPawn, spawns the default controller */
  SpawnDefaultController();
  ....
}
```

3. Now we need to do some work in our player controller class, in our case, `TutorialPlayerController` class. Let's add the variable array that we'll use to store the number of companions we want to spawn as shown in the following code snippet:

```
/** Array used for setting the number of spawned companions
 */
var Pawn    Companions[3];
```

4. Let's create a function in our `TutorialPlayerController` class to spawn these companions as shown in the following code:

```
/*****************************************************************
* Spawns companion pawns
*****************************************************************/
function PlayerSpawned(NavigationPoint StartLocation)
{
  Companions[0] = Spawn(Class'TutorialPawn',,,
  StartLocation.Location - vect(75,100,0),
  StartLocation.Rotation);
  Companions[1] = Spawn(Class'TutorialPawn',,,
  StartLocation.Location - vect(150,120,0),
  StartLocation.Rotation);
  Companions[2] = Spawn(Class'TutorialPawn',,,
  StartLocation.Location - vect(200,280,0),
  StartLocation.Rotation);
}
```

The fist parameter requires a class, so we're using our `TutorialPawn` class as the pawn we want to spawn, and then we're setting the spawn location.

`StartLocation.Location` is where our pawn spawns, so we're offsetting each of our companions by a bit, and then matching their rotation with ours, so that we're all facing the same way upon spawning.

5. That's all there is to it! Compile the project and watch what happens when you spawn. You'll have three followers alongside you! They'll even follow once you reach a certain distance. They're following you because the player controller attached to their pawn class is following you.

How it works...

We override the `SpawnDefaultFor()` function in our `TutorialGame` class, which spawns our default pawn in our `TutorialGame` class. Upon doing so, `PlayerSpawned()` in our `TutorialPlayerController` class is to be called. This function spawns our companion pawns.

We also tell our `TutorialPawn` class to spawn our default controller by making a call to `SpawnDefaultController()`. This is actually defined in `UTPawn`, which is placed in our `PostBeginPlay()` function so that our companions spawn right as the map finishes loading.

Finally, we create a function in our `TutorialPlayerController` class to spawn our companions, with `PlayerSpawned()`. This simply takes an array, `Companions[3]`, which we defined at the top of the class, and spawns each one of them at a set location.

Having enemies flash quickly as their health decreases

Gamers require some sort of feedback each time something happens to their pawn. This can be auditory, visual, or kinetic (controller vibrations). Most frequently the cues are visual, as many controllers still do not support a vibration feature, however. Unreal Tournament pawns flash red briefly each time they are hit. While UDK natively does this, why not extend that idea, and bring back something from the NES era of gaming, wherein enemies flash red as their health decreases, and continues to do so more quickly as it reaches zero.

Getting ready

Start by having your IDE open and ready to make some changes. We won't have to create any new classes, but we will alter the behavior of our existing ones by adding some functions.

How to do it...

1. Let's start by declaring the variables we'll use.

```
/** Used for flashing damage as pawn's HP drops */
var float           DamageOverlayTime;
var LinearColor     DamageBodyMatColor;
```

   These will be used to store how long the damage will flash over our pawn, along with the color used.

2. Let's set the values for those variables in our default properties right now:

```
DefaultProperties
{
  ....
  /** Used for flashing damage as pawn's HP drops */
  // The flash lasts this long (float)
  DamageOverlayTime=.1
  // Sets the pawn to flash red
  DamageBodyMatColor=(R=10)
  ....
}
```

3. We really only need to add two new functions here, and alter another pre-existing one. Let's add our first new function now, `FlashDmgTimer()`:

```
/*****************************************************************
* NES style flashing damage timer to indicate how hurt a
* pawn is
*****************************************************************/
function FlashDmgTimer()
{
  if (Health < HealthMax * .5)
  {
    'log("HP is less than 50%");
    SetTimer(2.2, true, 'FlashDmg');
  }

  else if (Health < HealthMax * .25)
  {
    'log("HP is less than 25%");
    SetTimer(1.5, true, 'FlashDmg');
  }

  else if (Health < HealthMax * 0.1)
  {
    'log("HP is less than 10%");
    SetTimer(0.7, true, 'FlashDmg');
  }
}
```

If our pawn's health is less than 50 percent, we call `SetTimer()`, which tells `FlashDmg()` to be called once every `2.2` seconds. This is a simple visual indication that our pawn (or any enemy who uses this function) is hurt. We then add two more `if` statements that work in the same manner, and simply adjust how frequently `FlashDmg()` is called. The more frequent it is called, the quicker the flashing occurs.

4. The `FlashDmg()` function is very simple. It sets our body material color by calling the `SetBodyMatColor()` function and passing in our color and time parameters that we defined just before this.

```
/*****************************************************************
* Sets the flashing overlay on the pawn to indicate damage taken
*****************************************************************/
simulated function FlashDmg()
{
  SetBodyMatColor(DamageBodyMatColor, DamageOverlayTime);
}
```

5. `PlayHit()` is called each time a pawn is hit. We want to override the current version of it and add our own behavior.

```
/*****************************************************************
* Called when a pawn is hit
```

```
**************************************************************/
function PlayHit(float Damage, Controller InstigatedBy, vector
HitLocation, class<DamageType> damageType, vector Momentum,
TraceHitInfo HitInfo)
{
  Super.PlayHit(Damage, InstigatedBy, HitLocation,
  DamageType, Momentum, HitInfo);

  // Calls flash timer
  FlashDmgTimer();
}
```

> I may not have mentioned this before, so this is an excellent time to do so. The `Super` function is used inside of a function, and calls the name of the function it resides in from the parent class.
>
> So in this example, `Super` is calling `PlayHit` from `UTPawn`, which is the class `TutorialPawn` is extending from. It's useful when you only want to add functionality to a function without overwriting what it does in the parent class.

Essentially, we're having it call our `FlashDmgTimer()` function each time the pawn is hit. This checks if our pawn's health is below the highest threshold; in our case, this is 50 percent health. If it is not, then the body material color is set to its default value.

6. Compile your project and either lower your own health down to 50 percent or less, or do the same to another pawn. Watch as the red material flashes over their body, and continues to do so more quickly as they reach the next threshold.

How it works...

We create a function called `FlashDmgTimer()`, which is really just a series of `if` statements that check to see if our pawn's health is below a certain threshold, and if so, calls `FlashDmg()` at a set interval. The lower the pawn's health is, the more often the pawn flashes red.

`FlashDmg()` is really just calling another function inside it, `SetBodyMatColor()`. We pass in our `DamageBodyMatColor` and `DamageOverlayTime` parameters to determine the new body material color (red) and how quickly it flashes over that pawn.

Finally, the `FlashDmgTimer()` function is called each time the pawn is hit.

Creating a crosshair that uses our weapon's trace

We've previously covered crosshairs and aiming in other recipes, but we're going to handle it in a different manner now. Rather than always have our pawn fire at the direct center of the screen, we'll change some behaviors so that our projectiles fire using the gun's rotation. Both look and feel more accurate and realistic.

We'll do this by drawing a trace from the barrel (socket) of our weapon, and using the weapon's rotator to draw the crosshair at the end of the trace.

Getting ready

Start by having your IDE open and ready to make some changes. We won't have to create any new classes, but we will alter the behavior of our existing ones by adding some functions.

How to do it...

1. The first thing we need to do is override the `GetBaseAimRotation()` function in our `TutorialPawn` class:

```
/*******************************************************************
* Returns base Aim Rotation without any adjustment.
* We simply use our rotation. Only use this if you want
* your weapon trace to follow where your gun is pointed.
* Comment out if you want to fire in middle of screen.
* @return  POVRot
*******************************************************************/
simulated singular event Rotator GetBaseAimRotation()
{
```

```
  local rotator  POVRot;

  // If we have no controller, we simply use our rotation
  POVRot = Rotation;

  // If our Pitch is 0, then use RemoveViewPitch
  if( POVRot.Pitch == 0 )
  {
    POVRot.Pitch = RemoteViewPitch << 8;
  }

  return POVRot;
}
```

This function sets our base aim rotator to use our point of view. It is called by our `GetAdjustedAimFor()` function in our `TutorialPlayerController` class.

That function gives our controller an opportunity to adjust the aiming of the pawn. Things such as aim error, auto aiming, and AI help for consoles are things we could put here. The weapon class requests `BaseAimRotation` before firing in order to compensate for any of these variables.

> `BaseAimRotation` takes the rotation from a weapon before any math or alterations are applied to it, such as auto aim, lock-on, or any other adjustments or variables may be applied to it.

It sets `BaseAimRot` by checking that we have a pawn. If we do have one, it uses the pawn's `GetBaseAimRotation()` function, otherwise it uses the weapon's rotation, without applying any sort of modifier like auto aim, as seen by this line:

```
  BaseAimRot = (Pawn != None) ? Pawn.GetBaseAimRotation()
                              : Rotation;
```

2. Therefore, we need to set our `GetBaseAimRotation` function now.

```
/*****************************************************************
* Returns base Aim Rotation without any adjustment.
* We simply use our rotation. Only use this if you want
* your weapon trace to follow where your gun is pointed.
* Projectiles will now follow your trace. Comment out if
* you want to fire in middle of screen.
* @return POVRot
*****************************************************************/
simulated singular event Rotator GetBaseAimRotation()
{
  local rotator  POVRot;
```

```
  // We simply use our rotation
  POVRot = Rotation;

  // If our Pitch is 0, then use RemoveViewPitch
  if( POVRot.Pitch == 0 )
  {
    POVRot.Pitch = RemoteViewPitch << 8;
  }

  return POVRot;
}
```

Here we are using our point of view rotation, that is, the rotation from our camera and not the pawn in this situation. This is what allows us to fire directly where the weapon is pointed by following a trace from the weapon's socket. Previously the weapon ignored our point of view and simply fired directly towards the center of the screen.

3. Our aim is corrected, but now we need a crosshair to display. This next function is a large one, but we'll break it down into small and logical steps. Let's create our `CheckCrosshairOnFriendly()` function as shown in the following code snippet:

```
/*****************************************************************
* Draws the crosshair
*****************************************************************/
function bool CheckCrosshairOnFriendly()
{
  local float       CrosshairSize;
  local vector     HitLocation, HitNormal,
                   StartTrace, EndTrace,
                   ScreenPos;
  local actor      HitActor;
  local MyWeapon   W;
  local Pawn       MyPawnOwner;

  /** Sets the PawnOwner */
  MyPawnOwner = Pawn(PlayerOwner.ViewTarget);

  /** If we don't have an owner, then get out of the
      function */
  if ( MyPawnOwner == None )
  {
    return false;
  }

/** Sets the Weapon */
W = MyWeapon(MyPawnOwner.Weapon);
```

We start by defining our local variables. We set `MyPawnOwner` to be the pawn that the HUD is currently drawn for. Our weapon, `W`, is our pawn's currently equipped weapon. If we don't have a pawn for whatever reason, get out of the function. There is no point in wasting precious CPU cycles if there is no need for the function to get called.

4. The next step involves our trace. We need to set the values for our trace, then perform one.

```
/** If we have a weapon... */
if ( W != None)
{
  /** Values for the trace */
  StartTrace = W.InstantFireStartTrace();
  EndTrace = StartTrace + W.MaxRange() *
  vector(PlayerOwner.Rotation);
  HitActor = MyPawnOwner.Trace(HitLocation, HitNormal,
  EndTrace, StartTrace, true, vect(0,0,0),,
  TRACEFLAG_Bullet);
  DrawDebugLine(StartTrace, EndTrace, 100,100,100,);
```

A trace draws a direct line from one point to another. In this situation we have our trace call our weapon's `InstanteFireStartTrace()` function, which in turn calls `GetPhysicalFireStartLoc()`, which looks like the following code snippet:

```
/*****************************************************************
* Location that projectiles will spawn from. Works for secondary
* fire on third person mesh
*****************************************************************/
simulated function vector GetPhysicalFireStartLoc(optional vector
AimDir)
{
  Local SkeletalMeshComponent AttachedMesh;
  local vector SocketLocation;
  Local TutorialPawn TutPawn;

  TutPawn = TutorialPawn(Owner);
  AttachedMesh = TutPawn.CurrentWeaponAttachment.Mesh;
  AttachedMesh.GetSocketWorldLocationAndRotation
  (MuzzleFlashSocket, SocketLocation);

    return SocketLocation;
}
```

Quite simply, we're using our weapon's `SocketLocation` to start the trace. Our trace end uses the `StartTrace` value and adds our weapon's `MaxRange`, then multiplies it by our player controller's rotation. The `HitActor` value keeps track of anything our trace hits along the way.

We then draw a debug line so that we can see the trace. The line takes our start trace and end trace as parameters. You don't need this line, but it certainly makes your job far easier.

5. Let's go over to the next step, that is, converting 3D coordinates into a 2D space.

```
/** Projection for the crosshair to convert 3d coords
      into 2d */
ScreenPos = Canvas.Project(HitLocation);
/** If we haven't hit any actors... */
if (HitActor == None )
{
  HitActor = (HitActor == None) ? None
                                : Pawn(HitActor.Base);
  HitLocation = EndTrace;
  ScreenPos = Canvas.Project(HitLocation);
}
```

`ScreenPos`, or the 2D vector we want to draw the crosshair on, is using our `Canvas.Project` function and takes `HitLocation` as a parameter. `Project` is used when you want to take a 3D coordinate and draw it on a 2D space, like our HUD. The opposite function, `Deproject`, takes a 2D coordinate and converts it to 3D space.

Our `HitLocation` parameter changes depending on how we set that value. Our `HitLocation` parameter is the actor our trace has crossed. If we haven't hit any actors, `HitLocation` is where the trace ends (that is, it may just extend off into the distance on a map without walls).

6. Now we need to draw the actual crosshair.

```
/** Draws the crosshair for no one - Grey*/
CrosshairSize = 28 * (Canvas.ClipY / 768) *
                     (Canvas.ClipX /1024);
Canvas.SetDrawColor(100,100,128,255);

// Crosshair in center of trace
Canvas.SetPos(ScreenPos.X - (CrosshairSize * 0.5f),
              ScreenPos.Y -(CrosshairSize * 0.5f));
Canvas.DrawTile(class'UTHUD'.default.AltHudTexture,
CrosshairSize, CrosshairSize, 600, 262, 28, 27);
return false;
```

Our crosshair size uses the edge of our screen, both the `X` and `Y` values, then divides those values by the standard screen resolution (1020 x 768) and multiples it by 28 so that it is large enough to see. Our crosshair color can be anything we'd like, but I've set it so that it is gray for now.

We set the crosshair to be in the center of the trace by taking its screen position and subtracting half of the crosshair size. This allows it to use the center of the crosshair, otherwise we'd be using the top-left corner of the crosshair as our center point.

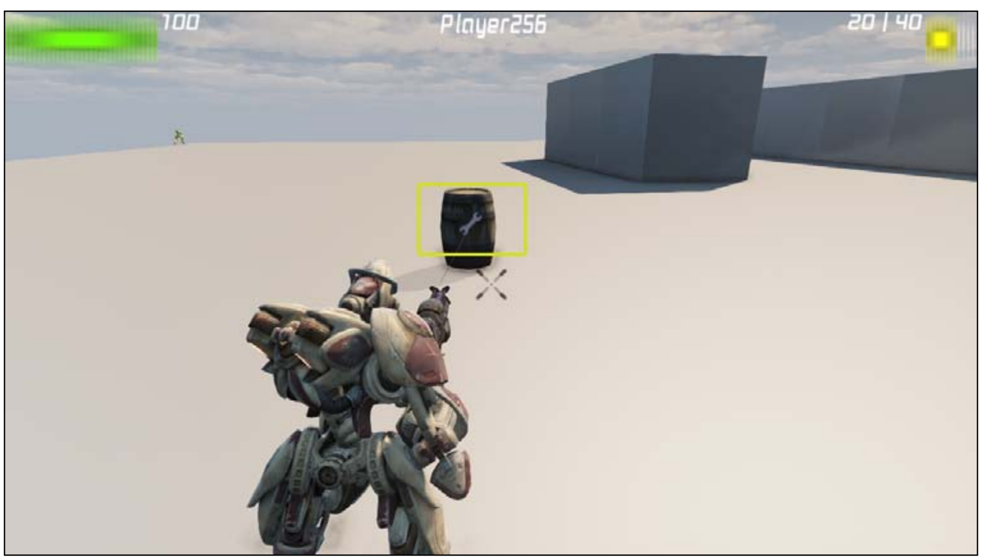
`DrawTile` takes our texture, `AltHudTexture`, and uses that as our crosshair image. For now it's just an image of a wrench.

7. Don't forget to add our `CheckCrossHairOnFriendly()` function to `DrawHUD` either. The function should now look like the following code snippet:

```
/******************************************************************
* Draws the HUD
******************************************************************/
function DrawHUD()
{
  ...
  CheckCrosshairOnFriendly();
  ...
}
```

We'll make the necessary changes in our next recipe so that the color adjusts over allies, but let's add the function now, which also stores the current info for our crosshair in this recipe.

8. Compile the project and take a look! We now have a crosshair that follows our weapon's trace! We still have that other crosshair drawn in the center of the screen, but that can obviously be removed by commenting out `DrawWeaponCrosshair()` from `DrawHUD()`.



How it works...

We need to set our base aim rotator to use our point of view. We do this by overriding the `GetBaseAimRotation()` function in our `TutorialPawn` class. It is called by our `GetAdjustedAimFor()` function in our `TutorialPlayerController` class. This gives our controller an opportunity to adjust the aiming of the pawn.

Afterwards, we set our base aim rotation. Here we are using our point of view rotation, that is, the rotation from our camera and not the pawn in this situation.

Next, we create the function `CheckCrosshairOnFriendly()` to draw the crosshair. We need to draw a trace first and have it check for actors in our way. If it hits an actor, we draw the crosshair there. If not, we draw the crosshair at the end of our trace, however long that may be.

This function also makes use of `Canvas.Project()`, which takes a 3D vector from our environment and converts it to a 2D vector, and that's what allows us to actually draw the crosshair on the HUD.

Changing the crosshair color when aiming at a pawn

Now that we have a more accurate representation of our crosshair working, why not take it to the next step and have it change colors to signify that we are pointing at a pawn?

In this next recipe, we'll add behavior to our crosshair that allows us to do just that.

Getting ready

Start by having your IDE open and ready to make some changes. We won't have to create any new classes, as we'll only have to make changes to an existing function in our `TestHUD` class.

How to do it...

This change requires us to add an `if` statement to our `CheckCrosshairOnFriendly()`.

1. Let's add it now as shown in the code snippet:

```
/** If our trace hits a pawn... */
if ((Pawn(HitActor) == None))
{
  /** Draws the crosshair for no one - Grey*/
```

```
  CrosshairSize = 28 * (Canvas.ClipY / 768) *
  (Canvas.ClipX /1024);
  Canvas.SetDrawColor(100,100,128,255);

  // Crosshair in center of trace
  Canvas.SetPos(ScreenPos.X - (CrosshairSize * 0.5f),
  ScreenPos.Y -(CrosshairSize * 0.5f));
  Canvas.DrawTile(class'UTHUD'.default.AltHudTexture,
  CrosshairSize, CrosshairSize, 600, 262, 28, 27);

  return false;
}
```

This is the same information we posted in the previous recipe, but we have an `if` statement checking if our `HitActor` parameter is equal to `None`. When drawing our trace from the weapon's socket, we check to see if we've run across any actors, as seen by the following bit of code:

```
/** If we haven't hit any actors... */
if (HitActor == None )
{
  HitActor = (HitActor == None) ? None
                                : Pawn(HitActor.Base);
  HitLocation = EndTrace;
   ScreenPos = Canvas.Project(HitLocation);
}
```

If we have hit an actor of type `Pawn`, as seen by our typecast `Pawn(HitActor.Base)`, then that is our `HitActor` parameter, otherwise our `HitActor` parameter is set to `None`. With that said, our function should make a bit more sense now. If we don't hit any pawns, the crosshair will be drawn gray.

2. But what if we do run across a pawn in our trace? Well let's add that functionality now, just beneath our `if ((Pawn(HitActor) == None))` statement for drawing the gray crosshair.

```
/** Draws the crosshair for friendlies - Yellow */
CrosshairSize = 28 * (Canvas.ClipY / 768) *
(Canvas.ClipX /1024);
Canvas.SetDrawColor(255,255,128,255);
Canvas.SetPos(ScreenPos.X - (CrosshairSize * 0.5f),
              ScreenPos.Y -(CrosshairSize * 0.5f));
Canvas.DrawTile(class'UTHUD'.default.AltHudTexture,
CrosshairSize, CrosshairSize, 600, 262, 28, 27);
return true;
```

3. Compile the project and see for yourself. If you run your crosshair over another pawn it will turn yellow!

How it works...

We add a simple `if` statement to check if the trace from our weapon has run across a pawn. If it has, then we draw a yellow crosshair. If not, we default to our gray crosshair.

Drawing a debug screen

UDK offers easy access to a plethora of debug options through the console commands. While this is useful, it can be tedious to constantly type in these commands. What if there was a more effective way to draw our debug options for individuals who may not be as savvy with programming, like a level designer?

As a programmer, one of your many roles may include supporting designers and creating tools. To make their world easier we'll be creating a debug menu that can be accessed with one key and allow access to a number of debug options. This is great in situations where you don't have a keyboard available, such as when you are demoing a project with a game pad. We'll bind them to keyboard keys for now, but understand that they can just as easily be done with the game pad.

Getting ready

Start by having your IDE open and ready to make some changes. We'll be creating a new class, as well as a new function in our HUD, and adding some game bindable actions in our `defaultInput.ini` file.

How to do it...

1. Start by creating a new class which will be our actual debug menu. Have it extend from `CheatManager`. You'll notice that we have this class within `PlayerController`. This allows our cheat manager (really, our debug menu) to capture all of our input commands. It is also a collection of executable functions capable of performing numerous commands.

   ```
   class DebugMenu extends CheatManager within
   PlayerController;
   ```

2. We also want to keep these organized, so we'll use a `struct` record type to do just that.

   ```
   struct DebugCommand
   {
       var string CommandName;
       var string Command;
   };

   struct SDebugCommandPage
   {
       var string PageName;
       var array<DebugCommand> PageCommands;
   };
   ```

3. We're going to create a number of commands, so let's add the variables for them as shown in the following code snippet:

   ```
   var array<SDebugCommandPage> CommandDebugPages;
   var int CurrentPage;
   var int CurrentIndex;
   var bool bShowDebugMenu;
   ```

> Be sure to add the structs *above* your variables! Otherwise, the compiler won't recognize the variables inside as they haven't been created yet.

4. We need a way to turn our debug menu on and off now, so let's add the following code snippet:

   ```
   /*****************************************************************
   * Toggles Debug Menu on and off
   *****************************************************************/
   exec function ToggleDebug()
   ```

```
{
    CurrentPage = -1;    // Starts on main page
    CurrentIndex = 0;
    bShowDebugMenu = !bShowDebugMenu;

    // Disables movement
    SetCinematicMode
    (bShowDebugMenu,false,false,true,true,true);
}
```

This starts with the main menu each time we pull up the debug screen and also prevents our pawn from accepting inputs like movement and firing.

5. With our main menu set, we now need a way to navigate through it. The following two functions will allow us to move to the previous or next item on our list:

```
/*****************************************************************
* Selects the next item on the list
*****************************************************************/
exec function NextItem()
{
  local int IndexMax;

  if (bShowDebugMenu)
  {
    if (CurrentPage != -1)
    {
      IndexMax =
      CommandDebugPages[CurrentPage].PageCommands.Length-1;
    }
    else
    {
      IndexMax = CommandDebugPages.Length-1;
    }
    CurrentIndex = Min(CurrentIndex +1, IndexMax);
  }
}

/*****************************************************************
* Selects the previous item on the list
*****************************************************************/
exec function PreviousItem()
{
  if (bShowDebugMenu)
  {
    CurrentIndex = Max(CurrentIndex -1, 0);
  }
}
```

We define the maximum number of pages available, as shown by `IndexMax`, and changing its value depending on how far we've scrolled in the list.

6. We're going to need a way to get in and out of the pages. Let's add the following code snippet for executing the chosen debug item:

```
/*****************************************************************
* Executes chosen debug item
*****************************************************************/
exec function DoDebugCommand()
{
    local DebugCommand command;
    if (bShowDebugMenu)
    {
      /** Leave menu & execute the chosen cmnd */
      if (CurrentPage != -1)
      {
        command = CommandDebugPages[CurrentPage].
        PageCommands[CurrentIndex];
        ToggleDebug();
        ConsoleCommand(command.Command);
      }
      else
      {
        /** Next page */
        CurrentPage = CurrentIndex;
        CurrentIndex = 0;
      }
    }
}
```

Add the following code snippet to come out of the selected page:

```
/*****************************************************************
* Back out of currently selected page
*****************************************************************/
exec function DebugBack()
{
    if (bShowDebugMenu)
    {
      if (CurrentPage != -1)// We're at the main menu
      {
        CurrentPage = -1;
        CurrentIndex = 0;
      }
```

```
        else
        {
          ToggleDebug(); // Can't go any further, so back out
        }
      }
}
```

The following code should be added to draw the debug menu:

```
/*******************************************************************
* Draws the debug menu
*******************************************************************/
function DrawDebugMenu(HUD H)
{
    local float XL, YL, YPos;
    local DebugCommand command;
    local SDebugCommandPage page;
    local int index_array;
    local Color cmnd_color;

    /** Draws the menu */
    if (bShowDebugMenu)
    {
      // Sets the font
      H.Canvas.Font = class'Engine'.Static.GetLargeFont();
      // Sets the length of the string (text)
      H.Canvas.StrLen("X", XL, YL);
      // Location on Y axis where text will begin (left)
      YPos = 0;

      // Top-left corner of the screen
      H.Canvas.SetPos(0,0);

      // Dark color
      H.Canvas.SetDrawColor(10,10,10,128);
      // Cover the size of the screen
      H.Canvas.DrawRect(H.Canvas.SizeX,H.Canvas.SizeY);
      if (CurrentPage == -1)
      {
        TutorialHUD(H).DrawDebugText("Debug
        Screen",vect2d(0,YPos),
        H.Canvas.Font,H.WhiteColor);
        YPos += YL;

       // Set the text color
```

```
        foreach CommandDebugPages(page,index_array)
        {
          // For the currently selected item in the array...
          if (index_array == CurrentIndex)
          {
            // set text color to red
            cmnd_color = H.RedColor;
          }
          else
          {
            // All other text is white
            cmnd_color = H.WhiteColor;
          }

/** Draws the text on screen based on the preceding info
    that we've provided */
         TutorialHUD(H).DrawDebugText
         (index_array$":"@page.PageName,vect2d(0,YPos),
         H.Canvas.Font,cmnd_color);
// Draws next line beneath current one
         YPos += YL;
        }
       }
       else
       {
         page = CommandDebugPages[CurrentPage];
         TutorialHUD(H).DrawDebugText("Debug Menu -
         "$page.PageName,vect2d(0,YPos),
         H.Canvas.Font,H.WhiteColor);

         // Draws next line beneath current one
         YPos += YL;
         foreach page.PageCommands(command,index_array)
         {
           if (index_array == CurrentIndex)
           {
             // Active text is red
             cmnd_color = H.RedColor;
           }
           else
           {
             // All other text is white
             cmnd_color = H.WhiteColor;
           }
```

```
            TutorialHUD(H).DrawDebugText(index_array$":
            "@command.CommandName,vect2d(0,YPos),
            H.Canvas.Font,cmnd_color);

            // Draws next line beneath current one
            YPos += YL;
          }
        }
      }
}
```

We start by setting our font to be drawn in the top-left corner, as indicated by the coordinates `0,0`. From there we use an `if-else` statement to make our currently highlighted text stand out by coloring it red and all other text white. We also draw a transparent gray rectangle across the entire screen.

How do we get one line to display beneath the other? Well the line `YPos += YL;` represents the `Y` position for our text, and we set it to be its current position plus the float `YL`.

We're also calling our HUD class and telling it to draw the `DrawDebugText()` function, which we'll cover shortly.

7. The final thing we need to do in this class is set the default properties of our variables:

```
DefaultProperties
{
    bShowDebugMenu=false
    CurrentPage=-1  // Starts us on the first page
    CurrentIndex=0

    /** Look in UDKInput.Ini to find additional debug
      commands to add to this list */
    CommandDebugPages(0)=(PageName="Debug
    Info",PageCommands[0]=(CommandName="Turn off Debug
    Info",Command="showdebug none"),
    PageCommands[1]=(CommandName="Toggle Camera Debug
    Info",Command="showdebug camera"),
    PageCommands[2]=(CommandName="Toggle Pawn Debug
    Info",Command="showdebug pawn"),
    PageCommands[3]=(CommandName="Toggle Pawn Weapon
    Info",Command="showdebug weapon"))

    CommandDebugPages(1)=(PageName="HUD",PageCommands[0]=
    (CommandName="Toggle HUD",Command="ToggleHUD"))
```

```
    CommandDebugPages(2)=(PageName="Collision |
    Pathfinding", PageCommands[0]=
    (CommandName=" Show Collision",Command="ShowDebug
    COLLISION"),PageCommands[1]=(CommandName=" Show
    Paths",Command="Show PATHS"))
}
```

It looks like a lot is thrown at you at once, so let's break it down carefully.

We've already seen `CurrentPage` and `CurrentIndex` used as local variables in the preceding functions, so we're aware of what they do.

`CommandDebugPages` is an array of our pages, or screens. We set the name of the page to be relevant to whatever we will fill the page with. For example, the third one is called `Collision | Pathfinding` as it holds all of our debug functions for those two categories.

Next, we add the command name as we want it displayed, for example, `Show Collision`. Following that we issue the actual command. These can be found in the `UDKInput.ini` file, so feel free to browse through those to find more.

8. Let's bind our inputs now, so that we can execute the functions we just created. Open up your `DefaultInput.ini` file, located at `UDK/DirectoryName/UDKGame/Config`.

```
;----------------------- ---------------------------------------
; CUSTOM BINDINGS FOR TUTORIALS
;----------------- ---------------------------------------------
;Bindings for Debug Menu
.Bindings=(Name="GBA_ToggleDebug" ,Command="ToggleDebug")
.Bindings=(Name="H" ,Command="GBA_ToggleDebug")
.Bindings=(Name="GBA_NextItem" ,Command="NextItem")
.Bindings=(Name="I" ,Command="GBA_NextItem")
.Bindings=(Name="GBA_PreviousItem" ,Command="PreviousItem")
.Bindings=(Name="U" ,Command="GBA_PreviousMenuItem")
.Bindings=(Name="GBA_DoDebugCommand" ,Command="DoDebugCommand")
.Bindings=(Name="O" ,Command="GBA_DoDebugCommand")
.Bindings=(Name="GBA_DebugBack" ,Command="DebugBack")
.Bindings=(Name="P" ,Command="GBA_DebugBack")
```

You could obviously bind the functions to any key of your choice, but I found that these work well for me. I use *H* to bring up my menu, and navigate with my *I*, *U*, *O*, and *P* keys.

Alternatively, you could just as easily bind these to the game pad.

9. We need to create the `DrawDebugText()` function in our `TestHUD` class which is called by `DrawDebugMenu()` in our `DebugScreen` class.

```
/****************************************************************
* Draws the text for the debug screen
****************************************************************/
 function DrawDebugText(string text, Vector2D position, Font
font,Color textColor)
{
    Canvas.SetDrawColorStruct(textColor);
    Canvas.SetPos(position.X,position.Y);
    Canvas.Font = font;
    Canvas.DrawText(text);
}
```

This draws the debug text and looks similar to the other canvas functions we used in *Chapter 7*, *HUD*.

10. Our `TutorialHUD` class's `PostRender()` event needs to call our `DebugMenu` as well, so let's do that now.

```
event PostRender()
{
  ....
  /** Draws debug HUD */
  DebugMenu(PlayerOwner.CheatManager).DrawDebugMenu(self);
  ....
}
```

11. The final step in this recipe has us assigning the `DebugMenu` class as the default cheat class in our `TutorialPlayerController` class. Add the following bit of code to the `defaultproperties` block:

```
defaultproperties
{
  ....
  CheatClass=class'DebugMenu' // Reference for DebugMenu
  ....
}
```

12. Compile the project and load your map. Hit the *H* key when it loads and take a look at your new debug menu!

How it works...

To make a debug menu we first needed to extend from UDK's `CheatManager`, which is a series of executable functions that allow us to alter the way the game works. We create our own cheat manager, which is really our debug screen, and allow it to combine various debug functions on one page.

Our functions are organized into a `struct` type, which holds the name and command of the debug function we're trying to call, such as `Show PATHS`. We then call a new function in our HUD that we created called `DrawDebugText`, which actually draws the text on the screen.

In the `defaultproperties` block of our `DebugMenu` class, we list the number of pages that we'll be using to organize our debug menu. Finally, we head to our `TutorialPlayerController` class and in the `defaultproperties` block we told it to use our `DebugMenu` class as the default cheat class, instead of the one previously defined.

> Be sure to go to `UDKGame.ini` and scroll down to the bottom; you will find your game HUD. Mine is listed as `[tutorial.TutorialHUD]`. Make sure that your `bShowHUD` value is set to `true`.

Drawing a bounding box around pawns

If you've ever played Eidos' excellent *Deux Ex* before, then you should be very familiar with bounding boxes around objects that your character is facing. We're going to replicate the same effect, but just for pawns. It's easy to alter and add functionality, however, so we really could draw a bounding box around nearly any object in the game.

This is great for highlighting objects that you want to point out to the player, or entice them to perform an action, such as pick up an object.

Getting ready

Start by having your IDE open and ready to make some changes. We won't have to create any new classes, as we'll only have to make changes to an existing function in our `TestHUD` class.

How to do it...

We need to use a trace to detect whether or not a pawn is standing in front of our character. Because invisible actors (that is, objects which don't use a sprite or something we can actually see on screen) may still block our path, we use `TraceActors` to determine what's in front of us.

1. This occurs in our `PostRender()` function within our `TutorialHUD` class. Let's add the variables we'll need right now.

```
  local Actor      HitActor;
  local Vector     HitLocation, HitNormal, EyeLocation;
  local Rotator    EyeRotation;
```

2. A check is necessary to verify that we have a player controller, and whether we grab the camera location and rotation or not. This will be used for our trace.

```
  /**Check for pawn owner*/
  if (PlayerOwner != None)
{
  /** Grab player camera loc & rot */
  PlayerOwner.GetPlayerViewPoint(EyeLocation, EyeRotation);
```

3. The pawn's `EyeLocation` is where we start the trace and extend from. We check to see if each actor is actually a pawn, and anything else is discarded, including our own pawn.

```
/* Trace to see where player is looking. Used to ignore
specific objects */
  ForEach TraceActors(class'Actor', HitActor, HitLocation,
  HitNormal, EyeLocation + Vector(EyeRotation) *
  PlayerOwner.InteractDistance, EyeLocation,
  Vect(1.f, 1.f, 1.f),, TRACEFLAG_Bullet)
  {
    /** If the hit actor is the player owner, player
    owner's pawn or if hit actor isn't visible, ignore
    it */
    if (HitActor == PlayerOwner ||
        HitActor == PlayerOwner.Pawn ||
        !FastTrace(HitActor.Location, EyeLocation))
    {
      continue;
    }
    /** Checks if the actor is a pawn */
    if (HitActor.IsA('Pawn'))
    {
      /** Draws the 2D brackets */
      RenderBoundingBox(HitActor);
    }
```

If a pawn is within our trace, we then call our `RenderBoundingBox()` function to draw a box around the pawn.

> Using the `if (HitActor.IsA())` statement, we can place our bounding box around any actor in UDK. Replace `'Pawn'` with any actor class of your choice to do this.

4. Now that our `RenderBoundingBox()` function is called, let's take a look at what it actually does.

```
/*****************************************************************
* Draws brackets around an actor, based on bounding box
* coordinates
*****************************************************************/
function RenderBoundingBox(Actor Actor)
{
  local Box    ActorBB;
  local int    ActualWidth, ActualHeight;

  /** If we don't have a canvas to draw  on or are
  targeting an actor, get out */
  if (Canvas == None || Actor == None)
  {
    return;
  }
```

We pass in our `Actor` parameter that we did a trace on. We have three parameters here: one `Box` variable, that is, `ActorBB`, which represents the bounding box around our actor, along with `ActualHeight` and `ActualWidth`, which measure the dimensions of the box.

As always, we need to have a check. This time we're verifying that we have a canvas to draw on in addition to an actor available to us. If either of those are not present, then we get out of the function.

5. We need a bounding box for our actor now, so we set our `ActorBB` variable to be set to the function `GetBB`, while passing in our `Actor` parameter. We'll get to know exactly how this function works shortly, but it's going to determine how far from the pawn the box should reside.

```
  /** Grabs the bounding box around our selected actor */
  ActorBB = GetBB(Actor);

  /** Math for the height and width */
  /** Change the float to adjust whether there are spaces
    between lines */
```

```
  ActualWidth = (ActorBB.Max.X - ActorBB.Min.X) * 0.4f;
  ActualHeight = (ActorBB.Max.Y - ActorBB.Min.Y) * 0.4f;

/** Draws colored brackets around anchor */
  Canvas.SetDrawColor(100, 255, 255);
```

Math needs to be performed to determine the size of the box around the pawn. We multiply by a `float` value to illustrate whether or not the box will be completely closed, or open on the top, bottom, and sides. We add the color of our bracket here too.

6. Let's draw the rectangle around the pawn now. We'll break it up into corners so that it's easy to understand the code.

```
/** Top Right */
  Canvas.SetPos(ActorBB.Max.X - ActualWidth,
  ActorBB.Min.Y);
  Canvas.DrawRect(ActualWidth, 10);
  Canvas.SetPos(ActorBB.Max.X , ActorBB.Min.Y);
  Canvas.DrawRect(2, ActualHeight);

  /** Top Left  */
  Canvas.SetPos(ActorBB.Min.X, ActorBB.Min.Y);
  Canvas.DrawRect(ActualWidth, 10);
  Canvas.SetPos(ActorBB.Min.X, ActorBB.Min.Y);
  Canvas.DrawRect(2, ActualHeight);

  /**  Bottom Right */
  Canvas.SetPos
  (ActorBB.Max.X - ActualWidth, ActorBB.Max.Y );
  Canvas.DrawRect(ActualWidth, 10);
  Canvas.SetPos
  (ActorBB.Max.X, ActorBB.Max.Y - ActualHeight);
  Canvas.DrawRect(2, ActualHeight );

  /** Bottom Left */
  Canvas.SetPos(ActorBB.Min.X, ActorBB.Max.Y);
  Canvas.DrawRect(ActualWidth, 10);
  Canvas.SetPos
  (ActorBB.Min.X, ActorBB.Max.Y - ActualHeight );
  Canvas.DrawRect(2, Actualheight);
}
```

The top-right corner sets our position to use the maximum X value from our ActorBB variable and subtract the actual width of the box, while the Y value is simply set to the minimum size of the actor's bounding box Y variable.

We draw our rectangle next. The first parameter is the X value and determines how thick or wide the bar appears on the sides. The second parameter, the Y variable, does the same for the top and bottom bar. The ActualWidth variable defines a very thin line, while a number such as 10 in the Y parameter creates a nice thick one to stand out.

7. Let's take a look at where we actually get our bounding box information from. This may look like quite a bit, but it's really just a lot of the same information repeated over and over. We'll create our GetBB() function, which grabs our bounding box dimensions from our actor.

```
/******************************************************************
* Grabs bounding box around an actor
******************************************************************/
function Box GetBB(Actor Actor)
{
  local Box       CompBBox, OutBox;
  local Vector    BoundingBoxCoords[8];
  local int       i;
```

```
/** grabs the bounding box for the specified actor */
Actor.GetComponentsBoundingBox(CompBBox);

/** X1, Y1 */
BoundingBoxCoords[0].X = CompBBox.Min.X;
BoundingBoxCoords[0].Y = CompBBox.Min.Y;
BoundingBoxCoords[0].Z = CompBBox.Min.Z;
BoundingBoxCoords[0] =
Canvas.Project(BoundingBoxCoords[0]);
/** X2, Y1 */
BoundingBoxCoords[1].X = CompBBox.Max.X;
BoundingBoxCoords[1].Y = CompBBox.Min.Y;
BoundingBoxCoords[1].Z = CompBBox.Min.Z;
BoundingBoxCoords[1] =
Canvas.Project(BoundingBoxCoords[1]);
/** X1, Y2 */
BoundingBoxCoords[2].X = CompBBox.Min.X;
BoundingBoxCoords[2].Y = CompBBox.Max.Y;
BoundingBoxCoords[2].Z = CompBBox.Min.Z;
BoundingBoxCoords[2] =
Canvas.Project(BoundingBoxCoords[2]);
/** X2, Y2 */
BoundingBoxCoords[3].X = CompBBox.Max.X;
BoundingBoxCoords[3].Y = CompBBox.Max.Y;
BoundingBoxCoords[3].Z = CompBBox.Min.Z;
BoundingBoxCoords[3] =
Canvas.Project(BoundingBoxCoords[3]);

/**X1, Y1 */
BoundingBoxCoords[4].X = CompBBox.Min.X;
BoundingBoxCoords[4].Y = CompBBox.Min.Y;
BoundingBoxCoords[4].Z = CompBBox.Max.Z;
BoundingBoxCoords[4] =
Canvas.Project(BoundingBoxCoords[4]);
/** X2, Y1 */
BoundingBoxCoords[5].X = CompBBox.Max.X;
BoundingBoxCoords[5].Y = CompBBox.Min.Y;
BoundingBoxCoords[5].Z = CompBBox.Max.Z;
BoundingBoxCoords[5] =
Canvas.Project(BoundingBoxCoords[5]);
/** X1, Y2 */
BoundingBoxCoords[6].X = CompBBox.Min.X;
BoundingBoxCoords[6].Y = CompBBox.Max.Y;
BoundingBoxCoords[6].Z = CompBBox.Max.Z;
```

```
BoundingBoxCoords[6] =
Canvas.Project(BoundingBoxCoords[6]);
/** X2, Y2 */
BoundingBoxCoords[7].X = CompBBox.Max.X;
BoundingBoxCoords[7].Y = CompBBox.Max.Y;
BoundingBoxCoords[7].Z = CompBBox.Max.Z;
oundingBoxCoords[7] =
Canvas.Project(BoundingBoxCoords[7]);
```

The `CompBBox` variable is a box that grabs the actor's bounding box, as definied at the beginning of the function with the line `Actor.GetComponentsBoundingBox (CompBBox);`.

The next step has us creating an array for each corner of our box. We need a vector for our rectangle in the previous function to draw at. We get this vector, `BoundingBoxCoordinates` by using the values of our actor's bounding box. The first array sets `BoundingBoxCoords.X` to be equal to the minimum `X` value for our pawn's bounding box, as found in the `Object` class. We do this for the `Y` and `Z` variables as well, before finally combining them all and using `Canvas.Project` to convert the 3D coordinates to 2D ones.

8. Now we have to locate the top, bottom, left, and right coordinates for the pawn's bounding box. Using the edge of the canvas we find the minimum values for our `X` and `Y` variables, and set the max values to be zero.

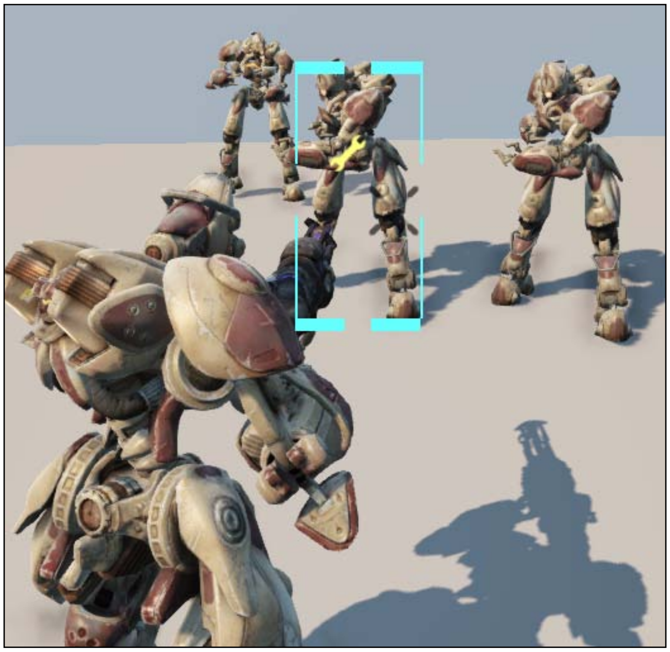
```
/* Locates left, top, right & bottom coords */
OutBox.Min.X = Canvas.ClipX;
OutBox.Min.Y = Canvas.ClipY;
OutBox.Max.X = 0;
OutBox.Max.Y = 0;
```

9. The final step has us iterating through our outside box coordinates to detect the smallest and largest coordinates. We do this by comparing the `X` and `Y` values of our pawn's bounding box. We return the value of `Outbox`, which is what we used in our `RenderBoundingBox()` function.

```
/** Iterate though bounding box coordinates */
for (i = 0; i < ArrayCount(BoundingBoxCoords); ++i)
{
  /** Detect the smallest X coords */
  if (OutBox.Min.X > BoundingBoxCoords[i].X)
  {
    OutBox.Min.X = BoundingBoxCoords[i].X;
  }

  /** Detect the smallest Y coords */
```

```
        if (OutBox.Min.Y > BoundingBoxCoords[i].Y)
        {
          OutBox.Min.Y = BoundingBoxCoords[i].Y;
        }

        /** Detect the largest X coords */
        if (OutBox.Max.X < BoundingBoxCoords[i].X)
        {
          OutBox.Max.X = BoundingBoxCoords[i].X;
        }

        /** Detect the largest Y coords */
        if (OutBox.Max.Y < BoundingBoxCoords[i].Y)
        {
          OutBox.Max.Y = BoundingBoxCoords[i].Y;
        }
      }
      return OutBox;
    }
```

10. That's all there is to it! Compile the project and point your pawn towards another pawn to see a bounding box drawn around it.



There's more...

We could easily adopt this to have a different appearance around specific actors. For example, we could have a thin white box drawn around any weapon. This would be done by using a `case` or `switch` statement in our `PostRender` function, where it checks whether `Actor` is of type `Weapon` or not. From there, we'd create a method similar to our `DrawBoundingBox` that could be called something like `DrawBoundingBoxWeap`. The properties inside would only have to be changed marginally to have a noticeable effect, such as having the box completely wrap around the actor, changing the color, or how far away the actor is resting.

How it works...

`PostRender` is an event called by the game during each frame, and `PostRender` calls `DrawHUD`, so really our call to `DrawBoundingBox` could occur in either function. Regardless, we start by making a trace from our pawn's `EyeLocaton`. A check is then performed to see if we run across an `Actor` object of type `Pawn`. If it is, then we call our `DrawBoundingBox` function.

This is responsible for the appearance of our box, from the color to whether or not the lines form one complete box around the pawn. Furthermore, it controls the thickness of each rectangle as well.

`DrawBoundingBox` grabs the coordinates for the box by calling the `GetBB` function which handles all of the heavy math for us. It grabs the bounding box from our pawn, and using that sets the vectors for us to draw our four rectangles around the pawn.

Index

Symbols

A

B

C

R

S

T

Thank you for buying

UnrealScript Game Programming Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

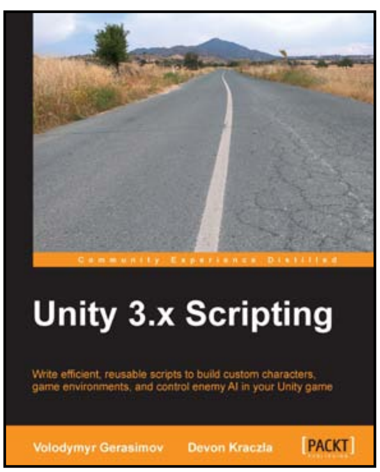
We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Unreal Development Kit Game Programming with UnrealScript: Beginner's Guide

ISBN: 978-1-84969-192-5 Paperback: 466 pages

Create games beyond your imagination with the Unreal Development Kit

1. Dive into game programming with UnrealScript by creating a working example game.
2. Learn how the Unreal Development Kit is organized and how to quickly set up your own projects.
3. Recognize and fix crashes and other errors that come up during a game's development.



Unity 3.x Scripting

ISBN: 978-1-84969-230-4 Paperback: 292 pages

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

1. Make your characters interact with buttons and program triggered action sequences
2. Create custom characters and code dynamic objects and players' interaction with them
3. Synchronize movement of character and environmental objects
4. Add and control animations to new and existing characters

Please check **www.PacktPub.com** for information on our titles