Community Experience Distilled

Getting Started with Unity

Learn how to use Unity by creating your very own "Outbreak" survival game while developing your essential skills

Patrick Felicia

[PACKT] PUBLISHING

Getting Started with Unity

Learn how to use Unity by creating your very own "Outbreak" survival game while developing your essential skills

Patrick Felicia

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Getting Started with Unity

Copyright © 2013 Packt Publishing

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

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

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

First published: August 2013

Production Reference: 1190813

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

ISBN 978-1-84969-584-8

`www.packtpub.com`

Cover Image by Suresh Mogre (`suresh.mogre.99@gmail.com`)

Credits

Author
Patrick Felicia

Reviewer
Marc Schaerer

Acquisition Editors
Saleem Ahmed
Erol Staveley

Commissioning Editor
Sruthi Kutty

Technical Editor
Pratik More

Project Coordinator
Deenar Satam

Proofreader
Lauren Harkins

Indexer
Priya Subramani

Graphics
Ronak Dhruv

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

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

I would like to thank the staff at Packt Publishing for their help throughout the writing of this book, as well as Marc Schaerer, the technical reviewer, for his valuable comments and feedback.

About the Reviewer

Marc Schaerer is an interactive media software engineer delivering interactive learning, training, and entertainment experiences on mobile, desktop, and web platforms for customers from all over the world through his company Gayasoft (`http://www.gayasoft.net`) located in Switzerland.

He makes use of Unity, which he has been using since the technologies 1.x days in 2007, and has been enhancing its capabilities through extensions where suitable.

Marc Schaerer has a strong background in the 3D graphics, network technology, software engineering, and interactive media fields; he started building up his knowledge in these fields right from his teenage years and later on solidified it with studies in Computational Science and Engineering at the Swiss Federal Institute of Technology, Zurich.

This knowledge found usage in Popper (`http://www.popper.org`), an interactive 3D behavioral research platform for Harvard, developed by Gayasoft and powered by Unity, Mathlab and ExitGames Photon.

With the rise of serious games, Marc is currently focusing his and his company's efforts to research options and technologies for the next generation of interactive and immersive experiences through AR and VR technologies (Metaio, OpenCV, Oculus Rift) and new forms of input (Razer Hydra, Leap Motion).

www.packtpub.com

Support files, eBooks, discount offers and more

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

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

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

http://packtLib.packtpub.com

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

Why Subscribe?

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

Free Access for Packt account holders

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

This book is dedicated to my beloved wife, Helena.

Table of Contents

Preface **1**

Chapter 1: Getting to Know Unity3D **7**

Introduction to game engines **8**
Why choose Unity **8**
What's new in Unity3D 4 **9**
Downloading Unity3D **9**
Launching Unity3D **10**
Unity3D's interface **11**
The Scene view 12
Navigating in the scene 12
The Hierarchy view 13
The Project view 14
The Inspector 14
The Console 15
Navigating through the AngryBots scene **15**
Creating a new project and scene **16**
Adding objects to a scene **17**
Creating a cube 18
Adding a texture to objects 22
Inserting imported objects 23
Implementing first- and third-person views **24**
Adding a first-person controller 25
Adding a third-person controller 27
Summary **29**

Chapter 2: Creating a Maze with Built-in Objects 31
Creating a maze based on built-in objects 31
Fine-tuning the level 38
Understanding colliders 39
Summary 42
Chapter 3: Using Scripts to Interact with Objects 43
Introduction to scripting in Unity3D 44
Importing necessary assets 44
Creating our first script 45
Collecting objects 52
Adding audio 55
Creating and displaying an inventory system 57
Finishing the game 64
Summary 65
Chapter 4: Creating and Tracking Objects 67
Displaying the health bar 69
Displaying a mini-map of the level 72
Creating a gun 79
Allowing for repeated shots 87
Summary 88
Chapter 5: Bringing Your Game to Life with AI and Animations 89
Importing and configuring the 3D character 90
Animating the character for the game 91
Creating parameters and transitions 94
Adding basic AI to enemies 97
Sending messages to alert other close enemies 101
Creating additional states 103
Using waypoints to define a path 110
Summary 113
Chapter 6: Finalizing and Optimizing Your Game 115
Improving the AI using breadcrumbing 116
Allowing enemies to throw and follow their own breadcrumbs 116
Allowing enemies to follow the player's breadcrumbs 121
Creating and updating prefabs 127
Keeping track of the number of lives 132
Animating the door that leads to the water section 133
Creating a menu system for your game 135
Keeping track of the number of lives 139
Optimizing the game 141

Exporting your game to the web **142**
Where to go from here **143**
Game designing 143
Artificial intelligence 143
3D characters 144
Creating your audio files 144
Learning more about Unity3D 144
Summary **145**
Index **147**

Preface

Getting Started with Unity will get you up to speed with Unity3D's core features, using an easy step-by-step approach. Throughout this book, you will progressively develop your skills to create a survival video game.

From Unity3D's interface to finite state machines with Mecanim, you will learn all of the necessary features to create a great game, including built-in objects manipulation, collision detection, textures, scripting, audio, particle effects, pathfinding, and raycasting.

You will create an indoor environment, where the player can collect objects (including a gun, ammunitions, or health packs), shoot at enemies, open doors based on some conditions, and much more.

You will include animated zombies with different levels of intelligence that follow and attack the player based on a finite state machine and some AI techniques (for example, breadcrumbing).

You will also learn how to create a menu system for your game, manage and display the health levels of the character, and keep track of these factors across levels.

What this book covers

Chapter 1, *Getting to know Unity3D*, is a general introduction to Unity3D. In this chapter, we will look at Unity3D's interface and how to include assets to scenes, using both built-in objects and external files, such as sounds and textures. We will also become familiar with the most commonly used components in Unity3D.

Chapter 2, *Creating a Maze with Built-in Objects*, illustrates how to create a simple, (yet challenging) indoor environment using Unity3D's primitives and standard assets.

Chapter 3, *Using Scripts to Interact with Objects*, explains how to use scripting in Unity3D to create a user interface, handle user interaction, and display customized messages on the screen.

Chapter 4, *Creating and Tracking Objects*, explains how to add more interaction to our game with special effects, GUI elements, and a mini-map. We will also look at advanced techniques to handle cameras and camera views.

Chapter 5, *Bringing Your Game to Life with AI and Animations*, teaches you how to bring the game to life by animating objects and characters, and by giving NPCs some levels of artificial intelligence to challenge the player. We will also learn how to set up and manage a finite state machine with Mecanim to manage these characters.

Chapter 6, *Finalizing and Optimizing Your Game*, will introduce you to a technique called breadcrumbing to improve the NPCs' intelligence and pathfinding. You will also learn how to create menus for the different stages of the game, and how to navigate through them.

What you need for this book

To complete the projects in this book, you only need Unity 4.x (or a more recent version) that you can download from `www.unity3d.com/download/`.

All instructions on how to download and install Unity3D are provided in the first chapter.

Who this book is for

This book is for game developers who would like to learn how to use Unity3D and become familiar with its core features. This book is also suitable for intermediate users who would like to improve their skills. No prior knowledge of Unity3D is required.

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: "Create a new Boolean parameter called `withinReach`."

A block of code is set as follows:

```
public var walking:boolean = false;
public var anim:Animator;
public var  currentBaseState:AnimatorStateInfo;
public var walkForwardState:int = Animator.StringToHash("Base
  Layer.WalkForward");
public var idleState:int = Animator.StringToHash("Base
  Layer.Idle");
private var playerTransform:Transform;
private var hit:RaycastHit;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
case:walkForwardState
  var zombies:GameObject [] = GameObject.FindGameObjectsWithTag("zomb
ie");
  for (var zombie:GameObject in zombies)
  {
    if (Vector3.Distance(transform.position, zombie.transform.
position) < 8.0f)
zombie.GetComponent(controlZombie).setWalking(true);
  }
  break;
```

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: "Click on the **Animations** tab, and then click on the label **attack**; this label will provide information on the attack clip."

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
Getting to Know Unity3D

This chapter is a general introduction to **Unity3D**, explaining the concepts of game engines and the general benefits that can be obtained from such a software. It presents some of the most interesting features of Unity3D, along with the novelties brought by Unity3D 4. In this chapter, we will look at Unity3D's interface, and how to include assets to scenes, using both built-in objects and primitives (for example, shapes, cameras, or lights) and external files such as 3D models, sounds, or textures. We will also become familiar with the most commonly used components in Unity3D. After completing this chapter we should be able to:

- Become familiar with Unity3D's interface and its different windows and menus
- Understand the different windows and layouts available in Unity3D
- Understand the main differences between the **Hierarchy**, **Project**, **Console**, and **Inspector** windows
- Navigate to the **Scene** view using shortcuts, create placeholders and duplicate objects in the **Hierarchy** window
- Monitor messages displayed in the console
- Understand and apply the concept of **GameObjects**
- Differentiate, choose between, and combine different GameObjects and components
- Apply transformations to objects (for example, moving, rotating, or scaling)
- Add textures to objects
- Add lights
- Add and use first- or third-person controllers

Introduction to game engines

Unity3D is a game engine and makes it possible for indie game developers, hobbyists, and those new to programming, to design and develop video games, focusing essentially on the game mechanics, rather than the underlying layers necessary to build a game. Game engines usually provide an Integrated Development Environment (**IDE**), where all activities and tasks related to game development are seamlessly integrated using coding, objects, and environment creation. They usually make it possible for designers to control the logic of their game using high-level programming or scripting languages, hence decreasing the learning curve and improving the workflow. With the evolution of technology, many game designers have used game engines. While game engines were initially essential for the production of video games, they are now used for a wide range of applications, with purposes other than gaming. For example, game engines are now employed for simulation, teaching, and training, as they often make it possible to create and manage very realistic environments easily. Moreover, such tools provide means for the creation of universally accessible environments, thanks notably to popular export formats for web and mobile devices.

Why choose Unity

Unity3D has been around for several years, with a growing number of users and several highly regarded video games produced till date. Many of these games are listed on the Unity3D website (`http://unity3d.com/gallery/made-with-unity/game-list`). Unity3D is built with simplicity and effectiveness in mind to allow both novice and advanced developers to maximize their game creation experience. It makes it possible to develop games of different genres (for example, platformers, role playing games, first-person shooters, massive multiplayer online role playing games, simulations, or strategy games), and for a comprehensive number of platforms (for example, Android, iOS, Windows Phone 8, PC, Mac, Linux, PS3, or XBOX 360). There are many game engines available out there, but Unity3D is one of the very few that provides a significant number of tools and techniques that simplify the development process, help to produce high-quality games, and addresses many aspects of game development, including an Integrated Development Environment (IDE), Artificial Intelligence (AI), animations, or lighting. As for other game engines, Unity3D makes it possible to code the game using relatively high-level programming and scripting languages, including JavaScript, Boo, or C#. While C# may require prior knowledge of Object Oriented Programming (OOP), JavaScript is an ideal scripting language for those with no or little background in programming. In this book, we will create our game using JavaScript.

In addition to its built-in capabilities, Unity3D offers the possibility to employ third-party plugins that greatly enhance the workflow and add some very interesting effects and functionalities. We will have the opportunity to discover some of these libraries throughout this book.

Finally, Unity3D includes a built-in access to the assets store (`https://www.assetstore.unity3d.com/`), an online store that provides material for our Unity projects (for example, textures, characters, GUI systems, or scripts). While the majority of these items have to be purchased, some of them can be imported in our project for free, so that we can create a game with a small budget. Most of these items can be integrated seamlessly in our game.

What's new in Unity3D 4

As I am writing this book, Unity3D is in its 4th version. The current version is the fruit of a rapid and consistent evolution. Although each version offers significant novelties and functionalities, the main components and layouts are rather similar across all versions, which means that what we will learn in this book should still be relevant for subsequent versions of Unity3D.

Unity3D 4 includes a number of very exciting features such as Mecanim, a new system to animate both objects and characters, enabling users to retarget animations, apply state machines and smooth transitions between these (blend trees), and Inverse Kinematics (IK).

Downloading Unity3D

We can download Unity3D from the Unity3D website (`http://unity3d.com/unity/download/`).

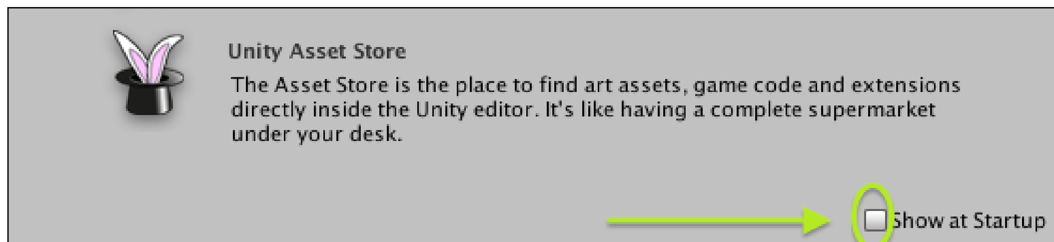
Before we download it, it is a good idea to check the requirements to make sure that our system (that is, software and hardware) is up-to-date. We can visit `http://unity3d.com/unity/system-requirements` to check whether our system complies with the requirements. Once we have checked the requirements, we can download the latest version of Unity3D for either Mac OS (`http://unity3d.com/unity/download/download-mac`) or Windows (`http://unity3d.com/unity/download/download-windows`). Opening either of these URLs will start the download of Unity3D automatically.

While the latest version of Unity3D is available on the official website, it is also possible to download previous versions of Unity3D from the following link: `http://unity3d.com/unity/download/archive`. If we use this book as a support for a course, it may be the case that the college or university where we study may have a previous version installed in the labs, and we may want to work on the project we have created during class from home. Another reason for this is that when we open a Unity3D project with a new version of the software, we may not be able to reopen it with the previous version.

Once we have downloaded Unity3D, launch the installer.

Launching Unity3D

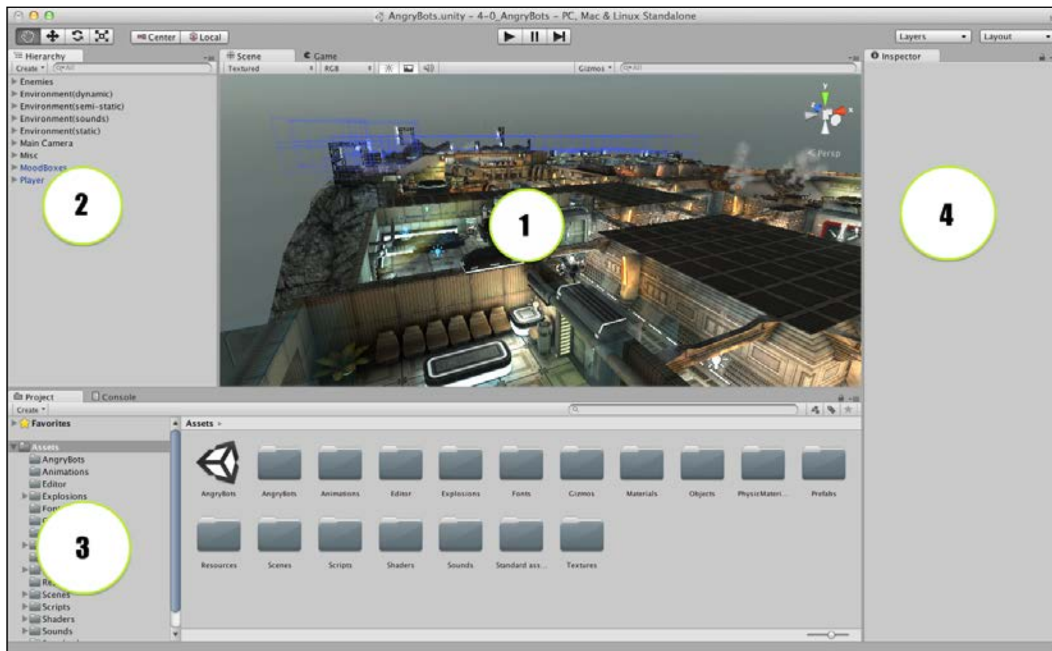
Once the installation is complete, we can launch Unity3D. The first time Unity3D is launched, we may be required to register the software and to provide an e-mail address, so that we can receive frequent updates on Unity3D. Once the registration process is complete, we can then start using Unity3D. The first time we open Unity3D, a pop-up window labeled **Welcome to Unity** will appear. This window can provide us with useful links to tutorials, the assets store, and additional help on Unity3D. However, if we don't want to display this window every time Unity3D is launched, we may uncheck the box located at the bottom right hand corner of the window, as we can see in the following screenshot, and close the window:



Unity3D's interface

By default, when we launch Unity3D for the first time, the project `AngryBots` should be open. The default layout is applied in Unity3D, and you will notice that the screen is divided into several sections or views (as highlighted in the following screenshot), including the following:

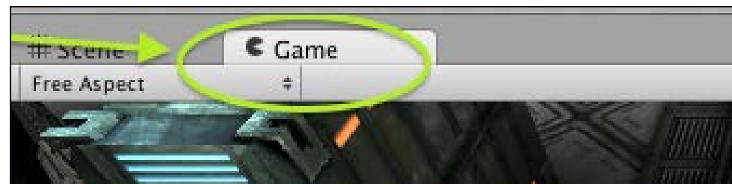
- The **Scene** view labeled as **1**, where we can visualize and modify the scene we have created for our game
- The **Hierarchy** view labeled as **2**, where we can see a list of all the objects included in our scene
- The **Project** view labeled as **3**, which contains all assets used in the current project
- The **Inspector** window labeled as **4**, which displays the properties of the object selected



The layout of this window can be modified using **Window | Layouts**. We can also resize the different windows and save our layout for a later use by navigating to **Windows | Layout | Save**.

The Scene view

The **Scene** view displays the scene and makes it possible to navigate through it. Scenes are comparable to game levels and include all relevant objects and environments. To the right of the tab **Scene**, we can find a tab labeled **Game**. This view displays the scene just the way it would appear when the game is launched (that is, from the active camera). For example, if we click on the **Game** tab, the view should switch to the camera used in the game to follow the player, as illustrated in the following image:



Navigating in the scene

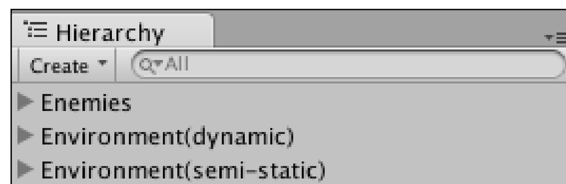
Unity3D provides several means and shortcuts to navigate through the scene, including the arrow keys from our keyboard. This mode is similar to the controls used in a First-Person Shooter (FPS), except that there is no **mouselook** option in this mode. Holding down the *Shift* key in this mode will increase our speed. To navigate in a mode that is closer to the controls usually found in FPS games, we can use the **Fly** mode, which is similar to the previous mode, except that it includes a mouselook feature. We can access this mode by holding down the mouse right button inside the scene. We can then navigate through the scene using the keys *W*, *A*, *S*, and *D*; float up and down using the keys *Q* and *E*, or look around by moving the mouse left, right, forward, or back. In addition to navigating through the scene, we can also focus on one object by double-clicking on this object in the **Hierarchy** window or by selecting this object in the hierarchy (by clicking on it once), and moving the mouse over the **Scene** view and pressing the *F* (focus) key (this will cause Unity3D to move the camera so that the object is displayed on the screen). We can zoom in and out by scrolling the mouse wheel; pan the view by clicking and dragging the mouse (note that panning the view works essentially when the **hand** tool is selected after pressing the *Q* key).

We can rotate the view by pressing the *Alt* key and clicking and dragging the mouse, or view the scene from the x, y, or z axes using the gizmo located in the top-right corner of the window, as illustrated in the following screenshot. Clicking on the **y** (green) arrow will display the view from the y-axis; the same applies to the **z** (blue), and **x** (red) axes. Clicking on the box in the middle of the gizmo will toggle the view between perspective and isometric modes.

There are many other shortcuts for Unity3D and we can find them in the documentation provided by Unity by selecting **Help | Unity Manual** from Unity3D and the pages dedicated to scene view navigation in Unity3D (`http://docs.unity3d.com/Documentation/Manual/SceneViewNavigation.html`). This being said, the shortcuts and controls described previously should be sufficient for us to start navigating through a Unity3D scene.

The Hierarchy view

This view lists all objects present in the scene. By default, a camera is present and the scene is viewed through its lens. Any subsequent object created or imported will appear in this view (for example, light, camera, or box) as illustrated in the following screenshot:



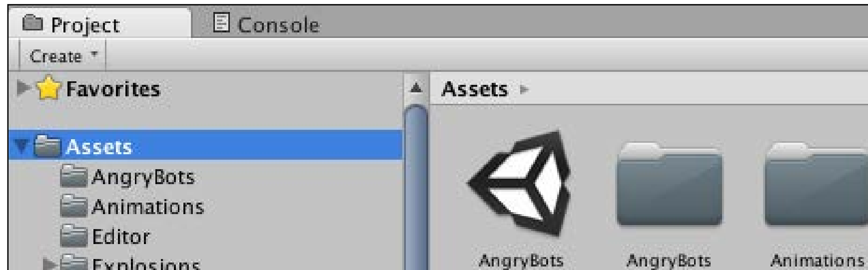
Note that objects can be duplicated in this window using the keys *Ctrl* + *D* (or right-click + duplicate). Groups can be created in this window; for example, when a group of objects need to share the same position or orientation (for example, transform properties), an empty object can be created, and these objects can then be added to the empty object. This way, any transformation applied to the empty object (container or parent) will be recursively applied to the children.

For example, in the `AngryBots` scene in the folder labeled `Environment (static)` we will notice a container labeled `barrels` that includes all barrels featured in the scene. If we apply a transformation to this container, this transformation will be applied to all objects included within this folder.

Throughout this book, Mac OS users need to use the *command* key or Apple key instead of the *Ctrl* key for keyboard shortcuts.

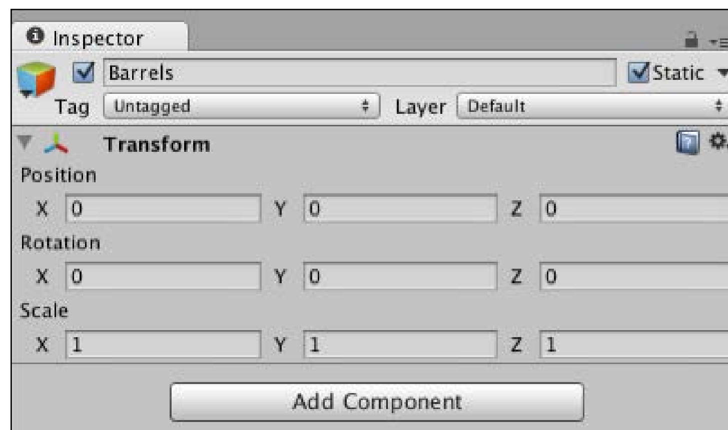
The Project view

This view features all assets used in the current project, including textures, audio, other scenes, prefabs (these are comparable to templates and will be explained later), fonts, or scripts (for example, JavaScript or C#). A project can include several scenes.



The Inspector

When an object is selected, its properties are displayed (and can be modified) in the Inspector window as illustrated in the following screenshot:



> Any modification applied to an object at runtime (when the scene is played), will not be saved. As a result, it is good practice to modify the properties of our objects before or after the scene is played, so that the changes are saved.

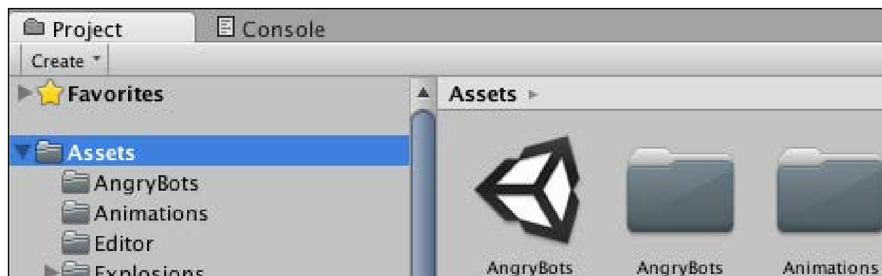
The Console

The Console window displays warnings, script errors, or user-generated messages for debugging purposes. We will look into this option later in this book. This window is accessible using **Window** | **Console** (or *Shift* + *Ctrl* + *C*).

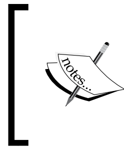
Navigating through the AngryBots scene

To become familiar with navigation controls and shortcuts, let's navigate through the `AngryBots` scene and do the following:

1. Open the `AngryBots` scene. If the `AngryBots` project is open, this scene should be located inside the `Assets` folder as illustrated in the following screenshot:



2. If, for some reason, you don't have the project `AngryBot`, you can download it from `http://unity3d.com/gallery/demos/demo-projects`.
3. Navigate through the `AngryBots` scene using the different navigation modes explained earlier. Select some of the items and look at their features using the Inspector. Look at the scene from the x, y, and z axes. Open the **Console** window and look for any message in this window. Play the scene using the play icon located at the top of the window (that is, black triangle pointing right) or by pressing *Ctrl* + *P*.
4. To exit the play mode, we can either click on the play icon or use the shortcut *Ctrl* + *P*.

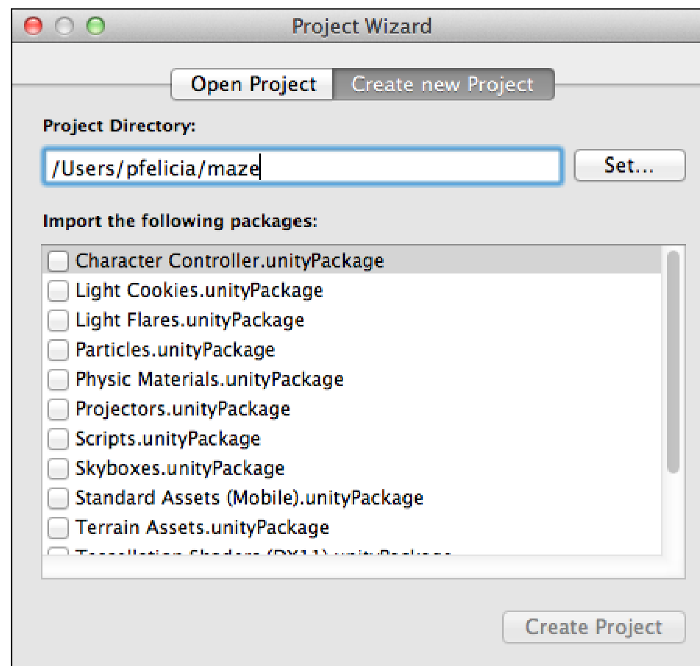


We can also play the scene in fullscreen mode by clicking on the option **Maximize On Play** available at the top-right corner of the game view. This mode makes it easier to test our game and to assess how it will look once it has been exported.

Creating a new project and scene

We will now create a new project for our first game, a 3D maze. Let's go through the following steps:

1. Select **File** | **New Project**. A window labeled **Project Wizard** should appear. This window provides a default directory for our project; however, we can set the location of our project by clicking on the **Set** button. By default, the name of the project is **New Unity Project**, but this can also be changed. At this stage, it is also possible to select and add the packages to include in our game. If we scroll down through the list below the label **Import the following packages**, we will see several packages, which consist of assets that can be used to enhance our game. These assets can be imported into our project at a later stage. Therefore, we will not select any of these for the time being.



2. Choose and set a directory and a name for our project (in the previous screenshot the project is called **maze** and is located in the directory `/Users/pfelicia/`).
3. Once you are happy with the project directory and its name, click on **Create Project**.
4. If you have made any changes to the `AngryBots` project, you may be asked whether you would like to save you changes. If this is the case, click on **Don't Save**.
5. After a few seconds, a new window should appear with our new project.

Adding objects to a scene

Now that we know how to navigate through the scene, we will learn how to create objects and add them to the scene. Unity3D makes it possible to add different types of built-in objects, including 3D primitives (for example, spheres, cubes, cylinders, or planes), lights (for example, point, directional, or area lights), Graphical User Interface (GUI) elements (for example, text or textures), or cameras. Each object can be added by selecting: **Game Object | Create Other**. Once an object has been created, we can change its properties using the Inspector or by directly moving, rotating, or scaling this object in the scene view. Once an object is selected, we can use the buttons located at the top-left corner of the scene view to apply transformations, as illustrated in the following screenshot:

The three buttons illustrated in the previous screenshot can be used to move, rotate, or scale the selected object, and can be accessed using the keys *W*, *E*, and *R* respectively. For example, if we select an object and click on the first button (or press the *W* key), three axes will appear on the object: a green axis (y), a blue axis (z), and a red axis (x). Dragging any of these axes will move the object in the corresponding direction. The same applies to rotating and scaling an object. Note that we can also constrain the object to a particular plan using the same technique but by dragging one of the colored plans that appear at the center of the object (for example, dragging the green box will move the objects in the horizontal x-z plane). For more information on the Unity3D interface, we can visit: `http://docs.unity3d.com/Documentation/Manual/LearningtheInterface.html`.

Creating a cube

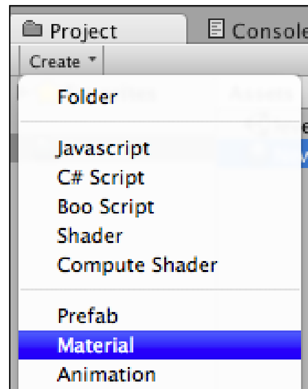
We will now create and texture a cube using Unity3D's built-in objects.

First, let's add a cube to the scene:

1. Select: **Game Object** | **Create Other** | **Cube**.
2. In the **Hierarchy** window, change the name of this cube from **Cube** to **box1** (right-click on the object then select **Rename** from the contextual menu, or select the object in the hierarchy, left-click on the object, and type the new name). We can also rename an object by selecting this object in the hierarchy and by pressing *Enter* for Mac OS or *F2* for Windows.
3. Make sure that the object is selected by clicking on it in the **Hierarchy** or in the **Scene** view.
4. In the **Inspector** window, change the x, y, and z position parameters of this object to (x=0, y=0, z=0).
5. Double-click on this object in the hierarchy to focus the camera on it.

Add a color to this box:

1. Our box looks pretty dull, and it would be great to add a color to it. This can be achieved by creating and applying a material (for example, color or texture) to the box.
2. From the **Project** window, select **Create** | **Material** as shown in the following screenshot:

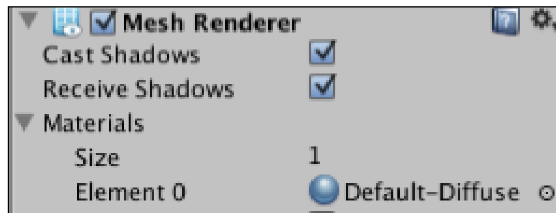


3. This will create a new material labeled **New Material** in the **Assets** window.
4. Change the name of this new material to **red** (right-click on the object and select **Rename** from the contextual menu, or left-click on the object).

5. Make sure that this material is selected, and look at its properties in the **Inspector** window.
6. One of the properties of this object is **Main Color**. We will modify this property by clicking on the white rectangle to the right of the label **Main Color**.
7. This should open a window labeled **Color**. This window makes it possible to pick a color for this material as shown in the following screenshot:

8. Pick a red color (for example, click on a red area) and close the **Color** window. We should see that the **Preview** of the material now shows a red sphere as shown in the following screenshot:

9. Apply the material to the box; this can be done in at least two ways.
    - The first way in which we can do is as follows:
        1. Make sure that we are in the **Scene** view.
        2. Select the box (**box1**) and drag-and-drop the material from the **Assets** window to the box created.
    - The second way is as follows:
        1. Click on the object **box1**.
        2. Look at the **Inspector** window and click on the **Materials** attribute of the **Mesh Renderer** component for this object as illustrated in the following screenshot:



        3. Click on the circle to the right of the label **Default-Diffuse** so that we can change the material. A new window will appear.
        4. From the new window, click on the tab labeled **Assets** and type the text `red` in the search field located at the top of this window as shown in the following screenshot:

        5. This should return one result, which is the material we have just created. Click on the material **red** and close the window. The cube should turn to red.

It is also possible to apply a texture to an object by dragging-and-dropping the texture on the object.

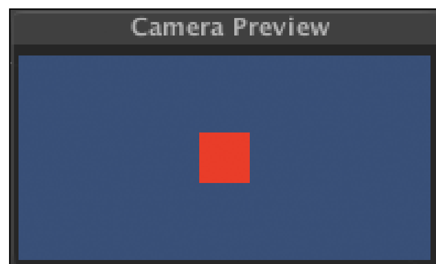
Add a light of our choice to the scene as shown in the following steps:

1. Select: **Game Object | Create Other | Directional Light**.
2. In the **Hierarchy** view, change the name of this light from **Directional Light** to **light1**.
3. In the **Inspector** window, change the position of this light to (x=0, y=4, z=0).
4. In the **Inspector** window, change the rotation parameters of this object to (x=90, y=0, z=0). This will rotate the light around the x axis, so that it points downwards (that is, toward the cube).

Any of these properties (for example, position or rotation) can be amended in the **Inspector** window by either entering the value in the text field, or by dragging the parameter we need to modify. For example, to change the x value of the position, we could position our mouse over the x label (the cursor will then turn into a double arrow) in the **Inspector** window and then drag-and-drop it to the right (to increase the value) or to the left (to decrease the value).

Move the camera so that the objects can be seen from the camera:

1. The camera is already present in the **Scene** and is labeled as **Main Camera** by default.
2. Select this camera (from the **Hierarchy** or **Scene** view).
3. In the **Inspector** window, change the position of this camera to (x=0, y=4, z=0).
4. In the **Inspector** window, change the rotation of this camera to (x=90, y=0, z=0). This will, as it was done for the light, rotate the camera around the x axis, so that it points downwards (that is, toward the cube). We can check the camera by looking at the camera view (that is, the rectangle located at the bottom-right corner of the **Scene** view), or by clicking on the **Game** tab (to display the game view).



Adding a texture to objects

So far, we have learned how to include objects and apply colors; however, for more realism, it would be better to use textures instead. Thankfully, adding textures is relatively easy in Unity3D. Before we can add a texture, we need to identify and acquire a texture. For many game programmers who need to create a prototype quickly and who prefer to focus on the mechanics of the game, it is often more convenient to use free online assets for games. For our first level, we will be using one of these resources:

When looking for free assets and textures, you can visit: `http://wiki.unity3d.com/index.php?title=Free_Game_Content`.

This wiki includes links to textures, models, music, sound effects, and fonts.

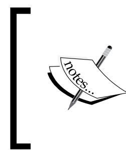
For our textures, we will use the site `www.cgtextures.com` as follows:

1. Launch your web browser and open the page `http://cgtextures.com`.
2. In order to use textures from this website, we need to create an account and register. If you already have an account on `www.cgtextures.com`, just log in (**Members | Login**); otherwise, you can create an account (**Members | Free Account**). The registration process should be relatively short. Once it is complete, use your new user name and password to log in (**Members | Login**).
3. On the left-hand side of the window, we can enter keywords to look for particular textures.
4. In the **Search** field, enter `plywood new 36438`. The site should return one match for the search.
    1. Click on the image returned from the search (in the right frame). A new window should now appear, displaying eight different textures.
    2. Click on the first texture (**image1: 640x640**); this will download the texture to our computer (`PlywoodNew0046_1_S.jpg`).

Import this image to your project as follows:

1. Switch to Unity3D.
2. Select the folder labeled `Assets` in the **Project** view (click on it once).
3. Create a new folder within this folder (from the **Project** window, select **Create | Folder**).

4. Rename the new folder `chapter1`.
5. Select this folder (that is, click on this folder once).
6. Select: **Assets** | **Import New Asset**.
7. Browse to the location where the texture was saved on our computer (for example, the `Download` folder).
8. Select the texture (`PlywoodNew0046_1_S.jpg`) and click on **Import**.
9. An asset labeled with the name of the texture (PlywoodNew0046_1_S) should now appear in the folder `chapter1`.
10. In the **Hierarchy** view, duplicate the game object labeled `box1`.
11. Call this new object `box2`.
12. Change this object's position to (x=4, y=0, z=0).
13. Drag-and-drop the plywood texture that we have just imported to the object `box2` either in the **Scene** view or in the **Hierarchy** view.
14. The object labeled `box2` should now feature a wooden texture.



> It is also possible to import assets in a project by simply dragging-and-dropping the assets (or folders) from the explorer (or finder) to the Unity3D **Project** window. It may be more efficient when importing folders with many assets.

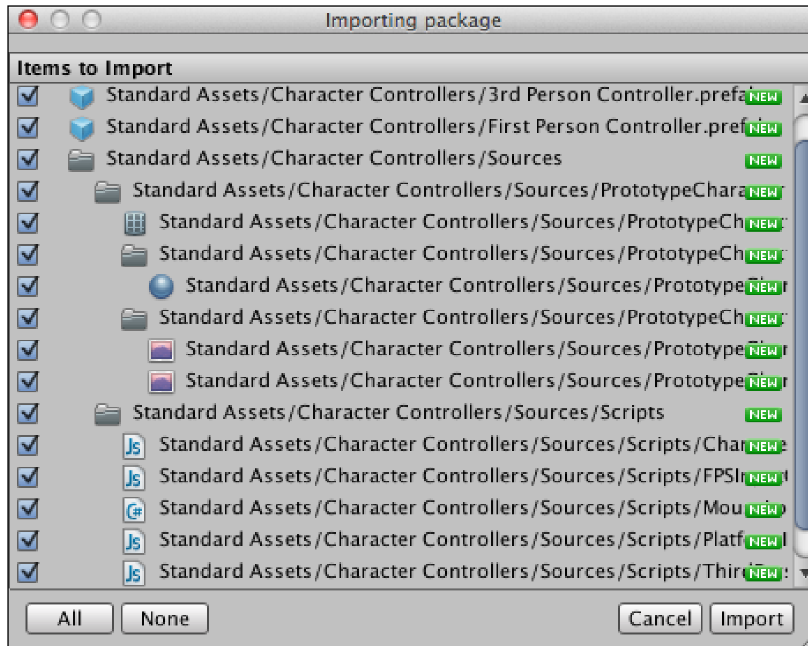
Inserting imported objects

Unity3D also makes it possible to import 3D objects from many different formats, including `.fbx`, `.dae`, `.2ds`, `.dxf`, and `.obj`. We can import objects by selecting **Assets** | **Import New Asset**. Both static and animated objects can be imported and animated (when applicable). For conversion to/from these formats, we can visit the Unity3D page dedicated to 3D object imports: `http://docs.unity3d.com/Documentation/Manual/HOWTO-importObject.html`.

Implementing first- and third-person views

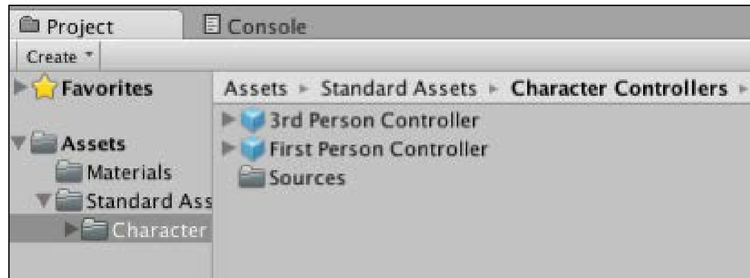
In some cases, we may need to navigate through the game using a first- or third-person view. This requires using a camera and the ability to move it based on the players' keyboard entries. Thankfully, Unity3D includes built-in objects to implement both types of navigation. If we have chosen to import these assets when creating our project (see previous sections), these assets are located in the **Project** view, inside the folder `Standard Assets`, and are named **3rd Person Controller** and **First Person Controller**. If you haven't done so yet, we will need to import the character controller package as follows:

1. Select: **Assets** | **Import Package** | **Character Controller**.
2. A new window labeled **Importing package** will appear.



3. Click on the button labeled **Import**.

4. This will import the character controllers in our project as illustrated in the following screenshot:



Once we have imported this package, both controllers appear with a blue box to their left. This indicates that they are prefabs. Prefabs are comparable to templates, and make it possible to reproduce similar objects based on the same template, without the need to recreate them. Once a template is created and instantiated, if the prefab is modified, all instances will also be modified accordingly, thus saving time of the game developer. We will look at the concept of prefabs in the next chapters.

If we click on any of these prefabs, we will see in the **Inspector** window that they include a set of components and attributes such as gravity, walk speed, or run speed, which can be modified and affect the behavior of the controller accordingly.

Adding a first-person controller

Before we add our first-person controller to the scene, we will create an object that will act as the ground on which the player will be able to walk or run.

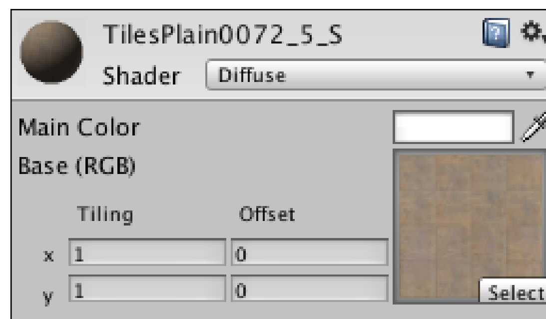
1. Create a new box (**Game Object** | **Create Other** | **Cube**).
2. Rename this box `floor`.
3. Scale-up this cube by 20 on the x and z axes:
    1. Select the cube labeled `floor`.
    2. In the **Inspector** window, change the position of this object to (x=0, y=-1, z=0) and its scale properties to (x=20, y=1, z=20). This will scale the cube on the x and z axes. Apply a texture to this cube.
4. Open the site `www.cgtextures.com`.
5. Search a tile texture using the keywords `tile 64722`.
6. Click on the image returned from the search. A new window should now appear, displaying three different textures.

7. Click on the first texture (**Image1: 700x700**); this will download the texture to your hard drive (`TilesPlain0072_5_S.jpg`).
8. In Unity3D, select the folder `chapter1` in the **Project** window, so that the image can be imported in this folder.
9. Import this texture as explained in the previous section (for example, **Assets | Import new Asset**).
10. Apply the texture to the object labeled `floor`.
11. Click on the object labeled `floor` and look at its properties in the **Inspector** window.

By importing these two images, Unity3D has automatically created the corresponding textures that will be used in the project. These textures are located in a folder called `material`, which is within the folder `chapter1` that was created previously.

Let's change the tiling properties of the texture used for the floor:

1. Click once on the object `floor` in the **Hierarchy** window.
2. In the **Inspector** window, click once on the component labeled **TilesPlain0072_5_S**, this should display more properties for this texture as illustrated in the following screenshot:



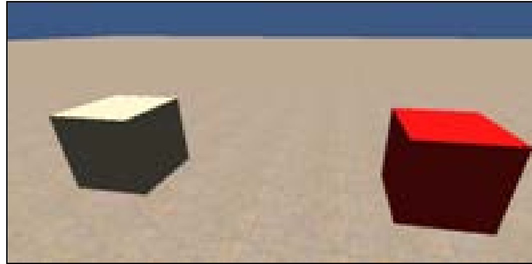
3. Change the tiling properties to (x=20, y=20).

Add First Person Controller to the scene:

1. Drag-and-drop the **First Person Controller** prefab (that is, blue box labeled **First Person Controller**) by selecting **Standard Assets | Character Controllers** onto the **Scene** view (or the **Project** view).
2. Move this **First Person Controller** to the position (x=0, y=0.6, z=-5) using the position properties in the **Inspector** window. This should place **First Person Controller** slightly above the ground; if it is too low, the collision may not be detected and the controller will fall indefinitely.

Test the scene:

1. Press the **Play** button located at the top of the window (or *Ctrl* + *P*).
2. Navigate through our scene using the keys *W*, *A*, *S*, and *D*, the arrow keys, or the mouse.



3. Quit the Play mode by clicking on the **Play** button or *Ctrl* + *P*.

Adding a third-person controller

In this section, we will add a third-person controller to the scene.

First, let's deactivate the third-person controller:

1. The scene already includes a **First Person Controller**. To avoid any conflict between these, it is better to deactivate the **First Person Controller**.
2. Select **First Person Controller** in our scene, and uncheck the box to the right of the label **First Person Controller** in the **Inspector** window as highlighted on the following screenshot:

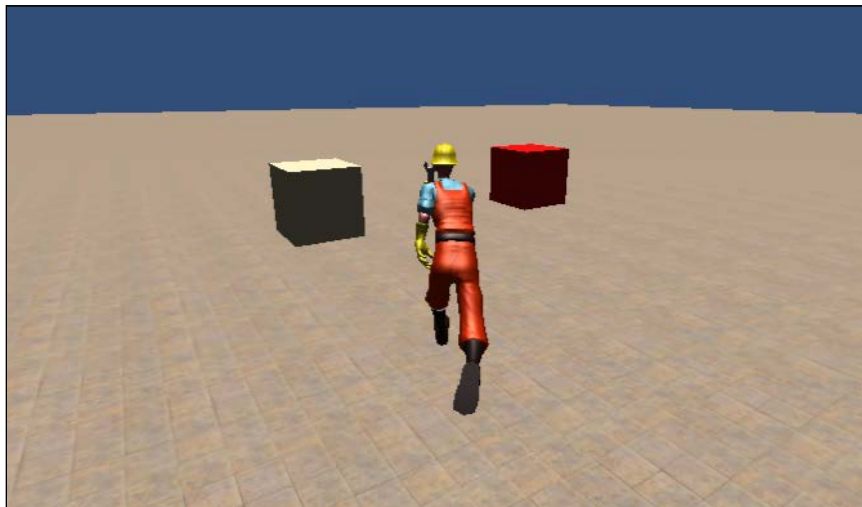
3. This will deactivate this object, which means that it will still be present in the project, but not used or seen when the scene is played until it is reactivated.

Add the third-person controller:

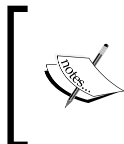
1. Open the folder `Character Controllers`, which is located in the folder `Standard Assets`. This folder should include two prefabs symbolized by a blue cube (**3rd Person Controller** and **First Person Controller**) and one folder (`sources`).
2. Drag-and-drop **3rd Person Controller** onto the **Scene** view (or the **Project** view).
3. Move this **3rd Person Controller** to the position (x=0, y=0.6, z=-5). This should place the controller slightly above the ground.

Test the scene:

1. Press the Play button located at the top of the window or (*Ctrl* + *P*).
2. Navigate through our scene using the keys *W*, *A*, *S*, *D*, or the arrow keys as illustrated in the following screenshot:



3. Quit the Play mode by clicking on the **Play** button or use the corresponding shortcut (*Ctrl* + *P*).
4. Save our scene (**File** | **Save Scene As**) and name it `chapter1`.



To use and develop the skills acquired in this chapter, we could create a simple platform game, using a third-person view, where the player has to jump from platform to platform to reach the end of the level.

Summary

In this chapter, you have learned about Unity3D. We have looked at the user interface employed in Unity3D, the different windows available, as well as useful shortcuts. Then, we have learned how to create and manipulate built-in objects, using both the inspector and contextual commands and menus. We have also learned how to apply colors and textures. Building on these skills, we then added both first- and third-person controllers for the player to navigate through the scene.

In the next chapter, we will combine the skills that we have acquired so far to create an indoor environment for our game including walls, doors, and water.

2

Creating a Maze with Built-in Objects

This chapter will illustrate how to create a simple indoor environment using Unity3D's primitives and standard assets. After completing this chapter, you should be able to:

- Transform basic shapes to create an indoor environment
- Understand how to modify the render settings for a scene
- Add and configure point lights in a scene
- Understand and use colliders for our game
- Add built-in water objects

Creating a maze based on built-in objects

Based on the skills we have covered in the previous chapter, let's create a maze for our game. The layout of our maze is described in the following diagram. The level will consist of two main rooms connected by a bridge. The different sections include the following:

- A platform from where the player will start
- Water surrounding the platform
- A maze where the player can find ammunition and eliminate all enemies

- An exit door for which the player will need to find the keys

This simple layout can be achieved by moving, scaling, and duplicating basic shapes. Before we start creating the maze, we will create a new scene and folder for this chapter as follows:

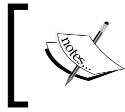
1. Open our previous project (`maze`), created in the previous chapter, if it is not open yet.
2. In the **Project** window, create a new scene (**File | New Scene**) and save it as `chapter2` (**File | Save Scene** as). In the **Project** window, click once on the `Assets` folder.
3. Select: **Create | Folder** from the **Project** window. Rename this folder `chapter2`.

Then, we will download all the textures required for the maze. We will need four textures for the walls, floor, ceiling, and doors. As per previous sections, we will download these textures from the website `www.cgtextures.com`, import them into our project and rename them to simplify their use. While some textures are specified in the next section, we can choose other textures of our choice.

1. Open the URL `http://cgtextures.com`.
2. If you are not already logged in, log in with your previous details by selecting **Members | Login**.
3. Once you have successfully logged in, we will search for our textures.
4. Add the keyword `MetalFloorsOthers0006` to the **Search** field located on the left-hand side of the window, and click on **Search**.
5. Doing so should return one result. Click on the image and download the first texture (that is, tiled) in the next window (**Image 1; 640x640**) to download it.
6. Once the file has been downloaded to your computer, (`MetalFloorsOther0006_1_S.jpg`), rename it `texture_floor.jpg`.
7. Repeat steps 4-6 using the following keywords, settings, and texture names:
    - **Second texture (walls)**: Use the keyword `concretebare0280`, choose image1 (`ConcreteBare0280_39_S`), and rename the texture `texture_walls`
    - **Third texture (ceiling)**: Use the keyword `ConcreteBare0314`, choose image1 and rename the texture `texture_ceiling`
    - **Fourth texture (door)**: Use the keyword `DoorsMetalBig0183`, and rename the texture `texture_door`

Next, we need to import these images inside Unity3D, so that they can be applied to objects:

1. Switch to Unity3D.
2. Select the folder `chapter2` that we created earlier (located in the `Assets` folder).
3. Select: **Assets | Import New Asset**.
4. Browse to the folder where we downloaded the textures.
5. Select the texture labeled `texture_floor` and click on Import.
6. Repeat steps 3-5 for the textures `texture_walls`, `texture_door`, and `texture_ceiling`.



The previous steps could also have been achieved by downloading the textures to the computer, and then renaming and dragging-and-dropping these textures in the folder `chapter2` inside Unity3D.

We can create the floor for our maze as shown in the following steps:

1. Create a new cube and rename it `floor`.
2. Change its position to (x=0, y=-1, z=0) and its scale to (x=49, y=1, z=49).
3. Apply the texture `texture_floor` to this object (drag-and-drop the texture on the object).
4. Change the tiling property of the texture to (x=20, y =20) in the **Inspector** window.

Then, we can create the blocks to be included in section **3** from the previous screenshot:

1. Create a new cube and rename it `block`.
2. Change the position of this cube to (x=-14, y=1, z=16) and its scale to (x=10, y=4, z=10).
3. Apply the texture `texture_walls` to this object.
4. Change the tiling property of the texture to (x=10, y=1).
5. Duplicate this object seven times (*Ctrl* + *D*) to the following locations:
    1. (x=0, y=1, z=16)
    2. (x=14, y=1, z=16)
    3. (x=-14, y=1, z=0)
    4. (x=14, y=1, z=0)
    5. (x=-14, y=1, z=-16)
    6. (x=0, y=1, z=-16)
    7. (x=14, y=1, z=-16)
6. We should now have eight blocks, arranged in a square.

Create the rocks that will be used as a bridge to access the area labeled as `1` on the previous screenshot:

1. Create a new cube and rename it `bridge`.
2. Apply the texture `texture_floor` to this object.
3. Change the tiling properties of this texture to (x=10, y=10).

4. Change the position of this cube to (x=-13, y=-1, z=50) and its scale to (x=2, y=1, z=2).
5. Duplicate this cube 10 times to the following locations:
    1. (x=-10, y=-1, z=50)
    2. (x=-7, y=-1, z=50)
    3. (x=7, y=-1, z=50)
    4. (x=10, y=-1, z=50)
    5. (x=13, y=-1, z=50)
    6. (x=0, y=-1, z=42)
    7. (x=0, y=-1, z=38)
    8. (x=0, y=-1, z=34)
    9. (x=0, y=-1, z=30)
    10. (x=0, y=-1, z=26)
6. Duplicate the last object labeled `bridge` and rename it `platform`.
7. Modify the position of this object (`platform`) to (x=0, y=-1, z=49) and its scale to (x=9, y=1, z=9).

Create horizontal walls as shown in the following steps:

1. Create a new cube and rename it `h_wall_water_long`.
2. Change its position to (x=0, y=0, z=57), its rotation to (x=0, y=90, z=0), and its scale to (x=2, y=6, z=50).
3. Apply the texture `texture_walls` to this object.
4. Duplicate this object and rename the duplicate `h_wall_water_short_left`.
5. Change the position of this new object to (x=-13, y=0, z=24), its scale to (x=2, y =6, z=24), and keep the rotation attributes (x=0, y=90, z=0).
6. Duplicate this object, and rename the copy `h_wall_water_short_right`.
7. Change the position of this new object to (x=13, y=0, z=24) and its scale to (x=2, y =6, z=24).
8. Duplicate this object and rename the duplicate `h_wall_short_right`.
9. Change the position of this new object to (x=13, y=1, z=-24) and its scale to (x=2, y =4, z=24).
10. Duplicate this object and rename the duplicate `h_wall_short_left`.
11. Change the position of this new object to (x=-13, y=1, z=-24) and its scale to (x=2, y =4, z=24).

Create vertical walls as shown in the following steps:

1. Create a new cube and rename it `v_wall_left`.
2. Apply the texture `texture_walls` to this object.
3. Change the position of this object to (x=-24, y=1, z=0) and its scale to (x=2, y=4, z=50).
4. Duplicate this object and rename the duplicate `v_wall_right`.
5. Change the position of this object (`v_wall-right`) to (x=24, y=1, z=0) and its scale to (x=2, y=4, z=50).
6. Duplicate this object and rename the duplicate `v_wall_water_right`.
7. Change the position of the object `v_wall_water_right` to (x=24, y=0, z=40) and its scale to (x=2, y=6, z=32).
8. Duplicate this object and rename the duplicate `v_wall_water_left`.
9. Change the position of the object `v_wall_water_left` to (x=-24, y=0, z=40) and its scale to (x=2, y=6, z=32).

Create doors as shown in the following steps:

1. Create a new cube and rename it `exit_door`.
2. Apply the texture `texture_door` to this object and check the tiling property of the texture (x=1, y =1).
3. Change the position of this object to (x=0, y=1, z=-24) and its scale to (x=2, y=4, z=2).
4. Duplicate this door and rename the duplicate `door1`.
5. Change the position of this object (`door1`) to (x=-2.2, y =1, z=24) and its scale to (x=2, y=4 and z=1.2). This door will open and close automatically.

Let's add water to the water area as shown in the following steps:

1. Select: **Assets | Import package | Water (Basic)**.
2. A new window labeled **Importing package** will appear.
3. Click on the **Import** button.
4. This will import all necessary assets to simulate water.
5. In the **Project** window, select **Assets | Standard Assets | Water (Basic)**.
6. It should include two prefabs, **Daylight Simple Water** and **Nighttime Simple Water**.

7. Drag-and-drop the prefab **Daylight Simple Water** to the **Scene** view and rename it `water`.
8. Change its position to (x=0, y=-2, z=42) and its scale property to (x=26, y=1, z=35).

Add lights as shown in the following steps:

1. We will start by adding lights to the four corners of the maze.
2. Add a point light by selecting **GameObject | Create Other | Point Light**.
3. Rename this light `light_corner`.
4. Change its position to (x=21, y=2, z=-21) and its range to 30.
5. Duplicate this object three times to obtain three additional lights at the positions (x=-21, y=2, z=21), (x=21, y=2, z=21), and (x=-21, y=2, z=-21).
6. Duplicate one of the objects labeled `light_corner` and rename it `light_middle`.
7. Change the position of this object (`light_middle`) to (x=0, y=2, z=0) and its range to 50.
8. Check the scene in the **Scene** view and add other lights as you like.

Now that all necessary objects have been added to the maze, we need to include a ceiling.

1. Create a new cube and rename it `ceiling`.
2. Apply the texture `texture_ceiling` to this object (by selecting **Assets | chapter2**).
3. Change the tiling property of the texture to (x=20, y =20).
4. Change the position of the ceiling to (x=0, y=3.5, z=16) and scale to (x=50, y=1, z=83).
5. Add a first-person controller to the scene (drag-and-drop the prefab **First Person Controller** from the folder **Assets | Standard Assets | Character Controllers**) and change its position to (x=3, y=0.6, z=3).

6. Play the scene (*Ctrl* + *P*) and navigate through the maze.

Navigate to the water area and try to reach the platform by jumping on the successive blocks.

Fine-tuning the level

We have created a very interesting level, using built-in shapes; however, we could modify some details. For example, you will notice that although the maze is sealed-off, we can still see some light in parts where there are no point lights. This is essentially due to ambient light, which can be modified in Unity3D. We can change the properties of the ambient light as follows:

1. Select: **Edit** | **Render Settings**.
2. A window labeled **Render Settings** should be displayed in the **Inspector** window, with several attributes available, including **Ambient Light**.

3. Click on the rectangle to the right of the label **Ambient Light**.
4. A new window labeled **Color** should appear.
5. Under the option labeled **Sliders**, we can see values for each component of the color of the ambient light expressed using the RGB code. The field labeled **R** provides the amount of red included in the color, the field labeled **G** provides the amount of green included in the color, the field labeled **B** provides the amount of blue included in the color, and the field **A** is for the alpha channel (transparency). For our game, we will set the ambient light to a dark gray, which means that the values for the fields **R**, **G**, and **B** components can all be set to `60`.
6. Close the **Color** window.
7. We could also enable the fog by checking the box to the right of the label **Fog**, and setting its density to `.05`.
8. Play the scene (*Ctrl* + *P*).

You may also notice that some lights are flickering and that the lighting changes constantly, especially on the floor. This is because at any given time, the floor may be lit by several point lights. We can solve this problem by changing the quality settings of our scene: select **Edit** | **Project** | **Settings** | **Quality**, and in the section **Rendering**, increasing the **Pixel Light Count** (for example, to `5`).

The render settings the render settings can increase the visual impact of your game, so it is good practice to tweak these settings until we are happy with the look and feel of our game. Render settings need to be defined for each scene and are also available through scripting, so that we can adjust these dynamically. For example, we could simulate a fog with increasing density over time, or ambient light with variable intensity overtime.

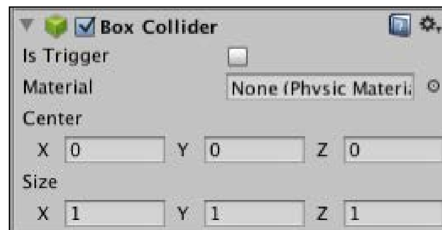
Understanding colliders

So far, we have managed to create a level on which the first-person controller can navigate and collide with both walls and the floor. However, this would not be possible without colliders. When we added the different component of the maze (for example, wall or ground), a collider was added by default to these objects. This makes it possible to detect collision between objects. In our case, the first-person controller includes a default collider. As a result, this character controller collided with the walls. There are several types of colliders in Unity3D, including box, sphere, capsule, wheel, or terrain colliders. Some of these colliders are based on basic shapes such as boxes (for example, box colliders), spheres (for example, sphere colliders), or capsule (for example, capsule colliders).

> It is usually a good practice to use colliders with a shape that is similar to the object on which they are applied. For example, we could use a box collider for a door, or a sphere collider for a ball. Using these basic shapes for collision also has some performance advantages, as they require less computational resources than more customized and precise colliders such as mesh colliders. Note that when the object attached to this collider is resized, the collider is resized accordingly.

Let's look at the floor object and its collider by following these steps:

1. In the **Scene** view, click on one object labeled **floor**.
2. In the **Inspector** window, we can see a list of components for this object. Components can be added to objects either by default (when they are created) or after creation (using the **Component** menu).
3. We can see that one of these components is a **Box Collider** as illustrated in the following screenshot:



Let's test the effect of colliders on walls by doing the following:

1. Select one of the walls in the hierarchy.
2. Identify its box collider in the **Inspector** window.
3. Deactivate the box collider (by checking the box to the right of the label **Box Collider** in the **Inspector** window).
4. Test the scene.
5. We should be able to walk through this wall.
6. Stop the scene and reactivate the box collider on this object.

> We can select several objects and change their properties simultaneously in Unity3D, provided that the property that we need to modify is available in all objects. For example, if we want to deactivate the colliders for two different walls, we could select the two walls (press *Ctrl* and click on the objects in the **Hierarchy** view) and deactivate the component **Box Colliders** once; this change will then be applied to both walls.

We will now remove the collider and add another collider to the object:

1. Select one of the walls in the hierarchy, identify its box collider in the **Inspector** window and remove the collider: right-click on the label **Box Collider** and select **Remove Component** from the contextual menu; doing so should remove the collider.
2. Test the scene; we should be able to walk through the wall.
3. Stop the scene.
4. Add a new collider to this object:
    1. Select the wall in the hierarchy.
    2. Select: **Component | Physics | Box Collider**.
    3. This should add a new box collider to the object.
5. Test the scene.
6. We should now be able to collide with the wall again.

We could do the same for the ground as shown in the following steps:

1. Select the object labeled **floor** in the **Hierarchy** view.
2. Deactivate its box collider in the **Inspector** window.
3. Test the scene.
4. Our character should literally fall indefinitely (because there is no collision detected with the floor).

Colliders can help to make our scene realistic. When colliders are activated (that is, during a collision), built-in functions are called so that we can in turn perform specific actions when this collision occurs (for example, destroy an object or play a sound). We will cover this aspect in the next chapters.

Summary

In this chapter, we have learned to create an indoor environment using basic shapes and standard assets. We have applied different transformations, including moving, scaling, and rotating. We have also learned about colliders and their importance for our game. Finally, we have used a built-in water object or prefab to simulate water in our game. In the next chapter, we will learn how to use JavaScript and interact with objects within the Unity3D scene.

3

Using Scripts to Interact with Objects

In this chapter, we will learn how to use scripting in Unity3D to create a user interface, handle user interaction, and display customized messages. After completing this chapter, we will be able to:

- Create, store, and execute scripts in Unity3D using JavaScript
- Know the main functions necessary for our game
- Know how to select and modify game objects' properties through scripts
- Know the syntax required for scripts
- Know how to use the console for debugging purposes or custom messages
- Know how to pass variables and call functions between scripts
- Create and update Heads-Up-Display information using text and textures

Throughout this chapter, we will code our first scripts. We will learn how to display messages in the console window, and also to create a counter (that is, a timer) that will be used for our game. Finally, we will learn to access objects' properties through scripts (for example, text, texture, or position) and attach a script that rotates an object indefinitely. We will then improve our game and create the Graphical User Interface (GUI), including a Heads-Up-Display (HUD) with the images of the objects collected, the time, or notifications to the player.

All material required to complete this chapter is available for free download on the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.

Introduction to scripting in Unity3D

While we have learned how to create and transform assets in Unity3D in the previous chapters, we will now discover how to code and use scripts. Scripts make it possible to add logic and more interactivity to our games, as well as customize interaction based on the players' actions. In Unity3D, we can create scripts using both JavaScript and C#. While JavaScript is usually considered an easy and accessible scripting language, C# is usually favored by intermediate and advanced Unity3D programmers, as it facilitates the programming workflow and makes it possible to develop more complex programs and interaction paradigms, notably due to its Object-Oriented Programming (OOP) capabilities. For these reasons, it is often difficult for beginners to know and decide what language they will start using. Both of these languages have their advantages and limitations.

All code created in this chapter is available from the companion website `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.

In this chapter, we will start to flesh-out the mechanics for the level we have created in the previous chapter, with the following features:

- The player will try to find the exit of the maze
- The player will have limited time
- Time will be displayed onscreen
- The player will need to find and collect different items
- Objects collected will be displayed onscreen

Importing necessary assets

Open our previous project, if it is not already open, save the scene we have been working on so far (**File | Save Scene**), and duplicate this scene by saving the current scene as `chapter3` (**File | Save Scene As**).

We will now create a container for all objects we created for the maze in the previous chapter, so that it looks tidier:

1. Create an empty object (**Game Object | Create Empty**) and change its name to `maze`.
2. Change its position to (x=0, y=0, z=0).
3. In the **Hierarchy** window, select all objects except the one we have just created (`maze`), and drag-and-drop these objects on the object labeled `maze`.

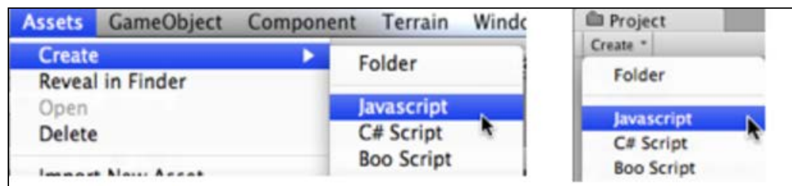
Before we start creating our level, we will need to download the necessary assets from the companion website as shown in the following steps:

1. Open the link for the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.
2. Download the material for this chapter by clicking (or right-clicking and selecting **download to** from the contextual menu) on the package for `chapter3`. Doing so will download a file labeled `chapter3.unitypackage`.
3. In Unity3D, create a new folder called `chapter3`, inside the `Assets` folder, and select this folder (`chapter3`).
4. Import the package we have just downloaded into Unity3D by selecting **Assets** | **Import Package** | **Custom Package**. Browse to the directory where we have saved the package downloaded from the companion website, select it, and click on **Open**.
5. A new window labeled **Importing package** will appear; click on **Import**.
6. This step should create a folder `chapter3_pack` within the folder labeled `chapter3`.

Creating our first script

Scripts can be created in several ways in Unity3D, including from the main menu (**Assets** | **Create** | **JavaScript**/**C# Script**/**Boo Script**) or from the **Project** window (**Create** | **JavaScript**/**C# Script**/**Boo Script**).

Once the script is created, it needs to be linked to an object.



From the scene we have created in the previous chapter, we will create a new script that displays a message in the console. Using scripts usually involves creating the script, and attaching the script to an object. The script is called (or executed) when the object is created in the scene; however, we won't be able to play the scene until all scripts included contain no errors (errors are displayed in the **Console** window). Before creating a new script, we will create a new folder in the `project` folder, so that we can locate the scripts easily:

1. Select the folder `chapter3` that we created earlier.
2. From the **Project** window, select **Create | Folder**.
3. Rename this folder `Scripts`.
4. Click once on this folder (so that the next script is created inside it).

We can now create a new script:

1. Check that the folder `Scripts` is selected.
2. From the top menu, select **Assets | Create | JavaScript**.
3. Doing so should create a new JavaScript script within the folder labeled `Scripts`.
4. Rename this script `timer`.

When the script has been created, we can see its content in the **Inspector** window.

Let's edit its content as shown in the following steps:

1. Double-click on the script labeled `timer`. Doing so should open `MonoDevelop`, which is the default editor for Unity3D scripts. There are several advantages to using `MonoDevelop`, including code auto-completion, so that we don't have to remember all of Unity3D's built-in functions and variables. However, if we had preferred to use other editors, we could have changed Unity3D's preferences accordingly (for example, **Unity | Preferences | External Tool**).
2. Once in `MonoDevelop`, we can see that there are two functions created in the `timer` script by default: these are the functions `Start` and `Update`. The `Start` function is called when the script is first called. For example, if this script is linked to an object, this function will be called when the object is created or added to the scene. The second function, `Update`, is called every frame (that is, when the screen is refreshed). It can be used to detect keystrokes or to create timers, as we will see later in this chapter.

3. Modify the `timer` script as follows:

```
private var time: float;
function Start ()
{
}
function Update ()
{
  time++;
  print(time);
}
```

    - In statement 1 of the previous code, we declare a new variable, `time`, of type `float`. It is only accessible within this script (that is, the type is `private`).
    - In statement 7 of the previous code, the variable time is incremented by 1 every frame (that is, every time the screen is refreshed).
    - In statement 8 of the previous code, the value of the variable `time` is displayed in the **Console** window.

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.

To be able to run this script, it needs to be attached to an object:

1. In Unity3D, create an empty object (**Game Object** | **Empty Object**) and rename it `timer`.
2. We can now attach the script by either dragging-and-dropping the script from the `Scripts` folder to the empty object labeled `timer`, or by selecting the object labeled `timer` and selecting **Component** | **Scripts** | **timer** from the top menu.
3. Once this step is done, if we click once on the `timer` object, the Inspector will reveal an additional component for this object, a script labeled `timer`.
4. Play the scene (*Ctrl* + *P*). If we open the **Console** window (*Ctrl* + *Shift* + *S*) while the scene is being played, we can see that the counter is displayed and that its value increases over time.

You will also notice a message in the **Console** window stating that **There are 2 audio listeners in the scene…** the reason being that each camera in the scene includes an **audio listener**, which can be compared to a microphone. We currently have two cameras in the scene, each with an audio listener component. However, Unity3D requires that only one audio listener be active at any given time. To avoid this warning message, we could deactivate the audio listener for the camera attached to the **First Person Controller** or the one for the other camera labeled **Main Camera**.

This is great; however, the value displayed in the console is the number of frames since the game has started. Instead, we need to display the number of seconds and minutes. To do so, we can use a built-in variable called `Time.deltaTime`. This variable provides the number of seconds elapsed since the last frame was displayed; effectively, the number of seconds is calculated every time the screen is refreshed. Let's modify our script accordingly as described in the following code snippet:

```
function Update ()
{
time = time +Time.deltaTime;
print(time);
}
```

In the previous code, the `timer` is incremented by the number of seconds elapsed since the last frame was displayed. If we play the scene (*Ctrl* + *P*), we should now see seconds displayed in the **Console** window (*Ctrl* + *Shift* + *C*). We are almost there and we need to create two variables (that is, for minutes and seconds), and calculate the minutes and seconds elapsed based on the variable `timer`. Modify the script as follows:

```
private var time: float;
private var minutes:int;
private var seconds:int;
function Start ()
{
}
function Update ()
{
  time = time + Time.deltaTime;
  minutes = time/60;
  seconds = time%60;
  print(minutes+":"+ seconds);
}
```

- In statements 1-3 of the previous code, the variables `time`, `minutes`, and `seconds` are declared
- In statements 10-11 of the previous code, minutes and seconds are calculated (the operand `%` is called a modulo; it will provide the reminder of the division)
- In statement 12 of the previous code, both variables `minutes` and `seconds` are displayed in the console

Play the scene and check that the minutes and seconds are displayed in the **Console** window.

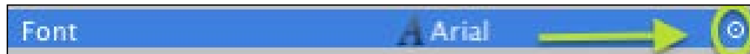
At this stage, we have a script that displays the time in the console; however, for our game, it would be better if this script could be displayed on the screen using a graphical interface. As mentioned in the previous sections, Unity3D provides several game objects for the creation of graphical interface objects, including `GUIText` and `GUITexture`:

1. Create a `GUIText` object: select **GameObject | Create Other | GUI Text**.
2. Doing so should create a new `GUIText` object. Rename it `GUIText_timer`.
3. If we switch to the game view, we should now see the text GUI Text in the middle of the screen (this is the default text).
4. Switch to the **Scene** view, select the object (`GUIText_timer`) in the **Hierarchy** view, and look at the **Inspector** window; we can see several components for this object, including its transform. Within this component, we can also see the position attributes of the object. The position of the `GUIText` objects is a proportion of width and height of the screen, and the origin (0,0) is located at the bottom-left corner of the screen, which means that a position (x=0.5, y=0.5) corresponds to the middle of the screen. For our game, we would like the time to be displayed in the top-left corner, so we will modify its coordinates to (x=0.02, y=0.9).
5. In the **Inspector** window, open the `GUIText` component of this object by clicking on the arrow to the left of the label `GUIText`. We can see several attributes including `Text` (that is, the text that will be displayed), `Anchor`, `Alignment`, `Font`, and `Font-size`. Change the `Font-size` attribute to `30`.

We can also change the font of the text displayed by dragging-and-dropping a new font on this variable. Let's use a new font for our timer:

1. Open the site `www.dafont.com` in your web browser. This site offers free fonts that can be used for personal purposes (that is, non-commercial).
2. Look for the font **Oh The Horror**.
3. Once you have found this font, make sure you read the terms and conditions.

4. Download this font using the **Download** button located on the right-hand side of the page and unzip the file (`OhTheHorror.ttf`).
5. Switch back to Unity3D, select the folder `chapter3`, and import the font into Unity3D (that is, select **Assets | Import New Asset**). Doing so should create a new font in the **Project** window (that is, **Oh the horror**).
6. Select the `GUIText` object labeled `GUIText_timer`.
7. In the **Inspector** window, click on the small circle to the right of the label **Font** as illustrated in the following screenshot:

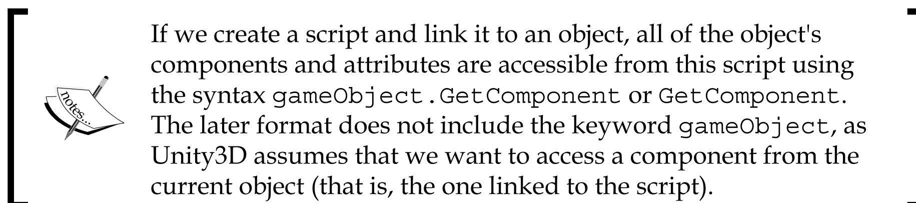


8. This should open a window labeled **Select font** that includes the new font **Oh The Horror**. Select this font (that is, click once on the font) and close the font selection window. The new font should now appear in the **Font** property of the previous object.

Last but not least, we need to link our timer script to the `GUIText`, so that the time is displayed onscreen. For this, we will use a new command called `GameObject.Find()`. This built-in function makes it possible to find an object and access its components and attributes from a script. In our case, we need to access the object `GUIText_timer` and its `GUIText` component, and modify its attribute `text`. The following code illustrates how this can be done and how it can be added at the end of the function `Update` in the script `timer`:

```
var textToDisplay:String = minutes+":"+seconds;
GameObject.Find("GUIText_timer").guiText.text=textToDisplay;
```

In the previous code snippet, we modify the `guiText` variable, which is in the component `GUIText`, attached to the object `GUIText_timer`. Accessing variables is done through a hierarchy: **Object | Component | Attribute**. Note that we use lowercase for the term `guiText` when referring to the `GUItext` from the object `GUITextTimer`. The first three letters (that is, gui) are in lower case, because we are accessing this particular `guiText` (it has been created).



If we create a script and link it to an object, all of the object's components and attributes are accessible from this script using the syntax `gameObject.GetComponent` or `GetComponent`. The later format does not include the keyword `gameObject`, as Unity3D assumes that we want to access a component from the current object (that is, the one linked to the script).

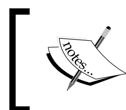
The full code for the function timer should look as follows:

```
private var time: float;
private var minutes:int;
private var seconds:int;
function Start ()
{
}
function Update ()
{
  time = time + Time.deltaTime;
  minutes = time/60;
  seconds = time%60;
  var textToDisplay : String = minutes + ":" + seconds;
  GameObject.Find("GUIText_timer").guiText.text = textToDisplay;

}
```

Play the scene and we should now see the time on the screen.

Note that while the code we have created to update the object `GUIText_timer` (that is, time onscreen) is syntactically correct, it generates memory garbage, as the `GUITexture` is searched for every frame and stored over and over. The same applies to the variable `textToDisplay`. Instead, a better practice would be to look for this object once, store it, and refer to this object in the `Update` function (that is, without calling `GameObject.Find` indefinitely. We could also declare the variable `textToDisplay` once.



It is good practice to use the function `GameObject.Find` and similar functions outside the `Update` function, as it may impact negatively on the performance of your game otherwise.

Let's modify our code accordingly:

Add two new variables at the start of the script as follows:

```
private var guiTextToDisplayTime : GameObject;
private var textToDisplay : String;
```

Add the following code to the `Start` function:

```
guiTextToDisplayTime = GameObject.Find("GUIText_timer");
```

In the `Update` function, replace the following line:

```
GameObject.Find("GUIText_timer").guiText.text = textToDisplay;
```

With this line:

```
guiTextToDisplayTime.guiText.text = textToDisplay;
```

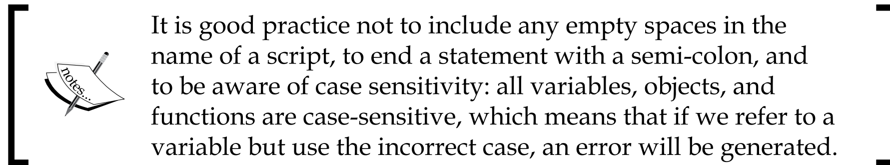
Replace the following line:

```
var textToDisplay : String = minutes + ":" + seconds;
```

With this line:

```
textToDisplay = minutes + ":" + seconds;
```

Play the scene and check that our code works properly.



It is good practice not to include any empty spaces in the name of a script, to end a statement with a semi-colon, and to be aware of case sensitivity: all variables, objects, and functions are case-sensitive, which means that if we refer to a variable but use the incorrect case, an error will be generated.

Collecting objects

So far, we have managed to implement navigation around the maze as well as a timer. As we initially planned, we will now need to make it possible for the player to collect objects (for example, med packs and keys) to escape the maze. To collect the objects, we will use collision detection and a script that will destroy the objects collected. First, let's create some items and add them to the scene:

1. Create a new cube, rename it `medpack`, then change its position to (x=3, y=1, z=6), and its scale to (x=0.5, y=0.5, z=0.2).
2. Locate the texture labeled `texture_medpack` in the folder **Assets** | **chapter3** | **chapter3_pack**. Apply the texture `texture_medpack` to the cube.
3. Create a new script inside the folder **chapter3** | **Scripts** and rename it `rotate`.

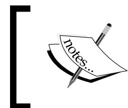
Open this script (double-click on it) and add the following code:

```
function Start ()
{
}
function Update ()
{
  transform.Rotate(Vector3(0,1,0), 180*Time.deltaTime);
}
```

- In statement 4 of the previous code, the function `Update` will be called every frame and modify the rotation of the object attached to the script.
- In statement 6 of the previous code, a rotation of 180 degrees per second around the y axis is applied to the transform of the object linked to this script. The y axis is specified as the axis of rotation using the syntax `Vector3(0,1,0)`. A transform component relates to any transformation applied to the object including translation, rotation, and scaling.

Link the script to the object labeled `medpack`, and play the scene (either by dragging-and -dropping the script onto the object or by selecting the object and then **Component | Scripts | Rotate**). Play the scene, and check that the med pack is rotating.

Now that we have made this med pack more visible, we need to detect collisions with the player as follows: objects will be given a tag (that is, a label), and when the collision occurs, a built-in function will be called to obtain information on the collider involved in the collision; we will then check the tag of the object we are colliding with based on the collider mentioned previously, and if this object can be collected, then we will destroy it and update the game information accordingly (for example, increase the score, update the inventory, or display a customized message).



It is good practice to give a tag to the objects used in your game; it makes it easier to identify them, to group objects based on common properties, and to create custom behaviors or scripts accordingly.

To add a tag, we can proceed as follows:

1. Select the object labeled `medpack` in the **Hierarchy** window (click once on the object).
2. In the **Inspector** window, click on the drop-down menu to the right of the label **Tag**.
3. From the drop-down menu, select the option **Add tag**.
4. This should open a window called **Tag Manager**.
5. In this window, click on the arrow to the left of the label **Tags**; this will display a list of elements (or tags) available.
6. Click to the right of the last elements listed (for example, `Element0`), type medpack, and press *Enter*. This will create a new tag named `medpack`.
7. To apply this tag, click on the object labeled `medpack` in the **Hierarchy** window and click on the drop-down menu to the right of the label **Tag** in the **Inspector** window. This time the new tag `medpack` should appear.
8. Click on this tag to select it for our object.

To handle the collision, we will now create a script to be added to the **First Person Controller** object:

1. Create a new script inside the folder **Assets | chapter3 | Scripts** and rename it `collisionDetection`.
2. Add the following code to the script:

   ```
   function OnControllerColliderHit(c : ControllerColliderHit)
   {
     print("collided with "+c.gameObject.tag);
   }
   ```

   - In statement 1 of the previous code, the function `OnControllerColliderHit` is declared. It is a built-in function that is used to handle collisions between a character controller and other objects. The parameter c is the collider of the object colliding with the controller.
   - In statement 3 of the previous code, we print a message in the console that indicates the tag of the object the controller is colliding with. The tag is obtained from the object, which is obtained from the collider (that is, **collider | object | tag**).

`OnControllerColliderHit` is a built-in function that Unity3D will call in case of a collision between the controller and other objects. If we misspell the name of this function (for example, use a lowercase `o` for its initial instead of an uppercase `O`, the function may not generate any error, but collision will not be handled in this function. The collision will not be handled because Unity3D will assume that we created a function called `onControlerColliderHit` rather than the one it is expecting.

Attach the script to the **First Person Controller** object (that is, to the top-most level) located in the folder (or empty object) labeled `maze`, open the **Console** window (*Shift* + *Ctrl* + *C*) and test the scene (*Ctrl* + *P*). After colliding with the med pack, we should see a message in the **Console** window saying **collided with medpack**.

Because the player is constantly walking (and colliding with the ground), and that the ground has no tag assigned yet, the **Console** window will display the message **collided with untagged**. Because this collision happens constantly (unless the player is jumping), the **Console** window may be flooded with messages. We may enable the option **collapse** in the **Console** window (button located at the top-left corner of the **Console** window); this will prevent messages from being displayed repeatedly and collapse identical messages accordingly.

Press *Ctrl* + *P* to return to the editing mode. We will now modify this script to destroy the med pack by adding the following line to the script:

```
if (c.gameObject.tag == "medpack") Destroy(c.gameObject);
```

In the previous line, we detect the tag of the object we are colliding with and then destroy this object accordingly. Test the scene by colliding with the med pack and check that it disappears.

Adding audio

Finally, it would be great to add additional feedback when the object has been collected, using an audio cue. This will be done as follows:

1. Open the script `collisionDetection`.
2. Add the following two lines at the start of the script:
   ```
   @script RequireComponent(AudioSource)
   public var collection_beep : AudioClip;
   ```
   - In line 1 of the previous code, we specify that the script requires an audio source component for this object
   - In line 2 of the previous code, we declare a variable that will be used for the sound to be played
3. Locate the sound `collection_beep` from the folder **Assets | chapter3 | chapter3_pack**.
4. In the **Hierarchy** window, click on the **First Person Controller** object (top-most level). In the **Inspector** window, drag-and-drop the file `collection_beep` to the variable `collection_beep` in the component **Collision Detection** of the object **First Person Controller**.
5. Check that the **First Person Controller** object is still selected.
6. Select **Component | Audio | Audio Source**.
7. This should add an **Audio Source** component to the **First Person Controller** in the **Inspector** window.
8. Check that the option **Play on Awake** is not selected for this component.

Modify the script `collisionDetection` so that it plays the sound on collision:

1. Open the script `collisionDetection`.
2. Add the following line to the script (that is, within the conditional statement):
   ```
   gameObject.audio.Play();
   ```

3. The function `OnControllerColliderHit` should now look as follows:

```
function OnControllerColliderHit(c : ControllerColliderHit)
{
  print("collided with"+c.gameObject.tag);
  if (c.gameObject.tag == "medpack")
  {
    Destroy(c.gameObject);
    audio.clip = collection_beep;
    audio.Play();
  }
}
```

   - In statement 7 of the previous code, we specify the audio clip that should be played.
   - In statement 8 of the previous code, we play the current audio clip. Note that we could also use the following code to play the sound:

     ```
     audio.PlayOneShot(collection_beep);
     ```

> The sound that we have created is automatically a 3D sound, which means that if it was located on a different object, the way the player perceives it could differ based on the position and orientation of the player in relation to the object. In this case, we would need to add an audio listener to the **First Player Controller**, so that this 3D sound can be heard and played accordingly. In our case, the sound will be played through the speakers, regardless of the relative position of the player and the object.

We could create other objects and use a similar technique to destroy them after collision:

1. Create a new box, rename it `keys`, change its position to (x=2, y=1, z=6), and its `scale` property to (x=0.2, y=0.5, z=0.5); then apply the texture called `texture_key` from the folder **Assets** | **chapter3** | **chapter3_pack** to this box.
2. Create a new box, rename it `gun`, change its position to (x=1, y=1, z=6), and its `scale` property to (x=0.2, y=0.5, z=0.5); then apply the texture called `texture_gun` from the folder **Assets** | **chapter3** | **chapter3_pack** to this box.
3. Attach the script labeled `rotate` to both objects (`key` and `gun`) and test the scene.

Creating and displaying an inventory system

In our game, in addition to med packs, we will be able to collect other types of objects. We will then need to keep track of these objects using variables and graphical representations. This can be done using a basic inventory system. Some objects will have an effect on the player (for example, increase health), while other objects will be used at a later stage. To keep track of these objects we will need to create corresponding variables, update these variables when the corresponding objects have been collected, display a graphical representation of the object(s) collected, and modify the players' attributes (for example, `its health`). We will be working with the script `collisionDetection`. First, let's create variables for the objects to be collected and add the following lines at the start of the script:

```
private var hasKey : boolean;
private var hasGun : boolean;
private var health : int;
```

- In lines 1 and 2 of the previous code, the varibles `hasKey` and `hasGun` are declared as Boolean variables to check whether the player has collected the keys or the gun.
- In line 3 of the previous code, the variable `health` is declared as an integer. It will be used to track the health levels of the player. It will decrease when hit by enemies. The health levels will increase to 100 percent after collecting the med pack.

Then, we need to specify what actions should be performed when these items have been collected. This will be done within the code dedicated to the collision detection. Let's add the following code to the script:

```
function OnControllerColliderHit(c : ControllerColliderHit)
{
  if (c.gameObject.tag == "medpack" || c.gameObject.tag ==   "key"
  || c.gameObject.tag == "gun")
  {
    print("collided with "+c.gameObject.tag);
    Destroy(c.gameObject);
    gameObject.audio.Play();
    if (c.gameObject.tag == "medpack") health = 100;
    if (c.gameObject.tag == "key") hasKey = true;
    if (c.gameObject.tag == "gun") hasGun = true;
  }
}
```

- In statement 3 of the previous code, we test for any type of collectable objects, since all of them will be destroyed and a sound will be played upon collision
- In statements 6 and 7 of the previous code, as per the previous examples, the objects are destroyed and a sound is played
- In statement 8 of the previous code, in the case of a med pack, the health of the player is set to `100`
- In statement 9 of the previous code, in the case of a key, the variable `hasKey` is set to `true`
- In statement 10 of the previous code, in the case of a gun, the variable `hasGun` is set to `true`

Before we can test the scene, we need to create tags for the keys and gun.

1. Select the object labeled `gun` in the **Hierarchy** window.
2. In the **Inspector** window, click on the drop-down menu to the right of the label **tag**.
3. From the drop-down menu, select the option **Add tag**.
4. In the **Tag Manager** window, click on the arrow to the left of the label **Tag**.
5. Click to the right of the last elements listed (**Element1**) and type `gun`.
6. Click on the object labeled `gun` in the **Hierarchy** window and click on the drop-down menu to the right of the label **tag** in the **Inspector** window. Select the tag `gun`.
7. Repeats steps 1-6 for the object labeled `key`, that is, create and apply a new label named `key` for the key.

Play the scene and check that when we collide with the key or gun, the **Console** window displays a message accordingly (for example, **collided with gun**).

At present, although we keep track of the different items collected, we need to look in the **Console** window to receive feedback. It would be great to display this information on the screen as well as a notification message that disappears after 3 seconds. First, let's create a script named `displayMessageToUser` that will display a notification message on the screen and hide it after few seconds. Add the following code to the script:

```
private var timer:float;
private var displayTime:float;
private var timerIsActive:boolean;
private var message: String;
```

- In statement 1 of the previous code, the variable `timer` is declared; it will be used to measure time elapsed since the message was first created.
- In statement 2 of the previous code, the variable `displayTime` is declared; it will be used to specify for how long the message will be displayed.
- In statement 3 of the previous code, the variable `timerIsActive` is declared; it will be used to specify whether the timer that controls the message is active.
- In statement 4 of the previous code, the variable `message` is declared. This message will be displayed on the screen.

Next, we will need to create a function that starts the timer and displays the message until the time is up. Include the following code to the script:

```
function startTimer()
{
  timer = 0.0f;
  guiText.text = message;
  timerIsActive = true;
  displayTime = 3.0f;
}
```

- In statement 1 of the previous code, the function `startTimer` is declared
- In statement 3 of the previous code, the `timer` is set to `0`
- In statement 4 of the previous code, the message is displayed on the screen (provided that the variable message was initially set; this will be explained in the next section)
- In statements 5-6 of the previous code, the `timer` is now active for 3 seconds

After the timer has been activated, we need to update its value overtime by adding the code highlighted in the next code snippet to the function `Update` as follows:

```
function Update()
{
  if (timerIsActive)
  {
    timer+=Time.deltaTime;
    if (timer > displayTime){ timerIsAcive=false;
   guiText.text="";}
  }
}
```

- In statement 3 of the previous code, we test whether the timer is active
- In statement 5 of the previous code, the value of the time is incremented by the number of seconds since the last frame was displayed
- In statement 6 of the previous code, if the timer has reached the time limit for the text to be displayed, the timer is made inactive and the text is deleted from the screen

Then, we need to create a function that displays the time:

```
function displayText(mes:String)
{
  message = mes;
  startTimer();
}
```

- In statement 1 of the previous code, the function has one parameter, which corresponds to the text to be displayed onscreen
- In statements 3 and 4 of the previous code, the message to be displayed is initialized with the variable passed to this function and the timer is started

After adding the code described in the previous section, the script `displayMessageTouser` should look as follows:

```
private var timer:float;
private var displayTime:float;
private var timerIsActive:boolean;
private var message: String;
function Start ()
{
}
function startTimer()
{
  timer=0.0f;
  guiText.text = message;
  timerIsActive = true;
  displayTime = 3.0f;
}
function Update ()
{
  if (timerIsActive)
  {
    timer+=Time.deltaTime;
    if (timer > displayTime)
    {
```

```
        timerIsActive= false; guiText.text="";
      }
    }
  }
  function displayText(mes:String)
  {
    message = mes;
    startTimer();
  }
```

We just need to create the `GUIText` component to display the information on the screen. Create a `GUIText` object, rename it `GUIText_displayMessageToUser`, and change its position to (x=0.5, y=0.7, z=0). In the `GUIText` component, change the **Anchor** attribute to **middle center** and **Font Size** to `40`. Attach the script `displayMessageToUser` to this object.

To complete this functionality, we now need to call the function `displayText` whenever the player collects an item. To do so, let's modify the script `collisionDetection` and add the following code after the line that starts with `Destroy(c.gameObject)`:

```
GameObject.Find("GUIText_displayMessageToUser").GetComponent(displ
  ayMessageToUser).displayText(c.gameObject.tag+" collected!");
```

In the previous code, we accessed the function `displayText` from the script `timer`. The text to display is a string that consists of the tag of the object collected (that is, `c.gameObject.tag`) followed by the text **collected**!

Play the scene (*Ctrl* + *P*) and check that the message is displayed accordingly when we collect an item. We may notice that the text **GUI Text** is displayed automatically when the scene starts; we can delete this text by adding the following line of code in the `Start` function of the script `displayMessageToUser`.

```
guiText.text = "";
```

Having displayed a notification text, we will display some of the items collected onscreen as part of our inventory system. Every time an item is collected, we will set the corresponding Boolean value to `true` and also display a texture onscreen. This will involve the following steps:

1. Create `GuiTexture` objects for each collectable item based on these textures.
2. Assign default textures.
3. Initially deactivate the assigned textures.

4. Activate these textures through JavaScript when the corresponding objects have been collected. Let's start by creating `GUITextures` for the keys:
    1. Create a new `GUITexture` object (**GameObject** | **Create Other** | **GUI Texture**).
    2. Rename this `GUITexture` object `GUITexture_key` and select this object; check that the **Inspector** window is visible.
    3. Locate the texture `icons_collectable_keys` in the folder **Assets** | **chapter3** | **chapter3_pack**.
    4. Drag-and-drop this texture in the **Inspector** window, to the left of the label **Texture** in the **GUITexture** component of the object `GUITexture_key`.
    5. Using the Inspector, change the position of this object to (x=0.05, y=0.1, z=0), this should display the texture in the bottom-left corner of the screen.
    6. Repeat the previous steps for the gun; the new object will be named `GUITexture_gun`, its position should be (x=0.15, y=0.1, z=0), and the texture used will be `icons_collectable_gun`.

Now that we have created the two `GUITextures`, we need to make them visible only when the corresponding object has been collected. To do so, we will update the script `collisionDetection`, which is linked to the **First Person Controller**, by adding the following code:

```
function changeGUITexture(toBeDisplayed:boolean, label:String)
{
  GameObject.Find("GUITexture_"+label).guiTexture.enabled =
  toBeDisplayed;
}
```

- In statement 1 of the previous code, the function is declared. It has two parameters: a string variable (`label`) and a Boolean variable (`toBeDisplayed`). The first parameter determines whether a specific texture needs to be displayed, whereas the second parameter will be used to identify which texture will be displayed. Depending on the value of the variable `toBeDisplayed`, the corresponding texture will be either displayed (`true`) or hidden (`false`).
- In statement 3 of the previous code, we use the built-in function `GameObject.Find` to identify the `GUItextutre` object with the corresponding label that we need to either hide or display (for example, `GUITexture_key` or `GUITexture_gun`).

Next, we will need to specify when these textures will need to be hidden or displayed. Add the code that is highlighted in the following code snippet:

```
function OnControllerColliderHit(c : ControllerColliderHit)
{
if (c.gameObject.tag == "medpack" || c.gameObject.tag == "key" ||
  c.gameObject.tag == "gun")
    {
      if (c.gameObject.tag == "key") {hasKey = true;
  changeGUITexture(true, "key");
    }
  if (c.gameObject.tag == "gun") {hasGun = true;
  changeGUITexture(true, "gun");}
```

- In statement 5 of the previous code, if the key has been collected, the variable `hasKey` is set to `true` and the function `changeGUITexture` is called with the parameters `true` and `key`, which means that we will display the `GUITexture` for which the name includes the text **key** (that is, `GUITexture_key`)
- In statement 7 of the previous code, we will display the `GUITexture` for which the name includes the text **gun** in the same way as for the key

Finally, we need to hide the `GUITextures` at the start of the scene. This can be done using the `Start` function within the `collisionDetection` script; add the following code:

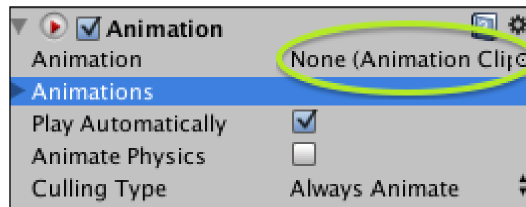
```
function Start ()
{
  hasGun = false;
  hasKey = false;
  health = 0;
  changeGUITexture(false, "key");
  changeGUITexture(false, "gun");
}
```

The highlighted statements in the previous code suggest that the `GUITextures` for the key and the gun are hidden.

Finishing the game

We should now have created four scripts as well as new objects: `GUIText_displayMessageToUser`, `GUIText_timer`, `GUITexture_gun`, `GUITexture_key`, `gun`, `key`, `medpack`, and `timer`. The player can navigate through the maze and collect items. At this stage, we just need to detect when the player has reached the exit doors (that is, using tags) and open the doors only if the player has the key. To do so, we will create a tag for the door, attach an animation to it, and trigger this animation only when the player has collected the key as follows:

1. Select the object labeled `exit_door` (within the empty object or folder `maze`).
2. Add an animation component to this object by selecting **Component** | **Miscellaneous** | **Animation**.
3. Locate the animation `open_door` from the folder **Assets** | **chapter3** | **chapter3_pack**.
4. Drag-and-drop this animation on the `Animation` attribute for the **Animation** component of the object `exit_door` as highlighted in the following screenshot:



Uncheck the box for the option **Play Automatically**, create a new tag titled `exit_door`, add this tag to the object named `exit_door`, and modify the script `collisionDetection` by adding the following lines within the function `OnControllerColliderHit`:

```
if (c.gameObject.tag == "exit_door")
{
  if (hasKey) c.gameObject.animation.Play ("open_door");
  else
  GameObject.Find("GUIText_displayMessageToUser").GetComponent(displ
    ayMessageToUser).displayText("Sorry, you need the key to open
      this door");
}
```

- In statement 1 of the previous code, we test whether we are colliding with the exit door
- In statement 3 of the previous code, if the player has the key, the door (which is the object we are colliding with) is open by playing the animation linked to its object (that is, `door_open`)
- In statement 5 of the previous code, if the player does not have the key, a message is displayed accordingly

Test the scene. Try to open the door with and without the key.

Summary

In this chapter, we have learned to create scripts using JavaScript. From the first section, we created our first script and gradually introduced more interaction into our game, including rotating objects, a timer, and collision detection. Throughout this section, we have also learned how to interact with other `GameObjects` in the game, including `GUIText` and `GUITexture` objects. We have then combined these skills to display a notification text every time an object has been collected, as well as a corresponding texture on the screen. Finally, we have learned how to play an animation through JavaScript to open the exit door. In the next chapter, we will add more elements to our GUI, and we will also create a script to fire bullets.

4
Creating and Tracking Objects

In this chapter, we will add more interaction to our game: with special effects as well as additional GUI elements, including a mini-map. We will also look at advanced techniques to handle cameras and camera views.

After completing this chapter, we will be able to:

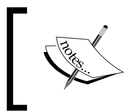
- Instantiate objects in real-time
- Switch between and display multiple camera views
- Define and apply layers to filter content displayed by a camera
- Apply special effects (for example, sparks)
- Display a real-time map of the current level

Throughout this chapter, we will improve the existing game, and add a script for the hero to fire bullets, as well as a real time mini-map (for example, with an updated position of the hero and the items to collect). All material required to complete this chapter is available for free download on the companion website `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.

In the previous chapters, we have managed to create an environment where the player could collect items and open the exit door if he/she had the corresponding keys. In this chapter, we will build upon the skills that we have acquired to improve the game play: a health bar will be displayed on the screen at all times, the player will be able to see a top view of the maze that reveals its position as well as the position of items that can be collected, and the player will also be able to fire a gun. Before we start creating our level, we will need to download the necessary assets from the companion website as follows:

1. Open the link for the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.
2. Download the material for this chapter by clicking (or right-clicking and selecting **Download to** from the contextual menu) on the link package for `chapter4`. This will download a file labeled `chapter4.unitypackage`.
3. In Unity3D, create a new folder titled `chapter4` inside the `Assets` folder, and select this folder (`chapter4`).
4. Import the package we have just downloaded into Unity3D. From Unity3D, select **Assets | Import Package | Custom Package**. Browse to the directory where we saved the package downloaded from the companion website, and select it.
5. This should create a folder titled `chapter4_pack` within the folder labeled `chapter4`.

Finally, we will duplicate the scene we have created in the previous chapter by saving the current scene (**File Save | Scene**), and then saving this scene as `chapter4` (**File | Save Scene As...**). This way, we have preserved the scene created in *Chapter 1*, *Getting to Know Unity3D*, as `chapter3`, and we can start a new scene that includes the same content and that is named `chapter4`.



It is also possible to duplicate the previous scene from the **Project** view by selecting the scene labeled `chapter3` in the **Project** view, duplicating it (**Edit | Duplicate**), and renaming the new scene `chapter4`.

Displaying the health bar

At present, the player's health is saved in the script `collisionDetection`; however, it is not represented on the screen. We will create a new script that displays a health bar symbolized by a rectangle in the top-left corner of the screen. Its length will be proportional to the player's health (that is, ranging from 0 to 100 percent), and the color will also vary accordingly. For example, it will be green when the health is between 67 percent and 100 percent, orange when the health is between 33 percent and 67 percent, and red when the health is between 0 percent and 33 percent. These visual cues will help the player to judge when it is time to look for and collect med packs. Follow these steps to display the health bar:

1. Create a new folder labeled `Scripts` by selecting **Assets** | **chapter4**.
2. Create a new script (JavaScript) inside this folder, rename it `HealthBar`, and add the following code to it:

   ```
   private var  currHealth : int = 45;
   private var currentColor:Texture2D;
   public var style:GUIStyle;
   public var redTexture:Texture2D;
   public var greenTexture:Texture2D;
   public var orangeTexture:Texture2D;
   public var blackTexture:Texture2D;
   ```

   - Line 1 of the previous code shows how the current health value is stored. It will be used to display health levels onscreen.
   - Line 2 of the previous code shows how the texture will be used when the health bar is declared.
   - Line 3 of the previous code shows how a `style` variable is defined and will be used later on.
   - Lines 4-7 of the previous code show how four `Texture2D` variables are created for the health bar.

We can now use each of these textures to draw the health bar. Add the following code to the script `HealthBar`:

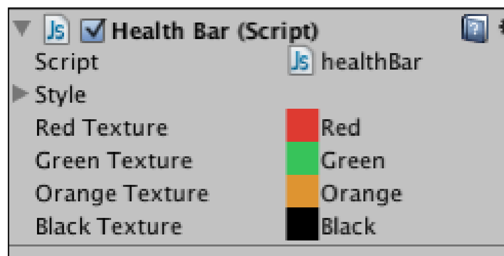
```
function OnGUI()
{
  if (currHealth > =67)
currentColor = greenTexture;
  else if (currHealth >= 34)
currentColor = orangeTexture;
  else currentColor = redTexture;
  style.normal.background = blackTexture;
```

```
    GUI.Box(Rect(0,0, 100,20),"", style);
    style.normal.background = currentColor;
    GUI.Box(Rect(0,0, currHealth,20),"",style);
}
```

- Line 1 of the previous code shows how the built-in function `OnGUI` is defined. It will be used for the GUITextures.
- Lines 3 to 7 of the previous code show how the color of the health bar is set according to the value of the variable `currHealth` (that is, red, orange, or green).
- Lines 8 and 9 of the previous code show how we can draw a black rectangle behind the health bar, so that it can be seen easily against the background.
- Line 9 of the previous code shows how the black rectangle is drawn. Its top-left corner is located at the position (x=0, y=0), and it is 100 pixels wide and 20 pixels high. No default text is displayed. The color used is black (that is, the texture `blackTexture`).
- Lines 10 to 11 of the previous code show how we choose the current color for the health bar (for example, green, orange, or red) and the health bar is drawn.

We now need to create an empty object and link it to this script to be able to display the health bar:

1. Create an empty object and rename it `healthBar`.
2. Attach this object to the script `healthBar` that we created previously.
3. Locate the **Red** texture by selecting to **Assets** | **chapter4** | **chapter4_pack**.
4. Select the object **healthBar** in the **Hierarchy** view, and drag-and-drop the **Red** texture to the right of the variable called `redTexture` in the component called **Health Bar**.
5. Repeat the previous two steps for the textures **Green**, **Orange**, and **Black** as illustrated in the following screenshot:



6. Play the scene; we should see a health bar in the top-left corner as illustrated in the following screenshot:

So far, it sounds great. However, we need to modify the script `collisionDetection`, so that when a med pack is collected, the health bar is updated accordingly and turns to green. For this, we need to create a function in the script `healthBar` that can be called from the script `collisionDetection` when we collide with a med pack.

Open the script `healthBar` and add the following function to it:

```
public function setHealth(updatedValue:int)
{
  currHealth = updatedValue;
}
```

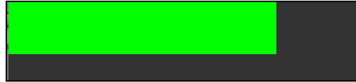
- In line 1 of the previous code, we create a function that will be accessible from outside the script. It takes one parameter that is the new value to be displayed for the player's health.
- In line 3 of the previous code, the variable health is updated accordingly.

Open the script `collisionDetection` and modify the conditional statement linked to the med pack as follows (that is, add the code that is highlighted):

```
if (c.gameObject.tag == "medpack")
{
  health = 100;
  GameObject.Find("healthBar").GetComponent(healthBar).
    setHealth(health);
}
```

- Statement 1 of the previous code already existed in this script, but we have added curly brackets, as two different sets of instructions will be performed if the med pack is collected
- In Statement 4 of the previous code, as in the previous sections, we access the function `setHealth` within the script called `healthBar`, and we pass the value of the variable health as a parameter

Play the scene, collect a med pack, and check that the health bar turns to green (100 percent) as illustrated in the following screenshot:



Displaying a mini-map of the level

In this section, we will create a mini-map of the level to help the player navigate and anticipate the position of collectable objects or enemies. First, let's create a top-down view of the maze. To add a different view of the maze, we will create a new camera and display its content in the top-right corner of the screen as shown in the following steps:

1. Create a new camera (**Game Object** | **Create Other** | **Camera**).
2. Rename this camera `camera1`.
3. Rotate this camera about 90 degrees around the x axis, so that its rotation properties are (x=90, y=0, z=0), and change its position to (x=0, y=50, z=0).
4. If we click on this camera in the **Hierarchy** view, and look at the camera preview (that is, the rectangle in the bottom-right corner of the **Scene** view), we should see our level from above.

We will now add this camera to the main view, so that the user can see this top view as part of the user interface. This will be done using view ports:

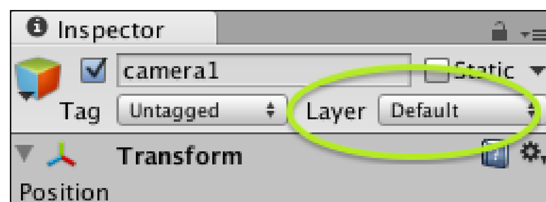
1. Click once on the camera labeled `camera1`.
2. Look at the **Inspector** window and click on the arrow to the left of the camera component to reveal its properties.
3. Change the attribute `Normalized View Port Rect`, as follows: x=0.75, y=0.75, w=0.25, and h=0.25.
4. Change the attribute `Depth` to `1`.

These changes affect the view port, or the area of the screen where the camera view will be displayed. The view port is defined by a rectangle and a depth parameter. The x, y, w, and h parameters correspond to the coordinates of the top-left corner (x and y), the width of the rectangle (w), and its height (h). However, all four parameters are expressed as a proportion of the screen (that is, from 0.0 to 1.0). In our example, the rectangle or view port will occupy an eighth of the surface of the screen. The last parameter (`Depth`) is set to 1, because we need it to be drawn on top of the camera used for the **First Person Controller**, for which the `Depth` is `0` (the camera with the highest depth value will be drawn on top).

5. Delete the components **Audio Listener**, **GUILayer**, and **Flare Layer** (right-click on the component and select **Remove Component** from the contextual menu), as we will not need these.

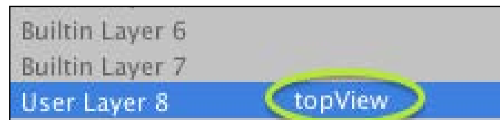
If we play the game, we can see a top-down view displayed in the top-right corner of the screen. However, some of the information displayed on this screen is not relevant (for example, **GUI text** or **GUItexture**). We need to filter the content displayed through this camera, and this can be achieved through layers. Unity3D makes it possible to define and apply layers. For example, some objects can be added to a layer, and we can then define what layers each camera will display. First, let's define layers for all active cameras:

1. Click on the object `camera1` from the **Hierarchy** window.
2. In the **Inspector** window, click on the drop-down menu to the right of the **Layer** label:



3. Select the option **Add Layer** from the drop-down menu.

4. This should open a new window labeled **Tag Manager**, which is the same window we used for the objects' tags. This window lists a series of built-in layers (for example, **Builtin Layer 0** or **Builtin Layer 7**) as well as user layers (for example, **User Layer 8** or **User Layer 31**).
5. Modify the first user layer by clicking on to the right of the label **User Layer 8**.
6. Type `topView` and press *Enter*. This should display the label **topView** to the right of **User Layer 8**.



7. Select the object `camera1` in **Hierarchy**.
8. In the **Inspector** window, within the component camera, modify the attribute **Culling Mask**, so that only the layer labeled **topView** is selected as illustrated in the following screenshot: select the option **Nothing** and then the option **topView**. This means that the camera `camera1` will only display objects that belong to this layer.

Next, we will make sure that this top-view camera is always above the player:

1. Drag-and-drop the camera `camera1` on the **First Person Controller** (top-most level; the **First Person Controller** is located in the folder or empty object `maze`) as illustrated in the following screenshot:

2. Change its position to (x=0, y=50, z=0).

Now that we have set the top-view camera, we can set layers for the objects that we need to display on the map. We could decide to display the first-person controller and other objects of interest; however, it would be great to have a simplified representation of these on the mini-map, and only display dots with corresponding colors. For example, we could have a red dot for each enemy, orange dots for med packs and other collectables, and a green dot for the player. An easy way to do this is to create spheres that will be displayed above these objects and only visible from the top-view camera. Let's start with the main character:

1. Create a new sphere.
2. Change its scale to (x=2, y=2, z=2).
3. Rename this object `dot_fpc`.
4. Locate the texture labeled **Green** by selecting **Assets | chapter4 | chapter4_pack** and apply this texture to the sphere.
5. Drag this object (`dot_fpc`) on the **First Person Controller** as illustrated in the following screenshot:

6. Change its position to (x=0, y=0, z=0).
7. This will include the sphere as a child of the first-person controller. In other words, any transformation applied to the first-person controller will be applied to the sphere. As a result, the sphere will move along with the character.

Next, we need to set the layer for this object, so that it is only displayed on the top-view camera:

1. Click on the object `dot_fpc` in the **Hierarchy** window to select it.
2. In the **Inspector** window, click on the drop-down menu to the right of the **Layer** label.
3. Select the option **topView** from the list.
4. Because we don't want the player to collide with these objects (although they will be invisible on the top view), we need to remove the colliders from this sphere. In the **Inspector** window, right-click on the component **Sphere Collider** for this object, and select the option **Remove Component** from the contextual menu. This will remove the collider from the sphere.

Finally, we will need to see this sphere regardless of the light around it and we will make it self-illuminated. This means that it will glow even in the absence of light. The following steps highlight what to do:

1. Select the object `dot_fpc`.
2. In the **Inspector** window, open the component **Mesh Renderer**, and change its **Shader** property to **Self-Illumin | Diffuse** (alternatively, using the unlit shaders may be more effective for game performance as they are less CPU/GPU resource-intense).
3. Leave the other options as default.

Play the scene; we should see a green dot in the middle of the mini-map.

We will now create the dots for the other objects:

1. Create a new sphere, change its scale to (x=2, y=2, z=2), and rename it `dot_medpack`.
2. Locate the texture labeled **Orange** by selecting **Assets | chapter4 | chapter4_pack**, and apply this texture to the sphere.
3. Change the `shader` property to `Self-Illumin/Diffuse`.
4. Remove the `SphereCollider` component from this object.
5. Drag-and-drop the object (`dot_medpack`) on the object labeled `medpack`.
6. Change the position of this object to (x=0, y=0, z=0).
7. Change its `Layer` property to `topView`.
8. Repeat the previous steps to create two other spheres named `dot_key` and `dot_gun` for both the objects labeled `key` and `gun`.

> If we look at the `scale` properties of the dot for the objects `key`, `medpack`, or `gun`, we will see that the initial values that we have entered (x=2, y=2, z=2) have changed. This is because the parent objects of these dots (that is, the objects `key`, `medpack`, and `gun`) are scaled down, which means that to preserve the aspect of the dots, and compensate for the scaling of their parents, Unity3D has adjusted their `scale` properties. However, if we combine the scale of the dots and the scale of the parents, we should find that the overall size of these dots is (x=2, y=2, z=2). For example, the object `dot_gun` has the `scale` properties (x=10, y=4, z=4), and the `scale` property of its parent is (x=0.2, y=0.5, z=0.5). If we combine these two scaling properties (x=10*0.2, y=4*0.5, z=4 *0.5) you obtain an overall sale of (x=2, y=2, z=2).

At this stage, the top-view camera displays the dots that indicate the position of the player and other items; however, it would also be great to display part of the environment, including the walls, on the mini-map. We also need to display these walls in the main view and we can only allocate one layer for this object. The solution is to create a layer labeled **topAndMain** that will be displayed by both the main camera and the top-view camera. Let's create this layer and allocate it to the views:

1. Select one of the walls in the scene.
2. In the **Inspector** window, click on the drop-down menu to the right of the label **Layer**.
3. From the drop-down menu, select the option **Add Layer**.
4. Create a layer, to the right of the label **User Layer 9**, that we will label **topAndMain**.

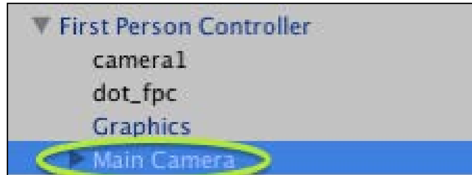
Once this step is done, as we have seen in the previous sections, we will allocate this layer to the corresponding objects:

1. Select all the walls in the level (or select them one-by-one if needed) as well as all objects labeled `block`.
2. In the **Inspector** window, click on the drop-down menu to the right of the label **Layer**.
3. From the drop-down menu, select **topAndMain**.
4. We may also apply this layer to other objects such as the rocks and platforms in the water area (that is, the objects labeled `bridge`).
5. Note that by selecting all walls at once, the new layer will be applied to all of them. Modifying the attributes of several objects at once can help us to save precious time when designing our game.

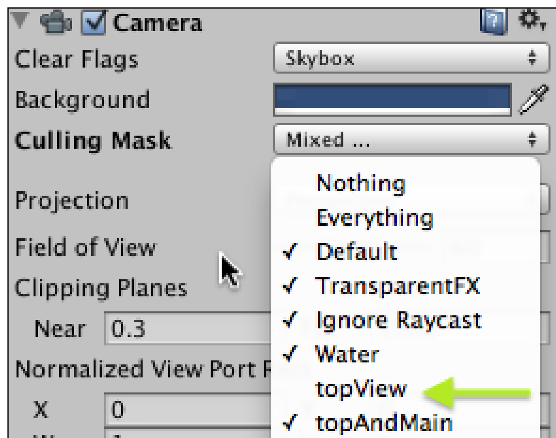
We now need to ensure that each camera will display this layer:

1. Click on `camera1`.
2. In the **Inspector** window, change the **Culling Mask** attribute of its **Camera** component so that it includes both the layers **topView** and **topAndMain**.

3. Select the camera that is within the **First Person Controller** and labeled **Main Camera** as illustrated in the following screenshot:



4. Change the **Culling Mask** attribute of its **Camera** component so that it displays everything but not the **topView** layer as illustrated in the following screenshot:



Play the scene and check the content displayed in the mini-map as highlighted in the following screenshot:

Creating a gun

Before creating the necessary script for the bullet and collision detection, we will add a crosshair in the middle of the screen to improve the player's accuracy.

1. Create a new GUI texture, and rename it `GUITexture_crossHair`.
2. Change its position to (x=0.5, y=0.5, z=0), so that it is displayed in the middle of the screen.
3. Drag-and-drop the texture labeled **Crosshair** by selecting **Assets** | **chapter4** | **chapter4_pack** to the **GUITexture** component of this object, as illustrated in the following screenshot:

4. This should display the crosshair in the middle of the screen in the game view.
5. In the **GUITexture** component of the object `GUITexture_croshair`, change the `width` and `height` properties to `128`.

Because bullets travel at a considerable speed, it may be difficult to detect when they collide with other objects. As a result, we will use a technique called ray casting to aim and fire a bullet. Put simply, using ray casting, we cast a ray from the middle of the screen forward (or any other position), the same way an infrared light could be used to aim at a target. When this ray intersects with an object, we can tell whether the virtual bullet has hit an object.

1. Create a new script by selecting **Assets** | **chapter4** | **Scripts** and rename it `shootBullet`.
2. Modify the script as described in the following code:

```
function Update ()
{
  if (Input.GetButtonUp("Fire1"))
  {
    var hit : RaycastHit;
    var ray = Camera.main.ScreenPointToRay (Vector3(Screen.
width/2,Screen.height/2));
    if(Physics.Raycast (ray, hit, 100))
    {
      print("You fired at the "+hit.collider.gameObject.tag);
    }
  }
}
```

- In line 3 of the previous code, we check whether the `Fire1` button is pressed (that is, the mouse left button).
- In line 5 of the previous code, we create a variable of type `RaycastHit`. It will be used to identify the object that intersects with the ray. This way, we will be able to identify the object hit by the bullet and perform actions accordingly (for example, apply damage).
- In line 6 of the previous code, a new ray is created. It starts from the center of the screen and points forward.
- In line 9 of the previous code, the function `Physics.Raycast` casts a ray using the three parameters `ray`, `hit`, and `100`. In our case, the ray previously created (that is, from the center of the screen) is used; it points forward and its length is 100 meters. When an object collides with this ray, its properties can be accessed through the variable `hit`.
- Line 11 of the previous code shows how we can access the tag of the object that collided with our ray. We access the collider, then the associated `gameObject`, and then the tag. Note that we could also access the exact point where the ray collided with the object using `hit.point`.
- Drag-and-drop this script on the **First Person Controller**.
- Play the scene and target one of the objects present in the scene (for example, med pack). If we fire at this object, the console should display a message, for example, **You fired at the medpack**.
- Stop the game and open the editor.

We will fine-tune the gun as follows in order to:

- Hide the mouse cursor on screen
- Play a sound when a bullet is fired
- Display and update the number of ammunitions left
- Allow the player to shoot only when there are enough ammunitions

To hide the mouse cursor, we can use the variable `Screen.showCursor`. This variable, when set to `false`, will hide the cursor.

1. Open the script `shootBullet`.
2. Add the following line within the function `Start`:
   ```
   Screen.showCursor = false;
   ```
3. Play the scene and check that the mouse cursor is hidden after the first shot.

Let's display and update the number of ammunitions left:

1. Open the script `shootBullets`.
2. Add the following line at the start of the script:

```
public var nbBullets:int;
```

3. Add the following code inside the `Start` function:

```
nbBullets = 0;
```

4. Modify the function `Update` as highlighted in the following code:

```
function Update ()
{
if (Input.GetButtonUp("Fire1"))
  {
    if (nbBullets >= 1)
    {
      var hit : RaycastHit;
      var ray = Camera.main.ScreenPointToRay (Vector3(Screen.
width/2,Screen.height/2));
      if(Physics.Raycast (ray, hit, 100))
      {
        print(hit.collider.gameObject.tag);
      }
   nbBullets--;
   print("nbBullets:"+nbBullets);
    }
  }
}
```

- In line 5 of the previous code, we check that we have at least one bullet to be able to fire the gun
- In line 13 of the previous code, since we have shot a bullet, we decrease the number of bullets that we currently have

Next, let's display the number of ammunitions (bullets) left on the screen: create a new `GUIText` object, rename it `GUIText_ammo`, change its position to (x=0.01, y=0.25), and its font size to `20`. Finally, we will update the text displayed for the number of ammunitions left:

1. Open the script `collisionDetection`.
2. Add the following line to the start of the script:

```
private var guiAmmo:GameObject;
```

3. Add the following line to the function `Start`:

```
guiAmmo = GameObject.Find("GUItext_ammo");
```

4. Add the following line of code to the function `Update`:

```
if (hasGun) guiAmmo.guiText.text= "Ammo:"+
  GetComponent(shootBullet).nbBullets;
```

In the previous line, we indirectly accessed the object `GUIText_ammo` and its `text` attribute, and set the initial text displayed.

We now need to make some adjustments: the player should only be able to fire the gun when he/she has collected the gun. As a result, we need to check whether the gun has been collected before the crosshair can be displayed or any of the bullets shot. In the previous chapters, we created a script called `collisionDetection`. Amongst other things, this script included a Boolean variable `hasGun` that was set to `true` when the player had collected the gun. We will need to test whether this variable is set to `true` before the player can use the gun. First, we will hide some of the contextual messages and textures when the game is created:

1. Create a new script called `initGame` by selecting **Assets | chapter4 | Scripts**.
2. Modify the `Start` function as follows:

```
function Start ()
{
GameObject.Find("GUIText_ammo").guiText.text="";
GameObject.Find("GUITexture_crosshair").guiTexture.enabled=false;
GameObject.Find("GUIText_displayMessageToUser").guiText.text="";
}
```

3. Add this script to the **First Person Controller**.
   - Line 3 of the previous code shows that the text that displays the number of ammunitions left is hidden
   - Line 4 of the previous code shows that the crosshair is hidden
   - Line 5 of the previous code shows that the text used to display messages to users is hidden

We now need to display the crosshair when the gun has been collected and also increase the number of bullets available to the player to `40`:

1. Open the script `collisionDetection`.

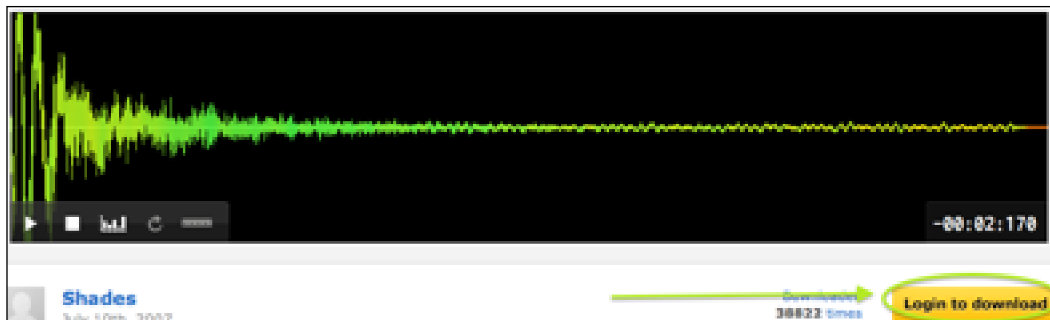
2. Add the code that is highlighted to the script in the section that detects whether the gun has been collected:

```
if (c.gameObject.tag == "gun")
{
hasGun = true; changeGUITexture(true, "gun");
GameObject.Find("GUITexture_crosshair").guiTexture.enabled=
  true;
GetComponent(shootBullet).nbBullets = 40;
}
```

- Lines 1 to 3 of the previous code were already in the script
- In line 4 of the previous code, the texture used for the crosshair is activated
- In line 5 of the previous code, we access the script `shootBullet` and set the variable `nbBullets` to `40`

Finally, we will add sound whenever the player fires a shot as shown in the following steps:

1. Open the URL `http://freesound.org/browse/tags/gun/`.
2. Select a sound of your choice for the gunshot.
3. Once you click on the name of the sound, you will be redirected to a new page with more details on the sound, and a button labeled **Login to download**. Click on this button, and register as highlighted in the following screenshot:



4. Read the terms and conditions.
5. After registering, you will receive an e-mail to activate your account.
6. Check your e-mail and activate your account accordingly.
7. Log in using your newly created account.
8. Open the page `http://freesound.org/browse/tags/gun/`.

9. Re-enter your login details.
10. Click on the button labeled **Download**.
11. Once the sound has been downloaded, import it in Unity3D (**Assets | Import New Asset**).
12. Rename this file `gunshot`.

Once we have imported this sound, we need to modify the script `shootBullet` so that a sound is played whenever a bullet is shot:

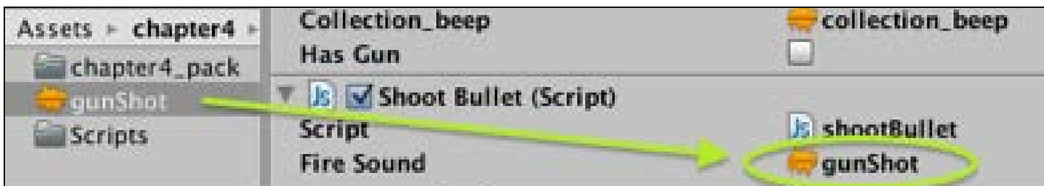
1. Open the script `shootBullet`.
2. Add the following line at the start of the script to declare a variable for the sound to be played:

```
public var fireSound:AudioClip;
```

3. Add the following code highlighted within the conditional statement that tests if we have enough bullets to fire the gun; if so, the gunshot sound can be played accordingly:

```
If (nbBullets >= 1)
{
audio.clip = fireSound;
audio.Play();
```

4. Select the object **First Person Controller** in the **Hierarchy** window.
5. Locate the component **Shoot Bullet** for this object in the **Inspector** window.
6. Drag-and-drop the sound `gunShot` to the variable `FireSound` within the component **Shoot Bullet** as described in the following screenshot:



7. Test the scene and check that the sound is played when we fire the gun.

At this stage, we can recognize what object we shoot at; however, it would be great to add a visual effect that simulates the impact of our bullet. This will be done using a particle system. Using the object `hit`, we will detect the location of the impact and generate a **particle emitter** at this particular position.

First, let's modify the script `shootBullet` to include variables for the particles:

1. Include the following line at the start of the script `shootBullet`:

    ```
    public var sparks:GameObject;
    ```

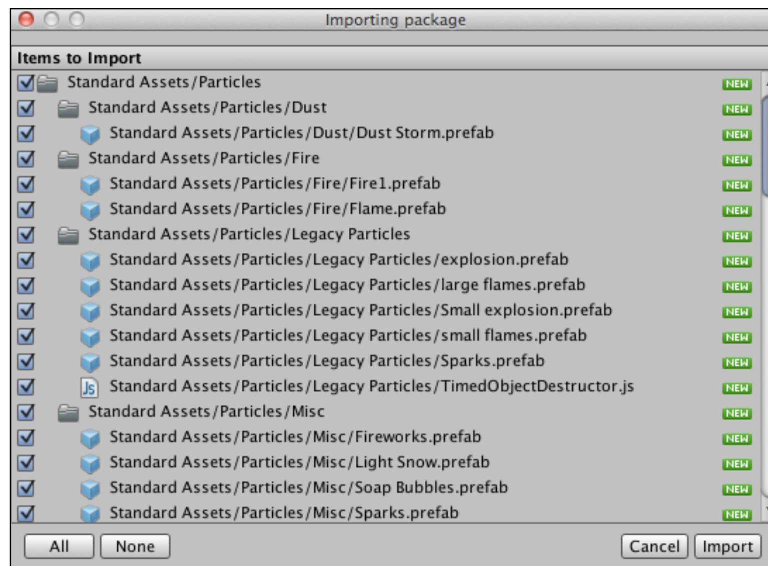
2. The variable defined in the previous line will be used as a container. It will make it possible to drag-and-drop the particle emitter to be used on impact. This particle emitter will be a prefab.
3. In the script `shootBullet`, add the following lines within the code that tests whether the ray casted from the middle of the screen has intersected with another object (that is, just after the line `print ("You fired at the "+hit.collider.gameObject.tag)`:

    ```
    var spark:GameObject = Instantiate( sparks, hit.point,
      Quaternion.identity );
    ```

4. In the previous code, we created a new spark based on the prefab mentioned earlier. The spark will be instantiated at the exact position where the ray has intersected with the object.

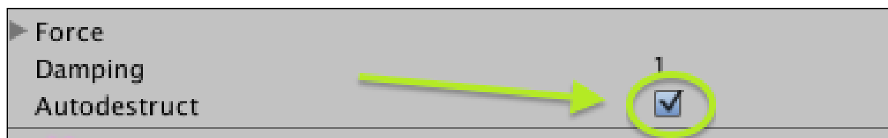
Next, we need to identify the prefab that will be used in this script (that is, instantiated). Thankfully, Unity3D includes built-in prefabs for particles, including a prefab to simulate sparks. However, this prefab needs to be imported as follows:

1. Select: **Assets** | **Import Package** | **Particles**. This should show a window labeled **Importing package**. As per previous sections, it includes all built-in particles (including legacy particles) that can be used in Unity3D.

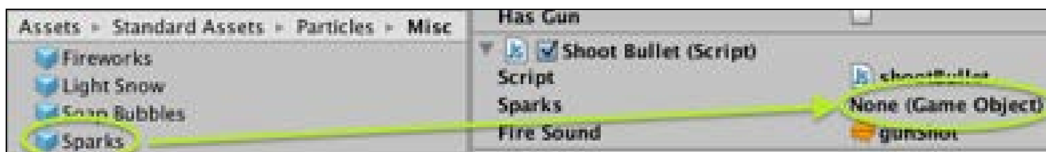


2. Click on **Import**.
3. This will create a new folder labeled `Particles` in **Assets** | **Standard Assets**, as illustrated in the following screenshot:

4. If we select: **Assets** | **Standard Assets** | **Particles** | **Misc**, we can find a prefab called **Sparks**. We will use this prefab to create the sparks.
5. Select the prefab **Sparks**.
6. In the **Inspector** window, open the **Particle Animator** component for this object, and check the box for the option **Autodestruct**.



7. Drag-and-drop the **Spark** prefab that we have identified previously to the variable called **Sparks** for the script `shootBullet`, which is a component of the **First Person Controller** object as illustrated on the following screenshot:



Test the scene; fire shots at objects, and check whether sparks appear at the point of impact.

Allowing for repeated shots

Now that we have managed to create a gun that shoots a bullet every time the player presses the left mouse button, it would be great to add an automatic feature to this gun, so that the player can fire the gun repeatedly by just holding the left mouse button down. This feature would be useful when many enemies surround the player, although it will also mean that the ammunitions will diminish faster. Let's implement this feature:

Add the following two lines to the start of the script `shootBullet`:

```
public var timeToReload:float = 0.2f;
public var timeForNextShot:float = 0.0;
```

In the previous code, we defined and initialized two variables. The first variable `timeToReload` is the time it takes to load the next bullet. Effectively, it will be the time between two consecutive shots. The second variable `timeForNextShot` is the time when the player will be able to fire the gun again (effectively, it will be the current time added to the time it takes to reload the gun).

Next, we will change the way the input is handled by replacing the line that starts with `if (Input.GetButtonDown("Fire1")` with the following code:

```
if (Input.GetButton("Fire1") && Time.time >= timeForNextShot)
```

In the previous code, we checked if the `Fire1` button is held down, and if the next bullet has been loaded. Note that while the previous code `Input.GetButtonDown` is checking whether the `Fire1` button has been pressed, the code `Input.GetButton` checks whether the `Fire1` button is held down. Finally, we need to update the variable `timeForNextShot` accordingly, as highlighted in the following code:

```
function Update ()
{
  if (Input.GetButton("Fire1") && Time.time >= timeForNextShot)
  {
    if (nbBullets >= 1)
    {
      audio.clip = fireSound;
      audio.Play();
      var hit : RaycastHit;
      var ray = Camera.main.ScreenPointToRay(Vector3 (Screen.width/2,
Screen.height/2));
      if (Physics.Raycast(ray, hit, 100))
      {
        print ("You fired at the "+hit.collider.gameObject.tag);
```

```
            var spark : GameObject = Instantiate (sparks, hit.point,
Quaternion.identity);

          }
          nbBullets --;
          print("nbBullets"+nbBullets);
        }
        timeForNextShot = Time.time + timeToReload;

    }
  }
```

Summary

In this chapter, you have learned to add more interaction into our game. First, we learned how to create a health bar and update its value and color based on the health of the player. Then, we looked into the creation of a top-down mini-map of the level. For this we used cameras, camera viewports, and layers, so that only specific objects are displayed on the map. We also added self-illuminated properties to all dots representing the objects on the map. After finalizing our interface, we created a gun, thanks to ray-casting, a technique used to detect objects within a specific range. To make this gun more precise and realistic, we added a crosshair that would only be displayed when the player has collected the gun, and sparks at the point of impact of the bullet. In the next chapter, we will add more interactivity to our game by including enemies with some levels of intelligence, and by applying damage to both the player and the enemies.

5
Bringing Your Game to Life with AI and Animations

In this chapter, we will bring the game to life by animating objects and characters, and by giving opponents some levels of artificial intelligence to challenge the player. We will also learn how to create animations in Unity3D using built-in functionalities (for example, Mecanim).

After completing this chapter, we will be able to:

- Understand how to animate objects and create custom animation based on Unity3D's built-in animations
- Include these animations in the game and control them through scripting
- Use Unity3D's built-in Mecanim system to animate existing characters

After going through these principles, we will be completing the tasks to enhance the maze game and the gameplay. We will apply animations to characters and trigger these in particular situations. Throughout this section, we will improve the gameplay by allowing NPCs to follow the player where he/she is nearby (behavior based on distance), and attack the user when he/she is within reach. All material required to complete this chapter is available for free download on the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.

> The pack for this chapter includes some great models and animations that were provided by the company Mixamo to enhance the quality of our final game. The characters were animated using Mixamo's easy online sequences and animation building tools. For more information on Mixamo and its easy-to-use 3D character rigging and animation tools, you can visit `http://www.mixamo.com`.

Before we start creating our level, we will need to rename our scene and download the necessary assets from the companion website as follows:

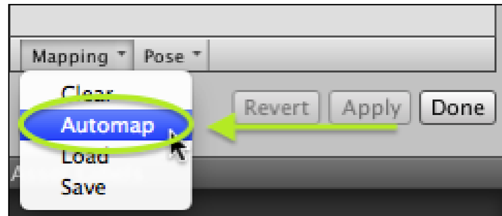
1. Duplicate the scene we have created in the previous chapter by saving the current scene (**File Save | Scene**), and then saving this scene as `chapter5` (**File | Save Scene As...**).
2. Open the link for the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.
3. Click on the link for the `chapter5` pack to download this file.
4. In Unity3D, create a new folder, `chapter5`, inside the `Assets` folder and select this folder (that is, `chapter5`).
5. From Unity, select **Assets | Import Package | Custom Package**, and import the package you have just downloaded.
6. This should create a folder, `chapter5_pack`, within the folder labeled `chapter5`.

Importing and configuring the 3D character

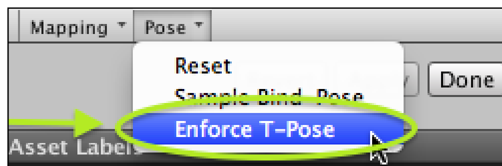
We will start by inserting and configuring the zombie character in the scene as shown in the following steps:

1. Open the Unity Assets Store window (**Window | Asset Store**).
2. In the **Search** field located in the top-right corner, type the text `zombie`.
3. Click on the search result labeled **Zombie Character Pack**, and then click on the button labeled **Import**.
4. In the new window entitled **Importing package**, uncheck the last box for the low-resolution zombie character and then click on **Import**.
5. This will import the high-resolution zombie character inside our project and create a corresponding folder labeled `ZombieCharacterPack` inside the `Assets` folder.
6. Locate the prefab `zombie_hires` by navigating to **Assets | ZombieCharacterPack**.
7. Select this prefab and open the **Inspector** window, if it is not open yet.
8. Click on the **Rig** tag, set the animation type to **humanoid**, and leave the other options as default.

9. Click on the **Apply** button and then click on the **Configure** button; a pop-up window will appear: click on **Save**.
10. In the new window, select: **Mapping** | **Automap**, as shown in the following screenshot:



11. After this step, if we check the **Hierarchy** window, we should see a hierarchy of bones for this character. Select **Pose** | **Enforce T-Pose** as shown in the following screenshot:



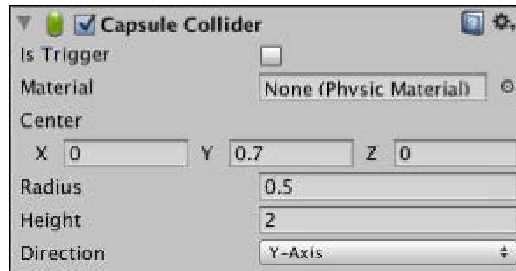
12. Click on the **Muscles** tab and then click on **Apply** in the new pop-up window.
13. The **Muscles** tab makes it possible to apply constraints on our character.
14. Check whether the mapping is correct by moving some of the sliders and ensuring that the character is represented properly. After this check, click on **Done** to go back to the previous window.

Animating the character for the game

Once we have applied these settings to the character, we will now use it for our scene.

1. Drag-and-drop the prefab labeled `zombie_hires` by navigating to **Assets** | **ZombieCharacterPack** to the scene, change its position to (x=0, y =0, z=0), and add a collider to the character.
2. Select: **Component** | **Physics** | **Capsule Collider**.

3. Set the center position of this collider to (x=0, y=0.7, z=0), the radius to `0.5`, the height to `2`, and leave the other options as default, as illustrated in the following screenshot:



4. Select: **Assets** | **chapter5** | **chapter5_pack**; you will see that it includes several animations, including `Zombie@idle`, `Zombie@walkForward`, `Zombie@attack`, `Zombie@hit`, and `Zombie@dead`.

We will now create the necessary animation for our character.

Click once on the object `zombie_hires` in the **Hierarchy** window. We should see that it includes a component called **Animator**. This component is related to the animation of the character through Mecanim. You will also notice an empty slot for an Animator Controller. This controller will be created in the next section so that we can animate the character and control its different states, using a state machine.

Let's create an **Animator Controller** that will be used for this character:

1. From the `project` folder, select the `chapter5` folder, then select **Create** | **Animator Controller** in the **Project** window. This should create a new Animator Controller labeled **New Animator Controller** in the folder `chapter5`.
2. Rename this controller `zombieController`.
3. Select the object labeled `zombie_hires` in the **Hierarchy** window.
4. Locate the **Animator Controller** that we have just created by navigating to **Assets** | **chapter5** (`zombieController`), drag-and-drop it to the empty slot to the right of the attribute controller in the Animator component of the zombie character, and check that the options **Apply Root Motion** and **Animate Physics** are selected. Our character is now ready to receive the animations.
5. Open the **Animator** window (**Window** | **Animator**). This window is employed to display and manage the different states of our character. Since no animation is linked to the character, the default state is **Any State**.
6. Select the object labeled `zombie_hires` in the **Hierarchy** window.

7. Rearrange the windows in our project so that we can see both the state machine window and the character in the **Scene** view: we can drag the tab labeled **Scene** for the **Scene** view at the bottom of the **Animator** window, so that both windows can be seen simultaneously.

We will now apply our first animation to the character:

1. Locate the prefab `Zombie@idle` by navigating to **Assets** | **chapter5** | **chapter5_pack**.
2. Click once on this prefab, and in the **Inspector** window, click the **Rig** tab.
3. In the new window, select the option **Humanoid** for the attribute **Animation Type** and click on **Apply**.
4. Click on the **Animations** tab, and then click on the label **idle**, this will provide information on the **idle** clip.
5. Scroll down the window, check the box for the attribute **Loop Pose**, and click on **Apply** to apply this change (you will need to scroll down to locate this button).
6. In the **Project** view, click on the arrow located to the left (or right, depending on how much we have zoomed-in within this window) of the prefab `Zombie@idle`; it will reveal items included in this prefab, including an animation called **idle**, symbolized by a gray box with a white triangle.
7. Make sure that the **Animator** window is active and drag this animation (**idle**) to the **Animator** window.
8. This will create an idle state, and this state will be colored in orange, which means that it is the default state for our character. Rename this state `Idle` (upper case I) using the Inspector.
9. Play the scene and check that the character is in an idle state.
10. Repeat steps 1-9 for the prefab `Zombie@walkForward` and create a state called `WalkForward`. To test the second animation, we can temporarily set the state `walkForward` to be the default state by right-clicking on the `walkForward` state in the **Animator** window, and selecting **Set As Default**. Once we have tested this animation, set the state `Idle` as the default state.

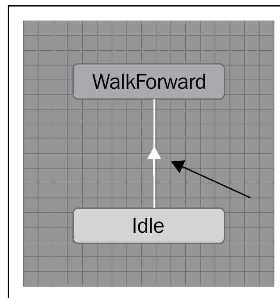
While the zombie is animated properly, you may notice that the camera on the **First Person Controller** might be too high. You will address this by changing the height of the camera so that it is at eye-level. In the **Hierarchy** view, select the object **Main Camera** that is located with the object **First Person Controller** and change its position to (x=0, y=0.5, z=0).

We now have two animations. At present, the character is in the `Idle` state, and we need to define triggers or conditions for the character to start or stop walking toward the player. In this game, we will have enemies with different degrees of intelligence. This first type will follow the user when it sees the user, is close to the user, or is being attacked by the user.

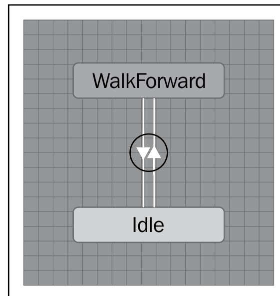
The **Animator** window will help to create animations and to apply transition conditions and blending between them so that transitions between each animation are smoother. To move around this window, we can hold the *Alt* key while simultaneously dragging-and-dropping the mouse. We can also select states by clicking on them or defining a selection area (drag-and-drop the mouse to define the area). If needed, it is also possible to maximize this window using the icon located at its top-right corner.

Creating parameters and transitions

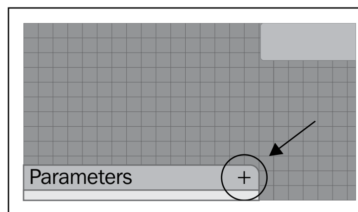
First, let's create transitions. Open the **Animator** window, right-click on the state labeled **Idle**, and select the option **Make Transition** from the contextual menu. This will create an arrow that symbolizes the transition from this state to another state. While this arrow is visible, click on the state labeled **WalkForward**. This will create a transition between the states **WalkForward** and **Idle** as illustrated in the following screenshot:



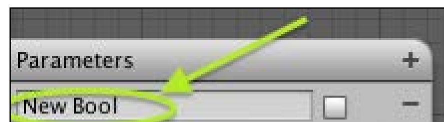
Repeat the last step to create a transition between the state **WalkForward** and **Idle**: right-click on the state labeled **WalkForward**, select the option **Make Transition** from the contextual menu, and click on the state labeled **Idle**.



Now that these transitions have been defined, we will need to specify how the animations will change from one state to the other. This will be achieved using parameters. In the **Animator** window, click on the **+** button located at the bottom-right corner of the window, as indicated in the following screenshot:

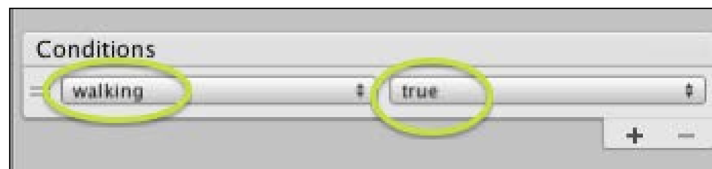


Doing so will display a contextual menu, from which we can choose the type of the parameter. Select the option **Bool** to create a Boolean parameter. A new window should now appear with a default name for our new parameter as illustrated in the following screenshot: change the name of the parameter to `walking`.



Now that the parameter has been defined, we can start defining transitions based on this parameter. Let's start with the first transition from the **Idle** state to the **Walkforward** state:

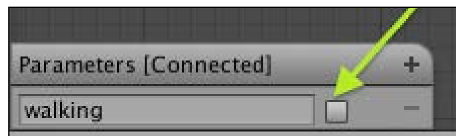
1. Select the transition from the **Idle** state to the **Walkforward** state (that is, click on the corresponding transition in the **Animator** window).
2. If we look at the **Inspector** window, we can see that this object has several components, including **Transitions** and **Conditions**. Let's focus on the **Conditions** component for the time being. We can see that the condition for the transition is based on a parameter called `ExitTime` and that the value is `0.98`. This means that the transition will occur when the current animation has reached 98 percent completion. However, we would like to use the parameter labeled **walking** instead.
3. Click on the parameter `ExitTime`, this should display other parameters that we can use for this transition.
4. Select **walking** from the contextual menu and make sure that the condition is set to **true** as shown in the following screenshot:



The process will be similar for the other transition (that is, from **WalkForward** to **Idle**), except that the condition for the transition for the parameter **walking** will be **false**: select the second transition (**WalkForward** to **Idle**) and set the transition condition of **walking** to **false**.

To check that the transitions are working, we can do the following:

1. Play the scene and look at the **Scene** view (not the Game view).
2. In the **Animator** window, change the parameter walking to **true** by checking the corresponding box, as highlighted in the following screenshot:



3. Check that the zombie character starts walking; click on this box again to set the variable **walking** to **false**, check that the zombie stops walking, and stop the **Play** mode (*Ctrl* + *P*).

Adding basic AI to enemies

In the previous section, we have managed to set transitions for the animations and the state of the zombie from **Idle** to **walking**. To add some challenge to the game, we will equip this enemy with some AI and create a script that changes the state of the enemy from **Idle** to **WalkForward** whenever it sees the player. First, let's allocate the predefined-tag player to **First Person Controller**: select **First Person Controller** from the **Hierarchy** window, and in the **Inspector** window, click on the drop-down menu to the right of the label **Tag** and select the tag **Player**.

Then, we can start creating a script that will set the direction of the zombie toward the player. Create a folder labeled `Scripts` inside the folder **Assets | chapter5**, create a new script, rename it `controlZombie`, and add the following code to the start of the script:

```
public var walking:boolean = false;
public var anim:Animator;
public var  currentBaseState:AnimatorStateInfo;
public var walkForwardState:int = Animator.StringToHash("Base
  Layer.WalkForward");
public var idleState:int = Animator.StringToHash("Base
  Layer.Idle");
private var playerTransform:Transform;
private var hit:RaycastHit;
```

- In statement 1 of the previous code, a Boolean value is created. It is linked to the parameter used for the animation in the **Animator** window.
- In statement 2 of the previous code, we define an **Animator** object that will be used to manage the animator component of the zombie character.
- In statement 3 of the previous code, we create an `AnimatorStateInfo` variable that will be used to determine the current state of the animation (for example, **Idle** or **WalkForward**).
- In statement 4 of the previous code, we create a variable, `walkForwardState`, that will represent the state `WalkForward` previously defined in the **Animator** window. We use the method `Animator.StringToHash` to convert this state initially from a string to an integer that can then be used to monitor the active state.

- In statement 5 of the previous code, similar to the previous comments, a variable is created for the state `Idle`.
- In statement 6 of the previous code, we create a variable that will be used to detect the position of the player.
- In statement 7 of the previous code, we create a ray that will be employed later on to detect the player.

Next, let's add the following function to the script:

```
function Start ()
{
  anim = GetComponent(Animator);
  playerTransform = GameObject.FindWithTag("Player").transform;
}
```

In line 3 of the previous code, we initialize the variable `anim` with the `Animator` component linked to this `GameObject`.

We can then add the following lines of code:

```
function Update ()
{
  currentBaseState = anim.GetCurrentAnimatorStateInfo(0);
  gameObject.transform.LookAt(playerTransform);
}
```

- In line 3 of the previous code, we determine the current state for our animation.
- In line 4 of the previous code, the transform component of the current game object is oriented so that it is looking at the **First Person Controller**. Therefore, when the zombie is walking, it will follow the player.

Save this script, and drag-and-drop it to the character labeled `zombie_hires` in the **Hierarchy** window.

As we have seen previously, we will need to manage several states through our script, including the states **Idle** and **WalkForward**. Let's add the following code in the `Update` function:

```
switch (currentBaseState.nameHash)
{
case idleState:
break;

case walkForwardState:
```

```
break;

default:
break;
}
```

- In line 1 of the previous code, depending on the current state, we will switch to a different set of instructions
- All code related to the state `Idle` will be included within lines 3-4 of the previous code
- All code related to the state `WalkForward` will be included within lines 6-7

If we play the scene, we may notice that the zombie rotates around the x and z axes when near the player; its y position also changes over time. To correct this issue, let's add the following code at the end of the function `Update`:

```
transform.position.y = -0.5;
transform.rotation.x = 0.0;
transform.rotation.z = 0.0;
```

We now need to detect whether the zombie can see the player, or detect its presence within a radius of two meters (that is, the zombie would hear the player if he/she is within two meters). This can be achieved using two techniques: by calculating the distance between the zombie and the player, and by casting a ray from the zombie and detecting whether the player is in front of the zombie. If this is the case, the zombie will start walking toward the player. We need to calculate the distance between the player and the zombie by adding the following code to the script, `controlZombie`, at the start of the function `Update`, before the `switch` statement:

```
var distance:float = Vector3.Distance(transform.position,
  playerTransform.position);
```

In the previous code, we create a variable labeled `distance` and initialize it with the distance between the player and the zombie. This is achieved using the built-in function `Vector3.Distance`.

Now that the distance is calculated (and updated in every frame), we can implement the code that will serve to detect whether the player is near or in front of the zombie.

Open the script entitled `controlZombie`, and add the following lines to the function `Update` within the block of instructions for the `Idle` state, so that it looks as follows:

```
case idleState:
  if ((Physics.Raycast
  (Vector3(transform.position.x,transform.position.y+.5,transform.po
    sition.z), transform.forward, hit,40) &&
      hit.collider.gameObject.tag == "Player") || distance <2.0f)
 {
    anim.SetBool("walking",true);
 }
break;
```

In the previous lines of code, a ray or ray cast is created. It is casted forward from the zombie, 0.5 meters above the ground and over 40 meters. Thanks to the variable `hit`, we read the tag of the object that is colliding with our ray and check whether this object is the player. If this is the case, the parameter `walking` is set to `true`. Effectively, this should trigger a transition to the state `walking`, as we have defined previously, so that the zombie starts walking toward the player.

Initially, our code was written so that the zombie rotated around to face the player, even in the `Idle` state (using the built-in function `LookAt`). However, we need to modify this feature so that the zombie only turns around to face the player while it is following the player, otherwise, the player will always be in sight and the zombie will always see him/her, even in the `Idle` state. We can achieve this by deleting the code highlighted in the following code snippet (from the start of the function `Update`), and adding it to the code for the state `WalkForward`:

```
case walkForwardState:
  transform.LookAt(playerTransform);
  break;
```

In the previous lines, we checked whether the zombie is walking forward, and if this is the case, the zombie will rotate in order to look at and follow the player. Test our code by playing the scene and either moving within two meters of the zombie or in front of the zombie.

Sending messages to alert other close enemies

Now that we have created a relatively simple behavior for each enemy, we can add an additional AI feature. We will modify our code so that if there are several enemies in a similar location, those that have not detected the player will be alerted of its presence by other enemies, and also start walking toward the player. First, we will create a tag for each enemy. This will make it easier to identify any enemy within range:

1. Select the object `zombie_hires` from the **Hierarchy** window; in the **Inspector** window, click the on the drop-down menu to the right of the label **Tag** and click on the option **Add Tag**.
2. Select the line from the last tag element (for example, **Element 7**), and type the word `zombie`. This will create a new tag; once this is done, select our object (`zombie_hires`).
3. In the **Inspector** window, click on the drop-down menu to the right of the label **Tag**.
4. Select the label **zombie** that we have just created.

Next, we will create a function that will be accessible from other objects and that will change the value of the parameter `walking` for the zombie animation. By changing this parameter to `true`, we will be able to change the state of any zombie, and in our case, make them walk toward the player. Let's add the following code to the script `controlZombie`:

```
function setWalking(newWalkingValue: boolean)
{
anim.SetBool("walking",newWalkingValue);
}
```

In the previous code, we declare a function labeled `setWalking`. This function takes one parameter (`newWalkingValue`) that will be used to trigger a transition between the states `Idle` and `WalkForward`. We set the value of the parameter `walking` in the animation for the zombie using the built-in function `anim.SetBool`.

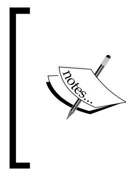
Then, we need to detect whether there are any zombies around the one that has detected the player. We will achieve this by adding the following code within the function `Update` in the script `controlZombie`, within the code dedicated to the state `walkForward`, as highlighted in the following code snippet:

```
case:walkForwardState
  var zombies:GameObject [] = GameObject.FindGameObjectsWithTag("zomb
ie");
  for (var zombie:GameObject in zombies)
  {
    if (Vector3.Distance(transform.position, zombie.transform.
position) < 8.0f)
zombie.GetComponent(controlZombie).setWalking(true);
  }
  break;
```

In the previous code, we create an array of objects. This array will be populated with all zombies' objects in our game. We loop through this array, and assess the distance between each of these objects and the current zombie. Any zombie within a distance of 8 meters from the zombie that has detected the player will also start to walk toward the player. To test this behavior, we could do the following:

1. Duplicate the zombie character twice to obtain a total of 3. Change the position of the two new zombies to (x=-2, y=-0.5, z=0) and (x=-4, y=-0.5, z=0), and rename them `zombie_hires2` and `zombie_hires3`, respectively.
2. Rotate the first two instances so that they look away from the player.
3. Rotate the third instance so that it is facing the player.
4. Play the scene and move the player so that it is in front of the third instance (that is, so that it can be seen). We should see that this zombie starts to follow the player as well as the other zombies, although these are not directly facing the player.

The script could be improved by also checking that the message is sent to an enemy only if it is not already walking toward the player.



Note that the method `FindGameOBjectsWithTag` can become computer intensive if our game includes many zombie characters. We may instead detect other objects using the built-in method `Physics.OverlapSphere` that makes it possible to detect colliders within a specific radius.

Once we have checked that this behavior works, we can delete or deactivate the two copies of the zombie.

Creating additional states

At this stage, we have created an interesting, yet simple, artificial behavior, whereby the enemies present in the maze will, if they see the character, or are within a specific radius, walk toward the player and alert all other enemies in this area. We will now include an additional state that we will call `Attack`, which will be triggered when these enemies are within reach of the player:

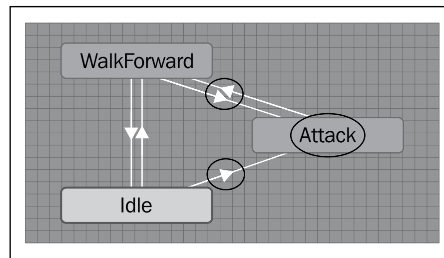
1. Locate the prefab `Zombie@attack` by navigating to **Assets** | **chapter5** | **chapter5_pack**.
2. Click once on this prefab, and in the **Inspector** window, click on the **Rig** tab.
3. In the new window, select the option **Humanoid** for the attribute **Animation Type** and click on **Apply**.
4. Click on the **Animations** tab, and then click on the label **attack**, this will provide information on the attack clip.
5. Scroll down the window, check the box for the attribute **Loop Pose** and click on **Apply** to apply this change.
6. In the **Project** view, click on the arrow located to the left of the prefab `Zombie@attack` it will reveal items included in this prefab, including an animation called `attack` symbolized by gray box with a white triangle.
7. Check that the **Animator** window is open and drag the animation attack to the **Animator** window.
8. This will create an attack state; rename this state **Attack** (upper case A) using the Inspector.
9. Check that the idle state is the default state

Once this is done, we will create transitions between these states:

1. Create a transition from the state **WalkForward** to the state **Attack**.
2. Create a transition from the state **Idle** to the state **Attack**.
3. Create a new Boolean parameter called `withinReach`.
4. Select the transition between the states **WalkForward** and **Attack**.
5. In the **Inspector** window, set the condition for the transition to `withinReach = true`, and leave other attributes as default.
6. Select the transition between the states **Idle** and **Attack**.
7. In the **Inspector** window, set the condition for the transition to `withinReach = true`, and leave other attributes as default.

8. Create a transition between the state **Attack** and the state **WalkForward**.
9. Set the conditions for the transition to `withinReach = false` and `ExitTime = 0.70` (to include an additional condition we can click on the **+** button in the same window).

The **Animator** window should now include three states and two transitions to the state **Attack** as highlighted in the following screenshot:



We have set up the animations in the **Animator** window by defining states and transitions conditions; we now need to trigger these states within the script. Ideally, we would like the attack to be triggered when the enemies are within approximately 1.5 meter from the player. Let's modify the `controlZombie` script to implement this behavior:

Open the script `controlZombie` and add the following lines of code to the function `Update` just after the code that calculates the distance:

```
if (distance < 1.5f) anim.SetBool("withinReach",true);
 else anim.SetBool("withinReach",false);
```

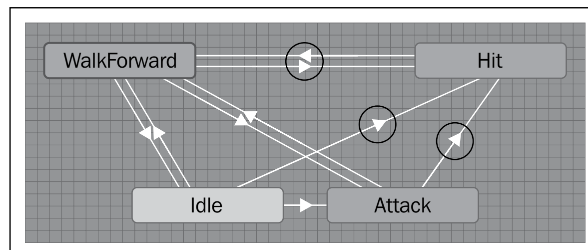
The variable `distance` has already been defined in this script to detect the distance between the player and the enemies. In the previous code, we use this variable to determine whether the player is within reach, so that the attack can be perpetrated. If the distance is less than 1.5 meters, the parameter `withinReach` is set to `true` (and the state should change accordingly to **Attack**). If the distance is more than 1.5 meters, then the animator parameter `withinReach` is set to `false`, and the state should change accordingly to `WalkForward`.

Finally, we will create two additional states: a state when a bullet hits an enemy and a state when the enemy dies following significant injuries. Following the steps and instructions described previously, create a state called **Hit**, based on the prefab `Zombie@hit`. We don't need to loop this animation.

We then need to create transitions to the state **Hit** from the states **WalkForward**, **Idle**, or **Attack**:

1. Create three transitions: from the state **WalkForward** to the state **Hit**, from the state **Idle** to the state **Hit**, and from the state **Attack** to the state **Hit**.
2. Create a new Boolean parameter called `hit`, select the transition from the state **Idle** to the state **Hit**, and set the condition for the transition to `hit = true`, and leave the other attributes as default.
3. Repeat the last step for the transition from the state **WalkForward** to **Hit**, and from the state **Attack** to **Hit**.
4. Finally, create a transition between the state **Hit** and the state **WalkForward**, and set the transition condition to `Exit Time = 0.9`. In this case, when the enemy is hit, it will start walking toward the user, regardless of its previous state (that is, `idle`, `walking`, or `attack`).

The following figure highlights the new state and transitions that we have just created:



Finally, to be able to trigger these states, we will need to modify the script that fires bullets. If the bullet hits the enemy, then its state will change accordingly. Open the script `shootBullet` and add the following code within the function `Update`, inside the conditional statement that starts with `if(Physics.Raycast (ray, hit, 100))`.

```
if (hit.collider.gameObject.tag == "zombie")
{
  hit.collider.gameObject.GetComponent(Animator).SetBool("hit",true)
  ;
}
```

- In statement 1 of the previous code, we check whether the bullet has collided with an enemy
- In statement 2 of the previous code, if this is the case, the Boolean parameter labeled `hit` is set to `true` for the corresponding animator

We have created all necessary transitions; if we test our scene, we can see that the enemy, when hit by a bullet, transitions indefinitely between the states **Hit** and **WalkForward**. This is because the Boolean variable `hit` is set to `true` all the time, causing the transition from the state **WalkForward** to the state **Hit** to occur indefinitely. To fix this, we need to set the variable `hit` to `false` once the transition has occurred from the state **Idle**, **Attack**, or **WalkForward** to the state **Hit**. This can be achieved by setting the variable `hit` to `false` when the enemy is in the state **Hit**. We can do this through script by adding the following line at the start of the script `controlZombie`:

```
var HitState:int = Animator.StringToHash("Base Layer.Hit");
```

The previous line of code declares a state for the animator, so that we can monitor the state **Hit**. Let's add the following lines in the function `Update`, within the `switch` structure:

```
case hitState:
  anim.SetBool("hit",false);
  break;
```

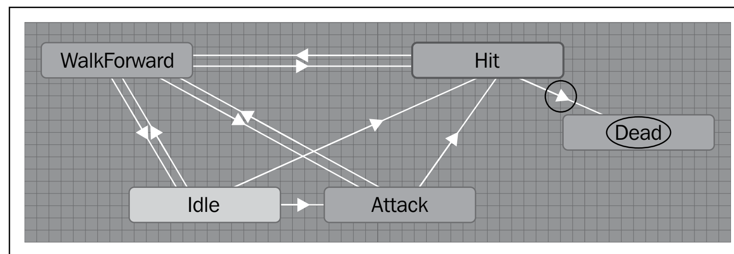
- In line 1 of the previous code, we test whether the enemy is in the state called **Hit**
- In line 3 of the previous code, if this is the case, we set the Boolean variable `hit` to `false`

Test the game, shoot at the zombie in the scene, and check that it behaves as expected (that is, following the player after being hit).

We now need to add a final state to our enemy, the state called **Dead**. The enemy will enter this state when it has sustained significant injuries. The zombie should not be able to transition to any other states from this state.

1. Following the steps described in the previous pages, create a state called **Dead**, based on the prefab `Zombie@dead`. This animation does not need to loop.
2. Create a new Boolean parameter called `die`.
3. Create a transition from the state **Hit** to the state **Dead**.
4. Set the condition for this transition to `die = true`.

The state machine should now look as illustrated in the following screenshot (the new transition and state have been highlighted with a circle).



We will manage this state through JavaScript and create the necessary code to increase damage to the enemy when it has been hit. Add the following two lines to the start of the script `controlZombie`:

```
public var damage:int;
public var DeadState:int = Animator.StringToHash("Base
  Layer.Dead");
```

- In statement 1 of the previous code, we declare a new integer variable called `damage`, that will be used to keep track of the damage inflicted to the enemy
- In statement 2 of the previous code, we declare a new state for the animator so that we can detect when the enemy is in the **Dead** state

Also, add the following line within the `Start` function to initialize the variable damage:

```
damage = 0;
```

Next, we need to apply damage to the zombie, every time it is being hit by a bullet. This can be done by adding the following code in the function `Update` within the `switch` structure:

```
case hitState:
  if (anim.GetBool("hit")) damage++;
  if (damage >=5) anim.SetBool("die",true);
  anim.SetBool("hit",false);
  break;
```

- In line 3 of the previous code, if the enemy has just been hit, then the damage is increased
- In line 4 of the previous code, if this damage is `5` or more, then the Boolean parameter `die` for the animation is set to `true`
- In line 5 of the previous code, the parameter `hit` is set to `false` so that the damage is not increased continuously while the animation is played

Last but not least, we need to assess the damage caused by the enemy on the player every time it is attacking the player. Add the following code to the start of the script `controlZombie`:

```
var attackState:int = Animator.StringToHash("Base Layer.Attack");
```

In the previous line of code, we declared a new state for the animator, so that we can detect when the enemy is in the state **Attack**.

Now that we have managed to create the corresponding state, we need to decrease the player's health when it is attacked.

First, let's create a new function in the script `healthBar` to decrease the player's health:

```
function decreaseHealth (increment : int)
{
  currHealth -= increment;
{
```

Then, let's create a new function in the script `controlZombie`:

```
function applyDamage()
{
GameObject.Find("healthBar").GetComponent(healthBar).SendMessage("
  decreaseHealth",5);
yield WaitForSeconds(3);
}
```

Finally, we can add the following line of code in the script `controlZombie` in the function `Update`, within the `switch` structure:

```
case attackState:
  applyDamage();
  anim.SetBool("withinReach",false);
  break;
```

- In line 1 of the previous code, we check whether the enemy is in the state called **Attack**.
- In line 2 of the previous code, if this is the case, we decrease the health of the player by `5`. This is done by calling the function `decreaseHealth` that is within the script `healthBar`. This function, as we have seen previously, takes one parameter that is the amount by which the health should be decreased.

Test the scene and check that the health of the character decreases as enemies are attacking the player. As we test the scene, we will notice that the health of the player drops to `0` after the first attack, whereas it should decrease progressively by `5` after each attack. This is because the health is decreased constantly as the attack animation is being played. As a result, we need to create a Boolean variable that will help us to ensure that the energy is decreased only once per attack.

Add the following line at the start of the script `controlZombie`:

```
public var hasAttacked:boolean = false;
```

Add the following code at the start of the code dedicated to the state `walkForwardState` as highlighted in the following code snippet:

```
case walkForwardState:
  hasAttacked = false;
```

Modify the code related to the state **Attack** as highlighted in the following code snippet:

```
case attackState:
  if (!hasAttacked)
  {
    applyDamage();
    hasAttacked = true;
  }
  anim.SetBool("withinReach",false);
  break;
```

Finally, we need to destroy the zombie a few seconds after it has entered the state **Dead**. This can be done by adding the following code in the function `Update` within the `switch` structure:

```
case deadState:
  Destroy(gameObject, 3.0);
  break;
```

- In line 1 of the previous code, we check whether the enemy has entered the **Dead** state
- In line 2 of the previous code, the zombie is destroyed after 3 seconds

Test the scene and check that the health of the player decreases progressively after each attack. Also, shoot at the zombie more than five times, and check that it disappears within three seconds.

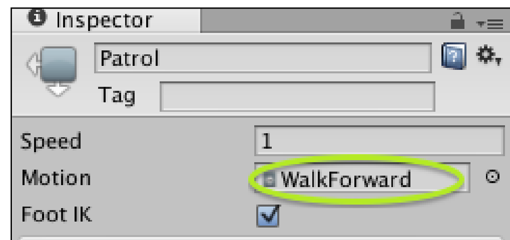
Once this is working, we will create a prefab from this enemy so that it can be duplicated later:

1. Select the folder **Assets** | **chapter5** in the **Project** window.
2. From the **Project** window, select **Create** | **Prefab**. This will create a new prefab.
3. Drag-and-drop the object labeled `zombie_hires` from the **Hierarchy** window on this prefab.
4. Rename this prefab `staticEnemy`.

Using waypoints to define a path

We will now create a new type of enemy that will patrol the maze. This character will navigate on a predefined path delimited by waypoints:

1. Duplicate the animator labeled `zombieController` from the **Assets** | **chapter5** by selecting this object and then navigating to **Edit** | **Duplicate**, and rename it `zombiePatrolController`.
2. Double-click on this animator so that it opens in the **Animator** window.
3. Rename the state **Idle** to **Patrol**.
4. Locate the animation `WalkForward` by selecting **Assets** | **chapter5** | **chapter5_pack**, within the prefab `Zombie@walkForward` (its icon is a white triangle within a gray box), or search it using the **Search** field in the **Project** window. Drag-and-drop it to the variable `Motion` for the state **Patrol** as highlighted in the following screenshot:



At this stage, we have created a new default state for our new type of enemy. This enemy will be moving based on waypoints. We will create four waypoints and determine, using scripting, which waypoint the zombie patroller should walk toward. Open the script `controlZombie` and add the next lines to the start of the script:

```
public var patrolState:int = Animator.StringToHash("Base
  Layer.Patrol");
private var wayPointIndex:int = 1;
```

- In statement 1 of the previous code, we create a new variable to monitor the state **Patrol**
- In statement 2 of the previous code, we create an index that will be used to determine the next way point to walk forward

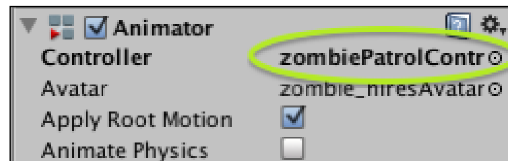
Next, let's add the following lines inside the function `Update` within the `switch` structure:

```
case patrolState:
  transform.LookAt(GameObject.Find("wayPoint"+wayPointIndex).transfo
  rm);
  var distanceToWayPoint:float =
  Vector3.Distance(transform.position,
  GameObject.Find("wayPoint"+wayPointIndex).transform.position);
  if ( distanceToWayPoint< 1.0f) wayPointIndex++;
  if (wayPointIndex > 4) wayPointIndex = 1;
break;
```

In the previous code, we check whether the state `Patrol` is active. The character looks in the direction of the next waypoint. We then calculate the distance between the zombie and the next waypoint; locate the corresponding waypoint, and if the next waypoint is close enough, a new waypoint is defined.

Last but not least, we need to create these waypoints, place them on the scene, and link them to the script as shown in the following steps:

1. Create an empty object and rename it `wayPoint1`.
2. Duplicate this object three times, and rename the copies `wayPoint2`, `wayPoint3`, and `wayPoint4`.
3. Change the positions of these waypoints to (x=-22, y=0, z=7), (x=-21, y=0, z=22), (x=-7, y=0, z=22), and (x=-7, y=0, z=7).
4. Duplicate the object `zombie_hires` and rename the duplicate patroller.
5. Change the position of this object patroller to (x=-14, y=-0.5, z=7).
6. Drag-and-drop the animator `zombiePatrolController` from **Assets | chapter5** to the **Animator** component of the object patroller as highlighted in the following screenshot:



Test the scene and check that the object patroller walks on the path determined by our waypoints. Note that we could create more waypoints if necessary, and the process would be similar to the one already described in this section.

Finally, we will create a prefab from this patroller object: select the folder **Assets | chapter5**; then navigate to **Create | Prefab** from the **Project** window, drag-and-drop the object labeled `patroller` on this prefab, and rename it `zombie_patroller`.

Summary

While this chapter has introduced basic AI principles, we can of course enhance our level by applying more complex behaviors and levels of intelligence. This can be done by using AI algorithms such as `A*` or other path-finding techniques, or by using dedicated libraries available from the assets store. Note that Unity3D includes a built-in path-finding feature referred as **Mesh Navigation**; however, this feature is only available and applicable in the Pro-version of Unity3D. Finally, while waypoints were employed to move the character along a path, you may find it useful to employ the library called `iTween`. This library, available for free in the assets store, is relatively easy to use and includes several interesting features that could definitely improve our game.

In this chapter, we have learned to apply animations to our character as well as some levels of artificial intelligence. We created different states and associated transitions using the **Animator** window, and we have also managed to monitor and trigger these states through scripting. Finally, we have used waypoints to define a path for one of the enemies. In the next chapter, we will build on these skills to finish the game: add a last level, a splash screen, and a menu system, as well as features that make it possible to save the score and other properties of the game across scenes.

6
Finalizing and Optimizing Your Game

In this chapter, we will finalize and optimize our game. After completing this chapter, we will be able to:

- Detect the current scene and load scenes
- Create a menu system and a splash screen for the game
- Improve the AI by adding breadcrumbing techniques
- Preserve and use data across levels
- Instantiate objects (for example, ammunitions or med packs)

In this chapter, we will add the ability for the zombies to follow the player using a technique called breadcrumbing. We will also create menus for the different stages of the game (that is, splash screen, instructions, or game over), and learn how to navigate through them. This chapter will also include information on how to keep data across levels so that information on the player is kept, even when the level is reloaded.

All material required to complete this chapter is available for free to download on the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.

Before we start creating our level, you will need to download the necessary assets from the companion website as follows:

1. Open the link for the companion website: `http://patrickfelicia.wordpress.com/publications/books/unity-outbreak/`.
2. Click on the link for the `chapter6` package, this will download a Unity package called `chapter6_pack`.

3. In Unity3D, create a new folder called `chapter6`, inside the `Assets` folder, and select this folder (`chapter6`).
4. Import the package that you have just downloaded into Unty3D. From Unity, select: **Assets | Import Package | Custom Package**.
5. This should create a folder labeled `chapter6_pack` within the folder labeled `chapter6`.

As for the previous chapters, we will save our current scene (**File | Save Scene**) and then rename it `chapter6` (**File | Save Scene as**).

Improving the AI using breadcrumbing

In this section, we will improve the AI for the enemies by implementing an effective, yet simple, technique called breadcrumbing. At present, while the enemies follow the player when he/she is in sight, they may stop progressing toward the player when they lose sight of the player. To add more realism, we will design an improved AI behavior whereby enemies are able to go back to their initial location after losing sight of the player, or follow the player, despite not seeing him/her. To do so, we ensure that the zombie is dropping crumbs while progressing toward the player, and that it then follows the breadcrumbs to find its way back to its initial position when it has lost sight of the player.

Allowing enemies to throw and follow their own breadcrumbs

First, let's modify the script `controlZombie`, and add the following lines at the start of the script:

```
public var breadCrumb:GameObject;
private var timeForNextCrumb:float;
private var currentTime:float;
private var breadCrumbs = new Array();
private var breadCrumbIndex:int;
```

- In line 1 of the previous code, we create a placeholder for the breadcrumb that will be dropped by the zombie
- In lines 2-3 of the previous code, these variables will be employed to determine the next time a breadcrumb will be dropped

- In line 4 of the previous code, this variable represents an array of all the breadcrumbs dropped for this zombie
- In line 5 of the previous code, this variable represents the current index for the breadcrumbs dropped

Add the following line of code to the function `Start` to initialize the variable `breadCrumbIndex`:

```
breadCrumbIndex = 0;
```

In the function `Update`, identify the `switch` case section related to the state `WalkForward` and add the following lines of code within:

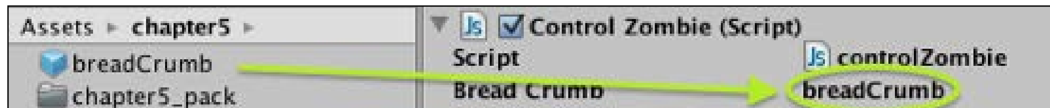
```
anim.SetBool("canSeePlayerWhileWalking",true);
if (Time.time > timeForNextCrumb)
{
  breadCrumbIndex++;
  timeForNextCrumb = Time.time+.5;
  breadCrumbs[breadCrumbIndex] = transform.position;
  var b:GameObject  =
  GameObject.Instantiate(breadCrumb,transform.position,
    transform.rotation);
  b.name = "breadcrumb_"+ name+"_"+breadCrumbIndex;
}
if (Physics.Raycast (Vector3(transform.position.x,
  transform.position.y+.5, transform.position.z),
    transform.forward, hit, 40) && hit.collider.gameObject.tag !=
      "Player") anim.SetBool("canSeePlayerWhileWalking",false);
```

- In statement 1 of the previous code, we set the variable `canSeePlayerWhileWalking` to `true`.
- In statement 2 of the previous code, we check whether we have reached the time to throw another breadcrumb.
- In statement 4 of the previous code, `breadCrumbIndex` is incremented by 1.
- Statement 5 of the previous code indicates the next time to throw a crumb (that is, `timeForNextCrumb`) is set to the current time, plus 500 milliseconds. Effectively, we will drop a breadcrumb every 500 milliseconds.
- In statement 6 of the previous code, the position of the new breadcrumb is saved (it is the same as the position of the zombie).
- In statement 7 of the previous code, we instantiate a new object (that is, a breadcrumb prefab that we will create in the next section) at this position.

- In Statement 8 of the previous code, the breadcrumb is given a name that will make it easier to identify it later on. The name includes the name of the zombie as well as an index (the name of the zombie `GameObject` is part of its properties and is therefore accessible through any script attached to this `GameObject`). This will be useful since all zombies will use this breadcrumbing feature, and we need to make sure that each of them follows their own breadcrumb (that is, in other words, their breadcrumb if uniquely identifiable).
- Line 10 of the previous code shows how if the zombie loses sight of the player, the parameter `canSeePlayerWhileWalking` is set to `false`.

For this script to be effective, we will need to create a breadcrumb prefab and assign it to the variable `breadCrumb` within the script. First, let's create a breadCrumb prefab:

1. Create a new empty object by selecting **Game Object | Create Empty**.
2. Rename this object `breadCrumb` and set its position to (x=0, y=0, z=0).
3. Select the folder **Assets | chapter** 6 and click once on this folder.
4. Create a new prefab; from the **Project** window, select: **Create | Prefab**.
5. Rename this prefab `breadCrumb`.
6. Drag-and-drop the object labeled `breadCrumb` from the **Hierarchy** window to this prefab.
7. Select the object `zombie_hires` in **Hierarchy** and drag-and-drop the new prefab `breadCrumb` to the `breadCrumb` variable for the script `controlZombie`, as described on the following screenshot:



8. Now that our breadCrumb prefab has been created, we can delete the `breadCrumb` object from the **Hierarchy** window.
9. Test the game, walk past the idle zombie and check that, after it starts walking, new breadcrumbs are created with a label that starts with `breadcrumb_zombe_hires`.

Next, we need to create a state and the associated transition for the zombie to start following its own breadcrumbs.

1. Open the **Animator** window, create a new state (right-click within the window and select **Create | Empty**), and rename this state `FollowBreadCrumbs`.

2. Drag-and-drop the animation `WalkForward` by selecting **Assets** | **chapter5** | **chapter5_pack** (the animation is within the prefab `Zombie@walkForward` and symbolized by a gray box with a white triangle) to the `Motion` property of the state `FollowBreadCrumbs`.
3. Within this window, we will create two Boolean parameters labeled `canSeePlayerWhileWalking` and `backToSquare1` as illustrated in the following screenshot:

4. Create a transition from the state **WalkForward** to the state **FollowBreadCrumbs**.
5. Select the transition between these two states and set the transition condition to `canSeePlayerWhileWalking = false`. This way, if the zombie is walking toward the player but loses sight of the player, it will start its way back to its initial position.
6. Create a transition from the state **FollowBreadCrumbs** to the state **Idle**.
7. Select the transition between these two states and set the transition condition to `backToSquare1 = true`. This way, when the zombie has reached its first breadcrumb (initial position), it will transition back to the **Idle** state.

Next, we are going to code the breadcrumbing behavior. Add the following lines at the start of the script `controlZombie`:

```
Public var FollowBreadCrumbsState:int =
  Animator.StringToHash("Base Layer.FollowBreadCrumbs");
Public var detectedPlayersClosestCrumb:boolean = false;
```

- In statement 1 of the previous code, the variable `FollowBreadCrumbsState` will be used to monitor the state `FollowBreadCrumbs` that we have created in the previous section
- In statement 2 of the previous code, the variable is used to initialize the first breadcrumb to be followed by the zombie

Add the following code to the function `Update`:

```
case FollowBreadCrumbsState:
  if (breadCrumbIndex >0)
  {
    anim.SetBool("backtoSquare1", false);
```

```
    var breadCrumbToFind: GameObject =      GameObject.
Find("breadcrumb_"+gameObject.name+"_"+breadCrumbIndex)
  ;
    transform.LookAt(breadCrumbToFind.transform);
    var distanceToBreadCrumb:float =
  Vector3.Distance(gameObject.transform.position,
    GameObject.Find("breadcrumb_"+gameObject.name+"_"+breadCrumbIndex)
      .transform.position);
    if ( distanceToBreadCrumb < 1.0f)
    {
       Destroy(GameObject.Find("breadcrumb_"+
  name+"_"+breadCrumbIndex));
       breadCrumbIndex--;
    }
  }
  else {
  anim.SetBool("walking",false);anim.SetBool("backToSquare1",true);
    }
  break;
```

- In statement 2 of the previous code, we check whether any breadcrumbs have been thrown yet.
- In statement 6 of the previous code, we describe how the zombie looks at the last breadcrumb.
- In statement 7 of the previous code, we determine the distance between the zombie and the next breadcrumb.
- In statements 8-11 of the previous code, we determine if the zombie is within one meter from the next breadcrumb; then it is destroyed and the zombie will walk toward the subsequent breadcrumb.
- Statement 10 of the previous code shows how the nearby breadcrumb is destroyed and the variable `breadCrumbIndex` is decreased.
- Statement 14 of the previous code determines if the zombie has reached the first breadcrumb; it has reached its initial position and can transition back to the state `Idle`. We also set the variable `walking` to false so that the zombie stays in the `Idle` state.

Save the script and test it by moving the player around the maze, and either firing at or walking in front of the zombie. Once the zombie starts following the player, move to a location where it cannot see the player, and using the **Scene** view, check that the zombie walks back to its initial position.

Allowing enemies to follow the player's breadcrumbs

It would also be great to improve this behavior by adding the ability for the zombie to follow the player. In the next section, we will create a new state (and associated transitions) called `FollowPlayersBreadCrumbs`. In this state, we need to detect the closest breadcrumb dropped by the player, detect the index of this breadcrumb, and start moving the zombie along the path defined by the breadcrumbs from the closest onward. For example, if the player has dropped 20 breadcrumbs, and the closest breadcrumb from the zombie is the tenth breadcrumb, then the zombie will walk toward the tenth breadcrumb, then the eleventh breadcrumb, and so on, until it reaches the last breadcrumb dropped by the player.

First, let's add the ability for the player to drop breadcrumbs. Create a new folder labeled `Scripts` by selecting **Assets | chapter6** and select it; create a new script and rename it `playerBreadCrumb` (this script should now be in **Assets | chapter6 | Scripts**), add this script to the **First Person Controller**, and add the following lines of code to the script:

```
var breadCrumb:GameObject;
var timeForNextCrumb:float;
public var currentTime:float;
public var breadCrumbs = new Array();
public var index:int;
function Start ()
{
  timeForNextCrumb = Time.time;
  index = 0;
}
function Update ()
{
  if (Time.time > timeForNextCrumb)
  {
    timeForNextCrumb = Time.time+.5;
    breadCrumbs[index] = gameObject.transform.position;
    var b:GameObject  =
  GameObject.Instantiate(breadCrumb,transform.position,
    transform.rotation);
    b.name = "breadcrumb_player_"+index;
    index++;
  }
}
```

The previous code is similar to the one created for the zombie, with the exception that the name of the breadcrumb dropped by the player will be different. Breadcrumbs will be dropped every 500 milliseconds and each of them will have a unique name. Every time a breadcrumb is dropped, the corresponding index is incremented by 1.

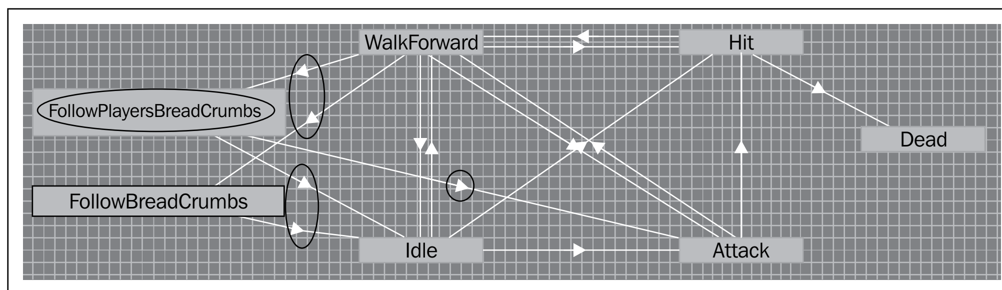
Next, let's create the corresponding new state for the zombie:

1. Select and open (double-click) the animation `zombieController` by selecting the folder **Assets | chapter5**.
2. Open the **Animator** window, create a new state, and rename this state `FollowPlayersBreadCrumbs`.
3. Add the animation `WalkForward` to the `Motion` attribute of this state as we have done for the state `FollowBreadCrumbs`.
4. Add an additional Boolean parameter labeled `canFollowPlayersCrumbs` as illustrated in the following screenshot. This variable will be used to determine the behavior of the zombie that could either follow its own breadcrumbs after losing sight of the player, or follow the player's breadcrumbs. This occurrence would add some challenge and uncertainty to our game as the player would not know what type of zombie he/she will encounter as well as its level of intelligence.

5. Create a transition from the state `WalkForward` to the state `FollowPlayersBreadCrumbs`.
6. Set the transition conditions to **canSeePlayerWhileWalking = false** and **canFollowPlayersCrumbs = true** (you can click on the **+** button to include the second condition).
7. Create a transition from the state `FollowPlayersBreadCrumbs` to the state `Idle`.
8. Set the transition condition to **backToSquare1 = true**.
9. Create a transition from the state `FollowPlayersBreadCrumbs` to the state `Attack` and set the transition condition to **withinReach = true**.

Since we want to differentiate between two types of zombies that can or cannot follow the player's breadcrumbs, we will also need to amend the transitions created previously between the states `WalkForward` and `FollowBreadCrumbs`: click on the transition between the states `WalkForward` and `FollowBreadCrumbs`, and add the condition **canFollowPlayersCrumb = false**, as illustrated in the following screenshot:

After adding these two new states and associated transitions, the animation `zombieController` in the **Animator** window should look like the following screenshot (new states and transitions are highlighted with circles):



Next, let's modify our script `controlZombie` to manage and trigger the state `FollowPlayersBreadCrumbs` by adding the following code at the start of the script:

```
public var FollowPlayersBreadCrumbsState:int =
  Animator.StringToHash("Base Layer.FollowPlayersBreadCrumbs");
public var playerBreadCrumbsIndex:int = 0;
public var canFollowPlayersCrumbs:boolean;
```

In the previous code, we create a variable to monitor the state `FollowPlayersBreadCrumbs`, an index for the breadcrumbs dropped by the player, and a variable that will be used to determine if the zombie to which this script is attached can actually follow the player's breadcrumbs. Add the following function to the script:

```
function setCharacterType(canFollow:boolean)
{
  anim = GetComponent("Animator");
  canFollowPlayersCrumbs = canFollow;
  anim.SetBool("canFollowPlayersCrumbs",canFollowPlayersCrumbs);
}
```

This function will define whether this enemy will be able to follow the player's breadcrumbs.

Add the following line in the `Start` function:

```
setCharacterType (canFollowPlayerCrumbs);
```

Next, we can create the code that handles the state `FollowPlayersbreadCrumbs`.

Add the following code line at the start of the script:

```
private var player:GameObject;
```

Add the following code line in the `Start` function:

```
player = GameObject.FindWithTag("Player");
```

Add the following code to the `Update` function, within the `switch` structure:

```
case FollowPlayersBreadCrumbsState:
anim.SetBool("backtoSquare1", false);
  if (!detectedPlayersClosestCrumb)
  {
    var closest:float = 200.0f;
    var indexOfClosest:int;
    var maxIndex:int =
  player.GetComponent(playerBreadCrumb).index;
    for (var i:int = 0; i < maxIndex; i++)
    {
      var objectToFind:GameObject =
  GameObject.Find("breadcrumb"+"_player_"+i);
      distance = Vector3.Distance(transform.position,
  objectToFind.transform.position);
      if (distance < closest)
  {
         indexOfClosest = i; closest = distance;
  }
    }
    playerBreadCrumbsIndex = indexOfClosest;
    detectedPlayersClosestCrumb = true;
  }
```

- Statement 1 of the previous code shows that the next lines will apply when the zombie is in the state `FollowPlayersBreadCrumbs`.
- Statement 3 shows that if the closest breadcrumb has not been defined or detected yet, then we will do so.

- In statement 4 of the previous code, we create an array that will include all of the breadcrumbs generated by this zombie.
- Statement 5 of the previous code explains that the variable `closest` is used to determine the distance of the closest breadcrumb. It is set to 200 initially, so that any breadcrumb originally included in the array would be the closest.
- In statement 6 of the previous code, the index of the closest breadcrumb is defined.
- In statement 7 of the previous code, we access the number of breadcrumbs dropped by the player.
- In statements 8-15 of the previous code, we loop through all breadcrumbs dropped by the player and identify the closest.
- Statement 17 of the previous code explains that once we have defined the closest breadcrumb, its index is used and saved in the variable `indexOfClosest`.
- Statement 18 of the previous code explains that since we have identified the closest breadcrumb, the corresponding variable, `detectedPlayersClosestBreadCrumb` is set to `true`.

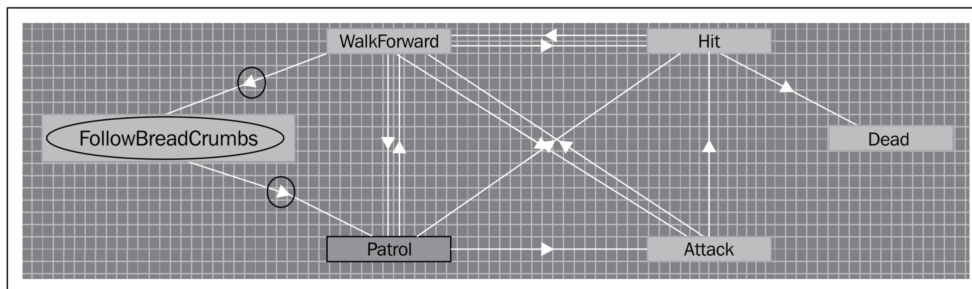
Add the following lines of codes following the last line we have just typed:

```
else
{
transform.LookAt(GameObject.Find("breadcrumb_player_"+playerBreadC
  rumbsIndex).transform);
  distanceToBreadCrumb =
  Vector3.Distance(gameObject.transform.position,
    GameObject.Find("breadcrumb_player_"+playerBreadCrumbsIndex).trans
      form.position);
  if ( distanceToBreadCrumb< 1.5f)
    {
  playerBreadCrumbsIndex++;
    }
  }
break;
```

The previous code is similar to the one created for the zombie in the state `FollowBreadCrumb`. The zombie looks at and walks toward the next breadcrumb. Finally, remember that we also have a different type of enemy, the patrollers with a dedicated animation `controlPatroller`. Because the prefab that we have created based on this animation is also linked to the script `controlZombie`, it will start throwing breadcrumbs as soon as it sees the player and will start walking toward him/her. However, we still haven't implemented a solution that ensures that it goes back to its initial path if the player is not in sight anymore. We could do this by modifying the corresponding animation and adding a few additional lines to the script `controlZombie`:

1. Locate and select the animation `zombiePatrolController` by selecting folder **Assets | chapter5**.
2. Open the **Animator** window (**Window | Animator**); we should see the states created previously for this animation. Create a new state and label it `FollowBreadCrumbs`.
3. Locate the animation `WalkForward` and drop it to the `Motion` attribute of the state `FollowBreadCrumbs`.
4. Create two new Boolean parameters, `canSeePlayerWhileWalking` and `backToSquare1`.
5. Create a transition from the state **WalkForward** to the state **FollowBreadCrumbs** and set the transition condition to **canSeePlayerWhileWalking = false**.
6. Create a transition from the state **FollowBreadCrumb** to the state **Patrol** and set the transition condition to **backToSquare1 = true**.

After these modifications, the animator should look like the following screenshot (the new state and corresponding transitions have been highlighted by circles):



Next, we will need to modify the script `controlZombie` to manage and trigger these states.

Open the script `controlZombie`, and add the following line at the start of the code that handles the state `Patrol` as highlighted in the following code example:

```
case PatrolState:
anim.SetBool("backToSquare1",false);
```

Finally, we need to add a breadCrumb prefab to the object `patroller` so that it can instantiate breadcrumbs overtime. Select the object `patroller` and drag-and-drop the prefab breadCrumb from **Assets** | **chapter6** to the variable (**placeholder**) breadCrumb located in the **Inspector** for the function `controlZombie`, within the object `patroller`. Play the scene, and check that after firing at the patroller, it follows you; move to a location where it can't see you, and check in the **Scene** view that it follows its own breadcrumbs and resumes the patrol where it had been stopped.

Creating and updating prefabs

Let's add a red dot to the prefab `zombie_idle`, so that any zombie can be detected on the mini-map. Perform the following steps:

1. Create a new sphere, rename it `dot_enemy`, set its size or scale to (x=2, y=2, z=2), remove its collider component, apply the `Red` texture to this object (this texture is located in **chapter4** | **chapter4_pack**), and set its `Layer` property to `topView`.
2. Set the `shader` of this object to `Self-Illumin/Diffuse` and duplicate this object.
3. Drop the first duplicate, `dot_enemy`, on the object labeled `zombie_hires` in the **Hierarchy** view and change the position of the object `dot_enemy` to (x=0, y=0, z=0).
4. Drop the second duplicate, `dot_enemy`, on the object labeled `patroller` in the **Hierarchy** view and change the position of the object `dot_enemy` to (x=0, y=0, z=0).
5. Drag-and-drop the object `zombie_hires` on the prefab `static_Enemy` located in **Assets** | **chapter5**; this will update the prefab with our latest changes.
6. Repeat the previous step with the object `patroller` and the prefab `zombie_patroller`.
7. Now that we have updated/created prefabs, we can deactivate the objects `patroller` and `zombie_hires` in the **Hierarchy** view (using the Inspector and unchecking the box to the right of the name of this object).

We will need to create ammunitions that the player will be able to collect:

1. Create a new cube, rename it ammunitions, change its `scale` properties to (x=0.2, y=0.5, z=0.5), its position to (x=0, y=1, z=1), and apply the texture `texture_ammo` located in **Assets | chapter6 | chapter6_pack** to this object.
2. Create a new tag, `ammunitions`, and apply it to this object.
3. Duplicate the object `dot_gun` from the `gun` object, rename it `dot_ammo`, drag-and-drop it to the object `ammunitions`, and set its position to (x=0, y=0, z=0).
4. Locate the script `rotate.js` (**Assets | chapter3**) and attach it to the object labeled `ammunitions`.

Now that the ammunitions have been created, we will modify our scripts so that the variable used to track ammunitions is increased when ammunitions have been collected. Open the script `collisionDetection` and modify the conditional statement at the start of the function `OnControllerColliderHit`, as highlighted in the following code:

```
if (c.gameObject.tag == "medpack" || c.gameObject.tag == "key" ||
  c.gameObject.tag == "gun" || c.gameObject.tag == "ammunitions")
  {
```

Add the code highlighted in the following code snippet:

```
if (c.gameObject.tag == "gun")
  {
    hasGun = true; displayGUITexture(true, "gun");
    changeGUITexture("true",gun);
    GameObject.Find("GUITexture_crosshair").guiTexture.enabled
  =true;
    GetComponent(shootBullet).nbBullets = 40;
  }
  if (c.gameObject.tag == "ammunitions")
  GameObject.FindWithTag("Player").GetComponent(shootBullet).nbBulle
    ts += 30;
}
```

Based on the previous code (statement 8), if we collide with ammunitions, the number of bullets is increased to 40. Note that the ammunitions will be destroyed as for the objects `medpacks`, `keys`, and `gun` based on the code we have already included in this script. The player can now collect a wide range of objects, including a key, a med pack, a gun, and ammunitions. While these objects have been created manually, it would be great to be able to create them at run-time from a script, when the game starts, or even during the game. To do so, we need to create the corresponding prefabs, and instantiate them at run-time. Using scripting, we can then modify the properties of the new instances (that is, copies of the prefabs), including their position or rotation. Interestingly, we could also, based on the health of the user, create a script that instantiates med packs relatively close to him/her. Let's create prefabs for these objects.

Create a prefab for the med pack:

1. Select the object labeled `medpack` from the **Hierarchy** view.
2. In the **Project** window, locate the folder, `chapter6`, by navigating to **Assets | chapter6** and select it.
3. Create a new prefab in this folder: select **Create | Prefab** from the **Project** window.
4. Rename this prefab `medpack`.
5. Drag-and-drop the object called `medpack` from the **Hierarchy** window to this new prefab.
6. Deactivate the object `medpack` in the **Hierarchy** view (that is, uncheck the box to the left of the label `medpack` in the **Inspector** window).
7. Repeat steps 1-5 to create a prefab labeled `ammunitions` from the object labeled `ammunitions`.

We can then use these prefabs and instantiate instances at different locations in the maze: check that the objects `medpack` and `ammunitions` are not active. Locate and open the script `initGame` that is currently attached to the **First Person Controller**, and add the following lines at the start of the script:

```
public var ammunitions:GameObject;
private var objectsInstantiated = new Array();
private var indexOfObjectInstantiated:int;
```

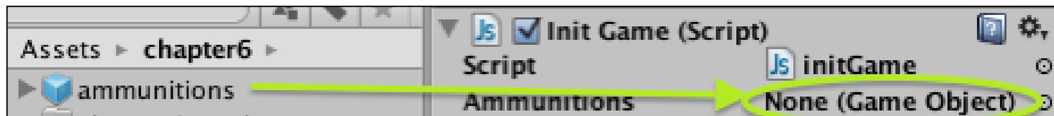
In the previous script, we create a placeholder for our ammunitions. Because this variable is public, we will be able to drag-and-drop an object (for example, the `ammunitions` prefab) to this variable from the **Inspector** window. We create an array for the new objects that will be created based on the prefabs (that is, instances) as well as an index for the previous array to be able to refer to all the new objects created.

Add the following code inside the function `Start`:

```
objectsInstantiated [indexOfObjectInstantiated++] =
  Instantiate(ammunitions, Vector3(0,1,1),
    Quaternion.Euler(0,0,0));
objectsInstantiated [indexOfObjectInstantiated++] =
  Instantiate(ammunitions, Vector3(3,1,-7),
    Quaternion.Euler(0,0,0));
```

In the previous code, we create two new objects as part of the array defined earlier. Once the objects have been created, the index of the array is incremented by 1. The new objects are instances of the object `ammunitions` defined earlier. They will be located at the position (x=0, y=0, z=1) and (x=3, y=0, z=-7), with no rotation.

Finally, select the object **First Person Controller** in the **Hierarchy** window and locate the script component `initGame` for this object in the **Inspector** window. We should see that it includes a placeholder (variable) called **Ammunitions**. Drag-and-drop the prefab **ammunitions** to this placeholder as illustrated in the following screenshot:



Play the scene and check that the two ammunition packs are included in the scene.

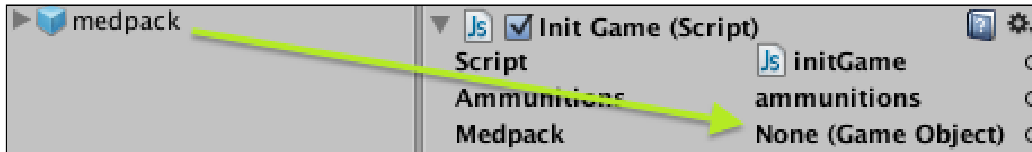
Let's do the same for the med packs. Add the following line to the start of the script `initGame`:

```
public var medpack:GameObject;
```

Add the following line within the `Start` function:

```
objectsInstantiated [indexOfObjectInstantiated++] =
  Instantiate(medpack, Vector3(3,1,6), Quaternion.identity);
```

Drag-and-drop the prefab `medpack` to the placeholder **Medpack** as described in the following screenshot:



Finally, we need to generate several enemies at runtime. Add the following lines at the start of the script `initGame`:

```
private var i:int;
private var j:int;
public var zombie_idle:GameObject;
public var breadCrumb:GameObject;
var newObject : GameObject;
```

Add the following lines within the `Start` function:

```
GameObject.Find("GUIText_ammo").guiText.text="";
GameObject.Find("GUITexture_crossHair").guiTexture.enabled =
  false;
GameObject.Find("GUIText_displayMessageToUser").guiText.text="";
newObject = Instantiate(ammunitions, Vector3(0,1,1),
  Quaternion.identity);
newObject = Instantiate(ammunitions, Vector3(3,1,-7),
  Quaternion.identity);
newObject = Instantiate(medpack, Vector3(3,1,6),
  Quaternion.identity);
for (i = 0; i<1; i++)
{
     for (j = 0; j<2; j++)
     {
         print(i+":"+j);
          var orientation:float = Random.Range(0,180);
          newObject = Instantiate(zombie_idle, Vector3(-4+i,0.5,-
            4+j), Quaternion.Euler(0,orientation,0)) as GameObject;
          newObject.GetComponent(controlZombie).breadCrumb =
            breadCrumb;
          newObject.GetComponent(controlZombie).
            setCharacterType(false);
          newObject.name = "zombie_a"+i+j;
          newObject = Instantiate(zombie_idle, Vector3(-
            4+i+7,0.5,-4+j), Quaternion.Euler(0,orientation,0)) as
            GameObject;
```

```
            newObject.GetComponent(controlZombie).breadCrumb =
             breadCrumb;
            newObject.GetComponent(controlZombie).
              setCharacterType(Random.Rang e(0,10) >5);
            newObject.name = "zombie_b"+i+j;
        }
    }
```

In the previous code, we create four zombie objects with a position that is partly defined by the two looping variables `i` and `j`. Each new object has a random orientation, and a breadcrumb prefab is attached so that the zombie can drop breadcrumbs. Move the **First Person Controller** to the position (x=21, y=0.6, z=8), move the key to the position (x=-20, y=1, z=0) and test the scene. Next, select the object **First Person Controller**, and drag-and-drop the prefabs `breadcrumb` (**Assets | chapter6**) and `staticEnemy` (**Assets | chapter5**) to their respective variables/placeholders `breadCrumb` and `zombie_idle` in the **Inspector** window, located in the component `initGame`.

Keeping track of the number of lives

So far, we have tracked the user's health levels and decreased these accordingly, whenever it had been hit by enemies or when it has collected a health pack. However, in addition, we could give several lives to players. Whenever they lose a life, they can restart the current level. To do so, we will need to keep track of the number of lives across levels, and also reload the current level when a life has been lost. In our game, the game will end whenever the player has sustained too many injuries (that is, `health = 0`), or the player has run out of time. In this case, we would like the player to restart the game. To do so, we will need to monitor the player's health or the time, and reload the level accordingly. Open the script `healthBar` and add the following code in the function `Update`:

```
if (currHealth <=0)
  Application.loadLevel(Application.loadedlevel);
```

This code checks whether the health levels are equal to or below `0`, and if so, the current level is reloaded.

Open the script `timer` and add the following code in the function `Update`:

```
if (minutes>=10)
  GameObject.Find("healthBar").GetComponent(healthBar).setHealth(0);
```

In the previous code, we check whether the timer has reached `10` minutes, if this is the case, the health of the player is set to `0`, which means that the level will be reloaded accordingly.

Lastly, we need to check whether the player has fallen into the water:

1. Select the object labeled `water` in the **Hierarchy** window (within the folder `maze`).
2. Add a collider to this object by selecting: **Components** | **Physics** | **Box Collider**.
3. In the **Inspector** window, change the center properties of this collider to (x=0, y=-4, z=0) and its size properties to (x=2, y=0, z=2).
4. Create a new tag called `water`, and apply it to the object labeled `water`.

Next, add the following code to the script, `collisionDetection`, at the end of the function `OnControllerColliderHit`:

```
if (c.gameObject.tag=="water")
{
  GameObject.Find("healthBar").GetComponent(healthBar).setHealth(0);
}
```

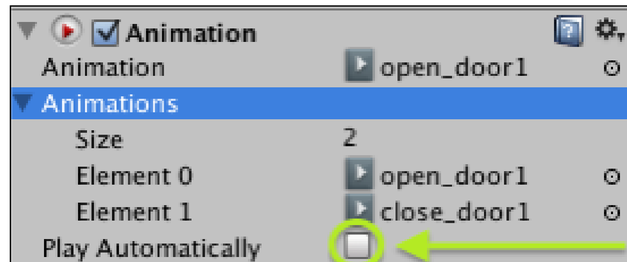
In the previous code, the health of the player is set to `0` when he/she collides with the water.

Animating the door that leads to the water section

Finally, we need to ensure that the door that leads to the section with the water is closed, and opens when the player collides with it. The door will close after 3 seconds. Perform the following steps:

1. Locate the object `door1` in the **Hierarchy** window; select it and change its position to (x=0, y = 1, z= 24).
2. Create a tag, `room2_door`, and apply it to this door.
3. Locate the animations `open_door1` and `close_door1` in the folder **Assets** | **chapter6** | **chapter6_pack** and drag-and-drop these animations on the object labeled `door1`.
4. Select the object `door1` and check that the two animations appear as components in the **Inspector** window.

5. In the component called **Animation**, uncheck the box to the right of the parameter **Play Automatically**, so that the animation does not play at startup, as illustrated in the following screenshot:



Open the script `collisionDetection` and add the following lines at the start of the script:

```
private var startDoor1Timer:boolean = false;
private var timer:float;
```

Add the following code at the end of the function `OnControllerColliderHit`:

```
if (c.gameObject.tag == "room2_door")
{
  c.gameObject.animation.Play("open_door1");
  startDoor1Timer = true;
}
```

In the previous code, the door is open when the player collides with it. The variable `startDoor2Timer` is set to `true`; this will, as we will see in the following code, start the timer, triggering the closure of the door after `3` seconds.

Add the following code to the function `Update`:

```
if (startDoor1Timer)
{
  timer+=Time.deltaTime;
  if (timer>3)
  {
    startDoor1Timer = false;
    GameObject.Find("door1").animation.Play("close_door1");
  }
}
```

In the previous code, we check whether the timer is active; the timer is updated every second and when it reaches 3 seconds, it is deactivated and the door is closed. Play the scene and check that the door opens and closes automatically after 3 seconds.

Creating a menu system for your game

We have almost finished our game and level. For our game to be complete, we will need to create a splash screen and instructions, as well as screens to be displayed when the player has lost or succeeded. These screens will be created using scenes, and navigation between them will be implemented using buttons. To open a particular menu (for example, splash screen, instructions, or game over), we will load the corresponding scene. The background of the splash screen will include the maze created earlier and some animated zombies (`idle`). First let's create a prefab for the maze so that it can be reused.

Open the last scene, select the folder **Assets | chapter6**, create a new prefab (that is, select: **Create | Prefab** from the **Project** window), and rename it `maze`. Locate the object labeled `maze` in the **Hierarchy** window and drag-and-drop it on the prefab we have just created. This prefab will be accessible from any scene within our project, including the splash screen scene.

Next, let's create the splash screen scene:

1. Create a new scene (**File | New Scene**) and save it as `splashScreen` (**File | Save Scene As**).
2. Rename the default camera present in the scene (**Main Camera**) `camera1` and change its position to (x=0, y=1, z=0).
3. Drag-and-drop the `maze` prefab we created earlier on the scene and change its position to (x=0, y=0, z=0).
4. Within the `maze` object in the **Hierarchy** window, deactivate the objects **First Person Controller** and **Main Camera** located in the folder `maze`.
5. Drag-and-drop the prefab labeled `staticEnemy` (**Assets | chapter5**) on the scene; change its position to (x=0, y=-0.5, z=3) and its rotational component to (x=0, y=180, z=0), and deactivate the script `controlZombie` that is attached to it (we will not need to manage different states for this character in the splash screen); we will also deactivate the object labeled `dot_enemy` that is within the zombie (`staticEnemy`) object, as the top view will not be used in the splash screen.
6. Duplicate this object (that is, `staticEnemy`) 14 times, and change the properties of the duplicates as follows (x=0, y=-0.5, z=3), (x=0.5, y=-0.5, z=4), (x=-0.5, y=-0.5, z=4), (x=-2, y=-0.5, z=5), (x=-1, y=-0.5, z=5), (x=0, y=-0.5, z=5), (x=1, y=-0.5, z=5), (x=2, y=-0.5, z=5), (x=-3, y=-0.5, z=6), (x=-1.5, y=-0.5, z=6), (x=0, y=-0.4, z=6), (x=1.5, y=-0.5, z=6), (x=3, y=-0.5, z=6), (x=-7, y=-0.4, z=10), and (x=7, y=-0.4, z=10).

7. Create three `GUItext` objects (**GameObject | Create Other | GUI Text**), rename them `GUIText_outbreak`, `GUIText_subtitle`, and `GUIText_clickToContinue`, and change their position to (x=0.5, y=0.97, z=0), (x=0.5, y=0.8, z=0), and (x=0.5, y=0.3, z=0), respectively.
8. For the object `GUIText_outbreak`: set the `text` attribute to `Zombie Outbreak!`, `Text Anchor` to `middle-center`, `Font` to `OhTheHorror`, and the `Font Size` to `60`.
9. For the object `GUIText_subtitle`: use the same values, and set the `Text` attribute to `They are ready for you but are you...?`, and `Font Size` to `20`.
10. For the object `GUIText_clickToContinue`: use the same values, and set the `Text` attribute to `Click to Continue` and the `Font Size` to `20`.
11. Create a new script (JavaScript) within the folder **Assets | chapter6 | Scripts**, rename it `continueButton`, attach it to the object labeled `GUIText_clickToContinue`, and open it so that we can edit it. Once open, add the following script within the function `Update`:

```
if (Input.GetMouseButtonDown(0))
{
  if (Application.loadedLevelName == "splashScreen")
  Application.LoadLevel("instructions");
  else if (Application.loadedLevelName == "instructions")
  Application.LoadLevel("chapter6");
  else Application.LoadLevel("splashScreen");
}
```

In the previous code, if the player clicks on the `GUIText` attached to the script, we check the name of the current scene. If we are in the `splashScreen` scene, then the `instruction` scene will be loaded; if the current scene is the `instruction` scene, then the `game` scene will be loaded. Test the `splashScreen` scene.

To add some atmosphere to our game, we will include a sound track in all scenes. The background sounds will be issued from the site `incompetech`, which features sounds created by Kevin MacLeod. Create a new folder labeled `Sounds` within the folder **Assets | chapter6**. This folder will be used to store the sounds we have imported. Open the URL `http://incompetech.com/music/royalty-free/`, read the terms and conditions that apply to the sounds provided by this site, select the option **Action**, and click on the button labeled **Search by Feel**. This should return a list of short loops that we can use outside the game for the different screens.

Download four sounds of your choice, rename them `sound_splashScreen`, `sound_gameOver`, `sound_success`, and `sound_inGame`, and import them inside your project in the folder **Assets | chapter6 | Sounds**. While you may use the sound of our choice, I have chosen `GustavSing` (for the `splashScreen` scene), `Take a Chance` (for the `gameSuccess` scene), `Feral Chase` (during the game), and `Exciting Trailer` (for the `gameOver` scene). Create an empty object, rename it `bgSound`, and drag-and-drop the sound `sound_splashScreen` on this object. This will be used as a new background sound for our scene. Select the object `bgSound`, and in the **Inspector** window, look at the **Audio Source** component, and check that the options **Play on Awake** and **Loop** are selected for this sound, so that it loops indefinitely.

Let's create a script that will make it possible to mute the background sound. Select the folder **Assets | chapter6**, create a new script, rename it `muteAudio`, and attach it to the object labeled `bgSound`. Once this is done, open the script `muteAudio` and add the following code within the function `Update`:

```
if (Input.GetKeyUp(KeyCode.M))
{
  audio.mute = !audio.mute;
}
```

In the previous code, we toggle the mute attribute of the background sound when the player presses the key *M*. Finally, we will drag-and-drop the object `bgSound` on the object `camera1`, and change the position of the object labeled `bgSound` to (x=0, y=0, z=0). This will make the object a child of `camera1` so that the sound is played exactly where the camera is. Now that you have created the `splashscreen` scene, we will create the screen for the game instructions as shown in the following steps:

1. Save the current scene (**File | Save Scene**). The scene should be present in the folder `Assets` and we will duplicate it to create the `instructions` scene. For this, we can select this scene (`splashScreen`), and press the keys *Ctrl* + *D*. This will create a new scene that we can rename `instructions`.
2. Double-click on the scene labeled `instructions` from the `Assets` folder to edit it. In the new scene (`instructions`), change the position of the object `camera1` to (x=0, y=1.24, z=9), add a new `GUIText` object to the scene, and rename it `GUIText_instructions`. Change the position of this object to (x=0.5, y=0.6, z=0) and locate its `GUIText` component in the **Inspector** window. Within this component (`GUIText`), change the `text` property to `"Collect the gun and the ammunitions, find the key to the exit door, and escape in one piece. Use the arrow keys and the mouse to navigate through the maze. Click on the mouse left button to shoot`." So that this text is displayed over several lines, place your cursor just before the word **Use** and simultaneously press the keys *Alt* + Return. Repeat this step to add a line break just before the sentence that starts with the text **Click on the mouse**.

3. For this `GUItext` object, change the `Anchor` property to `middle center`, the `Alignment` property to `center`, the `Font` property to `Arial`, and the `Font size` to `25`.
4. Save the scene (**File | Save Scene as**) and duplicate this scene twice. Rename the duplicates `gameOver` and `gameSuccess`.
5. Open the `gameOver` scene, and change the text for the `GUIText` object `GUIText_instructions` to "`Well, looks like you were not ready yet...try again`." Also, change the text of the object labeled `GUIText_clickToContinue` to `"Click for a New Game`."
6. Save this scene (**Files | Save Scene**).
7. Open the `gameSuccess` scene, and change the text for the object labeled `GUIText_instructions` to "`Well done, you've made it.`" Also, change the text of the object labeled `GUIText_clickToContinue` to `"Click for a New Game`."
8. Open the project settings (**File | Build Settings**) and drag-and-drop all the scenes we have created so far (for example, `chapter6`, `gameOver`, `instructions`, `splashScreen`, and `gameSuccess`) from the `Assets` folder to the window labeled **Build Settings**. This will ensure that we can load scenes after clicking on the corresponding buttons. We can now close the window **Build Settings**.

Open the script `collisionDetection` and modify it as highlighted in the following code:

```
if (hasKey)
{
  c.gameObject.animation.Play ("open_door");
  yield WaitForSeconds (1);
  Application.LoadLevel("gameSuccess");
}
```

In the previous code, we wait for one second, so that the animation for the door is completed. We then open the scene `gameSuccess`.

We can add a background sound to the scenes `gameOver`, `gameSuccess`, and `instructions`: open each scene, select the object `bgSound` from the **Hierarchy** window, and drag-and-drop the sound of your choice from the folder **Assets | chapter6 | Sounds** to the `AudioSource` attribute of the component `Audio Source` for the object `bgSound`. Note that the sound used for the splash screen can also be used for the instruction screen. We can also add a background sound for the scene `chapter6` by adding both the sound `sound_inGame` and the script `muteAudio` to the camera within the **First Person Controller** (**camera1**) and ensuring that the options **Play On Awake** and **Loop** are selected for this sound (that is, `Audio Component` for the object **First Person Shooter**).

Keeping track of the number of lives

To track the number of lives, we will need to create an object that includes information on the game (for example, number of lives) and that is kept across scenes (by default, objects are not kept across scenes).

Within the scene `splashScreen`, create a new empty object, and label it `playerData`. This object will be used to keep information on the player, including the number of lives. Create a new script in the folder **Assets | chapter6 | Scripts**, rename it `playerData`, and attach it to the object `playerData`. Open the script `playerData` and add the following code:

```
public var nbLives:int = 3;
function Start () {}
function getNbLives()
{
  return nbLives;
}
function Update () {}
function Awake ()
{
    DontDestroyOnLoad (transform.gameObject);
   if (Application.loadedLevelName == "splashScreen") nbLives = 3;
}
```

In the previous code, we create a new variable, `nbLives`, that will be used to store the number of lives for the player. We also create a function, `getNbLives`, that will return the number of lives. This will be useful in deciding whether the player can restart the current level after losing a life or if the game is over. Finally, we create a function `Awake`, and specify, within this function, that this object will not be destroyed when loading another scene. In other words, the information contained in this script will be kept across scenes or when the scene is reloaded. We also set the number of lives to `3` when the player starts a new game.

Open the script `healthBar` and add the following script to the function `Update`:

```
if (currHealth <=0)
{
  if
  (GameObject.Find("playerData").GetComponent(playerData).getNbLives
    ()>0)
  {
  GameObject.Find("playerData").GetComponent(playerData).nbLives -
    = 1;
    Application.LoadLevel(Application.loadedLevelName);
  }
  else  Application.LoadLevel("gameOver");
}
```

In the previous code, the game is over if the player's health is `0` or less and he/she has no more lives. To be able to display the number of lives, we can create a new `GUIText` object, rename it `GUIText_nbLives`, set its position to (x=0.16, y=0.98, z=0), and add the following code inside the script `healthBar`, within the function `Start`:

```
GameObject.Find("GUIText_nbLives").guiText.text="NB Lives: "+
  GameObject.Find("playerData").GetComponent(playerData).getNbLives(
    );
```

Play the scene `splashScreen` and check that the game behaves as planned.

Optimizing the game

At present, we will notice that the breadcrumbs are generated all the time for the player, although he/she is not moving. This has the disadvantage of flooding the scene with a significant number of breadcrumbs from the player. We can solve this in at least two ways: we could check that the player has walked at least 1 meter before the next breadcrumb is generated, or each breadcrumb could expire after a given number of seconds. The second option usually is the most believable. To implement the distance-based solution, modify the function `Update` in the script `playerBreadCrumb`, as highlighted in the following code:

```
function Update ()
{
  if (Time.time > timeForNextCrumb)
  {
    timeForNextCrumb = Time.time+.5;
    var lastBreadCrumb: Vector3;
    var distanceToLastBreadCrumb : float;
    if (index > 0) lastBreadCrumb = breadCrumbs[index-1];
    breadCrumbs[index] = gameObject.transform.position;
    distanceToLastBreadCrumb = Vector3.Distance(transform.position,
      lastBreadCrumb);
    if (index == 0 || (distanceToLastBreadCrumb > 1.0f && index >0))
    {
      var b:GameObject =
        GameObject.Instantiate(breadCrumb,transform.position,
    transform.rotation);
      b.name = "breadcrumb_player_"+index;
      index++;
    }
  }
}
```

In the previous code, we simply calculate the distance between the current position and the position of the previous breadcrumb. A new breadcrumb is created only if the player has moved 1 meter further.

Test this game and check that new breadcrumbs are created only if the player moves forward.

To implement the second solution, use the following code:

```
function Update ()
{
  if (Time.time > timeForNextCrumb)
  {
    timeForNextCrumb = Time.time+.5;
    breadCrumbs[index] = gameObject.transform.position;
    var b:GameObject  =
      GameObject.Instantiate(breadCrumb,transform.position,
    transform.rotation);
    b.name = "breadcrumb_player_"+index;
    Destroy(b, 10);
    index++;
  }
}
```

In the previous code, we added an instruction to destroy the breadcrumb after 10 seconds using the built-in function `Destroy`. Test the game with this solution.

Exporting your game to the web

While we have created our game in Unity, it would be great to make it available online, so that it is accessible to friends and other players. Thankfully, Unity3D includes an export feature to export our game to a web format. Let's use this feature to create a web version of your game as shown in the following steps:

1. Open the build settings (**File | Build Settings**), and reorder the scene by dragging-and-dropping them, so that `splashScreen` is first, `instructions` is second, `chapter6` is third, `gameSuccess` if fourth, and `gameOver` is fifth.
2. In the section **Platform**, select the option **Web Player**.
3. Click on the button labeled **Build**. This will open a window where we can specify the location of the exported files. Provide a name for the exported file and select a folder where the game should be created.
4. Click on **Save**. The conversion should last for just under one minute. Once the process is complete, open the folder that we have specified previously, it should include two files: an HTML file and a file with the extension `.unity3d`. Open the HTML file in your browser.

Where to go from here

Throughout this book and chapter, we have discovered several aspects of game development; we have learned how to use Unity3D and make the most of its numerous features. While the project presented in this book provides you with a good basis for your game, it can, of course be improved and extended in many ways, in terms of AI, user interface, or 3D design. To improve your skills and make better games, you may find the following references particularly helpful:

Game design

Game design is a wide area, but understanding its core principles will help you to create games that are fun to play, and that will keep the players immersed and engaged for long periods of time. The following references should definitely help:

- Rules of Play by Katie Salen and Eric Zimmerman (`http://www.amazon.com/Rules-Play-Game-Design-Fundamentals/dp/0262240459`)
- A Theory of Fun for Game Design by Ralph Koster (`http://www.amazon.com/A-Theory-Fun-Game-Design/dp/1932111972/ref=pd_sim_b_3`)
- Challenges for Game Designers by Brenda Braithwaite and Ian Schreiber (`http://www.amazon.com/Challenges-Game-Designers-Brenda-Brathwaite/dp/158450580X/ref=pd_sim_b_2`)
- Game Mechanics: Advanced Game Design (Voices That Matter) by Ernest Adams (`http://www.amazon.com/Game-Mechanics-Advanced-Design-Voices/dp/0321820274`)
- The Art of Computer Game Design by Chris Crawford (`http://www.amazon.com/The-Computer-Game-Design-ebook/dp/B0052QA5WU`)
- The 400 project: a project to capture essential rules for game design (`http://www.finitearts.com/Pages/400page.html`)

Artificial intelligence

Smart and believable AI always makes for better and more entertaining games. The following links and books will provide you with additional information on up-to-date AI plugins, resources, and techniques:

- AI for Game Developers by David Bourg and Glenn Seemann (`http://www.amazon.com/AI-Game-Developers-David-Bourg/dp/0596005555/`)
- Programming Game AI by Example by Mat Buckland (`http://www.amazon.com/Programming-Game-Example-Mat-Buckland/dp/1556220782/`)

- The Path project by AngryAnt (`http://angryant.com/path/`), is a solution that includes a free library, an editor and associated documentation for path finding
- AiGameDev (`http://aigamedev.com/`), is a comprehensive website on AI for video games with many articles and tutorials

3D characters

To create your characters, you can use a wide range of software, including:

- Maya (`http://www.autodesk.com/products/autodesk-maya/overview`)
- 3D Studio Max (`http://www.autodesk.com/products/autodesk-3ds-max/overview`)
- Blender (`http://www.blender.org/`)
- Sketchup (`http://www.sketchup.com/`)

Note that, while the first three applications are usually perceived as having a steep learning curve, Sketchup makes it possible to create 3D models relatively easily.

Creating your audio files

There are many softwares available to create your own sound effects and audio tracks; the following links may provide you with useful resources and tools for your games:

- Audacity (`http://audacity.sourceforge.net/`)
- BFXR (`http://www.bfxr.net/`)
- GarageBand (`http://www.apple.com/ilife/garageband/`)

Learning more about Unity3D

To learn more about Unity3D, there are many resources available both in text or video formats:

- Unity3D Learn (`http://unity3d.com/learn`) is a website dedicated to learning Unity3D with many tutorials and documentation on Unity3D.
- Mecanim (`http://video.unity3d.com/video/7362044/unity-40-mecanim-animation-tutorial`) is a video tutorial on how to use Mecanim.
- Design3 (`http://www.design3.com/`) is a great portal to learn about game designing and development. The site includes a wide range of resources on all necessary tools for game creation and development.

- Digital Tutors (`http://www.digitaltutors.com/11/index.php`) is a great portal to learn how to create games and digital art.
- Mixamo (`http://www.mixamo.com/`) is a site dedicated to character rigging and animation.

Summary

I hope that this book has inspired you to delve into the world of game design and development with Unity3D. Becoming a game developer will require some more work and perseverance, but by reading this book, you have taken the first steps that will, hopefully, lead you to the career path of your dreams.

Index

Symbols

3D character
about 144
animating, for game 91-94
configuring 90, 91
importing 90, 91
3D object imports
URL 23
3D Studio Max
URL 144
+ button 95, 104

A

additional states
creating 103-110
Add Layer option 73
Add Tag option 53, 58, 101
AI
about 143
adding, to enemies 97-100
game developers, URL 143
improving, breadcrumbing used 116
AiGameDev
URL 144
Alignment property 138
ammunitions
about 130
creating 128
displaying 81
updating 81
Anchor attribute 61
AngryAnt
path project, URL 144
AngryBot
URL 15
AngryBots scene
navigating through 15, 16
Animation attribute 64
Animations tab 93, 103
Animation Type attribute 103
Animator 92
Animator component 98, 112
Animator Controller 92
Animator object 97
AnimatorStateInfo variable 97
Animator.StringToHash method 97
Animator window 92-97, 103, 104, 118, 122, 126
anim.SetBool function 101
anim variable 98
Apply button 91
Artificial intelligence. *See* **AI**
assets
importing 44, 45
Attack state 103, 109, 122
Audacity
URL 144
audio
adding 55, 56
audio files
creating 144
Audio Listener component 73
AudioSource attribute 139
Awake function 140

B

backToSquare1 parameter 119, 126
BFXR
URL 144
Blender
URL 144
blocks
creating 34
Bool option 95
breadCrumbIndex variable 117, 120
breadcrumbing
about 115
enemies, allowing to follow player's breadcrumbs 121-127
enemies, allowing to throw and follow their own breadcrumbs 116-120
used, for improving AI 116
breadCrumb object 118
breadCrumb variable 118
built-in objects
maze, creating on 31-37

C

camera
creating 72
Camera component 78
canSeePlayerWhileWalking parameter 118, 119, 126
Challenges for Game Designers
URL 143
changeGUITexture function 63
closest variable 125
collection_beep file 55
collection_beep variable 55
colliders
about 39-41
effect, testing 40
removing 41
collisionDetection script 55, 57, 61-64, 69, 71, 81, 82, 128, 134, 138
Computer Game Design
URL 143
Configure button 91
console 15
Console window 54
controlZombie script 99, 104-111, 116, 123, 127, 135
cube
creating 18-21
Culling Mask attribute 77

D

Dead state 107, 110
decreaseHealth function 109
Design3
URL 144
Destroy function 142
detectedPlayersClosestBreadCrumb variable 125
die parameter 107, 108
Digital Tutors
URL 145
displayMessageTouser script 60
displayMessageToUser script 58, 61
displayText function 61
displayTime variable 59
distance variable 104
door
animating 133, 134
creating 36
dot_fpc object 75
dots
creating, for other objects 76
Download button 50, 84

E

enemies
AI, adding to 97-100
allowing, to follow player's breadcrumbs 121-127
allowing, to throw and follow their own breadcrumbs 116-120
exit_door tag 64
ExitTime parameter 96

F

FindGameOBjectsWithTag method 102
Fire1 button 80, 87
FireSound variable 84

first-person controller
  adding 25-27
First Person Controller object 55
first-person view
  implementing 24, 25
Flare Layer component 73
floor
  creating, for maze 34
floor object 40
Fly mode 12
FollowBreadCrumbs state 119, 126
FollowBreadCrumbsState variable 119
FollowPlayersBreadCrumbs state 121, 122
Font property 50
Font-size attribute 49
free assets and textures
  URL 22

G

game
  3D character, animating for 91-94
  designing 143
  exporting, to web 142
  finishing 64, 65
  menu system, creating for 135-139
  optimizing 141, 142
game engines 8
Game Mechanics
  URL 143
GameObject 118
GameObject.Find() command 50
GameObject.Find function 51, 62
gameObject.GetComponent 50
GarageBand
  URL 144
GetComponent 50
getNbLives function 140
GUILayer component 73
GUIText component 49, 50, 61
GUIText object 49, 50, 61, 81, 138
GUIText_outbreak object 136
GUIText_subtitle object 136
GUIText_timer object 50
GUITextTimer object 50
GUITexture component 79
GuiTexture object 61
guiText variable 50
gun
  creating 79-86
  URL 83
gun object 128

H

hasGun variable 57, 82
hasKey variable 57
health bar
  displaying 69-71
healthBar script 70, 71, 109, 132, 140
health variable 57
height property 79
Hierarchy view 11, 13, 41, 49, 70, 72, 93, 127
Hierarchy window 44, 55, 58, 73, 91, 92, 97, 101
hit parameter 105
Hit state 105
hit variable 100, 106
horizontal walls
  creating 35

I

Idle state 94-103, 110
imported objects
  inserting 23
Importing package 90
incompetech
  URL 136
indexOfClosest variable 125
initGame script 82, 129-131
Inspector window 11, 14, 15, 40, 46-55, 58, 72-77, 90, 96-103, 130
Integrated Development Environment (IDE) 8
inventory system
  creating 57-63
  displaying 57-63

L

label tag 58
label variable 62

layer
allocating, to objects 77
creating 77
Layer label 75
level
mini-map, displaying 72-78
tuning 38, 39
lights
adding 37
Loop Pose attribute 103

M

Maya
URL 144
maze
creating, on built-in objects 31-37
floor, creating 34
Mecanim
URL 144
medpack object 129
medpack tag 53
menu system
creating, for game 135-139
messages
sending, to alert other close enemies 101, 102
message variable 59
mini-map
displaying, of level 72-78
minutes variable 49
Mixamo
about 89
URL 89, 145
MonoDevelop 46
Motion attribute 126
Motion property 119
Motion variable 110
mouselook option 12
Muscles tab 91

N

navigation
in scene view 12, 13
in scene view, URL 13
through AngryBots scene 15, 16
nbBullets variable 83
nbLives variable 140
numbers of lives
tracking 132, 139, 140

O

objects
adding, to scene 17
collecting 52-55
creating 67, 68
cube, creating 18-21
dots, creating for 76
imported objects, inserting 23
interacting, script used 43
texture, adding to 22
texture, adding to objects 22
tracking 67, 68
OnControllerColliderHit function 54, 56, 64, 128, 133, 134

P

parameters
creating 94-97
Particle Animator component 86
particle emitter 84
path
defining, waypoints used 110-112
patroller object 127
Patrol state 110, 111
Physics.OverlapSphere method 102
Physics.Raycast function 80
Play button 27
playerBreadCrumb script 141
prefabs
creating 127-132
updating 127-132
Programming Game AI by Example
URL 143
project
creating 16, 17
Project view 11, 14, 68, 93, 103
Project window 32, 46, 92

R

repeated shots
allowing 87

Rig tab 103
Rig tag 90
rocks
creating 34, 35
rotate.js script 128
Rules of Play
URL 143

S

scale property 56, 76
scene
creating 16, 17
navigating through 13
object, adding to 17
scene view
about 11, 12, 40, 49, 72, 93, 96, 120
navigating through 12
Screen.showCursor variable 80
script
creating 45-50, 51, 137
used, for interacting with objects 43
Search field 22, 33, 90
seconds variable 49
Set button 16
setHealth function 71
setWalking function 101
Shader property 76
Shoot Bullet component 84
shootBullet script 83, 84, 106
Sketchup
URL 144
SphereCollider component 76
splash screen scene
creating 135
startDoor2Timer variable 134
Start function 46, 51, 63, 82, 107, 124, 130, 131
startTimer function 59
style variable 69
switch statement 99
switch structure 108-111
system requisites
URL 9

T

Tag Manager window 58
Text attribute 50, 136
textToDisplay variable 51
texture
adding, to objects 22
URL 22
Theory of Fun
for Game Design, URL 143
third-person controller
adding 27, 28
third-person view
implementing 24, 25
Time.deltaTime variable 48
timeForNextShot variable 87
timerIsActive variable 59
timer script 46, 50, 61, 132
timer variable 48, 59
timeToReload variable 87
time variable 47, 49
toBeDisplayed variable 62
topView label 74
transitions
creating 94-97
creating, between states 103

U

Unity3D
about 7, 8, 144
advantages 8, 9
archive, URL 10
assets store, URL 9
downloading 9, 10
for Mac OS, URL 9
for Windows, URL 9
launching 10
scripting 44
URL 8, 9
Unity3D 4 9
Unity3D interface
about 11
hierarchy view 11
inspector 11
project view 11
scene view 11
URL 17
Unity3D Learn
URL 144

Update function 46, 50-53, 59, 81, 98-119, 124, 132-136, 140

V

vertical walls
creating 36

W

WalkForward state 93, 96-101, 117-119, 122
walkForwardState variable 97
walking parameter 95, 100, 101
walking variable 120
water
adding, to water area 36
water tag 133
waypoints
used, for defining path 110-112
width property 79
withinReach parameter 104

Z

zombie_hires object 101, 112, 118
Zombie@walkForward 93

[PACKT] PUBLISHING

Thank you for buying
Getting Started with Unity

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

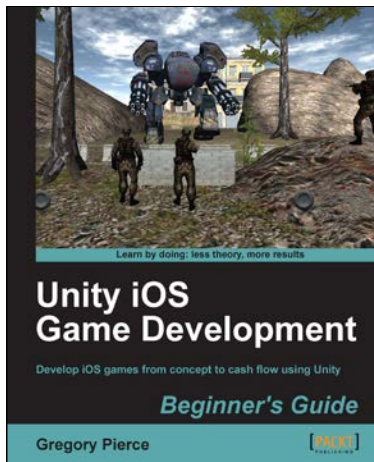
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



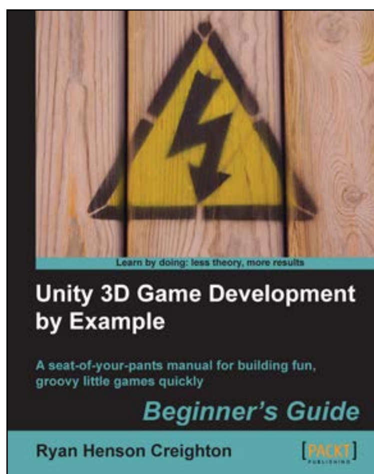
Unity iOS Game Development Beginners Guide

ISBN: 978-1-84969-040-9 Paperback: 314 pages

Develop iOS games from concept to cash flow using Unity

1. Dive straight into game development with no previous Unity or iOS experience
2. Work through the entire lifecycle of developing games for iOS
3. Add multiplayer, input controls, debugging, in app and micro payments to your game
4. Implement the different business models that will enable you to make money on iOS games

---



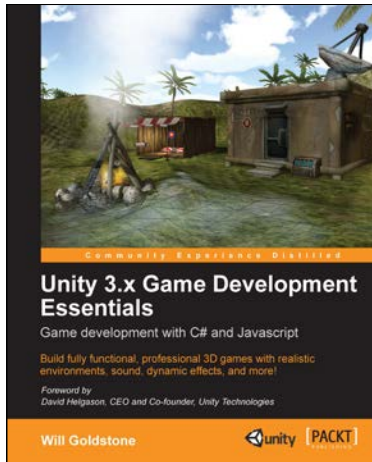
Unity 3D Game Development by Example Beginner's Guide

ISBN: 978-1-84969-054-6 Paperback: 384 pages

A seat-of-your-pants manual for building fun, groovy little games quickly

1. Build fun games using the free Unity 3D game engine even if you've never coded before
2. Learn how to "skin" projects to make totally different games from the same file – more games, less effort!
3. Deploy your games to the Internet so that your friends and family can play them
4. Packed with ideas, inspiration, and advice for your own game design and development

Please check **www.PacktPub.com** for information on our titles



Unity 3.x Game Development Essentials

ISBN: 978-1-84969-144-4 Paperback: 488 pages

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

1. Kick start your game development, and build ready-to-play 3D games with ease
2. Understand key concepts in game design including scripting, physics, instantiation, particle effects, and more
3. Test & optimize your game to perfection with essential tips-and-tricks

---

Unity 3 Game Development Hotshot

ISBN: 978-1-84969-112-3 Paperback: 380 pages

Eight projects specifically designed to exploit Unity's full potential

1. Cool, fun, advanced aspects of Unity Game Development, from creating a rocket launcher to building your own destructible game world
2. Master advanced Unity techniques such as surface shader programming and AI programming
3. Full of coding samples, diagrams, tips and tricks to keep your code organized, and completed art assets with clear step-by-step examples and instructions

Please check **www.PacktPub.com** for information on our titles