

Tommaso Lintrami

Unity 2017 Game Development Essentials

Third Edition

Build fully functional 2D and 3D games with realistic environments, sounds, physics, special effects, and more!



Packt

Unity 2017 Game Development Essentials

Third Edition

Build fully functional 2D and 3D games with realistic environments, sounds, physics, special effects, and more!

Tommaso Lintrami

Packt>

BIRMINGHAM - MUMBAI

Unity 2017 Game Development Essentials

Third Edition

Copyright © 2018 Packt Publishing

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

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

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

Commissioning Editor: Wilson D'souza
Acquisition Editor: Reshma Raman
Content Development Editor: Roshan Kumar
Technical Editor: Harshal Kadam
Copy Editor: Safis Editing
Project Coordinator: Devanshi Doshi
Proofreader: Safis Editing
Indexer: Aishwarya Gangawane
Graphics: Jason Monteiro
Production Coordinator: Shraddha Falebhai

First published: October 2009
Second edition: December 2011
Third edition: January 2018

Production reference: 1250118

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

ISBN 978-1-78646-939-7

www.packtpub.com

To my mother's memory, Carmela Bandera, who always believed in me and granted me my inclination toward IT and a good education. Thanks to my sister Viola, my partner Silvia, and my beloved Zoe, who grew up and took her first steps while this book was taking life, and who gives me daily inspiration for my creativity.

– Tommaso Lintrami



mapt.io

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

Why subscribe?

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

PacktPub.com

Did you know that Packt offers e-book versions of every book published, with PDF and ePub files available? You can upgrade to the e-book version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the e-book 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 e-books.

Contributors

About the author

Tommaso Lintrami has been programming games right from the age of 9; he studied music, micro electronics, and TV/cinema/game direction, and is a designer, developer, composer, and writer. He is a virtual reality and interactive installations enthusiast with at least 19 years, commercial experience working for several IT companies. He has been working with Unity for over 9 years now and delivers IT teaching and training in Unity. He has worked on a number of games on different platforms and is currently employed at Freejam, working on Robocraft.

About the reviewers

Ludovico Cellentani is a senior engine programmer at King AB, and he has been working as a professional game programmer for almost 20 years. During this time, he has worked on a number of games released on various platforms that span between PC, console, and mobile.

During the last 6 years, he has worked on a considerable number of games, VR experiences, and gamification projects released for PC, mobile, and custom-built computer installations, all powered by the Unity game engine.

He is currently living with his wife and son in Stockholm, Sweden.

Adam Larson has been a programmer on 15 titles across major consoles and PC platforms. He has worked on more than 50 mobile applications, with a few mobile games reaching more than a million downloads. Today, he runs and operates a studio in Green Bay, Wisconsin, called Zymo Interactive, where they focus on using high-end gaming lessons in business and mobile applications. His passion for building great products has led his team to work on some incredible projects in almost every conceivable industry.

Adam volunteers on the TEALS program to help get computer science in every high school across the United States. He also teaches part-time as an adjunct professor at the University of Green Bay. Adam is a father of three children and husband to Autum Larson. You may just find the five of them out playing Pokemon Go together.

Elias Tsiantas has been a programmer and content creator on five titles on mobile and PC platforms. Today, he works as a freelancer creating custom applications and/or content for companies on request. His passion is programming and 3Dmodelling/texturing since 25 years now. Elias is a father of four children and husband to Maria Vasileiadi.

Packt is searching for authors like you

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

Table of Contents

Preface	1
Chapter 1: Entering the Third Dimension	7
Getting to grips with 3D	7
Coordinates	8
Local space versus world space	9
Vectors	11
Cameras	12
Projection mode – 3D versus 2D	12
Polygons, edges, vertices, and meshes	13
Shaders, materials, and textures	15
Rigidbody physics	17
Collision detection	18
Softbody physics	19
The Cloth component	19
Getting to grips with 2D in 3D	19
Ignoring one axis	20
Understanding Sprites	20
Essential Unity concepts	20
The Unity way – an example	22
Assets	22
Scenes	23
GameObjects	23
Components	24
Scripts	24
Prefabs	25
The interface	26
The Scene view and Hierarchy view	27
Control tools	27
Scene navigation	28
Control bar	28
Create button	30
The Inspector	31
The Project window	32
The Game view	33

Summary	34
Chapter 2: Prototyping and Scripting Basics	35
<hr/>	
Your first Unity project	36
A basic prototyping environment	37
Setting the scene	38
Adding simple lighting	39
Another brick in the wall	41
Building the master brick	41
Creating our first prefab	46
And snap! It's a row	47
Grouping and duplicating with empty objects	48
Build it up, knock it down!	49
Introducing C# scripting	50
A new behavior script or class	50
What's inside a new C# MonoBehaviour class	51
Base MonoBehaviour methods	52
Variables in C#	53
Comments	53
Write the Shooter class	54
Declaring public variables	55
Assigning scripts to objects	55
Moving the camera	57
Local, private, and public variables	59
Understanding Translate	60
Implementing Translate	60
Testing the game so far	61
Making a projectile	63
Creating the projectile prefab	63
Creating and applying a material	63
Adding physics with a Rigidbody	64
Saving prefabs	65
Firing the projectile	66
Using Instantiate() to spawn objects	66
Adding a force to the Rigidbody	67
Resetting the wall to initial state and clearing the projectiles	69
Summary	70
Chapter 3: Creating and Setting Game Assets	71
<hr/>	
Setting up the scene and preparing game assets	73
Placing the prefabs in the game	76
Importing and placing background layers	77
Implementing parallax scrolling	79

Making it last forever	80
Using the Asset Store	80
The FreeParallax component	82
Understanding and setting up elements	83
Playing in the middleground	84
Avoiding holes and discrepancies on the ground collider	86
Putting in the foreground and special layers	87
Creating a death zone	88
Spawning collectable items	90
2D animation basics	93
Importing animated Sprites	93
The Animator component	97
Animator state-machine editor	98
Transitions between animation states	99
Conditions and parameters	99
Final words	102
Summary	102
Chapter 4: Player Controller and Further Scripting	103

Working with the Inspector	103
Tags	105
Layers	107
Prefabs and the Inspector	108
Scripting with Unity	108
Statements	109
Variables	109
Variable data types	110
Using variables	110
Public and private variables	111
Declaring variables	112
Full code example	112
Functions	113
Update()	114
OnMouseDown()	114
Writing custom functions	114
Return type	115
Arguments	116
Declaring a custom function	116
Calling a custom function	116
If else statements	118
Multiple conditions	120
For loops	120
Inter-script communication and Dot Syntax	122

Accessing other objects	122
Find() and FindWithTag()	122
SendMessage	123
GetComponent	124
Programming for mobile	125
Dot Syntax	125
Null reference exceptions	126
Coroutines	127
Comments	128
Further reading	128
Anatomy of a 2D character	129
Extending the Platformer2DUserControl class	129
Making the player and writing the game logic	129
Analyzing, understanding, and modifying the component's source code	130
Defining a namespace	132
Spawning collectible items	133
Adding a timer	136
Making things difficult with health and obstacles	138
Layer Collision Matrix	140
Making it look and sound better	140
Adding a simple shadow for the character	141
Inserting audio: environmental sfx, background music, sound events	142
Sprite shading with real-time lights	144
Writing some additional game logic	145
Homework	147
Summary	148
Chapter 5: Character Animation with Unity	149
<hr/>	
Unity Legacy Animation System	151
Importing character models and animations	152
Importing animations using multiple model files	153
Setting up the animation	154
Building the Player with Animator	156
What is an avatar in Unity?	157
Configuring an avatar	158
Fixing issues with the default bone mapping	163
Configuring muscle actions and settings	165
Importing animation clips for Animator	169
Setting up animations to ensure correct looping	171
Scaling the model	173
Understanding Animator	176
Animator component	176

Animator component properties	176
Animator controller	177
The Animator window	178
Animator state machine	179
Understanding the state machine	179
Controlling a character's behavior with state machines	180
Animation states	182
What's the Any State state?	183
Our first Blend Tree state	184
Conclusions	185
Controlling Animator through code with parameters	187
Root-motion animations	189
Using a better approach – Root motion	189
Animation transitions	192
Setting the transitions between states	193
Blend Trees	194
Modifying Unity standard assets classes to implement our playing character	196
Adding the final touches	198
Cloth simulation – giving life to hero's hair and skirt	198
Inverse Kinematics	201
Summary	205
Chapter 6: Creating the Environment	206

Designing the game	207
Understanding and using the Terrain tool	210
Setting terrain features	210
Setting the terrain size detail and resolution	211
Importing and exporting 2D heightmaps	213
The terrain tool	213
Terrain component	213
Raise height	215
Paint Height	216
Flatten Heightmap	217
Smooth Height	217
Paint Texture	218
Place Trees	220
Refresh (tree prototypes)	220
Edit Trees	220
Mass placement of trees	221
Paint details	222
Refresh (detail prototypes)	222
Edit details	222
Terrain Settings	223
Creating the island	224

Step 1 – Setting up the terrain	225
Step 2 – Creating the island outline	225
Step 3 – A small lake carving	227
Step 4 – Adding textures	228
Painting procedure	230
Sandy areas	231
Adding grass and rock	232
Step 5 – Tree time	233
Step 6: The grass is always greener	235
Step 7 – Let there be light!	238
Creating sunlight	239
Procedural skybox	240
Step 8 – Surrounding the island with sea water	242
A small lake	247
Step 9: What's that sound?	249
Positional audio versus non-positional audio	249
Audio file formats	250
The hills are alive!	250
Importing the book's asset package	251
Further audio settings	253
Enabling positional 3D sound and setting up curves	253
Step 10 – Walkabout	254
Step 11: Final tweaks	256
Summary	257
Chapter 7: Interactions, Collisions, and Pathfinding	258
Digital content creation applications	259
Common import settings for models	259
Model	260
Materials	261
Animation	261
Wrap Mode	262
Animation Compression	262
Setting up the hut model	263
Adding the model	264
Positioning the model	265
Manually adding the colliders	266
Physic Material	268
Adding audio	269
Collisions and triggers overview	269
Ray casting	273
Predictive collision detection	274

Opening the old man's hut door	276
Approach 1 – collision detection	276
Creating new assets	276
Character collision detection	278
Working with OnCollisionEnter	278
Writing the OpenDoor() method	280
Declaring the function	280
Checking the door status	281
Playing an audio event	281
Testing the script	282
Extending colliders	284
Playing door animations through Animator	285
Reversing the procedure	287
Code maintainability	289
Drawbacks of collision detection	291
Approach 2 – ray casting	292
Disabling collision detection with comments	292
Refactoring the code – writing a Door Manager class	293
Tidying PlayerCollisions	294
Casting the ray	295
Resetting the collider	297
Approach 3 – trigger collision detection	298
Creating and scaling the trigger zone	298
Scripting for trigger collisions	300
Removing the PlayerCollisions component	301
Placing additional models	301
Unity Navigation System	302
The Bake tab	303
The Agents tab	305
The Object tab	305
The Areas tab	306
Area Types and Navigation cost	307
Area mask	308
Summary	308
Chapter 8: AI, NPC, and Further Scripting	309
<hr/>	
Creating an NPC	310
Initial code and Animator Controller	314
Animator transitions	316
Navigation setup	319
Baking the NavMesh	319
Adding the rock stone and NPCsitSpot and NPCStartSpot points	320
Writing the Simple AI class	320
NPC interaction	324

Triggering the dialogue	325
Writing the DialogueManager class	329
Tying up the logic and UI events	334
Driving Animator Blend Tree with scripting	335
Creating a basic UI for displaying the dialogue	336
Creating the Canvas	337
Creating the start dialogue prompt	339
Creating the dialogue window	340
Creating the answer buttons	340
Making enemy AI	341
Using Unity Standard Assets in our favor	342
Making the AI smarter	343
Duplicate ThirdPersonCharacter class	344
Creating the Advanced AI Controller class	345
Custom AI state machine	347
Waypoints roaming	348
Chasing the target	348
Back to waypoints roaming	349
Enemy's Field Of View (or line of sight)	349
Player presence awareness	353
Fighting the player	353
Modifying the chase method	355
Debugging the Nav Mesh Agent	357
A better look for our custom components with PropertyDrawers	359
Further lectures and ideas	360
Tips for enhancing the AI	361
Summary	361
Chapter 9: Item Collection, Player Inventory, and HUD	362

Creating the ancient artifact piece prefabs	364
Downloading, importing, and placing	364
Tagging the artifact piece	365
Collider scaling and rotation	365
Enlarging the artifact piece	366
Adding a trigger collider	366
Collider scale and custom point light	366
Creating the artifact piece collection script	367
Making it spin	368
Adding trigger collision detection	369
Saving as a prefab	370
Placing the artifact pieces	371
Player inventory	372
Saving the artifact collected piece count value	373
Setting the variable start value	373

Audio feedback	374
Adding the PiecePickup() method	374
Adding the inventory to the player	374
Restricting hidden piece spot access	375
Restricting dialog access with a piece counter	375
Displaying the game progression status HUD	376
Import settings for UI images	377
Creating the HUD panel and background UI image	378
Scripting for UI image activation	382
Understanding arrays	382
Adding QuestIndicator images	383
Draw order of elements	384
Disabling the HUD for game start	386
Enabling the HUD	388
Hints for the player	389
Writing on screen with UI Text	389
Scripting for UI Text control	390
Adjusting hints to show progress	393
Summary	395
Chapter 10: Instantiation and Rigidbodies	396
<hr/>	
Implementing instantiation	397
Physics	399
Forces	399
The Rigidbody component	400
Designing the gameplay	402
Creating the heavy stone prefab	402
Adding physics to the stone prefab	403
Saving as a prefab	403
Throwing stones at guards	404
The StoneLauncher component	404
Checking for player input	405
Writing the Fight method	406
Instantiating the heavy stone	406
Naming instances	408
Assigning velocity	408
Adding development safeguards	409
Checking component presence	409
Safeguarding collisions	410
Using the IgnoreCollision() method	410
Ignoring collisions with layers	411

Final tweaks	413
Instantiation restriction and object tidying	414
Implementing stone throw animation	415
Activating the stone launch at the right frame	419
Removing stones in a smart way	421
Finishing touches	423
Ragdoll physics simulation	423
Scripting the ragdoll simulation	425
Reviving the enemy	426
Using coroutines to time game elements	427
Collision detection	430
Punching the guards	431
Playing stones collisions feedback sounds	432
Adding force impulse to stones impacts	433
Summary	433
Chapter 11: Unity Particle System	434

What is a Particle System?	435
Creating the fireplace	438
Creating wall torches	442
Modifying the Fire Light component	445
Enhancing environment ambience	448
Making the village ground dusty	448
Creating a sea breeze particle system	450
The Gradient Editor	452
Creating a waterfall	455
Summary	457
Chapter 12: Designing Menus with Unity UI	458

Unity UI	459
Canvas render modes	460
Screen space – overlay	461
Screen space – camera	462
World space	463
Preparing textures for UI usage	464
Creating the main menu scene	466
Adding the game title	468
Manually anchoring the title to the Canvas	470
Creating the main menu panel	472
Adding buttons	473
Clone the MainMenu to obtain the Options menu	473
Configuring UI buttons to show/hide menus with the OnClick() event method	474

Creating an audio options menu	475
Adding music volume control	476
Adding general volume control	477
Writing a listener script for UI slider elements	477
User interaction	479
Creating a video options menu	479
The power of UI dynamic variables and UI events	483
Creating a drop-down menu	485
Yet another slider controller	486
Loading the game	488
Final touches	489
Lights and shadows	489
Adding a Rim light	490
Real-time shadows	490
Post-processing image effects	491
FSAA (Full Screen Anti-Aliasing)	491
Bloom and HDR	492
Screen overlay	492
Vignette and chromatic aberration	492
Splitting the render on two different cameras	492
Adding jail bars to the scene	494
Conclusion	496
Testing screen sizes	496
Further looks	497
Summary	498
Chapter 13: Optimization and Final Touches	499
Tweaking the terrain	500
Using SpeedTree	500
Hills, troughs, rocks, and texture blending	504
Smoothing and painting	505
Keep on the right path by drawing path details with splat maps	506
Some level design: placing the guards	508
Unity lighting	511
Scene setup	512
The Environment settings	512
Realtime lighting and Mixed Lighting	514
Realtime only versus Baked only versus Mixed lighting	514
Lightmaps and baked Global Illumination	515
Baking the scene	516
Preparing for lighting	516
Including or excluding lights from the baking	517
Excluding GameObjects from the bake	519
Baking the scene	519

Lightmapping Settings	520
Global maps	523
Object maps	525
Other settings	525
Optimizing performance	527
Camera clip planes (frustum culling)	527
Standard fog versus Global Fog post effect	529
Occlusion culling	530
The Object tab	531
The Bake tab	531
The Visualization tab	533
Wrapping it all up	535
Rendering paths	535
Graphics pipelines	536
Forward Rendering	536
Deferred Rendering	537
Lighting performance	537
Should I use Deferred?	539
Physical-Based Rendering: Unity Standard Shader	539
Image effects	540
Hardware-based anti-aliasing (MSAA) versus shader-based anti-aliasing(FSAA)	540
Depth of field	541
Debugging depth of field	542
Crepuscular sun rays through Sun Shafts effect	542
The Post Processing Stack	543
Post Processing Stack V2 and Utilities	544
Focus puller	545
Advanced rendering features	545
Level of Detail (LOD)	545
High Dynamic Range	546
Asynchronous Texture Upload	546
Graphic Command buffers	547
Unity Engine automated optimizations	547
Static and dynamic batching	547
Static batching	547
Dynamic batching	548
GPU instancing	549
Cull and cull more!	553
Camera culling distance	554
Modifying the FireLight class	555
Further optimizations	556
Physics optimizations	557
Writing the DeactivateRagdollTimer class	558

Changing the HeavyStone class	560
Exercise proposal: a rough save game-status feature	563
Optimizing AI impact on the CPU	564
Setting up AI Trigger areas	565
Other ideas	567
Summary	567
Chapter 14: Building and Sharing	568

Supported platforms	570
PC (Windows), Linux, or Mac standalone	570
Android platform	570
iOS	571
WebGL	571
Virtual, augmented, and mixed reality	572
Player settings	572
Cross-platform general settings	573
Per-platform player settings	574
Mac - PC - Linux - standalone build	574
Resolution and Presentation	575
User input bindings	576
Icon	577
Splash image	578
Splash screen	579
Logos	580
Background	581
Other settings	581
Rendering	582
Configuration and optimization	584
Logging	585
XR settings	585
Android	586
iOS	588
WebGL	589
Player input settings	589
Graphics settings	591
Tier settings	592
Quality settings	592
Quality settings - Rendering	593
Quality settings - Shadows	595
Quality settings - Other	596
Building the game	597
Build settings	597

Quit button with platform automation	599
Our first build	600
Building the standalone for PC/Mac/Linux	600
Adapting for the web	601
Building for the web	602
Adapting and building for the mobile platform	603
Adapting for Android	604
Texture Compression formats	604
Building for Android	605
Choosing the preferred build system	606
Publishing settings	607
Building for iOS	608
Testing and debugging	609
Debugging with Visual Studio 2017	610
Testing and profiling with Unity Editor	611
Unity Profiler	612
The Physics Debug window	614
The Frame Debug window	614
Sharing your work	615
Sharing WebGL builds	616
Publishing on mobile stores	617
Publishing for the desktop	617
Digital Content Creation tools	618
Future of Unity and MonoBehaviour	618
Testing and further study	619
Learn by doing	619
Testing and finalizing	620
Public alpha testing and open beta	620
Frame rate feedback	621
Testing different video resolutions	624
Optimizing more	625
Approaches to learning	625
Don't reinvent the wheel	626
Editor extensions	626
Complete projects	628
If you don't know how to do it, just ask!	628
Summary	629
Other Books You May Enjoy	630
Index	633

Preface

Game engines such as Unity are the power tools behind the games we know and love. Unity is one of the most widely used and best loved packages for game development and is used by everyone, from hobbyists to large studios, to create games and interactive experiences for the web, desktops, mobiles, and consoles. With Unity's intuitive, easy-to-learn toolset and this book, it's never been easier to become a game developer.

Taking a very practical approach, this book will introduce you to the concepts of developing 2D and 3D games before getting to grips with development in Unity itself, prototyping a simple 2D scenario, and then creating a larger 3D game. From creating 3D worlds to scripting and creating game mechanics, you will learn everything you'll need to get started with game development.

This book is designed to cover a set of easy-to-follow examples, which culminate in the production of a Third Person 3D game, complete with an interactive island environment, dialogues, puzzles, and AI-driven enemies to fight. All the concepts taught in this book are applicable to other types of game; however, by introducing the common concepts of game and 3D production, you'll explore Unity to make a character interact with the game world, with NPCs, and with enemies, and build puzzles for the player to solve in order to complete a game level. At the end of the book, you will have a fully-working 3D game and all the skills required to extend the game further, giving your end user, the player, the best possible experience through a series of optimization and performance-tweaking solutions; soon, you will be creating your own 3D games and interactive experiences with ease!

Who this book is for

This book is intended for beginners and intermediate users and game developers in general. No previous Unity experience is needed, but some basic knowledge about programming is required to get to grip with C#.

What this book covers

Chapter 1, *Entering the Third Dimension*, introduces the key concepts that you need to understand to complete the book. However, 3D is a detailed discipline that you will continue to learn not only with Unity, but in other areas as well. We'll ensure that you're prepared by looking at some important 3D concepts before moving on to discuss the concepts and interface of Unity. You will learn more about coordinates and vectors, physics simulation, and about the Unity Editor basic user interface.

Chapter 2, *Prototyping and Scripting Basics*, familiarizes you with the Unity interface, working with GameObjects, components, and basic scripting. You will also learn about creating a new project in Unity, and you will be introduced to variables, functions, and commands.

Chapter 3, *Creating and Setting Game Assets*, teaches how Unity 3D manages 2D projects, the orthogonal camera, lighting, how to implement parallax scrolling backgrounds, and how to import animations from sprite sheets. You will learn to modify and customize the FreeParallax component and also take a first look at the Animator Controller.

Chapter 4, *Player Controller and Further Scripting*, dives a bit deeper into scripting, modifying existing classes, or creating your own. It also takes a look at how the new Physics2D framework works with different types of Colliders 2D and Rigidbody 2D. Also, you will become familiar with how to modify C# classes for your needs, how collision and trigger detection work in 2D, and will take a first look at Unity UI concepts. You will be introduced to the main features of the Inspector and take a Dot syntax and quick scripting course.

Chapter 5, *Character Animation with Unity*, looks into Unity's animation system and Mecanim. You will also see the difference between the Root Motion and non-Root Motion approach to avatar animations and discover how to set up both a basic and a complex state-machine to control the character animation behavior, as well as how IK pass in a layer can allow us to control a part of the skeleton through scripting. You will be introduced to how to modify the ThirdPersonCharacter setup from the Standard Assets to suit our player controller.

Chapter 6, *Creating the Environment*, assists with the basics of creating your first environment. Beginning with nothing but a flat plane, you will create a completely explorable island in a short span of time. You will also look at lighting and audio, two core principles that you'll apply in every kind of game project you encounter. Specifically, you will be looking into using Audio component to implement environmental ambient sounds.

Chapter 7, *Interactions, Collisions, and Pathfinding*, focuses on how to detect interactions between objects in 3D games with collision and raycasting. You will learn how to use the basics of pathfinding and how to set up the level you are making to optimize the NavMesh with Unity's built-in Navigation System.

Chapter 8, *AI, NPC, and Further Scripting*, teaches you to create a non-playing character visual prefab and its Animator Controller and components, a simple AI class for the NPC and the guards first, and an advanced AI class for the guards later on. You will also be using Unity UI, triggers, and some scripting to make the player interact with the NPC and implement a simple dialogue. Also, you will be setting up the Navigation System for NavMesh baking and will test the navigation with character movements.

Chapter 9, *Item Collection, Player Inventory, and HUD*, outlines creating a game scenario. Learn to display onscreen text with the UI Text component and how to work with the UI Image component to make an HUD, and learn how to manipulate game state based on tracked variables. You will create an item collection puzzle for the player.

Chapter 10, *Instantiation and Rigidbodies*, looks at implementing Rigidbody objects both for animated dynamic elements and instantiated projectiles. You will be looking at how coroutines can be used to provide structure and added functionality to your scripting. By creating a simple fight system for managing punching and launching stones, you will be introduced to setting up a ragdoll simulation and scripting a ragdoll manager for reviving stunned enemies.

Chapter 11, *Unity Particle System*, discusses how to build some cool particle effects and how to customize the particle system for our needs. You will also learn about how multiple particle systems can play together, and how to combine the same effect for different situations and learn to modify and adjust the FireLight component for moving around and changing the color/intensity of fire Point Lights.

Chapter 12, *Designing Menus with Unity UI*, helps you create the main menu scene from where the user will start the game or quit to windows. You will learn how to use animations on UI panels and how to catch mouse events and execute procedures upon firing an event. Also learn to create an in-game/pause menu feature with a World Space Canvas and also prepare texture images to work with the new Unity UI.

Chapter 13, *Optimization and Final Touches*, explores how to operate further optimization techniques and how to profile your game to find bottlenecks. You will look at standard Image Effects and be introduced to the new Post Processing Stack. You will add some cool bits and prepare everything for the final build.

Chapter 14, *Building and Sharing*, showcases the supported platforms and how to adapt your game for desktop, web, or mobile builds. You will explore Unity Player Settings and Build Settings and the main differences between the various platforms, and learn how to share your work with others.

To get the most out of this book

I assume that you are confident with modern programming; it's better if you have some previous experience with C#, but it's not mandatory. Decent knowledge of computer graphics in general and an interest in learning are a must. We suggest you to install The Gimp for editing textures and sprites, and Blender for editing 3D models.

Despite the difficulty they might have at start, they are very complete packages with nothing to envy of packages such as Photoshop or Maya. Of course, this is your call; if you are already a Photoshop guru or Maya expert, there is no need to tell you to keep using your favorite package!

Download the example code files

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

You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Unity-2017-Game-Development-Essentials-Third-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/Unity2017GameDevelopmentEssentialsThirdEdition_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "And, finally, we will add a method function to our `2DPlatformerCustomUserControl`, which our `collectable2D` class will call by sending a message to the player object to increase the score. You can place it right after its `Update()` function."

A block of code is set as follows:

```
bool grounded = false;
float speed;
void Update() {
    if (grounded == true) {
        speed = 5.0f;
    }
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In **Terrain Settings**, for example, you can adjust the **Base Map Dist.** in order to specify how far away a player must go before the terrain replaces high-resolution graphics for lower-resolution ones, making objects in the distance less expensive to render."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

1

Entering the Third Dimension

Before getting started with any 3D package, it is crucial to understand the environment you'll be working in.

As Unity was born as a 3D engine and development tool, many concepts throughout this book will assume a certain level of understanding of 3D development in general, as well as game engines. It is crucial that you equip yourself with at least a basic knowledge of these concepts before diving into the practical elements of the rest of this book.

In this chapter, we'll make sure you're prepared by looking at some important 3D concepts, before moving on to discuss the concepts and interface of Unity itself. You will learn about the following topics:

- Coordinates and vectors
- Polygons, edges, vertices, and meshes
- Shaders, materials, and textures
- Rigidbody physics simulation and collision detection
- Introduction to Sprites and the new 2D and **User Interface (UI)** systems
- game objects and scripted or built-in components
- Assets and scenes, the Project view
- Unity Editor UI

Getting to grips with 3D

Let's take a look at the crucial elements of the 3D world, and how Unity lets you develop games in three dimensions.

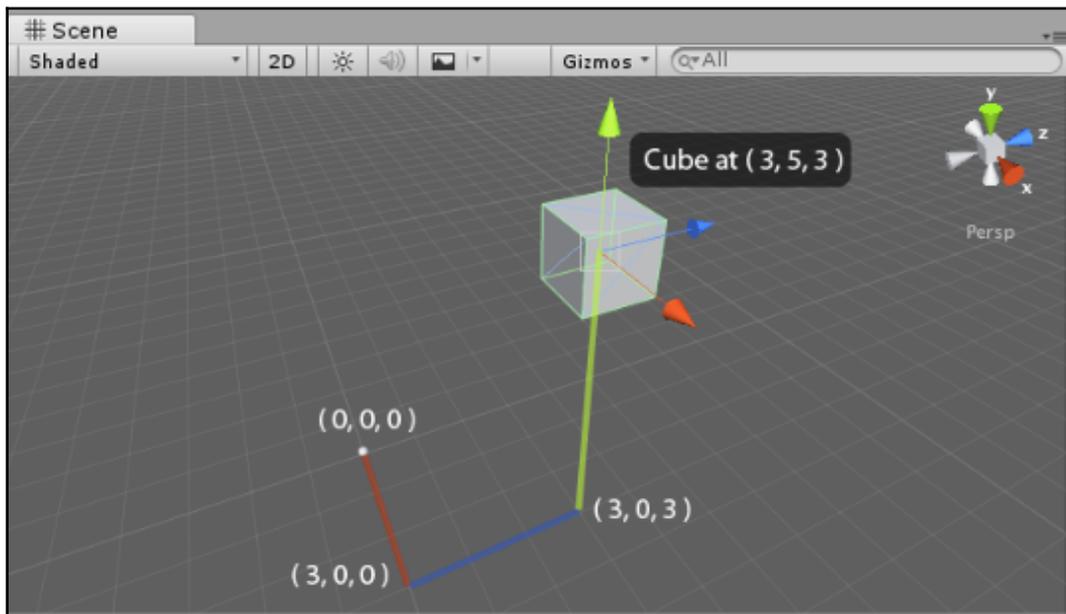
Coordinates

If you have worked with any 3D application before, you'll likely be familiar with the concept of the z axis. The z axis, in addition to the existing x for horizontal and y for vertical axes, represents depth. In 3D applications, you'll see information on objects laid out in an x, y, z format—this is known as the **Cartesian coordinate** method. Dimensions, rotational values, and positions in the 3D world can all be described in this way. In this book, as in other documentation of 3D, you'll see such information written with parentheses, shown as follows: **Cube at (3,5,3)**.

This is mostly for neatness, and also due to the fact that, in programming, these values must be written in this way. Regardless of their presentation, you can assume that any set of three values separated by commas will be in an x, y, z order.

This convention is derived from Maya, where the y axis is the up axis while z is the forward axis. Not all graphic packages and engines use this coordinate system. For example, in 3DS Max and Vicious Engine, y is the forward axis while z is the up axis.

In the following screenshot, a cube is shown at the location **Cube at (3,5,3)** in the 3D world, meaning it is 3 units from 0 in the x axis, 5 up in the y axis, and 3 forward in the z axis:



Local space versus world space

A crucial concept to begin looking at is the difference between local space and world space. In any 3D package, the world you will work in is technically infinite, and it can be difficult to keep track of the location of objects within it. In every 3D world, there is a point of origin, often referred to as the origin or world zero, as it is represented by the position **(0,0,0)**.

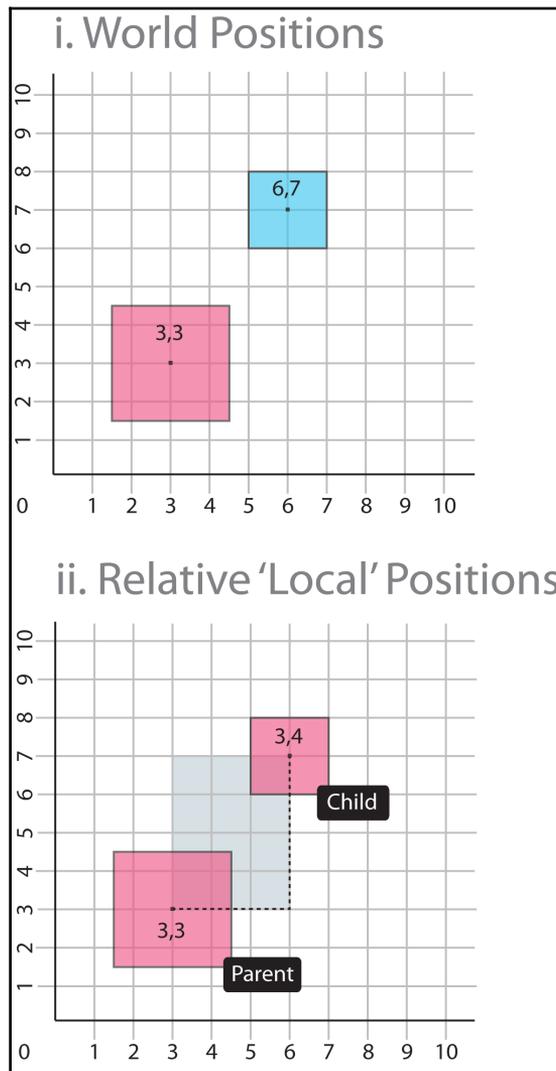
All world positions of objects in 3D are relative to world zero. However, to make things simpler, we also use local space (also known as object space) to define object positions in relation to one another. These relationships are known as parent-child relationships. In Unity, parent-child relationships can be established easily by dragging one object onto another in the hierarchy. This causes the dragged object to become a child, and its coordinates from then on are read in terms relative to the parent object. For example, if the child object is at exactly the same world position as the parent object, its position is said to be **(0,0,0)**, even if the parent position is not at world zero.

Local space assumes that every object has its own zero point, which is the point from which its axes emerge. This is usually the center of the object, and by creating relationships between objects, we can compare their positions in relation to one another. Such relationships, known as parent-child relationships, mean that we can calculate distances from other objects using local space, with the parent object's position becoming the new zero point for any of its child objects.

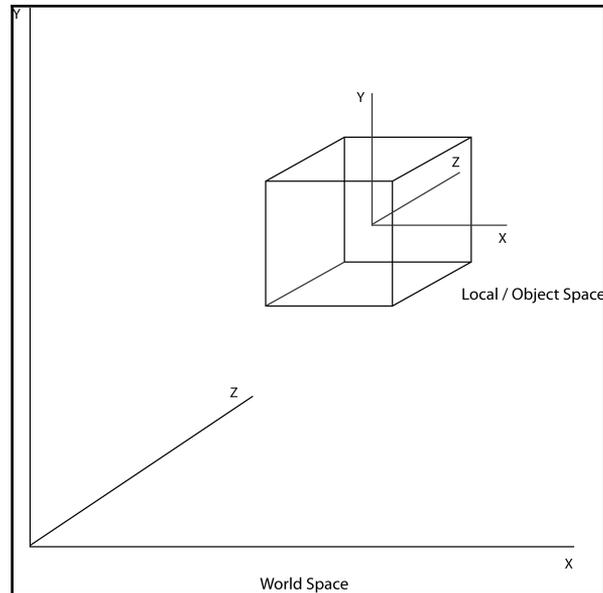
We can illustrate this in 2D, as the same conventions will apply to 3D. These bullet points explains the following figure:

- The first diagram, **i. World Positions**, shows two objects in world space. A large cube exists at coordinates **(3,3)**, and a smaller one at coordinates **(6,7)**.

- In the second diagram, **ii. Relative 'Local' Positions**, the smaller cube has been made a **Child** object of the larger cube. As such, the smaller cube's coordinates are said to be **(3,4)**, because its zero point is the world position of the **Parent**:



To extrapolate this to three dimensions, the same rule applies but with the addition of a z axis:



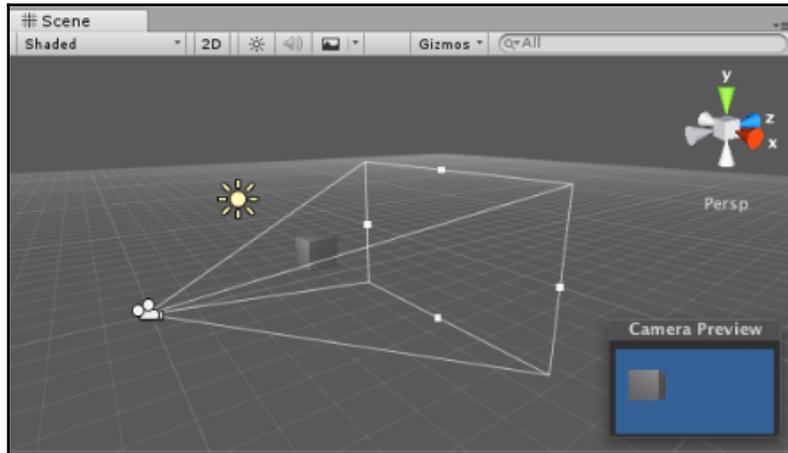
The beauty of using local space is that when a parent object is moved or rotated, the child object moves in relation to its parent. This often proves useful, not only to keep related objects together but also for practical applications during a game, such as making the origin of particles with their related 3D model objects.

Vectors

You'll also see 3D vectors described in Cartesian coordinates. Like their 2D counterparts, 3D vectors are simply lines drawn in the 3D world that have a direction and a length. Vectors can be moved in world space but remain unchanged themselves. Vectors are useful in a game engine context, as they allow us to calculate forces, distances, relative angles between objects, and the direction of an object.

Cameras

Cameras are essential in the 3D world, as they act as a viewport for the scene. Look at the following screenshot:



Cameras can be placed at any point in the world, animated or attached to characters or objects, as part of a game scenario. Many cameras can exist in a particular scene, but often a single main camera will always render the majority of what the player sees. This is why Unity gives you a main camera object whenever you create a new scene.

Projection mode – 3D versus 2D

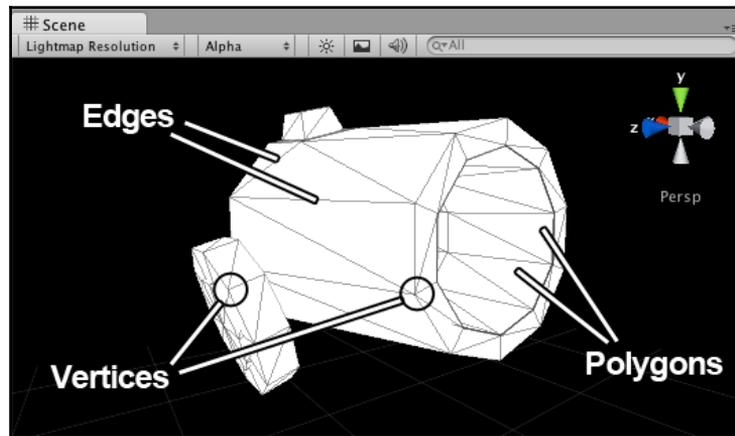
The projection mode of a camera states whether it renders in 3D (perspective) or 2D (orthographic). Ordinarily, cameras are set to the perspective projection mode and, as such, have a pyramid shaped **Field Of View (FOV)**. A perspective mode camera renders in 3D and is the default projection mode for a camera in Unity. Cameras can also be set to the orthographic projection mode in order to render in 2D, these have a rectangular FOV. This can be used on a main camera to create complete 2D games, or simply used as a secondary camera to render **Heads Up Display (HUD)** elements such as a map or health bar.

In game engines, you'll notice that effects such as lighting, motion blurs, and other effects are applied to the camera to help with the game simulation of a person's view of the world. You can even add a few cinematic effects that the human eye will never experience, such as lens flares or depth of field, which have more in common with cameras than our eyes, but are a given cinematic convention.

Most modern 3D games utilize multiple cameras to show parts of the game world that the character camera is not currently looking at, like a *cutaway* in cinematic terms. Unity does this with ease by allowing many cameras in a single scene, which can be scripted to act as the main camera at any point during runtime. Multiple cameras can also be used in a game to control the rendering of particular 2D and 3D elements separately as part of the optimization process. For example, objects may be grouped in layers, and cameras may be assigned to render objects in particular layers. This gives us more control over individual renders of certain elements in the game. We will look into these concepts in more depth later when explaining UIs and HUD.

Polygons, edges, vertices, and meshes

In constructing 3D shapes, all objects are ultimately made up of interconnected 2D shapes known as **Polygons**. On importing models from a modeling application, Unity converts all polygons to polygon triangles. Polygon triangles (also referred to as faces) are in turn made up of three connected **edges**. The locations at which these **Edges** meet are known as points or **Vertices**. Have a look at the following screenshot:

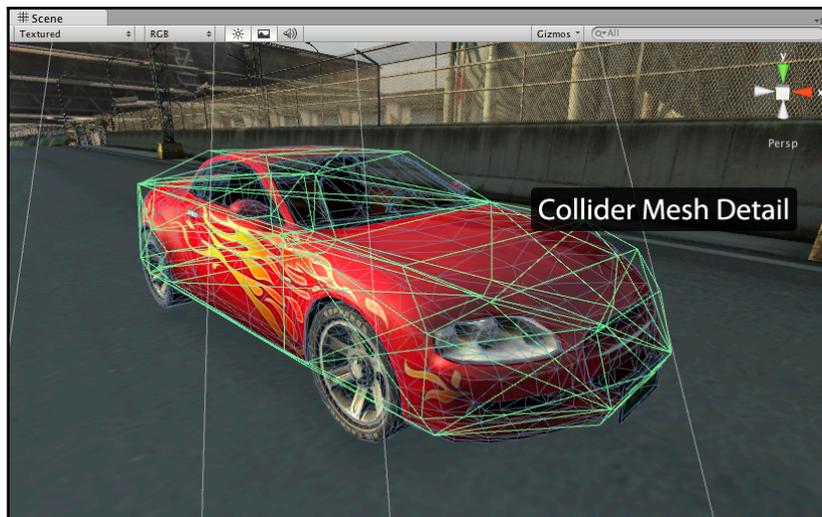


By knowing these locations, game engines are able to make calculations regarding the points of impact, known as collisions, when using complex collision detection with **Mesh Colliders**, such as in shooting games to detect the exact location at which a bullet has hit another object. By combining many linked polygons, 3D modeling applications allow us to build complex shapes known as meshes. In addition to building 3D shapes that are rendered visibly, mesh data can have many other uses.

For example, it can be used to specify a shape for collision that is less detailed than a visible object, but roughly the same shape. This can help save performance as the physics engine needn't check a mesh in detail for collisions. This is seen in the following screenshot from the Unity car tutorial, where the vehicle itself is more detailed than its collision mesh:



In the following screenshot, you can see that the amount of detail in the mesh used for the collider is far less than the visible mesh itself:



In game projects, it is crucial for the developer to understand the importance of the **polygon count**. The polygon count is the total number of polygons, often in reference to the models, but also in reference to the props or an entire game level (or in Unity terms, *scene*). The higher the number of polygons, the more work your computer must do to render the objects onscreen.

This is why, as the years progress, we've seen an increase in the level of detail from early 3D games to those of today. Simply compare the visual detail in a game such as id's *Quake* (1996) with the detail seen in Epic's *Gears Of War* (2006) just a decade later, then fast forward to a game such as *Call of Duty Advanced Warfare* (2014), which saw enough detail to render recognizable actors such as *Kevin Spacey*.

As a result of faster technology, game developers are now able to model 3D characters and worlds for games that contain a much higher polygon count and resultant level of realism, and this trend will inevitably continue in the years to come; this, combined with increasingly detailed shaders, means we get more realism as time progresses. That said, as more platforms emerge, such as mobile, VR, and online 3D such as WebGL, games previously seen on dedicated consoles can now be played on lower-powered devices thanks to Unity. As such, the hardware constraints are as important now as ever, as low-powered devices such as mobile phones and tablets are able to run 3D games, while VR requires higher performance to keep a consistent frame rate and avoid nausea for the player. For these reasons, when modeling any object to add to your game, you should consider polygonal detail and where it is necessary.

Shaders, materials, and textures

Materials are a common concept to all 3D applications as they provide the means to set the visual appearance of a 3D model. From basic colors to reflective image-based surfaces, materials handle everything.

Let's start with a simple color and the option of using one or more images, known as **textures**. In a single material, the material works with the **shader**, which is a script in charge of the style of rendering. For example, in a reflective shader, the material will render reflections of surrounding objects but maintain its color or the look of the image applied as its texture.

In Unity, the use of materials is easy. Any materials created in your 3D modeling package will be imported and recreated automatically by the engine and created as assets that are reusable. These assets are given the standard shader in Unity, which approximates the settings and texture assignments from most modeling applications, so if you have created an albedo or normal map in your content authoring tool, you will see this assigned correctly when importing in a new material. You can also create your own materials from scratch, assigning images as textures and selecting a shader from a large library that comes built-in. You can also write your own shader scripts or copy and paste those written by fellow developers in the Unity community, giving you more freedom for expansion beyond the included set.

Because Unity automatically texture downsizing at import time, it's better to start with the highest resolution possible, before eventually letting the importer scale down your textures for slower platforms such as mobile.

When creating textures for a game in a graphics package such as Photoshop or **GNU Image Manipulation Program (GIMP)**, you must be aware of the resolution. Larger textures will give you the chance to add more detail to your textured models, but they will be more intensive to render. Game textures imported into Unity will be scaled to a power of two resolution, for example:

- 128 px x 128 px
- 256 px x 256 px
- 512 px x 512 px
- 1024 px x 1024 px
- 2048 px x 2048 px
- 4096 px x 4096 px

In the case of rectangular textures, animated textures, or a texture atlas (see Chapter 13, *Optimization and Final Touches*), you can also use sizes such as the following:

- 256px x 64px
- 4096px x 512px



It's important for certain hardware that both the width and height of the textures are numbers that are a power of two.

Creating textures of these sizes with content that matches the edges (seamless) will mean that they can be tiled successfully by Unity, though it is always worth including higher-resolution textures that you can use by default, scaling them down for slower platforms when it comes time to build the game. In the case of graphics imported for use with UIs instead, Unity will use the original size even for uneven images, for example 392 px x 157 px, as you will discover over the course of this book. To optimize performance, Unity 2017 will automatically pack uneven UI textures into a bigger texture atlas, which will be the power of two, to send to the GPU.

Rigidbody physics

For developers working with game engines, physics engines provide an accompanying way of simulating real-world responses for objects in games. In Unity, the game engine uses Nvidia's PhysX engine, a popular and highly accurate commercial physics engine.



Note that this is built-in behind the scenes and you'll never need to license it directly as it is part of Unity engine.

In game engines, there is no assumption that an object should be affected by physics, firstly, because it requires additional processing power, and secondly because there is simply no need to do so. For example, in a 3D driving game, it makes sense for the cars to be under the influence of the physics engine, but not the track or surrounding objects, such as trees, walls, and so on, they will remain static for the duration of the game. For this reason, when making games in Unity, a Rigidbody component is given to any object that you wish to be under the control of the physics engine.

Physics engines for games uses the Rigidbody dynamics system of creating realistic motion. This simply means that instead of objects being static in the 3D world, they can have properties such as mass, gravity, velocity, and friction.

As the power of hardware and software increases, Rigidbody physics is becoming more widely applied in games, as it offers the potential for more varied and realistic simulation. We'll be utilizing Rigidbody dynamics as part of our prototype in *Chapter 1, Enter to the Third Dimension*, and as part of the main game of the book in *Chapter 10, Instantiation and Rigidbodies*.

Collision detection

More crucial in game engines than in 3D animation, collision detection is the way we analyze our 3D world for inter-object collisions. By giving an object a **Collider** component, we are effectively placing an invisible net around it. This net usually mimics its shape and is in charge of reporting any collisions with other Colliders, making the game engine respond accordingly.

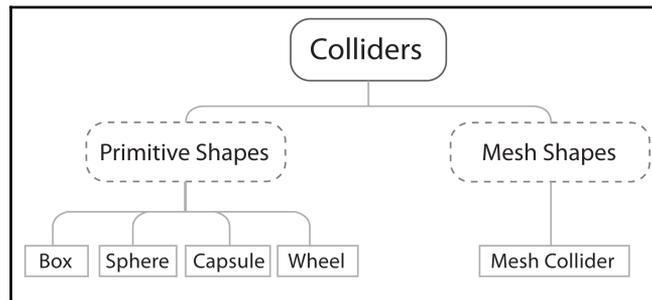
There are two main types of collider in Unity:

- Primitive
- Mesh

Primitive shapes in 3D terms are simple geometric objects such as box, sphere and capsule. Therefore, a Primitive Collider such as a Box Collider in Unity has that shape, regardless of the visual shape of the 3D object that is applied to it. Primitive Colliders are often used because they are computationally cheaper or because there is no need for precision.

A **Mesh Collider** is more expensive as it can be based upon the shape of the 3D mesh it is applied to; therefore, the more complex the mesh, the more detailed and precise the collider will be, and more computationally expensive it will become. However, as shown in the car tutorial example earlier, it is possible to assign a simpler mesh than that which is rendered in order to create simpler and more efficient Mesh Colliders.

The following diagram illustrates the various types and subtypes of **Colliders**:



Primitive and Mesh Colliders take a different amount of CPU to be calculated. The **Sphere** Collider is the faster one, while the **Mesh Collider** is the one to be calculated slower. The **Sphere** Collider is just a radius around a point in the world space to be calculated, while the **Mesh** Collider usually has a higher poly count, hence it is the most expensive Collider.

For example, in a ten-pin bowling game, a simple **Sphere Collider** will surround the ball, while the pins themselves will have either a simple **Capsule Collider** or, for a more realistic collision, employ a **Mesh Collider**, as this will be shaped the same as the 3D mesh of the pin. On impact, the **Colliders** of any affected objects will report to the physics engine, which will dictate their reaction based on the direction of impact, speed, and other factors.

In this example, employing a **Mesh Collider** to fit exactly to the shape of the pin model would be more accurate but is more expensive in processing terms. This simply means that it demands more processing power from the computer, the cost of which is reflected in slower performance, hence the term expensive. **Colliders** and Rigidbody components work together to perform collisions in the game. In addition, raycasting against **Colliders** needs a Rigidbody component present in the `GameObject`. Note that Rigidbodies and **Colliders** work together to perform the collision, and it is important to include both on an object that will be involved in collisions in your game.

Softbody physics

Even though Unity 5 does not have a big set of tools for Softbody physics, it provides a **Cloth** component that is used to emulate soft surfaces (such as clothes) physics behavior. It can be used to simulate soft items such as flags, curtains, hair, and so on.

The Cloth component

The Cloth simulation solution used in previous versions of Unity was quite expensive. In the latest version, clothes do not react to all the colliders in a scene. It also does not apply forces back to the world. Instead, as of Unity 5 onward, we have a faster, multithreaded, more stable character clothing solution, which involves creating specific colliders to make sure that the Cloth reacts correctly to the world. However, even in that scenario, the simulation is still one-way. The Cloth element does not apply forces back to the colliders. At the time of writing, the Cloth component is best used to simulate clothing or flags, but it's not really prepared to be used as a full Softbody physics solution.

Getting to grips with 2D in 3D

Since the latest versions, Unity has provided 2D-specific tools to create 2D and 2.5D games. Game developers have been using Unity for years to create 2D games, simply ignoring the z axis.

Ignoring one axis

As we discussed previously, Unity uses three numbers to represent positions in the 3D world. To work with 2D, we just need to ignore one of them. This means that in writing code to adjust transform properties, we can keep the z value constant, most often leaving it at 0. So, if you want an object to change from point $(0, 0)$ to $(0, 10)$, you would basically tell Unity to transform the object from $(0, 0, 0)$ to $(0, 10, 0)$, where z is constant as 0.

Since Unity now provides a whole set of tools for 2D games, you can define a new project as 2D when you create it in the Unity hub (the launcher which appears when you first run Unity). If you create a 2D project, the UI and other elements are already configured to work well in 2D.

You can actually configure an existing 3D project to be 2D, by setting the camera as orthographic and setting the view as 2D. Unity also provides 2D-specific physics for 2D games with many different `Colliders2D` and `Rigidbody2D` for its separate 2D physics engine.

Understanding Sprites

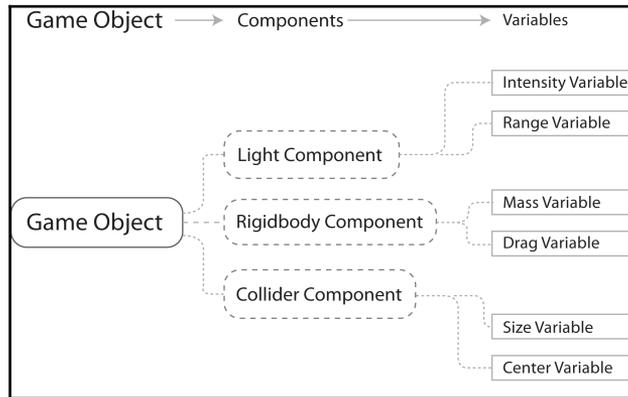
Sprites are 2D graphic objects used to render images. They can be used to hold characters, props, and even scene objects. Sprites can also be animated, either as elements in a character that moves or as a flip book. For the latter, Unity uses its animation system to exchange Sprites and create the animation. Animation frames can be inserted in a single Sprite and broken up in several images to be animated. This technique is called *sprite at lasing*, and the collections of animations in a single image are called *Sprite sheets*.

Sprites can also be used to create an atlas. An atlas is a big image that contains several Sprites. These are compressed into one image to be able to lower significantly the number of draw calls needed to display all the Sprites and video memory usage. Mobile games can gain a lot by using an atlas to hold all the visual elements and UI objects.

Essential Unity concepts

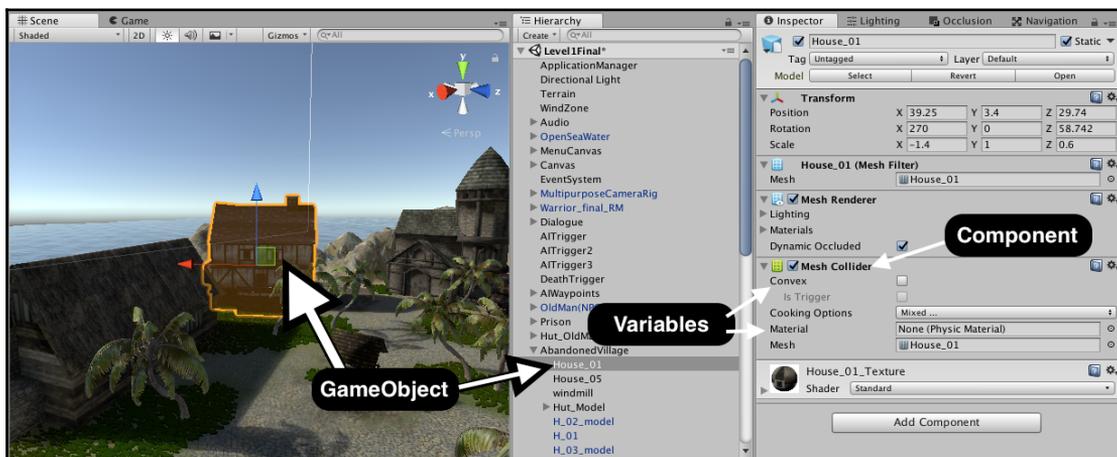
Unity makes the game production process simple by giving you a set of logical steps to build any conceivable game scenario. Renowned for being non-game-type specific, Unity offers you a blank canvas and a set of consistent procedures to let your imagination be the limit of your creativity.

By establishing the use of the **Game Object** concept, you are able to break down parts of your game into easily manageable objects, which are made of many individual component parts. By making individual objects within the game, introducing functionality to them with each component you add, you are able to infinitely expand your game in a logical, progressive manner. Component parts in turn have **Variables**, which are essentially properties of the component, or settings to control them with. By adjusting these variables, you'll have complete control over the effect that a component has on your object. The following diagram illustrates this:



structure of a unity GameObject

In the following screenshot, we can see a **Game Object** with a **Light** component, as seen in the Unity interface:



Now let's look at how this approach would be used in a simple gameplay context.

The Unity way – an example

If we wished to have a bouncing ball as part of a game, then we would begin with a Sphere. This can quickly be created from the Unity menus, and will give you a new `GameObject` with a sphere mesh (the 3D shape itself) assigned to the Mesh Filter component. Unity will also automatically add a Renderer component to make it visible. Having created this, we would then add a Rigidbody component. A Rigidbody (Unity refers to most two-word phrases as a single-word term) is a component which tells Unity to apply its physics engine to an object. With this comes properties such as mass, gravity, and drag, and also the ability to apply forces to the object, either when the player commands it or simply when it collides with another object.

Our sphere will now fall to the ground when the game runs, but how do we make it bounce? This is simple! The Collider component has a variable called **Physic Material**, which is a setting for the physics engine, defining how it will react to other objects' surfaces. Here, we can select **Bouncy**, a ready-made Physic Material provided by Unity as part of an importable package, and voila! Our bouncing ball is complete in only a few clicks.

This streamlined approach for the most basic task, such as the previous example, seems pedestrian at first. However, you'll soon find that by applying this approach to more complex tasks, they become very simple to achieve. Here is an overview of some further key Unity concepts that you'll need to know as you get started.

Assets

These are the building blocks of all Unity projects. From textures in the form of image files, through 3D models for meshes, and sound files for effects, Unity refers to the files you'll use to create your game as assets. This is why, in any Unity project folder, all files used are stored in a child folder named `Assets`. This `Assets` folder is mirrored in the **Project** panel of the Unity interface; see the *The interface* section in this chapter for more detail.

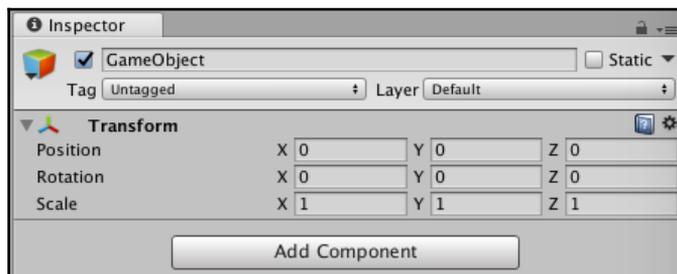
Scenes

In Unity, you should think of scenes as individual levels or areas of game content. However, some developers create entire games in a single scene, such as puzzle games, by dynamically loading content through the code. By constructing your game with many scenes, you'll be able to distribute loading times and test different parts of your game individually. New scenes are often used separately to a game scene you may be working on in order to prototype or test a piece of potential gameplay.

Any currently open scene is what you are working on. In Unity 2017, you can load more scenes into the hierarchy while editing, and even at runtime, through the new SceneManager API, where two or more scenes can be worked on simultaneously. Scenes can be manipulated and constructed by using the **Hierarchy** and **Scene** views.

GameObjects

Any active object in the currently open scene is called GameObject. Certain assets taken from the **Project** panel such as Models and Prefabs become assets when placed (or *instantiated*) into the current scene. Other objects such as particle systems and primitives can be placed into the scene by using the **Create** button on the **Hierarchy**, or by using the **GameObject** menu at the top of the interface. All game objects contain at least one component to begin with, that is, the **Transform** component. **Transform** simply tells the Unity engine the **Position**, **Rotation**, and **Scale** of an object, all described in **x, y, z** coordinates. In turn, the component can then be addressed in scripting in order to set an object's position, rotation, or scale. From this initial component, you will build upon game objects with further components, adding the required functionality to build every part of any game scenario you can imagine. In the following screenshot, you can see the most basic form of a GameObject, as shown in the **Inspector** panel:



GameObjects can also be nested in the **Hierarchy** in order to create the parent-child relationships we mentioned previously.

Components

Components come in various forms. They can be for creating behavior, defining appearance, and influencing other aspects of an object's function in the game. By attaching components to an object, you can immediately apply new parts of the game engine to your object. Common components of game production come built-in with Unity, such as the Rigidbody component mentioned earlier, down to simpler elements such as lights, cameras, particle emitters, and more. To build further interactive elements of the game, you'll write scripts, which are also treated as components in Unity. Try to think of a script as something that extends or modifies the existing functionality available in Unity, or creates behavior with the Unity scripting classes provided.

Scripts

While being considered by Unity to be components, scripts are an essential part of game production and deserve a mention as a key concept. In this book, we will write our scripts in **C Sharp** (more often written as **C#**). We have chosen to primarily focus on C# as this is the main language used by many Unity developers for many reasons, and soon the JavaScript syntax of **UnityScript** and **Boo** will be abandoned. Unity does not require you to learn how the coding of its own engine works or how to modify it, but you will be utilizing scripting in almost every game scenario you develop. The beauty of using Unity scripting is that any script you write for your game will be straightforward enough after a few examples, as Unity has its own built-in behavior class called `MonoBehaviour`, a set of scripting instructions for you to call upon. For many new developers, getting to grips with scripting can be a daunting prospect, and one that threatens to put off new Unity users who are more accustomed to design. If this is your first attempt at getting into game development, or you have no experience in writing code, do not worry. We will introduce scripting one step at a time, with a mind to showing you not only the importance, but also the power of effective scripting for your Unity games.

To write the code, Unity offers a Visual Studio 2017 community installation, which perfectly integrates with Unity and is definitively the best **Integrated Development Environment (IDE)** in which to write your code when using Unity on the Windows platform. On MacOS, Unity offers MonoDevelop. This separate application can be found in the Unity application folder on your PC/Mac and will be launched every time you edit a new script or an existing one. Amending and saving scripts in Visual Studio (PC) or MonoDevelop (macOS) will immediately update the script in Unity as soon as you switch back to Unity. You can also designate your favorite script editor in the Unity preferences if you wish to.

Prefabs

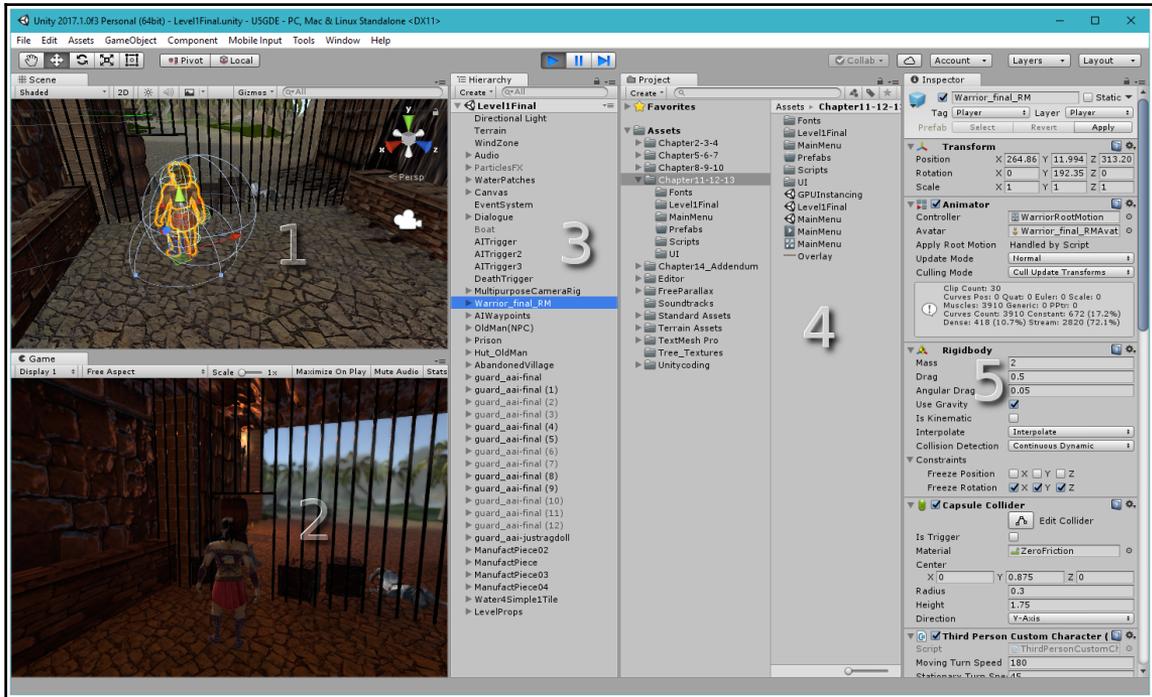
Unity's development approach hinges around the `GameObject` concept, but it also has a clever way of storing objects as assets to be reused in different parts of your game, and then instantiated (also known as *spawning* or *cloning*) at any time. By creating complex objects with various components and settings, you'll be effectively building a template for something you may want to spawn multiple instances of (hence *instantiate*), with each instance then being individually modifiable.

Consider a crate as an example. You may have given the object in the game a mass, and written scripted behaviors for its destruction. Chances are, you'll want to use this object more than once in a game, and perhaps even in games other than the one it was designed for.

Prefabs allow you to store a `GameObject`, complete with all its components attached and its children as well. Think of prefabs simply as empty containers that you can fill with objects to form a data template you'll likely recycle many times. The other great feature of prefabs is that you can have many of them in a single scene and even tweak the same value for all of them from the prefab in one single shot!

The interface

The Unity interface, like many other working environments, has a customizable layout. Consisting of several dockable spaces, you can pick which parts of the interface appear where. Let's take a look at a typical Unity layout:



This layout can be achieved by going to **Window | Layouts | 2 by 3 in Unity**.

As the preceding screenshot demonstrates (Mac version shown), there are five different panels or views you'll be dealing with, which are as follows:

- **Scene [1]:** Where the game scene is constructed
- **Game [2]:** The preview window, active only in Play Mode
- **Hierarchy [3]:** A list of game objects in the scene
- **Project [4]:** A list of your project's assets; acts as a library
- **Inspector [5]:** Settings for currently selected asset/object/setting

The Scene view and Hierarchy view

The **Scene** view is where you will build the entirety of your game project in Unity. This window offers a perspective (full 3D) view, which is switchable to orthographic (top-down, side-on, and front-on) views or can be put into 2D mode using the 2D toggle button, locking it to only showing an orthographic side view. When working in one of the orthographic views, rotating the view will display the scene isometrically. The **Scene** view acts as a fully rendered **Editor** view of the game world you build. Dragging an asset to this window (or the Hierarchy) will create an instance of it as a `GameObject` in the scene.

The **Scene** view is tied to the **Hierarchy**, which lists all game objects in the currently open scene in ascending alphabetical order.

Control tools

The **Scene** view window is also accompanied by six useful control tools, as shown in the following screenshot:



Accessible from the keyboard using keys *Q*, *W*, *E*, *R*, *T*, and *Y*, these keys perform the following operations:

- **The Hand tool** [*Q*]: This tool allows the navigation of the **Scene** window. It allows you to drag objects around in the **Scene** window with the left mouse button to pan your view. Holding down the *Alt* key with this tool selected will allow you to orbit your view around a central point you are looking at, and holding the *Ctrl* key will allow you to zoom, as will scrolling the mouse wheel. Holding the *Shift* key down will speed up both of these functions.
- **The Move tool** [*W*]: This is your active selection tool. As you can completely interact with the **Scene** window, selecting objects either in the **Hierarchy** or **Scene** means you'll be able to drag the object's axis handle in order to reposition it.
- **The Rotate tool** [*E*]: This works in the same way as Translate, using visual *handles* to allow you to rotate your object around each axis.
- **The Scale tool** [*R*]: This tool also works as the Translate and Rotate tools. It adjusts the size or scale of an object using visual handles.

- **The Rect Transform tool [T]:** The Rect Transform tool is used to change size and position of a 2D sprite or UI element.
- **The Transform tool [Y]:** This is a three in one tool. It combines the Move, Rotate and Scale tools. Its Gizmo provides handles for movement and rotation. When the Tool Handle Rotation is set to **Local**, the Transform tool also provides handles for scaling the selected GameObject.

Having selected objects in either the **Scene** or **Hierarchy**, they immediately get selected in both. The selection of objects in this way will also show the properties of the object in the **Inspector**.

For detailed info about the control tools for positioning GameObjects, head to: <https://docs.unity3d.com/Manual/PositioningGameObjects.html>.

Scene navigation

Given that you may not be able to see an object you've selected in the **Hierarchy** in the **Scene** window, Unity also provides the use of the *F* key to focus your **Scene** view on that object. Simply select an object from the **Hierarchy**, hover your mouse cursor over the **Scene** window, and press *F*. Alternatively you can double click on the GameObject in the **Hierarchy** to obtain the same.

To move around your **Scene** view using the mouse and keys, you can use Flythrough mode. Simply hold down the right mouse button and drag to look around in first person. You can use *W*, *A*, *S*, and *D* to move, and *Q* and *E* to descend and ascend respectively.

Alternatively, if you need to focus the view on a specific object, you can select it, and then press *F*, just like on Maya, to focus the view on this object. With a combination of *Ctrl* + *Alt* keys (*command* + *alt* on the Mac) and *F* you can align the selected object to the view, or, vice versa, align the view with the selected object.

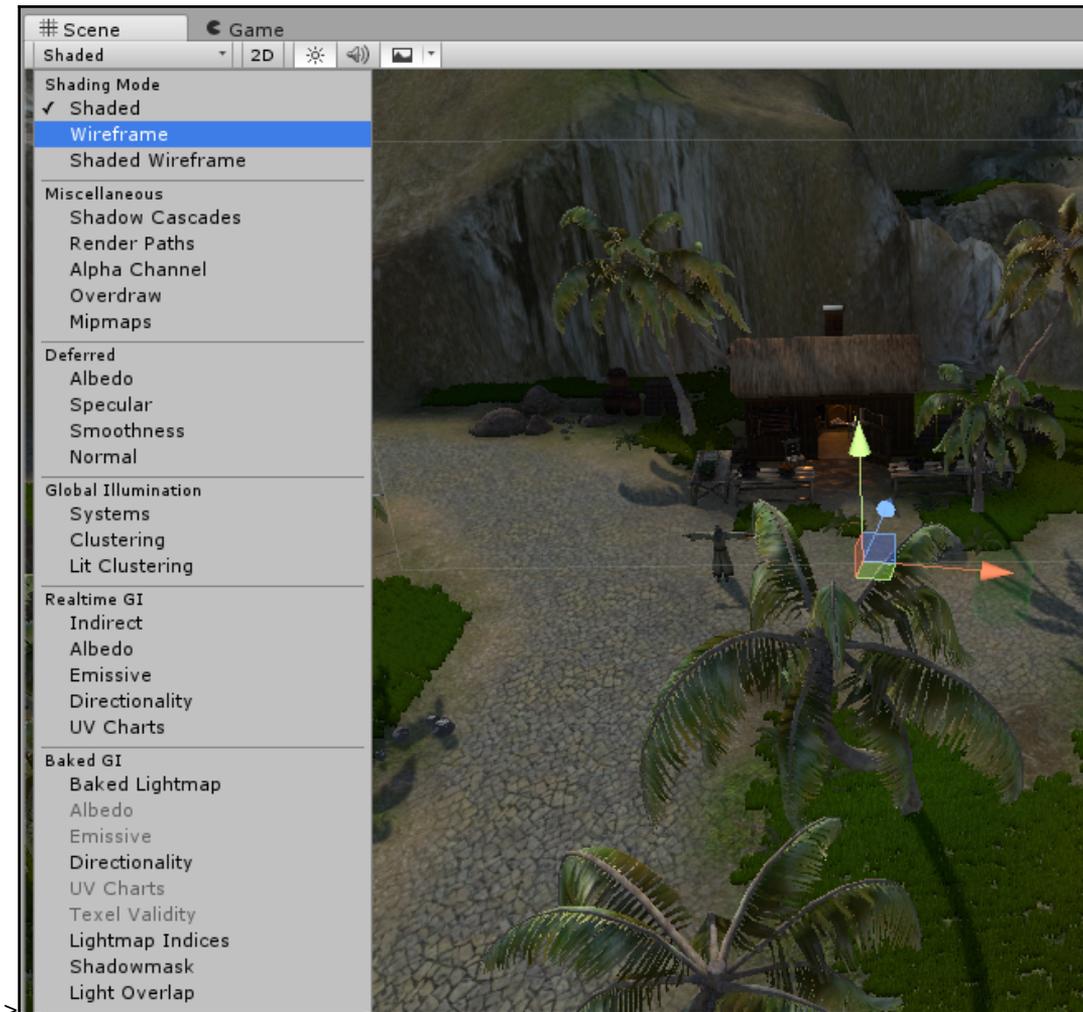
Control bar

In addition to the control tools, there is also a scene control bar with additional options in the **Scene** view:

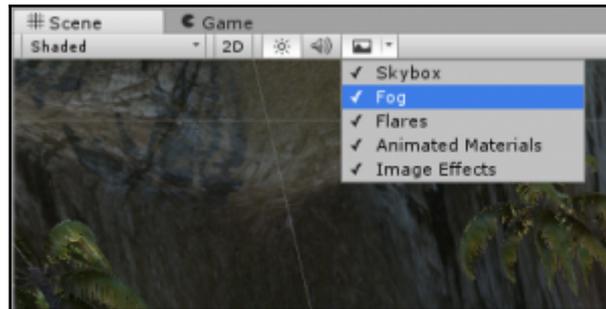


The **Scene** view's control bar allows you to adjust the following:

- **Shading Mode:** The default is set to **Shaded**, which means that the scene will be rendered in full shaded and textured mode:



- **2D/3D View toggle:** This toggles the orthogonal 2D camera for the scene view for 2D game scene editing.
- **Scene Lighting toggle:** This can be switched ON/OFF by clicking on it.
- **Audio toggle:** toggle audio sources preview on and off.
- **Skybox, fog and other FX toggle:** enable the preview of skybox, fog, and other effects in the **Scene** view:



- **Gizmos:** Use this pop-out menu to show or hide gizmos, the 2D icons of cameras, lights, and other components shown in the **Scene** view to help you instantly visualize the different type of objects.
- **Search box:** While the **Scene** view is intrinsically linked with the **Hierarchy**, often you may need to locate an item or type of item in the **Scene** view itself by searching. Simply type the name or data type (in other words, an attached component) of an object into the search, and the **Scene** view will *gray* out other objects in order to highlight the item you have searched for. This becomes very useful when dealing with more complex scenes, and should be used in conjunction with *F* on the keyboard to focus on the highlighted object in the **Scene** window itself.

Create button

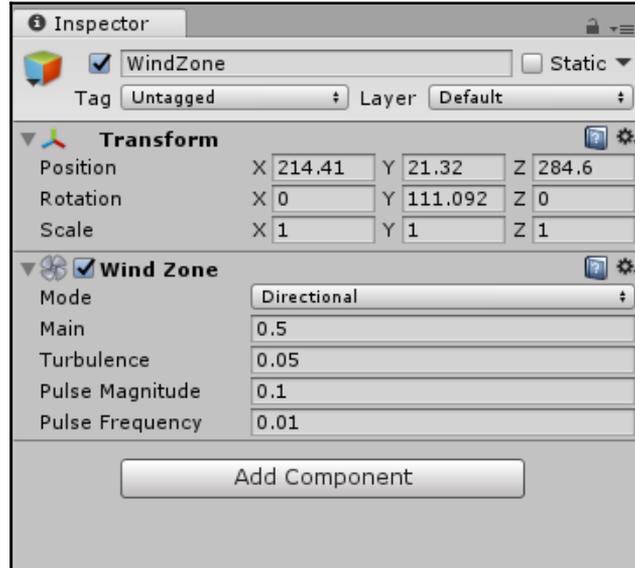
As many of the game assets you'll use in Unity will be created by the editor itself, the **Hierarchy** has a **Create** button that allows you to create objects that are also located within the top **GameObject** menu. Similar to the **Create** button on the **Project** panel, this drop-down menu creates items and immediately selects them so that you may rename or begin working with them in the **Scene** or **Inspector**.

The Inspector

Think of the **Inspector** as your personal toolkit to adjust every element of any `GameObject` or asset in your project. Much like the Property Inspector concept utilized by Blender or Maya, this is a context-sensitive window. All this means is that whatever you select in the scene or in the **Project** view, the **Inspector** will change to show its relevant properties, which makes it sensitive to the context in which you are working.

The **Inspector** will show every component part of anything you select and will allow you to adjust the variables of these components using simple form elements such as text input boxes, slider scales, buttons, and drop-down menus. Many of these variables are tied into Unity's drag and drop system, which means that, rather than selecting from a drop-down menu, you can drag and drop to choose settings or assign properties if it is more convenient.

This window is not only for inspecting objects. It will also change to show the various options for your project when choosing them from the **Edit** menu, as it acts as an ideal space to show your preferences, such as changing back to showing component properties as soon as you re-select an object or asset. The following screenshot shows the **Inspector** window:



In this screenshot, the **Inspector** is showing properties for a **target** object in the game. The object itself features two components, namely **Transform** and **Wind Zone**. The **Inspector** will allow you to make changes to settings in either of them. Also note that in order to temporarily disable any component at any time, which will become very useful for testing and experimentation, you can simply deselect the checkbox to the left of the component's name. Likewise, if you wish to switch off an entire object at a time, then you may deselect the checkbox next to its name at the top of the **Inspector** window.

The Project window

The Project window is a direct view of the `Assets` folder of your project. Every Unity project is made up of a parent folder containing three subfolders—`Assets`, `Library`, and, while the Unity Editor is running, a `Temp` folder. Placing assets into the `Assets` folder means you'll immediately be able to see them in the **Project** window, and they'll also be automatically imported into your Unity project. Likewise, changing any asset located in the `Assets` folder, and resaving it from a third-party application, such as Photoshop, will cause Unity to reimport the asset, reflecting your changes immediately in your project and any active scenes that use that particular asset.

Asset management



It is important to remember that you should only alter asset locations and names using the **Project** window; using Finder (Mac) or Windows Explorer (Windows) to do so may break connections in your Unity project. Therefore, to relocate or rename objects in your `Assets` folder, use Unity's **Project** window instead of your operating system.

The **Project** window, like **Hierarchy**, is accompanied by a **Create** button. This allows the creation of any assets that can be made within Unity, for example, scripts, prefabs, and materials.

The Game view

The Game view is invoked by pressing the Play button, and acts as a realistic test of your game. It also has settings for the screen ratio, which will come in handy when testing how much of the player's view will be restricted in certain ratios, such as 4:3 (as opposed to wide) screen resolutions. Having pressed Play, it is crucial that you bear in mind the following advice:

Play Mode - testing only!

In Play Mode, the adjustments you make to any parts of your game scene are merely temporary. It is meant as a testing mode only, and when you press **Play** again to stop the game, all changes made to active game objects during Play Mode will be undone. This can often trip up new users, so don't forget about it! Unity can help with colors. You can set a color overlay for the Unity interface itself for the Play Mode. In this way, when you are in Play Mode, the Unity interface will be colored with that color. Go into **Edit | Preferences | Colors** to change it to something other than full white (no color change).



This also works in the black Unity Plus Unity Pro interface. Another cool thing you can do while in Play Mode is copy a whole GameObject, a group of game objects, or a component of a GameObject while in play and paste it after you stop the execution. This is very handy while editing your game.

The **Game** view can also be set to **Maximize** when you invoke Play Mode, giving you a better view of the game at fullscreen, in other words, the window expands to fill the interface. It is worth noting that you can expand any part of the interface in this way, simply by hovering over the part you wish to expand and pressing the *Spacebar*.

In addition to using **Play** to preview your game, the live game mode can also be paused by pressing the **Pause** button at the top of the interface, and play can be advanced a frame at a time using the third **Advance Frame** button next to **Pause**. This is useful when debugging, which is the process of finding and solving problems or *bugs* within your game development.

Summary

Here, we have looked at the key concepts you will need to understand in order to complete the exercises in this book. However, 3D is a detailed discipline that you will continue to learn not only with Unity, but in other areas as well. With this in mind, you are recommended to continue reading on the topics discussed in this chapter in order to supplement your study of 3D development. Each individual piece of software you encounter will have its own dedicated tutorials and resources for learning it. If you wish to learn 3D artwork to complement your work in Unity, you are recommended to familiarize yourself with your chosen package, after researching the list of tools that work with the Unity pipeline and choosing which one suits you best.

Now that we've taken a brief look at 3D concepts and the processes used by Unity to create games, we'll begin by completing a simple exercise before getting started on the larger game element of this book.

In the following chapter, we'll begin with a short exercise in which you will prototype a simple game mechanic using primitive shapes and some basic coding to get you started in C#. It is important to kick-start your Unity learning with a simple example using primitives, as you will often find yourself prototyping game ideas in this way once you feel more comfortable using Unity.

Let's get started!

2

Prototyping and Scripting Basics

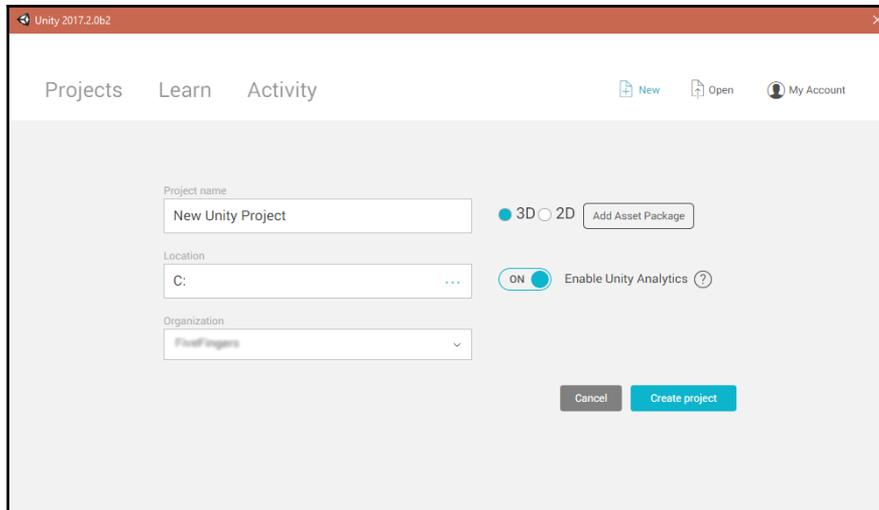
When starting out in game development, one of the best ways to learn the various parts of the discipline is to prototype your idea. Unity excels in assisting you with this, with its visual scene editor and public member variables that form settings in the **Inspector**. To get to grips with working in the Unity Editor, we'll begin by prototyping a simple game mechanic using primitive shapes and basic coding.

In this chapter, you will learn about the following:

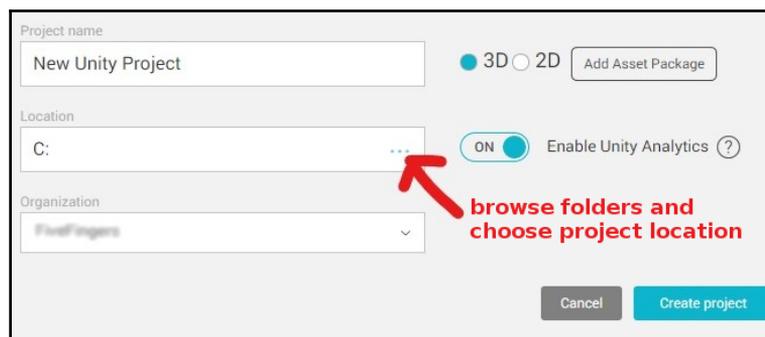
- Creating a New Project in Unity
- Importing asset packages
- Working with GameObjects in the **Scene** view and **Hierarchy**
- Adding materials
- Writing C# code
- Variables, functions, and commands
- Using the `Translate()` command to move objects
- Using prefabs to store objects
- Using the `Instantiate()` command to spawn objects

Your first Unity project

When you're launching Unity, you'll be presented with a launcher. Take a look at the following steps for setting up your first Unity project. In the Unity launcher that appears when you first run Unity, select **New**, to the right of the window, and then **New Project**, and you will be presented with the **Projects** tab:



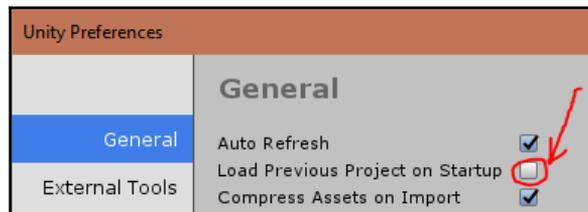
From here, select the **New** (project) tab and then the **3D** type of project. Click the **...** (browse) button and choose where you would like to save your new Unity project folder on your hard drive. The new project has been named **UGDE** after this book, and we have chosen to store it on the desktop for easy access:



When you are happy with your selection, simply choose **Create project** at the bottom of this dialog window. Unity will then create your new project and you will see progress bar progressing.



You can change the default behavior (showing the launcher) and load the previously open project by changing the settings in the **Unity Preferences**, as shown in the following screenshot:



Once the Editor opens, from the top menu, choose: **Assets | Import packages** and from the list of packages to be imported, import one by one the following packages:

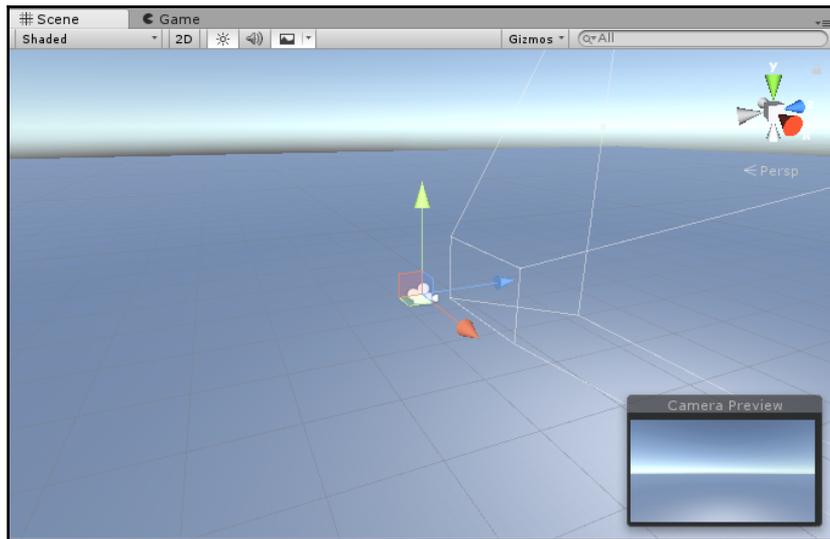
- Characters
- Cameras
- Effects
- Terrain Assets
- Environment

We are ready to work, let's see how to setup a simple prototype environment!

A basic prototyping environment

To create a simple environment to prototype some game mechanics, we'll begin with a basic series of objects with which we can introduce gameplay that allows the player to aim and shoot at a wall of primitive cubes.

When complete, your prototyping environment will feature a floor comprised of a cube primitive, a Main Camera through which we view the 3D world, and a point light set up to highlight the area where our gameplay will be introduced. It will look something like the following screenshot:

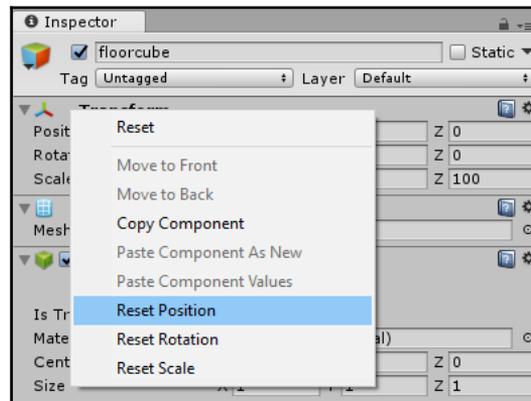


Setting the scene

As all new scenes come with a Main Camera object by default, we'll begin by adding a floor for our prototyping environment.

On the **Hierarchy** panel, click the **Create** button, and from the drop-down menu, choose **Cube**. The items listed in this drop-down menu can also be found in the **GameObject | Create Other** top menu. You will now see an object in the **Hierarchy** panel called **Cube**. Select this and press *return* (Mac) or *F2* (Windows) or double-click the object name slowly to rename this object (on both platforms). Type in `floorcube` and press *return* to confirm this change.

For the sake of consistency, we will begin our creation at world zero, the center of the 3D environment we are working in. To ensure that the floor cube you just added is at this position, ensure it is still selected in the **Hierarchy** panel and then check the **Transform** component on the **Inspector** panel, ensuring that the position values for **X**, **Y**, and **Z** are all at 0. If not, change them all to zero either by typing them in or by clicking the cog icon to the right of the component and selecting **Reset Position** from the pop-out menu:



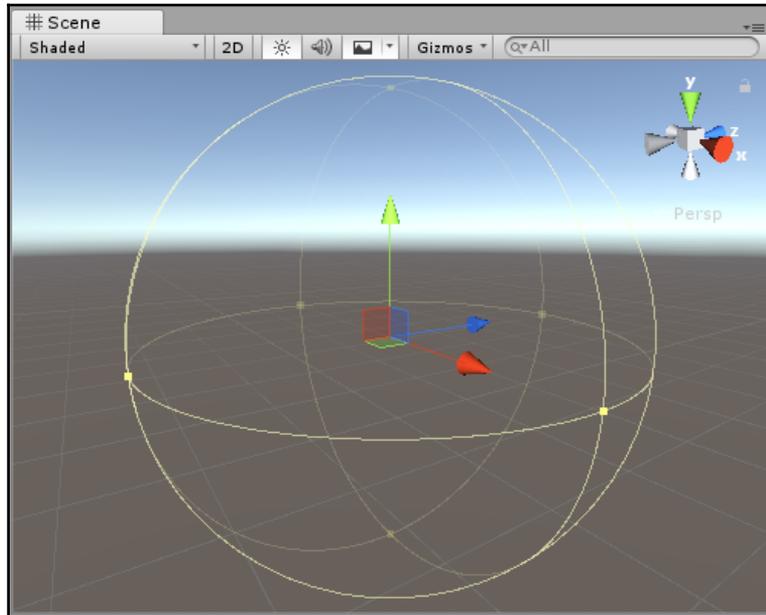
Next, we'll turn the cube into a floor by stretching it out in the *x* and *z* axes. Into the **X** and **Z** values under **Scale** in the **Transform** component, type a value of 100, leaving **Y** at a value of 1.

Adding simple lighting

Now we will highlight part of our prototyping floor by adding a point light. Select the **Create** button on the **Hierarchy** panel (or go to **GameObject** | **Create Other**) and choose point light. Lighting is crucial in 3D scenes as it allows the surrounding 3D world to be illuminated by all the lights present in the scene according to their type, range, intensity, and color. While the directional light has infinite range but changes lighting direction with its orientation (rotation), the point light has no rotation and gives light in all directions for a given range. The third type of light is the spotlight, which has both direction and range, and the size of the cone angle can be set as well.

Position the new point light at $(0, 20, 0)$ using the **Position** values in the **Transform** component, so that it is 20 units above the floor.

You will notice that this means that the floor is out of range of the light, so expand the range by dragging on the *yellow* dot handles that intersect the outline of the point light in the **Scene** view, until the value for range shown in the **Light** component in the **Inspector** reaches something around a value of 40, and the light is illuminating part of the floor object:



Bear in mind that most components and visual editing tools in the **Scene** view are inextricably linked, so altering values such as **Range** in the **Inspector**, Light component will update the visual display in the **Scene** view as you type, and stay constant as soon as you press *return* to confirm the values entered.

Another brick in the wall

Now let's make a wall of cubes that we can launch a projectile at. We'll do this by creating a single master brick, adding components as necessary, and then duplicating this until our wall is complete.

Building the master brick

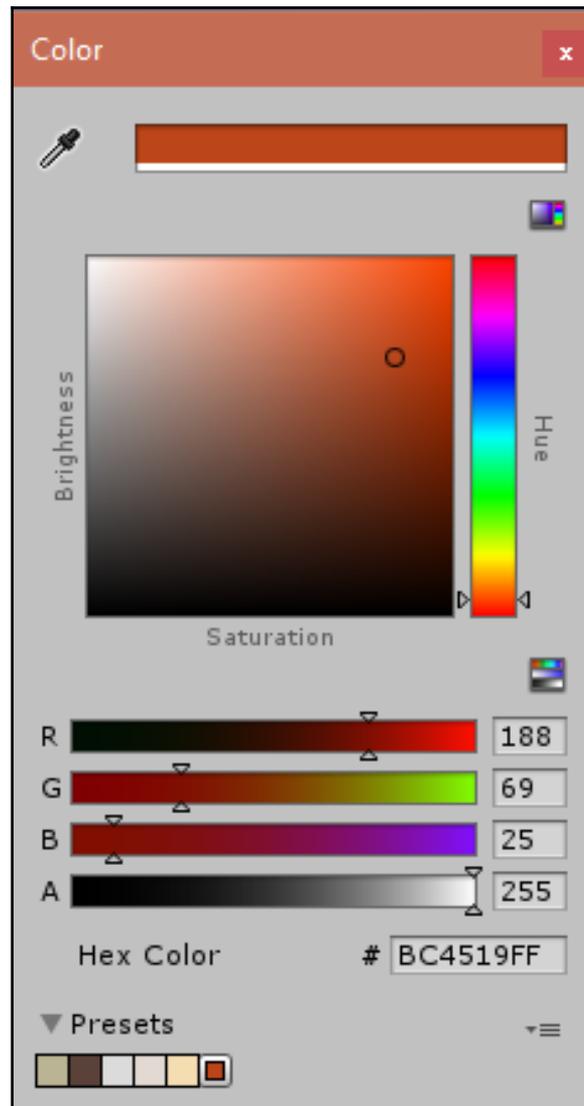
In order to create a template for all of our bricks, we'll start by creating a master object, something to create clones of. This is done as follows:

1. Click the **Create** button at the top of the **Hierarchy** and select **Cube**. Position this at $(0, 1, 0)$ using the **Position** values in the **Transform** component on the **Inspector**. Then, focus your view on this object by ensuring it is still selected in the **Hierarchy**, by hovering your cursor over the **Scene** view and pressing *F*.
2. Add physics to your **Cube** object by choosing **Component** | **Physics** | **Rigidbody** from the top menu. This means that your object is now a Rigidbody, it has mass and gravity, and is affected by other objects using the physics engine for realistic reactions in the 3D world.
3. Finally, we'll set a tint for this object by creating a material. Materials are a way of applying color and imagery to our 3D geometry. To make a new one, go to the **Create** button on the **Project** view and choose **Material** from the drop-down menu. Press *Return* (Mac) or *F2* (Windows) to rename this asset `Red` instead of the default name **New Material**.

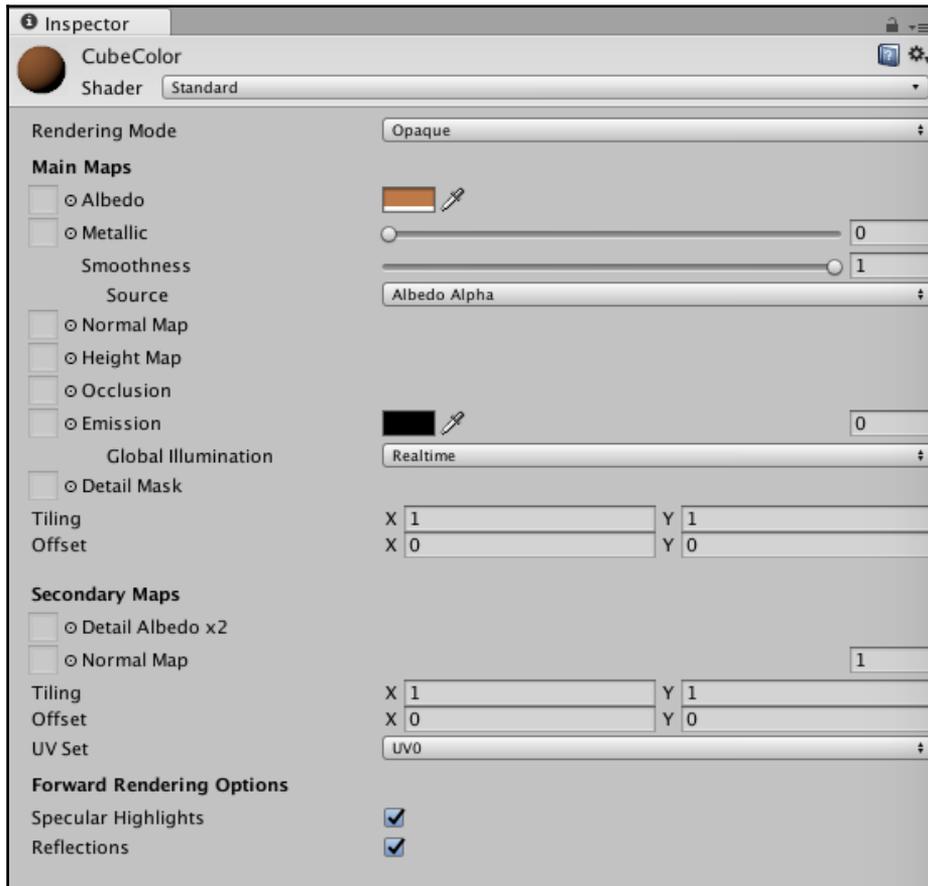


You can also right-click in the `Materials` folder in the **Project** view and select **Create** | **Material**, or alternatively, you can use the editor main menu: **Assets** | **Create** | **Material**.

4. With this material selected, the **Inspector** shows its properties. Click on the color block to the right of **Albedo** to open the **Color Picker**. This will differ in appearance depending upon whether you are using Mac or Windows. On Windows 10, it will look like something like this:



5. Choose a shade of orange/red and then close the window.
The **Albedo** block should now have been updated with the color we have selected, as in the following screenshot:



6. To apply this material, drag it from the **Project** view and drop it onto either the cube as seen in the **Scene** view, or onto the name of the object in the **Hierarchy**. The material is then applied to the Mesh Renderer component of this object and immediately seen following the other components of the object in the **Inspector**. Most importantly, your cube should now be red! Adjusting settings using the preview of this material on any object will edit the original asset, as this preview is simply a link to the asset itself, not a newly editable instance.

7. Now that our cube has a color and physics applied through the Rigidbody component, it is ready to be duplicated and act as one brick in a wall of many. However, before we do that, let's have a quick look at the physics in action. With the cube still selected, set the **Y** position value to 1.5 and the **X** rotation value to 40 in the **Transform** component in the **Inspector**. Press **Play** at the top of the Unity interface and you should see the cube fall and then settle, having fallen at an angle.



The shortcut for Play is *Ctrl + P* for Windows and *command + P* for Mac.

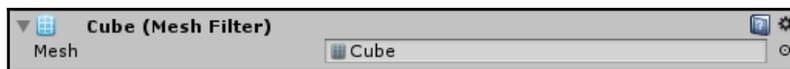
8. Press Play again to stop testing. Do not press Pause as this will only temporarily halt the test, and changes made thereafter to the scene will not be saved.
9. Set the **Y** position value for the cube back to 1, and set the **X** Rotation back to 0.

This object's only purpose is to represent our cube visually and contains only two key components that make it visible:

- **Mesh Filter**
- **Mesh Renderer**

But what do these two components do?

A **Mesh Filter** is simply a component containing the mesh—the 3D shape itself. It then works with the renderer to draw a surface based on the mesh. It is named **Cube** in this instance. The name of the **Mesh Filter** component in any 3D mesh is usually the name of the mesh asset that it represents. Therefore, when you are introducing externally created models, you will notice that each **Mesh Filter** component is named after each part of the model:



A **Mesh Renderer** component must be present in order to draw surfaces onto the mesh of a 3D object. It is also in charge of the following:

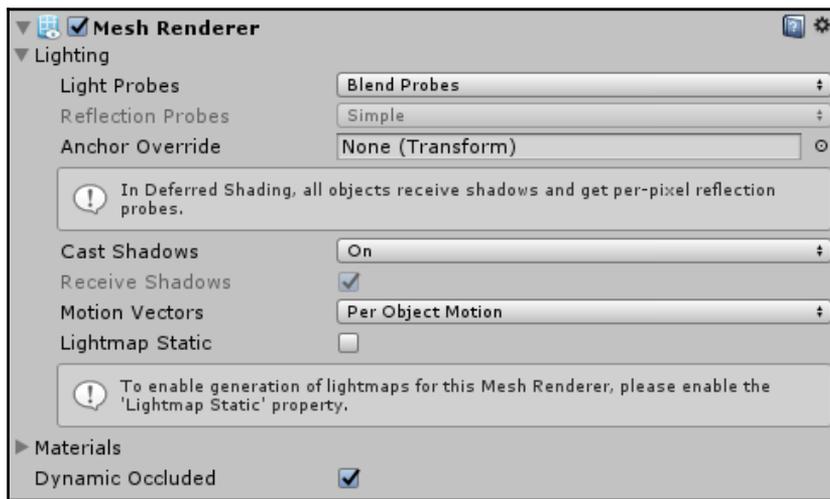
- How the mesh responds to lighting
- Materials used on the surface to show color or textures
- Light Probes and Reflection Probes

Mesh renderers, hence, have the following parameters:

- **Cast Shadows:** Whether light cast onto this object will cause a shadow to be cast on the other surfaces.
- **Light Probes:** Special transforms that provide a way to capture and use information about light that is passing through the empty space in your scene. For more info head to: <https://docs.unity3d.com/Manual/LightProbes.html>.
- **Reflection Probes:** Improvement on basic reflection mapping. They allow the visual environment to be sampled at strategic points in the scene. For more info: <https://docs.unity3d.com/Manual/ReflectionProbes.html>.
- **Anchor Override:** A Transform used to determine the interpolation position when the Light Probe or Reflection Probe systems are used.
- **Cast Shadows:** These options specify how the object should cast shadows on other objects, and it has four possible values:
 1. **On:** The Mesh will cast a shadow when a shadow-casting Light shines on it.
 2. **Off:** The Mesh will not cast shadows.
 3. **Two Sided:** Shadows are cast from either side of the Mesh.
 4. **Shadows only:** Shadows from the Mesh will be visible, but not the Mesh itself.
- **Receive Shadows:** Whether shadows cast by other objects are drawn onto this object.
- **Motion Vectors:** If enabled, the line has motion vectors rendered into the Camera motion vector Texture. See `Renderer.motionVectorGenerationMode` in the Scripting API reference documentation to learn more.
- **Lightmap Static:** Tick this checkbox to indicate that this object will be lightmapped even if it is not marked as static. When you enable this, additional info are shown about Lightmapping and Realtime illumination.

- **Materials:** This section uses the **Size/Element** system seen in the **Tags & Layers settings** earlier in this chapter. It allows you to specify one or more materials and adjust settings for them directly without having to find out the material in use and then adjust it separately in the **Project** view.
- **Dynamic Occluded:** Tick this checkbox to indicate to Unity that occlusion culling should be performed for this object even if it is not marked as static.

The material in the preview is the **CubeColor** we made earlier so you are shown a Standard Shader in a simple basic fashion with a color set and nothing more, that will receive lights with a plain white color. The **Mesh Renderer** shadows default parameters are set to receive and cast shadows:



For additional info about the Mesh Renderer component head to: <https://docs.unity3d.com/Manual/class-MeshRenderer.html>.

Now that we know our brick behaves correctly, let's start creating a row of bricks to form our wall.

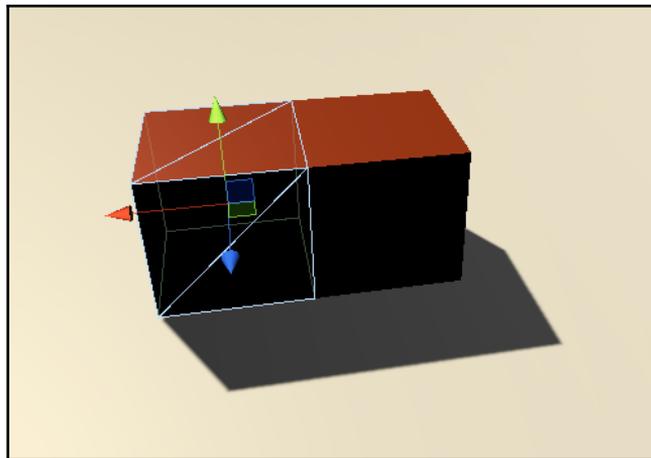
Creating our first prefab

Saving a prefab out of our brick cube is very simple: drag the GameObject from the **Hierarchy** view to the **Project** view, and voila!

You will see a new prefab object (a blue 3D box icon) appear in the folder you dragged to, named in the same way as the GameObject is in the **Hierarchy**. Notice that the object in the **Hierarchy** has turned blue. You have just created a prefab, so that blue means that you also created a prefab connection between the objects. If you change some parameter on the saved Prefab, all the cloned objects will change as well. To clone the prefab after you have saved it, you can either drag a new instance of the prefab from the **Project** view to the **Hierarchy** view or to the **Scene** view, or just duplicate the prefab clone in the scene.

And snap! It's a row

To help you position objects, Unity allows you to snap to specific increments when dragging. These increments can be redefined by going to **Edit | Snap Settings**. To use snapping, hold down *command* (Mac) or *Ctrl* (Windows) when using the Translate tool (*W*) to move objects in the **Scene** view. So, in order to start building the wall, duplicate the cube brick we already have using the shortcut *command + D* (Mac) or *Ctrl + D* (Windows), then drag the red axis handle while holding the snapping key. This will snap one unit at a time by default, so snap-move your cube one unit in the *x* axis so that it sits next to the original cube, shown as follows:



Repeat this procedure of duplication and snap-dragging until you have a row of 10 cubes in a line. This is the first row of bricks, and to simplify building the rest of the bricks, we will now group this row under an empty object and then duplicate the parent empty object.



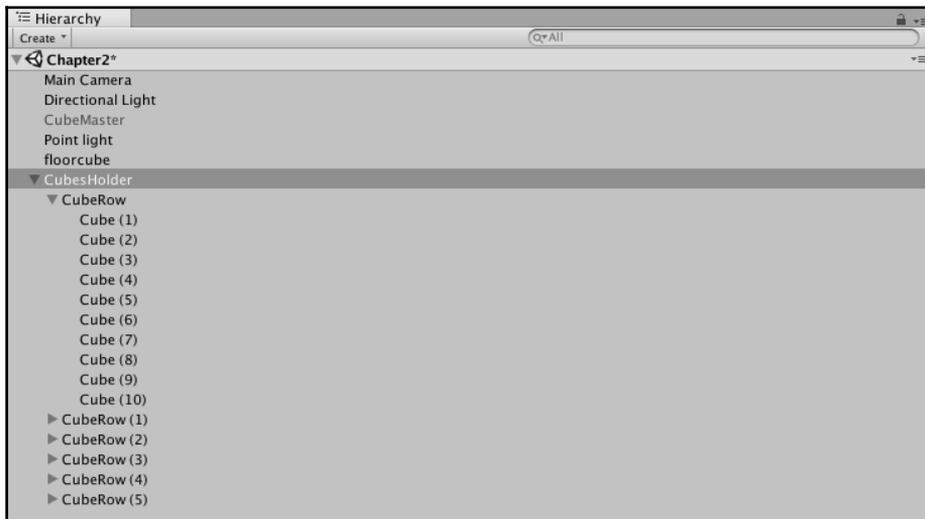
Vertex snapping

The basic snapping technique used here works well as our cubes are a generic scale of 1, but when scaling more detailed shaped objects, you should use vertex snapping instead. To do this, ensure that the **Translate** tool is selected and hold down **V** on the keyboard. Now hover your cursor over a vertex point on your selected object and drag to any other vertex of another object to snap to it.

Grouping and duplicating with empty objects

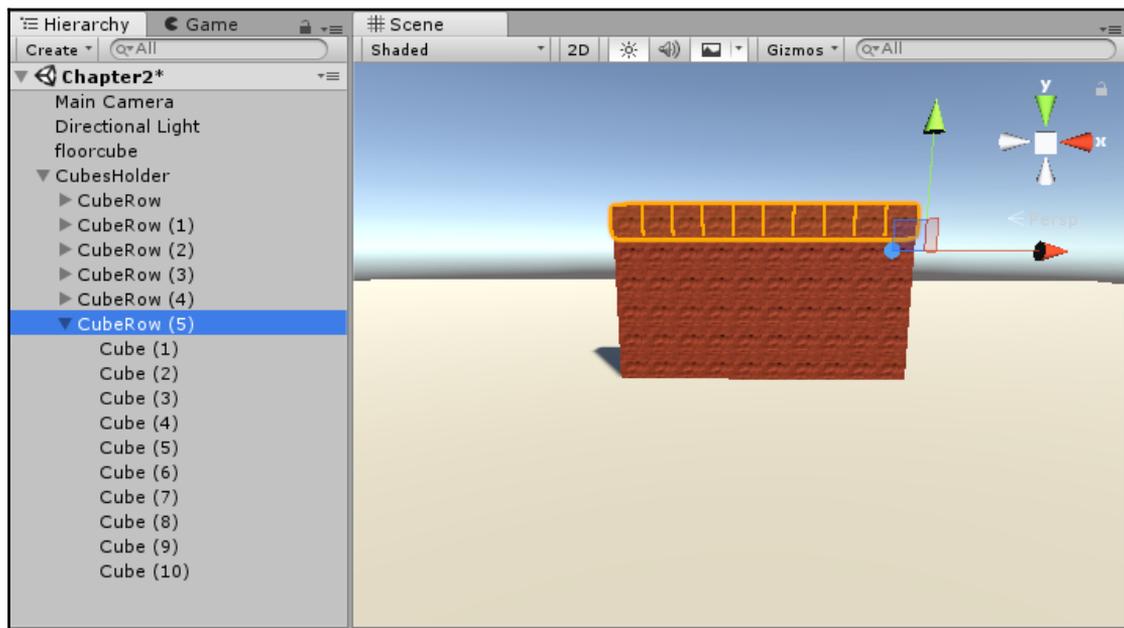
Create an empty object by choosing **GameObject | Create Empty** from the top menu, then position this at (4.5, 0.5, -1) using the Transform component in the **Inspector**. Rename this from the default name `GameObject` to `CubeHolder`.

Now select all of the cube objects in the **Hierarchy** by selecting the top one, holding the *Shift* key, and then selecting the last. Now drag this list of cubes in the **Hierarchy** onto the empty object named `CubeHolder` in the **Hierarchy** in order to make this their parent object. The **Hierarchy** should now look like this:



You'll notice that the parent empty object now has an arrow to the left of its object title, meaning you can expand and collapse it. To save space in the **Hierarchy**, click the arrow to hide all of the child objects, and then reselect the `CubeHolder`.

Now that we have a complete row made and parented, we can simply duplicate the parent object and use snap-dragging to lift a whole new row up in the y axis. Use the duplicate shortcut *Command* (Mac) or *Ctrl + D* (Windows) as before, then select the Translate tool (*W*) and use the snap-drag technique (hold *command* on Mac, *Ctrl* on Windows) outlined earlier to lift it by 1 unit in the Y axis by pulling the green axis handle. Repeat this procedure to create eight rows of bricks in all, one on top of the other. It should look something like the following screenshot:



Build it up, knock it down!

Now that we have built a wall, let's make a simple game mechanic where the player can maneuver the camera and shoot projectiles at the wall to knock it down. Set up the camera facing the wall by selecting the Main Camera object in the **Hierarchy** and positioning it at $(4,3,-15)$ in the Transform component. Also ensure that it has no rotation values; they should all be set to 0.

Introducing C# scripting

To take your first steps in programming, we will look at a simple example of the same functionality in C#, the main programming language used by Unity developers. To begin, click the **Create** button in the **Project** view, then simply press the **Add Component** button on the Main Camera GameObject in the **Inspector**.

Your new script will be placed into the **Project** view named `NewbehaviourScript`, and you will see an icon of a document with C# stamped on it. When selecting your new script, Unity offers a preview of what is in the script already in the view of the **Inspector**, as well as an accompanying **Edit** button that, when clicked, will launch the script into the default script editor. You can also launch a script in your script editor at any time by double-clicking on its icon in the **Project** view.

A new behavior script or class

It is recommended that you read through both parts of the ensuing section of this book as it contains information about scripting.

New scripts can be thought of as a new class in Unity terms. If you are new to programming, think of a class as a set of actions, properties, and other stored information that can be accessed under the heading of its name. For example, a class called `Dog` may contain properties such as `color`, `breed`, `size`, or `gender` and have actions such as `rollover` or `fetchStick`. These properties can be described as variables, while the actions can be written in functions, also known as methods. In this example, to refer to the `breed` variable, a property of the `Dog` class, we might refer to the class it is in (`Dog`) and use a period (full stop) to refer to this variable in the following way:

```
Dog.breed;
```

If we want to call a function within the `Dog` class, we might say, for example:

```
Dog.fetchStick();
```

We can also add arguments into functions—but not the everyday arguments we have with one another! Think of them as more like modifying the behavior of a function; for example, with our `fetchStick` function, we might build in an argument that defines how quickly our dog will fetch the stick. This might be called as follows:

```
Dog.fetchStick(25);
```

While these are abstract examples, it can often help to transpose coding into commonplace examples in order to make sense of them. As we continue in this book, think back to this example or come up with some examples of your own to help train yourself to understand classes of information and their properties.

When you write a script in C# you are writing a new class or classes with their own properties (variables) and instructions (functions) that you can call into play at the desired moment in your games.

What's inside a new C# MonoBehaviour class

When you begin with a new C# script, Unity gives you the following code to get started:

```
using UnityEngine;

using System.Collections;

public class NewbehaviourScript : MonoBehaviour {
    // Use this for initialization
    void Start () {
        }

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

}
```

This begins with the necessary two calls to the `UnityEngine` itself:

```
using UnityEngine;
using System.Collections;
```

It goes on to establish the class named after the script. With C#, you'll be required to name your scripts with matching names to the class declared inside of the script itself. This is why you will see: `public class NewbehaviourScript : MonoBehaviour {` at the start of a new C# document, as `NewbehaviourScript` is the default name that Unity gives to newly generated scripts. If you rename your script in the **Project** view when it is created, Unity will rewrite the class name in your C# script.

Code in classes



When writing code, most of your functions, variables, and other scripting elements will be placed within the class of a script in C#. *Within*, in this context, means that it must occur after the class declaration, and following the corresponding closing `}` of that, at the bottom of the script. So, unless told otherwise, while following the instructions in this book, assume that your code should be placed within the class established in the script.

Base Monobehaviour methods

Unity as an engine has many of its own functions that can be used to call different features of the game engine, and it includes three important ones when you create a new script in C#.



Functions (also known as methods) most often start with the term `void` in C#. This is the function's return type, which is the kind of data a function may result in. As most functions are simply there to carry out instructions rather than return information, often you will see `void` at the beginning of their declaration, which simply means that no data will be returned.

Some basic functions are explained as follows:

- `Awake()`: This is called when the scene first launches, *before* the `Start` function. It is thrown when the `GameObject` with this component *awakens*, meaning that is in the scene and is `Active`. If the `GameObject` is inactive, when you activate it manually or by code while running the game, the `OnEnabled` event will be fired and the method will be called.

The official manual states the following:

Awake() is used to initialize any variables or game state before the game starts. Awake() is called only once during the lifetime of the script instance. Awake() is called after all objects are initialized so you can safely speak to other objects or query them using example, `GameObject.FindWithTag`. Each `GameObject`'s `Awake()` is called in a random order between objects. Because of this, you should use `Awake()` to set up references between scripts, and use `Start` to pass any information back and forth. `Awake()` is always called before any `Start()` functions. This allows you to order initialization of scripts. `Awake()` differently from `Start()` can not act as a co-routine.



Note: use `Awake()` instead of the constructor for initialization, as the serialized state of the component is undefined at construction time. `Awake()` is called once, just like the constructor.

- `Start()`: This is called when the scene first launches, so is often used as it is suggested in the code for initialization. For example, you may have a core variable that must be set to 0 when the game scene begins, or perhaps a function that spawns your player character in the correct place at the start of a level. This method can be run as a co-routine.
- `Update()`: This is called in every frame that the game runs, and is crucial for checking the state of various parts of your game during this time, as many different conditions of `GameObjects` may change while the game is running.

For more information about these and other basic methods, head to: <https://docs.unity3d.com/ScriptReference/Monobehaviour.html>.

Variables in C#

To store information in a variable in C#, you will use the following syntax:

```
typeOfData nameOfVariable = value;
```

Here is an example:

```
int currentScore = 5;
```

Another example would be as follows:

```
float currentVelocity = 5.86f;
```



Note that the examples here show numerical data, with `int` meaning integer—a whole number, and `float` meaning floating point—a number with a decimal place, which in C# requires a letter `f` to be placed at the end of the value.

Comments

You can write comments using the following:

```
// two forward slashes symbols for a single line comment
```

Another way of doing this would be as follows:

```
/* forward-slash, star to open a multi line comment and at the end of  
it,star, forward-slash to close */
```



You can write comments in the code to help you remember what each part does as you progress through the book. Remember that, because comments are not executed as code, you can write whatever you like, including pieces of code. As long as they are contained within a comment, they will never be treated as working code.

Write the Shooter class

Now let's put some of your new scripting knowledge into action and turn our existing scene into an interactive gameplay prototype. In the **Project** view in Unity, rename your newly created script `Shooter` by selecting it, pressing *return* (Mac) or *F2* (Windows), and typing in the new name.

If you are using C#, remember to ensure that your class declaration inside the script matches this name of the script:

```
public class Shooter : MonoBehaviour {
```

To kick-start your knowledge of using scripting in Unity, we will write a script to control the camera and allow the shooting of a projectile at the wall we have built. To begin with, we will establish three variables:

- `bullet`: This is a variable of the type `Rigidbody`, as it will hold a reference to a physics-controlled object we will make
- `power`: This is a floating point variable number we will use to set the power of shooting
- `moveSpeed`: This is another floating point variable number we will use to define the speed of movement of the camera using the arrow keys

These variables must be public member variables in order for them to display as adjustable settings in the **Inspector**. You'll see this in action very shortly!

Declaring public variables

Public variables are important to understand as they allow you to create variables that will be accessible from other scripts, which is an important part of game development as it allows for simpler inter-object communication. Public variables are also really useful as they appear as settings that you can adjust visually in the **Inspector** once your script is attached to an object. Private variables are the opposite—they are designed to be only accessible within the scope of the script, class, or function they are defined within, and do not appear as settings in the **Inspector**.

Before we begin, as we will not be using it, remove the `Start()` function from this script by deleting `void Start() {}`. To establish the required variables, put the following code snippet into your script after the opening of the class:

```
using UnityEngine;

using System.Collections;

public class Shooter : MonoBehaviour {

    public Rigidbody bullet;
    public float power = 1500f;
    public float moveSpeed = 2f;

    void Update () {

    }

}
```



Note that, in this example, the default explanatory comments and the `Start()` function have been removed in order to save space and make the code easier to read.

Assigning scripts to objects

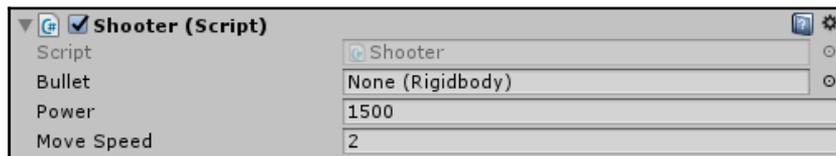
In order for this script to be used within our game, it must be attached as a component of one of the `GameObjects` within the existing scene.

Save your script by choosing **File | Save** from the top menu of your script editor and return to Unity. There are several ways to assign a script to an object in Unity:

- Drag it from the **Project** view and drop it onto the name of an object in the **Hierarchy** panel
- Drag it from the **Project** view and drop it onto the visual representation of the object in the **Scene** panel
- Select the object you wish to apply the script to and then drag and drop the script to empty space at the bottom of the **Inspector** view for that object
- Select the object you wish to apply the script to and then choose **Component | Scripts** | and then the name of your script from the top menu

The most common method is the first approach, and this would also be the most appropriate as trying to drag to the camera in the **Scene** view, for example, would be difficult, as the camera itself doesn't have a tangible surface to drag to.

For this reason, drag your new `Shooter` script from the **Project** view and drop it onto the name of **Main Camera** in the **Hierarchy** to assign it, and you should see your script appear as a new component, following the existing **Audio Listener** component. You will also see its three public variables, `bullet`, `power`, and `moveSpeed`, shown in the **Inspector**, as follows:



You can, alternatively, act in the **Inspector** directly; press the **Add Component** button and look for `Shooter` by typing in the search box. You must also remember that this is only valid if you didn't add the component in this way initially. In that case, the `Shooter` component will be already attached to the camera `GameObject`.

As you will see, Unity has taken the variable names and given them capital letters, and in the case of our `moveSpeed` variable, it takes a capital letter in the middle of the phrase to signify the start of a new word in the **Inspector**, placing a space between the two words when seen as a public variable.

You can also see here that the `bullet` variable is not yet set, but it is expecting an object to be assigned to it that has a `Rigidbody` attached—this is often referred to as being a `Rigidbody` object. Despite the fact that, in Unity, all objects in the scene can be referred to as `GameObjects`, when describing an object as a `Rigidbody` object in scripting, we will only be able to refer to properties and functions of the `Rigidbody` class. This is not a problem, however; it simply makes our script more efficient than referring to the entire `GameObject` class. For more on this, take a look at the script reference documentation for both the classes:

- **GameObject:** <http://unity3d.com/support/documentation/ScriptReference/GameObject.html>
- **Rigidbody:** <http://unity3d.com/support/documentation/ScriptReference/Rigidbody.html>



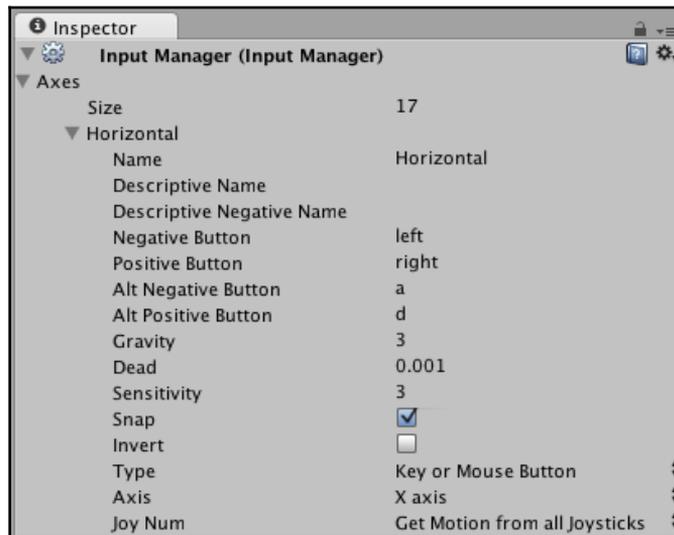
Be aware that when adjusting values of public variables in the **Inspector**, at stop time any values changed will overwrite the defaults specified in the script. This mechanism is called serialization and is very useful for letting developers set values from the **Inspector**. If you want to have a public variable but want it hidden from serialization, you should specify `[HideInInspector]` in front of the declaration.

Let's continue working on our script and add some interactivity, so return to your script editor now.

Moving the camera

Next, we will make use of the `moveSpeed` variable, combined with keyboard input, in order to move the camera and effectively create a primitive aiming of our shot, as we will use the camera as the point at which to shoot from.

As we want to use the arrow keys on the keyboard, we need to be aware of how to address them in the code first. Unity has many inputs that can be viewed and adjusted using the **Input Manager**; choose **Edit | Project Settings | Input**:



As seen in the screenshot above, two of the default settings for input are **Horizontal** and **Vertical**. These rely on an axis-based input that when holding the **Positive Button** builds to a value of 1, and when holding the **Negative Button** builds to a value of -1. Releasing either button means that the input's value springs back to 0, as it would if using a sprung analog joystick on a gamepad. Because input is also the name of a class, and all named elements in the **Input Manager** are axes or buttons, in scripting terms we can simply use the following:

```
Input.GetAxis("Horizontal");
```

This receives the current value of the horizontal keys—a value between -1 and 1 depending on what the user is pressing. Let's put that into practice in our script now, using local variables to represent our axes.

By doing this, we can modify the value of this variable later using multiplication, taking it from a maximum value of 1 to a higher number and allowing us to move the camera faster than 1 unit at a time.

This variable is not something we will ever need to set inside the **Inspector**, as Unity is assigning values based upon our key input. As such, these values can be established as local variables.

Local, private, and public variables

Before we continue, let's take an overview of local, private, and public variables in order to cement your understanding:

- **Local variables:** These are variables established inside a function; they will not be shown in the **Inspector**, and are only accessible to the function they are within.
- **Private variables:** These are established outside of a function, and therefore accessible to any function within your class. However, they are not visible in the **Inspector**.
- **Public variables:** These are established outside of a function, accessible to any function in their class and also to other scripts, as well as being visible for editing in the **Inspector**.

Local variables are declared and used as follows:

```
void Update() {  
    float h = Input.GetAxis("Horizontal") *  
        Time.deltaTime * moveSpeed;  
    float v = Input.GetAxis("Vertical") *  
        Time.deltaTime * moveSpeed;  
}
```

The variables declared here—`h` for `Horizontal` and `v` for `Vertical`—could be named anything we like; it is simply quicker to write single letters. Generally speaking, we would normally give these a name, because some letters cannot be used as variable names, for example, `x`, `y`, and `z` because they are used for coordinate values and therefore reserved for use as such.

As these axes' values can be anything from `-1` to `1`, they are likely to be a number with a decimal place, and as such we must declare them as floating point-type variables. They are then multiplied using the `*` symbol by `Time.deltaTime`—this simply means that the value is divided by the number of frames per second (the `deltaTime` is the time it takes from one frame to the next or the time taken since the `Update()` function last ran), meaning that the value adds up to a consistent amount per second regardless of the frame rate.

The resultant value is then increased by multiplying it by the public variable we made earlier, `moveSpeed`. This means that although the values of `h` and `v` are local variables, we can still affect them by adjusting the public `moveSpeed` in the **Inspector**, as it is part of the equation that those variables represent. This is common practice in scripting as it takes advantage of the use of publicly-accessible settings combined with specific values generated by a function.

Understanding Translate

To actually use these variables to move an object, we will use the `Translate` command. When implementing any piece of scripting, you should make sure you know how to use it first.

`Translate` is a command which is part of the `Transform` class:

<http://unity3d.com/support/documentation/ScriptReference/Transform.html>.

This is a class of information that stores the position, rotation, and scale properties of an object, as well as functions that can be used to move and rotate the object.

The expected usage of `Translate` is as follows:

```
Transform.Translate(Vector3);
```

The use of `Vector3` here means that `Translate` is expecting a piece of `Vector3` data as it's the main argument, `Vector3` data is simply information that contains a value for the `x`, `y`, and `z` coordinates; in this case, coordinates to move the object by.

Implementing Translate

Now let's implement the `Translate` command by taking the `h` and `v` input values that we have established and placing them into `Vector3` within the command.

Place the given line within the `Update()` function in your script, meaning after the opening curly brace of `Update()` { and before the function closes with a right curly brace }. Note that this does not differ between languages:

```
transform.Translate(h, v, 0);
```

We can use just the word `transform` here because we know that any object we attach this script to will have a `Transform` component. Attached components of an object can be addressed using the lowercase version of their name, whereas accessing components on other objects requires the use of the `GetComponent` command and their uppercase equivalent, for example:

```
gameObject.GetComponent<Transform>().Translate(h, v, 0);
```

However, we do not need that in this instance. Accessing components on other objects is covered later in this book in [Chapter 4, Player Controller and Further Scripting](#).

Here, we are using the current value of `h` to affect the x axis, `v` to affect the y axis, and simply passing a value of `0` to the z axis, as we do not wish to move back and forth.

Save your script now by going to **File | Save** in your script editor and return to Unity. Save the scene we have worked on thus far by going to **File | Save Scene As**, and name this scene `Prototype`.

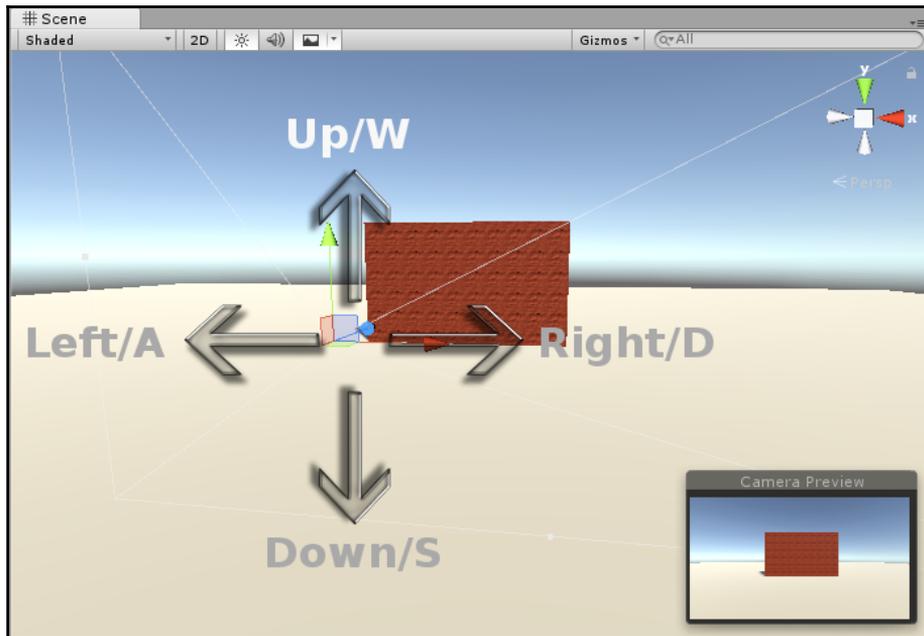
Unity will prompt you to save into the `Assets` folder of your project by default and you should always ensure that you do not save outside of this folder as you will not be able to access the scene through the **Project** view otherwise. You may also create a subfolder within `Assets` in which to keep your scenes if you wish to be extra tidy. This is not necessary but is generally considered to be good practice.

Testing the game so far

In Unity, you can playtest at any time, provided there are no errors in your scripts. If there are, Unity will ask you to fix all errors before allowing you to enter the Play Mode.

Once all errors are fixed, this will be signified by an empty or cleared **Console** bar at the bottom of the Unity interface, the **Console** bar represents the most recent entry into the Unity console. You can check this by choosing **Window | Console** (shortcuts `Ctrl + Shift + C` [for Windows] and `command + Shift + C` [for Mac]) from the top menu. All the errors will be listed in red, and double-clicking on the error will take you to the part of the script that is causing the issue described. Most errors are often a forgotten character or a simple misspelling, so always double-check what you have written as you go.

If your game is free of errors, click the Play button at the top of the screen to enter Play Mode. You will now be able to move the Main Camera object around by using the arrow keys: *up*, *down*, *left*, and *right*, or their alternates *W*, *A*, *S*, and *D*, as shown in the following screenshot:



Once you have tested and confirmed that this works, press the Play button at the top of the interface to switch off Play Mode.



Switching off Play Mode before continuing to work is important because the settings of components and objects in the current scene that are adjusted during Play Mode will be discarded as soon as Play Mode is switched off. Therefore, leaving Unity in Play Mode as you continue to work will mean you lose work. However, with the latest version of Unity, you can copy component values while in Play Mode after having tweaked them while running the game, and then, when stopped, have them *pasted* on the component to replace the original ones by right-clicking on the component and choosing **Paste Component Values**.

Now let's finish our game mechanic prototype by adding the ability to shoot projectiles at the wall to knock it down.

Making a projectile

In order to shoot a projectile at the wall, we will need to first create it within the scene and then store it as a prefab.



A prefab is a `GameObject` stored as an asset in the project that can be instantiated. It is created during runtime and can hence be manipulated through code.

Creating the projectile prefab

Begin by clicking the **Create** button at the top of the **Hierarchy** and then selecting **Sphere** from the drop-down menu that appears. As mentioned previously, you can also access primitive creations from the **GameObject | Create Other** top menu.

Now, ensure that the **Sphere** object is selected in the **Hierarchy**, hover your cursor over the **Scene** view, and press *F* on the keyboard to focus your view on the **Sphere**.



If your Sphere has been created at the same position as one of your other objects, simply switch to the **Translate** tool (*W*) and drag the relevant axis handle until your Sphere is out of the way of the object blocking your view. Refocus your view by pressing *F* again.

Taking a look at the **Inspector** panel, you will notice that when introducing new primitive objects to the scene, Unity automatically assigns them three new components in addition to the existing Transform component, which are as follows:

- **Mesh Filter:** This component is to handle the shape
- **Renderer:** This component is to handle the appearance
- **Collider:** This component is to manage interactions (known as collisions) with other objects

Creating and applying a material

We'll begin with the visual appearance of the projectile, and will alter this by creating a material to apply to the renderer. Whenever you need to adjust the appearance of an object, you'll likely look to alter settings in some kind of Renderer component.

For 3D objects, it will be the Mesh Renderer, on particle systems it will be a particle renderer, and so on.

To keep things neat, we'll make a new folder within our `Assets` folder to store all the materials that we will create in this project. On the **Project** view, click the **Create** button and choose **Folder** from the drop-down menu. Rename this folder `Materials` by pressing *Return* (Mac) or *F2* (Windows). Take the time now to place the red brick material you made earlier inside this new folder.



To create any new asset inside of an existing folder in the **Project** view, simply select the folder first and then create using the **Create** button.

Now we will create the material we need and apply it to our object:

1. With the new `Materials` folder still selected, click the **Create** button on the **Project** view once again, and this time choose `Material`. This creates a new material asset that you should rename `bulletColor`—or something of your own choosing that reminds you that this asset is to be applied to the projectile.
2. With your new material still selected, click on this block to open the **Color Picker** window, select a shade of blue, and then close this window when you are happy with your selection.
3. Now that you have chosen your color for the material, drag and drop the `bulletColor` material from the **Project** view and drop it onto the name of the **Sphere** object in the **Hierarchy** to assign it.



Note that if you want to test how a material will look when applied to a 3D object in Unity, you can drag the material to the **Scene** view, hovering the cursor over meshes. Unity will show you a preview of what it will look like and you can then move the mouse away or press *Esc* if you wish to cancel, or release the mouse to apply.

Adding physics with a Rigidbody

Next, we'll ensure that the physics engine has control of the projectile **Sphere** by adding a **Rigidbody** component. Select the **Sphere** in the **Hierarchy** panel and choose **Component | Physics | Rigidbody** from the top menu.

Your **Rigidbody** component is now added, with settings available to adjust in the **Inspector**. For the purpose of this prototype, however, we needn't adjust any settings from the default.

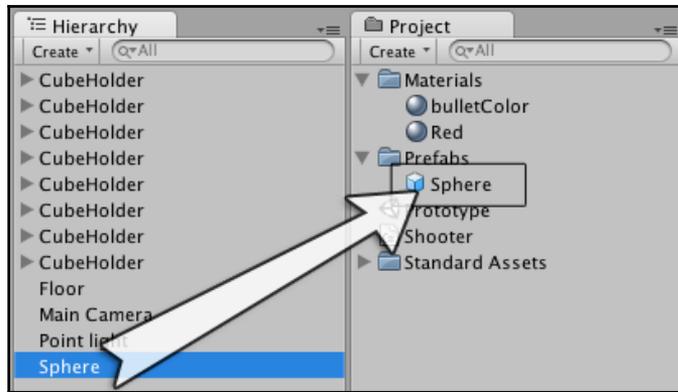
Saving prefabs

As we wish to fire this projectile when the player presses a key, we do not want the projectile to be in the scene by default, but instead want it to be stored and created when the key is pressed. For this reason, we will store the object as a prefab and use our script to instantiate (that is, create an instance of) it at the precise moment a key is pressed.



Prefabs are Unity's way of storing GameObjects that have been set up in a particular way; for example, you may have configured an enemy soldier with particular scripts and properties that behaves a certain way. You can store this object as a prefab and instantiate it when necessary. Similarly, you might have a differing soldier that behaves differently; this might be a different prefab, or you might create an instance of the first and adjust the settings in the soldier's components, making him faster or slower upon instantiation. The prefab system gives you a lot of flexibility in this regard.

Dragging the GameObject from the **Hierarchy** into the **Project** view automatically saves a new prefab. Dragging a GameObject into an existing Prefab, will overwrite the Prefab:



Click the **Create** button at the top of the **Project** view and choose **Folder**, then rename this `Prefabs`. Now drag the **Sphere** from the **Hierarchy** and drop it onto the `Prefabs` folder in the **Project** view, as shown in the previous screenshot. Dragging a GameObject such as this anywhere into the **Project** view will save it as a prefab; we simply created this folder for neatness and good practice. Rename this new prefab `Projectile`.

You can now delete the original **Sphere** GameObject from the **Hierarchy** by selecting it and pressing *command + backspace* (Mac) or *Delete* (Windows); alternatively, you can right-click the object in the **Hierarchy** and choose **Delete** from the pop-out menu that appears.



You can also rename the **Sphere** GameObject into `Projectile`. Then, dragging into the **Project** view will create a prefab with that name instead. You can also right-click on the **Project** view and select **Create | Prefab**, rename it `Projectile`, and then drag to it the **Sphere** object from the **Hierarchy**.

Firing the projectile

Return to the `Shooter` script we have been working on so far by double-clicking its icon in the **Project** view, or by selecting it in the **Project** view and clicking the **Open** button at the top of the **Inspector**.

Now we will make use of the `bullet` variable we declared earlier, using it as a reference to the particular object we wish to instantiate. As soon as the object is created from our stored prefab, we will apply a force to it in order to fire it at the wall in our scene. Go to the following line:

```
transform.Translate(h, v, 0);
```

After this line, within the `Update()` function in your script, add the following code:

```
if (Input.GetButtonUp("Fire1")) { }
```

This `if` statement listens for the key applied to the `Input` button named `Fire1` to be released. By default, this is mapped to the left *Ctrl* key or left mouse button, but you can change this to a different key by adjusting the settings in the **Input Manager (Edit | Project Settings | Input)**.

Using `Instantiate()` to spawn objects

Now, within this `if` statement, meaning after the opening `{` and before the closing `}`, put the following line:

```
Rigidbody instance = Instantiate(bullet, transform.position,  
transform.rotation) as Rigidbody;
```

Here, we are creating a new variable named `instance`. In this variable, we are storing a reference to the creation of a new object that is of type `Rigidbody`. The `Instantiate` command requires three pieces of information, namely what to make, where to make it, and a rotation to give it.

In our example, we are telling our script to create an instance of a `GameObject` or prefab assigned to the `bullet` public member variable. We want it to be created using the values of **Position** and **Rotation** from the **Transform** component of the `GameObject` that this script is attached to. That's why you will often see `transform.position` written in scripts, as it refers to the **Transform** component's **Position** of the `GameObject` with the attached script.

Adding a force to the Rigidbody

Having now created our object, we need it to be immediately fired forward using the `AddForce()` command:

```
Rigidbody.AddForce(Direction and amount of force expressed as a Vector3);
```

Before we add the force, we will create a reference to the direction we wish to shoot it in. The camera is facing the brick wall, so it makes sense to shoot objects at the wall in the camera's forward direction. Following the `Instantiate` line you just added, still within the `if` statement, after the `Instantiate` instruction, add the given code:

```
Vector3 fwd = transform.TransformDirection(Vector3.forward);
```

Here, we have created a `Vector3` type variable called `fwd`, and have told it to represent the forward direction of the current transform this script is attached to. The `TransformDirection` command can be used to convert a local direction, that of the forward direction of the camera, to a world direction, as objects and the world each have their coordinate system, and the forward direction of an object may not necessarily match that of the world, so conversion is crucial. `Vector3.forward` in this context is simply a shortcut to writing `Vector3(0, 0, 1)`. It is one unit in length on the `z` axis.

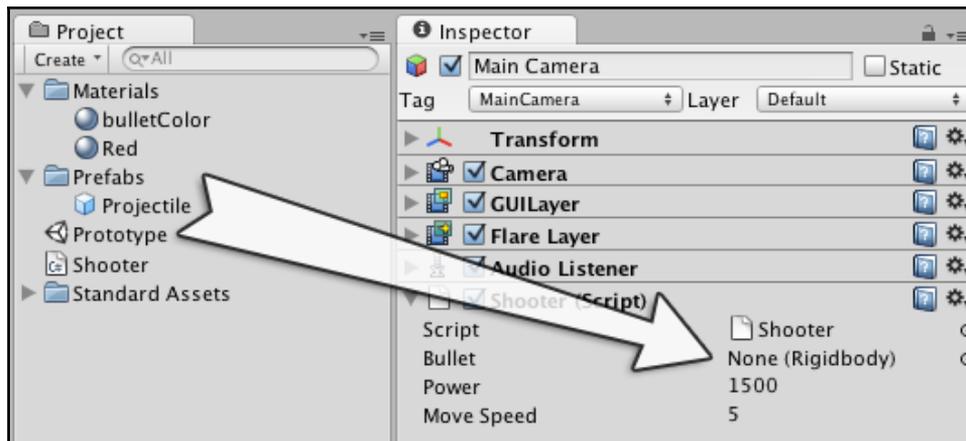
Finally, we will apply the force by first referring to our variable that represents the newly created object, `instance`, then using the `AddForce()` command to add a force in the direction of the `fwd` variable, multiplied by the public variable named `power` that we created earlier. To add force, we must insert the following beneath the last line you added:

```
instance.AddForce(fwd * power);
```

Save your script and return to Unity Editor.

Now, before we can test the finished game mechanics, we need to assign the Projectile prefab to the `bullet` public variable. To do this, select the **Main Camera** in the **Hierarchy** in order to see the `Shooter` script as a component in the **Inspector**.

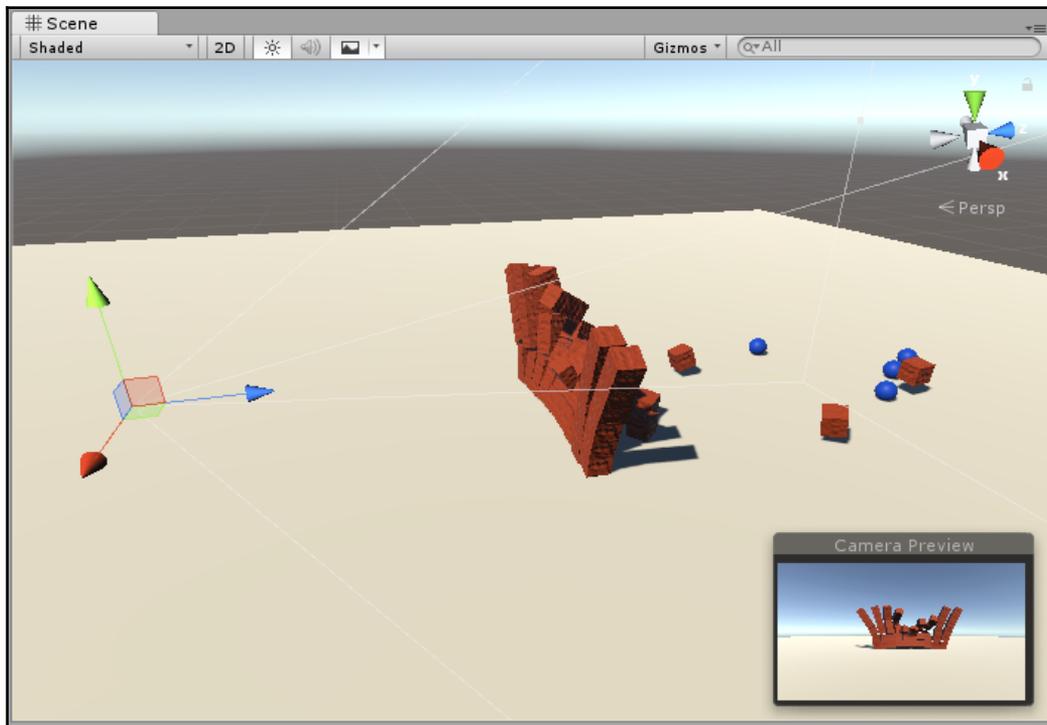
Now, drag the **Projectile** prefab from the **Project** view and drop it onto the `bullet` variable in the **Inspector**, where it currently says **None (Rigidbody)**, as shown in the following screenshot:



Once applied correctly, you will see the name of the projectile in this variable slot. Save the scene now (**File | Save Scene**) and test the game by pressing the Play button at the top of the interface.

You will now be able to move the camera around and fire the projectiles we created earlier using the left *Ctrl* on the keyboard. If you wish to adjust how much power you give to the projectiles when they are fired, simply adjust the number in the `power` public variable by selecting the **Main Camera** and adjust the value of 1500 currently assigned to it in the `Shooter` script component, increasing or decreasing as you see fit.

Remember to always press the Play button again to stop testing:



To read more about Rigidbody and how to apply force in different ways for different game cases, read the official Unity Manual, at <https://docs.unity3d.com/ScriptReference/Rigidbody.html>.

Resetting the wall to initial state and clearing the projectiles

There are a lot of ways to implement this. In this chapter, we will do it in the simplest way possible. We will basically use the `SceneManager.LoadScene` API to reload the level.

In the next chapter, we will discover how to use `GameObject` layers and tags to choose among objects in a complex scene, as well as how to `Destroy()` the objects individually or per group.

After the first line of the C# script add the following:

```
using UnityEngine.SceneManagement;
```

Within the `Update()` function in your script, add the following code:

```
if (Input.GetButtonUp("Cancel")) {  
    SceneManager.LoadScene("Prototype");  
}
```

Test the game and, when you have destroyed part of the wall by shooting some bullets, press the *Esc* key.

Summary

Congratulations! You have just created your first Unity prototype. In this chapter, you should have become familiar with the basics of using the Unity interface, and working with `GameObjects`, components, and basic scripting. This will hopefully act as a solid foundation upon which we can build further experience in Unity game development.

Now you might want to relax a little and take time to play your prototype, or even create one of your own based on what you have learned. Or you may just be eager to learn more; if so, keep reading! Let's move on to the main game of this book, now that you are a little more prepared on some of the basic operations of Unity game development.

3

Creating and Setting Game Assets

Unity was born as a 3D game engine and 2D tools were not added until Unity 4.3. Before then, there were simple UI tools only.

The rise of mobile platforms has been in part thanks to their popularity with indie developers, who prefer the short development cycles. The most prevalent medium on mobile is 2D and Unity has a host of features that support 2D game development, including Sprite Editing and Packing, as well as physics specifically designed for 2D games.

As it evolved to versions 4 and 5, Unity introduced a new 2D system that blew away the previous one.

The Sprite Editor and the Sprite Packer allow you to pack the Sprites into a single Sprite atlas texture; in this way, instead of loading multiple files to the GPU, the textures can be sent as one, which in turn saves memory and increases performance. The cool thing about making 2D games with Unity3D is that now you can take advantage of the full potential of accelerated hardware to draw things on screen with the great design potential of a **WYSIWYG** editor, such as Unity's one.



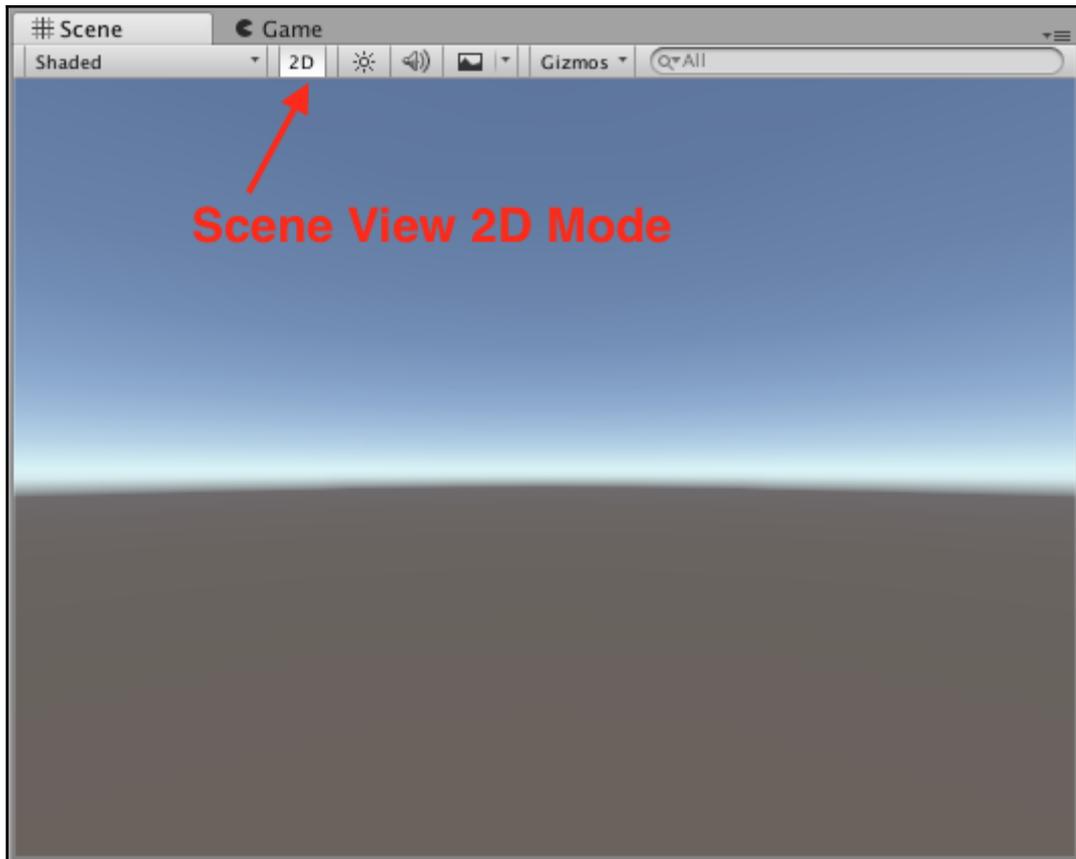
What You See Is What You Get (WYSIWYG) is the term for such visual editors, just like MS Word is for text, Adobe Photoshop or The Gimp for images, and Maya or 3DSMax for modeling, rigging, and animation. What you see in Unity Editor is exactly what you will get when you build the final executable.

In this chapter, we will build our first simple 2D game prototype and you will learn the following things:

- Importing and inserting images as Sprites that we can use for the environment
- Using third-party code to manage a parallax scrolling effect
- Importing animated characters from Sprite sheets
- Giving life to characters with Animator's state-machine
- Making the game more difficult by adding a timer
- Basics of 2D physics
- Modifying the existing and writing new C# code for the game logic, player, and collectables
- Spawning collectable objects from a given spot in random directions
- Adding a working fake blob shadow for the hero
- Adding some basic audio for music soundtrack and sound effects
- Understanding how to make Sprites be influenced by real-time lighting by changing the material/shader

Setting up the scene and preparing game assets

Create a new scene from the main menu by navigating to **Assets | Create | Scene**, and name it `ParallaxGame`. In this new scene, we will set up, step by step, all the elements for our 2D game prototype. First of all, we will switch the camera setting in the **Scene** view to 2D by clicking on the button as shown by the red arrow in the following screenshot:



As you can see, now the **Scene** view camera is orthographic. You can't rotate it as you wish, as you can do with the 3D camera. Of course, we will want to change this setting on our Main Camera as well.

Also, we want to change the Orthographic size to 4.5 to have the correct view of the scene. Instead, for the Skybox, we will choose a very dark or black color as *clear color* in the depth setting. This is how the **Inspector** should look when these settings are done:

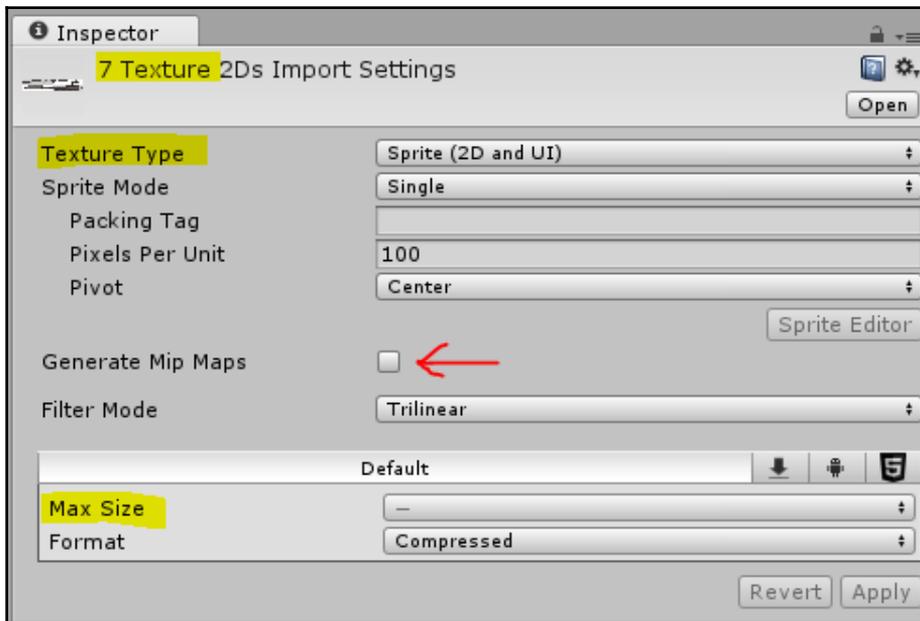


While the **Clipping Planes** distances are important for setting the size of the frustum cone of a 3D, for the Perspective camera (inside which everything will be rendered by the engine), we should only set the **Orthographic Size** to 4.5, to have the correct distance of the 2D camera from the scene.

When these settings are done, proceed by importing `Chapter2-3-4.unitypackage` into the project. You can either double-click on the package file with Unity open, or use the top menu: **Assets | Import | Custom Package**. If you haven't imported all the materials from the book's code already, be sure to include the `Sprites` subfolder. After the import, look in the `Sprites/Parallax/DarkCave` folder in the **Project** view and you will find some images imported as textures (as per default).

The first thing we want to do now is to change the import settings of these images, in the **Inspector**, from **Texture** to **Sprite (2D and UI)**. To do so, select all the images in the **Project** view in the `Sprites/Parallax/DarkCave` folder, all except the `_reference_main_post` file.

Which is just a picture used as a reference of what the game level should look like:

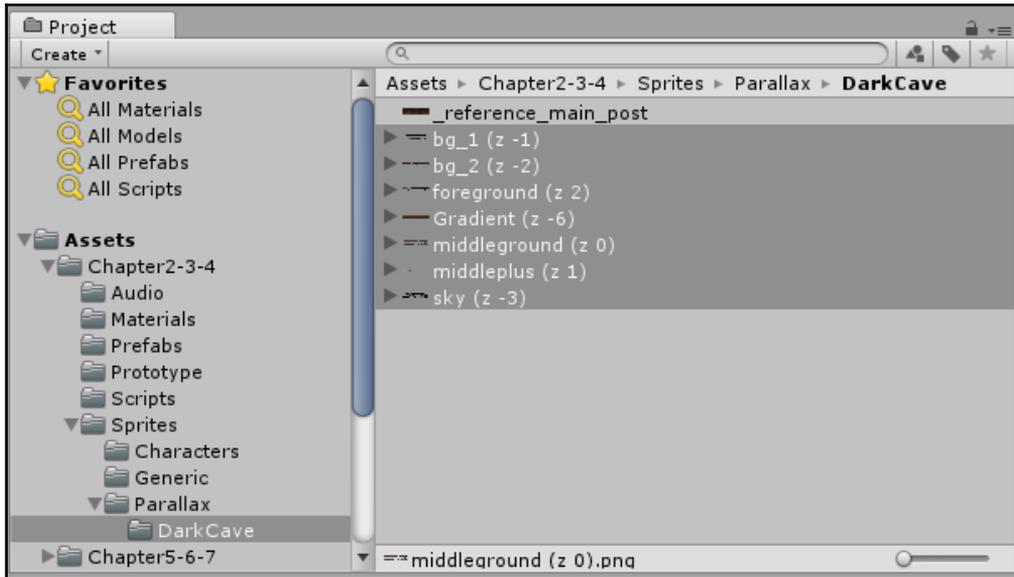


The Import Settings shown in the Inspector after selecting the seven images in the Project view

The **Max Size** setting is hidden (-) because we have a multi-selection of image files. After having made the multiple selections, again, in the **Inspector**, we will do the following:

1. Set the **Texture Type** option to **Sprites (2D and UI)**. By default, images are imported as textures; to import them as Sprites, this type must be set.
2. Uncheck the **Generate Mip Maps** option as we don't need MIP maps for this project as we are not going to look at the Sprites from a distant point of view, for example, games with the zoom-in/zoom-out feature (like the original *Grand Theft Auto 2D* game) would need this setting checked.
3. Set **Max Size** to the maximum allowed. To ensure that you import all the images at their maximum resolution, set this to 8192. This is the maximum resolution size for an image on a modern PC, imported as a Sprite or texture. We set it so high because most of the background images we have in the collection are around 6,000 pixels wide.

4. Click on the **Apply** button to apply these changes to all the images that were selected:

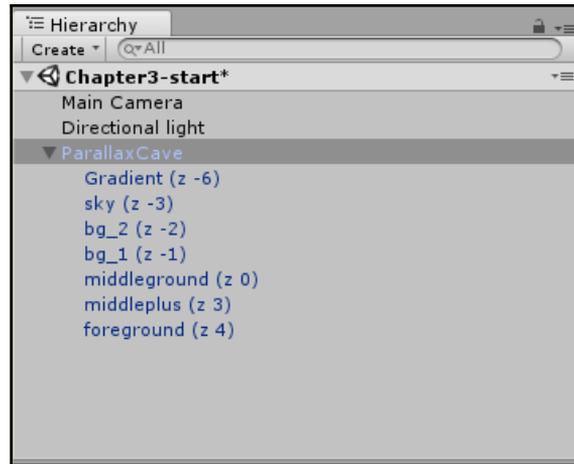


The Project view showing the content of the folder after the images have been set to Sprite in the Import Settings

Placing the prefabs in the game

Unity can place the prefabs in the game in many ways, the usual, visual method is to drag a stored prefab or another kind of file/object directly into the scene. Before dragging in the Sprites we imported, we will create an empty `GameObject` and rename it `ParallaxCave`.

We will drag the layer images we just imported as Sprites, one by one, from the **Project** view (pointing at the `Assets/Chapters2-3-4/Sprites/Background/DarkCave` folder) into the **Scene** view, or more simply, directly in the **Hierarchy** view as the children of our `ParallaxCave` `GameObject`, resulting in a scene **Hierarchy** like the one illustrated here:



You can't drag all of them instantly because Unity will prompt you to save an animation filename for the selected collection of Sprites; we will see this later for our character and for the collectable graphics.



The `ParallaxCave` `GameObject` and its children are in blue because this `GameObject` is stored as a prefab. When the link with the prefab is broken for a modification, the `GameObject` in the **Hierarchy** will become black again.

When you see a red `GameObject` in the scene, it means that the prefab file that was linked to that `GameObject` was deleted.

Importing and placing background layers

In any game engine, 2D elements, such as Sprites, are rendered following a *sort order*; this order is also called the z-order because it is a way to express the depth or to cope with the missing z axis in a two-dimensional context. The sort order is assigned an integer number which can be positive or negative; 0 is the middle point of this draw order.

Ideally, a sort order of zero expresses the middle ground, where the player will act, or near its layer. Look at this image:

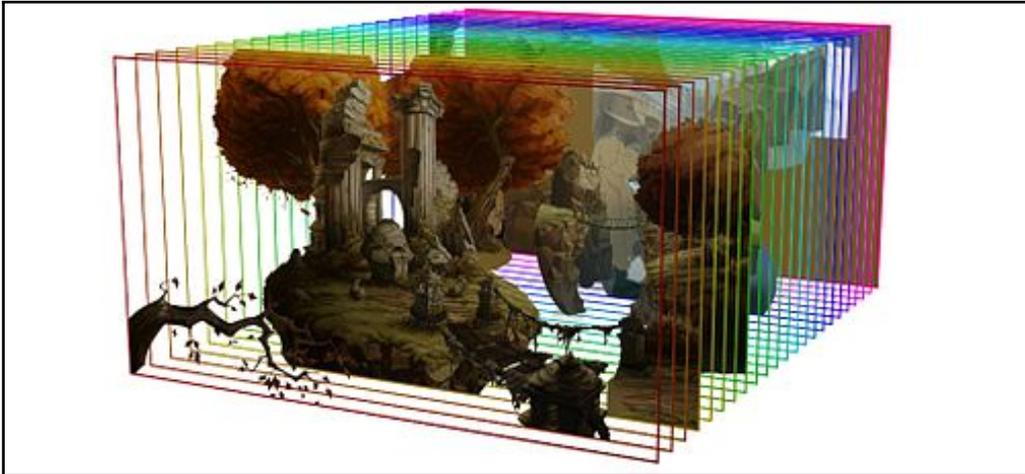


Image courtesy of Wikipedia: parallax scrolling

All positive numbers will render the Sprite element in front of the other elements with a lower number. The graphic set we are going to use was taken from the Open Game Art website at <http://opengameart.org>.

For simplicity, the provided background image files are named with a number within parentheses, for example, `middleground(z1)`, which means that this image should be rendered with a z sort order of 1.

Change the *sort order* property of the Sprite component on each child object under `ParallaxCave` according to the value in the parentheses at the end of their filenames. This will rearrange the graphics into the appropriately sorted order.

After we place and set the correct layer order for all the images, we should arrange and scale the layers in a proper manner to end as something like the reference image furnished in the `Assets/Chapters2-3-4/Sprites/Background/DarkCave/` folder.

You can take a look at the final result for this part anytime, by saving the current scene and loading the `Chapter3_start.unity` scene.



On the optimization side, Sprites can be packed together in a single atlas texture with the Sprite Packer into a single image atlas (a single image containing a whole group of Sprites). We will see this passively with already-made-for-the-book Sprite sheets containing the animations of the character in this chapter and actively for packing all the generic graphics to make the best running performance possible in Chapter 13, *Optimization and Final touches*.

Implementing parallax scrolling

Parallax scrolling is a graphic technique where the background content (that is, an image) is moved at a different speed than the foreground content while scrolling. The technique was derived from the multiplane camera technique used in traditional animation since the 1930s. Parallax scrolling was popular in the 1980s and early 1990s and started to see light with video games such as *Moon Patrol* and *Jungle Hunt*, both released in 1982. On such a display system, a game can produce parallax by simply changing each layer's position by a different amount in the same direction. Layers that move more quickly are perceived to be closer to the virtual camera. Layers can be placed in front of the playfield, the layer containing the objects with which the player interacts, for various reasons, such as to provide increased dimension, obscure some of the action of the game, or distract the player.

Here follows a short list of the first parallax scrolling games which made the history of video games:

- *Moon Patrol* (Atari, 1982)
 - <https://youtu.be/HBOKWCpwGfM>
 - https://en.wikipedia.org/wiki/Moon_Patrol
- *Shadow of the Beast* (Psygnosis, 1989)
 - <https://youtu.be/w6Osnolfxqw>
 - https://en.wikipedia.org/wiki/Shadow_of_the_Beast

- *Super Mario World* (Nintendo, 1990)
 - <https://www.youtube.com/watch?v=htFJTtVH5Ao>
 - https://en.wikipedia.org/wiki/Super_Mario_World
- *Sonic The Hedgehog* (Sega, 1991)
 - <https://youtu.be/dws4ij2IFH4>
 - [https://en.wikipedia.org/wiki/Sonic_the_Hedgehog_\(1991_video_game\)](https://en.wikipedia.org/wiki/Sonic_the_Hedgehog_(1991_video_game))

Making it last forever

There are many roads we could take to make the hero run last forever and to achieve parallax scrolling. You can find a lot of different ready-made solutions in the **Asset Store** and there are also many **General Public License (GPL)** open source pieces of code written in C that we could take inspiration from.

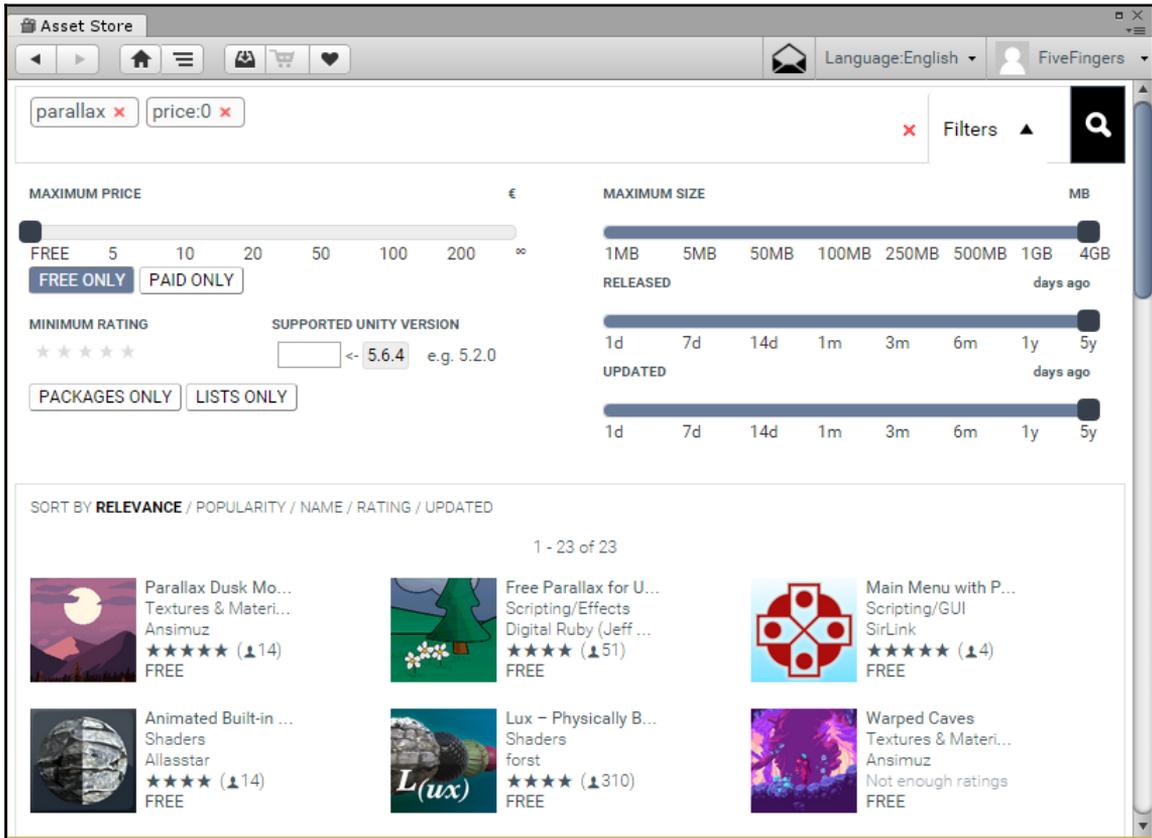
Using the Asset Store

I chose FreeParallax from the **Asset Store** because it is powerful, free, and a well-written piece of code. Also, the modifications needed to achieve our game prototype on this class are very few. First of all, I wish to thank Jeff Johnson for this great piece of code and for making it available for free for the community and having put it under the MIT license; read more info at: https://en.wikipedia.org/wiki/MIT_License.

Let's download and import the system from the **Asset Store**. First, navigate to <http://u3d.as/bvv>:

Click on the **Open in Unity** button to allow Unity to open this entry in the **Asset Store** window.

You can, alternatively, search for the package directly in Unity by opening the store from the top menu: **Windows | Asset Store** (recommended). In the search box type: `parallax`; also choose **FREE ONLY** like in this screenshot:



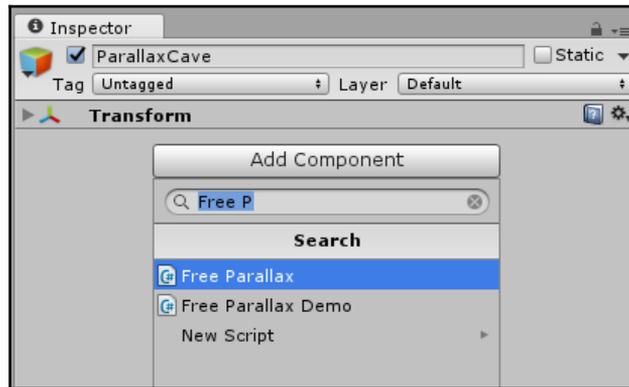
You should now find the correct entry, the Free Parallax for Unity(2D) package. You can now download the package and import it into your project straight away.



After importing the package, you can safely delete the demo scene and demo script as we don't need them, but you can also keep them for reference if you want. Get ready to implement it in our 2D game!

The FreeParallax component

Let's dive in right away. Select the `ParallaxCave` object in the scene and click on the **Add Component** button:



Type `parallax` or `free p` in the search box of the panel to search for the *Free Parallax C#* component, then double-click on it to add it to the `GameObject`. Once it is added, you will see its configuration in the **Inspector** panel.

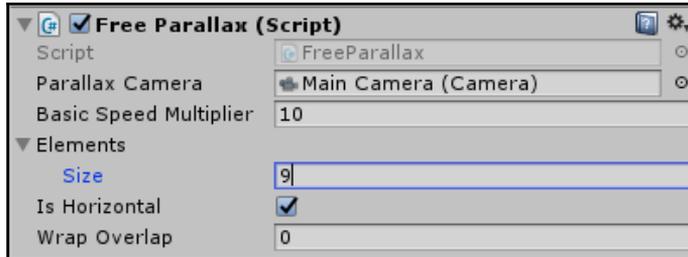
To set it up, follow this simple checklist:

1. Drag your Main Camera into the Parallax Camera slot
2. Leave 10 for the speed multiplier
3. Keep **Is Horizontal** checked
4. Put 8 in the **Wrap Overlap** field

The `FreeParallax` component will take care of drawing, cloning, and repeating the background layers. Also, in this component, we will need to set the same sorting order that we set earlier in the configuration of its elements.

Understanding and setting up elements

Open the **Elements** section and create nine elements by inserting 9 in the **Size** box:

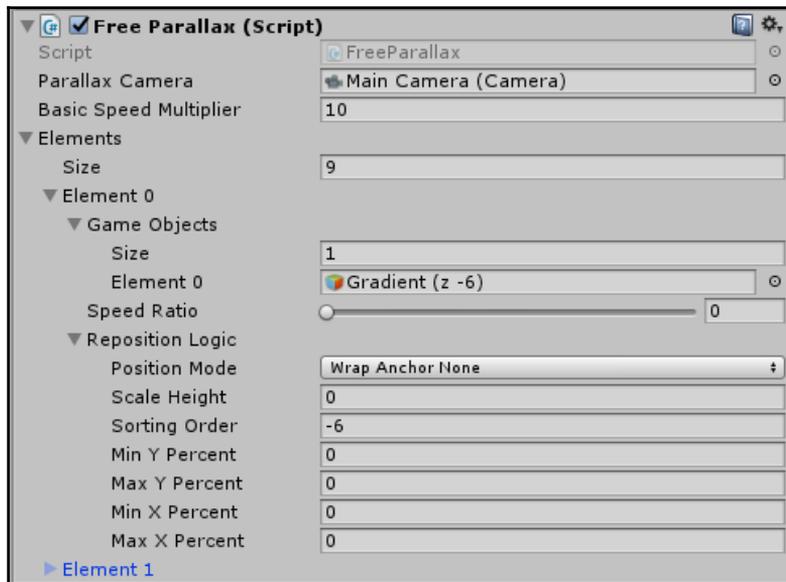


Now, let's set them up. Let's start with the first layer, which will appear to be furthest from the camera and, as a result, needs to move the slowest. This layer will be the brown gradient image which resides in the child GameObject.

At this point, we will perform the following steps:

1. Open the first element of the list
2. Add at least one item to the GameObjects list to host the correct object
3. Assign the `Gradient (z-6)` GameObject to the Parallax Element by dragging it into that slot
4. Just add in the sort order
5. Set the **Speed Ratio** to zero because this layer will not scroll, and leave the other settings untouched

The settings for the first layer element are as illustrated:



Repeat the same steps for the next three layers and set the **sky(z-3)**, **bg_1(z-2)**, and **bg_2(z-1)** objects for each of them, setting 0.05, 0.2, and 0.3 for the **Speed Ratio**, in this sequence.

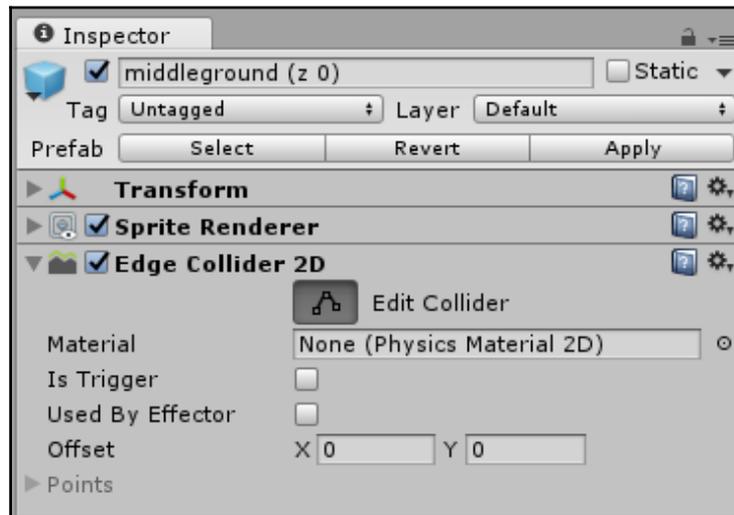


A **Speed Ratio** of 1 (max value) means full speed. The closer the layer is to the point of view, the faster we want to move the layer.

Playing in the midground

The midground is where the parallax scrolling action really takes place; we will use two layers for this part, where the real ground is the `midground (z 0)` GameObject.

On the middleground Sprite object, which is our *ground* layer, where the player will walk, we need to define the collision of the ground and we will use the **Edge Collider 2D**, which is specifically designed for this:



Select the `middleground (z 0)` GameObject and then add the **Edge Collider 2D** component to it by clicking on the **Add Component** button in the **Inspector** and searching for its name. Now click on the **Edit Collider** button at the top of the **Edge Collider 2D** component in the **Inspector**.

Keeping the object selected, you should see a green line in the **Scene** view, which represents the collider. It has only a start and an end point now, so we have to add some points to be able to create our edgy ground collider that will follow the path graphics. With the *Shift* key pressed, click on the green line and see a new point coming; you can drag this point on the two axes to set its position. Create the number of edges and vertices you think is correct to follow the player path on the middleground picture and eventually adjust their positions.

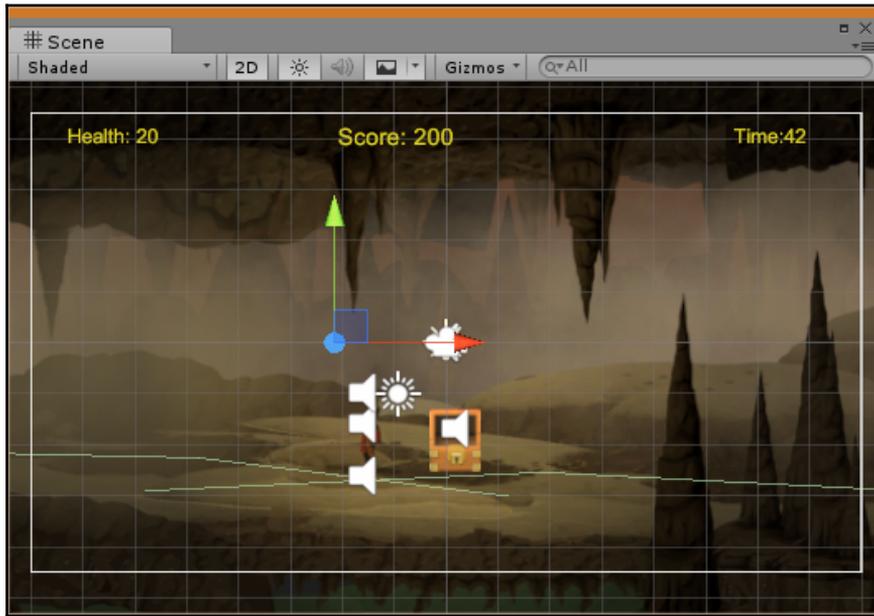
Avoiding holes and discrepancies on the ground collider

When editing the **Edge Collider 2D**, responsible for the ground collision between the player and the path walk, we must watch out for eventual nasty holes or gaps due to overlapping. This screenshot shows what happens when the game runs and the **FreeParallax** component clones and overlaps two middleground objects:



As we are using an overlap of eight units in the **FreeParallax** component, the cloned second middleground object will create an intersection between the two **Edge Collider 2D** lines. To avoid gaps like in the preceding screenshot, where the two colliders overlap creating a gap among their final and initial parts, we need to be make sure this doesn't happen or we might experience unwanted player behavior.

To fix an eventual analog situation, you may want to edit the Edge Collider 2D starting and final points, to obtain a situation where the two objects are overlapping as in this screenshot:



Now, repeat the same steps we did for the background layer for the next two layers and set `middleground(z 0)` and `middleground(z 3)` objects for each of them, setting `0.6` for both the **Speed Ratio** parameter of the elements. We chose `0.6` as the speed for `middleground` objects to be able to play the foreground layer at a faster speed.

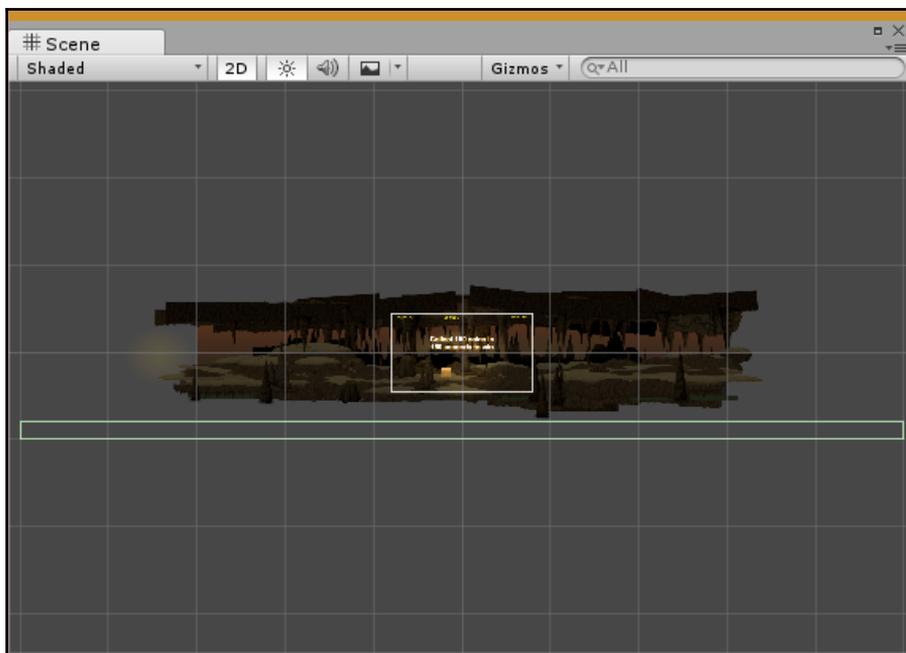
Putting in the foreground and special layers

We will repeat the previous steps for the `foreground(z 4)` object, but we will set a speed ratio of `1.0` to make it move faster. These layers will not wrap or repeat, but just pop out at a certain point on the screen. This is perfect for dynamic objects, collectables, or spawn points. This object will carry a script that will spawn collectable coins whenever the player hits its trigger. Add, as a child, the `goldchest(z1)` `GameObject`, which is obtained by dragging the imported `Sprite` in the scene to the `ParallaxCave` in the **Hierarchy** and set up **Element 8** in the `Element` list of the `FreeParallax` component in a way that will not wrap, but will use a floating number in the range `0.0-1.0` as a percentage value to determine in what area of the screen the object or layer should appear, using minimum and maximum values as ranges for the computation of the random spot.

To disable wrapping for a layer, choose **Individual Start Off Screen** for **Position Mode**.

Creating a death zone

Finally, we will create another empty `GameObject` and call it *Killzone*, and we will add a `Box Collider 2D` component (from the main menu: **Component** | **Physics2D** | **Box Collider 2D**) to it. Look at the following screenshot for reference and edit the box collider (in green) to make it way larger than the canvas area (in white) that represents the screen area that you will see in the **Game** view:



To make it interactive, add a new component, but this time you will create a new C# script; call it `Restarter.cs`. Double-click on the file or on the component slot in the **Inspector** to edit the file, which will be as follows:

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace UnityStandardAssets._2D
{
    public class Restarter : MonoBehaviour
    {
```

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
    else
    {
        Destroy(other.gameObject);
    }
}
}
```

Note that we simply reload the scene when the player collides with the kill zone to keep the game simple.

You can later expand this part, and have a `KillPlayer` method being called on the player to, for example, let him go to a dead animation and then respawn it to the start point or whatever may fit your game design. All other objects falling in this zone, accidentally or not, will be just destroyed.



Also note that the tag `Player` to check against is the tag reserved for the hero character we will set up in the next chapter. Here, we are preparing the code for the collision between the *death zone* and the player.

This class will just check for trigger collisions and make something only if the `GameObject` that entered the trigger is tagged `Player`. Checking the `GameObject.tag` property is just a way to filter out other collisions; it is one of the most used, but you could also check the **Layer** of the object, or even its name. You might want to add the code for managing other kinds of objects that might fall down with physics gravity in the vacuum space.

The *Killzone* will restart the game when the player eventually falls down below the scenery colliders. This `GameObject` will be a child of the `FreeParallax` object but it won't be drawn or referenced in any of its layers, so it is not added to the `Elements` list but is parented with the main `parallax GameObject` for coherency.

Spawning collectable items

This feature needs a small modification of the `FreeParallax` component class to be able to reset the opening status of the gold chest that will be repeated by the component.

To solve this problem, we will modify the `position` method of the `FreeParallax` component a little.

Go to line 120. You will be adding some code right after the following:

```
obj.transform.position = pos;
```

This is the piece of code to add:

```
if(obj.name== "goldchest(z1)") obj.SendMessage("CloseBox");
```

Do the same change after line 127 as well. This change will ensure that when the element is rendered again, the chest state will be reset, by closing it up, as if it was a new one, ready to be opened again the next time it is needed on a new screen.

Now we will take care of the `goldchest(z1)` `GameObject`, which will be responsible for spawning the collectables (the coins) in a decent manner, using for the very first time, a cool C# feature: coroutines.

It would be ugly if the 10 coins we want to spawn when the chest opens spawn all at the same time and same position. To spawn them consecutively with a small delay between each of them, we will use a coroutine. Coroutines can execute in *steps*, each one delayed by a certain amount of time, while the rest of the code execution keeps going seamlessly. This is implemented by bringing in the class the `System.Collections` API from the .NET Framework so that we will be able to use the `Yield` instruction and declare a function as a coroutine, with the `IEnumerator` keyword. Also, we will take advantage of real-time physics for the coin movement, hence we will give a pull to the object by adding a force to its `Rigidbody` after having instantiated it.

Look carefully at this C# class, that will manage the collectable Prefab:

```
using UnityEngine;
using System.Collections; // This is needed to bring in coroutines support

public class GoldChest : MonoBehaviour {
    public Sprite OpenChest; // A reference of the open chest graphic
    public Sprite ClosedChest; // A reference of the closed chest graphic
    public GameObject coin; // The reference to the prefab of the collectable
    to spawn off
```

```
private bool wasOpen; // Private boolean that determines if this
chest was already opened before to re-generate (see FreeParallax.cs
modifications)

// Trigger collision detection, if the player touches the chest
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Player" && !wasOpen)
    {
        GetComponent<SpriteRenderer>().Sprite = OpenChest;
        GetComponent<AudioSource>().Play();
        wasOpen = true;
        // Start our first coroutine ;-)
        StartCoroutine(SpawnCoins());
    }
}

// Coroutine for spawning the coins out of the chest
IEnumerator SpawnCoins()
{
    int coins = 10;
    GameObject go;
    Rigidbody2D rb;
    while (coins > 0)
    {
        --coins;
        go = Instantiate(coin,transform.position+ new
Vector3(0f,0.5f,0f),Quaternion.identity,transform) as GameObject;
        rb = go.GetComponent<Rigidbody2D>();
        rb.AddRelativeForce(new Vector2( Random.Range(-1f,1f)*250f,250f
));
        go = null;
        rb = null;
        yield return new WaitForSeconds(0.25f);
    }
}

// We call this function from the FreeParallax component
// using a SendMessage method, when the Chest need to be redrawn as a new
one
void CloseBox()
{
    wasOpen = false;
    GetComponent<SpriteRenderer>().Sprite = ClosedChest;
}
}
```

Let's now create the coin by dragging the coin `Sprite GameObject` in the scene.

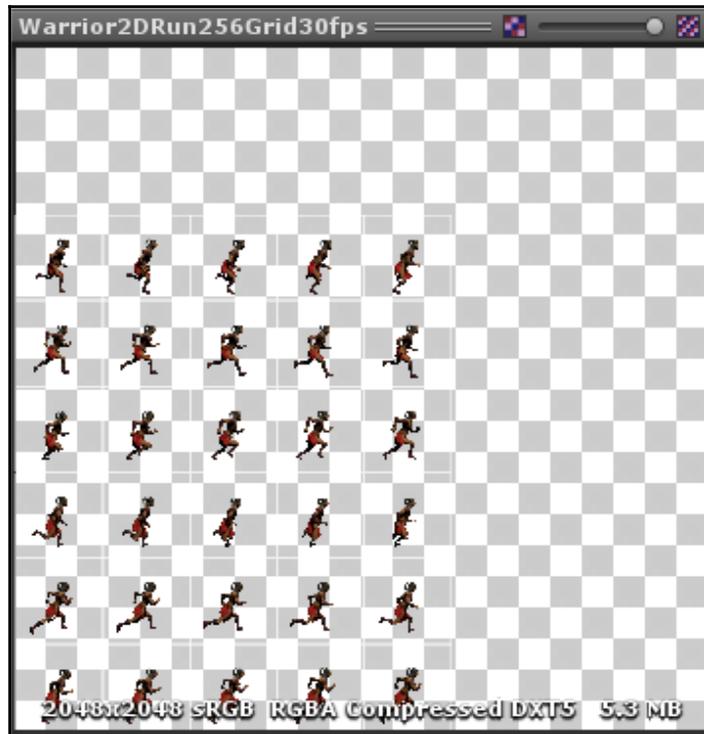
This coin is a Sprite sheet made of three frames of animation and you will find the animation for it ready otherwise you would have dragged the frames of the Sprite, like we did for the player character, into the scene, to be prompted for an animation file. Drag the already-made coin animator controller into the slot of the Animator component on the board of the `GameObject`. Add a `Rigidbody 2D` component to it, then add the `collectable2D` script to the object. And, finally, we will add a method function to our `2DPlatformerCustomUserControl` that our `collectable2D` class will call by sending a message to the player object to increase the score. You can place it right after its `Update()` function:

```
// Method called by the collision between player and the coin
take place
// we add the passed quantity to the score and
// update the scoreLabel Text component string
void AddCoins(int quantity)
{
    score += quantity;
    scoreLabel.GetComponent<Text>().text = "Score: " + score;
}
```

Press Play and test the game. After it has run for a bit, on the right you should see a gold chest appear. When the hero touches it, the chest should open and spawn coins all around.

2D animation basics

Sprite animations are managed by Unity 2D system in two different ways. You can create an animation from different Sprites, or use a single Sprite sheet with all the frames of the animation in it:



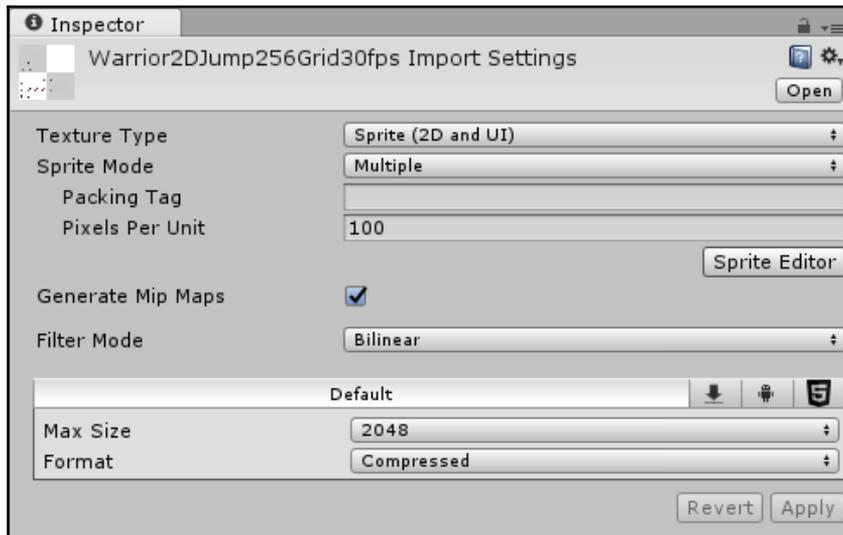
The Sprite sheet of a running animation in the Inspector preview

Importing animated Sprites

Unity made this task really easy and seamless. If the power of this automatic tool is not enough, you can still go for different Sprite cut areas with the Sprite Editor, or opt for a different packing with the Sprite Packer tool.

Sprites can be imported as a single Sprite or as multiple Sprites when setting the **Texture Type** of an image as **Sprite (2D and UI)**. Unity can treat this image as a collection of frames of an animation when importing an image. To allow this, we will change its **Sprite Mode** to **Multiple** in the **Inspector**.

The Sprite will now have multiple frames of animation generated by splicing the Sprite sheet:

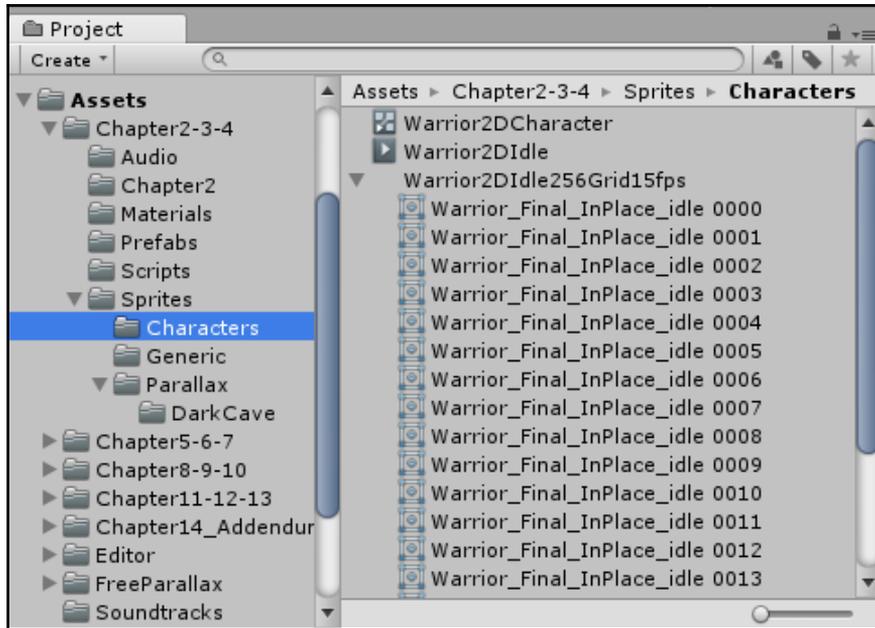


The Texture Import Settings panel, set up for multiple Sprite sheet



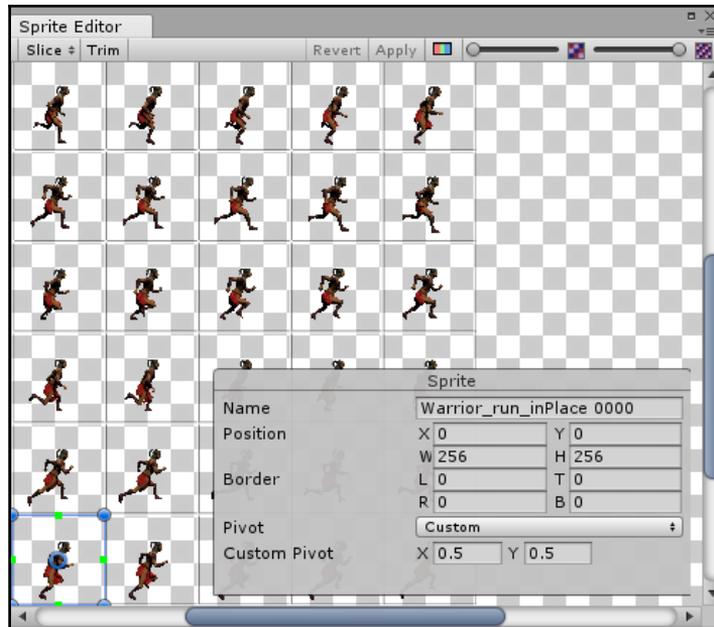
It is enough to leave **2048** for the **Max Size** for the furnished Sprite sheets, as this is the maximum size of the provided images.

We want to remove the **Generate Mip Maps** option for the sprite sheet containing the animations for the character, as they are not needed in a 2D game. Click on the **Apply** button and note how the file changed in the **Project** view. Now the main icon has a small arrow; you can click on it to expand it and see all the frames of the animation as in the following screenshot:



To adjust the slicing of the Sprites manually, you can always open the Sprite Editor from the **Inspector** by reselecting the asset in the **Project** panel later.

This is usually not needed and is not recommended in our case, but could save some video memory in the event that your game does not perform well on mobile platforms:



We have kept the animations separated into different Sprite sheets for simplicity and because it is easier to manage them this way. If we drag the imported Sprite sheet directly into the scene, a new `GameObject` with a `Sprite` component attached will be created automatically by Unity but will have just the first frame of the animation sheet.

To create the animation clip for this Sprite sheet, simply expand the `Sprite` object in the **Project** view, by clicking the small gray arrow to the side of it, to see all the frames that were created automatically by Unity. Select all of them and drag them instead into the scene; you will be prompted for a filename to create the animation file.

Name this file in a proper way, such as `Warrior2DRun`, so that you can recognize it when you have to search for this file in the **Project** view. Repeat the same previous steps for the other three furnished animation Sprite sheets: walk, idle, and jump.

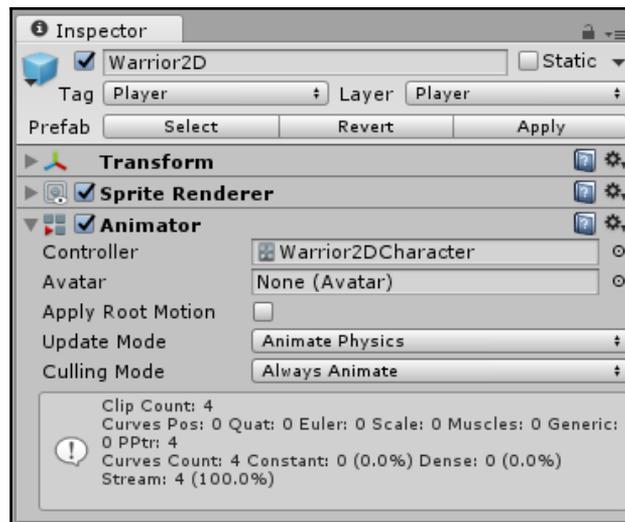
Finally, drag any one of them, for example, the run animation (the main Sprite sheet, not the frames) into the **Hierarchy** and rename it `Warrior2D`. Now we are ready to create our very first `Animator` state-machine to use with the `Animator` component.

The Animator component

The Animator component is needed to sequence and play all the animations in Unity, not only 2D animations, but also 3D objects and rigged skinned mesh animations. This component is needed for animation to be played and blend inside a visual state-machine, the **Animator** window.

Let's add this component to the `Warrior2D` `GameObject` we just added to the scene and let's see how to set it up.

The Animator component has two slots, one for the Avatar and one for holding the **Animator Controller** `Warrior2DCharacter`:

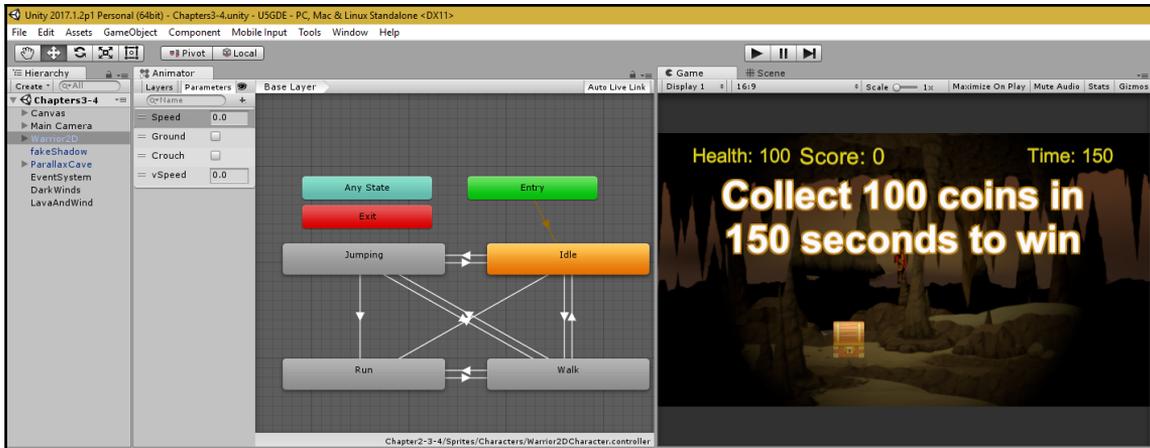


We need to assign to the Animator component the controller for our character; we can use the ready-made existing one, or create our own Animator controller file, then drag it in the slot.

Let's open the **Animator** window from the top menu: **Window | Animator**. Because the `Warrior2D` `GameObject` was the last one we selected with an Animator component attached, we will see the diagram related to that Animator controller in the **Animator** window.



It is better to dock the **Animator** window in such a way that both the **Scene** view or **Game** view and the **Animator** window are visible. In this way, we can see what happens while the game runs as the Animator states are changing. An editor layout example is provided in the following screenshot.



The Texture Import Settings panel, set up for multiple Sprite sheet

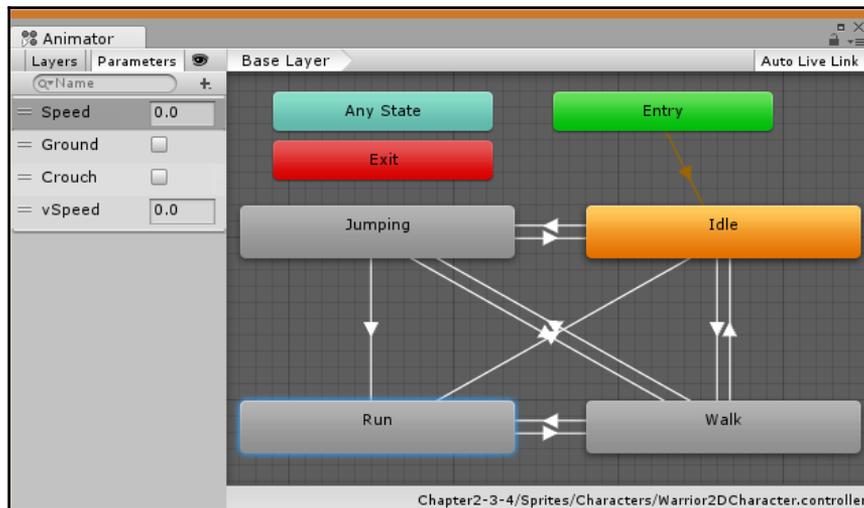
Animator state-machine editor

A character usually has many different animations for different actions to perform in a game. It may walk and raise its arms with a sword to hit an enemy. Controlling when these animations are played back is potentially quite a complicated scripting task; a better technique is to utilize Unity's Animator state-machine, which is the view we see in the **Animator** window.

The **Animator** window is made up of the following things:

- The Animation **Layers** menu tab (at the top-left corner of the window)
- The **Parameters** menu tab (at the side of the Animation Layer)
- The state-machine scheme itself

The final scheme for our hero character will look like the following screenshot:



The values of existing parameters in this Animator will be set from script and will drive the behavior of the state-machine itself using transitions and conditions. We will look at Animator parameters and how to use them in more detail in [Chapter 5, *Character Animation with Unity*](#).

Transitions between animation states

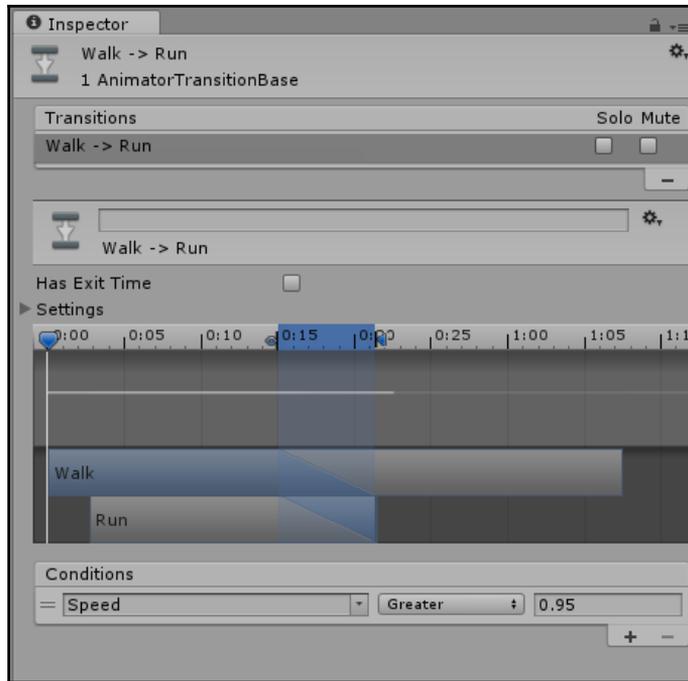
The lines you see in the diagram connecting the animation states are the transitions between them; they decide the way the transition from a state to another is performed and work by setting a condition that depends on the value of a given parameter. When this condition is met, the current state changes to the next one, until another condition is met.

Conditions and parameters

Conditions are the way to instruct the state-machine on when the current state should change to another.

We will change from the Idle state to the **Walk** state when the **Speed** of the character is higher than zero, hence when the user begins an arrow key press. To manage all the cases, we will connect all the states with two transitions, one for one way, and one for the way back, like we saw in the previous screenshot.

When a connecting arrow (transition) in the **Animator** window is selected, the **Inspector** will show its details:

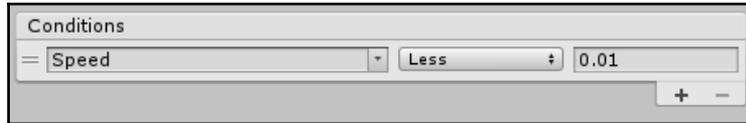


The Inspector showing the Walk->Run transition details

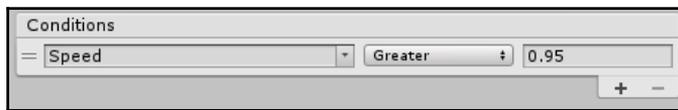
Let's set all these connections step by step: first, select the *Idle to Walk* transition arrow then, in the **Inspector**, click on the plus button to add a condition and set it to the **Speed** parameter we created as the conditional value to check. Set **Greater** and type 0.01 in the value field:



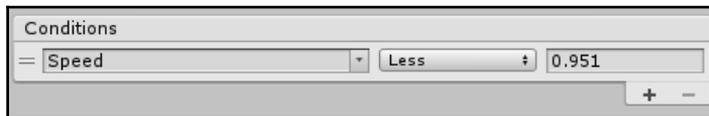
Repeat the step for the *Walk to Idle* transition add the condition for the **Speed** parameter and set it to **Less** than 0.01:



Set the **Speed** parameter for the *Run to Idle* transition and set **Less** than 0.01 as well. Then, for the *Walk to Run* transition, we set the **Speed** condition to **Greater** than 0.95:

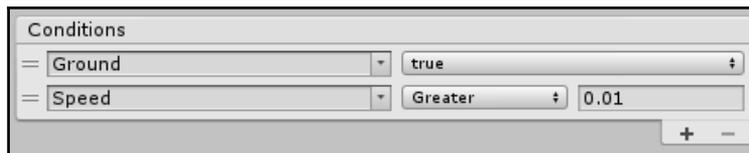


And **Less** than 0.951 for the way back (**Run to Walk** transition) **Speed** condition:



Now we will set the *Idle to Jumping* and the *Jumping to Idle* transitions. Set a condition for the **Ground** parameter to true for the former and to false for the latter.

For the *Walk to Jumping* transition as well, we will set a condition for the **Ground** parameter to be false, while for the former we will set only the **Speed** parameter to **Less** than 0.01 like we did for the *Walk to Idle* transition, while for *Jumping to Walk*, we will set two conditions: the **Ground** must be **true** and the **Speed** **Greater** than 0.01 as shown in the following screenshot:



Finally, for the *Jumping to Run* transition, we will set **Ground** to true and **Speed Greater than 0.95** conditions as we did for the *Jumping to Walk* transition. We will not set a **Run -> Jumping** transition. Now the character graphics and the Animator are ready to be controlled by the scripts, which will also take care of player collisions, generic game logic and the UI element update. Press play and verify the character behaves according to the inputs given and that the animation transition conditions are met.

Final words

Even though we have seen the Animator component, transition conditions, and the parameters needed to animate our 2D hero and the spinning collectable coin, which are based on Sprites, we will look at the animation system in more detail, including the Layers and Masks, Sub Animation States, Blend Trees, Transition blending, and complex Conditions in [Chapter 5, *Character Animation with Unity*](#), where it will play an important role with logic-driven animation blending functionalities.

Summary

We have seen how Unity manages 2D projects, the orthogonal camera, the lighting, how to implement parallax scrolling backgrounds, and how to import animations from Sprite sheets. We modified the FreeParallax component so as to be able to spawn random collectables and prepare them for player collection. We learned about the Sprite sort order and took a first look at the Animator Controller. In the next chapter, we will go further with our 2D game and create the player controller and collectables, and learn more about C# scripting.

4

Player Controller and Further Scripting

In this chapter, we'll expand the 2D game with our own player controller that will implement our parallax scrolling mechanism. Take a look at the construction of the player character that you have already added to the scene. This object is an example of a 2D side-scrolling player character. How does its combination of objects and components achieve this effect?

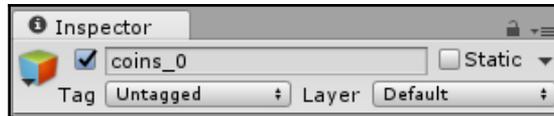
We will take a look under the hood of this prefab while looking at how all the components work together to create our player character. As we have already added our prefab to the game scene, it would be all too easy to continue with the development and accept that this object just works. Whenever you are implementing any externally-created assets, you should make sure that you always try to understand how they work, even if you cannot recreate them yourself just yet. Otherwise, if anything needs adjusting or goes wrong, you'll be in the dark, which can be detrimental when asking others for help.

With this in mind, we'll take a look at the following topics in order to help you understand how combining just a few objects and components can create a fully fledged character:

- Main features of the **Inspector**
- **Dot Syntax** and a quick scripting course
- Anatomy of a 2D character
- The `PlatformerCharacter2D` controller component
- Exploring OOP as we extend the `Platform2DUserControl` class
- Modifying classes for player movement and collision detection

Working with the Inspector

As we are dissecting an object's details in the **Inspector**, let's begin by looking at the features of the **Inspector** that are common to GameObjects in the active scene and prefabs in the project. At the top of the **Inspector**, you will see the name of the object that you have currently selected, along with a GameObject or prefab icon (a red, green, and blue-sided cube or a light blue cube, respectively) with an arrow and a checkbox to allow you to temporarily deactivate the object, for example, when first creating a GameObject (not from an existing prefab):



The top of the Inspector

Here, you can see the red, green, and blue box icon, which represents a standard GameObject. Clicking on the small black arrow will allow you to choose a Gizmo icon to be displayed for this kind of object in the **Scene** view.



The name box of this part of the **Inspector** can be used to rename an object simply by clicking and typing, as an alternative to renaming in the **Hierarchy** panel as we have done so far.

To the right of the name field is the **Static** checkbox. Checking this box on a GameObject tells Unity that a particular object in your scene will not be moving during the game. There are a number of reasons that the user might want something in their game to be static, and clicking the down arrow will reveal further options, such as the **Lightmap Static** option, where we tell Unity that non-moving, so called static, GameObjects will have their lighting/shadows baked into textures in order to save performance when the game runs. We will make use of the **Lighting** tools later in the book to improve the aesthetics of our 3D scenes. This tool allows us to bake the lighting of a scene into its textures to add depth and realism. We will also look at the Navigation system that Unity offers to manage AI path finding. Following the icon, the active checkbox, and the name, you will see the **Tag** and **Layer** settings.

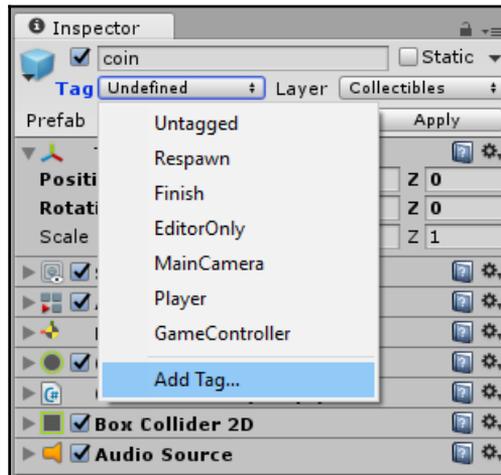
Tags

Tags are simply keywords that can be assigned to a `GameObject`. By assigning a tag, you can use the chosen word or phrase to address the `GameObject` (or objects) to which it is assigned within your game scripting (one tag can be used many times). You could think of tags as keywords that are used to represent the type of objects in your game, for example: `Enemy`, `Player`, `Projectile`, `Weapon`, `Medikit`, `Walls`, and `Walkable`, could be valid names to specify types of objects and collect them later by code with `FindWithTag` and `FindGameObjectsWithTag`, which respectively return one or a collection of objects.

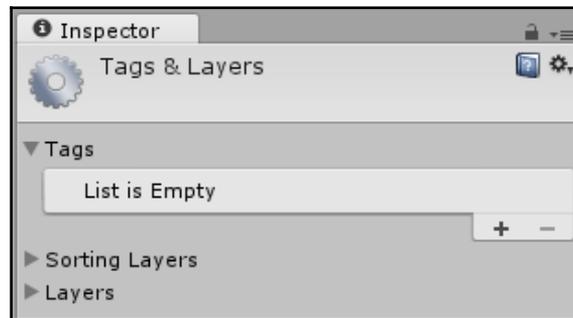
Adding a tag to a `GameObject` is a two-step procedure:

1. First your new tag must be added to a list within the **Tags & Layers Manager**
2. Then the created tag is applied to your chosen object

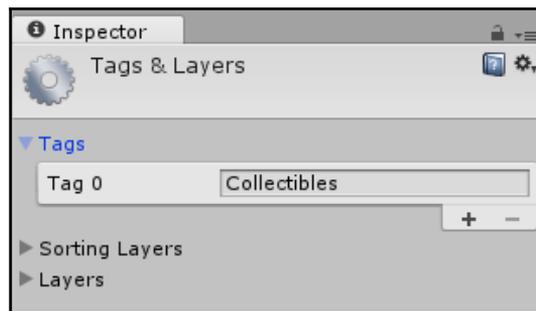
To help you get used to using tags, let's make and apply our first tag to the coin `GameObject`. With this object selected in the **Hierarchy** panel, click the **Tag** drop-down menu where it currently says **Untagged**. You will now see a list of existing tags. You can either choose one of these or make a new one. Select **Add Tag...** at the bottom in order to add a tag to the list in the **Tag Manager**:



The **Inspector** panel will now display the **Tag Manager**:



The **Tags & Layers** settings shows **Tags**, **Sorting Layers**, and **Layers** lists. To add a new tag, you'll need to click the plus button below the list. Once you have done this, you will be able to type in the space to the right of **Tag 0**:

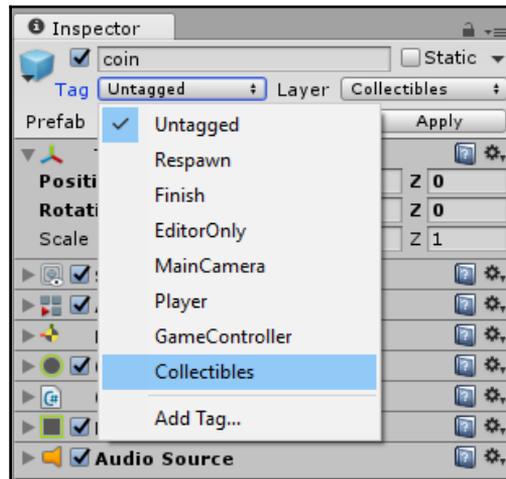


Type in the name `Collectibles` (tags may be named however you please), and press *Enter* to confirm.



Note that once you have named a Tag, you cannot rename it. To remove a Tag, click it and then click the minus (-) button at the bottom-right of the list. Read more at <https://docs.unity3d.com/Manual/class-TagManager.html>.

You have added a tag to the list. However, it has not yet been assigned to the **Coin** object. Therefore, select this object in the **Hierarchy** panel, and then click **Untagged** next to **Tag** at the top of the **Inspector**. You will now see your newly-created tag, called **Collectibles**, at the bottom of your tag list. New tags are always present after the default tags that are common to all new projects. To apply it, simply select it from the drop-down menu, as shown in the following screenshot:



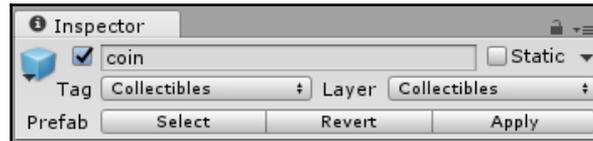
Layers

Layers are an additional way of grouping GameObjects in order to apply specific rules to them. These groups are mostly used to apply rendering culling on certain objects hence to void for example, a camera from rendering a group or more groups of objects, or a realtime light to void cast its light on a group or more groups of objects. However, they can also be used with a physics technique called **Raycasting** in order to selectively ignore certain objects. We will look into Raycasting in *Chapter 7, Interactions, Collisions, and Pathfinding*, when we will look into game *Interactions*. Another common use of layers is for physics where you place objects on the same layer that should interact with one another, effectively creating collision layers with objects that should collide on the same layer.

By placing objects on a layer, for example, they can be removed from a light's **Culling Mask** parameter, which would exclude them from being affected by the light. Layers are added in the same way you add tags, they are accessible from the tag manager too.

Prefabs and the Inspector

If the active `GameObject` you select from the **Hierarchy** panel originates from a prefab, then you will be presented with some additional settings, as shown in the following screenshot:



A prefab is a data structure saved into a `.prefab` file for storing `GameObject` properties. Beneath the **Tag** and **Layer** fields, you can see three additional buttons for interacting with the object's originating **Prefab**:

- **Select:** This simply locates and highlights the prefab the object belongs to in the **Project** panel.
- **Revert:** This reverts any settings for components in the selected object back to the settings used in the prefab in the **Project** panel.
- **Apply:** This changes the settings of the prefab to those used in the currently selected instance of that prefab. This will update any other instances or *Clones* of this prefab wherever they exist in the scene.

Now that you are aware of the additional settings available on the **Inspector**, let's start using it to inspect our player character.

Scripting with Unity

Scripting is one of the most crucial elements in becoming a game developer. While Unity is fantastic at allowing you to create games with minimal knowledge of game engine source code, you will still need to understand how to write code that instructs the Unity Engine. Code written for use with Unity to draw upon a series of ready-built classes, which you should think of as libraries of instructions or behaviors. By writing scripts, you will create your own classes by drawing upon commands in the existing Unity Engine.

In this book, you will be introduced to the basics of C# (pronounced *C-Sharp*) scripting and it is highly recommended that you read the *Unity Scripting Reference* alongside it. This is available as part of your Unity installation in the `documentation` subfolder, and also online at <https://docs.unity3d.com/ScriptReference/>.

Problems encountered while scripting can often be solved with reference to this, and if that doesn't work, then you can still ask a question on Unity answers: <http://answers.unity3d.com>. Or you can look for help on the Unity forums: <http://forum.unity3d.com>.

When writing a new script or using an existing one, the script will become active only when attached to a `GameObject` in the current scene, though scripts attached to objects can call static (global) functions in scripts not attached to objects. By attaching a script to a `GameObject`, it becomes a component of that object, and the behavior written in the script can apply to that object. However, scripts are not restricted to calling (the scripting term for activating or running) a behavior on the `GameObject` they belong to, as other objects can be referenced by a name or a tag and can also be given instructions.

In order to get a better understanding of the scripts we are about to examine, let's take a look at some core programming principles.

Statements

Statements are instructions written in a script. Although statements may be loose inside a script, you should try and ensure that they are contained within a function to give you more control over when they are called. All statements must be terminated (stated as finished) with a semicolon as follows:

```
speed = 5;
```

This tells the script to terminate the execution of that line of code.

Variables

Variables are simply containers for information. Variables may be named anything you want, provided that:

- The variable name does not conflict with an existing word in the Unity Engine API namespace
- The variable name contains only alphanumeric characters and underscores and does not begin with a number

For example, the word *transform* already exists to represent the Transform component of a `GameObject`, so naming a variable or function with that word would cause a conflict.

Variables may contain text, numbers, and references to objects, assets, or components. Here is an example of a variable declaration:

```
float speed = 9.0f;
```

Our `speed` variable is then set (given a value) using a single equals symbol and is terminated like any other statement with a semicolon.

Variable data types

You should note that usually in C# the data type, in this case the word `float`, prefixes the variable instead of the word `var` and therefore explicitly applies a data type to the new variable.

By specifying the data type for any variable we declare, we are able to make our scripting more efficient, as the game engine does not need to decide upon an appropriate type for the variable it is reading. Here are a few common data types you will encounter when starting scripting with Unity (case-sensitive differences in language are noted):

- `string`: A combination of text and numbers.
- `int`: Short for integer, meaning a whole number with no decimals.
- `float`: A floating point or a decimal-placed numeric value.
- `bool`: A true or false value, often used as a switch.
- `Vector3`: A set of XYZ values, a three-dimensional vector; technically this vector is made up of three float values. Data types are often made up of other data types in this way.

Using variables

After declaring variables, the information they contain may then be retrieved or set, simply by using the variable name. For example, if we were trying to set the `speed` variable, then we would simply say:

```
speed = 20;
```

We can also query or use the value of a variable in parts of our script. For example, if we wished to store a value that was half of the current `speed` variable, then we could establish a new variable, as shown in the following example:

```
float speed = 9.0f;
float halfSpeed;
halfSpeed = speed*0.5f;
```

Also, notice that where the `halfSpeed` variable is declared, it is not set to a value. This is because a value is given to it in the command following it by dividing the existing `speed` variable's value by two. Note here that we could have set the value for the variable when establishing it for efficiency, in which case it would have looked like the following code snippet:

```
float speed = 9.0f;
float halfSpeed = speed/2.0f;
```

Public and private variables

Variables can be `public` or `private` members of a script, which means they can be exposed to external classes or remain for the `private` use of this class only.



Only public variables will automatically show up as parameters of the script when it is viewed as a component in the **Inspector**.

Typically, these public variables should only be used if adjustment through the **Inspector** is necessary, because the values of variables will be locked to what is written in the **Inspector**, and not what is written in the script. Therefore, if a variable's value is only going to be assigned by the same script, then ideally it should be `private`.

Public variable conflicts

It is worth noting that, if you establish a variable as public, then its value is controlled by the value written into the **Inspector**. If you rewrite your script later, changing its declared value within the script itself, then you should ensure that the value seen in the **Inspector** is not overriding your new value. You may wish to reconsider keeping this value public, or simply update the **Inspector** declared value to avoid this conflict. If you wish to have a public variable in order to let other scripts read/assign its value, you can hide it from appearing in the **Inspector** by not serializing it with the `[HideInInspector]` metadata statement.



This means placing the following line before the variable when establishing it:

```
[HideInInspector] public float runSpeed = 8.0f
```

Will make the `runSpeed` float variable public, without exposing it in the **Inspector** for quick editing.

Declaring variables

The following section shows how you can declare public and private variables in C#. Whether within or outside a function, a variable in C# is private. It must have a `public` prefix to make it visible/adjustable in the **Inspector**:

```
public float gravity = 20.5f; // this is public
private float gravity = 20.5f; // this is private
float gravity = 20.5f; // this is also private in C
```

Full code example

In the following screenshot, we see a script applied to a `GameObject`. There are two private variables, `m_MaxSpeed` and `m_JumpForce`, in the `PlatformerCharacter2D` class, with the `[SerializeField]` directive, which will serialize them as if they were public variables. A public variable, `m_Crouch`, and a private variable, `m_Grounded`, are declared. The private variable is not given a value when declared but is assigned one in the `Update()` function, where it is given the value of the `moveSpeed` public variable, multiplied by 2.

The value of this private variable is then used in a new `Vector3` variable as the z-coordinate, which in turn is used to set the velocity value of a `Rigidbody` object:



Be aware that any value adjusted in the **Inspector** will override the original value given to a variable within the script. It will not rewrite the value stated in the script, but simply replaces it when the game runs. You can also revert to the values declared in the script by clicking the cog icon to the right of the component and choosing **Reset** from the drop-down menu that appears.

Functions

Functions, or methods as they are also known, may be described as sets of instructions that can be called at a specific point in the game's runtime. A script may contain many functions and each one may call any function within the same script or other external scripts. In Unity scripting, there are many built-in functions ready-made to carry out your commands at predefined points in time or as a result of user input. You may also write your own functions and call them when you need to carry out specific sets of instructions.

All functions are declared by the `void` prefix in C#. This is followed by the function name and a set of brackets into which the developer may pass additional arguments if necessary. A function's span then ranges from the opening curly bracket, `{`, and the closing bracket, `}`.

Let's look at some examples of the existing functions that you may use.

Update()

A new C# file created in Unity already contains the `Start ()` and the `Update ()` methods, as we saw in our earlier exercise. When you open it for editing for the first time, the class body will look like this:

```
// Use this for initialization
void Start () {
}
// Update is called once per frame
void Update () {
}
```

Games run at a certain number of **Frames Per Second (FPS)**, and the `Update ()` function is called when each frame of the game is rendered. As a result, it is mostly used for any commands that must be carried out constantly or for detecting changes in the game world that occur in real time, such as input commands, key presses, or mouse clicks. As emphasized by the `ThirdPersonCustomCharacter` class, when dealing with physics-related commands, the alternative function, `FixedUpdate ()`, should be used instead, as it is a function that syncs with the physics engine, whereas `Update ()` itself can be variable depending on the frame rate.

OnMouseDown()

As an example of a function that is called at a specific time, `OnMouseDown ()` is only called when the player's mouse clicks on a `GameObject`'s collider or on a GUI element in the game.

This is most often used for mouse-controlled games or detecting clicks in menus. Here is an example of a basic `OnMouseDown ()` function that, when attached to a `GameObject`, will quit the game when the object is clicked:

```
void OnMouseDown(){
    Application.Quit();
}
```

There are also a variety of similar mouse functions such as `OnMouseUp ()` and `OnMouseEnter ()`. Search for `Mouse` in the script reference to see what you can do.

Writing custom functions

In creating your own functions, you will be able to specify a set of instructions that can be called from anywhere within the scripts you are writing. If we were setting up some instructions to move an object to a specified position in the game world, then we could write a custom function to carry out the necessary instructions so that we could call this function from other functions within a script.

Return type

Functions that you create to perform a specific task may not simply carry out instructions, but instead return information. For example, if we needed to create a function that carried out arithmetic and returned it, we could write something like the following code snippet:

```
float currentAmount;

float DoSums () {
    float amount = 5.0f + 55.8f;
    return amount;
}

void Update () {
    if (Input.GetButtonUp ("Jump")) {
        currentAmount = DoSums ();
    }
}
```

Here we have a custom function called `DoSums ()` that we have given a `return` type, like a data type in a variable declaration, which means it will return data of the `float` type. In order to get this function to return a value, we have given it this type and also used the `return` command, which means that when this function is used in another part of the script, its resultant value is returned, much like using a variable but with more power, as its value may change depending upon what occurs within the function.

In the example given, a sum is calculated and assigned to the `currentAmount` variable when the player releases the *Spacebar* (the default for `Input "Jump"`). So, in this simple example, `currentAmount` will be set to `60.8`, as this is the returned value of this function.

Arguments

Arguments are variables within a function that allow you to send differing information to the function in order to alter what it does, this means you needn't repeat the same function with differing commands.

For example, on falling into a trap, a player character may need to be moved if it has died and is returning to the start of a level. Rather than writing the player relocation instructions into every part of the game that causes the player to die, the necessary instructions can be contained within a single function that is called many times. The advantage here is that we can modify the behavior of this function simply by providing part of its command as an argument.

Typically, arguments are declared in the function in the following manner:

```
void FunctionName(DataType argument1, DataType argument2){
    //commands here that may use the value of arguments
}
```

Within the parentheses you simply declare each new argument, separating them with commas, in a similar style to declaring variables, with the name of the variable and the datatype.

Declaring a custom function

To illustrate this further, you could create a function called `SpawnEnemy()`, which might look like the following code snippet:

```
void SpawnEnemy(GameObject enemy, Transform spawnTrans, string enemyName){
    GameObject newEnemy = Instantiate(enemy, spawnTrans.position,
    spawnTrans.rotation) as
    GameObject;
    newEnemy.name = enemyName;
}
```

Calling a custom function

In this example, in order to call the function, we would need to write the name of the function and then, within its parentheses, three pieces of data that correspond to the arguments in the declaration of the function shown earlier, for example:

```
public GameObject enemyPrefab1;
public Transform spawn1;
```

```
public Transform spawn2;
int enemyCount = 0;
void Update(){
    if(enemyCount < 1){
        SpawnEnemy(enemyPrefab1, spawn1, "Ogre");
        enemyCount++;
    }
}
```

In this example, the `SpawnEnemy()` function (often referred to as a method of the class) is called and its arguments are given the following values:

- `enemy`: This argument is of the `GameObject` type, and we are feeding it whatever `GameObject` is assigned to a public variable called `enemyPrefab`. It is likely that we would assign this in the **Inspector**; because it is public we can simply drag and drop a prefab onto it.
- `spawnTrans`: This argument has `Transform` as its data type, and it uses this information to declare a position and rotation for the instantiate command (see the declaration in the earlier section). We feed it the value of `spawn1`, a public member variable, and we'll also likely drag and drop the transform of an empty `GameObject`, in order to mark a spawn position.
- `enemyName`: We simply give this argument the `Ogre` value, as it is a `string` data type.

These function arguments rely on being written in the correct order. For example, here we got them in the wrong order when calling the function:

```
SpawnEnemy("Ogre", enemyPrefab1, spawn1);
```

Here, we would get an error in our script telling us that this line of code has some invalid arguments. This simply means that we have offered data to the function that does not match what it expects, as in the declaration it is expecting (pseudo code shown) the following:

```
SpawnEnemy(enemy - a GameObject, spawnTrans - a transform, enemyName - a
"string" of information);
```

But by getting the order wrong, the script compilation will halt at the first problem: we would be feeding a `string` where a `GameObject` is expected.

Here is the script in full:

```
using UnityEngine;
using System.Collections;
public class test : MonoBehaviour {
    public GameObject enemyPrefab1;
    public Transform spawn1;
    public Transform spawn2;
    int enemyCount = 0;
    void Update(){
        if(enemyCount < 1)
        {
            SpawnEnemy(enemyPrefab1, spawn1, "Ogre");
            enemyCount++;
        }
    }
    void SpawnEnemy(GameObject enemy, Transform spawnTrans, string enemyName){
        GameObject newEnemy = Instantiate(enemy, spawnTrans.position,
        spawnTrans.rotation) as GameObject;
        newEnemy.name = enemyName;
    }
}
```

If else statements

An **if statement** is used in scripting to check for conditions. If its conditions are met, then the `if` statement will carry out a set of nested instructions. If they are not, then it can default to a set of instructions called `else`. In the following examples, the `if` statement is checking whether the Boolean grounded variable is set to `true`:

```
bool grounded = false;
float speed;
void Update(){
    if(grounded==true){
        speed = 5.0f;
    }
}
```

If the condition in the `if` statement's brackets is met, that is, if the `grounded` variable becomes `true`, as a result of this being assigned in this script by another function, for example, then the `speed` variable will be set to 5. Otherwise it will not be given a value.

As we are checking whether a boolean variable is true here, we could simply write the following code:

```
if(grounded){}
```

This means exactly the same as writing `if(grounded == true)`. To check whether this is false, we could write the following code:

```
if(!grounded){}
```

The use of an exclamation mark simply serves to say *not*, so we would be saying, *if grounded is not true*.



Note that when setting a variable, a single equals symbol, `=`, is used, but when checking the status of a variable, we use two, `==`. This is known as a comparative equals—we are comparing the information on either side of the equals symbols.

If you want to say NOT equal to, you may replace the first equals symbol with an exclamation mark, like so:

```
if(grounded != true){}
```

If we wanted to set up a fallback condition, then we could add an `else` statement after the `if`, which is our way of saying that if these conditions are not met, then do something else:

```
if(grounded==true){
    speed = 5.0f;
}else{
    speed = 0.0f;
}
```

So, unless `grounded` is `true`, `speed` will equal 0.

To build additional potential outcomes for checking conditions, we can use an `else if` before the `else` fallback. If we are checking values, then we could write the following code snippet:

```
if(speed >= 6){
    //do something
}
else if(speed >= 3){
    //do something different
}
else{
    //if neither of the above are true, do this
}
```

Be sure to remember that, where I have written `//`, I am simply writing code that would appear as a **comment** (non-executed code).

Multiple conditions

We can also check for more than a single condition in one `if` statement by using two ampersand symbols, `&&`.

For example, we may want to check on the condition of two variables at once and only carry out our instructions if the variables' values are as specified. We would write the following code snippet:

```
if(speed >= 3 && grounded == true){
    //do something
}
```

If we wanted to check for one condition or another being true, then we can use two vertical line characters, `||`, in order to mean **OR**. We would write this as follows:

```
if(speed >= 3 || grounded == true){
    //do something
}
```

This means that the commands within the `if` statement will run if at least one of the conditions is met.

For loops

Often in scripting, you will need to carry out sets of instructions repetitively, until certain conditions are met. For these situations, it is often best to make use of `for` loops.

A `for` loop has three parameters within its parentheses, which are as follows:

```
for(declaration of integer variable; condition to continue; instruction to
    carry out at the end of each loop);
```

Unlike arguments in a function declaration, these are separated by a semicolon instead of a comma. For example, a simple `for` loop that adds to a counter would look like the following:

```
public int counter = 0;
void Start() {
    for(int i=0; i<=10; i++){
        counter++;
        Debug.Log(counter);
    }
}
```

The `for` loop itself begins with an integer variable simply titled `i`, which is set to `1`; the loop then continues so long as the value of `i` is less than or equal to `10`, and with each loop we increment `i` by `1`. This is done using `++`, which is the same as writing `+=1`.

As the loop's variable `i` has a starting value of `0` and a continue value that is less than or equal to `10`, the loop will run `11` times, and therefore, any commands in the loop will be carried out this many times.

In this example, we are creating a public variable called `enemyCount` that is incremented with each loop. We then use `Debug.Log()` to print the value of this variable in the **Console**. This command is useful, as it will allow you to check on the value of variables as the scripts run. The console in Unity shows its latest value in the bar at the bottom of the interface where errors and warnings in scripts, editor warnings, and `Debug.Log` information are shown. The console window itself can be accessed from the top menu **Window | Console**. It's often a good practice to have the **Console** window tabbed somewhere in the Editor so you can always have a quick look.

`for` loops like this are useful for various purposes, such as dynamically creating prefabs at the start of a game, for example, by instantiating and altering a position variable with each loop, a row or grid of objects could be instantiated.

`for` loops may also be other types; they are usually integers (`int`) when used with the `For` statement when they can be any type, if you use the `System.Collections` namespace and the `Foreach` statement, for example:

```
foreach(Rigidbody rb in this.GetComponentsInChildren<Rigidbody>() )
{
    // Do something
}
```

This kind of loop is very powerful, but slower than the former. Remember that it is always better to implement caching for calls, such as `GetComponentsInChildren<Rigidbody>()`, at the `Start()` level to avoid being called at each loop, for example:

```
using UnityEngine;
using System.Collections;

private Rigidbody selectedRigidbodies[];
void Start()
{
    selectedRigidbodies = this.GetComponentsInChildren<Rigidbody>();
}
void SetChildrenRigidbodiesKinematic(bool flag){
```

```
foreach(Rigidbody rb in selectedRigidbody)
{
    rb.isKinematic = flag;
}
}
```

In this way, when your code requires calling the `SetChildrenRigidbodyKinematic` method to set the children rigidbodies `isKinematic` property, the method execution time will be much faster, because the call has been done at the `Start()` time just once and not for each loop iteration.

Inter-script communication and Dot Syntax

In order to create games effectively, you'll often need to communicate between scripts in order to pass data around, adjust variables, and call functions in external scripts, by external here we can mean either a separate script or one attached to a different object from the given script.

Accessing other objects

Often you may be in a situation where your script is located on one object in the current scene, and you wish to communicate with a script on another object, for example, your player character may shoot an enemy and this results in a need for their health to decrease. However, each enemy has an independent script storing its own health, so a script on the player or bullet must address the script on the enemy that its health is stored within.

To do this, prior to accessing the script, you'll need to refer to the object, which can be done in various ways including using the `Find()` and `FindWithTag()` commands, or in the case of a collision, by referring to the collided-with object, or in a basic sense, by using public member variables to establish references to an object.

Find() and FindWithTag()

A computationally more expensive way to refer to an object is to use the find commands. This is not something that should be done often, so where possible avoid using these within `Update()` or other functions that run each frame. Often, this is best used to set a particular variable to an object using a `Start()` or `Awake()` function.

For example, we may set up a non-serialized or private variable (so that it is not altered by the **Inspector**) within our script to represent a particular `GameObject`, then set it when the game begins.

The `Find()` command itself expects to be given the name of a `GameObject` within its parentheses, using its hierarchical name as a string, while the `FindWithTag()` command is looking for the tag applied to an object, also written in a string that is shown as follows:

```
GameObject.FindWithTag("Collectables");
```

Here is a quick example using `Find()` to address an object in the `coin` hierarchy:

```
GameObject collectibleObj;
void Start(){
    collectibleObj = GameObject.Find("coin");
}
```

Here, the `collectibleObj` variable is assigned the `coin` `GameObject` thanks to the use of `Find()`, by addressing its name in the hierarchy.

We now have a reference to our `coin` object in the current scene, and we can refer to it using the `collectibleObj` variable. Now we can use this variable to access component scripts on that object.

The more common way to do something like this is not by using `GameObject.Find`. It is fine when used in the `Start()` method, but can lead to many errors while developing, such as the prefab name being changed in the scene, and will prevent the script from working properly. Most of the time, you want to declare `collectibleObj` as a public variable to be able to set the prefab or other object reference simply by dragging the required element in the **Inspector**.

SendMessage

A basic way of calling instructions on another object is to simply write a custom function in a script, and then use the `SendMessage()` command to call the instructions on that object.

For example, in our previous section on *Writing custom functions*, we created a function called `SpawnEnemy` that had three arguments. For the purposes of simplifying this example, let's look at a function with just one argument:

```
void Renamer(string newName){
    GameObject.name = newName;
}
```

The single argument `newName` here can of course be set when calling the function. If we were calling this from the same script, we could simply write the following:

```
Renamer ("Steve");
```

However, if this function occurred in a script external to a script we were working in, then we could use `SendMessage()` to call it. For this, we would refer to an object and then `SendMessage()`, which is shown as follows:

```
GameObject.Find("target").SendMessage("Renamer", "target_down");
```



Note here that we have not needed to name the script at all, as Unity simply looks at the object specified. In this case, an item in the current scene, called `target`, seeks a function in any of that object's attached scripts, called `Renamer`. When it finds it, it executes it, sending the `"target_down"` value to the `newName` argument of the function.

GetComponent

`GetComponent()` works slightly differently but is more flexible as it is used to address all manner of components. We'll demonstrate here how to use it to perform the same function as we just saw with `SendMessage()`.

First, let's imagine that our `Renamer()` function is inside a script called `Setter`. This is attached to an object called `target` in the current scene, and we need to address the object, then use `GetComponent()` to refer to that particular script, and finally call the function itself:

```
GameObject.Find("target").GetComponent<Setter>().Renamer("target_down");
```

Note here that regardless of language, we're using a dot to separate each action of the command, in this style:

```
FindObject.GetComponent.PerformAnAction;
```

This approach of drilling down to the specific part using dots to separate between classes and variables is an example of using what is called **Dot Syntax**.

Programming for mobile

It is also worth noting that, when coding in this style for the mobile, it is important to break down an operation such as this into two steps, for example:

```
GameObject theTarget = GameObject.Find("target");
theTarget.GetComponent<Setter>().Renamer("target_down");
```

Dot Syntax

In the previous example, we used a script technique called **Dot Syntax**. This is a term that may sound more complex than it actually is, as it simply means using a dot (or a period) to separate elements you are addressing in a hierarchical order.

Starting with the most general element or reference, you can use Dot Syntax to narrow down to the specific parameter you want to set.

For example, to set the vertical position of the `GameObject`, we would need to alter its `Y` coordinate. Therefore, we would be looking to address the position parameters of an object's `Transform` component. When adjusting the `Transform` component's `Position` property of an object on which a script is attached, we write the following:

```
transform.position = new Vector3(0f, 5.5f, 4f);
```

If addressing this component on another object, however, we would need a reference to it first, either through the creation of a public variable that has a `GameObject` assigned to it through drag and drop, or through the use of a function such as `Find()`. For example, if using a public variable, we could write the following:

```
public GameObject theTarget;
theTarget.transform.position = new Vector3(0f, 5.5f, 4f);
```

We would then simply assign the object to this public variable in the **Inspector**, making the `theTarget` variable valid for use in the script.

This could be made even more efficient if we were only using this variable to set parameters of the `Transform` component itself, or carrying out functions of the `Transform` class. By making a variable of the `Transform` type, we can then skip the reference to that component:

```
public Transform theTarget;
theTarget.position = new Vector3(0f, 5.5f, 4f);
```

This would also mean that the script itself would be more efficient programmatically as it would not need to store as much data as a reference to the entire `GameObject` class.

This means that any parameter can be referenced by simply finding the component it belongs to by using Dot Syntax. To further illustrate this, if we wanted to adjust the intensity value of a light, we could write the following parameter:

```
light.intensity = 8;
```

As illustrated earlier, if we wanted to adjust a light on an external object, we would simply use Dot Syntax to address that object first, and then use the same approach as shown earlier:

```
GameObject.Find("streetlight").GetComponent<Light>().light.intensity = 8;
```

Null reference exceptions

If a variable is left unassigned at any time in scripting, it is considered null, meaning *not set*. You may encounter this when forgetting to assign objects or prefabs in Unity. The editor will give an error stating **Null Reference Exception**. This may seem confusing at first glance but makes sense when you understand that null means not set, reference refers to a variable or parameter, and exception simply means a problem causing a part of the script to be invalid.

You will also be told which part of a particular script is causing this, and you can even safeguard against it occurring or halting a script by first checking whether a reference is not null (that is, set), for example:

```
public Transform theTarget;
if (theTarget != null) {
    theTarget.position = new Vector3(0f, 5.5f, 4f);
}
```

Here we are using `theTarget` within an `if` statement, we're querying it in a similar way to checking a Boolean variable, but here we are simply checking that it is not null.

Coroutines

When a function is being called, it runs to its completion before returning. This means that any action taking place in a function must happen within a single frame update; a function call can't be used to contain a procedural animation or a sequence of events over time. Consider the task of gradually reducing the audio volume of an `AudioSource` until it becomes completely muted:

```
public AudioSource soundtrack;
    void Fade()
    {
        for (float f = 1f; f >= 0; f -= 0.01f)
            soundtrack.volume = f;
    }
```



We can avoid typing the `{ }` brackets to contain the code executed by the loop in C derivate languages if the line executed by the loop is a single line.

As it stands, the `Fade` function will not have the effect you might expect. In order for the fading to be audible, the volume must be reduced over a sequence of frames to show the intermediate values being processed. However, the function will execute in its entirety within a single frame update. The intermediate values will never be heard and the volume of the audio will mute instantly.

It is possible to handle situations like this by adding code to the `Update` function that executes the fade on a frame-by-frame basis. However, it is often more convenient to use a coroutine for this kind of task.

A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame. A coroutine is declared like this:

```
IEnumerator Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        soundtrack.volume = f;
        yield return null;
    }
}
```

A coroutine is basically a function declared with a return type of `IEnumerator` and with the `yield return` statement included somewhere in the body. The `yield return` line is the point at which code execution will pause for being resumed at the following frame. To start a coroutine, you need to use the `StartCoroutine` function:

```
void Update() {  
    if (Input.GetKeyDown("f")) {  
        StartCoroutine("Fade");  
    }  
}
```



The `StartCoroutine` method has two overloads, which means the parameter accepts two types: the method string name and the method itself. For example: `StartCoroutine(Fade());` You can use `StopAllCoroutines()` to stop all the running coroutines of this `MonoBehaviour` class or call `StopCoroutine("Fade");` to stop the first running coroutine with that name.

Comments

In many scripts written by coders, you will notice that there are some lines written with two forward slashes prefixing them. These are simply comments written by the author of the script. These lines will not be taken into account by the compiler. It is generally a good idea, especially when starting out with scripting, to write your own comments in order to remind yourself how a script works. There are two ways to comment; single-line and multi-line:

```
// This is a single line comment  
/* This is a multiline comment and will continue to be a comment until it  
is closed by a star and another forward-slash, in other words, the reverse  
of how it began */
```

Further reading

As you continue to work with Unity, it is crucial that you get used to referring to the *Scripting Reference* documentation, which is installed with your copy of Unity and also available on the Unity website at the following address: <http://unity3d.com/support/documentation/ScriptReference/>.

You can use the scripting reference to search for the correct use of any class, function, or command in the Unity Engine.

Now that you're familiar with the basics of scripting, let's take a look at another example script for character movement.

Anatomy of a 2D character

To get an overview of the characters available in the 2D package we have imported into our project, let's take a look at how the 2D Character Controller prefab works in Unity terms.

Extending the Platformer2DUserController class

We will extend the standard `Platformer2DUserController` class and modify it for our needs, but it will still rely on the `PlatformerCharacter2D` class.

Making the player and writing the game logic

We will create our player character, similar to the `CharacterRobotBoy` prefab located in the `Standard Assets/2D/Prefabs` folder when you imported the 2D standard package.

This `GameObject` is done by a main parent object with all the important components attached and two children, which are just empty `GameObjects` that are needed by the script for to check the head and foot distance.

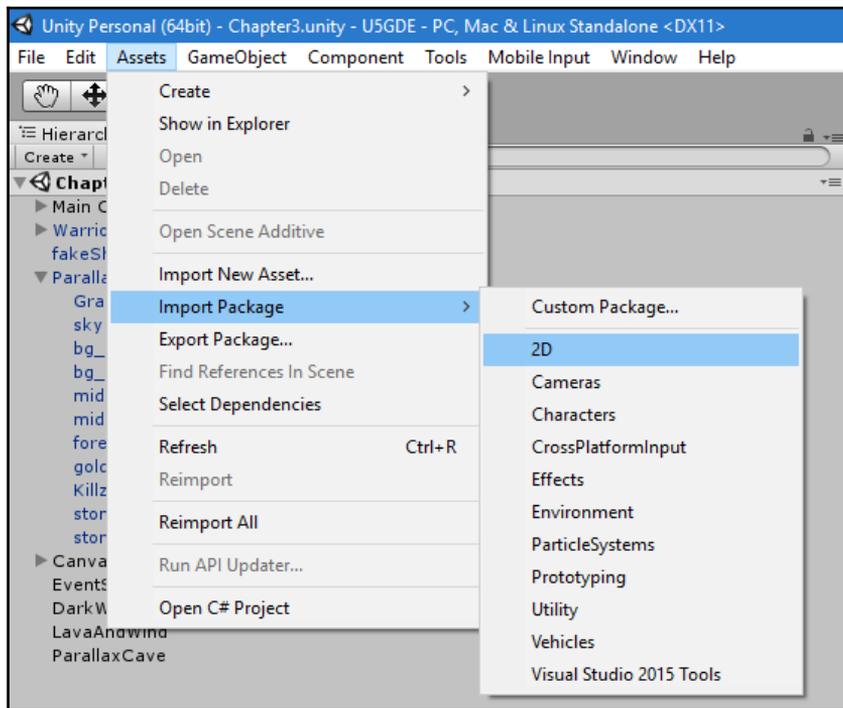
To create the player we could also have taken the `CharacterRobotBoy` prefab and deconstructed that, by changing the art for the sprite, the `Animator` controller, and the other components or, even better, by taking the prefab `Warrior2D` in the `Chapter2-3-4/Prefab` folder, but we will do it from scratch for the sake of learning.

We will make it similar to `CharacterRobotBoy` to be able to take advantage of the two companion classes of that prefab meant for driving any basic 2D character of that sort:

- `PlatformerCharacter2D`
- `Platformer2DUserController`

We will change one line of the `PlatformerCharacter2D` to avoid problems between the parallax component and the RigidBody movement of the character and we will write our own version of the `Platformer2DUserControl` class by extending it thanks to the OOP features that C# offers.

It's time to bring in the 2D Standard Asset package. We didn't import it when we first created the project, so we are doing it now, straight into the project from the main menu: **Assets | Import Package | 2D**. This package will bring in 2D examples: a ready-made character, and some important C# classes we will find very useful to implement our gameplay:



Analyzing, understanding, and modifying the component's source code

Now that we have analyzed the source code of the `FreeParallax` component a bit, we need to instruct this component on how we want to draw, and how fast we want to move the layers of the parallax background.

The component would be good as it is, although for implementing our example, we will have to modify it a little bit. We will also need to modify the `Standard Assets PlatformerCharacter2D` class and extend the `2DPlatformerUserController` class with our `2DPlatformerCustomUserController` class.

In brief, we want to:

1. Avoid the movement of the player conflicting with parallax background scrolling.
2. Allow the re-use of the collectable prefab by setting it as a not-wrapped layer element properly in the `FreeParallax` component. We will need to send a signal to our `Old Chest GameObject` before `FreeParallax` renders it again.
3. Modifying/extending the `Platformer2DUserController` to make space for additional game logic.

First of all, add the `PlatformerCharacter2D` component to the `Warrior2D GameObject`.

Then click the component script in the Inspector to find the script in the **Project** view; now double-click the script to open Visual Studio to edit the file. Go to line 82, comment this line, and after it, use this line instead:

```
//m_Rigidbody2D.velocity = new Vector2(m_MaxSpeed,
m_Rigidbody2D.velocity.y);
// We want to avoid a double movement for character and parallax we don't
move the player
m_Rigidbody2D.velocity = new Vector2(0,m_Rigidbody2D.velocity.y);
```

As you can guess by reading the comment before the new code, we are not moving the character on the x -axis because this will be performed by the background layer by moving in the opposite direction, hence, giving the effect of movement.

If we don't make this modification, the character will always move at the wrong speed compared to the parallax speed, resulting in a wrong layout, no matter what speed you do set in the **Inspector**. Save the file and go back to the editor.

Now we are going to extend the `Platformer2DUserController` class to become our `Platformer2DCustomController` class.

The best bet is to clone the `Platformer2DUserControl.cs` class and rename it to `Platformer2DCustomController.cs`, then we will make the following changes:

```
using UnityEngine;
// This is added because is needed to manage UI elements
using UnityEngine.UI;
// This is added to be able to reload the scene
using UnityEngine.SceneManagement;
// This to bring in coroutines (IEnumerator)
using UnityEngine.StandardAssets.CrossPlatformInput; using System.Collections;
// This is Unity Standard Assets namespace, where all the relevant classes
belong namespace UnityEngine.StandardAssets._2D
{ [RequireComponent(typeof(PlatformerCharacter2D))]
public class Platformer2DCustomController : Platformer2DUserControl {
    public FreeParallax parallax; //This is added for accessing the
component
    static private PlatformerCharacter2D m_Character;
```

The first difference we can notice with the original class is in the first lines, where parts of the Unity API and System.net are included. We have added `UnityEngine.UI` to access Unity UI components.

We have added `UnityEngine.SceneManagement` to be able to re-load the scene through the Unity Scene Manager API. Finally, we added `System.Collections` from the .NET framework, to be able to use coroutines.

We put our class extension as well in the `UnityEngine.StandardAssets._2D` namespace, because we are still relying on the `PlatformerCharacter2D` class that is encapsulated in that namespace too, otherwise classes won't find their companion classes.

A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace do not conflict with the same class names declared in another.

Defining a namespace

A namespace definition begins with the keyword `namespace` followed by the namespace name as follows:

```
namespace namespace_name {
    // code declarations
}
```

To call the namespace-enabled version of either a function or variable, prepend the namespace name as follows:

```
namespace_name.item_name;
```

We are declaring an additional public variable of the `FreeParallax` type to be able to access the `FreeParallax` component from this class. We are also declaring these variables to manage the game logic and the UI:

```
private int score;

// Boolean indicating when the class should receive input
private bool gameInactive;
public GameObject scoreLabel;
```

Spawning collectible items

To add the feature of collectible items, we need a small modification of the `FreeParallax` component class to be able to reset the opening status of the goldchest that will be repeated by the component.

To solve this problem, we will modify the `Position` method of the `FreeParallax` component a little.

Open the file with your favorite editor and make two changes, one at line 120 and another at line 127 right after:

```
obj.transform.position = pos;
```

Insert the following piece of code:

```
if(obj.name== "goldchest(z1)") obj.SendMessage("CloseBox");
```

Do this change at lines 120 and 127 and save the script. This will make sure that when the source element is rendered again, the chest state will be reset, by closing it up as if it was a new one, ready to be opened again.

Now we are going to take care of the `goldchest(z1) GameObject`, which will also be responsible for spawning the `Collectables` (the coins), using, for the very first time, a cool C# feature: coroutines.

It would be ugly if the 10 coins we want to spawn when the chest opens all spawn at the same time and position. To spawn them consecutively, with a small delay between each of them, we will use a coroutine. Coroutines can execute in *steps*, each one delayed by a certain amount of time, while the rest of the code execution keeps going seamlessly. This is implemented by adding the `System.Collections` API from the .NET framework to the class, so that we will be able to use the `Yield` instruction and declare a function as a coroutine with the `IEnumerator` keyword. Also, we will take advantage of physics for the coins' movement, hence we will add a force to the `Rigidbody` to pull the object after having instantiated it.

Look carefully at this C# class that will manage the `GoldChest`:

```
using UnityEngine;
using System.Collections; // This is needed to bring in coroutines support

public class GoldChest : MonoBehaviour {
    public Sprite OpenChest; // A reference of the open chest graphic
    public Sprite ClosedChest; // A reference of the closed chest
    graphic
    public GameObject coin; // The reference to the prefab of the
    collectible to spawn off
    public float spawnRange = 250f;
    private bool wasOpen; // Private boolean that determines if this
    chest was already opened before to re-generate (see FreeParallax.cs
    modifications)

    // Trigger collision detection, if the player touches the chest
    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.GameObject.tag == "Player" && !wasOpen)
        {
            GetComponent<SpriteRenderer>().sprite = OpenChest;
            GetComponent<AudioSource>().Play();
            wasOpen = true;
            // Start our first coroutine ;- )
            StartCoroutine(SpawnCoins());
        }
    }

    // Coroutine for spawning the coins out of the chest
    IEnumerator SpawnCoins()
    {
        int coins = 10;
        GameObject go;
        Rigidbody2D coinRb;
        while (coins > 0)
```

```

    {
        --coins;
        go = Instantiate(coin,transform.position+ new
            Vector3(0f,0.5f,0f),Quaternion.identity,transform) as
            GameObject;
        coinRb = go.GetComponent<Rigidbody2D>();
        coinRb.AddRelativeForce(new Vector2(
            Random.Range(-1f,1f)*spawnRange, spawnRange));
        yield return new WaitForSeconds(0.25f);
    }
}
// We call this function from the FreeParallax component
// using a SendMessage method, when the Chest need to be redrawn as
// a new one
void CloseBox() {
    wasOpen = false;
    GetComponent<SpriteRenderer>().sprite = ClosedChest;
}
}
}

```

Let's now create the coin by dragging the **coin** sprite into the scene from the **Project view**, hence creating the **GameObject**.

This coin is a *sprite sheet* made up of three frames of animation. Otherwise, you would have dragged the frames of the sprite, like we did for the player character, into the scene and would be prompted for an animation file. Drag the ready-made Animator Controller for the coin into the slot of the Animator component attached to the coin **GameObject**.

Add a **Rigidbody 2D** to it, then add the **Collectible2D** script to the object. Finally, we will add a method function to our **2DPlatformerCustomUserControl** that our **collectible2D** class will call, sending a message to the player object to increase the score when collision occurs. You can place it right after its **Update()** function:

```

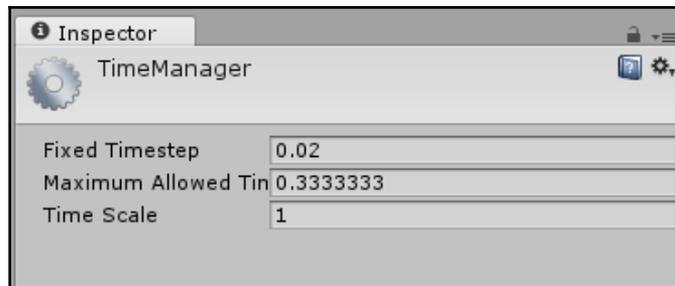
// Method called by the coin when collected
void AddCoins(int quantity)
{
    score += quantity;
    scoreLabel.GetComponent<Text>().text = "Score: " + score;
}

```

Press Play and test the game. After letting the hero player run to the right for a while, a gold chest should appear on the stage; when the hero touches it, it should open and spawn coins all around.

Adding a timer

To calculate elapsed time, we will use Unity Engine's `Time` class. This class is tied up with the engine's **TimeManager** settings accessible from the main menu, **Edit | Project Settings | Time**:



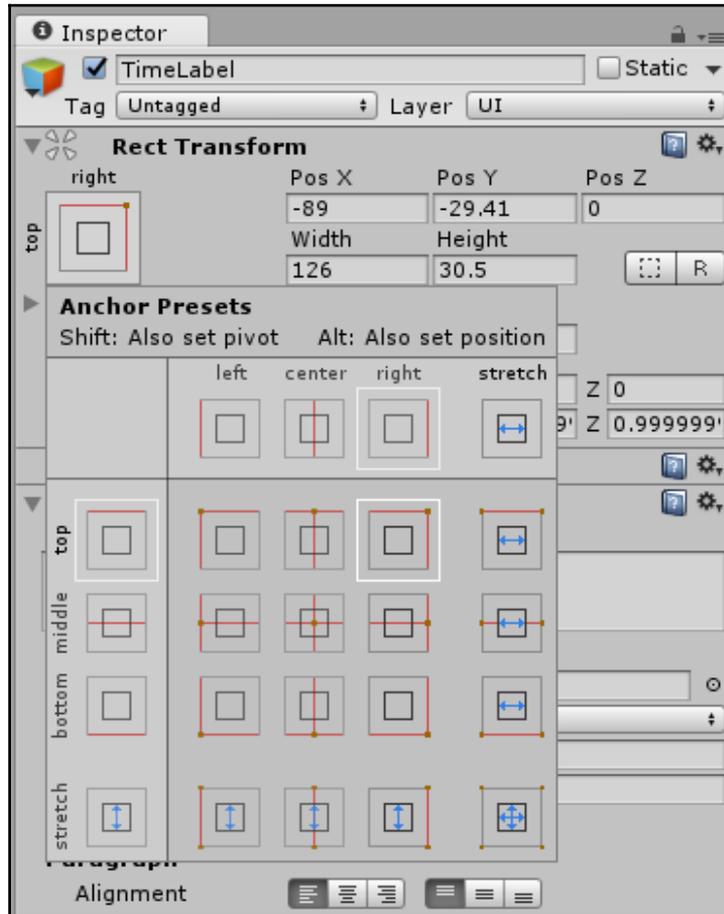
`Time.scale` is the actual scale of the time in the application; it can be set globally from the **Inspector**, or changed at runtime with scripting. A `Time.scale` of 1 means that the time is real-time, while 0.5 means half speed and 2.0 double speed. When `Time.scale` is set to 0, the engine is paused.

The `Time.time` method returns the time elapsed since the start of the application in milliseconds, while `Time.deltaTime` returns the delta (difference) since the last update.

We will sum the elapsed (delta) time every update cycle, and check against our set time limit of 150 seconds with a chunk of code added to our `Platform2DCustomController` class, so now open it for editing and add these variables after the other `int` variable's declaration:

```
private float elapsedTime;
private float timeLimit = 150f; // 150 seconds limit to finish the level
public GameObject timeLabel;
and, at the beginning of the Update() function paste this lines:
// Calculate elapsed time
elapsedTime += Time.deltaTime;
// Update time display
timeLabel.GetComponent<Text>().text = "Time:" + Mathf.RoundToInt(timeLimit
- elapsedTime);
```

The latter will access the `TimeLabel` Text component to update the value, with the actual remained time. To make it work, we will need to add a Text GameObject to our Canvas like we did earlier for the score count and name it `TimeLabel`. Align it at the top-right corner in the UI setting as illustrated here:



After saving the script, you should see the new slot in the **Inspector**. Finally, we can drag this new UI GameObject into the `TimeLabel` slot in the `Warrior2D's PlatformerCharacter2D` component.

Making things difficult with health and obstacles

To enhance the difficulty with something more dynamic, we will add rolling stones popping out from above and rolling toward the player. The player will have to jump out of the way of them to avoid losing health. This will be funny to play and cheap to realize; we will try to quickly code this nice feature in 10 simple steps:

1. Drag the stone sprite into the scene and rename it `stone`.
2. Add a Rigidbody 2D and a Circle Collider 2D component to it.
3. Store it as a prefab, then, delete the one in the scene **Hierarchy**.

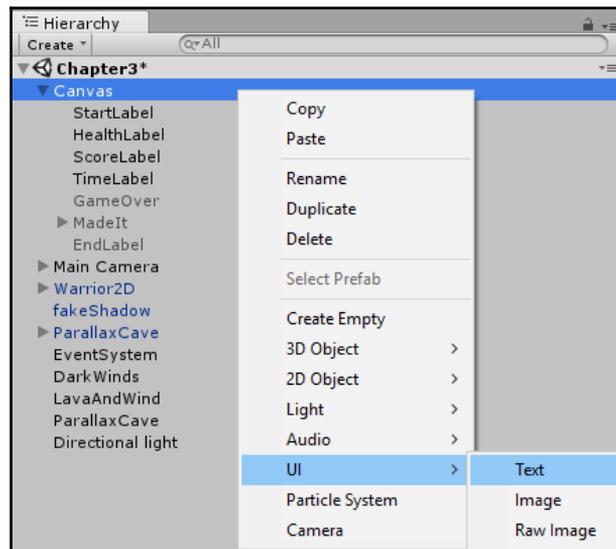
Now that we have the stone with the physics store as a prefab, we want to spawn it randomly. The approach will be a bit different than for `GoldChest`, because we want the stone to be independent of the `FreeParallax` component, but we want the latter to scroll and randomly pop out a `StoneSpawner`, a dummy `GameObject` that will spawn a stone obstacle whenever it becomes visible.

4. Create a new `Sprite GameObject` child of the `Parallax` object and call it: `StoneSpawner`.
5. Add an `Element` layer to the `Parallax` component and assign the new `GameObject`.
6. Set the positioning rules like for the `goldchest`, varying the min max X percentage to 0.7.
7. Add a new `C#` component called `StoneSpawner` to the homonym `gameObject`. Open it for editing and write this code:

```
using UnityEngine;
public class StoneSpawner : MonoBehaviour {
    // The stone prefab we want to spawn from here
    public GameObject stonePrefab;
    // This standard event method will be called whenever this
    object is rendered
    void OnBecameVisible () {
        Instantiate(stonePrefab, transform.position,
            Quaternion.identity);
    }
}
```

8. Drag the stone prefab we prepared into the `StoneSpawner` component's pertinent slot in the **Inspector**.

9. Now let's add a **Text** GameObject to the **Canvas**. We will select the **Canvas** in the hierarchy, right-click it, select **UI | Text**, and call it `TimeLabel` as seen in the following screenshot:



10. Finally, we can add the code needed to manage collision detection between the stones and the player, removing some of their health each time. The class will access the `healthLabel` Text component to update the value whenever a big enough collision (magnitude) occurs with a stone:

```
public GameObject healthLabel;
// Check collision with rolling stones
void OnCollisionEnter2D(Collision2D other)
{
    // Tweak magnitude comparison to make stones less or more evil
    if (other.relativeVelocity.magnitude > 3 && other.GameObject.layer
    == 11)
    // 11 is the id of Obstacle layer
    {
        other.GameObject.GetComponent<AudioSource>().Play();
        health -= 10;
        healthLabel.GetComponent<Text>().text = "Health: " + health;
    }
}
```

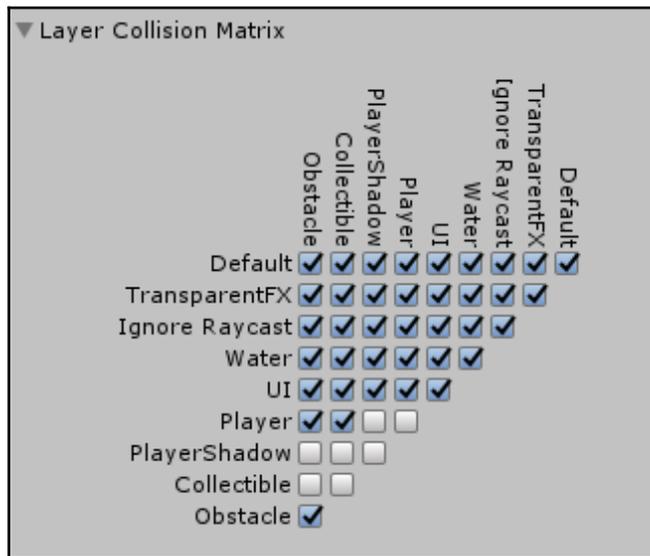


We need to have a dummy, invisible but active (disabled), Sprite component on the `StoneSpawner` `GameObject`, otherwise this event method won't be called at all. Adding a small sprite source image and setting the color to 0% alpha should do the trick.

Now you can press Play, and check how the game runs.

Layer Collision Matrix

The Layer Collision Matrix is a one-to-many entries table where we decide what layer will collide with what layer in the game. You can access it from the main menu: **Edit | Project Settings | Physics2D**. When things get more complex, it is crucial to pre-plan the usage of **Layers** and why we are assigning them to `GameObjects` during the design phase. As an example, **Collectible** layer objects are different from **Obstacle** objects, and the **Player** wants to collide with the ground and with Collectables, and to be able to be hit by moving obstacles. However, we don't want the obstacles to collide with each other or with the Collectables, so we use the **Collision Matrix** to filter those collisions out. Notice how **Obstacle** layer objects will collide with other **Obstacle** layer objects. This can be changed, depending on the game design:



How we should set Layer Collision Matrix for our game prototype

Making it look and sound better

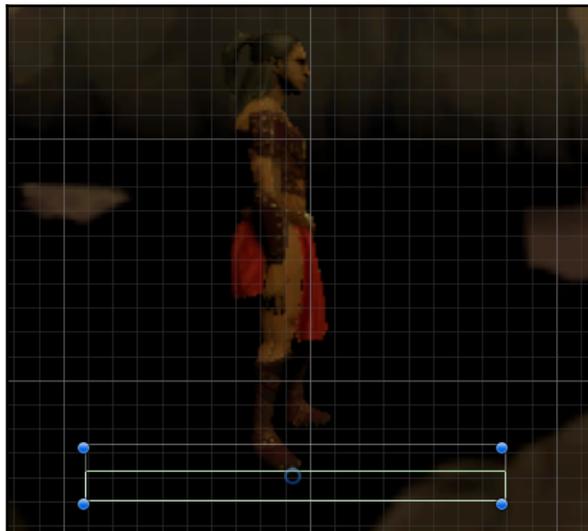
In this section we will implement some cool features to our 2D game prototype:

- Player shadow
- Audio
- Sprite shading

Adding a simple shadow for the character

Drag the `bshadow` you previously imported in the project into the scene, and place it just below the hero in the **Scene** view. Since we want the shadow to stay on the bottom of the character, below the feet, even when the character jumps, it is not a good idea to place the shadow `GameObject` in a parent-child relationship with the character's `GameObject`. The correct idea is to have it outside the hero's hierarchy with a short class instructing it to follow only the x axis of the playing character.

Now let's add a Box 2D Collider (**Menu** | **Component** | **Physics2D** | **BoxCollider2D**) to the `bshadow` `GameObject` and resize the height of the collider a bit, so that it is around half the height of the actual shadow sprite boundary, like in this **Scene** view screenshot:



The box collider in the Scene view with the 2D tool selected

Add a new component to the `GameObject` and call it `Follow2DTargetX.cs`, then open it and paste this C# code into the file body:

```
using System;
using UnityEngine;

public class Follow2DTargetX : MonoBehaviour
{
    public Transform target;
    private void LateUpdate()
    {
        transform.position = new Vector3(
            target.position.x,
            transform.position.y,
            transform.position.z
        );
    }
}
```

This script will make the fake shadow, which is a separated object and not a child of the player for technical reasons. It follows the x axis position of the player (but not the y and, of course, not the z axis).

We will add also a `Rigidbody 2D` to the shadow object and we will assign it a gravity multiplier of 3 and also a `Box Collider 2D`, to make sure it will collide and stay on the ground collider. We won't freeze the Z rotation like we did for the player, because we want the shadow to align to the `Edge Collider 2D` slopes of the ground for a better effect.

Try the game with the `Play` button and see how the shadow behaves under the feet of the character. If everything is okay, let's move on to the next objective.

Inserting audio: environmental sfx, background music, sound events

Let's try now to add some ambiance, music in the background, and some sound effects. We downloaded the sounds from freesound.org or opengameart.org. All the audio clips collected for the book are mostly free from copyright or under the creative commons license with some rights reserved for some soundtrack and sounds.

Open the `Chapter2-3-4/Audio` folder. Drag the `Volcano Lava Fire` and `Low Wind` audio clips into the scene.

Then, import the book's code, `Soundtrack.unitypackage`. After importing, drag the file into the `Assets/Soundtrack` directory and `DarkWinds` into the scene. On both `GameObjects` `Audio Source` components, set the `Loop` checkbox to `true`.

Now choose the `Warrior2D` player `GameObject` and manually add an `AudioSource` component to it. Activate the loop on this too, uncheck the **Play On Awake** option, because we don't want this sound effect to start automatically when the scene is loaded, and select the `Chapter2-3-4/Audio/hardrunfootsteps` file. Instead, we want to loop the audio when the character is running. To implement it, add these lines of code before the end of the `Move()` method in the `PlatformerCharacter2D`:

```
if (m_Anim.GetBool("Ground") && Math.Abs(move)>0.5f)
{
    if(!GetComponent<AudioSource>().isPlaying)
    GetComponent<AudioSource>().Play();
}
else GetComponent<AudioSource>().Stop();
```

To avoid calling the `GetComponent<AudioSource>()` API method each frame we are going to cache the value of this component in the `Start()` method, so because we did remove it, we will add it in again and add a new private variable of the `AudioSource` type to store the object:

```
m_audioSource = GetComponent<AudioSource>();
```

We will also change the code we wrote earlier into this:

```
if (m_Anim.GetBool("Ground") && Math.Abs(move)>0.5f)
{
    if(!m_audioSource.isPlaying) m_audioSource.Play();
}
else m_audioSource.Stop();
```

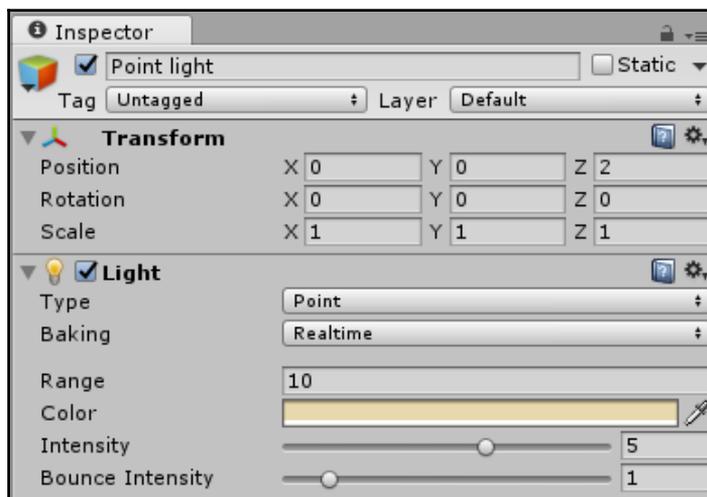
Press `Play` and test the game; you should hear the background music, the ambient sound, and, when the player runs, the footsteps audio playing. There is a little issue with sound volume, as we can barely hear the footsteps, and the ambient sound is way too loud for the music soundtrack. Change the overall volume of these two audio components, setting the `DarkWave` soundtrack to `0.5` and the ambient sound to `0.25`. It should sound a lot better now!

Another easy enhancement to apply would be to tweak the pitch (hence, the speed) of the footstep sound. Try to set a value between `1.1` and `1.25` (which means slightly faster and higher pitched) to hear footsteps in sync with the steps taken in the character animation while running.

Sprite shading with real-time lights

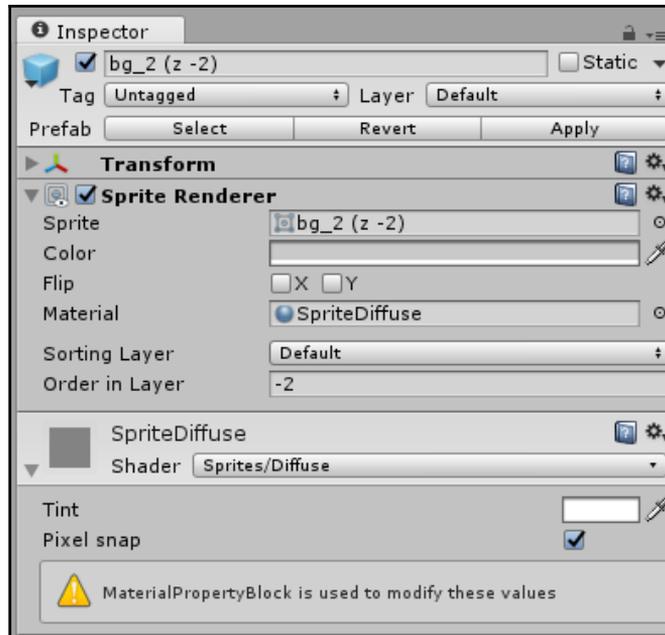
As you might notice, sprites are not influenced by real-time lighting. This is the default behavior in Unity for some reason; most 2D games are created with hand-painted, bi-dimensional pictures and don't require real-time lighting at all! Though we might want to have this feature, our game is not for children, it must be creepy, or have a particular environmental lighting. Think about re-making something such as the glorious *Commodore Amiga: Alien Breed* in 2D with Unity. Of course, we would like to use real-time lighting with sprites for many reasons: effects for shooting weapons, light effects in special rooms or corridors, or other special FX that will create light with little effort!

Let's try to achieve this at least for our parallax background graphics. Let's suppose that the *DarkCave* is really DARK! We should have a way to see through the player, which will be fully illuminated (we are going to change the shader for the hero and the Collectables) with a sort of spotlight effect. Because spotlights are more expensive on the CPU, we will achieve this using a point light instead. Create one from the main menu, **Assets | Create | 3D | Point Light**, then add this GameObject as a child of the camera. It should have a relative local position of 0, 0, 2, to allow the rest of the following settings to influence the backgrounds. Set the point light as shown in this screenshot:



The magic is done by changing the Sprite component material; the default sprite material uses the Sprite Default shader, but our material will use the Sprite Diffuse shader, which will be influenced by lighting. To do so, create a new material and call it **SpriteDiffuse**, and choose the **Sprite/Diffuse** shader from the **Shader** list.

See the following showing one of the background layers, with the new **SpriteDiffuse** material assigned:



All you have to do is to change the material to all the sprites of the background layers. Check the **Game** view; if you did everything right, you should see a black circle of light in the middle, revealing the player and part of the background.

Writing some additional game logic

Although it might look great already, we will write some more code in the player controller to use the time and score to determine win and loss conditions, check health status, and declare the game over when it is exhausted.

Before starting on the code, we will create four new Text UI objects in the canvas:

- StartLabel
- EndLabel
- GameOver
- MadeIt

These objects will basically just contain some text, but will start disabled to appear when the game logic needs them.

We need to add four relative public variables to our custom class and then to drag those three UI objects to the respective slots in the player's component.

After the other variables at the start of the class, add the following:

```
public GameObject StartLabel;
public GameObject EndLabel;

// Final UI objects
public GameObject wonObject;
public GameObject lostObject;
```

Open the `Platformer2DCustomUserControl` class in the code editor (Visual Studio or Mono Develop).

Go to the start of the `Update()` function in the code.

First of all, we will use our Boolean `gameInactive` variable to check the status:

```
// Update is called once per frame
void Update()
{
    // Process the inputs, time and game logic, if game is active
    if (!gameInactive)
    {
```

Then, we will check whether the time limit has been reached, after our code that calculates and prints the time. If the time limit is reached, the game input will stop setting `gameInactive` to `True` and display the appropriate message:

```
// Calculate elapsed time
elapsedTime += Time.deltaTime;
// Update time display
timeLabel.GetComponent<Text>().text = "Time:" + Mathf.RoundToInt(timeLimit
- elapsedTime);

// Check game logic
if (timeLimit - elapsedTime < 0)
{
    if (score >= 1000) wonObject.SetActive(true); else
    lostObject.SetActive(true);
    gameInactive = true;
    EndLabel.SetActive(true);
}
```

```
// If score 1000 is reach, win the game
if (score >= 1000) {
    wonObject.SetActive(true);
    gameInactive = true;
    EndLabel.SetActive(true);
}
```

To stop the execution at the beginning for two seconds and to display the instructions in StartLabel UI Text, we will write these few lines of code, using the ability to run the Start () function as a coroutine:

```
// Start() automated function can be used as a coroutine
IEnumerator Start()
{
    gameInactive = true;
    // Wait 2 seconds to start the game
    yield return new WaitForSeconds(2f);
    StartLabel.SetActive(false);
    gameInactive = false;
}
```

Homework

We can enhance this game in many ways, and, from my point of view, there are still some glitches that we could try to fix or work around. This will be your task, to find ways to implement a better feature.

Try to work on these issues/requirements:

- Spawned coins should be destroyed after the gold chest that spawned them is out of view
- Walk to jump and run to jump can be done better; find a way to do this
- Rolling stones catch the parallax movement and accelerate when the direction is opposite; try to fix this
- Play with different background sizes and positions and see which is the best fit
- Try to implement your own background graphics

Summary

We learned a bit more about scripting, going deep enough to modify existing classes or create our own! We also took a look at how the new Physics2D framework works with different types of Colliders 2D and Rigidbody 2D. We learned how to modify C# classes for our needs, and how collision and trigger detection works in 2D. We had our first look at the new Unity UI. We looked at how the Unity Editor works in general and used real-time lighting with sprites for the first time!

In the next chapter, we will explore how to implement a complete, 3D, fully-animated, player-controlled character.

5

Character Animation with Unity

Unity was born as a 3D game engine; the 2D elements included until version 3.x were just related to GUI elements used in 3D games, and there was not a real 2D game development toolkit. With the rise of the mobile platform, one of the most acclaimed genres has been 2D hand-painted games such as *Plants vs. Zombies*, *Angry Birds*, *Cut The Rope*, and many others. In version 4, Unity introduced a new full set of 2D game development tools, such as the Sprite Packer, the 2D Rigidbody, and the 2D colliders we just chewed over in the previous chapters, to help developers create even more stunning 2D titles easily and proficiently. These were not the only additions to version 4 of the engine; one of the most valuable additions was the Mecanim system with the new Animator component, which completed its evolution in the 5.x cycle, a completely new way to manage rigged 3D character models and complex animation state machines.

Some history of character design

Once upon a time, this specific part of game development was a real nightmare. Animation packages, such as 3DSMax, Maya, Lightwave, and Softimage, were all different and also used a different coordinate system (Maya/Max). They all implemented character rigging and animation with bones in different ways.



I would say that rigging and implementing an animated character in games was the hardest part of game development. Before 1996 (*Quake 2*), there was no bones support in any game engine and all animations were done frame by frame after exporting all the frames models from the animation software package. With *Half Life*, a modification of the *Quake* (1) engine, 3D character models with bones were introduced for the first time in a game engine.



This new real-time feature could also bring out fantastic mods, such as *Counter Strike* (1998), featuring specific hit boxes (colliders) corresponding to specific parts of the body. Luckily for developers, back in 2007, the .fbx 3D file format came out from Autodesk, when it acquired Maya and finally brought out a standard 3D format that supported smoothing groups, multi-material, and, most importantly, weighted bones and animations. This format unified Maya and 3ds Max translation and workflow. Even if differences still persist, Unity took advantage of this cool, new 3D format and its unique features.

The existing model formats, especially engine-specific ones, are still very fragmented. For this reason, there are tools such as Ultimate Unwrap 3D (<http://unwrap3d.com>), born as a UV mapping tool, which can convert/translate almost every 3D model format.

Unity Animation system introduces a totally new way to manage human and non-biped characters, called Animator made of the Animator component and the Animator controller file, which can be edit in the **Animator** window. Rigging animations with ease with a powerful set of tools that allows you to: set up an avatar and reuse its properties for other characters; set masks on the body rig; animate part of the body with one animation and another part of the body with another animation; and set up Blend Trees to control a full set of movement animations with input coordinates. Furthermore, with the help of some more coding, you can have part of the skeleton be driven by user input through **Inverse Kinematics (IK)**.

We are now switching back to 3D development to understand how to deconstruct the `ThirdPersonCharacter` class and use it for our needs, and how to use the Animator component and its `Controller` state-machine editor to control animations' blending and behavior.

In this chapter, we will learn the following things:

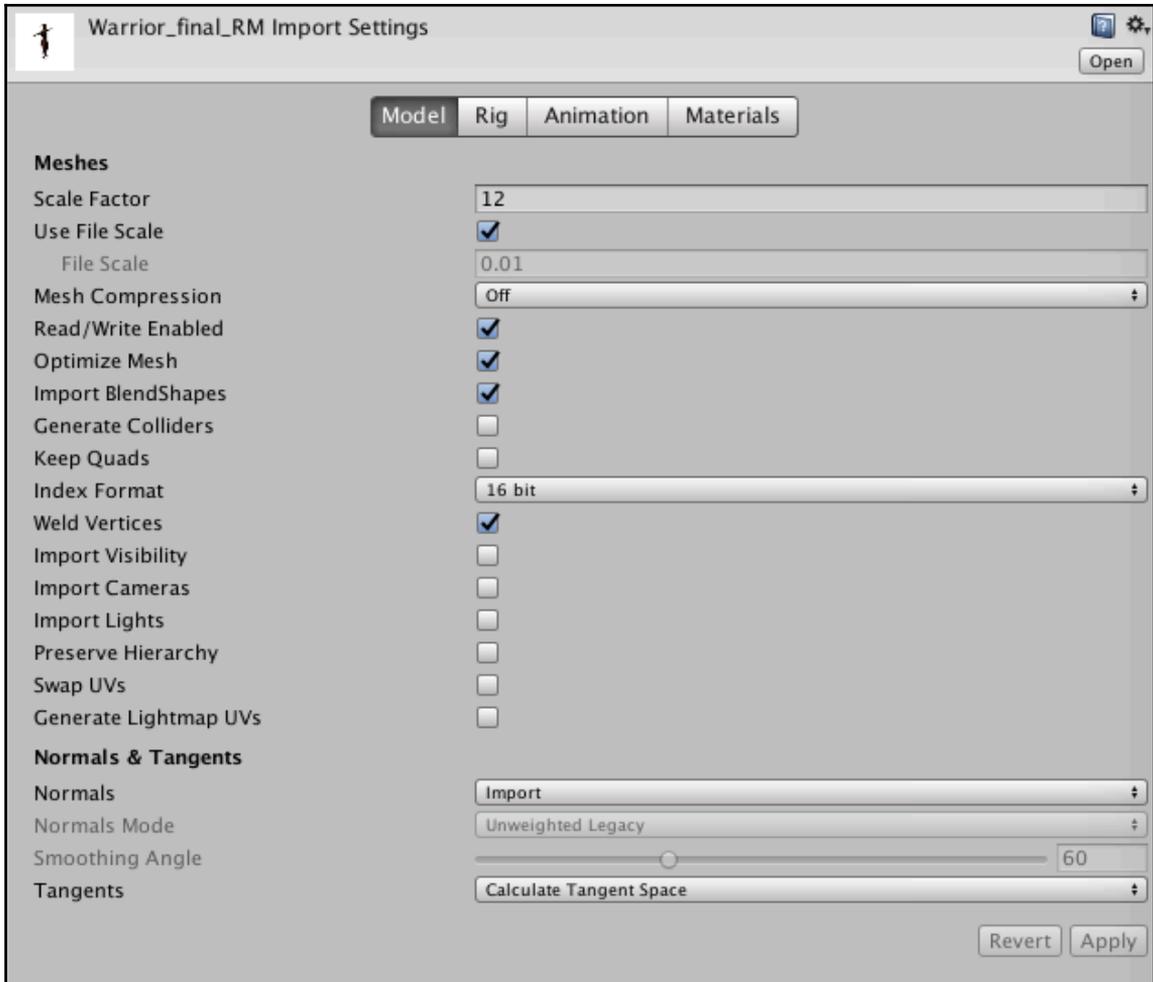
- Importing split animation from multiple animation files and setting up a character with the Legacy Animation System
- Setting up the `.fbx` rigged model and `.fbx` animation files in the model importer settings for Animator
- Writing a simple custom class to move a character and interact with the `Animator Controller` states
- How to really take advantage of this powerful system to implement a player with **Root Motion** enabled
- Modifying the C# code of standard assets' `ThirdPersonCharacter` and the way it interacts with `Animator Controller` through its *parameters*' variables
- Modifying the `ThirdPersonUserController` component for our needs
- Implementing cloth physics simulation on a character's beard, hair, or skirt
- Implementing Inverse Kinematics (IK) animation on a part of the skeleton through coding

Unity Legacy Animation System

The early Unity versions' Legacy Animation System can still be used in Unity for a wide range of things, such as animating the color of a light or other simple animations on 3D objects in a scene, as well as animating skinned characters for certain kinds of games. We will look at the basic settings for the Legacy Animation System. Afterwards, we will step into the new animation system, gaining an understanding of the `ThirdPersonCharacter` prefab, and looking at the difference between the in-place and Root Motion animation methods available within Animator.

Importing character models and animations

To import a model rig or an animation, just drag the model file to the `Assets` folder of your project. When you select the file in the **Project** view, you can edit the **Import Settings** in the **Inspector** panel:

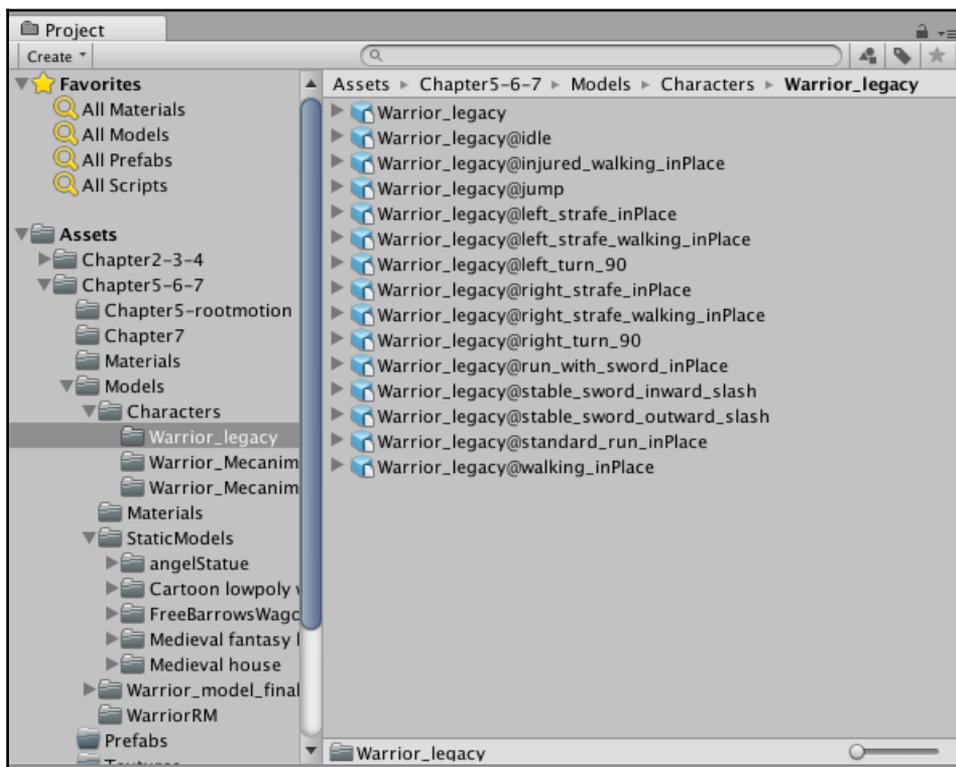


Please refer to the updated Unity online manual for a full description of the available import options: <https://docs.unity3d.com/Manual/FBXImporter-Model.html>.

Importing animations using multiple model files

The common way to import animations in Unity is to follow a naming convention scheme that is recognized automatically. You basically create, or ask the artist to create, separate model files and name them with the `modelName@animationName.fbx` convention.

For example, for a model called `Warrior_legacy`, you could import separate idle, walk, jump, and attack animations using files named `Warrior_legacy@idle.fbx`, `Warrior_legacy@jump.fbx`, `Warrior_legacy@standard_run_inPlace.fbx`, and `Warrior_legacy@walking_inPlace.fbx`. Only the animation data from these files will be used, even if the original files are exported with mesh data from the animation software package:

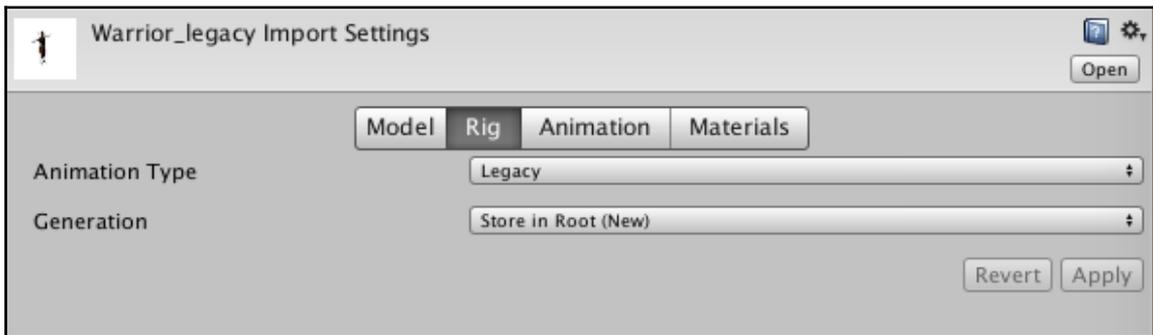


A list of animation clip files for an animated character

In the editor's **Project** view, the `.fbx` suffix is not shown in the preview, but can still be seen in the bottom line of the view. Unity automatically imports all the files, collects all the animation clips from them, and associates them with the file without the `@` symbol. In the example above, the `Warrior_legacy.fbx` file will be set up to reference `offensive_idle`, `jumping`, `running_inPlace`, and `sword_and_shield_walk_inPlace`.

To export the base rig, simply export a model file from your favorite digital content creation package with no animations ticked in the FBX exporter (for example, `Warrior_legacy.fbx`) and the four animation clips as `Warrior_legacy@animname.fbx` by exporting the desired keyframes for each one of them (enabling animation in the graphic package's FBX export dialog).

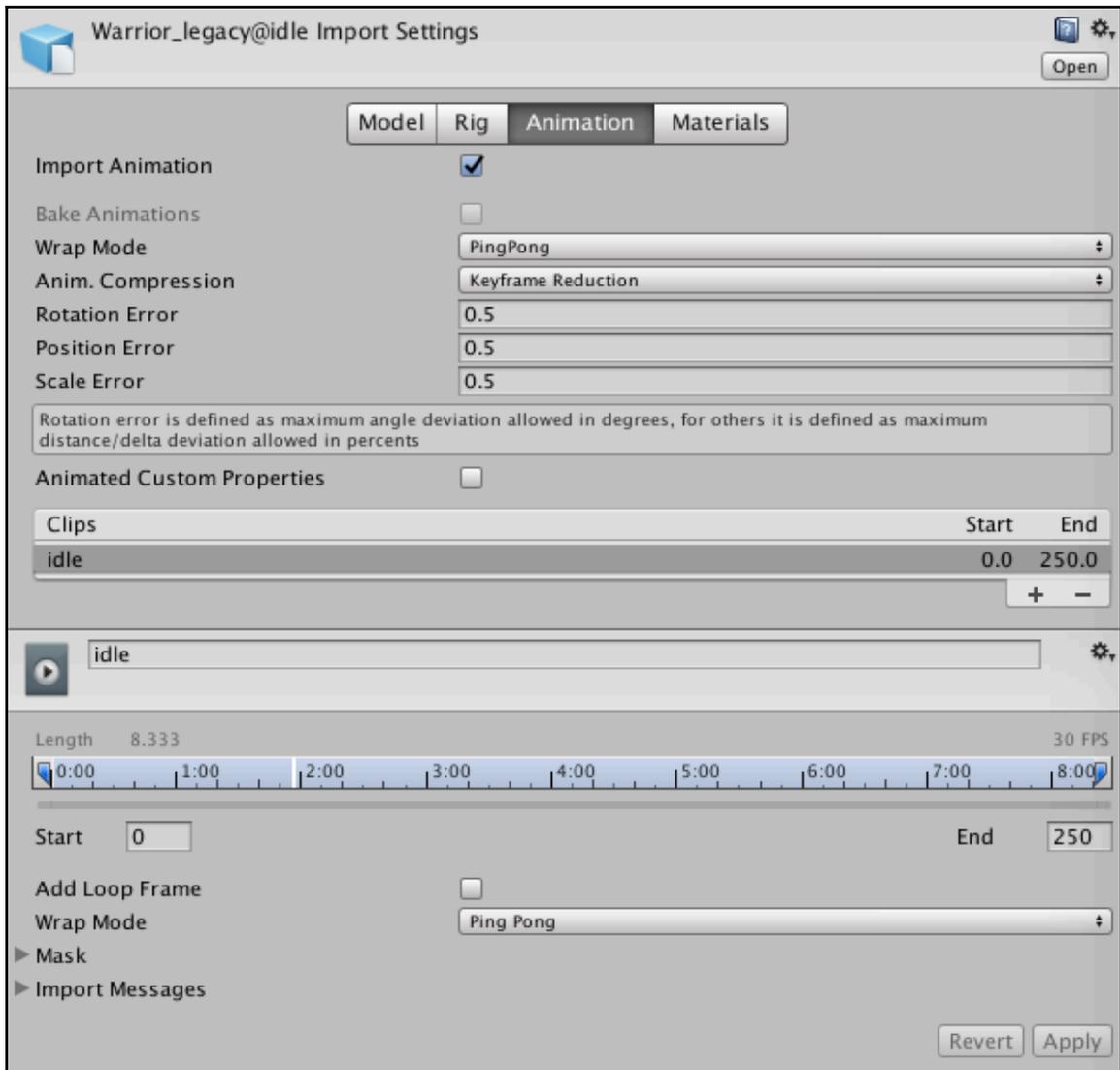
When imported in Unity, we will select the main rig file (`Warrior_legacy.fbx`) and set its **Rig** type to **Legacy**:



Setting up the animation

We need to instruct Unity on how we want to play these animation clips, for instance, we certainly want the walk, idle, and running animation clips to play in a loop, while the jump and the attack animation clips should play in a single shot.

Choose the **Idle** animation clip in the **Project** view folder where the legacy animation reside and then switch to the **Animations** tab in the **Inspector**:



Set the **Wrap Mode** to **PingPong** in both the top and bottom parts of the panel, as shown in the preceding image.

In many cases, you might also want to create an additional in-between loop frame, checking the **Add Loop Frame** option. This is needed to avoid an ugly animation loop being performed because the first and last frame of the animation are too different from each other. Click on the **Apply** button at the bottom of the panel to apply the changes. This will be required if the first and last frames of the animation are much different and require an additional frame in-between to interpolate between the two in order to obtain a good loop for this Animation Clip. Now, drag the `Warrior_legacy.fbx` main file into the scene. You should see a new GameObject with an **Animations** component attached, with all the reference clips already set up, and with the first specified to play at start when the **Play On Awake** checkbox is selected in the component (default).



You can look at the final result for this part in the `Chapter5_legacy` Unity scene in the book's code project folder.

Building the Player with Animator

The Animator component was introduced in Unity 4 to replace the older Legacy Animation System. If you are completely new to Unity, you should start directly with the new animation system and consider the old one as still being good for many things, not only related to character animation. Animator introduced many cool things that were only partially available (and only through coding) with the old animation system.

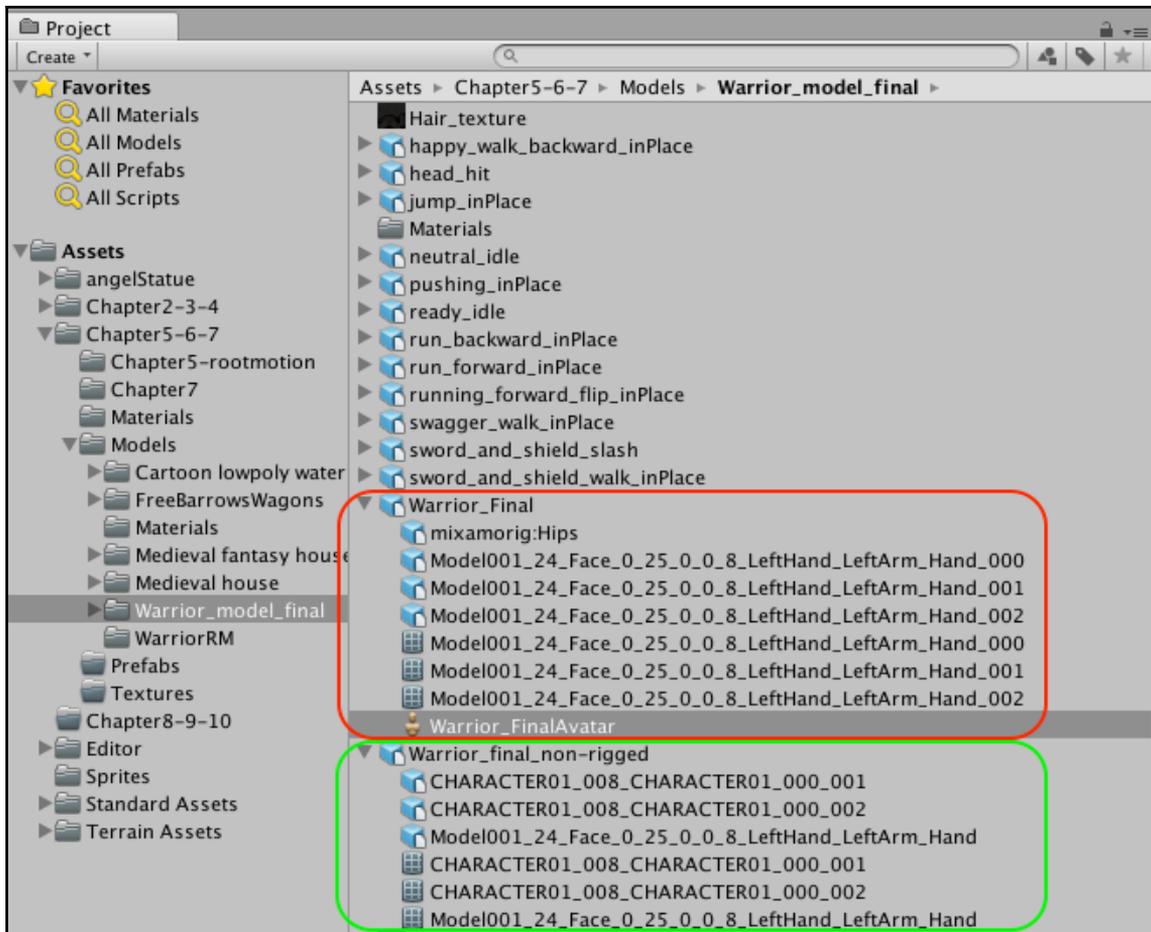


In the code folder, under `Chapter 5-6-7/Models/Characters`, you will find three folders for the warrior model rig. One is meant for the old Legacy Animation component, and the other two are for use with the Animator.

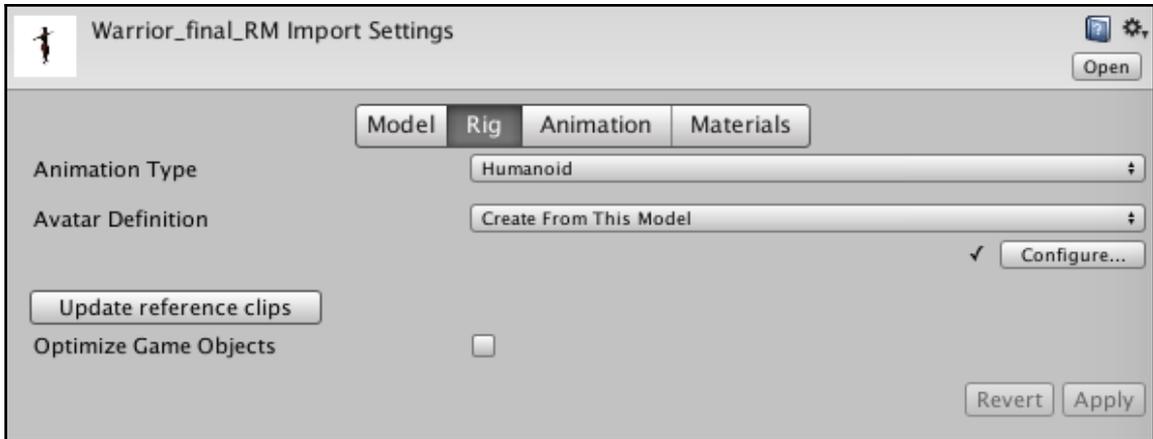
The new system is made by a new animation component, the `Animator`, a powerful state-machine that controls the whole animation process and the Avatar configuration system. The `Animator` component will be mapped to a corresponding avatar and to an `Animator Controller` asset file, which can be created, like other files, from the **Project** view and edited in the **Animator** window.

What is an avatar in Unity?

When an `.fbx` 3D model file with a skeleton made of joints/bones is imported in Unity, if you expand the file in the **Project** view, you will see, among the various parts of it, an avatar's icon. The following screenshot represents the `Warrior_Final.fbx` rigged model automatically created by the `Warrior_FinalAvatar` component:



When importing a rigged model instead (an FBX model with a skeleton or bones and, optionally, animations), Unity will configure it automatically for a **Generic** avatar. A Generic avatar is meant for any kind of non-human character rig, such as animals, non-biped monsters, plants, and so on. Typically, for your biped/humanoid characters, you want to switch the default import flag for the **Rig Animation Type** to **Humanoid**:



Biped character



This term comes from the Latin word *bi* (two) and *ped* (foot); this 3D animation-specific term indicates an anthropomorphic/humanoid character standing and walking on two legs. This name was introduced into 3D animation by 3D Studio Max, where Biped was the term for Character Studio to manage a rigged human character and its animations.

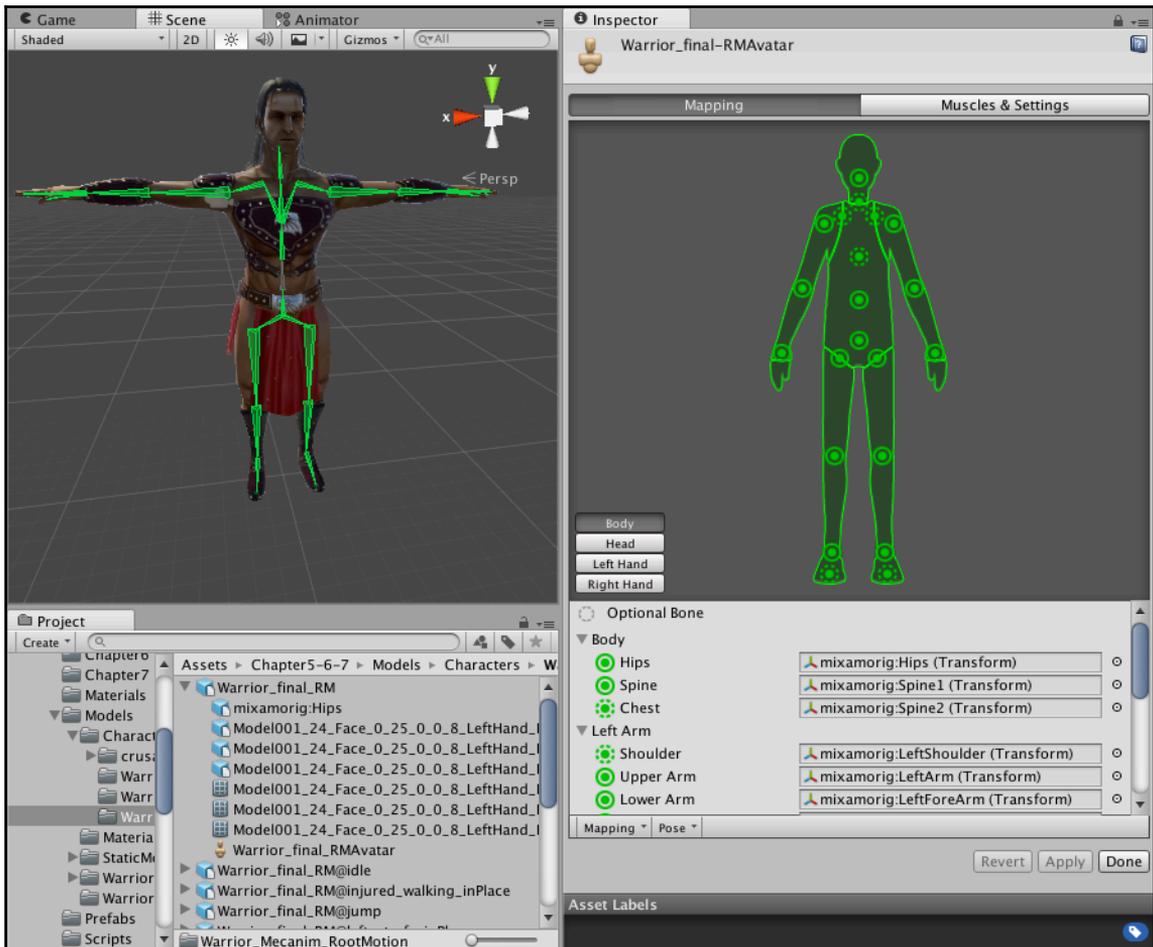
As the default import setting for the **Rig** is **Generic**, we will switch to **Humanoid** for all the `.fbx` files in the `Warrior_Mecanim_InPlace` folder with the only exclusion being the non-rigged `Warrior_final_non-rigged.fbx` sample model mentioned earlier.

Configuring an avatar

Now, hit the **Configure** button and the actual scene and the **Inspector** will be temporarily replaced with the avatar, (as in the following screenshot), until the **Done** button is clicked and the editor returns to the previously loaded scene.

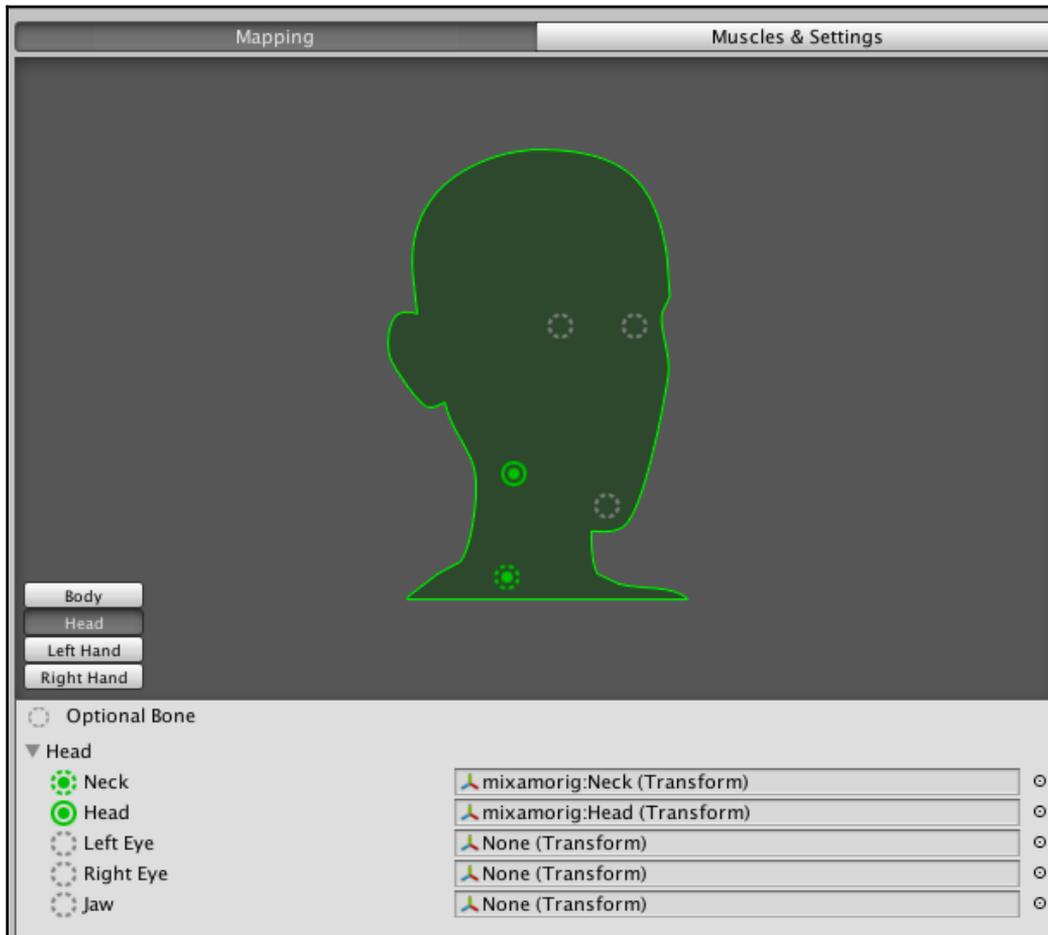


Because the model included in the book codes is already a Mecanim-ready rig, you can just click on the **Done** button.



On the left, the Scene view switched temporary for showing Avatar configuration results and the Inspector on the right showing configuration options

Most of the time, and in this case, you will not set the mapping for hands, fingers, and eyes separately, so the first of the four tabs (**Body**, **Head**, **Left Arm**, and **Right Arm**) will be enough for our purpose. The head is usually mapped for independent eyeball movement and/or jaw movement to allow a basic character speech movement whenever your game needs any of these features. The head part of the avatar configuration **Inspector** panel is shown as follows:





A quick note on lip sync. Lip sync is an advanced technique, where a 3D character's face will change and animate its mouth and eyes when a certain audio file is playing. Unity doesn't support lip sync out of the box, but it can be done in many ways with external libraries and an appropriate model rig. Since Unity 4.5 onward, animation Blend Shapes are supported, allowing facial muscles' gestures to be embedded in the .fbx model and used in real time by the application. This technique is more modern than standard lip sync for game character's speeches; in both cases, a library or a discreet amount of coding would be needed to make the character speak correctly when the corresponding audio file is played.

Hands mapping will be used only when your characters need fine finger movements hence will not be covered.

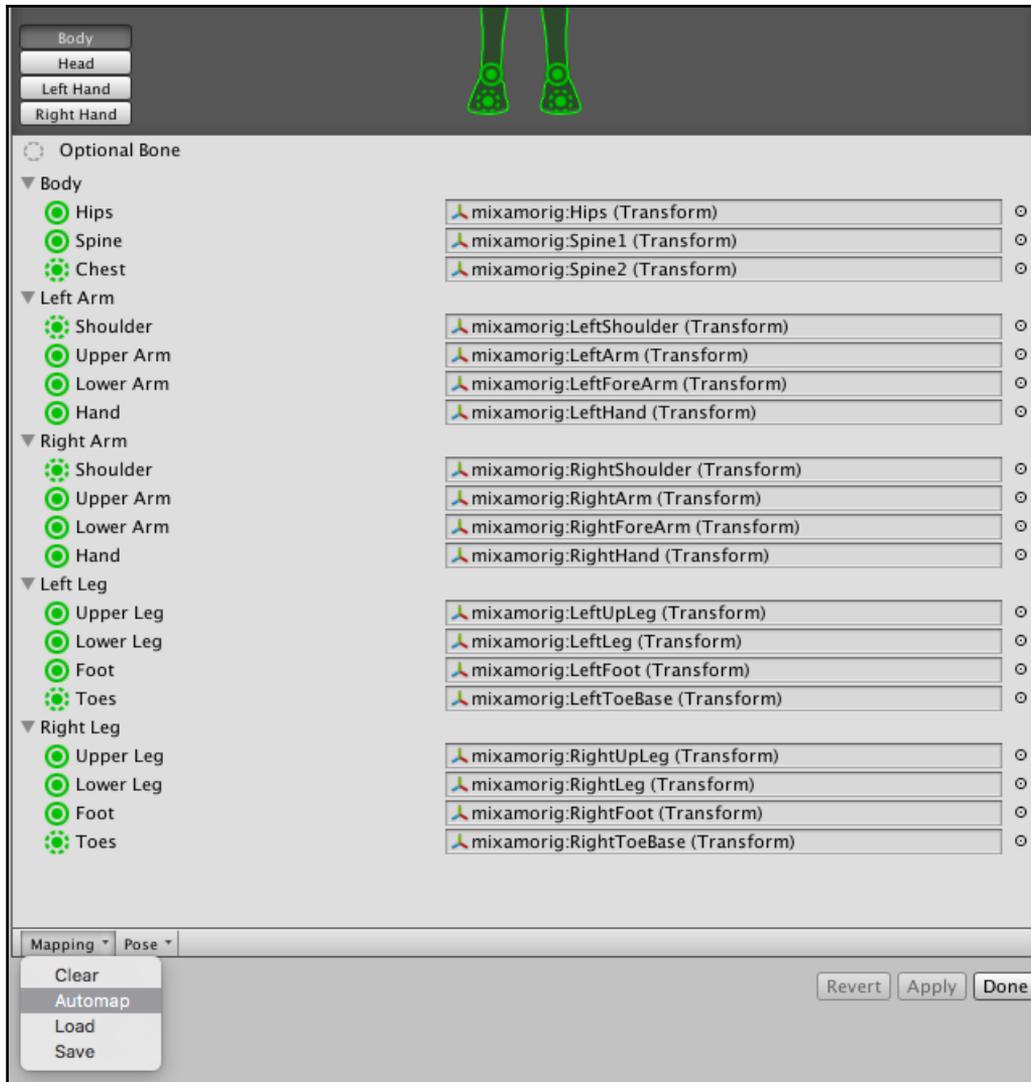
The best scenario for this is a game where characters perform a lot of specific actions with many different tools (guns, hammers, knives, hacking, or maybe just making gestures while talking during a cinematic cut scene). Another example would be an avatar for virtual reality, where the Leap Motion, Data Gloves, or similar devices are used to track the hands of users with the 3 phalanges of their 10 fingers.

If the rig you are importing is not *Mecanim-ready*, this is the place to map your bones to the appropriate spots on the green diagram in the **Inspector**, which is subdivided into body, head, left hand, and right hand.

To configure an Avatar from a model that was not rigged following Mecanim's skeleton rules, we have the following two options:

- Using the auto-mapping feature available, which will try to automatically map the bones for you
- Manually map the bones of your model to the corresponding spots on the diagram

The avatar configuration **Inspector** panel shows the skeleton's main bones mapped to the avatar:





The **Automap** feature, accessible from the drop-down menu at the bottom-left part of the avatar **Inspector**, can automatically assign the bones of your models to the correct ones for a mecanim rig. This is mainly performed by reading bone names and analyzing the structure of the skeleton, and it is not 100% rig proof. So, you might need some tweaking (manual mapping) of your custom character models.

As you can see, there are also **Load** and **Save** options to store this mapping. This is useful if you have a whole bunch of rigged character models all done with the same skeleton naming convention.

The **Clear** option will clear up all the current bone mapping. The **Pose** drop-down menu is needed only if you want to enforce the T-pose, or sample the actual pose, and is rarely needed, but can help fix eventual modeler/3D artist mistakes or to make variations of an avatar.

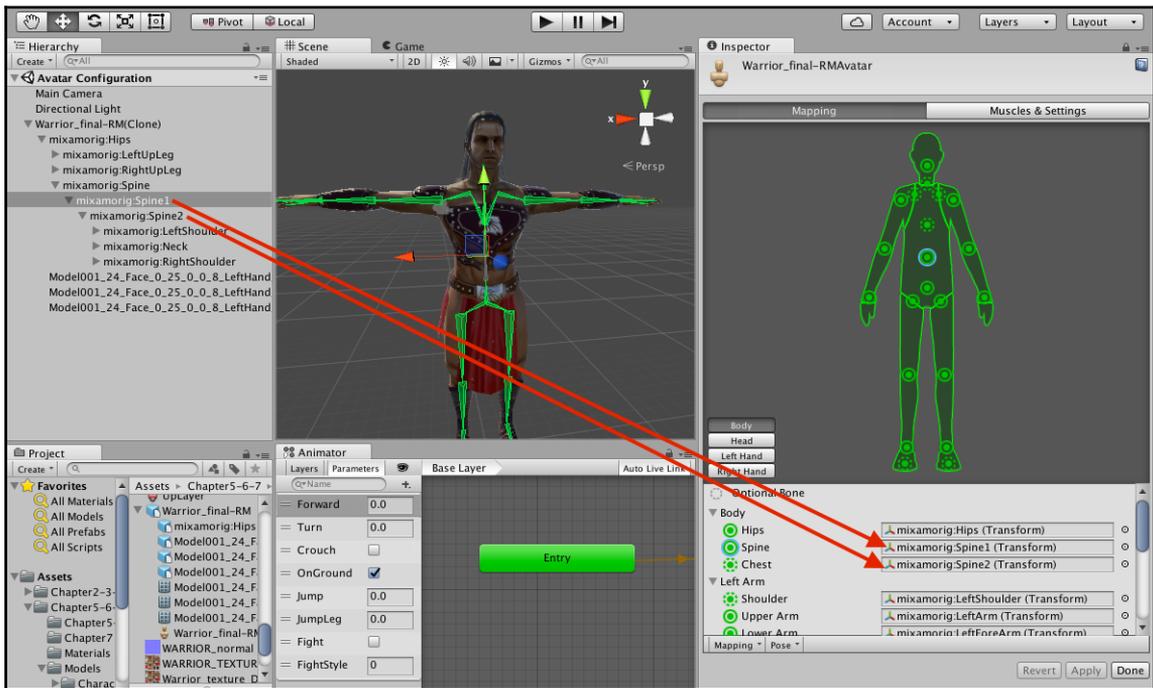
Fixing issues with the default bone mapping

Our model animates and moves just fine, but we might want to tweak-fix the bones mapping for the **Spine** and the **Chest** bones.

If you look at our model, the **Spine 1** bone is too low for the **Spine** spot and **Spine 1** is too low for the **Chest** avatar spots.

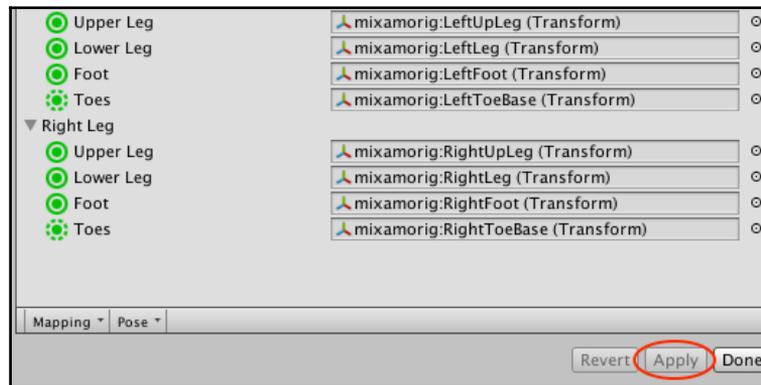
The best way to address this problem when you have many models in your project is not via the small circle on the right of the **Transform** name in the avatar bones mapping tool; instead, we perform this by dragging the correct bone from the **Hierarchy** view into the spot for maximum control.

Open the **Hierarchy** of the skeleton of the model in the **Hierarchy** view and look for the `mixamorig:Spine1` GameObject, then drag it into the **Spine** spot in the avatar bone mapping tool window, as shown:



Now repeat the step for the `mixamorig:Spine2` GameObject and drag it into the **Chest** spot.

When these steps are done, click on the **Apply** button to save your changes:



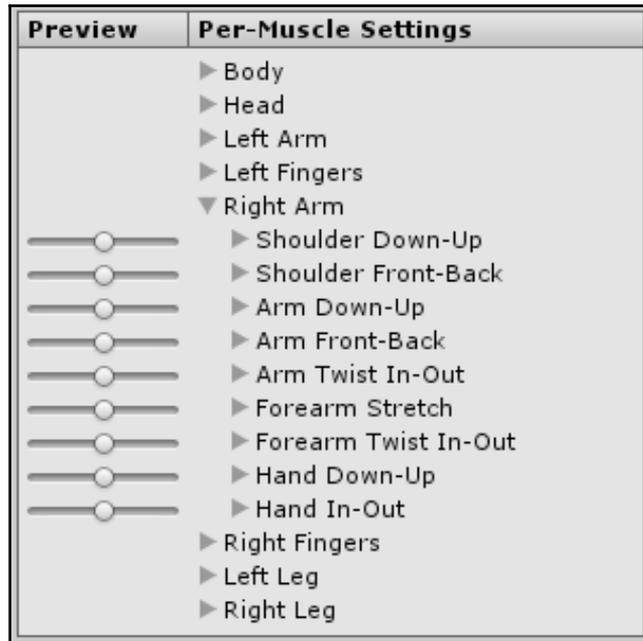
Continue editing by going to the other tab of the avatar configurator, the **Muscle & Settings** tab, as described in the following section.

Configuring muscle actions and settings

Avatar's **Muscle & Settings** allows you to control the range of motion of the avatar of different bones using muscles. Once the avatar has been properly configured, Unity will automatically recognize the bone structure and allow you to start working in the **Muscles & Settings** tab of the avatar configuration inspector to check the model for eventual errors or possible geometry penetration while performing certain animations:

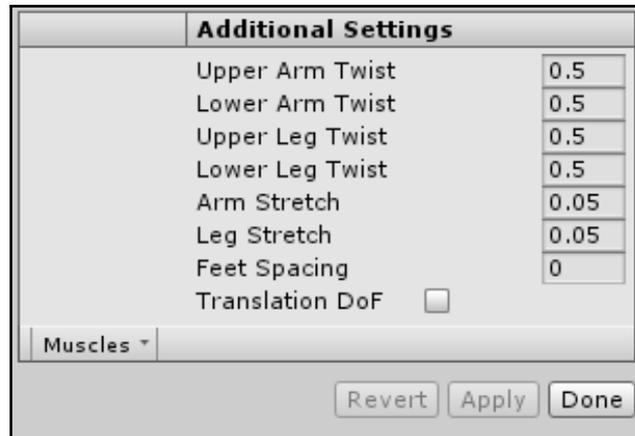


With the first two panels, it is very easy to tweak the character's range of motion and ensure that the character skin (the mesh deformed by bones animation) deforms in a convincing way when its skeleton is animating and that it is free from any visual issues or self-overlaps before proceeding with animation setup:



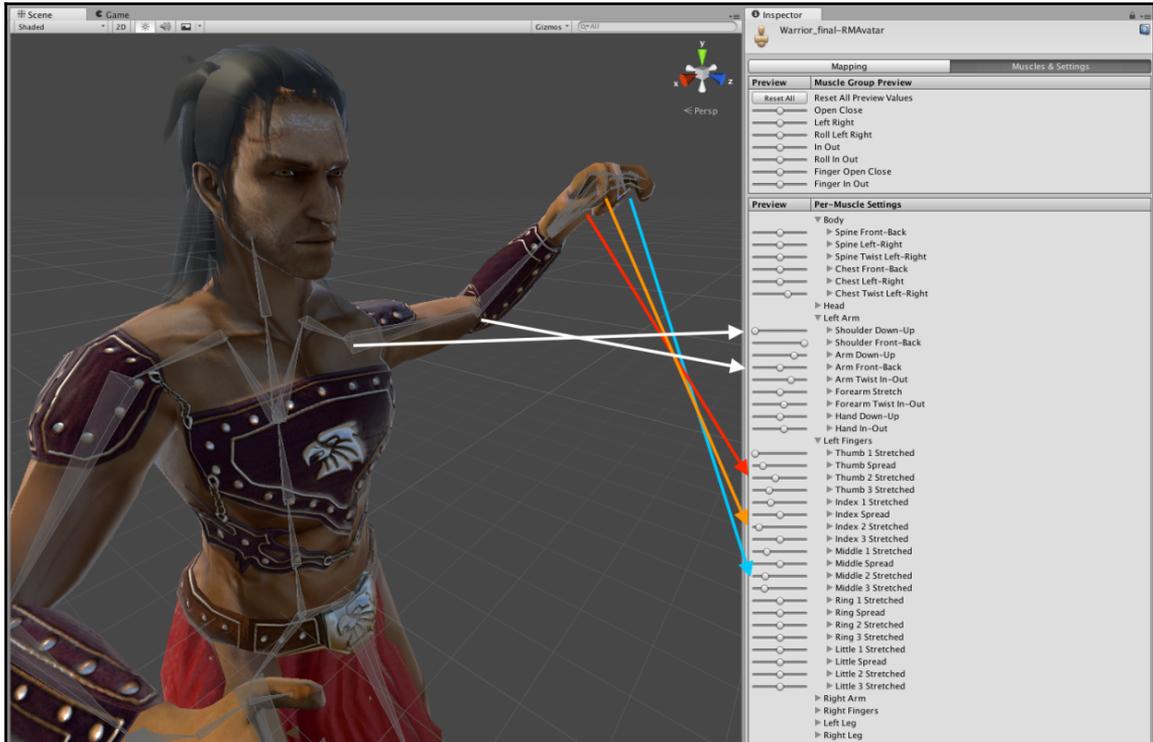
The first two panels are just for preview purposes and are meant to enable you to understand whether your rig will behave correctly with standard animations or other rig animations that might fit the character we are working on.

The last panel, **Additional Settings** is where we can tweak the rig muscles setup to fix/tweak eventual geometry penetration or skin misbehaving:



The **Translation DoF** checkbox, shown at the bottom of the **Additional Settings** panel, allows you to enable the use of translation animations for the humanoid. With this option disabled, the bones are animated using only rotations. You want to enable this option only if you know for sure that your animation contains animated translations of some bones in the space.

Enabling Translation DoF comes with some performance cost because the animation system needs to do an extra step to retarget the animation. Later, in the *Using a better approach – Root Motion animation* section, we will see in more detail what Root Motion is and how to take advantage of it to achieve more natural and fluid animations:



An example of a geometry coherency test with the Muscle & Settings tab on the avatar

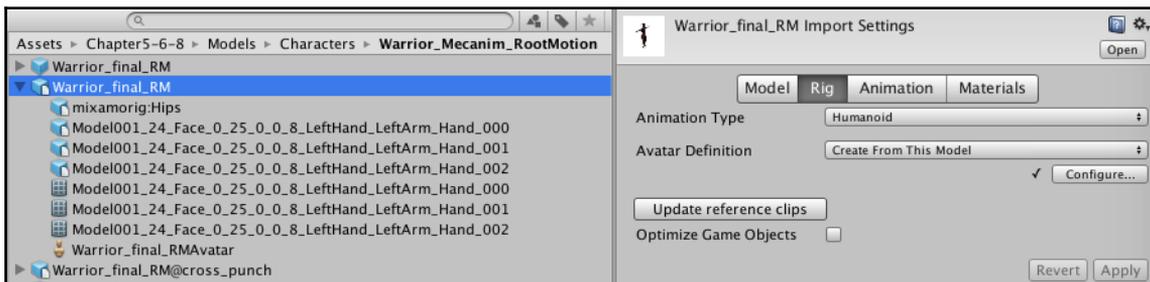
Importing animation clips for Animator

Before to dive into clips importing we should note that book codes provides three sets of animations. The first, is for the old legacy animation system, in the folder: `Warrior_legacy` and other two for working with Animator, one for in-place animations and one for root motion animations. Because in-place and root motion animation types are so different in the setup of the Digital Content Creation tool the animators used to realize the rig and the animations .fbx exports, they were split in two sub folders: `Warrior_Mecanim_inPlace` and `Warrior_Mecanim_RootMotion`. In each folder you can find: the main rig .fbx file, the animation clips .fbx files and the Animator Controller file for each setup. Also, consider, that as the animation type is chosen by the artist, there is no difference in the import setup to follow for the two modes, so in this case, the screenshots will show the root motion setup only.

Naming conventions



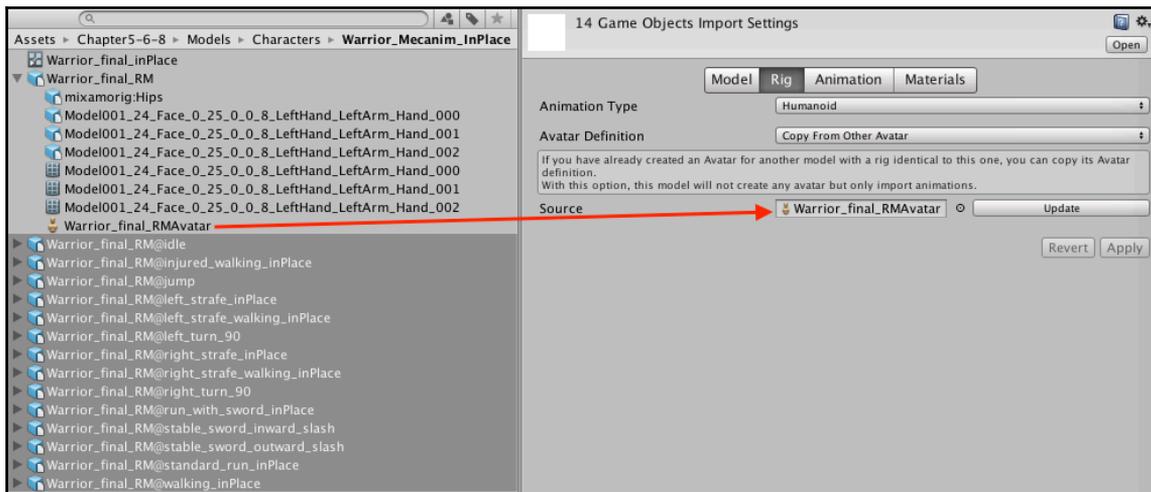
Take note of the naming convention used to export the warrior model and animation files: the name of animation @ the name of the model is used because this also supports the old Legacy Animation System, and these clips will be automatically associated to the main rig model. With Animator this naming convention can be void, as we are specifically assigning the animation clips to the corresponding avatar from the main rig, but when used, it can lead to faster animation clips avatar assignment at edit time.



The base rig .fbx file for Root Motion prepared for Animator with Humanoid Animation Type, below the Update reference clips button

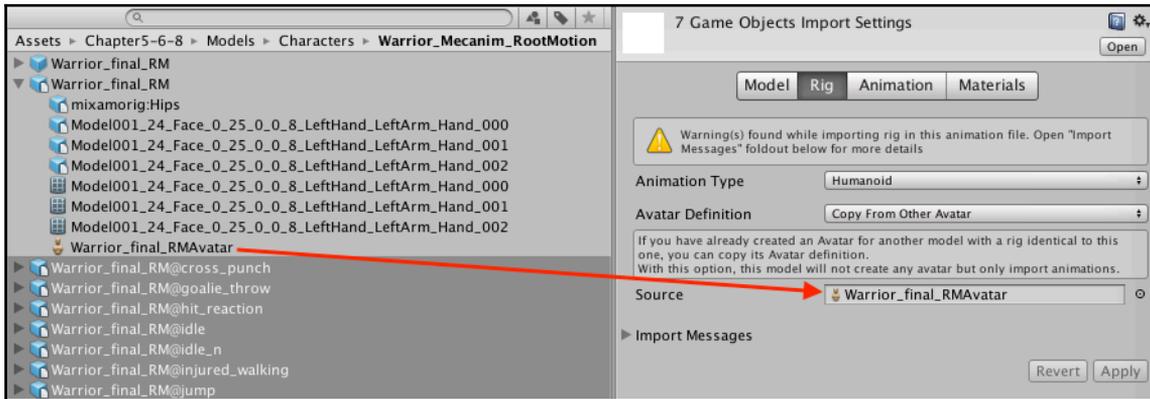
If the naming convention is followed, you can just select the main rig with no animation imported (For example: `Warrior_final_RM.fbx`) and then click the **Update reference clips** button to let Unity automatically assign all the clips for this **Avatar** (see preceding screenshot).

Otherwise, you will need to manually choose which animation files are mapped to that avatar by dragging the base rig **Avatar** into the **Source** (below **Avatar Definition**) spot of the **Rig** panel:



The **Inspector** panel can edit multiple GameObjects if the components attached to them are of the same type. This works for GameObjects in the **Hierarchy** view representing **Scene** content, as well as for assets in the **Project** view, such as `.fbx` models. Thanks to this, we could have used a multiselection approach to perform the same job. Select all the other `.fbx` files from the **Project** view (the files with `@` in their filename) in the **Inspector**, in the **Rig** tab, and instead of choosing **Create From This Model**, we will choose **Copy From Other Avatar**.

We will see a new slot appear below the **Avatar Definition** selector, the **Source** spot. We will drag the main rig (`Warrior_final_RMAvatar`) there, or simply click on the circle beside the slot and choose `Warrior_Avatar` from the list, as is explained in the following screenshot:



The drag method is preferred as, when working on setting up your project's assets, there might be more than one Avatar named: `Warrior_Avatar` in the list because we have many versions of the same character in the `Assets` folders and it can be confusing. Dragging the `Warrior_final_RMAvatar` from the base rig into the **Source** slot is safer and faster if you choose multiple animation clips.

In the next section, we are going to see how to set up various animation clips to play individually, so we will switch to the tab **Animations** of the **Import Settings**.

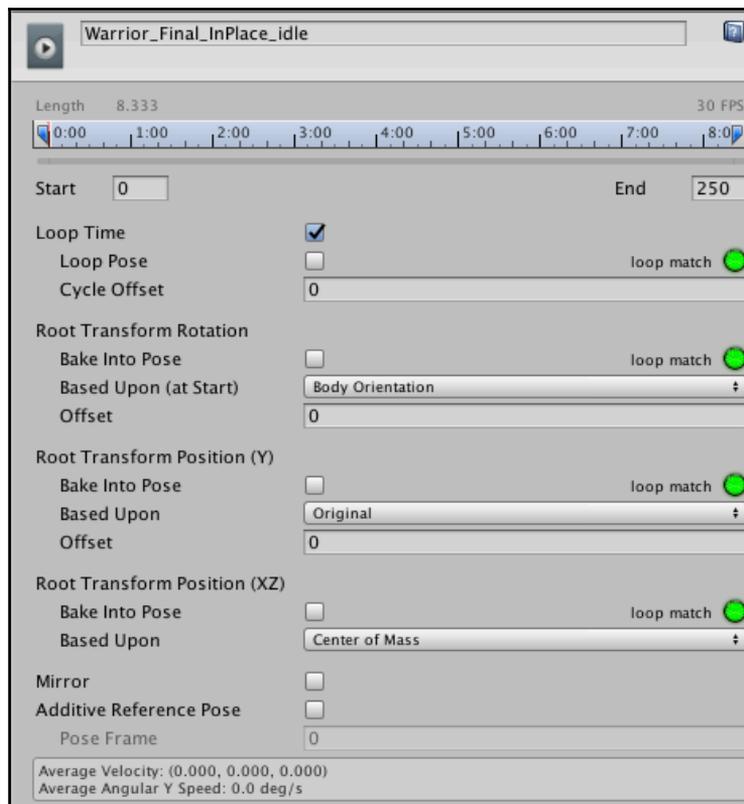
Setting up animations to ensure correct looping

A common operation when working with animations is to ensure that they loop correctly. It is important that the animation clip representing the walk cycle begins and ends in a similar pose (for example, with the right foot on the ground) to ensure that there is no foot sliding or weird jerky motions. The Animator component provides convenient tools for this. Animation clips can loop based on pose, rotation, and position.



As multi selection is not supported (yet) on animation settings of multiple selected files, repeat this step for each animation clip that needs loop play, such as running, running with sword, walking, idle, and so forth.

If you drag the **Start** or the **End** handles (blue markers) of the animation clip and place them at different times on the scale, you will see the looping curves for all of the parameters based on which it is possible to loop. For example, if you place the **Start/End** markers at a moment where the curve for the property is green, it is more likely that the clip will loop properly. The **loop match** indicator (green/orange/red circle) will show how well the looping matches the selected animation time range:



Once the **loop match** indicator is green, enabling **Loop Pose** will make sure that the looping animation will be gap free.



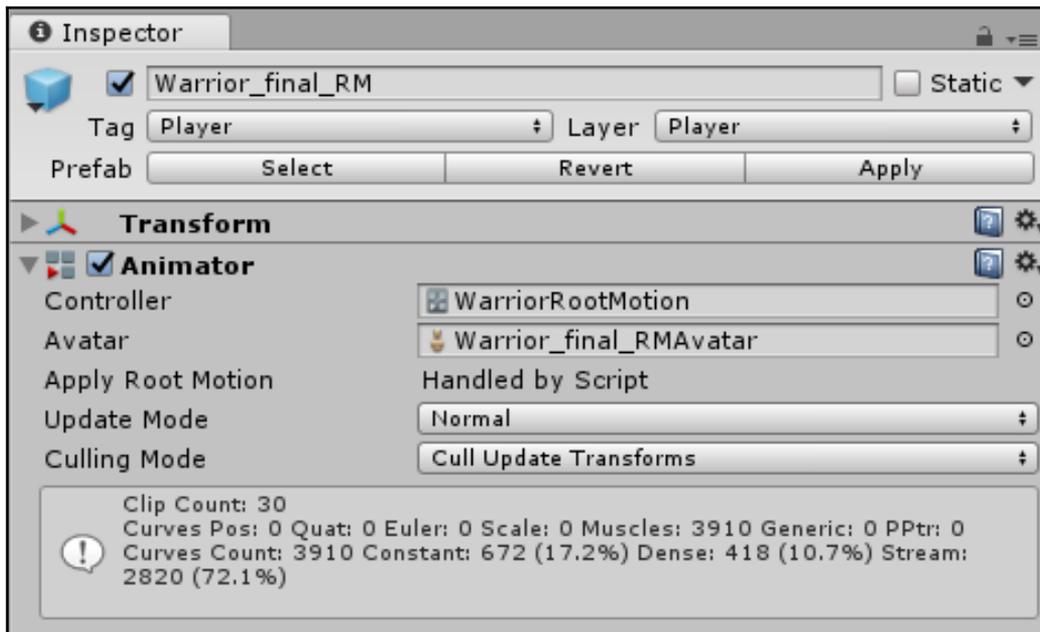
One-shot sequence animation clips don't need anything special to tweak.

Scaling the model

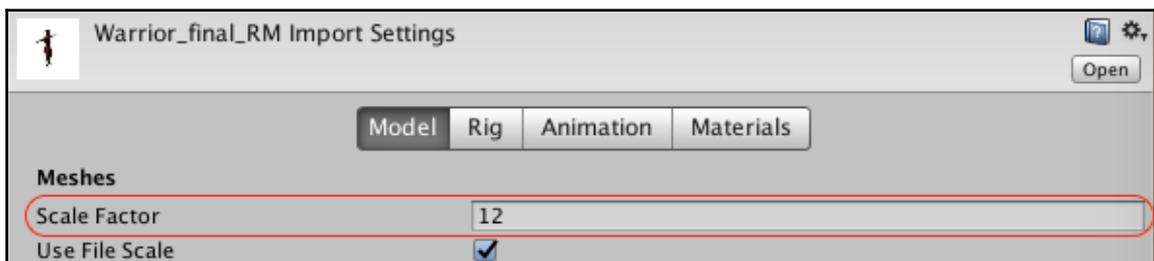
Now that we have set the import options for all the files in the folder, we can finally add our model base rig by dragging the `Warrior_final_RM.fbx` file into the **Scene** view. Look at the **Inspector** and change the position for the **Transform** to `0,0,-5` to have it closer to the camera. Now, let's create a simple plane below the character, go to the main menu **GameObject | 3D Object | Plane**. Translate the plane you created to the global coordinates `-0,0,-5` as well.

We can see that the character is still not facing the camera; we can see his back! To resolve this and turn the character to face the camera, we will set the Y rotation of the **Transform** to `180`.

In the **Inspector** panel for the newly created GameObject, an **Animator** component is automatically added because the source FBX model is imported as a **Generic** or a **Humanoid Rig**:



As you can see, now the model is very small because the scale used by the modeler was too small. Switch again in the **Model** tab and adjust the scale by putting 12 instead of 1 as the **Scale Factor**, then click on **Apply**:



Create a standard cube in the scene, place it inside the character, and then scale it to 1,2,1 by changing the values in the **Inspector**. Finally, the model gets the correct scale with the Unity standard scale, where 1 is supposed to be 1 meter; the character now is almost 2 units (meters) tall:



Check the ready-made scene `Chapter5-6-7/chapter 5.unity` in the book's code to find this example.

Understanding Animator

Now, let's look at the two main parts that make up the animation system in detail:

- The `Animator` component, attached automatically to every `GameObject` created by importing an animated `.fbx` model, is responsible for driving the correct animations to the rig
- The `Animator controller` file, usually associated with the `Animator` component

Animator component

As we have seen before, a `GameObject` dragged into the scene that has an avatar will also have an `Animator` component automatically added. This component is the link between the character mesh and its logic behavior and animations.

The `Animator` component must always refer to an `Animator Controller` file that is used to set up the behavior of the character. This includes: state machines, sub-state machines, Blend Trees, and events that can be controlled by scripts.

Animator component properties

The following explains the properties of the `Animator` component:

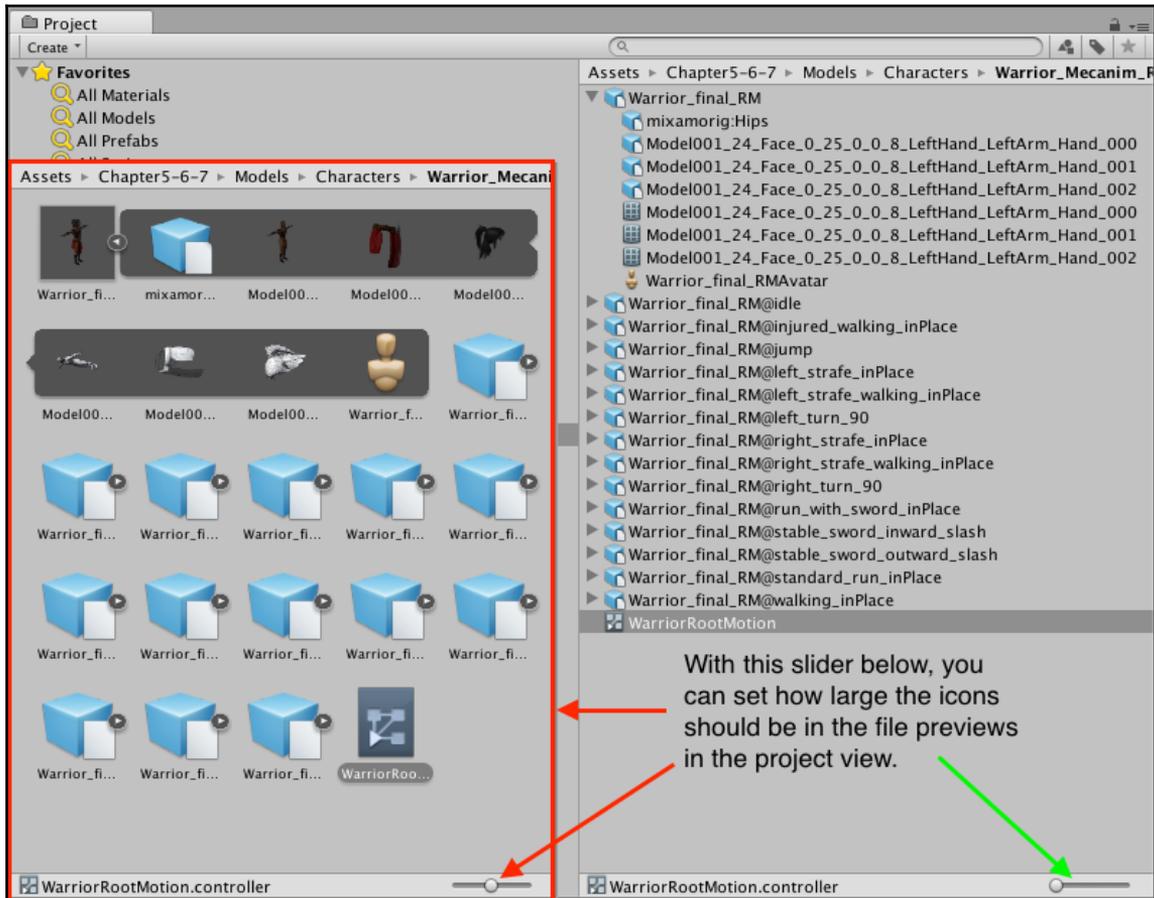
- **Controller:** This is the `Animator controller`, which will drive the current `Animator`.
- **Avatar:** This is the avatar linked to this `Animator`.

- **Apply Root Motion:** This enables you to let the animation root motion control a character's movement in space. When a class attached to a `GameObject` with the `Animator` component overrides the `OnAnimatorMove()` method, this option will be automatically set to **Handled by Script**.
- **Update Mode:** This refers to how the animation behaves with physics:
 - **Animate Physics:** The animation will react to physics simulation when rigid bodies are attached to character skeleton bones.
 - **Normal:** No physics interaction.
 - **Unscaled time:** This is often useful. The animator will be updated in sync with the `Update` call, but the animator's speed ignores the current timescale and animates at 100% speed regardless. This is used to animate a UI system at normal speed, while using modified timescales for special FX or to pause the execution of the game.
- **Culling Mode:** This refers to the culling mode for animations:
 - **Always animate:** This will always animate and doesn't do any culling based on renderers.
 - **Cull Update Transform:** When the renderers are invisible, only root motion is animated. All the other body parts will remain static while the character is invisible.
 - **Cull completely:** Animation is completely disabled when the renderers are not visible.

Animator controller

You can view and set up character behaviour from the **Animator Controller** view (Menu: **Window | Animator Controller**). An Animator Controller can be created from the **Project** view (Menu: **Create | Animator Controller**).

This creates a `.controller` asset on disk, which looks like the following screenshot in the project browser:



The Animator Controller asset on disk after the state machine has been set up

The Animator window

The **Animator** window is made up of the following parts:

- Animation Layer Widget (in the top-left corner of the window)
- Event Parameters Widget (besides the Animation Layer)
- Node diagram visualization of the state machine itself



The Animator Controller window will always show the most recent state machine (from the selected `.controller` asset or from the selected GameObject in the **Hierarchy** view with an animator component aboard).

Animator state machine

A character usually has many different animations for the different actions to be performed in a game. For example, it may walk and raise its arms with a sword to hit an enemy. Controlling when these animations are played back is potentially quite a complicated scripting task. The best technique to use is known as a state machine. Animator uses a state machine to simplify the control and sequencing of a whole pool of a character's animations.

Understanding the state machine

The idea is that a character will perform a particular action at a given time. The available actions typically include things such as staying idle, walking, running, and fighting. These actions are referred to as states, in the sense that the character is in a state in which it is walking, idling, or whatever. In general, a character will have restrictions on the next state it can go into rather than being able to switch immediately from one state to another. In some cases, the main states will be just, for example, **Grounded** or **Airborne**, to specify states that are usually more complex and contain many animation clips (see the next part, about Blend Trees).

For example, a running sword hit can only be made when a character is already running and not when it is at a standstill; so it should never switch straight from the Idle state to the running hit state. The options for the next state a character can enter from its current state are referred to as state *transitions*. Taken together, the set of states, the set of transitions, and the variable to remember the current state will form a state machine for the assigned avatar, or better, its `Animator Controller`.

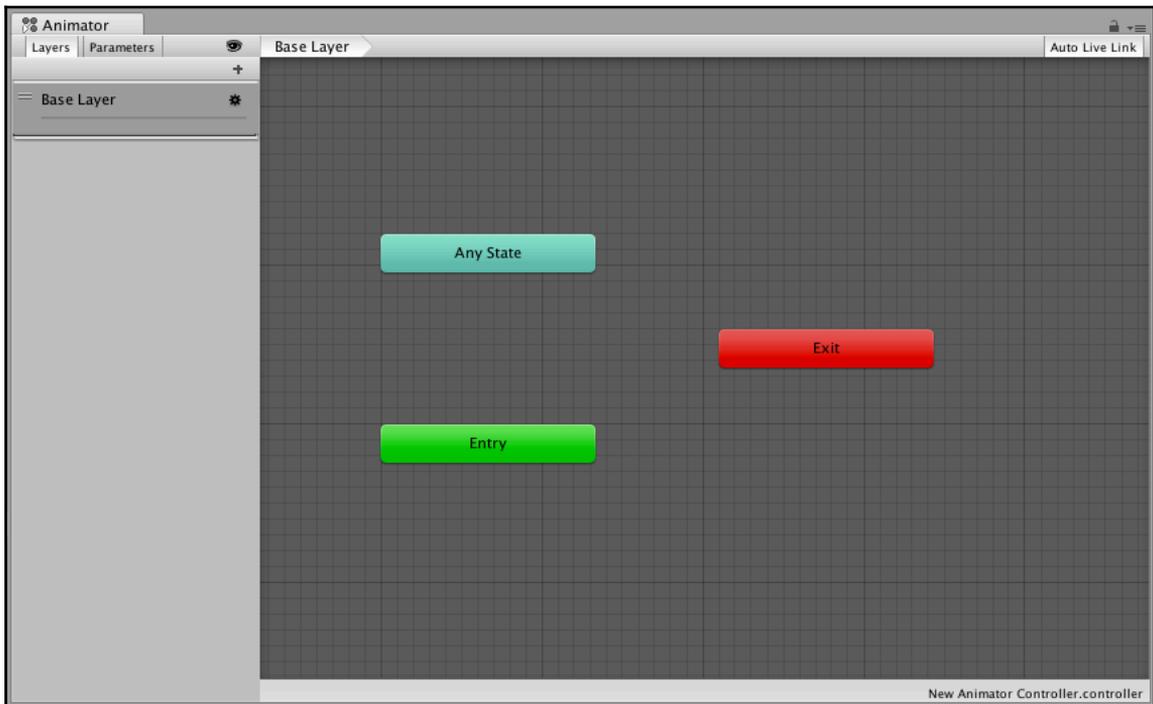
The states and transitions of a state machine are represented using a graph diagram, where the nodes represent the states and the arrows represent the transitions. You can think of the current state as being a marker that is placed on one of the nodes and can then only step to another node along one of the arrows in the diagram.

The good thing about using state machines for animation is that they can be designed and updated quickly with a relatively small amount of code being involved. Each state has an animation sequence associated with it that will play whenever the machine is in that state.

This enables an animator or designer to define the possible sequences of character actions and animations without being concerned about how the code will work, enhancing the development process workflow.

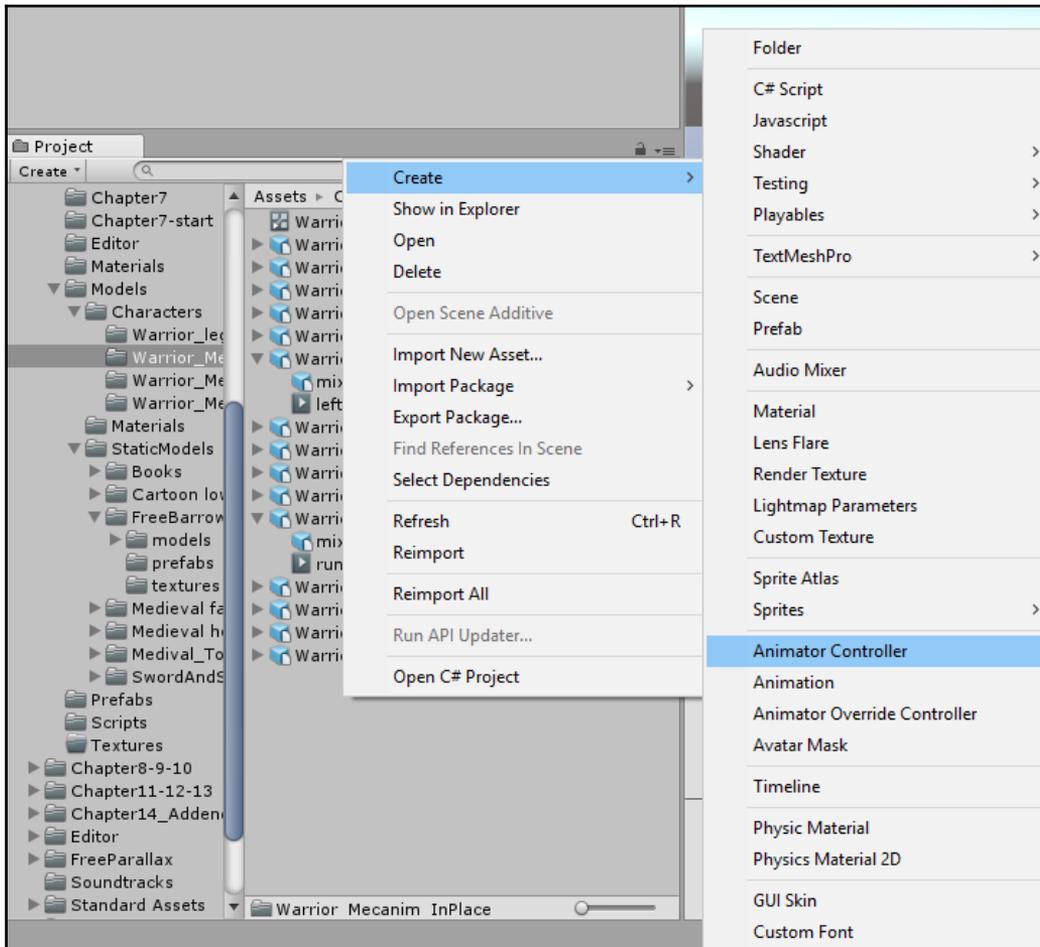
Controlling a character's behavior with state machines

State machines provide a way to overview all of the animation clips related to a particular character and allow various events in the game (for example, user input) to trigger different animations. Animation state machines saved into the Animator Controller file can be edited in the `Animator` window. By default, the **Animator** window is hidden. This window can be shown by going to the top menu—**Windows | Animator**. When no controller is loaded or when we created an empty one, will look like the following:



State machines consist of States, Transitions, and Events, and smaller substate machines can be used as components in larger machines. The animator state machine editor is a powerful tool that allows us to control a character with hardly any coding.

First, we will create the new `Animator Controller` file. Right-click the **Project** view | **Create New** | **Animator Controller**:



Name it properly so that we will know that this is our player hero controller for InPlace animation, for example, `WarriorController_InPlace` or `WarriorInPlaceController`.

Before you start to working on it, we will drag this file to the corresponding slot in the Animator component of the model we just dragged into the scene. Now, double-click on the file or on the Controller spot of the component; the Animator state machine editor tab will show up and open the file. As you can see, the sheet is almost empty. The starting point of a controller has an entrypoint and a default state (in orange), which is automatically connected to.

Animation states

Animation states are the basic pieces of an animation state machine. Each state contains an individual animation sequence (or Blend Tree) that will play while the character is in that specific state. When an event in the game triggers a state transition, the character will be left in a new state whose animation sequence will then take over.

When you select a state in the Animator Controller, you will see the properties for that state in the inspector (see the following screenshot), which are as follows:

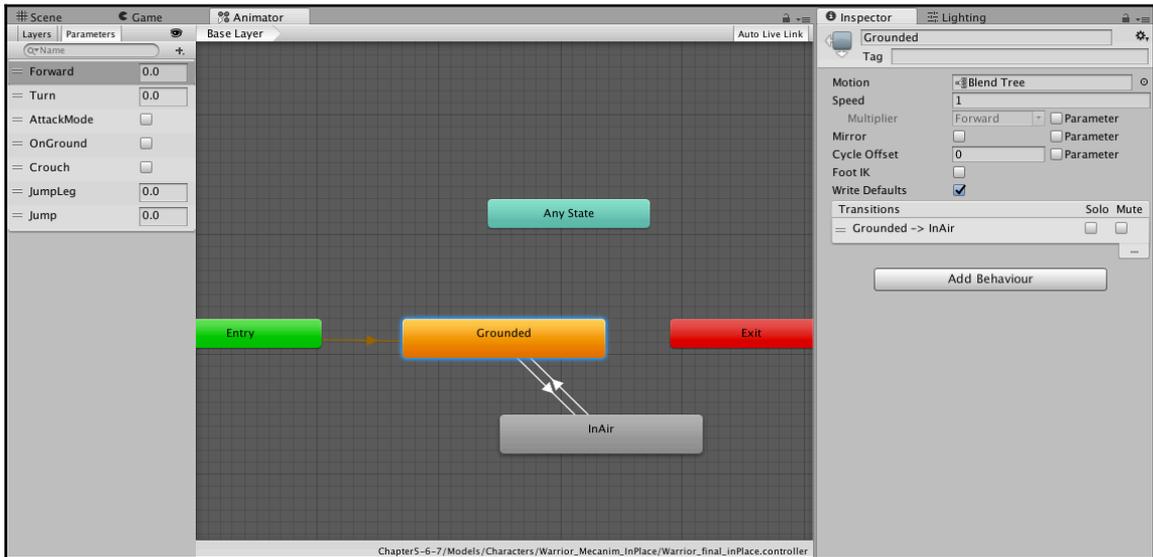
- **Speed:** This is the default speed of the animation
- **Motion:** This is the animation clip assigned to the state
- **Foot IK:** This refers to whether Foot IK should be respected for this state
- **Transitions:** This is the list of transitions originating from this state



The default state, displayed in brown in the diagram, is the starting state for the machine, which will go into this state when it is activated.

You can change the default state, if necessary, by right-clicking on another state and selecting **Set As Default** from the context menu. The **Solo** and **Mute** checkboxes on each transition are used to control the behavior of animation previews. A new state can be added in the diagram by right-clicking on an empty space and selecting **Create State | Empty** from the context menu. You can also drag an animation clip directly into the Animator Controller window to create an animation state to play that animation. States can also contain Blend Trees.

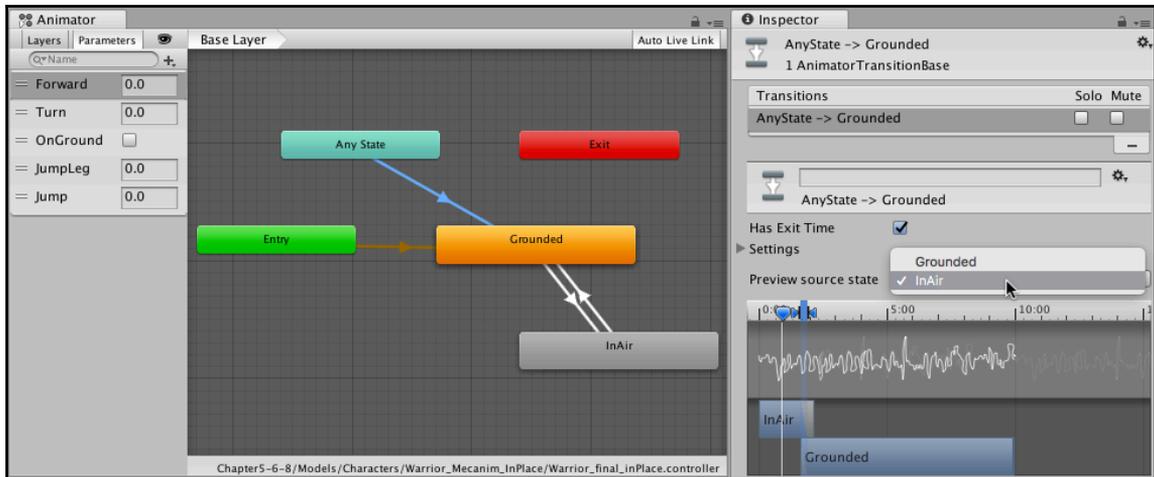
The **Grounded** state that we selected in the preceding picture will be explained later in this chapter. On the right-hand side, you can see the **Inspector** displaying the state properties, as shown:



What's the Any State state?

The **Any State** state is a special state that is always present, such as the **Entry** and **Exit** states. They exist for when you want to switch to a specific state regardless of the current state. This is a quite useful way of adding the same outward transition to all the states in your machine. You can preview different source states for a transition by changing the **Preview source state** pulldown, like can be seen in the next screenshot.

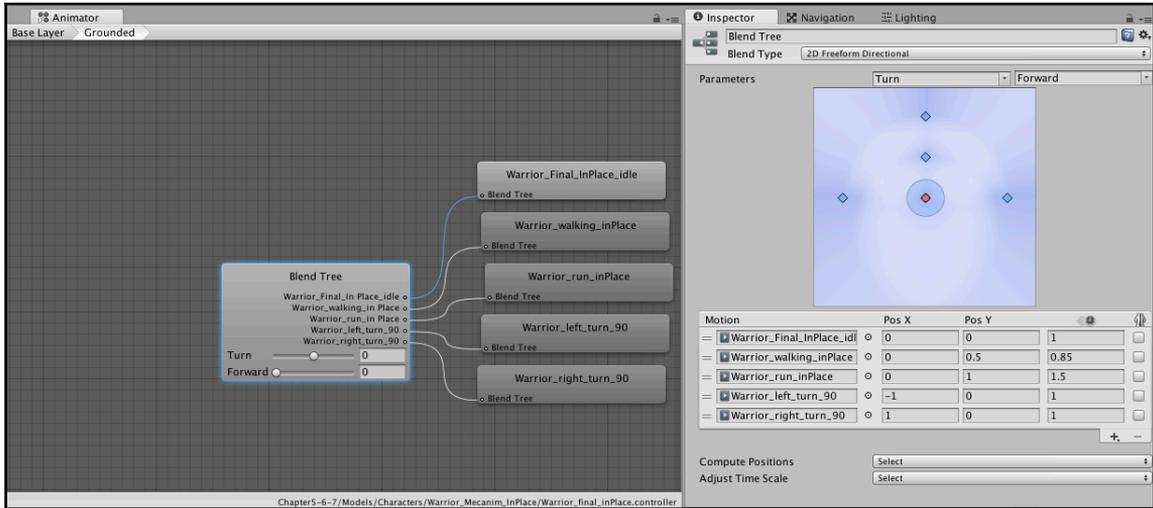
The **Any State->Grounded** transition is selected in the diagram so that you can see the transition properties on the right-hand side in the **Inspector** panel:



Our first Blend Tree state

Now we are ready to create a new state by right-clicking on the empty part of the chart, selecting **Empty State**, and naming it **Grounded**. Now, right-click on it and select **Create new Blend Tree in State** from the context menu. As you can see, this state has an orange color because it was the first one created and is automatically tied to the **Entry** state. Double-click on the **Grounded** state to open now Blend Tree's visualization. The editor will now show the content of the Blend Tree.

We will choose 2D Freeform Directional as the **Blend Type** in the pull-down menu and setup 5 motions in place for the Blend Tree, like in the following image:

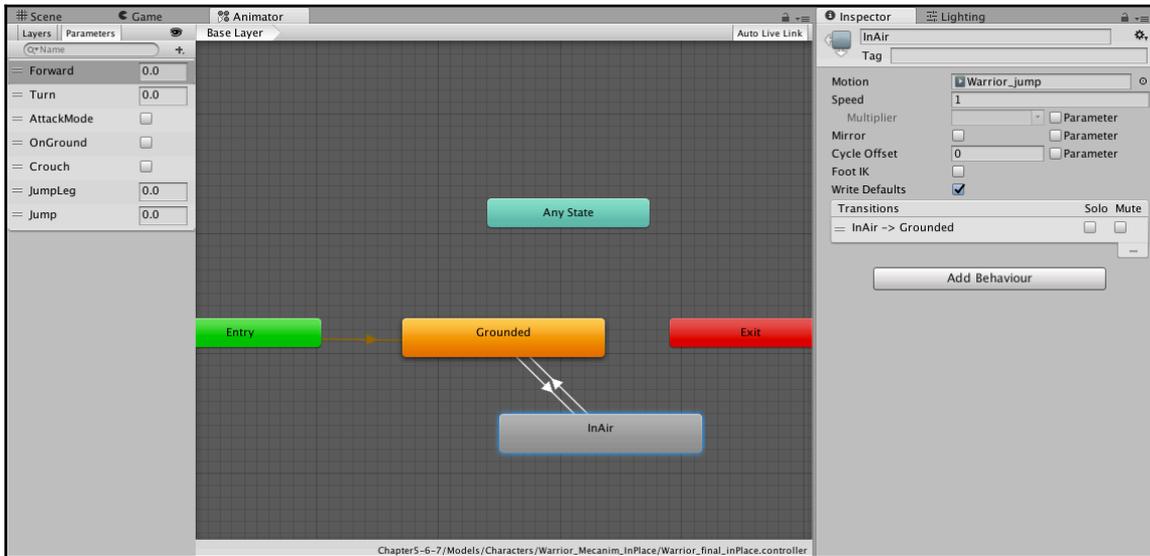


The state will manage to play the correct motion animation when the Turn and Forward parameters change with user inputs. We will get deeper into Blend Tree later in this chapter when looking at the final implementation that uses Root Motion.

Conclusions

Double-click on the empty part of the blend tree diagram to go back to the main state-machine scheme. Then, create another empty state and name it `InAir`.

Then, drag the `Warrior_jump` animation clip in its Motion slot, like in the following screenshot:



Finally, we will connect back and forth the **Grounded** and the **InAir** states and set the **Jump** parameter for the two conditions to change the current state, by selecting the transition arrows and adding the condition one after the other.

Now, if you press play, you should see the character performing the **Idle** animation forever played looping. Switch back to the `Animator` window; we need to connect the two states in order to play the walking animation as well. Right-click on the **Idle** state and connect it to the **Moving** state. Now, save (`Ctrl + S`) your controller and switch back to the **Game** tab, press Play, and see what happens. The character should first play out the idle animation, and when finished, start to play the walking animation in an infinite loop. This happens because no condition was set for the **Idle->Moving** transition. Click on the arrow line connecting the two states and see that no condition was specified in the **Inspector**. To add the condition, we need first to create some Animator parameters. We have given a very quick look to Animator with a in-place animation setup, let's now see how to drive those parameters with user inputs through code!

Controlling Animator through code with parameters

Animator's parameters are variables that are defined within the animation system but can also be read and store values from scripts. Default parameter values can be set up using the **Parameters** widget in the bottom-left corner of the **Animator** window when you create them. They can be of the following four types:

- **Vector**: A point in space (x,y,z)
- **Int**: An integer (whole) number
- **Float**: A number with a fractional part
- **Bool**: This refers to true or false

Parameters can be assigned values from a script using the methods in the Animator class: `SetVector`, `SetFloat`, `SetInt`, and `SetBool`, while they can be accessed (read) with the `GetVector`, `GetFloat`, `GetInt`, and `GetBool` methods. Create a float type parameter and call it `Speed`. As you can see, the default value can be specified, so leave it as `0.0`. Now, press the connecting line and, finally, note that you can choose the parameter `Speed` from the drop-down menu. Choose *when value is greater than* and put `0.1` in the box.

The character stays in **Idle** forever again, but is waiting for scripts (user input in this case) that change the speed value to make the character walk. Create a float parameter; we will call it `Turn`. Create a new component script by clicking on the **Add Component** button after the last component in the `Warrior GameObject` in scene. Name the class `SimpleThirdPersonCharacter`.

Let's kick off by declaring a private variable that we can utilize throughout the script, referring to our Animator component:

```
private Animator currentAnimator;
```

Define a float variable to store `runMultiplier` to apply it when the *Shift* key is kept pressed and specify a default value (walking) of `0.5f`:

```
float runMultiplier = 0.5f;
```

We could have declared this variable as public, as we previously did, in [Chapter 3, Creating and Setting Game Assets](#), to allow the dragging of the respective `GameObject` from the **Scene Hierarchy** into our Component slot. For learning purposes, we will now retrieve a component at runtime instead, without the need to drag the component on the script, as we usually do.

Additionally, the component we want to retrieve is on the same `GameObject` that this script will be executed from.

This can be useful when we want to set the value of the variable at runtime or because the value might change and we want to retrieve a particular object component seeking in the scene by name or by tag (refer to [Chapter 8, AI, NPC, and Further Scripting](#)).

Unity allows us to make everything from code or with a lot of help from the editor; it is your choice. Whether you are a skilled coder or a graphic designer, you choose the path you prefer for these kinds of things. As the `Animator` component, we want to track with this variable on to the same `GameObject` that is driving our C# class component; it is just simpler this way.

Define these variables by writing the following code at the top of the `SimpleThirdPersonCharacter` class:

```
using UnityEngine;
public class SimpleThirdPersonCharacter : MonoBehaviour {
    Animator currentAnimator;
    float runMultiplier = 0.5f;

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

Next, we will write the `Update()` method:

```
// Update is called once per frame
void Update ()
{
    //if(Input.GetAxis("Horizontal")>0)
    float verticalForce = Input.GetAxis("Vertical");
    float horizontalForce = Input.GetAxis("Horizontal");
    float absVertForce = Mathf.Abs(verticalForce);

    // Run if Shift is kept pressed
    if (Input.GetKeyDown(KeyCode.LeftShift) ||
        Input.GetKeyDown(KeyCode.RightShift) ) runMultiplier = 1.0f;
    if (Input.GetKeyUp(KeyCode.LeftShift) ||
        Input.GetKeyUp(KeyCode.RightShift) ) runMultiplier = 0.5f;

    //Debug.Log("user input: horizontal="+horizontalForce+"
    vertical="+ verticalForce);

    currentAnimator.SetFloat("Forward", absVertForce*runMultiplier);
}
```

```
currentAnimator.SetFloat("Turn", horizontalForce);

// Because we don't use RootMotion on this prefab, we need to
// both translate and rotate him directly from user input data
transform.Translate( transform.InverseTransformVector(
transform.forward *( verticalForce * runMultiplier *0.075f) );
if(Mathf.Abs(horizontalForce)>0.1f) transform.Rotate
(transform.up, horizontalForce*2f);
}
```

This code will read in arrow keys and *Shift* key to apply a force to the character directly. Press play and you should see the character walk and move forward when you press the UP arrow key. We have just given a quick look to the old legacy animation system and on how to import .fbx file for the new Animator, and how to drive quickly an Animator setup with in-place animations. Now, it's time to dive into the best option for modern character animation with Animator: Root-Motion animation.

Root-motion animations

You can map your humanoid character to essential bones and share this information together with the Animation Controller on different characters if the class of characters allows it; you will save a lot of time by sharing the same avatar among characters.

This is a good practice that will save the developers and artists lot of time, especially when many different model rigs, share the same animation sets. Usually, a `player` class and enemy and **non-playing characters** (NPC) classes will be different and customized but, often, the same animator and the same third-person controller are used for different or similar purposes. We will see this in Chapter 8, *AI, NPC, and Further Scripting*, when making AI and NPC.

Using a better approach – Root motion

Root Motion allows animators to control the quantity of movement of each animation frame. The output result is a much more weighted animation feeling, as seen in games such as *Assassin's Creed* or *Legend of Zelda: Wind Waker*. See this short video tutorial on how to perform such player movements: <https://youtu.be/zy9E-w4QM2o>.

Root Motion animation can be implemented in Unity with two different ways:

- Root Motion handled by script
- Root Motion animation controlled

For our game, we will use a modified version of the `ThirdPersonCharacter` class in the Standard Assets, which overrides the Root Motion method to implement the control of the Root Motion from scripting. When this method is overridden in the controller class code, the checkbox option `Apply Root Motion` in the Animator component will disappear. Instead, you will see a message stating: *handled by script*. This is very useful and gives better results in modern games, because you will not need to tweak movement speed with animation speed to find a proper values' setup to make it look good.

Look in the book's `Chapter5-6-7/Characters/Warrior_Mecanim_RootMotion` code folder. This is a root motion version of our hero with some more animations available (some of them will be used in the next chapter, where we will see our hero fighting), we will use/set these animations for root motion animation.



In our project, we imported the Characters module of the Standard Assets, containing the assets setting up the project in `Chapter 2, Prototyping and Scripting Basics`; if you decide to split the 2D game example and the 3D game into two different Unity projects and this is a new project instead, please reimport this Asset Package from the top menu: **Assets | Import | Characters**.

- Repeat the steps in *Configuring the avatar* for the `Warrior_final_RM` base rig model.
- Setup quickly its relative animation clips by clicking on **Update reference clips** in the **Rig** tab to automatically map all the animation for this model to the correct avatar.
- Create a modified version of the `Animator Controller` used for the `ThirdPersonController` prefab that can be found in the Unity Standard Assets.

- Clone the file: `ThirdPersonAnimatorController.controller` and rename the clone in `WarriorRootMotion.controller`. Note that the **Project** view hides the file extension in the elements listed, but you can see it in its bottom bar.
- Assign the cloned `Animator Controller` to the **Animator** component attached to our hero character `GameObject`, then, open the `Animator` window and be sure to have the hero `GameObject` selected, to load up the associated controller file assigned to its **Animator** component.
- Remove the default **Blend** parameter by right-clicking on it and then deleting it.



You can also double click directly the controller in the component slot, this action will automatically open the **Animator** window, if it was previously closed, with the correct controller file loaded up.

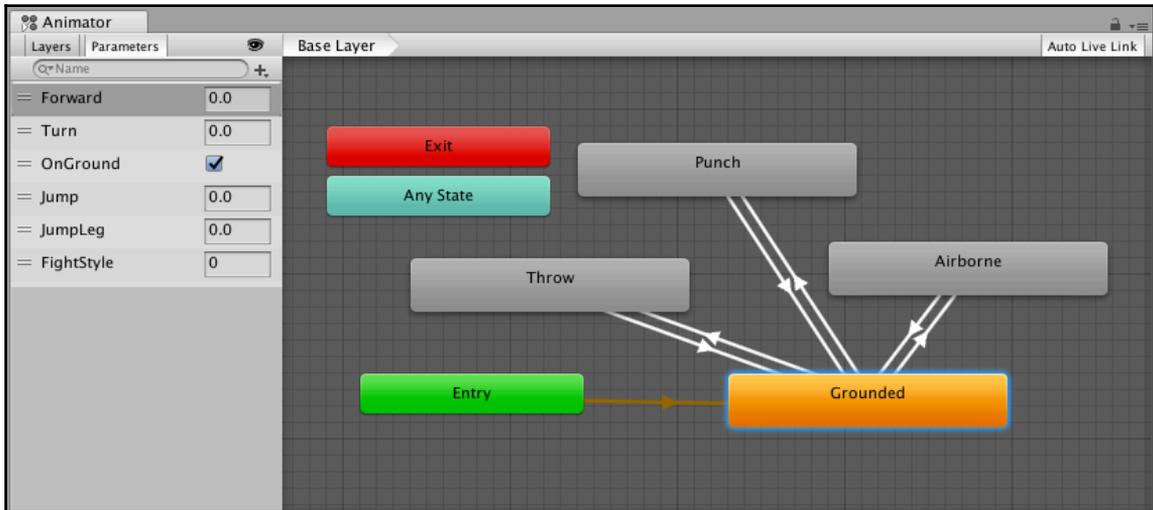
Then, we will add the following parameters:

1. Add a `float` parameter: `Forward`
2. Add a `float` parameter: `Turn`
3. Add a `bool` parameter: `OnGround`
4. Add a `int` parameter: `FightStyle`
5. Add another `float` parameter: `JumpLeg`
6. Add another `float` parameter: `Jump`

What we are building is very similar to the Unity Standard Assets `ThirdPersonCharacter`. We could also have copied the standard assets `ThirdPersonCharacter` animator file and started modifying this one, but for the purpose of learning, we will make it from scratch and also modify it in [Chapter 8, AI, NPC, and Further Scripting](#):

1. Add a new state and call it `Grounded`
2. Create a new `BlendTree` from the `Grounded` state
3. Add a new state and call it `Airborne`
4. Create a new `BlendTree` from the `airborne` state
5. Add another two new states and call it **Punch** and **Throw**

Connect the states to create a return way for each state as well, as shown in this screenshot:

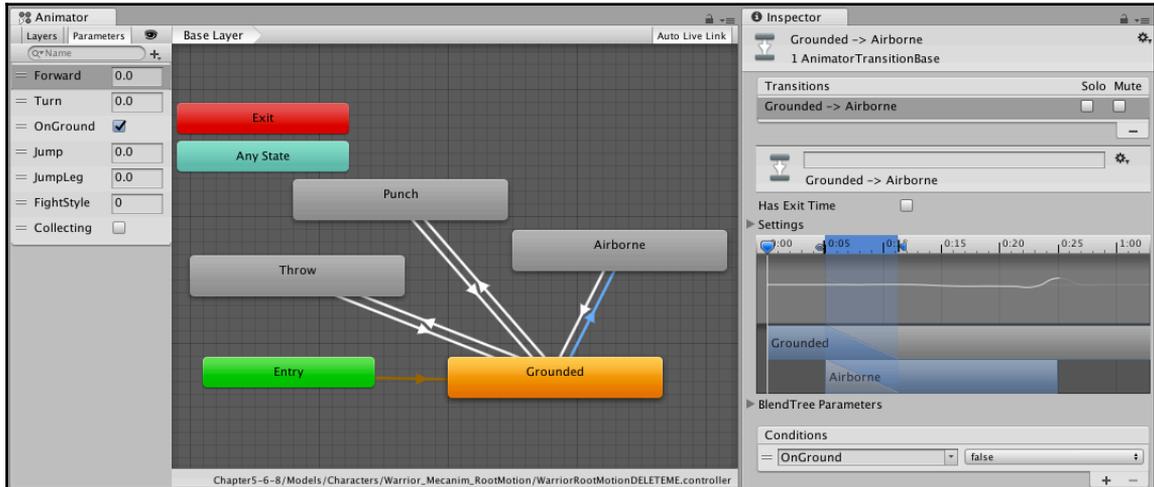


Animation transitions

We will now connect the **Grounded** Blend Tree we have just created to the **Airborne** state. Animation transitions describe the logic conditions of when/why an animation state should change and what happens when you switch from one animation state to another. There can be only one transition active at a given time in a running state machine.

Setting the transitions between states

Choosing the transition will show its properties in the **Inspector**, as follows:



In the **Inspector**, after the animation timeline under the **Settings**, look at the **Conditions** section; this is where we specify when the transitions will be triggered. A condition is made of one or more parameters:

- Instead of a parameter, you can also use the **Exit Time** setting and specify a number that represents the normalized time of the source state (for example, 0.95 means the transition will be triggered when we've played the source clip 95% through)
- A conditional predicate, if needed (for example, less/greater for float numbers and true/false for Boolean)
- A parameter value (if needed)

You can adjust the transition between the two animation clips by dragging the start and end cues of the overlap.

Blend Trees

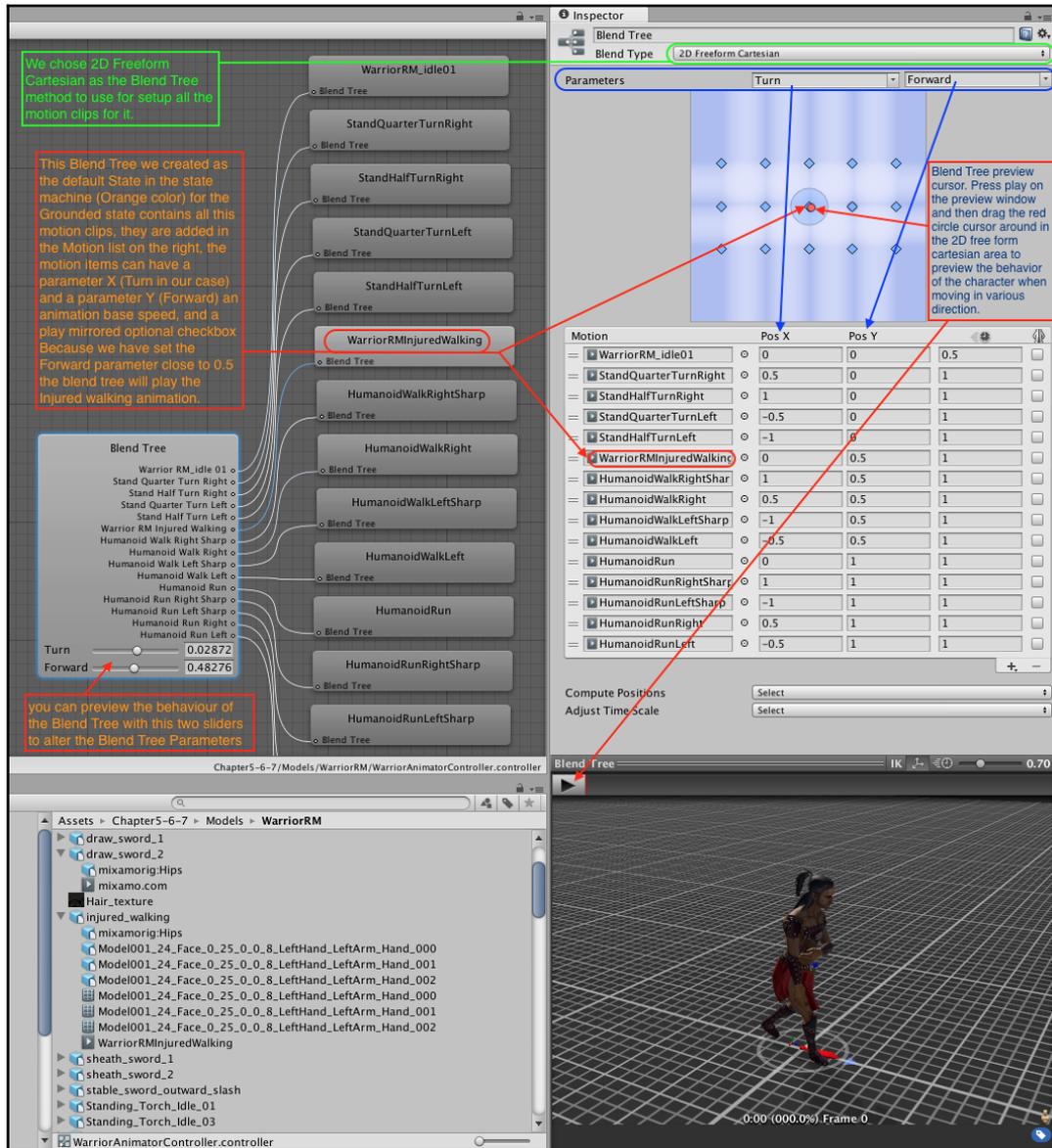
A common task in game animation is to perform a correct transition between two or more similar sequences. Perhaps the best-known example is the transition between walking and running animations according to the character's speed (that is, running movements are not just faster versions of walking movements, so they require separate animation clips).

Another example is a character leaning to the left or right as they turn while running. The most important detail of the transition is to ensure that it happens smoothly, without a noticeable jerk where the animations are switched. Blend Trees are Animator's method of allowing one animation to be blended smoothly with another. By tracking the bone movements of the two animations precisely, Animator can incorporate parts of both to varying degrees. The amount that each of the two animation clips contributes to the final effect is controlled using a blending parameter, which is just one of the numeric animation parameters associated with a character. To make for a smooth transition, Animator requires the two clips that are to be blended to be aligned so that the corresponding movements take place at the same points in normalized time.

For example, walking and running animations can be aligned so that the individual footfalls take place at the same points in normalized time, even though the running cycle is faster in real time (the left foot hits at 0.0, the right foot at 0.5).

This Blend Tree we created as the default state in the state machine (orange color) for the `Grounded` state contains all the motion clips; they are added in the motion list on the right, the motion items can have an X parameter (turn in our case) and a Y parameter (Forward), an animation base speed, and a play mirrored optional checkbox.

As we have set the forward parameter close to 0.5, the Blend Tree will play the Injured walking animation:



We will choose **2D Freeform Cartesian** as a Blend Tree method to set up all the motion clips for it. This method requires two parameters to work; we will use the **Forward** and **Turn** parameters like for the previous setup.

While playing the animation in the preview area of the inspector for the rigged model, you can preview the behavior of the Blend Tree animation with these two sliders to alter the Blend Tree parameters.

With the Blend Tree preview cursor, you can visually have an idea and preview the behavior by dragging it in the desired direction. Again, press Play on the **Preview** window and then drag the red circle cursor around the **2D free form Cartesian** area to preview the behavior of the character when moving in various directions.

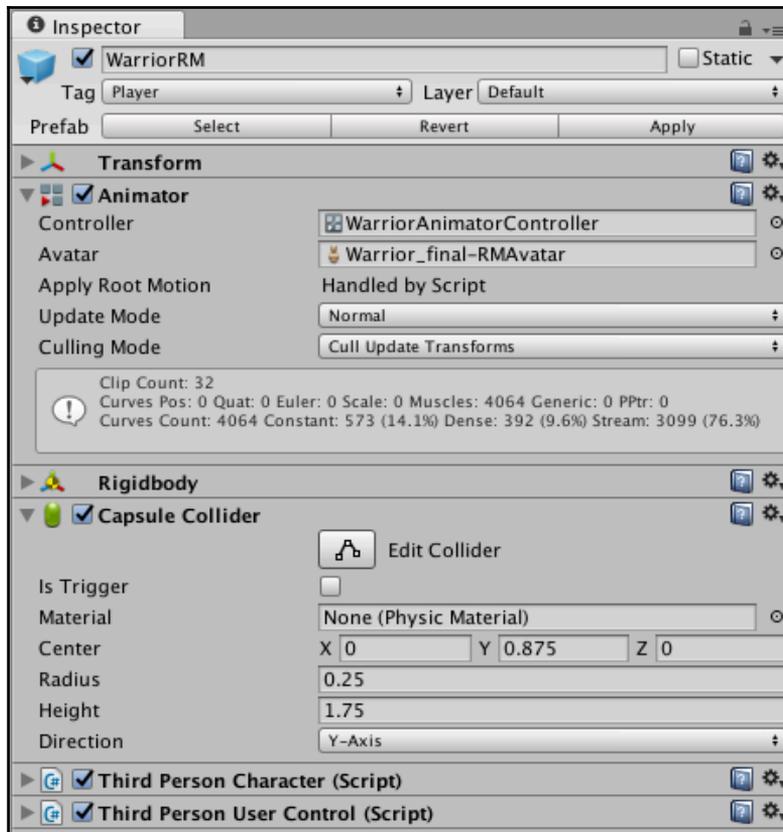
Modifying Unity standard assets classes to implement our playing character

We will take advantage of the basic classes in the Unity standard assets folder (we imported the characters module when we created the project in Chapter 1, *Entering the Third Dimension*), modifying them as per our needs.

We won't take the `ThirdPersonCharacter` prefab as it is, instead, we will build our own following the original guidelines; we will just take advantage of the two main classes for this prefab: the `ThirdPersonCharacter` and `ThirdPersonUserControl` classes. In Chapter 8, *AI, NPC, and Further Scripting*, we will take advantage of **Object Oriented Programming (OOP)** and extend the `ThirdPersonCharacter` for the AI enemies we will create for our game.

First of all, we will replicate almost the same setup of the `ThirdPersonCharacter` prefab in the Standard Assets furnished by Unity Technology by adding a `Rigidbody` and a `Capsule Collider` component, as well as the `ThirdPersonCharacter` and the `ThirdPersonUserController` components to our `Warrior_final_RootMotion` `GameObject` in the scene.

The `WarriorRM` GameObject should look as follows in the **Inspector**:



When this is done, click on **Apply** at the top of the GameObject to store the prefab changes (save). The `ThirdPersonCharacter` class will be good for our purpose in this chapter, but will be changed later and extended to implement the playing hero's fighting scheme and also the enemy AI controller fight in the next chapter. Let's take a look at and modify the `ThirdPersonUserController` class now.



Clicking on the component script slot on our hero will open it in Visual Studio 2017 Community Edition on both Windows 7/10 and macOS (when installed) with the actual project solution loaded. We strongly suggest using Visual Studio to edit the code if you are already familiar with it, as it gives better control, speed, and better real-time debugging tools on both platforms.

The modified version of the class allows you to quit the game by pressing the *Esc* key and also has other small modifications. We will add features to this class in the next chapter. The class is also responsible for the user input that will instruct the `ThirdPersonCustomCharacter` component, which is required, as you can see from the start of the code of the class, give a look at it by open in the editor the file:
`Assets/Chapter5-6-8/Scripts/ThirdPersonCustomUserController.cs`.

Adding the final touches

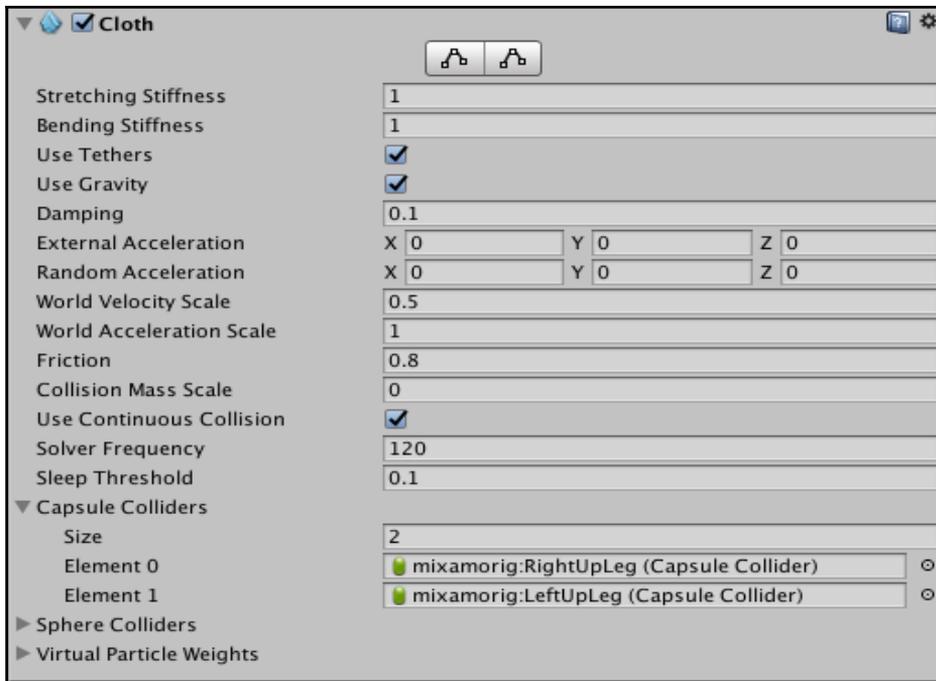
Now, we have a pretty nice movement and animation set for our hero, but still, for how the model is designed, we can do it a little better, with the help of Cloth Physics Simulation. Also, we will write a custom Inverse Kinematic script to see how we can add forced postures for arm bones by programming without using animations.

Cloth simulation – giving life to hero's hair and skirt

To enhance the realism and look of our hero, we might want to let these clothes and the hair blow in the wind and react to character animation. To allow this on a Skinned Mesh Renderer, we will use the Cloth component, as illustrated in the following screenshot.

The Cloth component is a physics-based component found under **Components** | **Physics** | **Cloth** in the main menu, and is meant to provide a realistic feeling and interaction with the cloth part of a character or things such as a flag blowing in the wind or a big medieval window curtain. This component is added to a GameObject with a Skinned Mesh Renderer component attached.

Usually, these GameObjects are part (submeshes) of a character model or may also be a simple subdivided 3D plane (in the top menu: **GameObject** | **3D** | **Plane**):



The Cloth component seen in the Inspector

We will see this directly applied to our hero character's clothes, which is one of the more difficult, but also satisfying parts, of the preparation of our hero character:

1. Select the red skirt part of the character model in the scene editor
2. **Add Component** (or menu Component) | **Physics** | **Cloth**
3. Disable **Gravity** and set **Bend Stiffness** to 1
4. Set damping to 0.1
5. Click on the **Edit Vertex** button placed at the top of the **Cloth** component; this will allow you to set the influence weight for each vertex or group of selected vertices
6. Select the belt and the first skirt vertices, as in the following screenshot, check the **Max Distance** checkbox, and leave the value as 0
7. Select the other vertices of the skirt, check **Max Distance**, and set the value to 0.2

The following screenshot represents the Cloth component in action in the **Scene** view with the vertex weights-painting tool enabled:



Then, browse the hierarchy of the hero character to find the legs and torso:

1. Add a Capsule Collider component on each leg **GameObject**, and another Capsule Collider for the **Spine** **GameObject** in the skeleton **Hierarchy** of the character
2. Resize the collider with the **Inspector** to make it fit the legs shape and the torso/back part of the character

3. Now, put 3 in the Capsule Colliders setter in the **Cloth** component to enable 3 new slots for 3 colliders
4. Drag the 3 skeleton GameObjects with the colliders we just created in the three spots of the Cloth component
5. Press Play; you should see the cloth simulation take effect on the character's skirt

Carry out more test and fine-tuning of the **Max Distance** value for the vertices to obtain a viable result. Be patient; you can go back to editing your prefab again later to achieve a better result if you are not satisfied with it yet.

Now, let's look at how we can play with skeleton bones to override character's animation with Inverse Kinematics.

Inverse Kinematics

Let's take a look at this simple Inverse Kinematic implementation (a modification of the IK implementation found in the official Unity manual) to make the hero able to hold their two hands close to each other in a forced way, as if they are being kept by some kind of dungeon handcuff without the need of a full set of animations with the arm extended and joined together.

Animations are produced by the rotation of the bone joints to values stored in the animation itself. The position of a child joint changes according to the rotation of its parent so that the endpoint of a chain of joints can be determined from the angles and relative positions of the individual joints it contains.

This method of posing a skeleton is known, in character animation, as **Forward Kinematics (FK)**.

Sometimes, it may be useful to look at the task of positioning joints from the opposite point of view; given a chosen position in space, work backwards and find a valid way of orienting the joints so that the endpoint lands at that specific position. This is handy if you want a character to touch an object at a given space point selected by the user or, for example, to keep the two arms together while pointing a pistol in the aim position.

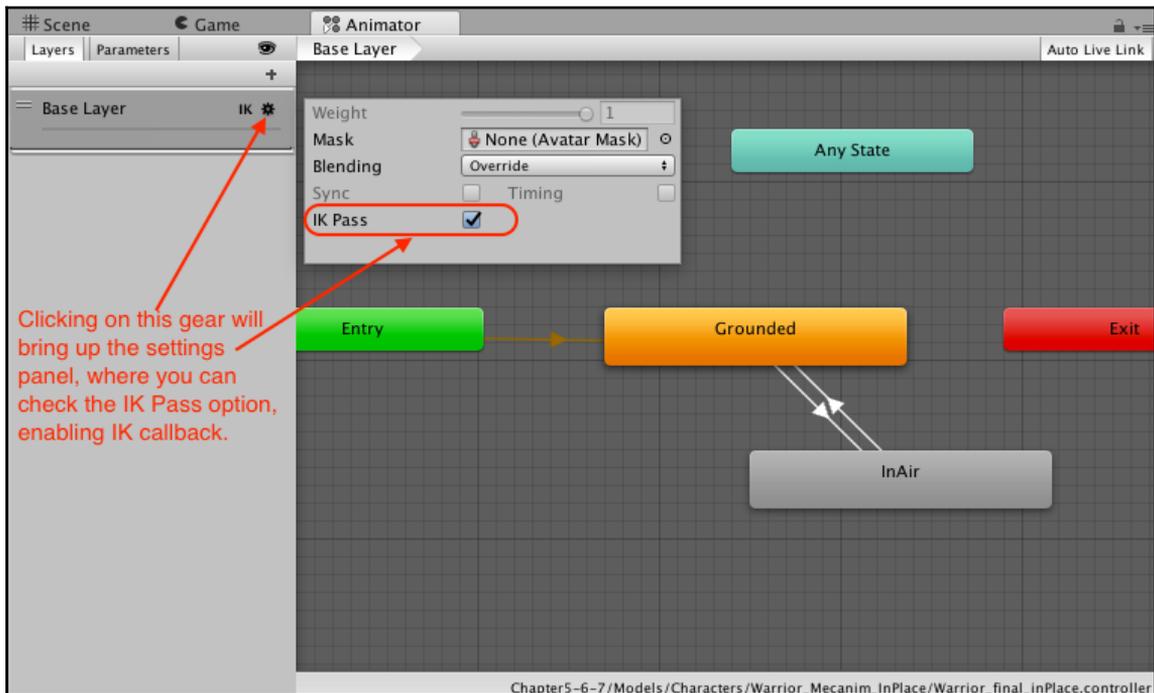
This approach is known as Inverse Kinematics and is well supported in Animator for any humanoid character with a correctly configured avatar:



A character with tied hands. How do we make this work?

To set up IK for a character, you typically want to have objects around the current scene that the character should interact with, and set up the IK through scripting, in particular, using Animator methods such as `SetIKPositionWeight` and `SetIKRotation`.

Starting with our character, who has a valid avatar configured, we will modify the **Base Layer** for its Animator Controller by checking the **IK Pass** checkbox, as in the following screenshot:



Setting the IK Pass checkbox for the Default Layer

Then, we will attach a new component to it; a script that actually takes care of the IK. We will name this script `IKHandController` and type in the following code:

```
using UnityEngine;
using System;
using System.Collections;

[RequireComponent (typeof (Animator))]
public class IKHandController : MonoBehaviour {
    protected Animator theAnimator;
    public bool ikIsActive = false;
    public Transform rightHandT = null;
    public Transform lookObj = null;

    void Start () {
        theAnimator = GetComponent<Animator> ();
    }
}
```

```
    }

    void Update () {
        // We want to switch the IK simulation when the I key is pressed
        if(Input.GetKeyUp(KeyCode.I) ) ikIsActive = !ikIsActive;
    }
}
```

The important part is the override of the `OnAnimatorIK()` method. Adding this code to the class, will enable IK control with scripting:

```
// callback for calculating Animator IK
void OnAnimatorIK()
{
    if(theAnimator) {
        //if the IK is active, set the position and rotation
        directly to the goal.
        if(ikIsActive) {
            // Set the look target position, if one has been
            assigned
            if(lookObj != null) {
                theAnimator.SetLookAtWeight(1);
                theAnimator.SetLookAtPosition(lookObj.position);
            }

            //set position and rotation of both hands where the
            external object is
            if( rightHandT != null)
            {
                theAnimator.SetIKPositionWeight
                (AvatarIKGoal.RightHand,1);
                theAnimator.SetIKRotationWeight
                (AvatarIKGoal.RightHand,1);
                theAnimator.SetIKPositionWeight
                (AvatarIKGoal.LeftHand,1);
                theAnimator.SetIKRotationWeight
                (AvatarIKGoal.LeftHand,1);
                theAnimator.SetIKPosition
                (AvatarIKGoal.RightHand,rightHandT.position);
                theAnimator.SetIKRotation
                (AvatarIKGoal.RightHand,rightHandT.rotation);
                theAnimator.SetIKPosition
                (AvatarIKGoal.LeftHand,rightHandT.position);
                theAnimator.SetIKRotation
                (AvatarIKGoal.LeftHand,rightHandT.rotation);
            }
        }
        //if the IK is not active, set hands position and
```

```
        rotation back to original
    else
    {
        theAnimator.SetIKPositionWeight
            (AvatarIKGoal.RightHand, 0);
        theAnimator.SetIKRotationWeight
            (AvatarIKGoal.RightHand, 0);
        theAnimator.SetIKPositionWeight
            (AvatarIKGoal.LeftHand, 0);
        theAnimator.SetIKRotationWeight
            (AvatarIKGoal.LeftHand, 0);
        theAnimator.SetLookAtWeight (0);
    }
}
}
```

As we don't want the character to grab the entire object with his hand, we position a target object where the hand should be on the cylinder and rotate it accordingly. This sphere `GameObject` should then be placed as the right-hand object reference property of the `IKCtrl` component by dragging it into the Right Hand Obj slot of the component in the **Inspector**. Observe the character looking at the grab object, with the head as you press the *F* key on the keyboard.

In the same way, we can have our characters grab objects, hold a sword, or keep their head looking at a certain point while walking in a different direction, and many other cool things, all thanks to IK solvers. An example of this being used in a video game is *Hitman*, where part of the gameplay involves hiding dead bodies by dragging them around by their hand.

Summary

In this chapter, we looked into Unity's animation system, the older, legacy one, and the new one based on `Animator`. We saw the difference between the Root Motion and non-Root Motion approaches to animate a character with `Animator` and discovered how to set up both basic and complex state-machines to control character animation behavior, as well as how IK pass in a layer can allow us to control a part of the skeleton through scripting.

In the next chapter, we will dive into the creation of a third-person adventure game called *The Devil Island*. You will learn how to use the terrain tools to create a tropical-like island, complete with vegetation, hills, picks, canyons and a small hidden lake! Once you have created your island, you'll add your player character prefab and take a stroll around your mediterranean paradise!

6

Creating the Environment

When building your 3D world, you'll utilize two different types of environments:

- **Geometry:** Made of buildings, scenery models such as rocks, trees, and bushes, and props usually built in a third-party 3D modeling application
- **Terrains:** Created using the Unity Terrain Editor

This part of game development is called **level building**, and it usually comes after a level has been designed (usually on paper) and detailed in hand-drawn sketches, screenplay scripts, and so on. This part is very important because it describes the world that *you see* in a graphic manner, and also the environment in which the player will be *living* and acting.

In this chapter, we'll look at the use of the Terrain Editor, and how we can use its toolset to build the environment for our game, including sculpting, geometry, and painting with textures to create an entire island terrain.

We will be looking into the following specific topics:

- Working with the terrain tools to build an island
- Basic scene lighting and using the new procedural skybox
- Using the built-in water prefab to create a sea and a small lake
- Using the **AudioSource** component to implement environmental ambient sounds
- Adding the player prefab and a ready-made third person camera rig to walk around the island

Before we start making use of the terrain tools to build the island part of our environment, let's have an overview of what we are going to create in its entirety, in order to help you keep the final goal in mind as you work through this book.

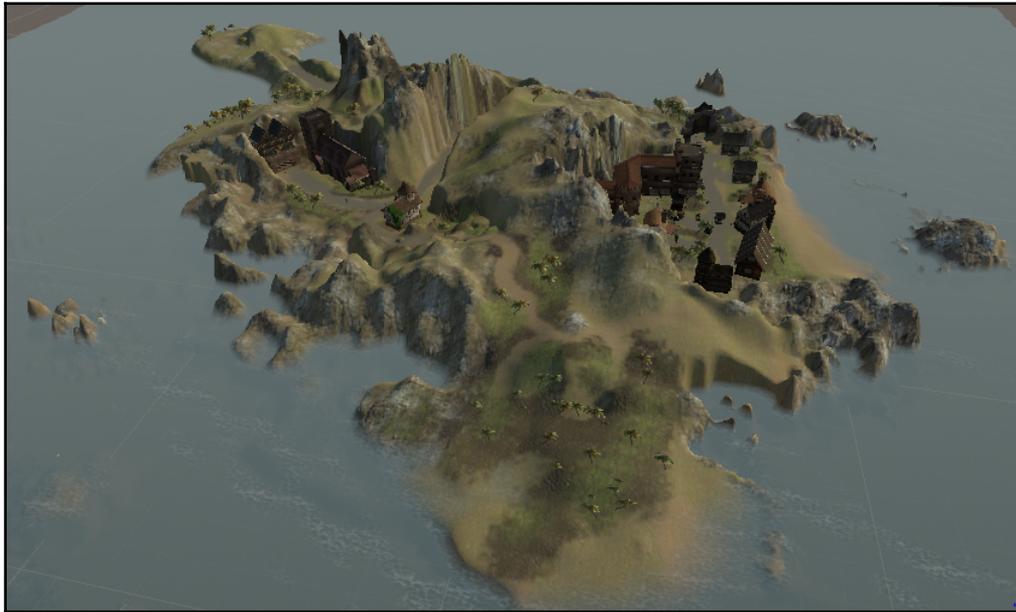
Designing the game

In order to prepare you for the game you will make over the course of this book, let's first get a good idea of what you're going to make, so that you can better understand what you're doing at each step along the way and why.

The game we will design is called: The Devil Island, a third person adventure where the basic concept is that your character has been imprisoned on an (almost) deserted island and locked in an underground prison. You could escape the jail by fighting a drunk guard, but you don't know how to leave the island and avoid the other guards. Only an old, free man, a **non-playing character** (NPC), can help you in the quest, so be prepared to face a lot of guards trying to imprison you again!

Your first goal will be to heal your wounds by receiving food and medicines after talking to the old man. When this is done, you will finally be able to fight your way out. In order to solve the game, you need to knock out the guards to be able to explore the island in search of what the old man asks you to find. To knock out guards, you must throw stones, with the objective of hitting the guards and stunning them for enough time for you to be able to walk away. While exploring the island, you will also be looking for four pieces of an ancient artifact hidden somewhere on the island.

This artifact was lost by the old man on his arrival, and he needs you to find the hidden pieces so that he can give you tips about a hidden boat you can use to escape the island. One of the pieces is in a locked house; in order to unlock the door, you must pick up several power cells to fuel the generator that powers the door to the cabin. A sneak peek of what the island should look like after our level-building sessions in this and the next chapter looks something like this:



A perspective view of the finished island terrain, enriched with trees, rocks, foliage, and buildings

The path will bring the character to a half-abandoned village, where he will meet an old man who can help him escape and leave the island in return for pieces of an ancient artifact that was broken into pieces and lost during a battle many years before. Other hidden and non-hidden locations can be found by following the path across the island.

We will begin by using the Unity terrain tools to create and design the island. Toward the end of the book, we will add further details to the terrain. It should look something like the following screenshot, if we utilize an orthographic top view:



A top shaded view of the island

It may be useful to keep the prototype scene for reference, so we will simply leave it within this project in a scene unrelated to our main game. Ensure that your prototype scene is saved and then go to **File | New Scene**. Now go to **File | Save Scene As** and save the scene in your `Assets` folder, naming it `Island`.



Naming Conventions

Take note here that it is important to name elements in a case sensitive manner because, when writing code to address this scene, for example, we will refer to the name `Island` with a capital `I`.

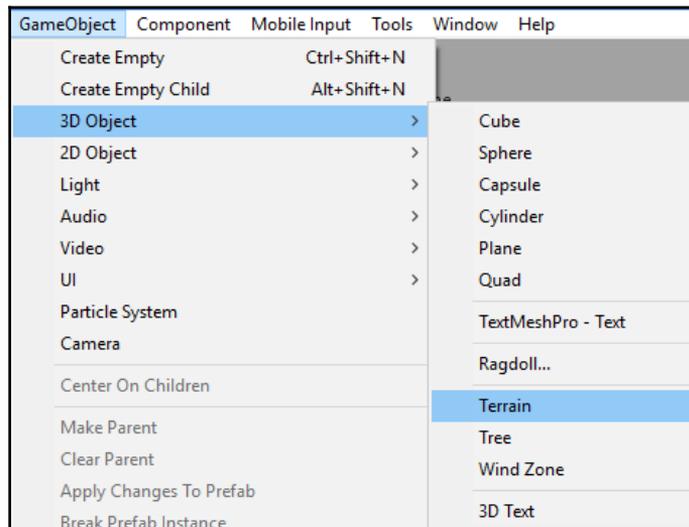
Understanding and using the Terrain tool

In Unity terms, think of a terrain as simply a game object that has a terrain toolkit component applied to it. Beginning as a plane—a flat, single-sided 3D shape, the terrain you'll create shortly will be transformed into a complete set of realistic geometry, with additional details such as trees, rocks, foliage, and even atmospheric effects such as wind, which can move tree branches and foliage. In building any game that involves an outdoor environment, a Terrain Editor is a must-have for any game developer. Unity has its own built-in Terrain Editor, and this makes building complete environments quick and easy.

Setting terrain features

In order to take a look at the features outlined in the following section, you will need to create a terrain.

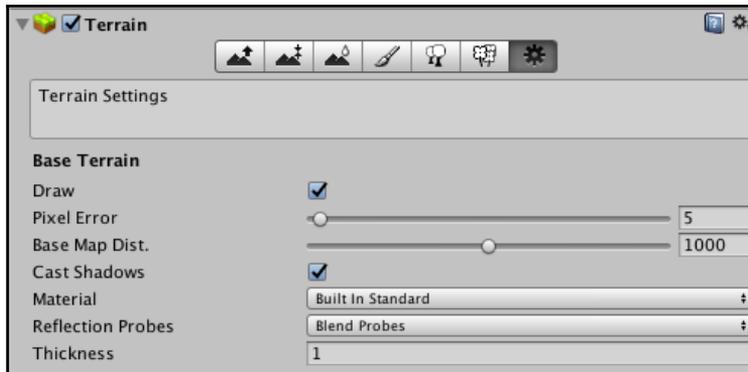
So, let's begin by introducing a new terrain object to the game; this is an asset that can be created within Unity, so simply go to **GameObject | 3D Object | Terrain** from the top menu:



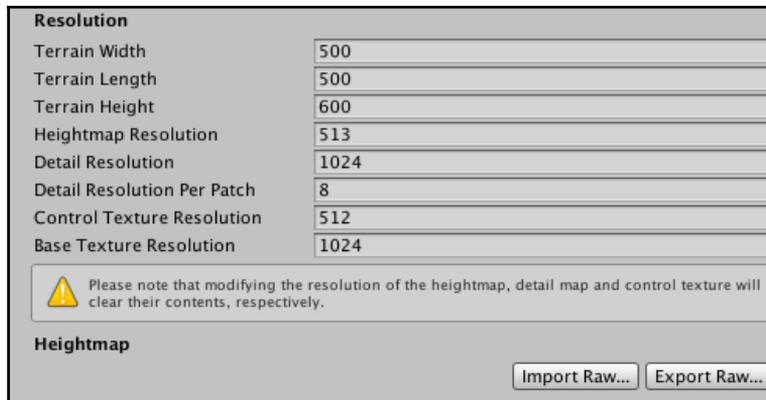
Before you can begin to modify your terrain, you should set up various settings for its size and detail.

Setting the terrain size detail and resolution

Clicking the last button of the terrain tool will show you the **Terrain Settings**:



The following part of the panel allows you to set a number of properties for any new terrain object you have created:



The following settings should always be adjusted before the topography of the terrain is created, as adjusting them later can cause any work done on the terrain to be reset:

- **Terrain Width, Terrain Height, and Terrain Length:** Measured in meters (1 meter = 1 Unity unit).



Note that *height* sets the maximum height that the terrain's topography can be.

- **Heightmap Resolution:** The resolution of the texture that Unity stores to represent the topography in pixels. Note that, although most terrain textures in Unity are created at a power of two dimension, 128x128, 256x256, 512x512, and so on, **Heightmap Resolutions** always add an extra pixel because each pixel defines a vertex point. So, in the example of a 4x4 terrain, four vertices would be present along each section of the grid, but the points at which they meet, including their endpoints, would be equal to five.
- **Detail Resolution:** This is the resolution of the graphic, known as a **Detail resolution map**, which Unity stores in your project metadata. This defines how precisely you can place details on the terrain. Details are additional terrain features such as plants, rocks, and bushes. The larger the value, the more precisely you can place details on the terrain in terms of positioning.
- **Detail Resolution Per Patch:** This is the resolution of each patch of the terrain that is painted on.
- **Control Texture Resolution:** This is the resolution of textures when painted onto the terrain. Known as **Splatmap** textures in Unity, the **Control Texture Resolution** value controls the detail of the gradients created between textures that you paint onto the terrain. As with all texture resolutions, it is advisable to keep this figure low to increase performance. With this in mind, it is a good practice to leave this value set to its default of 512. The higher this value, the finer control you will have of the edges between multiple textures in the same area.
- **Base Texture Resolution:** This is the resolution of the texture used by Unity to render terrain areas in distances that are further away from the in-game camera.

Importing and exporting 2D heightmaps

Heightmaps are 2D graphics with light and dark areas to represent terrain topography and can be imported as an alternative to using Unity's height painting tools.

Created in an art package such as Photoshop and saved in the `.RAW` format, heightmaps are often used in game development, as they can easily be exported and transferred between art packages and development environments, such as the Unity engine.

As we will be using the Unity terrain tools to create our environment, we will not be utilizing externally created heightmaps as part of this book.



When using the **Paint Height** tools in Unity, you are creating a **Height Map** (a grayscale 8-bit image that defines the heights of the terrain) and a **Splat Map** (an RGB texture that defines the amount of the color map texture you will paint on the terrain surface). These tools make it easier for you. Readers from an artistic background may use this option to import or export existing maps if they have already made them in other graphics packages.

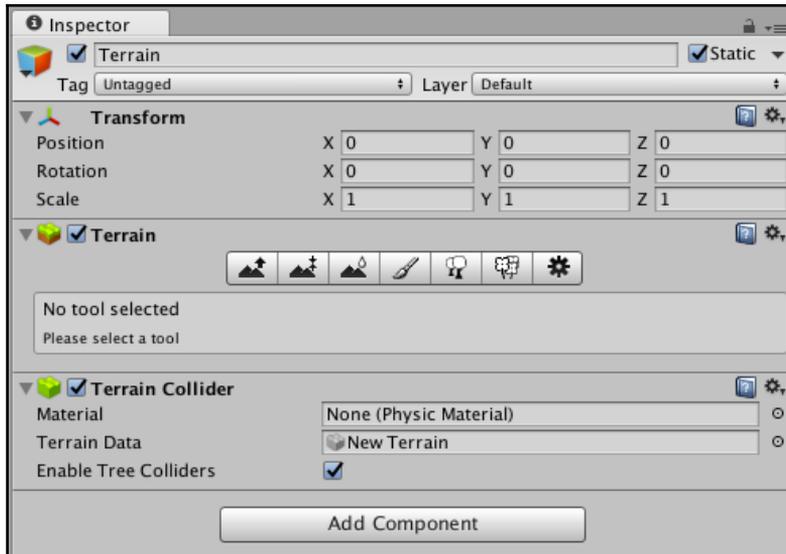
The terrain tool

Before we begin to use them to build our island, let's take a look at the terrain tools available, so that you can familiarize yourself with their functions. As you have just created your `Terrain`, it should be selected in the **Hierarchy** window. If it is not selected, then select it now in order to show its properties in the **Inspector**.

Terrain component

On the **Inspector**, the terrain toolset is referred to in component terms as the `Terrain`. The `Terrain` component gives you the ability to utilize the various tools and specify settings for the terrain, in addition to the functions available we outlined earlier.

In the following screenshot, you can see that this is the second of three components on the Terrain game object: the usual **Transform** component, responsible for the position, rotation, and scale of the object, and the **Terrain Collider**, responsible for the collision of other objects with the Terrain itself:

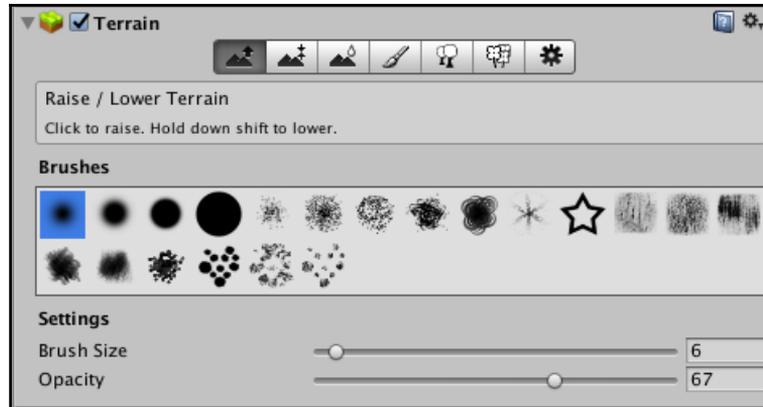


How the Inspector looks when selecting the Terrain game object

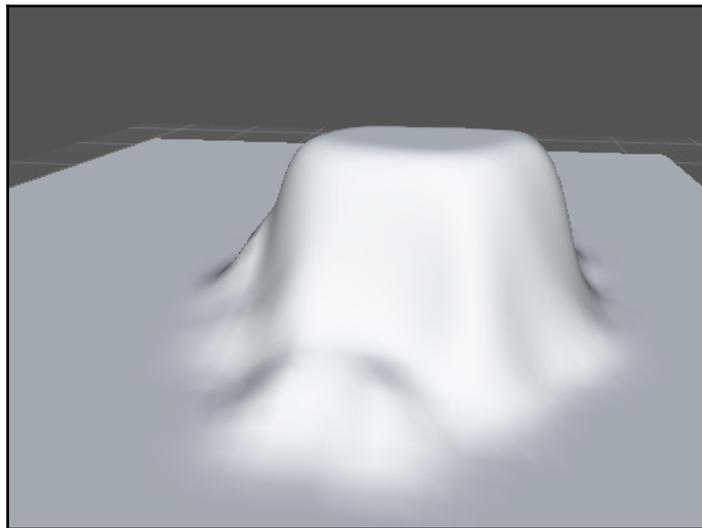
The Terrain component has seven sections, which are easily accessible from the icon buttons at the top of the component. Before we begin, here is a quick overview of their purpose in building terrains.

Raise height

This tool allows you to set areas of the terrain at a given point by painting the area you want to flatten at that specific height. You can also sample the terrain height by entering values manually. Do this by holding down the *Shift* key and clicking the area of the terrain that you want to sample the height of:

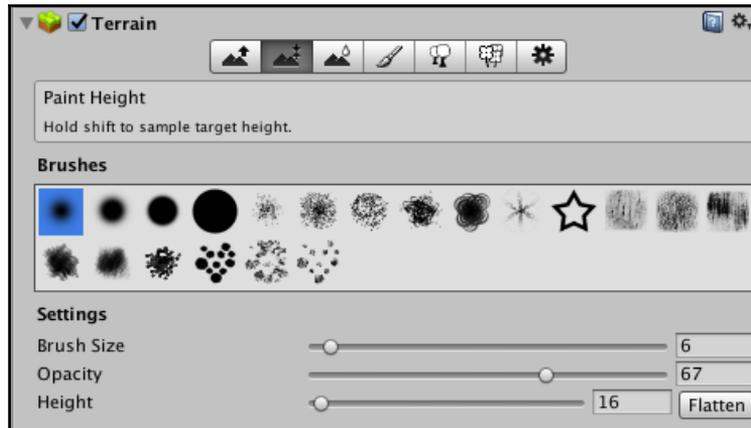


You also have the ability to specify brush styles under the **Brushes** section, as well as **Brush Size** and **Opacity** (effectiveness) for the changes you make. Holding the *Shift* key while using this tool creates the opposite effect, lowering the height, as shown in the following screenshot:

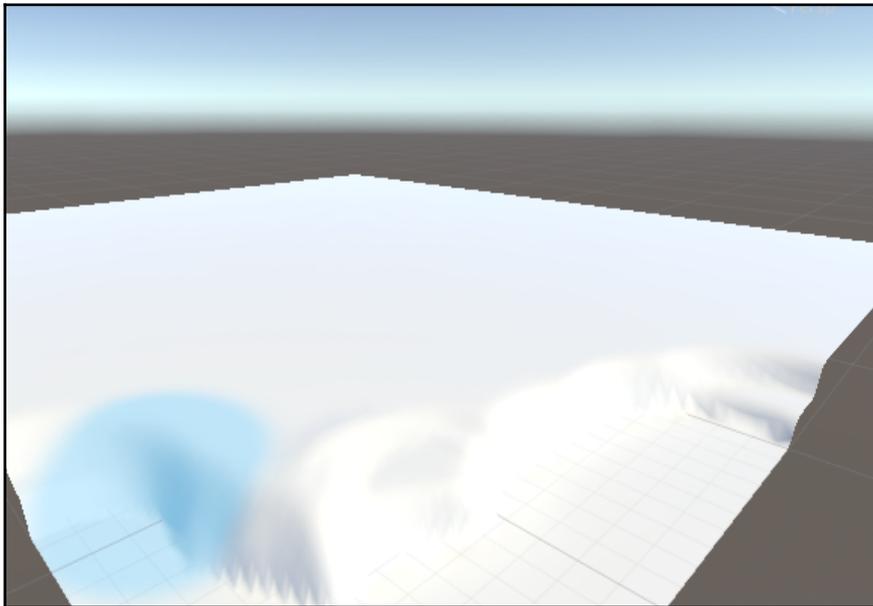


Paint Height

This tool works in a similar way to the **Raise Height** tool, but gives you an additional setting, **Height**:



This means that you can specify a height to paint towards, which means that, when the area of the terrain that you are raising reaches the specified height, it will flatten out, allowing you to create plateaus, as shown in the following screenshot:

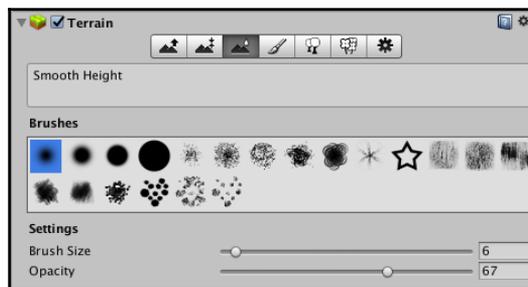


Flatten Heightmap

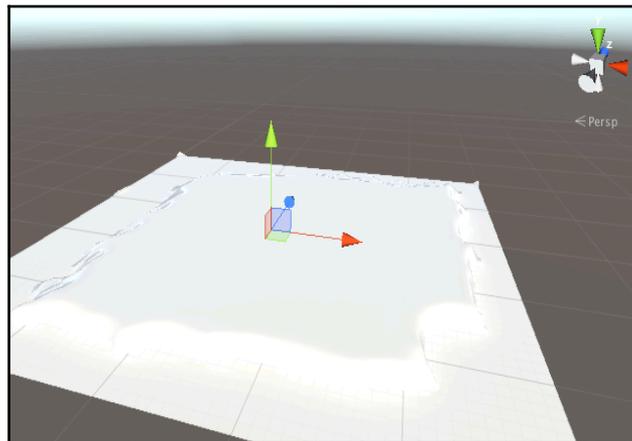
The **Flatten Heightmap** tool allows you to flatten the entire terrain at a certain height. By default, your terrain height begins at zero, so if you wish to make a terrain with a default height above this, as we do for our island, then you can specify the height value here.

Smooth Height

This tool is mainly used to complement other tools, such as **Paint Height**, in order to soften step areas of topography:

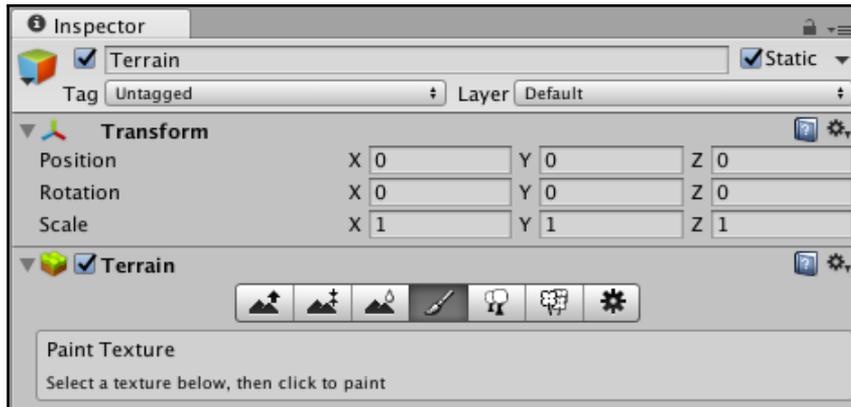


In the previous plateau, the land goes straight up. Should I wish to soften the edges of the raised area, I would use this tool to round off the harsh edges, creating the result shown in the following screenshot:



Paint Texture

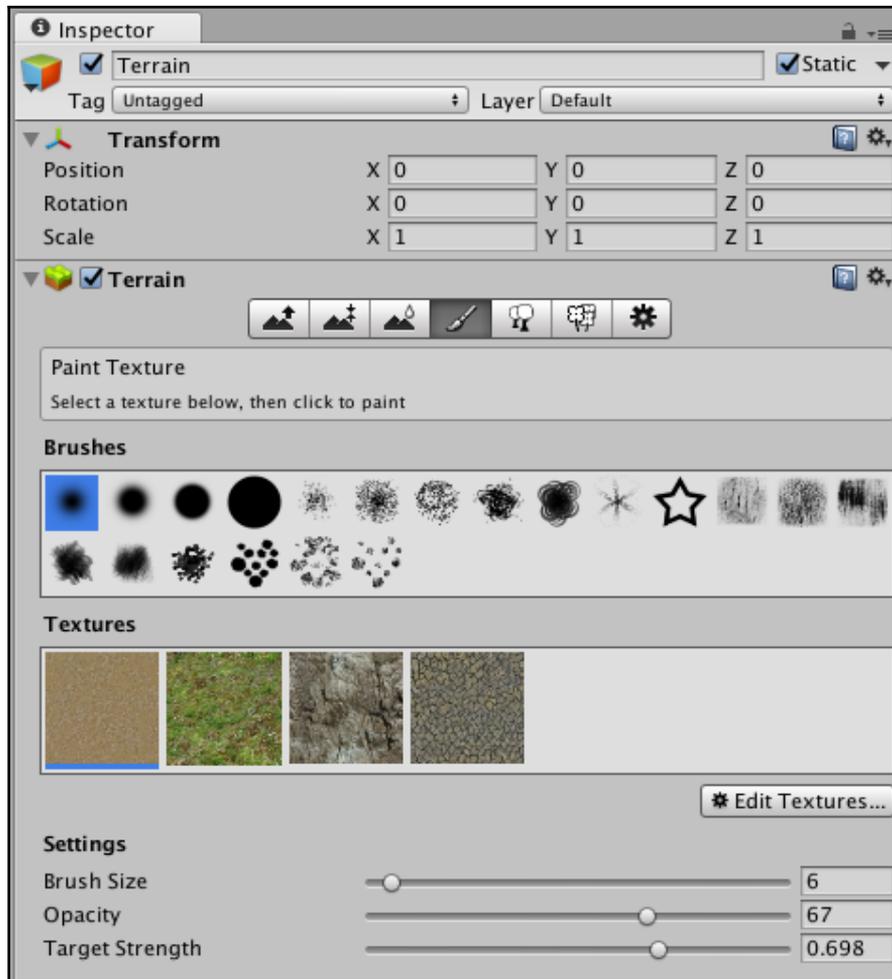
Paint Texture is the tool used to brush textures, referred to as *Splatmaps* in Unity terrain terms, by mixing them with different levels of opacity onto the surface of the terrain:



In order to paint with textures, the textures must first be added to the palette in the **Textures** area of this tool, which starts as null by default. Textures can be added by clicking on the **Edit Textures** button (see the following screenshot) and selecting **Add Texture**, which will allow you to choose any texture file currently in your project, along with parameters for tiling the chosen texture.

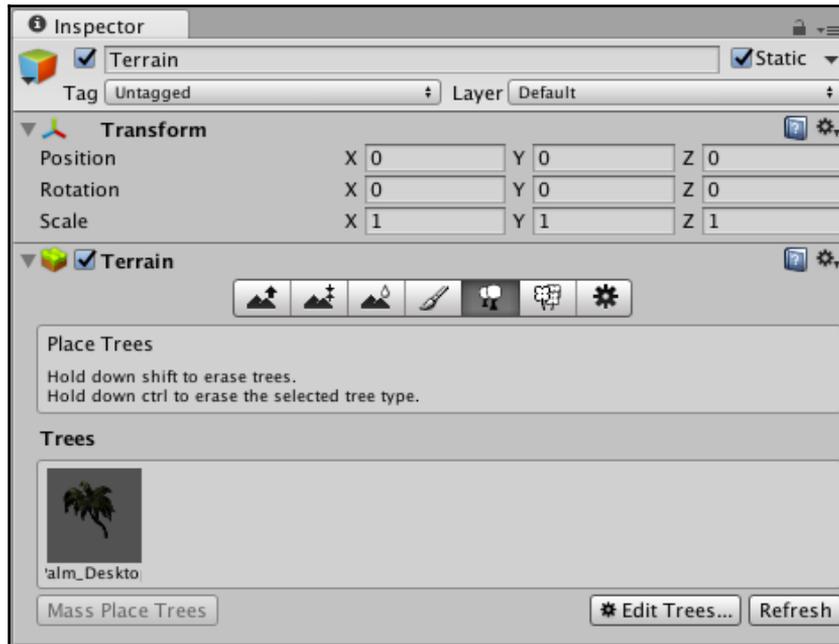
The following screenshot is an example of this tool with three textures in the palette. The first texture you add to the palette will be painted over the entire terrain by default. Then, by combining several textures at varying opacity and painting manually onto the terrain, you can get a realistic portrayal of the kind of surface you're hoping to get. To choose a texture to paint with, simply click on its preview in the palette.

The currently chosen texture is highlighted with a blue outline, and the brush used to paint this texture is also highlighted in blue:



Place Trees

This is another tool that does as its name suggests. By brushing with the mouse, or using single clicks, **Place Trees** can be used to paint trees onto the terrain, having specified which asset to use when painting:



Refresh (tree prototypes)

If you make changes to the assets that make up any trees and details that have already been painted onto the terrain, then you'll need to select **Refresh** to update them on the terrain.

Edit Trees

Similar to specifying textures for the **Paint Texture** tool, this tool gives you an **Edit Trees...** button to add, edit, and remove assets from the palette. These assets will be prefab tree game objects. In this book, you will use the Palm tree prefab game object that was imported with the `Terrain Assets` package when you created the new project at the start of the book.

In its **Settings**, you can specify:

- **Brush Size:** The amount of trees to paint per click
- **Tree Density:** The proximity of trees placed when painting
- **Color Variation:** The application of random color variation to trees when painting several at once
- **TreeWidth/Height:** The size of the tree asset you are painting with
- **TreeWidth/HeightVariation:** This variable gives you random variation in sizing to create more realistically forested areas

This tool also utilizes the *Shift* key hold to reverse its effects. In this example, keeping the *Shift* key pressed erases the trees already painted on the terrain and can be used in conjunction with the *Ctrl* key hold to only erase trees of the type selected in the palette.

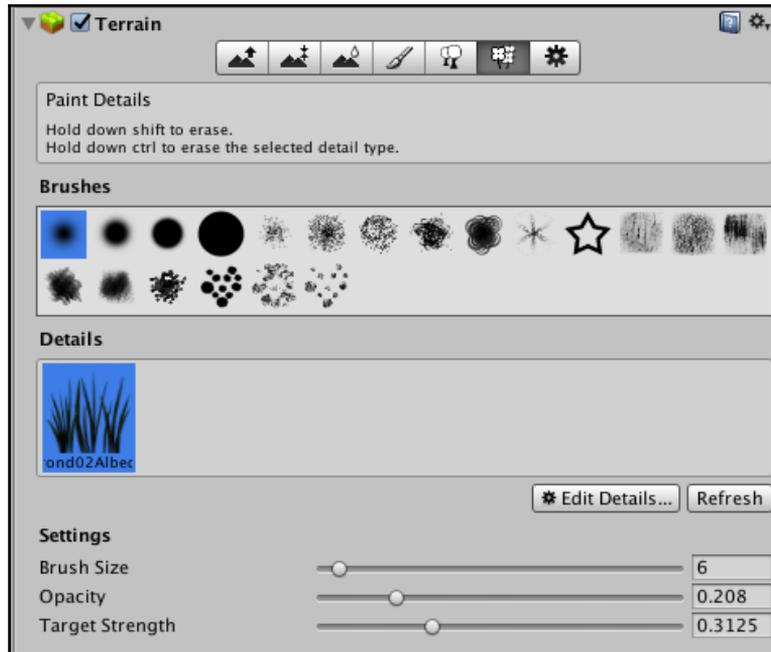
Mass placement of trees

This function does exactly what its name says, places a specified number of trees onto the terrain, with specific trees and associated parameters (width, height, density, and variation) which are currently set in the **Place Trees** area of the terrain script component in the **Inspector**.

This function is not recommended for general use, as it gives you no control over the position of the trees, only their density, width, and height as set in the component itself. It is recommended that you use the **Place Trees** part of the terrain tools instead, in order to manually paint a more realistic placement.

Paint details

This tool works in a similar manner to the **Place Trees** tool but is designed to work with detail objects such as flowers, plants, rocks, and other foliage:



Refresh (detail prototypes)

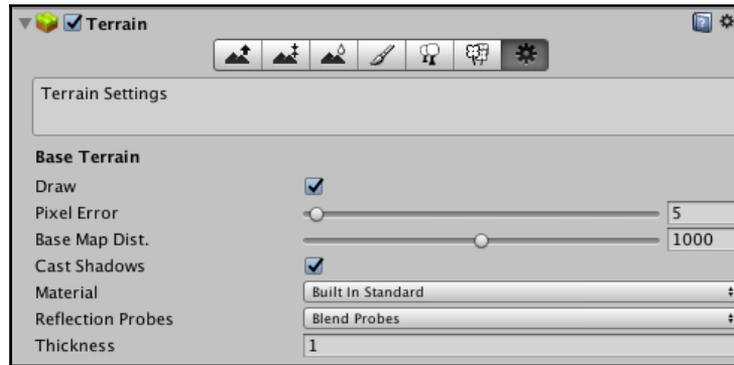
If you make changes to the assets that make up any trees and details that have already been painted onto the terrain, then you'll need to select **Refresh** to update them on the terrain.

Edit details

In the same way as specifying textures for the **Paint Texture** tool, this tool gives you an **Edit Details** button to add, edit, and remove assets from the palette. These assets will be prefab mesh detail game objects or a simple texture to generate grass.

Terrain Settings

The **Terrain Settings** area of the **Terrain (Script)** contains various settings for the drawing of the terrain by the computer's **Graphical Processing Unit (GPU)**:



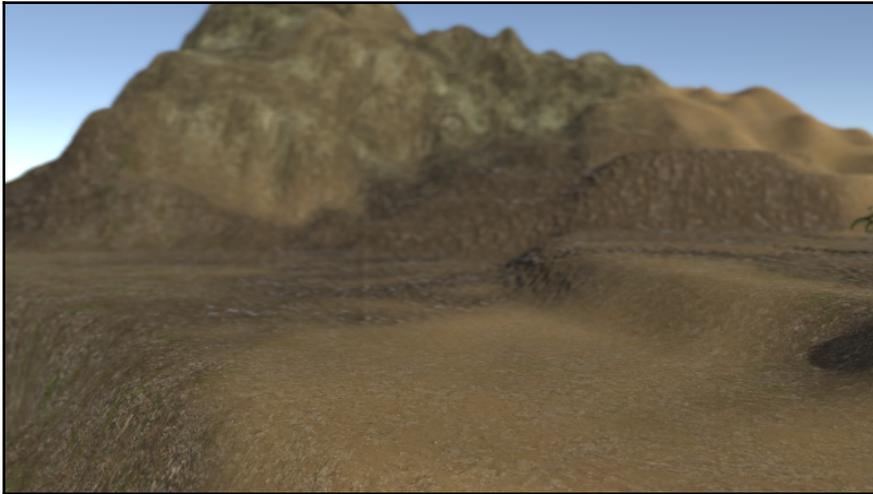
Here, you can specify various settings that affect the **Level of Detail (LOD)**.



LOD in game development defines the amount of detail specified within a certain range of a player. In this example, a terrain, we need to be able to adjust settings such as **Draw** distance, which is a common 3D game concept that renders less detail after a certain distance from the player in order to improve performance.

In **Terrain Settings**, for example, you can adjust the **Base Map Dist.** in order to specify how far away a player must go until the terrain replaces high-resolution graphics for lower-resolution ones, making objects in the distance less expensive to render. The following screenshot is an example of a low **Base Map Dist.** of around 10 meters (the unit of measure in Unity).

As you can see, the textures further than the specified distance are drawn at far lower detail:



Pixel Error indicates the overall quality setting and, the lower it is, the more polygons, and hence quality, the terrain will have at a certain distance. This setting should be changed depending on the target platform and the overall detail that you want to give to the terrain part of the environment. We'll look at **Terrain Settings** more as we begin to build our island.

Creating the island

Now, let's dive in to the Terrain Editor and create our island as well as adding grass, trees, and plants to the terrain, surrounding and embedded water (a sea and a lake), and environmental audio.

Step 1 – Setting up the terrain

Now that we have looked at the tools available for creating our terrain, let's get started by setting up our terrain using the **Terrain** top menu. Ensure that your **Terrain** is still selected in the **Hierarchy** and go to **Terrain | Set Resolution**.

As we don't want to make too large an island for our first project, set the terrain **width** and **length** to 500.



Remember to press *Enter* after typing these values so that you effectively confirm them before clicking on the **Set Resolution** button to complete the step.

Next, our island's height needs to begin at its ground level, rather than at zero, which is the default for new terrains. If you consider the terrain height of zero to be the seabed, then we can say that our ground level should be raised to be the surface height of the island. Go to **Terrain | Flatten Heightmap**.

Click inside the **Height** box, place in a value of 30 meters, and then press *Enter* to confirm. Click on **Flatten** to finish.

This change is a blink-and-you'll-miss-it difference in the **Scene** view, as all we've done is shift the terrain upward slightly. However, the reason we've done this is that it is a great timesaver, and we can now flatten around the edges of the terrain using an inverse **Raise Height** to leave a raised island in the center of the terrain. This is a more time-efficient method than beginning with a flat zero-height terrain and then raising the height in the center.

Step 2 – Creating the island outline

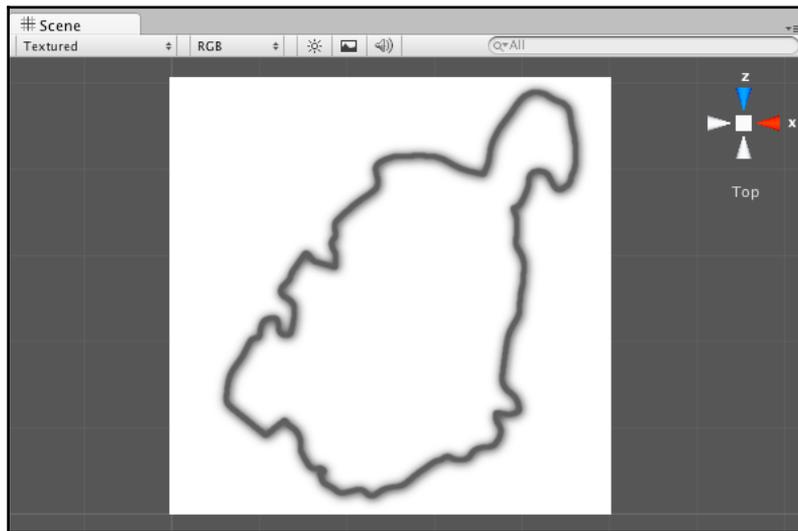
On the **Inspector** for the Terrain game object's **Terrain** component, choose the **Raise Height** tool, the first of the seven buttons. Select the first brush in the palette and set its **Brush Size** to 100. Set the **Opacity** for the brush to 75. Change your view in the **Scene** panel to a top-down view by clicking on the **Y-axis** of the view gizmo in the top-right.

The view gizmo helps to set the view with a single click on one of the axes or on the center point is shown here:



Using the *Shift* key to lower height, paint around the outline of the terrain, creating a coastline that descends to zero height, you will know that it has reached zero because the terrain will flatten out at this minimum height.

While there is no need to exactly match the outline of the island suggested at the start of this chapter, try not to make a wildly different shape either, as you will need a flat expanse of land for buildings later. If you need a guide for the shape, either look at the following screenshot or the map diagram shown earlier in the chapter. Once you have painted around the entire outline, it should look something like the following screenshot:



Now, switch back to a **Perspective** (3D) view of your **Scene** by clicking on the center cube of the view gizmo in the top-right of the **Scene** window and admire your handiwork.

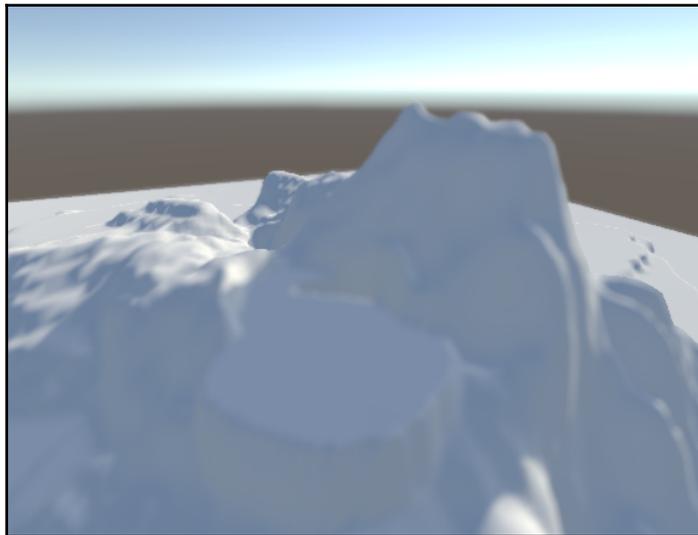
Now, spend some time going over the island, using the **Raise Height** tool to create some further topographical detail, and perhaps using lower height (with the *Shift* key) to add a lake. Then, make use of the **Smooth Height** tool to go around the edge of the terrain and soften some of those steep cliff edges. Leave a flat area in the south-west of your terrain (lower-left in the earlier screenshot) and one free corner of your map, in which we are going to add the base shape to the terrain for our small mountain lake!

Step 3 – A small lake carving

Now we are going to create a small lake! For this, we will combine the use of the **Paint Height**, **Raise Height**, and **Smooth Height** tools. First, select the **Paint Height** tool.

Choose the first brush in the palette and set the **Brush Size** to 75, **Opacity** to 50, and **Height** to 100. Select the **Top** view, again using the **view gizmo**, and paint on a plateau in the south-west, which you left free on your terrain.

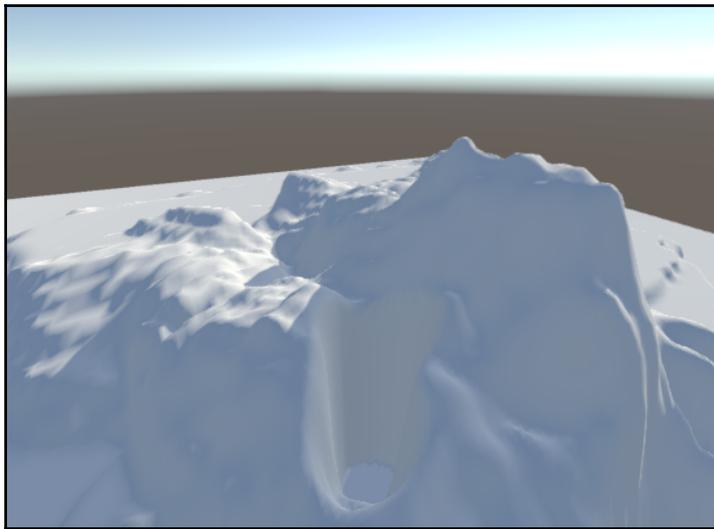
Remember that this tool will stop affecting the terrain once it reaches the specified height of 100:



How your island may now look in the Scene view

Although this plateau may look crude (we'll rectify this with smoothing shortly), we first need to create the lake carving. So now, with the **Paint Height** tool still selected, change the **Height** setting to 20 and the **Brush Size** to 30. The next step will also be easier to accomplish using a top-down view, so click the *y* axis spoke of the view **Gizmo** to change the **Scene** view. Now, hold down the mouse and start painting from the center of the mountain plateau outwards toward its edge in every direction, until you have effectively hollowed out the plateau.

We still have a fairly solid edge to this ridge, and when switching to the **Perspective** view, you'll see that it still doesn't look quite right. This is where the **Smooth Height** tool comes in. Select the **Smooth Height** tool and set the **Brush Size** to 30 and **Opacity** to 100. Using this tool, paint around the edge of the ridge with your mouse, softening its height until you have created a rounded soft ridge, as shown in the following image:



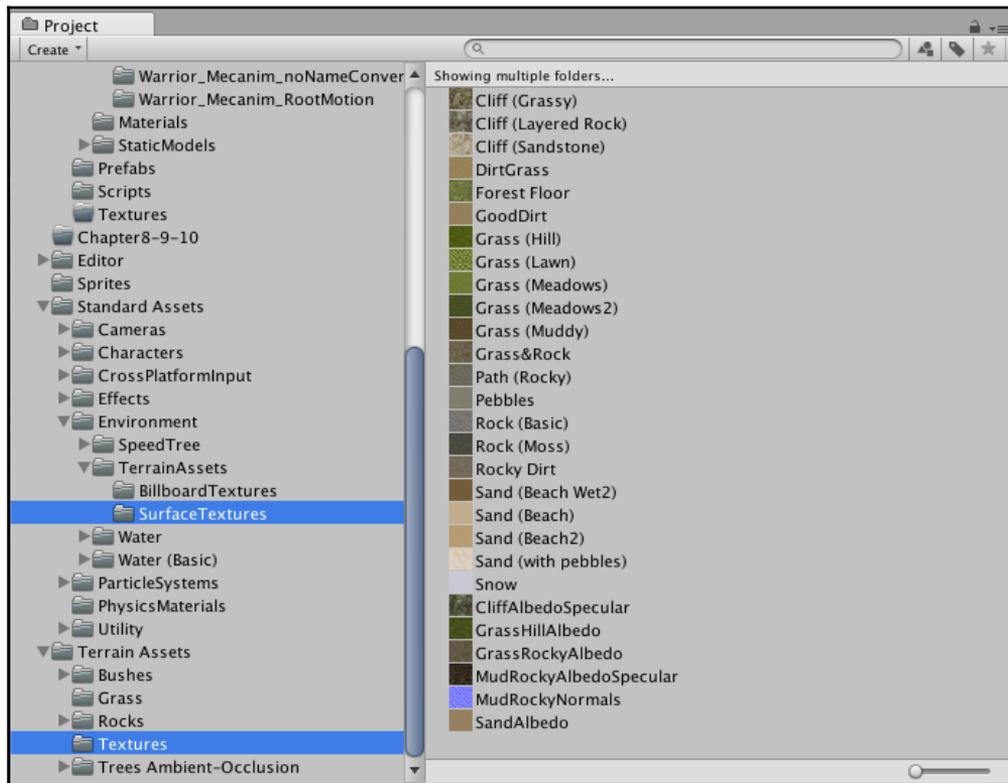
Finally, take some time to create a few ridges around the island terrain, leaving space for a path leading from the south of the island, the bottom of the shape shown in the previous image, to where we stated our buildings would be placed on the map illustration earlier in this chapter.

Now that our lake has taken shape, we can begin to texture the island to add realism to our terrain.

Step 4 – Adding textures

When texturing your terrain, it is crucial to remember that the first texture you add will cover the terrain entirely. With this in mind, you should ensure that the first texture you add to your texture palette is the texture that represents the majority of your terrain.

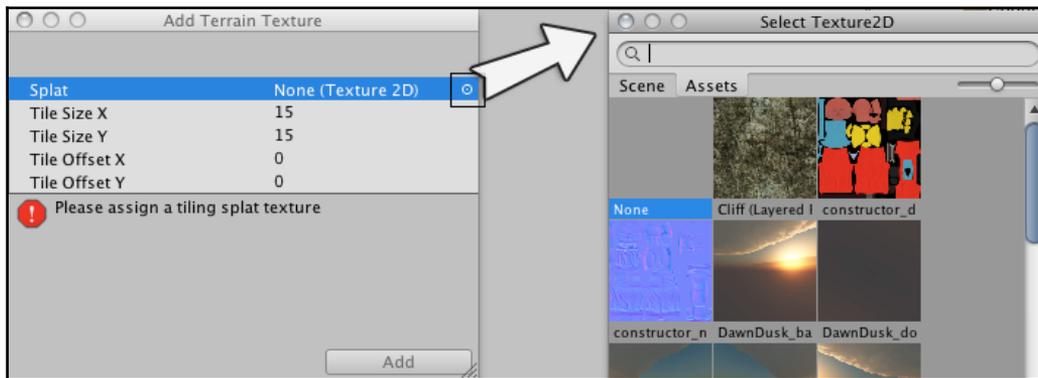
In the `TerrainAssets` package we included when we started the project, we are given various assets with which to customize terrains. As such, you'll find a folder called `Standard Assets` in your **Project** panel, which contains the folder **Terrain Assets**. Expand this downward by clicking on the gray arrow to the left of it, and then expand the subfolder called `TerrainTextures`:



These are the four textures we'll use to paint our island terrain, so we'll begin by adding them to our palette. Ensure that the **Terrain** game object is still selected in the **Hierarchy**, and then select the **Paint Texture** tool from the **Terrain** component in the **Inspector**.

Painting procedure

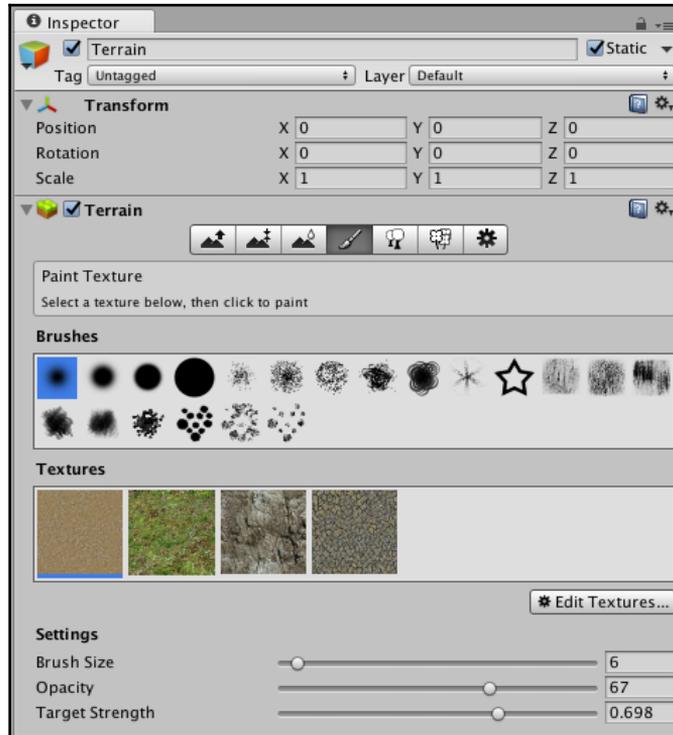
To introduce the four textures for our terrain, begin by clicking on the **Edit Textures** button and select **Add Textures** from the menu that pops out. This will launch the **Add Textures** dialog window, in which you can select any texture currently in your project. Click on the circle selection button to the far right of the **Splat** setting (see the following screenshot) to open an **Asset Selection** window, showing you a list of all the available textures, and then select the texture called **Grass (Hill)**. This **Select** dialog can be searched, so if you like, you can start typing the word **Grass** in order to narrow down the textures that you are shown:



Leave the **Tile Size X** and **Tile Size Y** values on 15 here, as this texture will cover the entire map, and this small value will give us a more detailed-looking grass. The **Tile Offset** values can be left as 0, as this texture is designed to tile as standard, so does not need offsetting.

Click on **Add** to finish. This will cover your terrain with the grass texture, as it is the first one we have added. Any future textures added to the palette will need to be painted on manually.

Repeat the previous step to add three further textures to the palette, choosing the textures named **Grass&Rock**, **GoodDirt**, and **Cliff (LayeredRock)**, while leaving the **Tile Size** settings unchanged for all except the **Cliff (LayeredRock)** texture, which should have a **Tile Size X** and **Tile Size Y** of 70, as it will be applied to a stretched area of the map and will look distorted unless tiled at a larger scale:



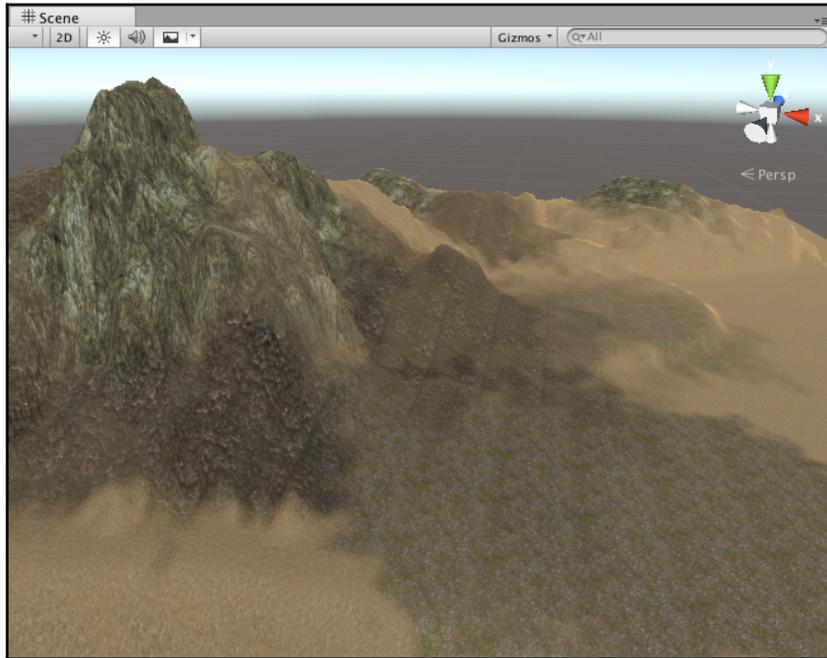
Sandy areas

You should now have all four textures available in your palette. Matching the above settings, choose the texture called **GoodDirt**. It should become highlighted with a blue underline, as shown in the preceding screenshot. Set the **Brush Size** to 60, **Opacity** to 50, and **Target Strength** to 1. You can now paint around the coast of the island using either the **Top** or **Perspective** view.



If you are using the **Perspective** view to paint textures, it will help to remember that you can use the *Alt* key while dragging the mouse, with either the **Hand** (Shortcut: *Q*) or **Transform** (Shortcut: *W*) tools selected, in order to rotate your view. You can also maximize any of the panels in the interface by hovering and pressing the *spacebar* to toggle maximization.

When finished, you should have something like the following screenshot:

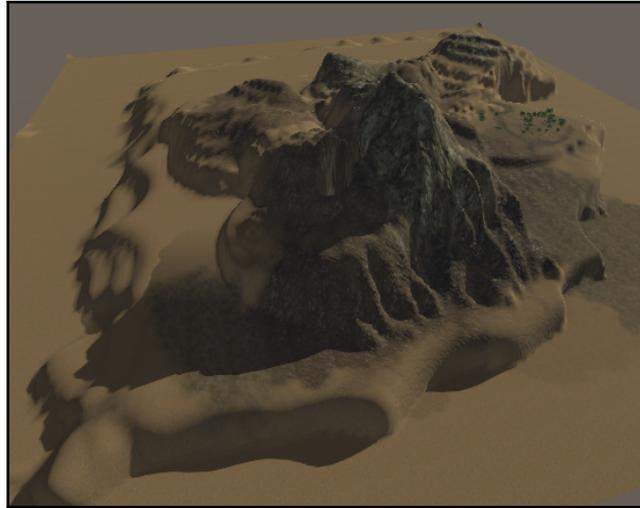


If you make a mistake while painting, you can either use **Edit | Undo** to step back one brush stroke, that is, a single-held mouse movement, or select the texture from the palette that you do want to be where you accidentally painted it and repaint over your mistake with that.

Adding grass and rock

Next, select the **Grass&Rock** texture by clicking on the second thumbnail in the palette. Set the **Brush Size** to 25, **Opacity** to 30, and **Target Strength** to 0.5. Now, brush over any hilly areas on your terrain and around the top half of the mountain.

Paint until you are satisfied, considering that you can always go back and adjust the shape, even after you have added trees and dynamic grass on your terrain:



The island terrain, shaped to hold our scene buildings and our mountain lake, fully textured and smoothed out

Step 5 – Tree time

Now that we have a fully textured island, we need to spruce up the place a little with some trees. In our `TerrainAssets` package, there is a tree provided to get us started with the Terrain Editor and, thankfully for us, it is a palm tree asset.



There is a Tree creator available in Unity, which you should try when you've finished this book, once you have more experience in Unity. For more information on this, visit:

<http://unity3d.com/support/documentation/Components/class-Tree.html>.

For now, we will make use of the palm tree we have already created. Select the **Place Trees** tool of the **Terrain(Script)** component and click on the **Edit Trees** button. From the drop-down menu that appears, choose **Add Tree**.

The **Add Tree** dialog window will appear. As with some of the other terrain tools, this *add* dialog allows us to select any object of an appropriate type from our `Assets` folder.

This is not restricted to trees provided in **Terrain Assets**, which means that you can model your own trees by saving them into the `Assets` folder of your project in order to use them with this tool (see the documentation on trees for further details). However, we are going to use the `Terrain` assets provided by the palm tree. Click on the circle selection button to the far right of the **Tree** setting. As with the texture painting earlier, this will open an **asset selection** window from which you can choose the **Palm** tree.

Bend factor here allows our trees to sway in the wind. This effect is computationally expensive, and any value will likely incur a performance reduction when play testing your game. If you experience poor performance, return to this setting and replace it with a value of **0**. For now, we'll simply use a low number for a small amount of tree sway, so type in a value of **2** and press *Enter* to confirm. If you find that this is causing low performance later in the development, then you can always return to this setting and set it back to **0**. Click on the **Add** button to finish:



With your palm tree in the palette, you should see a small preview of the tree with a blue background to show that it is selected as the tree to place on the terrain.

Set **Brush Size** to 15 (painting 15 trees at a time) and **Tree Density** to 40 (giving us a wide spread of trees). Set **Color Variation** to 0.4 to give us a varied set of trees and **Tree Height / Width** to 50 with their **Variation** settings at 30.

Using single-clicks, place trees around the coast of the island, near the sandy areas that you would expect to see them. Then, to complement the island's terrain, place a few more palm trees at random locations inland:



Remember that, if you paint trees incorrectly at any time, you can hold the *Shift* key and click, or paint (drag) with the mouse to erase trees from the terrain.

Step 6: The grass is always greener

Now that we have some trees on our island, we'll add a small amount of grass to complement the grass textures we covered the terrain with. Select the **Paint Details** section of the **Terrain** component and click on the **Edit Details** button. Select **Add Grass Texture** from the pop-up menu.

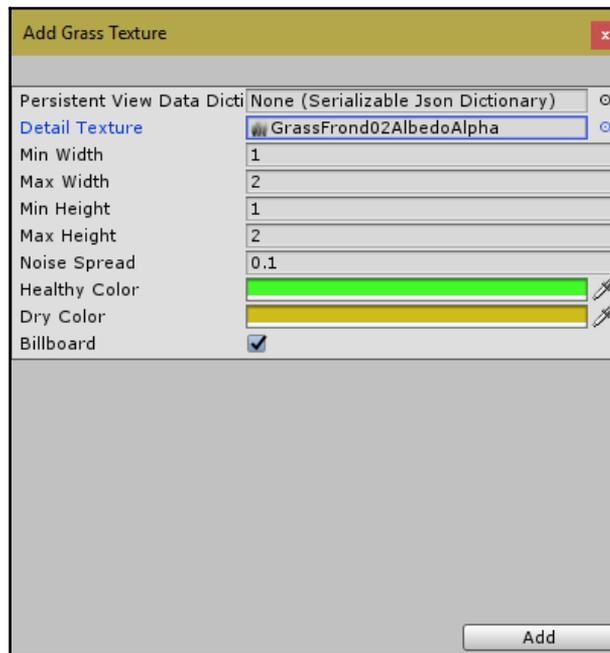
The `Terrain Assets` package provides us with a grass texture to use, so click the circle selection button to the right of the **Detail Texture** setting, and then, in the asset selection window that appears, select the texture simply called **Grass**.

Having chosen the **Grass** texture, leave the **Width** and **Height** values at their default and ensure that **Billboard** is selected at the bottom of this dialog. As our grass detail textures are 2D, we can employ billboarding, a technique in game development that rotates a texture mapped on two triangles to face the camera during the game play in order to make the grass seem more three-dimensional.



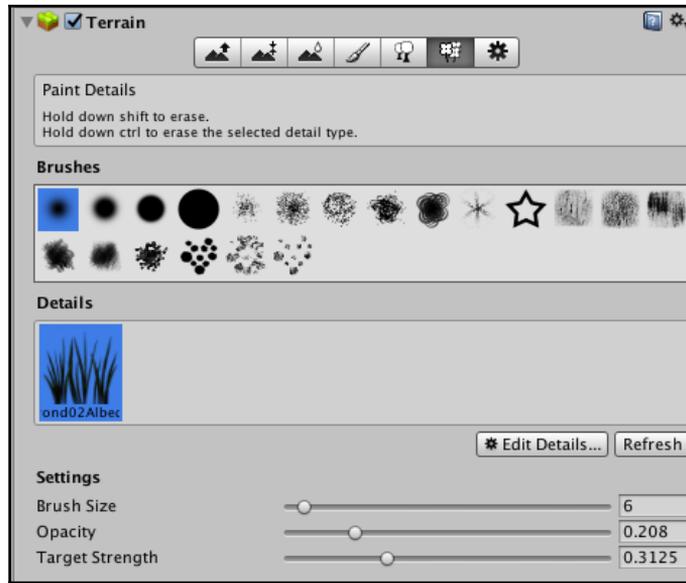
Using the color-picker boxes, ensure that the **Healthy** and **Dry** colors are similar shades of green to the textures you have painted onto the terrain, because leaving them as the default bright green will look out of place.

Click on the **Add** button at the bottom of the dialog window to confirm adding this texture to your palette:



To paint grass onto our map, we can yet again use mouse based brushing in a similar way to the other terrain tools. Firstly, we'll need to choose a brush and settings in order to ensure wide and disparate painting of grass detail onto our map. Given that rendering grass is another expensive feature for the computer to render, we'll keep the grass to a minimum by setting the **Brush Size** to 100, but **Opacity** to 0.1, and **Target Strength** to 0.3.

This will give us a wide spread, with very little grass, and by choosing a stipple brush (see the following screenshot) we can paint patchy areas of grass:



Now, zoom into the terrain surface by selecting the **Hand** tool and holding the *command* key (Mac) or the *Ctrl* key (Windows) while dragging the mouse to the right. Once at a close level of zoom, you'll need to reselect **Paint Details** on the Terrain tools and then click the mouse to paint areas of grass.

Move around the island painting a few grassy areas, do this sparingly, for performance reasons, and you can always come back and add more grass later if the game performs well:



A view of the terrain hills with grass and trees

Step 7 – Let there be light!

Now that our island terrain is ready to explore, we'll need to add lighting to our scene. When first approaching lighting in Unity, it's best to be aware of what the three different light types are used for:

1. **Directional Light:** Used as the main source of light, often as sunlight, directional light does not emanate from a single point, but instead simply travels in a single direction
2. **Point light:** This light emanates from a single point in the 3D world and is used for any other source of light, such as indoor lighting, fires, glowing objects, and so on
3. **Spot light:** Exactly what it sounds like, this light emanates from a single point, but has a radius value that can be set, much like focusing a flashlight

Creating sunlight

To introduce our main source of light, we'll add **Directional Light**. Click the **Create** button on the **Hierarchy** (or go to **GameObject | Create Other** from the top menu) and choose **Directional Light**. This adds the light as an object in our scene, and you will see that it is now listed in the **Hierarchy**.

As the **Directional Light** does not emanate from a point, its position is ordinarily irrelevant, as it cannot be seen by the player, only the light it casts is seen. However, in this tutorial, we're going to use the **Directional Light** to represent the sun by applying a **Flare** to the light.

In order to represent the sun, we'll position the light high above the island, and to ensure that it is consistent with the direction of the light it casts, we'll position the light away from the island on the z axis.

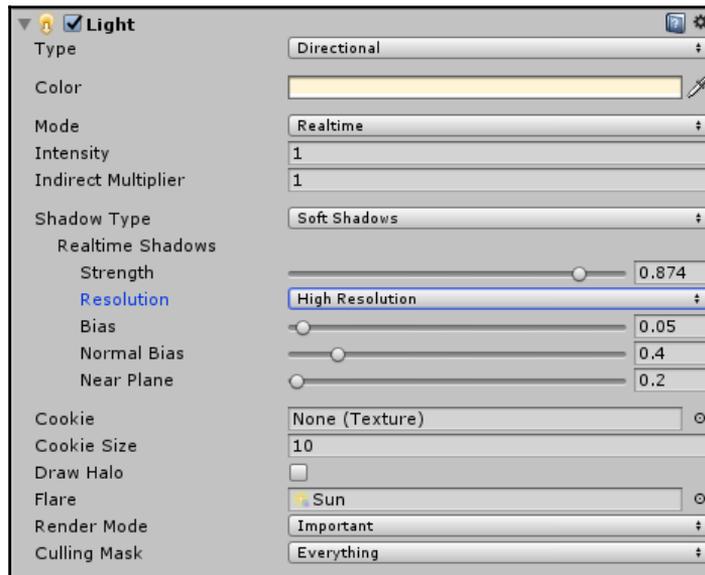
Position the light at $(0, 250, -200)$ by typing the **Position** values into the **Inspector Transform** component. Set the **X rotation** to 15 in order to tilt the light downward in the x axis, and this will cast more light onto our terrain.

Next, we need to make the light visible. To do this, we'll make use of some **Light Flares**. Unity provides these as a standard package that we can import. So, let's practice importing a new `Asset` package, as I did not ask you to include these when we began our new project!

Let's introduce that package by choosing **Assets | Import Package | Light Flares** from the top menu.

An **Importing Package** dialog window will appear, where you can simply click the **Import** button to confirm the addition of this package, it's that simple!

Now, you can click the circle selector button to the far right of the **Flare** setting in the **Light** component, which will open an asset selection window, from which you can choose from the various light flares we just imported. Choose the one titled **Sun** before closing the asset selection window. The component in the **Inspector** will look like this:

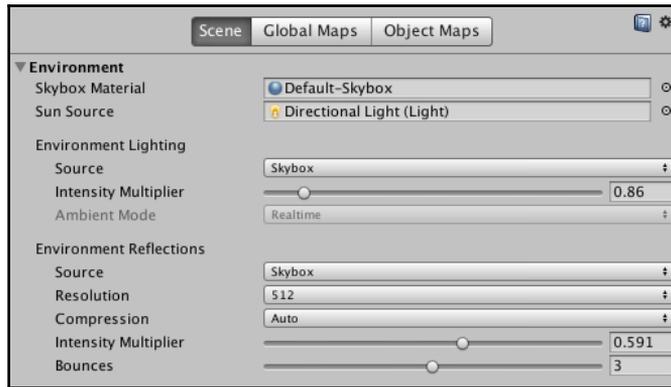


In [Chapter 13, *Optimization and Final Touches*](#), we will make use of the **Lighting** tool, which will take the lighting we have added here, and in other parts of the book, and *bake* it onto the environment, giving great quality **shadows** and great performance to your game.

Procedural skybox

When creating 3D environments, the horizon or distance is represented by the addition of a **skybox**. A skybox is usually a **cubemap**, a series of six textures placed inside a cube and rendered seamlessly to appear as a surrounding sky and horizon. This cubemap sits around the 3D world at all times, and much like the actual horizon, it is not an object the player can ever get to. Unity, however, recently introduced a new procedural skybox that will dynamically change color with the orientation of the Directional Light (the sun). This shader also implements a halo and the sun's **green flashes** and **green rays**.

In this kind of skybox, if you want clouds, rain, or weather simulation in general, you should think about developing or buying a ready-made system to dynamically manage the generation of rain, cloud movement, and so forth, in front of the *empty* procedural skybox, in which you might want to change the base color according to the weather conditions:

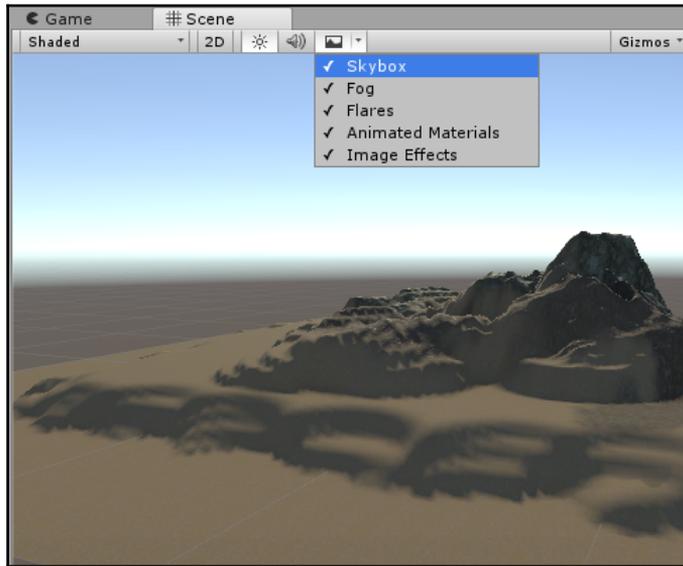


To apply a skybox to your scene, open the **Lighting** window from the main menu **Windows | Lighting | Settings**.

The **Inspector** area of the **Lighting** window will show you the preferences for the lighting of this scene.

To the right of the **Skybox** setting, click the circle selection button to open an asset selection window titled **Choose Material**. This is because, technically, a cubemap is a type of material, a way to apply textures to the 3D world. From this selection window, choose **Default-Skybox**. This will apply the procedural skybox.

It is crucial to understand that any type of **Skybox** you may specify will be shown unless you click the fourth button at the top of the **Scene** view, as shown in the following screenshot:

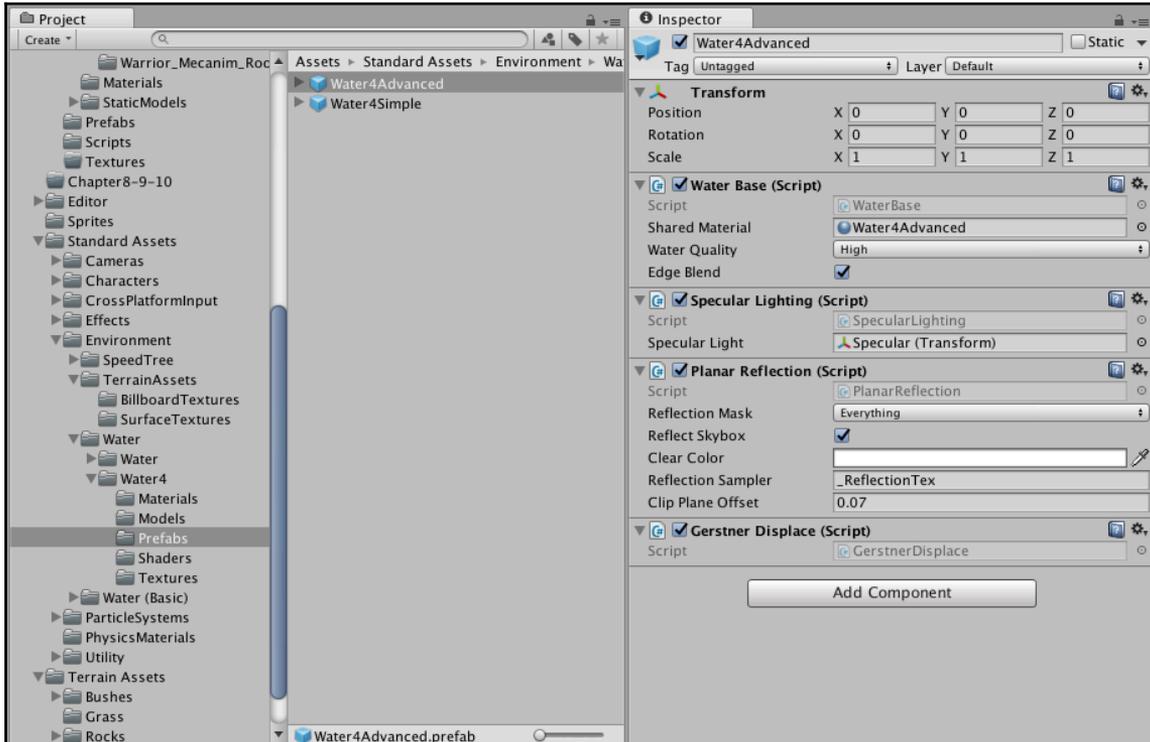


If you want, you can still use the older, fixed cubemap-based skybox instead. This choice usually depends on the type of game and on the fact that, by design, the game must implement a night/day cycle and/or different weather conditions.

Step 8 – Surrounding the island with sea water

As we have constructed an island terrain, it's only right that our land mass should be surrounded by water.

While it is possible to create further dynamic water effects by utilizing particles, the best way to add a large expanse of water is to use one of the water prefabs provided by Unity or by buying a water-making system on the **Asset Store**, such as the **Aquas** package:

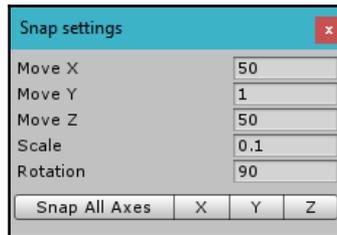


The Water4Advanced prefab from the legacy Unity Standard Assets

In the `Water4` folder, we find two ready-made surfaces with the water material applied. Saved as prefabs, these objects can be easily introduced from the **Project** panel. Open the subfolder of `StandardAssets` called `Water/Water4/Prefabs` or search for `Water4Advanced`. Drag the `Water4Advanced` prefab into the scene and position it at (0, 5, 0).

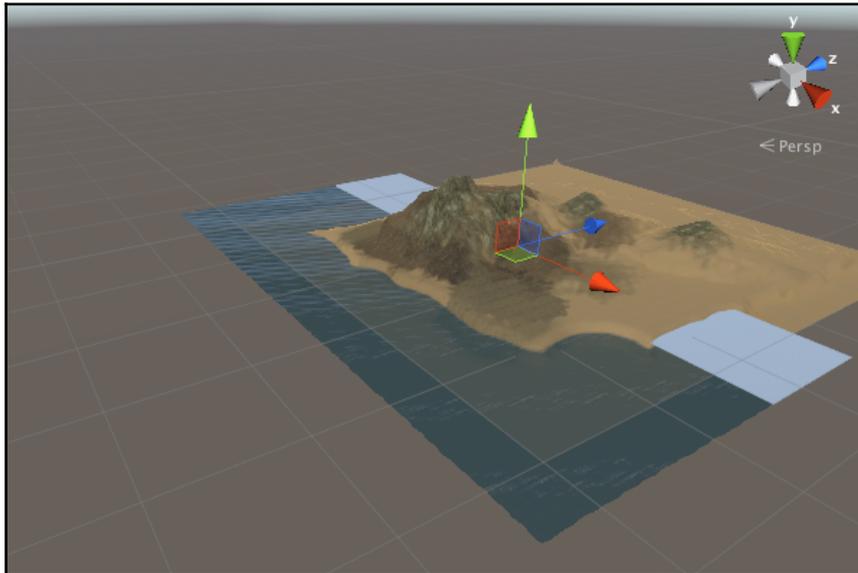
Rename the game object `OpenSeaWater`, so that we can store it later on a new prefab and be sure that the original prefab will stay untouched. Now, with the technique used in Chapter 1, *Entering the Third Dimension*, to make the wall of bricks, we will fill the surroundings of the island with multiple tiles to achieve the illusion of a large sea around it.

First of all, let's change the snap settings of the editor and put 50 and 50 for **Move X** and **Move Z**, and leave 1 for **Move Y**, as shown in the following capture of the **Snap settings** window:

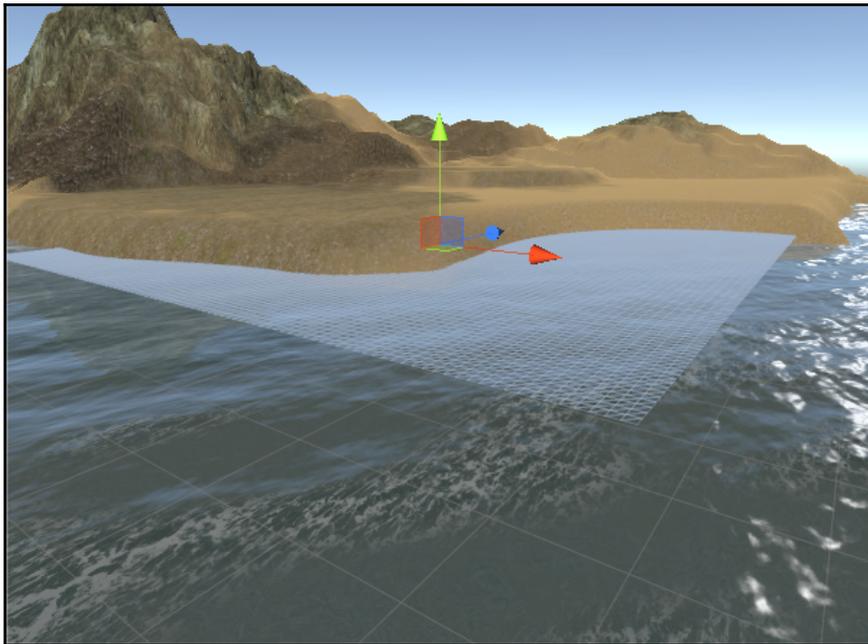


Now, clone the game objects called `Tile`, children of the `OpenSeaWater` game object with *Ctrl + D* and then move it, holding the *Ctrl* key pressed for keeping the object snapped; the snap settings we made will make it move to the next 50 meters, right by the side of the previous one. Repeat the operation another fifteen times to obtain a row of sixteen patches.

Now, with all sixteen patches selected, press *Ctrl + D* again and then move this row of patches along side the other axis. Repeat the operation another 14 times, so that you finish with 16 x 16 tiles for a total of 256 `Tile` objects:

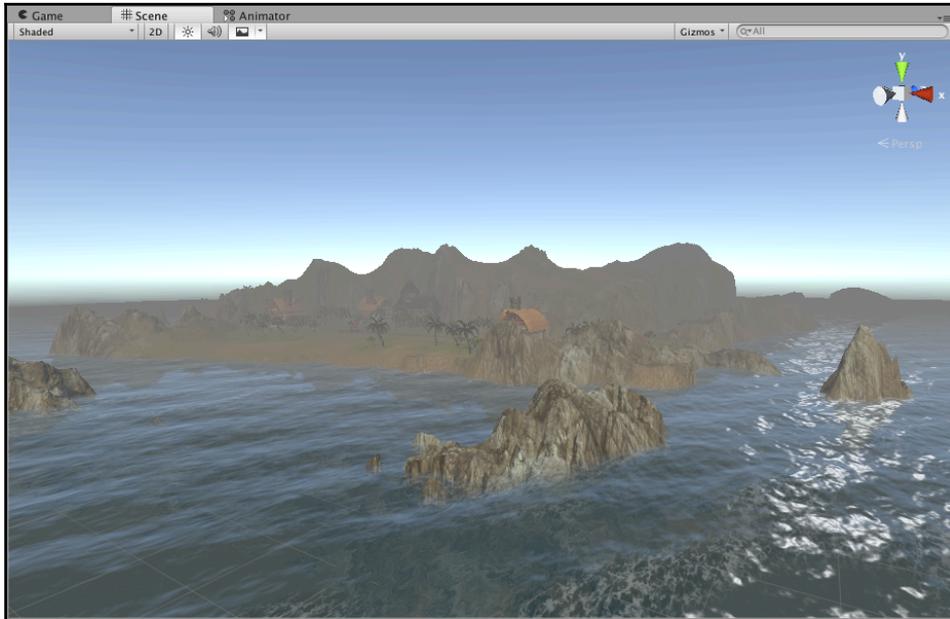


We can eliminate any patches we don't need at the center of the island. So, in the scene editor, we will go *under the terrain* and select the patches that are not seen outside. For example, the patch in the following screenshot should not be deleted:



After finishing our patient work, let's save this new prefab by dragging the game object to the **Project** view and into the `chapter5-6-8/prefabs` folder.

We should end up with something like this, if you have left fog enabled in the **Lighting settings**:



If you feel brave enough, you can do the same job we did manually with an editor script, which will take a given size for the sea you want to make, and will generate the correct tile square for you at build time.

Alternatively, by scripting, if your game design allows it, you may decide to let the engine generate the grid of water patches for you at runtime.

Another cool optimization path you might take is a script that will switch the water quality (low, medium, high) according to the distance from the player. But, again, this decision strongly depends on the kind of game you are making.

You can find a good modification of the Unity water system at this URL:

<http://kostiantyn-dvornik.blogspot.gr/2016/02/skyship-aurora-unity-water-tutorial.html>.

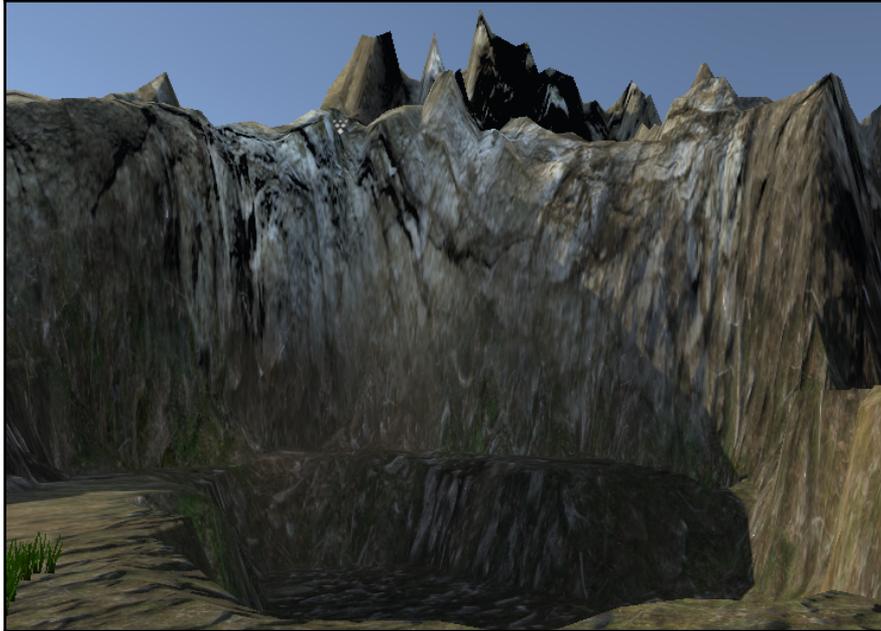
You can try the **AQUAS (Lite)** package by downloading it from the Asset Store; info at this URL:

<https://assetstore.unity.com/packages/vfx/shaders/aquas-water-lite-53519>.

A small lake

For this task, we are going to use the other water prefab at our disposal within the standard Unity assets: `Water4Simple`.

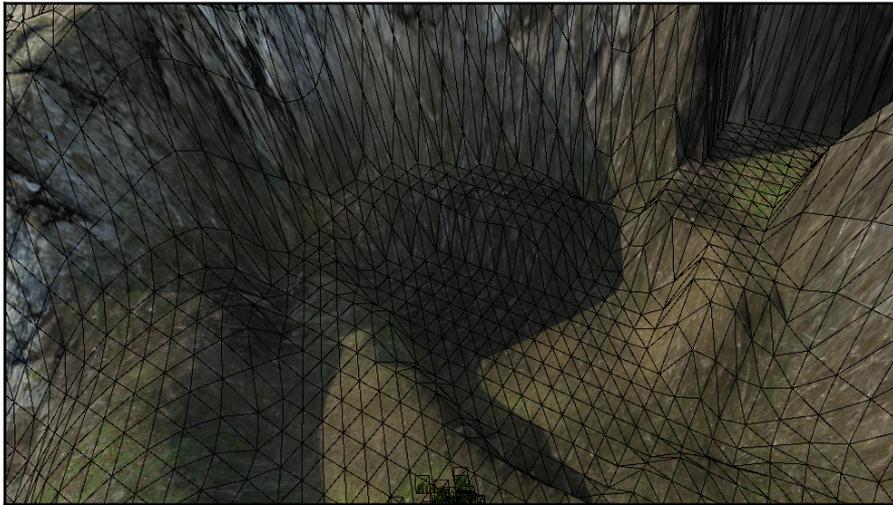
To make our lake look more realistic, we will paint the **Cliff(LayeredRock)** texture at the top of the mountain:



Select the cliff texture from the palette and set the **Brush Size** to 20, **Opacity** to 100, and **Target Strength** to 1.

Paint the top of the cliffs with this texture, while keeping the area for the lake carving green and sandy.

While this will take some experimentation, when finished, your small lake should look something like the following editor screenshot:



To complete the job, we want to add a small plane of water to give life to our small lake.

Search the **Project** view for the `Water4Simple` prefab and drag it as close as possible to the carved area that you prepared for the lake, as in the following screenshot:



Adjust with the Move tool or by dragging the arrows of the water-simple game object to cover the area of the lake in a realistic way; you should finish up with something like the preceding picture. We will now add some rock model prefabs (search for `rockmesh` in the **Project** view).

The cliff over the lake should be a little high, because later, in [Chapter 11, Unity Particle System](#), we will finalize the scene with a waterfall created with the Unity Particle System.

Step 9: What's that sound?

An often overlooked and sometimes underestimated area of game development is sound. For a truly immersive experience, your player needs to not only see the environment you've made, but to hear it as well!

Sound in Unity is handled through two basic components, the Audio Source, usually attached to a game object, or created at runtime with scripting, and the Audio Listener. Think of the Audio Source as a speaker in your game world, and the Audio Listener as a microphone or the player's ear. By default, Camera game objects in Unity have an Audio Listener component. So, given that you always have a camera in your game, chances are you'll only ever have to set up sound sources.



It is also worth noting that the Audio Listener component has no properties to adjust, it just works. Unity will inform you with an error if you accidentally remove the only listener in any scene, or if you have more than one active listener at any one time.

Positional audio versus non-positional audio

Unity 5 introduces, along with audio special effects and reverb areas, a new way of managing positional sounds through parameters and curves, as well as a brand new feature not covered in this book, the audio mixer.



Look at <https://docs.unity3d.com/Manual/Audio.html> for additional information about Unity audio.

For example, you would choose positional or non-positional audio for the following purposes:

- **In game music:** Non-positional audio would be best, as it would remain constant, no matter where the player's listener goes in the game.
- **Environmental ambience sounds:** Here, positional audio would be best, especially when you want to mix and fade in/out specific areas of your game level.
- **Sound fx tied to events:** Sound effects such as swords clashing, dialog voice speeches, footsteps, and other sounds are better treated as 2D non-positional sounds, as they usually have a fixed, or almost fixed, distance from the Audio Listener.
- **Sound effect of a radio inside a building:** Here, positional audio would be best. Although you may be playing music as your sound effect, using 3D audio will make the sound play spatially, allowing the sound source to become louder as the player gets closer to it and pan as they move away for a more immersive experience.

Audio file formats

Unity will accept the most common audio formats, WAV, MP3, AIFF, and OGG. Upon encountering a compressed format such as MP3, Unity converts your audio file to the `OggVorbis` file format, while leaving uncompressed sounds such as WAVs unconverted. Remember that all compression settings can be changed in the **Inspector** when selecting an asset in the **Project** panel.

As with other assets you import, audio files are converted as soon as you switch between another program and Unity, as Unity scans the contents of the `Assets` folder each time you switch to it to look for new files. The option to compress to another format, however, is not available for already-compressed files, such as MP3s that you may have added to your project.

The hills are alive!

To make our island feel more realistic, we'll add a sound source playing constant outdoor ambient environmental sounds using a 3D sound.

Begin by selecting the terrain object in the **Hierarchy**. Go to **Component | Audio | Audio Source** from the top menu. This adds an **Audio Source** component to the `Terrain` object. As the volume remains constant when using 2D sounds, and position is irrelevant, we could place the ambient sound source on any object; it simply makes logical sense that ambient sound of the terrain should be attached to that game object.

In the **Inspector** of the terrain, you will now see the Audio Source component, with which you may either choose a file to play or leave blank if you are planning to play sounds through scripting. Before you assign a sound, we'll need to get the relevant clip and the other assets for this book.

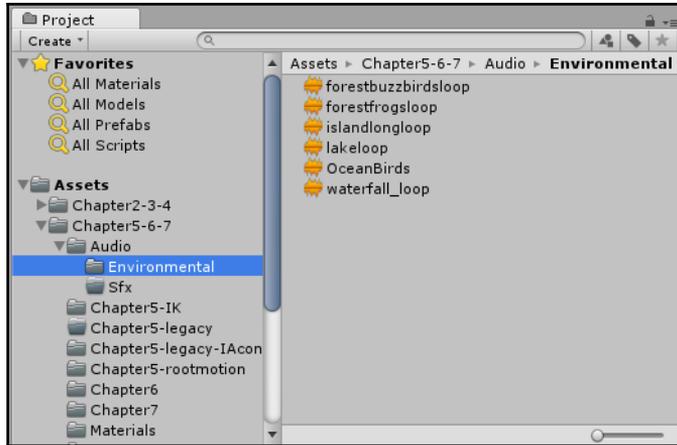
Importing the book's asset package

In this step, we will import the rest of the elements of this game, the village, the ancient artifact pieces, the health potions, and other assets from an asset package specially created for this book. With your project opened, double click on the `Chapter5-6-7.unitypackage` file and import it into Unity, or download the custom-made asset package for this book from: <http://www.packtpub.com/support>. Visit this page and select this book from the list of titles on the menu. You will then be able to enter your email address and you will be emailed a direct link to the asset package download. Once you receive this email and download the compressed file, unzip the package; you will be left with a file with the extension `.unitypackage`. Now, return to Unity and go to **Assets | Import Package | Custom Package**.

You will be presented with a file selection **Import Package** dialog window, from which you should navigate to the location on your hard drive that you saved the downloaded `unitypackage` file in. Select it and then click on **Open** to choose it.

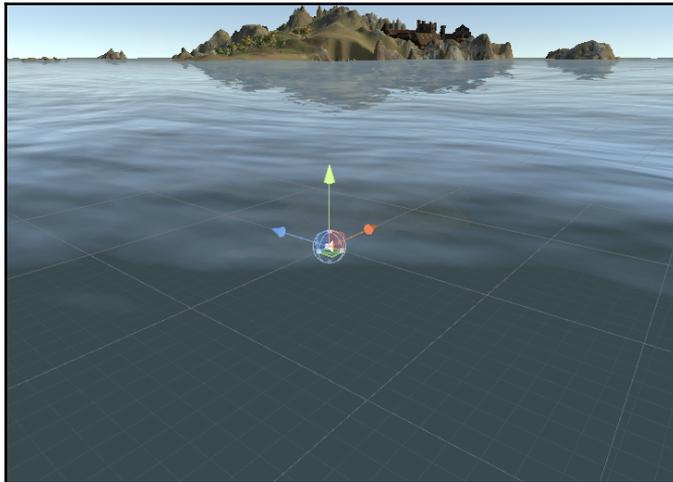
Click on **Import** to confirm that you want to add these files to your project and, after a short conversion progress bar, you should see them in your **Project** panel, contained within a parent folder named **Book Assets**. Within this, you'll need to open the `Sounds` folder you added by clicking on the gray arrow to the left of it in order to see the `hillside` sound file; select it here in order to see the **Import Settings** for this asset in the **Inspector**.

Sound files are represented in your **Project** panel by the speaker icon, as shown in the following screenshot:



Select the **OceanBirds** audio file in the **Project** view and drag it into the scene directly. This is the best way to add ambient environmental audio sources because Unity will automatically create an empty game object that you can move around and position wherever you like.

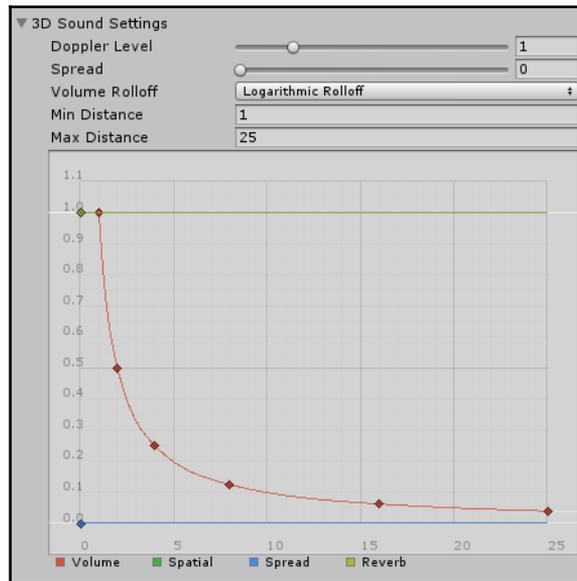
The empty game object will be created at 0,0,0 in your scene, resulting in a corner of your terrain:



Further audio settings

The **AudioSource** component has various settings for controlling how an audio clip sounds, as well as how it plays back. For our ambient `OceanBirds` sound, simply ensure that the checkboxes for **Play On Awake** and **Loop** are selected. This will play the sound clip when the player enters the scene (or level) and will continually loop the clip until the scene is exited.

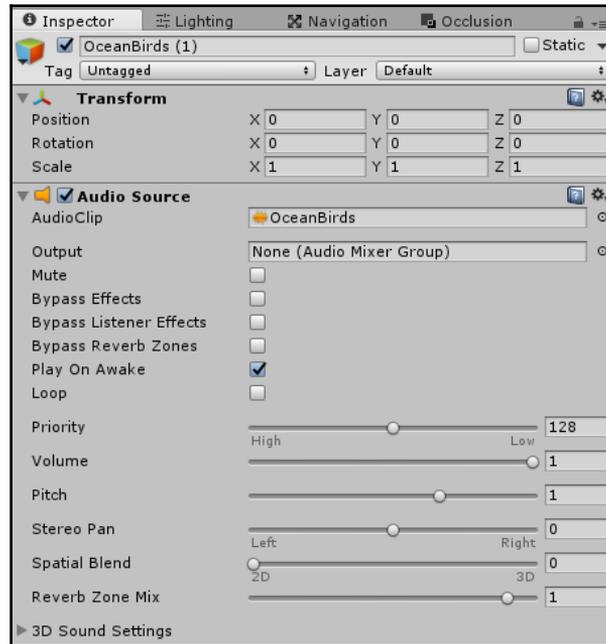
Your **Audio Source** component should look like this:



Enabling positional 3D sound and setting up curves

The **Audio Source** needs to be set up to play 3D positional sounds. The first parameter we want to change is **Spatial Blend**, and setting this value to **1** will ensure that the audio source will be fully positional. After you have done this, open the **3D Sound Settings** subpanel by clicking the small horizontal arrow before the text.

You should see the **3D Sound Settings** panel:



- **Doppler Level:** This is how audible the Doppler effect is. When it is zero, it is turned off. 1 means it should be quite audible for fast-moving objects.
- **Spread:** This sets the spread angle to 3D stereo or multichannel sound in the speaker space.
- **Volume Rolloff:** This describes how the sound will decrease its volume gradually with the distance from the **Audio Listener** and **Min Distance** and **Max Distance** adjusting the values of the curves for us. Its possible values are:
 - **Logarithmic:** Use this mode when you want a real-world rolloff
 - **Linear:** Use this mode when you want to lower the volume of your sound over the distance
 - **Custom:** Use this when you want to use a custom rolloff



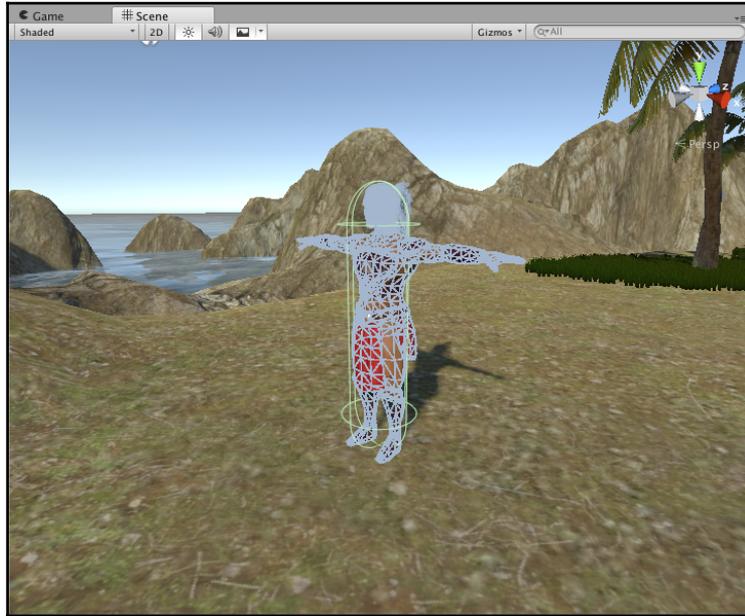
Visit: <https://docs.unity3d.com/Manual/class-AudioSource.html>, for additional information about the Audio Source component.

Step 10 – Walkabout

We are going to use our ready-made prefab from the previous chapters, `Warrior_final_RM`, and drag it into the scene.

In the **Project** view, expand the `StandardAssets` folder and then expand the subfolder called `CharacterControllers`. In here, you will find two prefabs, a ready-made `FirstPersonController` and a `ThirdPersonController` object. We'll be making a third-person viewed game, so select `ThirdPersonController`.

In the **Scene** view, zoom in to the part of the island you wish to drop your `FirstPersonController` player object onto. Your view should look something like the following screenshot:



Drag this prefab from the **Project** panel onto the **Scene** view, noting that Unity tries to position your object on top of any collider it encounters; in this instance, the in-built Terrain Collider. When you release the mouse, however, focus your view on the newly placed **Third Person Controller** by hovering over the **Scene** view and pressing `F`.

Now, press the Play button to test the game, and you should be able to walk around on your island terrain!

The default controls for the character are as follows:

- Up arrow/*W*: Walk forward
- Down arrow/*S*: Walk backward
- Left arrow/*A*: Sidestep left (also known as *Strafing*)
- Right arrow/*D*: Sidestep right
- Mouse: Look around/turn the player while walking

Once you've had a good stroll around the island, stop testing the game by pressing the Play button again.



It is crucial to stop the game when you continue editing. If you leave it playing or paused, any edits you make to the scene will be temporary, only lasting until you press Play again or quit Unity.

Step 11: Final tweaks

Now that we have added our modified version of the `ThirdPersonController` prefab, we are going to have a look at the **Multipurpose Camera Rig** in the `Camera` prefab folder.

When you drag this prefab in the scene, you will notice an info warning in the console:

There are 2 audio listeners in the scene (Main Camera). Please ensure there is always one audio listener in the scene.

To rectify this, simply remove the **Main Camera** object, as we no longer need it. To do this, select it in the **Hierarchy** and press *command + backspace* (Mac) or *Ctrl + Backspace* (Windows), right-click this object's name in the **Hierarchy**, and choose **Delete** from the pop-up menu that appears.

Now that this object is gone, we are ready for the last step: baking the NavMesh for the AI enemies.

Your island terrain is complete. Save your scene so that you do not lose any work. Go to **File | Save Scene** to save your work. Here, Unity will assume that you want to save it in the `Assets` folder of your project, and will choose that folder as the default save location, because all assets must be in this folder. To keep things neat, you can make a `scenes` subfolder and save it in there.

Congratulations, your island terrain is ready for exploration, so hit the **Play** button again and go exploring!

Just remember to press Play again to stop when you are finished to go back to editing.

Summary

In this chapter, we've explored the basics of developing your first environment. Beginning with nothing but a flat plane, you have now created a fully explorable island in a relatively short amount of time. We've also looked at lighting and sound, two core principles that you'll need to apply in every kind of game project you encounter.

Remember, you can return to the terrain tools covered in this chapter at any time in order to add more detail to your terrain, and once you feel more confident with sound, we'll return to adding further audio sources to the island later in the book.

As you continue to work through this book, you'll discover all types of additional nuances that you can bring to environments in order to further suspend players' disbelief.

We'll be looking at adding a dynamic feel to our island when we look at the use of particles in [Chapter 11, *Unity Particle System*](#), such as adding fire torches, sandstorms, and even a waterfall diving in to our small lake!

In the next chapter, we'll add some village buildings to our scene, and look at how we can trigger the animation of a door opening when a player approaches it. To do this, we'll expand upon your existing knowledge of scripting for Unity, and take our first leap into developing real game interactions.

7

Interactions, Collisions, and Pathfinding

In this chapter, we will be looking at interactions and diving into three of the most crucial elements of game development, namely, the following elements:

- **Collision detection:** This refers to detecting interactions between objects by detecting when their colliders collide with one another
- **Trigger collision detection:** This refers to detecting when a collider set to trigger mode has other colliders within its boundary
- **Raycasting:** This refers to drawing a line (or vector) in the 3D world from one point to another, in order to detect potential intersections with one or more colliders

In order to learn about these three topics, we will add some *abandoned village* buildings and a prison wall model to our island, and learn how to write code for implementing user interactions by making a house door open when the player character walks toward it. We will look at how to achieve this with each of the listed techniques before choosing the most appropriate one for use in our game. After placing some other additional models, we will bake the NavMesh, and eventually bake it again after adding new static models to the scene, giving a look at:

- Importing and placing some additional 3D model over our Terrain
- Setting up collisions and tweak colliders in the scale and offset positions
- Using what we learned about the Navigation System and baking the NavMesh
- Learning some NavMesh baking tips for optimal AI usage

Digital content creation applications

Given that 3D design is an intensive discipline in itself, it is recommended that you invest in a similar tutorial guide for your application of choice. If you're new to 3D modeling, then here is a list of DCC software packages currently supported by Unity:

- Maya
- 3D Studio Max
- Cheetah3D
- Cinema 4D
- Modo
- Blender
- Sketchup
- Lightwave

These are the nine most suited applications, as recommended by Unity Technologies. The main reason for this is that they export models in a format that can be automatically read and imported by Unity once saved into your project's `Assets` folder. These application formats will carry their meshes, textures, animations, and bones (a form of skeletal rigging for characters) across to Unity, whereas some smaller packages may not support animation with bones upon import to Unity.

For additional info about 3D model importing, visit: <https://docs.unity3d.com/Manual/HOWTO-importObject.html>.

Common import settings for models

In the **Project** view, expand the book `Assets` folder to show the `Models` folder inside the `Chapters7-9-10` folder. You should find a folder called `village`. Select the `hut_model.fbx` model to see its **Import Settings** in the **Inspector**.

Before you introduce any model to the active scene, you should always ensure that its settings are as you require them to be in the **Inspector**. When Unity imports new models to your project, it is interpreting them with its **FBX Importer** settings. Using the **FBX Importer**, you can select your model file in the **Project** window and adjust settings for its **Meshes**, **Materials**, and **Animations** import settings before your 3D object becomes part of your game once it is added to the scene.

Model

In the **Model** section of the **FBX Importer**, you can specify the following things:

- **Scale Factor:** This may need to be set to a value of 1, 0.1, 0.01, or something similar, the simplest way to approach this is to export a cube primitive, depending on the scale of models exported by your chosen 3D modeling application. You could check that your exported cube primitive model matches the size of the Cube primitive Unity creates when you go to the top menu | **GameObject** | **3D Objects** | **Cube** at Scale 1,1,1 in the Transform component. This will allow you to set up your modeling workflow with the appropriate world scale. In general, we tend to consider 1 Unity unit equivalent to 1 meter, but this might not fit all types of games or projects. If you want your models to be scaled differently, then you can adjust them here, before you add the model to the scene, with this option. However, you can always scale objects once they are in your scene using the **Transform** component's **Scale** settings.
- **Mesh Compression:** This drop-down menu allows you to specify settings to compress the complexity of your mesh as it is interpreted by Unity. This is useful for optimizing your game and should generally be set to the highest setting possible without the model's appearance being affected too drastically.
- **Import Blendshapes:** This checkbox will tell Unity to import `Blendshapes` for this model. If this information is present in the file, it will be imported and made available. This is generally used for character face expression/animation.
- **Generate Colliders:** This checkbox will find every individual mesh of the model and assign a mesh collider to it. A mesh collider is a complex collider that can fit to complex geometric shapes and, as a result, is the usual type of collider you would expect to want to apply to all parts of a map or 3D model of a building. However, mesh colliders are also the most computationally expensive way of adding colliders to a model and, as such, should not be used when primitive shaped colliders will suffice.
- **SwapUVs:** UV coordinates define, in the 3D model, how the texture should be displayed on each triangle that compose the mesh. A mesh can have more than one UV channel, allowing two layers of texture with different UV coordinates, hence to implement static lightmapping on the meshes. Sometimes, when importing 3D models the wrong channel is picked up by Unity, resulting in a failure to map correctly. This checkbox switches the channel 0 and channel 1 (commonly reserved for colormap and lightmap, respectively) to the correct order to rectify this problem.

- **Generate Lightmap UVs:** This checkbox means that Unity will plot coordinates from the mesh to allow the lightmapping tool to successfully bake a texture based upon the shape of an object.
- **Normals and Tangents:** Normals are data not always included in an exported 3D model. This data contains the directions of the triangle's faces. To implement complex shaders, such as dot3 normal maps bump mapping, this data is needed. When the model doesn't contain this data, and you need to use a special shader on it, Unity will rise a warning message. You can fix this problem by setting to **Calculate Normals** in the settings. You can also set to disable the import of this data, or to import it from file.
- **Split Tangents:** This setting allows corrections by the engine for models imported with incorrect bump mapped lighting. Bump mapping is a system utilizing two textures: one is a graphic to represent a model's appearance and the other is a height map. By combining these two textures, the bump map method allows the rendering engine to display flat surfaces of polygons as if they have 3D deformations. When creating such effects in third-party applications and transferring to Unity, sometimes lighting can appear incorrectly, and this checkbox is designed to fix that by interpreting their materials differently.

Materials

The **Materials** section allows you to choose how to interpret the materials created in your third-party 3D modeling application. The user can choose either of the following options from the **Generation** drop-down menu:

- **Per Texture** (this creates a Unity material for each texture image file found)
- **Per Material** (this creates materials only for the existing materials in the original file)

Animation

The **Animation** section of the **Importer** allows you to interpret the animations created in your modeling application in a number of ways. From the **Generation** drop-down menu, you can choose the following methods:

- **Don't Import:** Sets the model to feature no animation
- **Store in Original Roots:** Sets the model to feature animations on individual parent objects, as the parent or root objects may import differently in Unity

- **Store in Nodes:** Sets the model to feature animations on individual child objects throughout the model, allowing more script control of the animation of each part
- **Store in Root:** Sets the model to only feature animation on the parent object of the entire group

Wrap Mode

The **Wrap Mode** drop-down menu allows you to choose several different settings for how animations will play back (once, in a loop, and so on), these can be set individually on the animations themselves, so if you do not want to adjust a setting for all, then this can be left on **Default**.

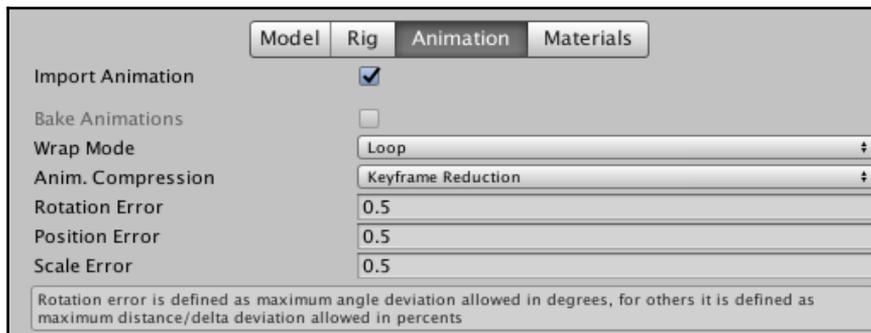
Checking **Split Animations** will mean that when creating models to be used with Unity, animators create timeline-based animation, and by noting their frame ranges, they can add each area of animation to their timeline by specifying a name and the frames in which each animation takes place. The specified animation name can then be used to call individual animation when scripting.

Animation Compression

The **Animation Compression** section handles the compression and correction of errors that may occur on import from a 3D modeling app.

The **Animation Compression** drop-down menu has three settings:

- **Off**
- **Keyframe Reduction**
- **Optimal:**



Furthermore, using keyframe compression, Unity will attempt to save on file size. As this may result in errors, you can use the three *error values*: Rotation Error, Position Error, and Scale Error to set the precision you'd like, with smaller numerical values giving tighter and more precise animation. Now that we have an overview, let's get started with our first externally created model asset, the old man's hut model.



It is not recommended that you set **Animation Compression** to **Off** unless you need total precision in your animations, as Unity will, by default, reduce keyframes to save memory using **Keyframe Reduction**.

Setting up the hut model

In the **Project** view, open the `Book Assets` folder and within the `Models` folder, select `hut_model`. We will use the **FBX Importer** in the **Inspector** to adjust the settings for the `hut_model` prefab. Leave the defaults settings, ensuring the following conditions are met:

- **Meshes-Scale Factor** is set to 30
- **Generate Colliders** and **Generate Lightmap UVs** options are checked
- **Materials-Generation** is set to **Per Texture**

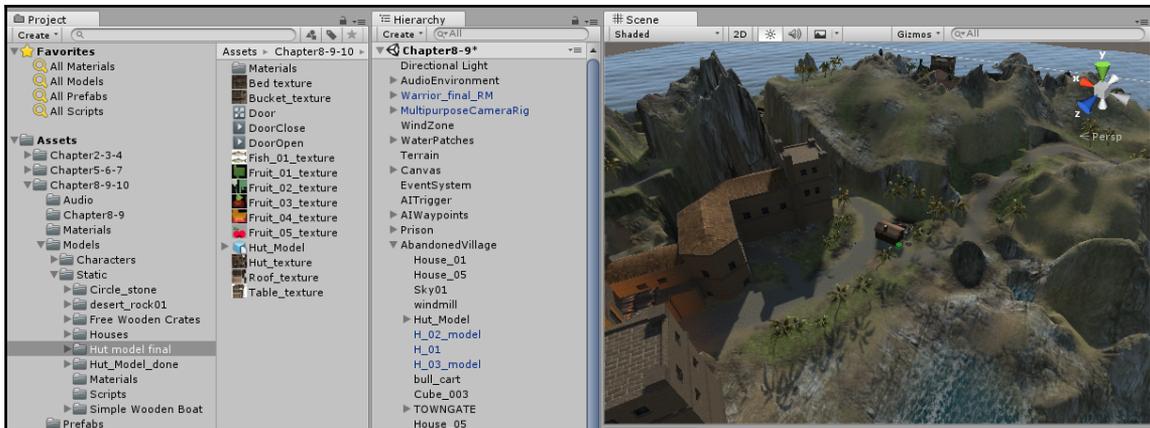
For the animation, we will set up an Animator Controller with two states that will carry the following two attached door animation files in the same folder:

- `dooropen`
- `doorsclose`

Provided that your hut model is set up as described earlier, click the **Apply** button to confirm these import settings and you're all done; the model should be ready to be placed into the scene and used in our game.

Adding the model

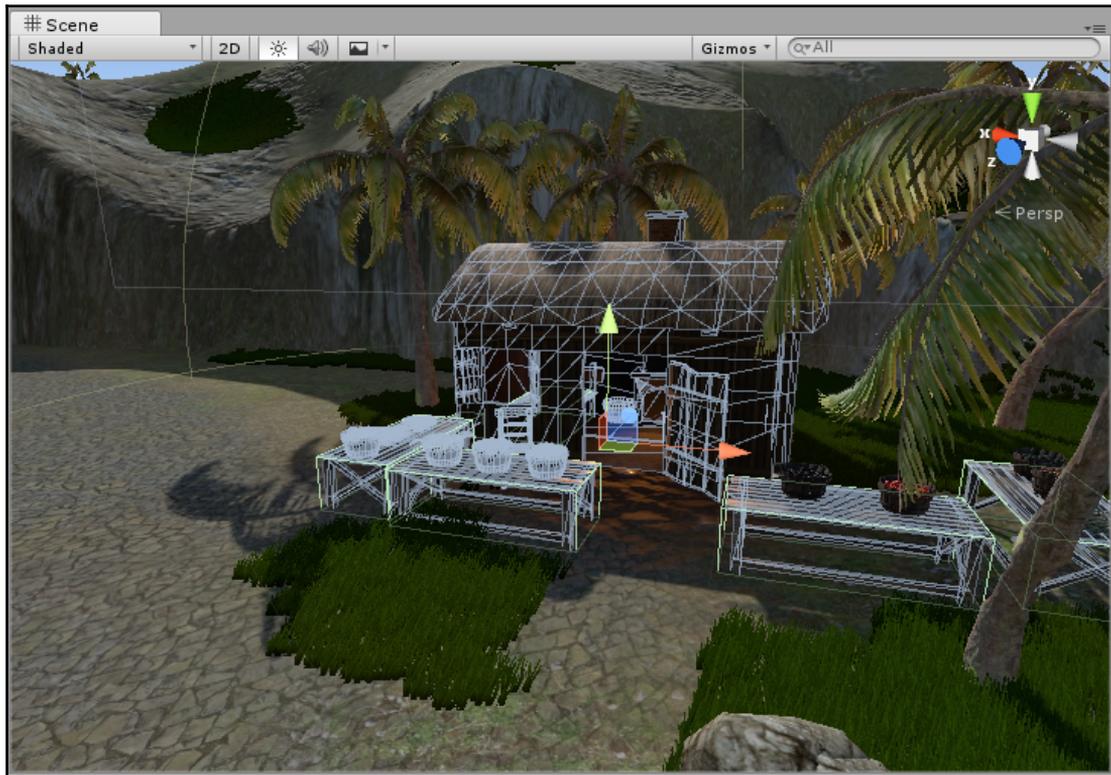
Before we begin to use both the collision detection and ray casting techniques to open the door of our hut, we'll need to introduce it to the scene. To begin, drag `Hut_Model.fbx` from the **Project** view to the Hierarchy or the **Scene** view, trying to drop it onto an empty area of land (see the suggested position in the next screenshot) near the prison walls models. You'll note that when dragging 3D objects to the **Scene** view, Unity positions them by dropping them onto any collider that it finds beneath your dragged cursor. In this case it's the in-built Terrain Collider, but often you'll need to do your own tweaking using the **Translate** tool (*W*) once your objects are in the scene:



Once the hut model is in the **Scene**, you'll notice that its name has also appeared in the **Hierarchy** and that it has automatically become selected. To get a better look at it, hover your mouse over the **Scene** view now and press *F* to focus the view on this object.

Positioning the model

As your terrain design may be somewhat different to the one shown in the images in this book, select the **Transform** tool and position your `outPost` in a free, flat area of land by dragging the axis handles in the scene. If you do not have a flat area, go back to the terrain tools by selecting the **Terrain** object in the **Hierarchy** and using the tools we looked at in Chapter 6, *Creating the Environment*, to flatten a particular area using the **Paint Height** tool to paint to the ground height a level of 30 meters:

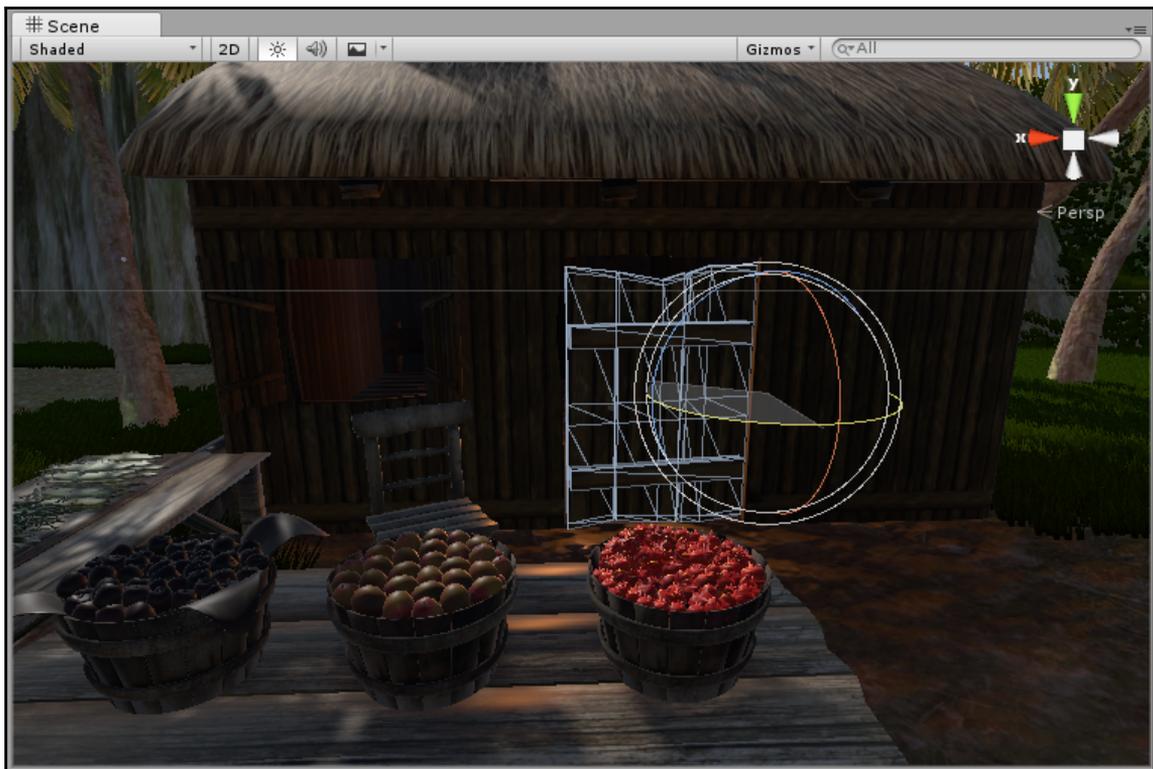


In the preceding image, the `hut_model` `GameObject` is at the position of $(240, 12, 252.3289)$, see the **Transform** component, but you may need to position it manually to match terrain alignment and also show the hut floor instead of the terrain. Remember that, once you have positioned using the axis handles in the **Scene** window, you can enter specific values in the **Position** values of the **Transform** component in the **Inspector**.

We will ensure that the front of the building is facing the prison jail entrance. To do this, simply rotate the object by 176 on the y axis using the Transform component in the **Inspector**.

Manually adding the colliders

In order to open the door, we need to identify it as an individual object when it is collided with by the player, this can be done because the object has a Collider component and through this we can check the object for its name or a specific tag. Expand the `hut_model` parent object by clicking the dark gray arrow to the left of its name in the **Hierarchy**. You should now see the list of all child objects beneath it. Select the object named **door** and then, with your mouse cursor over the **Scene** window, press *F* on the keyboard to focus your view on it. If you are not shown the door face-on, simply hold the *Alt* key and drag to rotate your view around it until you see what you want:



The hut with the animation ready—separates object door selected with the rotation tool enabled

You should now see the door in the **Scene** window, and as a result of selecting the object, you should also see its components listed in the **Inspector**. You should note that one of the components is a Mesh Collider. This is a detailed collider assigned to all meshes found on the various children of a model when you select **Generate Colliders**, as we did for the `hut_model` asset earlier.

A Mesh Collider component is assigned to each child element because Unity does not know how much detail will be present in any given model you choose to import. As a result, it defaults to assigning a Mesh Collider component for each part of the model, as they will naturally fit to the shape of the mesh. Our door is simply a cube, hence, we should replace this Mesh Collider with a simpler and more efficient Box Collider. This is a very important step when importing even simple meshes, as you will often see better performance with primitive colliders than mesh colliders.

For our case, we will use two colliders on the door, a regular one and one set as a trigger.



A trigger collider is a primitive collider with the Trigger flag enabled. This kind of collider will not block other objects or generate any collision event, but will generate an `OnTriggerEnter` event whenever they are entered by a physic object.

From the top menu, go to **Component | Physics | Box Collider**.

You will then receive two prompts. First, you will be told that adding a new component will cause this object to lose its connection with the saved prefab asset. This dialog window, titled **Losing Prefab Link**, simply means that your copy in the **Scene** will no longer match the original asset, and as a result, any changes made to the asset in the **Project** view in Unity will not be reflected in the copy in the **Scene**. Simply click the **Add** button to confirm that this is what you want to do.

This will happen whenever you begin to customize your imported models in Unity and there is nothing to worry about. This is because, generally, you will need to add components to a model, which is why Unity gives you the opportunity to create prefabs.

Unity GameObjects can have more than one collider attached, giving the ability to mix *trigger area* colliders and physic collision colliders together on the same GameObject; we will use this feature for the door, leaving its Mesh Collider component so that the door will block the way when it is closed, but it will also have a Box Collider set as **Is Trigger** on board that will listen to trigger events.

You will now see a green outline around the door representing the **Box Collider** we have added. Looking at the next image, you will notice the **Is Trigger** flag checked in the **Inspector**:



If you click the **Edit Collider** button, just above the **Is Trigger** flag the Box Collider handles (the green dots in the image) will become enabled (and visible) so that you can actually use them to visually resize the collider. Try to drag one of the handles to see one side of the box resize in one direction. You can achieve the same results by setting values in the Center and Size values just below the Physic Material slot, but sometimes it's just harder that way.



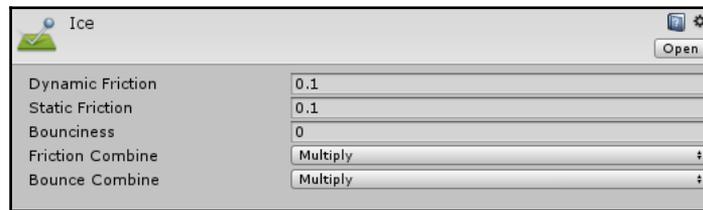
A **Box Collider** is an example of a **Primitive Collider**, so called because it is one of the several scalable primitive shape colliders in Unity, including box, sphere, capsule, and wheel, that have predetermined collider shapes. In Unity, all primitive colliders are shown with this green outline. You may have noticed this when viewing the character controller collider, which is technically a capsule collider shape and, as such, is also displayed in green.

Physic Material

A physic material can be specified to define how the Collider surface should react to collision.

To simulate rubber, or an icy slippery surface, with more or less friction, you can optionally specify a Physic Material for your collider.

This image shows the settings in the **Inspector** for the **Ice** Physic Material:



Adding audio

As the door will be automatically opening and closing, we'll need to add an audio source component to allow the door to emit sound effects as it is opened and closed. With the door child object still selected, choose **Component | Audio | Audio Source** from the top menu. We don't need to add an audio clip to this Audio Source component because we will manage this dynamically from the code.

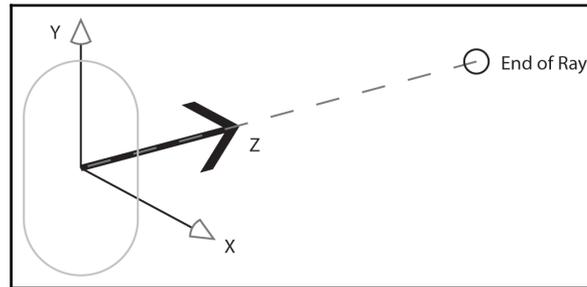
The hut model object is now ready to be interacted with by our player character, so we will begin writing code for implementing collision detection.

Collisions and triggers overview

To detect physical interactions between GameObjects, the most common method is to use a Collider component, an invisible net that surrounds an object's shape and is in charge of detecting collisions with other objects. The act of detecting and retrieving information from these collisions is known as collision detection.

Not only can we detect when two colliders interact (collision detection), but we can detect when particular colliders are intersecting (trigger-mode collision detection), and even preempt a collision and perform many other useful tasks by utilizing a technique called **Ray casting**. Ray casting, in contrast to detecting intersecting 3D shaped colliders, draws a Ray. Simply put, it is an invisible (non-rendered) vector line between two points in 3D space, which can also be used to detect an intersection with a GameObject's collider.

Ray casting can also be used to retrieve lots of other useful information, such as the length of the ray (therefore, distance) and the point of impact of the end of the line, for example, where a bullet might impact another object in a game scenario:



In the given example, a ray facing the forward direction from our character is demonstrated. In addition to the direction, a ray can also be given a specific length or be allowed to cast until it finds an object.

Over the course of the chapter, we will work with the outpost model that we have added to the terrain. As this asset has been animated for us, the animation of the outpost's door opening and closing is ready to be triggered. This can be done either with collision detection, trigger collision detection, or ray casting, and we will explore what you will need to do in order to implement each approach.

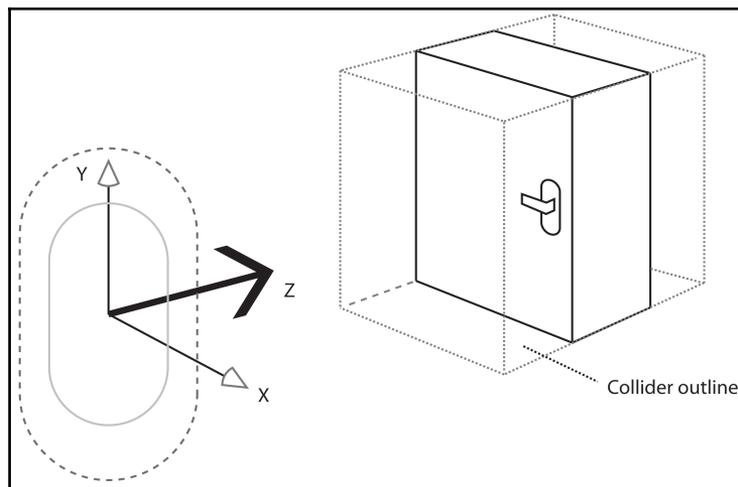
Let's begin by looking at collision and trigger detection, when it may be appropriate to use a trigger-mode collider, or ray casting instead of, or in complement to, standard collision detection. In the early part of this chapter, we will look at these three approaches in the context of opening the `hut_model` door, before moving on to implement each approach.

When objects collide in the game engine, information about the *collision event* becomes available. By recording a variety of information on the moment of impact, the engine can respond in a realistic manner. For example, in a game involving physics, if an object falls to the ground from a height, then the engine needs to know which part of the object hit the ground first. With that information, it can correctly and realistically control the object's reaction to the impact.

Of course, Unity handles these kinds of collisions and stores the information on your behalf, and you only have to retrieve it in order to do something with it.

In the example of opening a door, we will need to detect collisions between the player character's collider and a collider on or near the door. It will make little sense to detect collisions elsewhere, as we will likely need to call the animation of the door when the player is near enough to walk through it or to expect it to open for them.

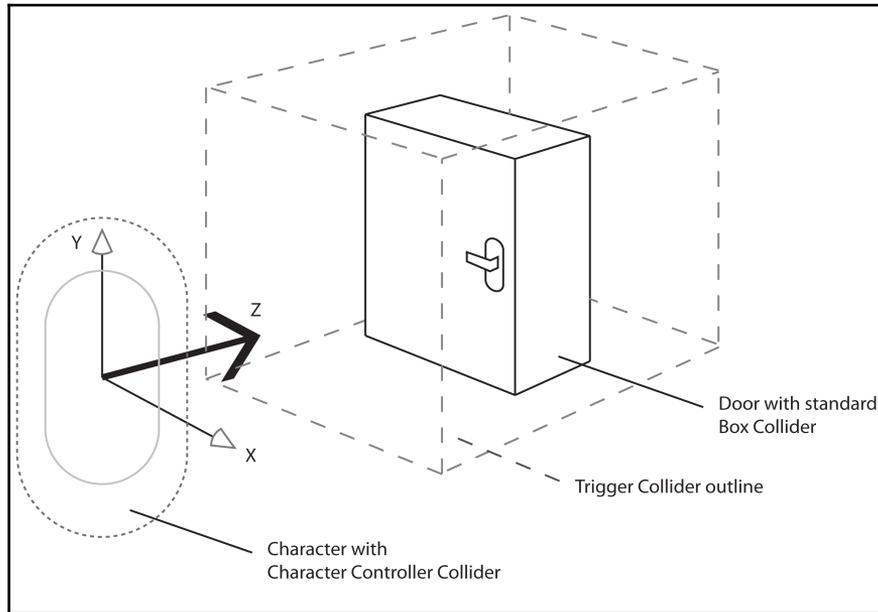
As a result, we will check for collisions between the player character's collider and the door's collider. However, we will need to extend the depth of the door's collider so that the player character's collider does not need to be pressed up against the door in order to trigger a collision, as shown in the following illustration. However, the problem with extending the depth of the collider is that the game interaction becomes unrealistic with this setting:



In the example of our door, the extended collider protruding from the visual surface of the door means that we will bump into an invisible surface that will cause our character to stop in their tracks. Although we will use this collision to call the opening of the door through animation, the initial bump into the extended collider will seem unnatural to the player and thus detract them from their immersion in the game.

In order to avoid this, colliders can be set to **Trigger** mode, a mode where colliders intersecting the trigger collider can be detected, but will not be repelled as if it were a physical object. These are often used to detect when a player character is in a particular area. With this approach, two colliders may be used: one collider placed on the door that fits its exact shape and size, while another larger collider is placed around this object and set as a **Trigger** (see the next diagram).

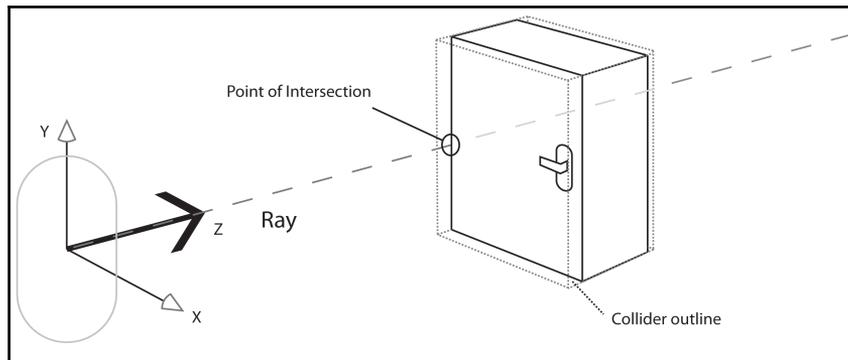
In this approach, we will use the trigger collider to detect the presence of the player and therefore call actions, such as the animation, on the door when it opens. Meanwhile, the standard collider on the door itself will react to objects hitting the door directly, perhaps if the player character runs into the door before the animation finishes or if an object is thrown at the door and must bounce off:



So, while collision detection will work perfectly well between the player character collider and the door collider using a trigger collider occupying space around the door, we are able to detect the player character before they bump into the door itself. In addition to the use of triggers to detect the intersection of colliders, sometimes we must detect a potential collision much further away from an object; for this, we can use a technique called ray casting.

Ray casting

While we can detect collisions between the player character's collider and a collider that fits the door object or a trigger collider near the door, we can also check whether the player collider is about to intersect with another collider by casting a ray forward from where the player is facing. This means that when approaching the door, the player need not walk right up to it, or walk into an extended trigger collider, in order for the door to be opened:

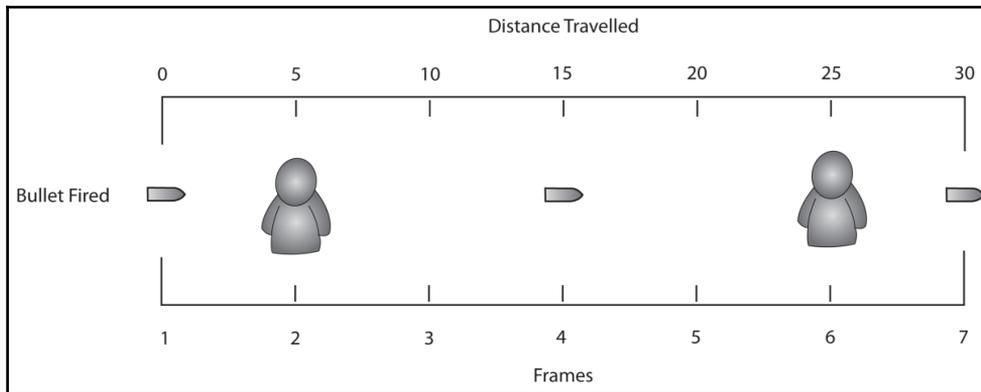


Ray cast in action from the player to the door's collider

However, the drawback of this approach is that it means the player must be facing the door's collider in order for the ray to intersect it (given that the ray is cast in the forward direction of the player), which, as you'll likely know, is not how an automatic door works, it simply detects motion near it.

Despite its drawbacks and the collision detection approach, as you will discover in your time learning game development in Unity, it is often good to try a number of approaches to a problem in order to decide which is the most efficient one. For this reason, we will learn collision detection and ray casting to open the door alongside the more appropriate trigger collision detection.

It is also important to learn ray casting, as it is often used in other parts of game development to solve specific problems, such as preempting a potential collision where two colliders intersecting may not be appropriate, or for implementing an AI field of view feature, which we will see in the next chapter. Let's look at a practical example of this problem:



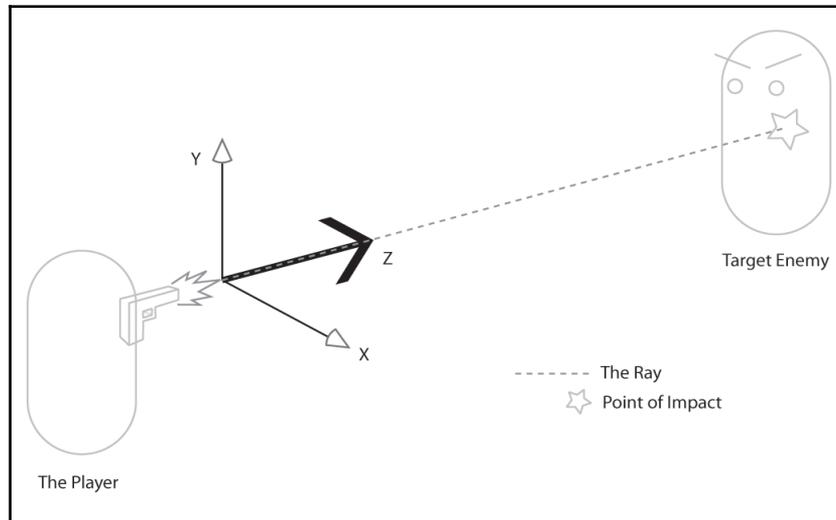
In the preceding illustration, a bullet is fired from a gun. In order to make the bullet realistic, it will have to move at a speed of 500 feet per second. If the frame rate is 25 frames per second, for example, the bullet will move at 20 feet per frame. The problem with this is that a person is about two feet in diameter, which means that the bullet will very likely miss the enemies at 5 and 25 feet away that a player is expecting to hit. This is where prediction comes into play.

Predictive collision detection

Instead of checking for a collision with an actual bullet object, we find out whether a fired bullet will hit its target. By casting a ray forward from the gun object (thus using its forward direction and therefore the bullet's trajectory) on the same frame that the player presses the fire button on, we can immediately check which objects intersect the ray.

We can do this because rays are drawn immediately. Think of them like a laser pointer, when you switch on the laser, you do not see the light moving forward because it travels at the speed of light—to us, it simply appears.

Rays work in the same way so that whenever the player in a ray-based shooting game presses fire, they draw a ray in the direction that they are aiming. With this ray, they can retrieve information on the collider that is hit. Moreover, by identifying the collider, the `GameObject` itself can be addressed and scripted to behave accordingly. Even detailed information, such as the point of impact, can be returned and used to affect the resultant reaction, for example, causing the enemy to recoil in a particular direction:



In our shooting game example, we will likely invoke scripting to kill or physically repel the enemy whose collider the ray hits, and as a result of the immediacy of rays, we can do this on the frame after the ray collides with or *intersects* the enemy collider. This gives the effect of a real gunshot because the reaction is registered immediately.

It is also worth noting that shooting games often use the otherwise invisible ray casts to get position data in order to render visible lines to help with aim and give the player visual feedback. However, do not confuse these lines with ray casts, because the rays are simply used as a path for such a line's rendering.



Continuous collision detection

If you are working with projectiles that are fast moving but not at the speed of a bullet, the frame miss problem may still occur. This can be corrected by setting the **Collision detection** type of your Rigidbody component to **Continuous** or **Continuous Dynamic** depending upon what other objects your Rigidbody interacts with.

Now, let's return to our door example and make use of the approaches we just outlined in order to make the door of the `old house` `GameObject` interactive.

Opening the old man's hut door

In this section, we will look at the three different approaches for interacting with the door in order to give you an overview of the techniques that will become useful in many other game development situations:

1. In the first approach, we will use collision detection, a crucial concept to get to grips with as you begin to work on games.
2. In the second approach, we'll implement a simple ray cast forward from the player, another important skill to learn, which means we can detect interactions without colliders actually colliding.
3. Finally, we'll implement the most efficient approach for this scenario, using a separate trigger collider to call the animation on the door.

This means that you will have tried three different approaches to the problem and you will have code to refer to once you begin your own development.

Approach 1 – collision detection

To begin writing the script that will play the door-opening animation and thereby grant access to the outpost, we need to consider which object to write a script for.

In game development, you should see your player character as a unique entity that all other objects are awaiting interaction with. Rather than establishing a master script for the player that will account for all eventualities, we can write smaller scripts that simply know what to do when they encounter the player.

Creating new assets

Before we introduce any new kind of asset into our project, it is a good practice to create a folder in which we will keep assets of that type. In the **Project** view, click the **Create** button and choose **Folder** from the drop-down menu that appears.

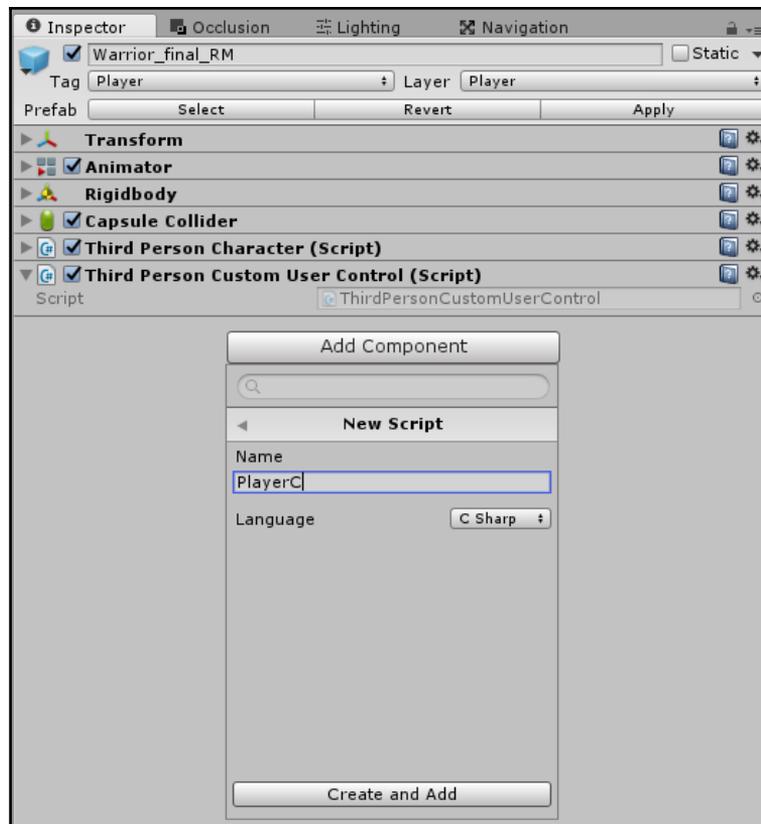
Rename this folder as **Scripts** by selecting it and pressing *return* (Mac) or *F2* (PC). Move your scripts file into this folder by dragging and dropping it to keep things neat.

Next, create a new C# file within this folder simply by leaving the `Scripts` folder selected and clicking the **Project** view's **Create** button again, this time choosing the relevant language.



By selecting the folder that you want a newly created asset to be in before you create them, you will not have to create and then relocate your asset, as the new asset will be made within the selected folder.

Rename the newly created script from the default, `NewBehaviourScript`, to `PlayerCollisions`. Script files have file extensions such as `.cs` for C#, but Unity Project view hides file extensions, so there is no need to add it when renaming your assets. Another way to add a new C# monobehaviour class is through the Add Component button on the inspector directly:



Character collision detection

To start editing the script, double-click its icon in the **Project** view to launch it in the default script editor, Visual Studio 2017 Community Edition in our case.

Working with OnCollisionEnter

By default, all new C# classes include the `Update()` and the `Start()` methods, and this is why you'll find it present when you open the script for the first time. However, we are about to take a look at another MonoBehaviour base event method now, called `OnCollisionEnter`, a collision detection event method, a raised when a collision happens on the current `GameObject` collider, such as the one that will detect our player character collisions.



In C# MonoBehaviour classes we should always ensure that the class name of the script matches the filename before you continue working. So pay attention when renaming classes to also rename the script file. When a new class is created it is automatically changed by Unity from `NewBehaviorScript` to `PlayerCollisions`.

Let's kick off by declaring variables that we can utilize throughout the script.

Our script begins with the definition of two private variables and three public member variables. Their purposes are as follows:

- `doorIsOpen`: This is a private `true/false` (Boolean) type variable acting as a switch for the script to check whether the door is currently open.
- `doorTimer`: This is a private floating-point (decimal-placed) number variable, which is used as a timer so that once our door is open, the script can count a defined amount of time before self-closing the door.
- `doorOpenTime`: This is a public floating-point (potentially decimal) numeric public member variable, which will be used to allow us to set the amount of time that we want the door to stay open in the **Inspector**.
- `doorOpenSound/doorShutSound`: These are two public member variables of the `AudioClip` data type for allowing sound clip drag-and-drop assignment in the **Inspector**.

Define these variables at the top of the `PlayerCollisions` class body:

```
bool doorIsOpen = false;
float doorTimer = 0.0f;
public float doorOpenTime = 3.0f;
public AudioClip doorOpenSound;
public AudioClip doorShutSound;
```

Next, we'll leave the `Start()` and the `Update()` methods for later, while we establish the collision detection event method itself.

Move down after the end of the `Update()` method and write the following:

```
void OnCollisionEnter(Collision hit) { }
```

This establishes a new standard Unity API event method, called `OnCollisionEnter`. This collision detection function will fire player character collision events each time a collision of its collider is detected.

Its only argument, `hit`, is a collection of the `Collisions` variable type, which is a class that stores information on any collision that occurs. By addressing the `hit` variable, we can query information on the collision, including, for starters, the specific `GameObject` our player has collided with.

We will do this by adding an `if` statement to our function. So, within the function's curly brackets, that is, between `{` and `}`, add the following statement:

```
void OnCollisionEnter(Collision hit)
{
    if(hit.gameObject.tag == "playerDoor" &&& doorIsOpen ==
        false)
    {
    }
}
```

With this `if` statement, we are checking two conditions: first, that the object we hit is tagged with the `playerDoor` tag, and second, that the `doorOpen` variable is currently set to `false`. Remember that two equals symbols (`==`) are used as a comparative and the two ampersand symbols (`&&`) simply say *and also*.

The end result means that if both the conditions are met, because the player hit the hut trigger, which is attached on the `GameObject` that we have previously tagged `playerDoor` and we have also not already opened the door (`doorIsOpen == false`), it will carry out a set of instructions.

We have utilized the dot syntax to address the object that we are checking for collisions with by narrowing down from `hit` (our variable storing information on collisions) to `gameObject` (the object hit) to the `tag` on that object.

If this `if` statement's conditions are met, then we need to carry out a set of instructions to open the door.

This will involve playing a sound, playing one of the animation clips on the model, and setting our `doorOpen` Boolean variable to `true`. As we will call multiple instructions and may need to call these instructions as a result of different conditions when we implement the ray casting approach, we will place them in our own custom function, called `OpenDoor`.

Writing the `OpenDoor()` method

Storing sets of instructions you may want to call at any time should be done by writing your own functions. Instead of having to write out a set of instructions or commands many times within a script, writing your own functions containing the instructions means that you can simply call that function at any time to run that set of instructions again. This also makes tracking mistakes in code, known as **debugging**, a lot simpler as there are fewer places to check for errors.

In our collision detection function, we wrote a call to a function named `OpenDoor`. The brackets after `OpenDoor` are used to store parameters we may want to send to the function. Using a function's brackets, you may set additional behaviors to pass to the instructions inside the function. We looked at an example of this earlier in the book; to remind yourself, go back to [Chapter 4, *Player Controller and Further Scripting*](#).

We'll take a look at this in detail later in the chapter. The brackets of `OpenDoor()` contain a single argument—a `GameObject` type reference that is sending the currently-collided with object using `hit.gameObject` as a reference.

Declaring the function

To write the function that we need to call, begin by writing the following code after the closing curly bracket, `}`, of the `OnCollisionEnter` method:

```
void OpenDoor(GameObject door){ }
```

Here, the function has the corresponding argument named `door` that is awaiting a reference of the `GameObject` type, which we already know is being sent to it by the collision detection we just wrote.

Much in the same way as the instructions of an `if` statement, we place any instructions to be carried out when this function is called within its curly brackets.

Checking the door status

One condition of the `if` statement within our collision detection function was that our `doorIsOpen` Boolean variable must be `false`. In order to stop this function from recurring, the first command inside our `OpenDoor()` function is to set this variable to `true`. This is because the player character may collide with the door several times when bumping into it, and without this Boolean, they could potentially trigger the `OpenDoor()` function many times, causing sound and animation to recur and restart with each collision. By adding in a variable that when `false` allows the `OpenDoor()` function to run and then disallows it by setting the `doorIsOpen` variable to `true` immediately, any further collisions will not retrigger the `OpenDoor()` function.

Add the following line to your `OpenDoor()` function now by placing it between the curly brackets:

```
doorIsOpen = true;
```

Playing an audio event

Our next instruction is to play the audio clip assigned to the variable, called `doorOpenSound`. To do this, add the following line to your function by placing it within the curly brackets:

```
door.GetComponent<Audio>().PlayOneShot(doorOpenSound);
```

Here, we are addressing the Audio Source component attached to the `GameObject` currently contained in the `door` argument, which you'll remember is the `playerDoor` tagged object that is hit in the collision detection. Addressing the audio source using the term `audio` gives us access to four functions: `Play()`, `Stop()`, `Pause()`, and `PlayOneShot()`.



We are using `PlayOneShot` because it is the best way to play a single instance of a sound, as opposed to playing a sound and then switching clips, which would be more appropriate for continuous music than for sound effects.

In the brackets of the `PlayOneShot` command, we pass the `doorOpenSound` variable, which will play whatever sound file is assigned to that variable in the **Inspector**. This is how the `OpenDoor()` method will look:

```
void OpenDoor(GameObject door)
{
    doorIsOpen = true;
    door.GetComponent<Audio>().PlayOneShot(doorOpenSound);
}
```

Finally we will call the method when the conditions checked in the `OnCollisionEnter` method are true:

```
void OnCollisionEnter(Collision hit)
{
    if (hit.gameObject.tag == "playerDoor" &&& doorIsOpen ==
        false)
    {
        OpenDoor();
    }
}
```

Testing the script

Before we continue to write the script, assuming that it works thus far, let's save it and test it out. So far, all that our script should do is detect a collision between the player object, `Warrior_RM`, and the door child object of the `hut_model`. When this occurs, it is set to play a sound that we must assign to our `doorOpenSound` public variable.

Go to **File | Save** in MonoDevelop and switch back to Unity. We must assign this script to our First Person Controller, but first, ensure that it is free of errors. Check the bar at the bottom of the Unity interface because this is where any errors made in the script will be shown. The bottom bar of the interface shows the latest line output by the **Console** window in Unity (**Window | Console**). If there are any errors, then double-click the error and ensure that your script matches the previous snippets. As you continue to work with scripting in Unity, you'll get used to using error reporting to help you correct any mistakes that you may make.

The best place to begin double-checking that you have not made any mistakes in your code is to ensure that you have an even number of opening and closing curly brackets—this means that all functions and statements are correctly closed.

If you have no errors, then simply select the object you want to apply the script to in the **Hierarchy**—our `Warrior_RM` player hero object. Attach the script to this object using one of the various methods we looked at in the previous chapters, the simplest method is just to drag the script's icon and drop it onto the name of the object that you want to apply it to, the `Warrior_RM`. Unity will now prompt you with a **Losing prefab** dialog window that simply asks whether you want to make the object in your scene different to the original asset in the **Project**. This is standard practice, so just click **Continue**.

Now, select the `Warrior_RM` GameObject in the **Hierarchy** and you should then see the component in the **Inspector**:



Now, expand the **Book Assets | Sounds** folder in the **Project** view and you will see audio clips named `door_open` and `door_shut`. Keep the `Warrior_RM` player controller object selected so that you can see the **Door Open Sound** and **Door Shut Sound** public variables on the **Player Collisions (Script)** component, and drag and drop these audio clips from the **Project** view to the relevant public variables in the **Inspector**. Once assigned, they should look like this:



Now we're ready for some action! Navigate to **File | Save Scene** in Unity to update our scene and then test the game by pressing the Play button at the top of the interface.



If you do not assign a clip to the **Door Open Sound** public variable, when the collision occurs in the script, it will attempt to play an audio clip but find that none is assigned. This will result in a **Null Reference Exception** error, literally speaking, a reference that is null—meaning not set. This can often occur if you set a variable to public and forget to assign it or if you have set up a reference in a script but it is not assigned to.

Walk the `Warrior_RM` player until it reaches the `hut_model` and try to interact with the door. You'll notice that you cannot interact with the door collider until the player's Capsule Collider is pressed up against it.

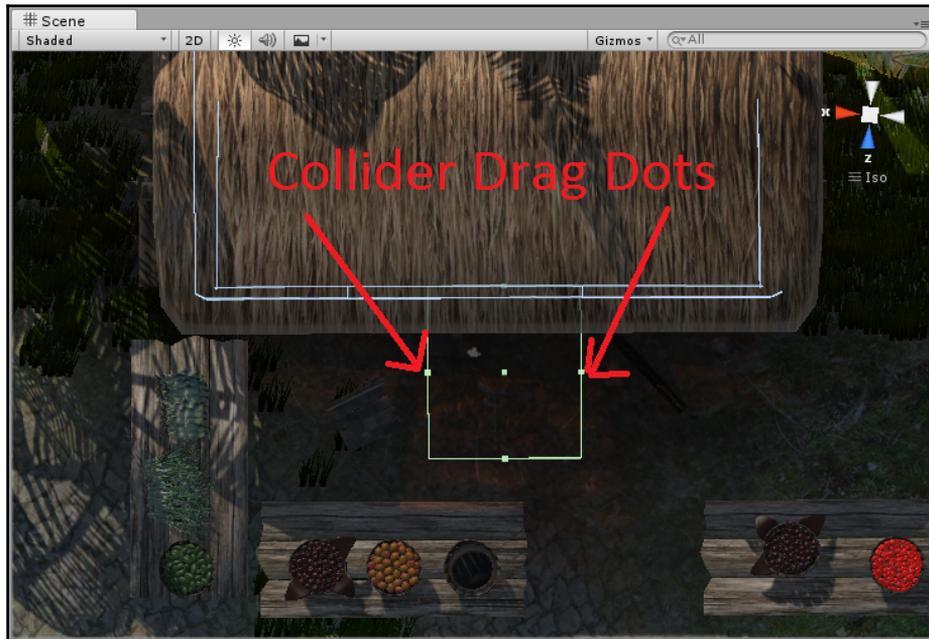
Once the colliders touch, you should hear the audio clip for the door opening, no animation just yet however, as we have not called it in our script. Remember that if you want to check on what is occurring in the game, you should watch the **Scene** view as you test your game with the **Game** view.

Extending colliders

We can make this interaction occur sooner by extending the collider; let's try this out now. Press the **Play** button at the top of the interface again to stop testing.

Select the door child object of the `hut_model` and focus your view on it by hovering over the **Scene** view and pressing *F*. Switch to top view by clicking the **Y** (green) handle of the view Gizmo in the top-right of the **Scene** view.

Now, press the **Edit Collider** button in the **Inspector**. As you can see, new green dot handles appeared in the middle of the cube faces. Now, drag the green collider boundary dots on the front and back of the door in order to extend the collider, as shown in the following screenshot:



Now that our collider is extended, we will collide with it as we approach the steps. Play the test once more and confirm that this works as expected, remembering to press the **Play** button again to stop testing before you continue working. Now, let's get that door open, it's animation time!

Playing door animations through Animator

To perform the door animations by selecting the asset in the **Project** view, we specified in the **Inspector** that it will feature two clips:

- dooropen
- doorshut

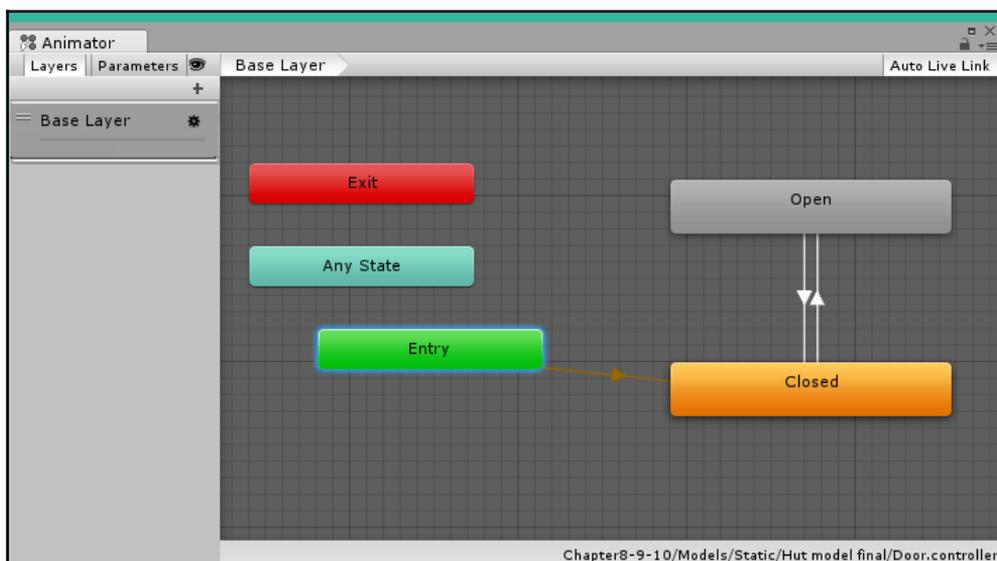
In our `OpenDoor()` function, we'll call upon a named clip using a *string* of text to refer to it. However, first, we'll need to state which object in our scene contains the animation we want to play. As the script we are writing is attached to the player, we must refer to another object before referring to the Animation component. We can do this yet again by referring to the object hit by our player character. In our `OpenDoor()` function, we have an argument that receives the door object from the `OnCollisionEnter()` function. We just used this to play a sound file on the door by saying `door.audio.PlayOneShot("doorOpenSound");` unfortunately, our Animation component is not attached to the door itself, but to the parent object of the `hut_model` object.

To address the parent of an object, we can simply use the `transform.parent` shortcut when referring to the door. Inside the `OpenDoor()` function, beneath the last command you added, `door.audio.PlayOneShot(doorOpenSound);`, insert the following line:

```
door.transform.parent.GetComponent<Animator>().SetBool("dooropen");
```

This line simply says find the door's parent object, address the Animator component on this object, and play the animation clip named `dooropen` through the `dooropen` state. So, let's try this out, save the file and switch back to Unity Editor.

The Animator window shows the door states. After entering (Entry) the state machine, it will automatically follow the Closed state, which can go to the Open state and back to the Closed state:



Press **Play** at the top of the interface, and play test your game. Walking up to the collider on the door should cause the sound and animation of the door to play. As always, stop play testing before you continue your development.

Reversing the procedure

Now that we have created a set of instructions that will open the door, how will we close it once it is open? To aid playability, we will not force the player to actively close the door themselves, but establish some code that will cause it to shut after a defined time period. This is where our `doorTimer` variable comes into play. We will begin counting by adding a value of time to this variable as soon as the door is opened, and then check when this variable has reached a particular value using an `if` statement.

As we will be dealing with time, we need to utilize a function that is constantly updated, such as the `Update()` function we had when we created the script earlier. Create some empty lines inside the `Update()` function by moving its closing curly bracket, `}`, a few lines down. We want to increment the timer only if the door has been opened. Write in the following `if` statement to increment the timer if the `doorIsOpen` variable is set to `true`:

```
if(doorIsOpen){
    doorTimer += Time.deltaTime;
}
```

Here, we check whether the door is open. If the `doorIsOpen` variable is `true`, then we add the value of `Time.deltaTime` to the `doorTimer` variable. Bear in mind that simply writing the variable name like we have done in our `if` statement's condition is the same as writing `if(doorIsOpen == true)`.



`Time.deltaTime` is a Time Unity API property that will return the time passed in milliseconds from the last frame independently from the game's current frame rate. This is important because your game may be running on varying hardware when deployed, and it would be odd if time slowed down on slower devices and was faster when better hardware ran it. As a result, when adding time, we can use `Time.deltaTime` to calculate the time taken to complete the last frame, and with this information, we can automatically correct real-time counting.

Next, we need to check whether our timer variable, `doorTimer`, has reached a certain value, which means that a certain amount of time has passed. We will do this by nesting an `if` statement inside the one we just added, this will mean that the `if` statement we are about to add will only be checked if the `doorIsOpen` `if` condition is valid.

Add the following code after the time-incrementing line *inside* the existing `if` statement:

```
if (doorTimer > doorOpenTime) {
    ShutDoor();
    doorTimer = 0.0f;
}
```

This addition to our code will be constantly checked as soon as the `doorIsOpen` variable becomes `true` and will wait until the value of `doorTimer` exceeds the value of the `doorOpenTime` variable, which, because we are using `Time.deltaTime` as an incremental value, means that three real-time seconds have to pass. This is, of course, unless you change the value of this variable from its default value of 3 in the **Inspector**.

Once `doorTimer` has exceeded a value of 3, the `ShutDoor()` function is called, and the `doorTimer` variable is reset to zero so that it can be used again the next time the door is triggered. If the timer is not reset, then the `doorTimer` will get stuck above a value of 3; as a result, as soon as the door opens, it will close.

Now, you should note that the `ShutDoor()` function has no argument, this is because we need to establish a variable to store a reference to the current door we are interacting with. We did not need to do this earlier because the call to the `OpenDoor()` function was within the collision detection function in which we had a reference to the door in the form of the `hit.gameObject` variable.

However, as we are writing code to call a `ShutDoor()` function within `Update()` and the `hit` variable does not exist there, we should establish a private variable at the top of the script that any function can access, and that is set by our collision detection. Beneath the other variables at the top of your script, add the following variable:

```
GameObject currentDoor;
```

Now, within your collision detection function, add in a line that assigns this variable a value, and then amend the call to `OpenDoor()`, as follows:

```
OnCollisionEnter(Collision hit) {
    if (hit.gameObject.tag == "playerDoor" && doorIsOpen == false) {
        currentDoor = hit.gameObject;
        OpenDoor(currentDoor);
    }
}
```

Now that we have this established variable, place `currentDoor` as the argument of your call to `ShutDoor()` in the `Update()` function:

```
ShutDoor(currentDoor);
```

Your completed `Update()` function should now look like this:

```
void Update () {
    if(doorIsOpen){
        doorTimer += Time.deltaTime;
        if(doorTimer > doorOpenTime){
            ShutDoor(currentDoor);
            doorTimer = 0.0f;
        }
    }
}
```

Now, add the `ShutDoor()` function itself after the existing `OpenDoor()` function, place your cursor at its ending `}` and move down to the next line. As it largely performs the same function as `OpenDoor()`, we will not discuss it in depth. Simply observe that a different animation is called on the outpost and that our `doorIsOpen` variable gets reset to `false` so that the entire procedure may start over:

```
void ShutDoor(GameObject door){
    doorIsOpen = false;
    door.audio.PlayOneShot(doorShutSound);
    door.transform.parent.animation.Play("doorshut");
}
```

It's testing time again! Go to **File | Save** in MonoDevelop, return to Unity, and test your game as before. Now that the timer is established, once your player character has collided with the door, three seconds should pass before it is automatically closed again.

Code maintainability

Now that we have a script in charge of opening and closing our door, let's look at how we can expand our knowledge of custom functions to make our scripting more maintainable.

Currently, we have two functions we refer to as custom or bespoke: `OpenDoor()` and `ShutDoor()`. These functions perform the same three tasks: playing a sound, setting a Boolean variable, and playing an animation. So, why not create a single function and add arguments to allow it to play different sounds and have it choose either `true` or `false` for the Boolean and play different animations? Making these three tasks into arguments of the function will allow us to do just that.

After the closing curly bracket of `ShutDoor()` in your script, add the following function:

```
void Door(AudioClip aClip, bool openCheck, string animName,
  GameObject thisDoor){
  audio.PlayOneShot(aClip);
  doorIsOpen = openCheck;
  thisDoor.transform.parent.animation.Play(animName);
}
```

You'll note that this function looks similar to our existing `OpenDoor` and `ShutDoor` functions, but has four arguments in its declaration: `aClip`, `openCheck`, `animName`, and `thisDoor`. Effectively, these are variables that get assigned when the function is called, and the values assigned to them are used inside the function. For example, when we want to pass values for opening the door to this function, we will call the function and set each parameter by writing the following line:

```
Door(doorOpenSound, true, "dooropen", currentDoor);
```

This feeds the `doorOpenSound` variable to the `aClip` argument, a value of `true` to the `openCheck` argument, a string of text(`dooropen`), to the `animName` argument, and sends the `GameObject` assigned to the `currentDoor` variable to the `thisDoor` argument.

Now we can replace the call to the `OpenDoor()` function inside the collision detection function. First, remove the following line that calls the `OpenDoor()` function inside the `OnCollisionEnter()` function:

```
OpenDoor(currentDoor);
```

Replace it with the following line:

```
Door(doorOpenSound, true, "dooropen", currentDoor);
```

Now, we have a single function that is called with the following four arguments being sent:

- `doorOpenSound` as the sound to play
- `true` as the value for `doorIsOpen`
- `dooropen` as the animation to play
- `currentDoor` as the object we're currently interacting with

Finally, as we are using this new method of opening and closing the doors, we'll need to amend the door closing code within the `Update()` function. Within the `if` statement that checks for the `doorTimer` variable exceeding the value of the `doorOpenTime` variable, replace the call to the `ShutDoor(currentDoor)` function with this line:

```
Door(doorShutSound, false, "doorshut", currentDoor);
```

You may now delete the two original methods, `OpenDoor()` and `ShutDoor()`, as our customizable `Door()` function now supersedes both of them. By creating functions in this way, we are not repeating ourselves in scripting, and this makes our script shorter, simpler to read and debug, and saves time from writing two functions.

If you would like to keep these functions to remind you of what you have done, you may comment them out. When turned into comments, they are no longer executable parts of the script, so simply place `/*` before the opening of your `OpenDoor()` function and `*/` after the closing curly bracket of the `ShutDoor()` function. In your script editor, the code will change color to show that it has been made into a multiline comment. Ensure that you save your code now.

Switch back to Unity now and press Play to test your game; you should now see that your door opens and closes, but we still have the issue of bumping into our extended collider.

Drawbacks of collision detection

Our first implementation of the door opening is complete. Making use of `OnCollisionEnter()` works perfectly but is still not the most efficient method of creating the door opening mechanism.

The main drawbacks here are as follows:

- Code is stored on the player object and means that as we create further interaction code, this script becomes longer and difficult to maintain.
- The Collider extension means that the player bumps into an invisible surface that will open the door but cause the player to stop in their tracks, which interrupts gameplay.

We will now move on to try our second of the three approaches—using ray casting.

Approach 2 – ray casting

In this section, we will implement our second approach to opening the door. Although character controller collision detection may be a valid approach, by introducing the concept of ray casting, we can try an approach where our player only opens the door if they are facing it because the ray will always face the direction that the player controller is facing and that this direction is not intersecting the door if, for example, the player backs up to it.

Disabling collision detection with comments

To avoid the need to write an additional script, we will simply comment out, that is, temporarily deactivate the part of the code that contains our collision detection function. To do this, we will add characters to turn our working collision code into a comment. Ensure that you still have the `PlayerCollisions` script open in the script editor. Go to the following:

```
OnCollisionEnter(Collision hit){
```

Place the following characters before the preceding `OnCollisionEnter` method starting line:

```
/*
```



Remember that putting a forward slash and asterisk into your script begins a multiline comment (as opposed to two forward slashes that simply comment out a single line).

After the collision detection function's closing right curly bracket (`}`), place the reverse of this, that is, an asterisk followed by a forward slash, `*/`. Your entire function should have changed the syntax color in the script editor to show that it has been *commented out*. At this point, Visual Studio and MonoDevelop will offer the ability to fold the commented function, leaving the first line visible for better code readability.

Refactoring the code – writing a Door Manager class

Now that we are about to use a ray to open the door, we should reconsider where the code for opening the door is placed. Our current door opening code is located on the player, and as the player may encounter many objects within our game, we should consider that the logic for the door opening and closing would be better stored on the door itself, meaning that the door need only be aware of its own functions and the requirement to open for the player. We can then make use of the ray cast in this approach and the trigger in our third approach to call upon the opening code on the door object.

In Unity, click the **Create** button in the **Project** view, and choose the relevant language you are working with. Name your new script `DoorManager`, and then launch it in `MonoDevelop`.

Here, we will migrate the majority of our door logic from the `PlayerCollisions` script.

As we have already covered what this code does, simply place all the code into your new `DoorManager` class:

```
using UnityEngine;
using System.Collections;

public class DoorManager : MonoBehaviour {
    bool doorIsOpen = false;
    float doorTimer = 0.0f;
    public float doorOpenTime = 3.0f;
    public AudioClip doorOpenSound;
    public AudioClip doorShutSound;
    void Start(){
        doorTimer = 0.0f;
    }
    void Update(){
        if(doorIsOpen){
            doorTimer += Time.deltaTime;
            if(doorTimer > doorOpenTime){
                Door(doorShutSound, false, "doorshut");
                doorTimer = 0.0f;
            }
        }
    }
    void DoorCheck(){
        if(!doorIsOpen){
            Door(doorOpenSound, true, "dooropen");
        }
    }
    void Door(AudioClip aClip, bool openCheck, string animName){
```

```
        audio.PlayOneShot(aClip);
        doorIsOpen = openCheck;
        transform.parent.gameObject.animation.Play(animName);
    }
}
```

The key change from the code we had in `PlayerCollisions` is the addition of the `DoorCheck()` function. By adding this, we now have a function that can be called easily, without the need for additional arguments that check whether the door is not currently open:

```
if(!doorIsOpen){
    Door(doorOpenSound, true, "dooropen");
}
```

It then goes on to call the `Door()` function with its opening arguments.

This switches on the `Update()` function's timer, which will reset the door as before. The following image is the `Door Manager` component setup shown in the **Inspector**:



Now we can remove the duplicate code from `PlayerCollisions` and implement our **Raycast**.

Tidying PlayerCollisions

Now that we have our `DoorManager` script handling the door's state, we can cut the `PlayerCollisions` script code down to very simple elements. Given that you are learning Unity, as stated before, you may want to comment out code done previously instead of deleting it, but in the following example, only the remaining code is shown instead of showing you what should be commented also, so it is your choice whether to delete the code or comment it.

Regardless, you should remove (delete or comment-out) the code from `PlayerCollisions` so that you are only left with the following piece of code inside the `PlayerCollisions` class:

```
using UnityEngine;
using System.Collections;

public class PlayerCollisions : MonoBehaviour {
    GameObject currentDoor;
    void Update () {
    }
}
```

Now we have only the `Update()` function. This is where we will cast our ray and use a private variable to hold a reference to the door that we're currently interacting with.

Let's add the code that will perform the `Raycast` and call the `DoorCheck()` function with a `SendMessage` method in our `DoorManager`.

Casting the ray

In the `PlayerCollisions` script, move your cursor a couple of lines down from the opening of the `Update()` function. We will place the `Raycast` operation inside the `Update()` function because we need to technically cast our ray forward every frame, as the player's direction may change at any time. For optimization, you might want to restrict this call to 1 time on 10 loops or something like this, to avoid CPU overhead but I will leave this code tweak to you as an exercise. Now, add the following code within the `Update` method:

```
RaycastHit hit;

if(Physics.Raycast (transform.position, transform.forward, out hit, 3))
{
    if(hit.collider.gameObject.tag=="playerDoor")
    {
        currentDoor = hit.collider.gameObject;
        currentDoor.SendMessage ("DoorCheck");
    }
}
```

At the outset, a ray is created by establishing a local variable called `hit`, which is of the `RaycastHit` type. Note that it does not need to be made private in order to not be seen in the Inspector, it is not seen because it is a local variable (declared inside a function). This will be used to store information on the ray when it intersects colliders. Whenever we refer to the ray, we use this variable.

Then, we use two `if` statements. The parent `if` is in charge of casting the ray and uses the variable we created. As we place the casting of the ray (the `Physics.Raycast()` function) into an `if` statement, we are able to only call the nested `if` statement if the ray hits an object, making the script more efficient.

Our first `if` statement contains `Physics.Raycast()`, the actual function that casts the ray. This function has four arguments within its own brackets:

- The position from which to create the ray (`transform.position`—the position of the object that this script applies to, the **First Person Controller**)
- The direction of the ray (`transform.forward`—the forward direction of the object that this script applies to)
- The `RaycastHit` data structure we set up called `hit`—the ray stored as a variable
- The length of the ray (3—a distance in the game units, meters)

Note that in C#, we must use a precursor `out` parameter before the `hit` variable in order to get the function to assign data to it; this is done implicitly in JavaScript by simply naming the variable to use.

Then, we have a nested `if` statement that first checks the `hit` variable for collision with colliders in the game world, specifically whether we have hit a collider belonging to a `GameObject` tagged `playerDoor`, we will write:

```
hit.collider.gameObject.tag == "PlayerDoor"
```

Once both the `if` statements' conditions are met, we simply set the `currentDoor` variable to the object stored in the collision `hit` collection and then call the `DoorCheck()` function using the `SendMessage()` method.

Using `SendMessage`, we can call a method on a `GameObject` without a reference to the particular script, simply by naming the function:

```
currentDoor.SendMessage("DoorCheck");
```

`SendMessage()` here simply checks any scripts attached to the object assigned to the `currentDoor` variable, finds the `DoorCheck()` function, and calls it.



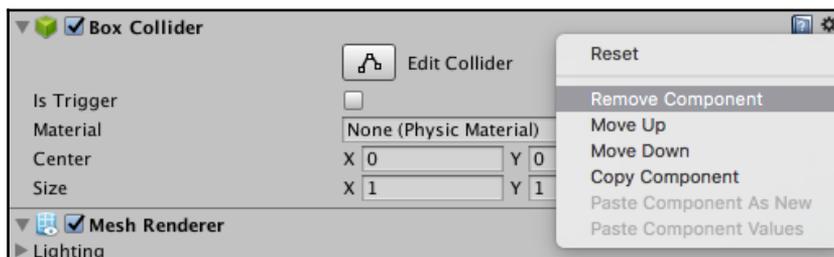
For more information on `SendMessage()`, see the Unity script reference at <http://unity3d.com/support/documentation/ScriptReference/GameObject.SendMessage.html>.

Save your script and return to Unity now. All that is left is to apply our `DoorManager` to the door. Ensure that the **outPost** parent object is expanded in the **Hierarchy** so that you can see the door child object and then drag and drop your `DoorManager` script from the **Project** view onto the door child object in the **Hierarchy**.

Assign the `door_open` and `door_shut` audio clips to the public variables on the script component in the **Inspector**, as you did previously when they were members of the `PlayerCollisions` component.

Resetting the collider

Now that we are using ray casting to look ahead of the player, we no longer need our expanded collider on the door, it can be reset to its default dimensions, those of the mesh that it is applied to. To do this, simply select the **door** child object in the Hierarchy and then in the Box Collider component, click the Cog icon to the right-hand side, and choose **Reset** from the drop-down menu that appears:



Now, let's see it in action! Play your test by pressing the Play button at the top of the Unity interface and you will note that, when approaching the door, it not only opens before you bump into it, but only opens when you are facing it, as the ray that detects the door is cast in the same direction that the player character faces.

Ray casting can be used in this way to detect colliders near other objects or in their line of sight in many other game mechanics, as we will see in the next chapter.

It is recommended that when you finish this book, you try prototyping some ideas that use techniques like this because of how valuable they are in practical usage. However, as stated previously, this is not the expected behavior for a door, as we expect it to simply detect a person within its vicinity. Consider the behavior of a security light motion detector or a burglar alarm sensor as an ideal example. For this reason, let's move on to approach number 3, the most efficient approach to opening our door, using a trigger collider.

Approach 3 – trigger collision detection

As our first two approaches had drawbacks—bumping into a collider with standard `OnCollisionEnter()` and only opening the door when facing it with `Raycast`—using a trigger collider should be considered the best approach to create a door opening mechanism because there is no physical collider to bump into, and it will be triggered regardless of the direction the player is facing.

As previously stated, we will leave our existing **Box Collider** on the door so that it has a physical presence that objects can collide with, if necessary. This means that we should place a **Box Collider** set to trigger mode on a separate object—this is also important because our door is animated to move and we do not want our trigger area to move when the door is opened. For this reason, we can attach our **Box Collider** to the parent object, `hut_model`, and then simply position this collider as a large trigger area around the door, as shown in the illustrations at the beginning of this chapter. Once this is positioned, we can apply a new short script that uses the `OnTriggerEnter()` function, simply a function that detects other colliders entering a trigger mode collider. We will use this function to call upon the same `DoorCheck()` function as the ray in our second approach.

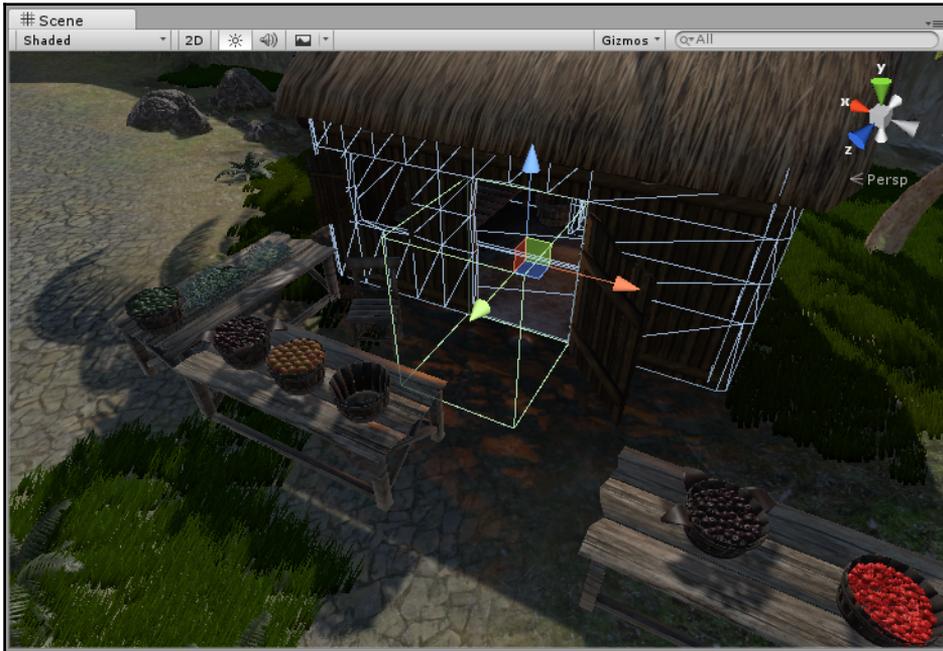
This will make our `PlayerCollisions` script obsolete for now, which means that the logic for our door is entirely self-contained; this is considered a good practice in development.

Creating and scaling the trigger zone

Select the parent object named `hut_` in the **Hierarchy** and choose **Component | Physics | Box Collider** from the top menu. Your **Box Collider** will be created at the standard (1,1,1) Cube primitive scale, but we can use the collider editing feature and *drag* in the **Scene** view in order to resize and reposition it.

You will also note that it has been created with the center of the `hut_model` as its base; this is simply where the axes of this model were in the package it was created.

Ensure that you have the Hand Tool (you can quickly select it by pressing the `Q` key) selected as this hides the 3 axis handles for the object, making it easier to adjust the collider boundaries. Hold the `Shift` key and drag the boundary dots on the sides of the Box Collider in the **Scene** view until you have something that looks like the next screenshot:



It will help if you use the orthographic (top and side) views to achieve this.

Now, we will ensure that this is not an ordinary collider but a trigger! In the **Inspector** view for this object, check the **Is Trigger** checkbox on the Box Collider component.

Let's see now how to write code for managing triggered events.

Scripting for trigger collisions

Now that we have a trigger collider in place, we simply need to detect our player entering the trigger area (also sometimes referred to as a *trigger zone*). In the **Project** view, click the **Create** button and create a new script in your chosen language. Rename your new script as `TriggerZone` and launch it in `MonoDevelop`. Our trigger code will be applied to the parent `_hut` object and is very simple as all we need to do is the following:

- Detect the player
- Find the child object of the `hut_model` named `door`
- Send a message to the `door` to call the `DoorCheck()` function

Let's begin by establishing an `OnTriggerEnter()` function in our script:

```
void OnTriggerEnter(Collider col){  
}
```

This will look familiar as it works in a similar way to the `OnCollisionEnter()` function we wrote earlier, but the information stored by the collision event is the particular `Collider` that the trigger collider has collided with or *intersected*. This is stored in a variable we have called `col` for simplicity.

From this, we can check the object as earlier, using an `if` statement. Simply add the following code to your `OnTriggerEnter()` function:

```
if(col.gameObject.tag == "Player"){  
    transform.Find("door").SendMessage("DoorCheck");  
}
```

Here, we check the `col` argument as to whether the collider belongs to a `GameObject` with a `Player` tag. This tag was already present on the First Person Controller when we imported it as a prefab when we began working. Equally, we could use the hierarchical name:

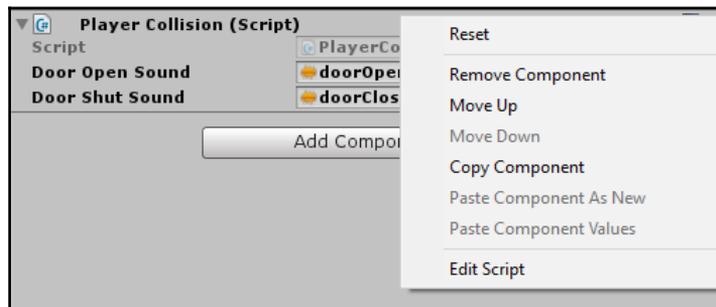
```
if(col.gameObject.name == "Warrior_RM"){
```

However, making use of the tag is quicker and will remain consistent should we rename our First Person Controller at any point.

When the `if` condition is met, we locate the `door` child object using the `transform.Find()` command, which is more efficient than `gameObject.Find()` as it only searches the children of the object this script is attached to, whereas `gameObject.Find()` searches the entire **Hierarchy**. This is also useful as there may be other buildings in our scene with a door and we want to ensure that we are only sending the message to the door on the building whose trigger the player currently stands within. Finally, `SendMessage()` is used, as before, to call the `DoorCheck()` function in the `DoorManager` script attached to the door. Save your script in `MonoDevelop/Visual Studio` and switch back to Unity now. In Unity, attach your new `TriggerZone` script to the `hut_model` object in the **Hierarchy** by dragging it from the **Project** view and dropping it onto `hut_model` in the **Hierarchy**.

Removing the PlayerCollisions component

Your new trigger is ready to use, but we should remove our previous approach in the `PlayerCollisions` script. You may keep the current state of `PlayerCollisions` in your Project for future reference, but currently, we no longer need it to be applied to the player. Select `Warrior_RM` in the **Hierarchy** and locate the **Player Collisions (script)** component. To remove the component, click the Cog icon to the right-hand side of the component and choose **Remove** from the pop-out menu that appears:



Okay, let's play! Choose **File | Save Scene** from the top menu and then click the **Play** button to test your new trigger zone. Now the hut door will open whenever the player enters the trigger zone, regardless of the direction they are facing, the ideal way to have the door open in our game. In development terms, we will call this an ideal approach because it does not rely on the player checking for an object that is interacting with; the hut simply knows that if the player approaches, it must open its door.

Placing additional models

We will now add the prison and the village buildings models in two specific spots of the island that may be adjusted a bit with the Terrain Editor toolset before or after we place the buildings and are happy with the results. Look for the prison model prefab and drag it into the scene. Repeat these steps for the other static models: House, Hut, Store, Windmill, Water Well, and Chariot. Now have fun importing, setting, placing, and scaling the models at an appropriate scale:



A top view of the village part of the island with some house models



In a perfect world, and with high budgets, your artists should have modeled everything already perfect in scale. Playing with the Scale values in the Transform component of the imported models in Unity Editor is usually a bad idea, but is very handy for prototyping.

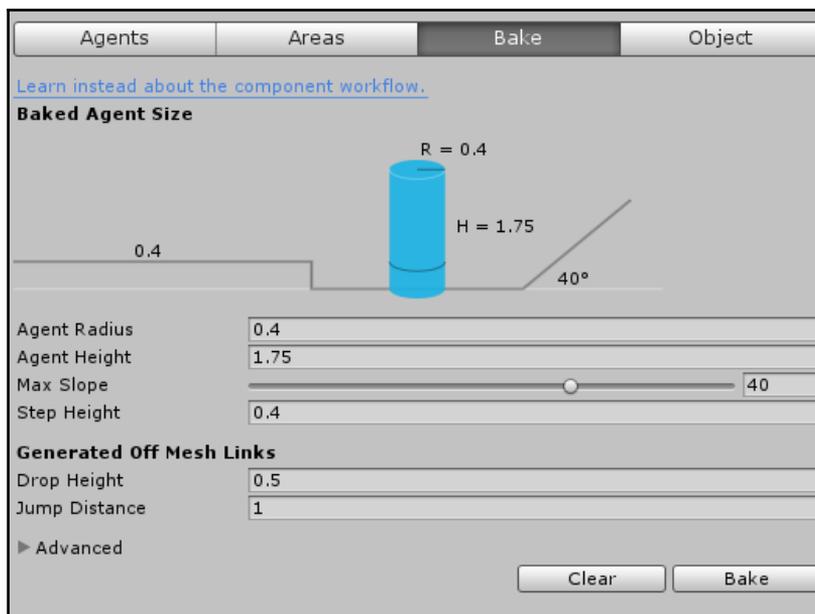
Unity Navigation System

To be able to use the enemies AI, which is based on the NavAgent component that implements pathfinding and basic AI, your map must be ready for it, and should include a baked NavMesh.

Unity Navigation System implements quick pathfinding on complex levels by preparing, at build-time, a navigation mesh (NavMesh) calculated taking into consideration Static and Navigation-Static GameObjects to create a huge mesh divided into smaller patches that define where the AI can walk when they should jump to reach a place, and so forth.

The Bake tab

The following image displays the **Bake** tab of the **Navigation** window, showing the rules for baking the NavMesh:

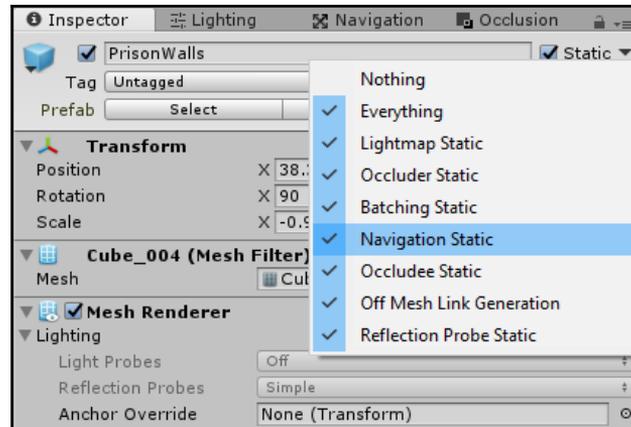


To quickly bake the NavMesh, open the Navigation window from **main menu** | **Windows** | **Navigation**. Then, in the **Bake** tab, hit the **Bake** button. The **Clear** button is meant to clear any previously baked data.



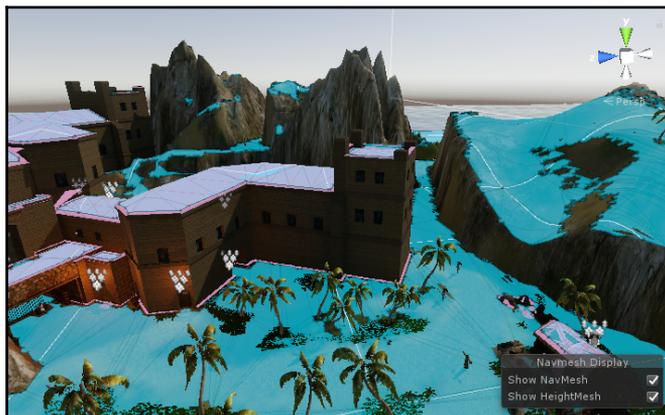
There are some settings that are important to know before proceeding with the baking, which are mostly **Agent Radius** and **Agent Height**. Set the values like in the preceding picture for now, and refer to <https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html> for further reading about baking the NavMesh.

All the models that you have imported, which should obstruct the navigation of the Agents, should be set as **Static** from the top menu of the GameObject:



You can also choose what kind of static this object is, for example, it could be static for lightmapping but not for Navigation System (for example, a soft bush you can pass through) or other combinations that your game design requires.

After the baking process is completed, you should see the result in the **Scene** view with the Navmesh display in overlay with the **Show NavMesh** and **Show HeightMesh** options, like in the following picture:



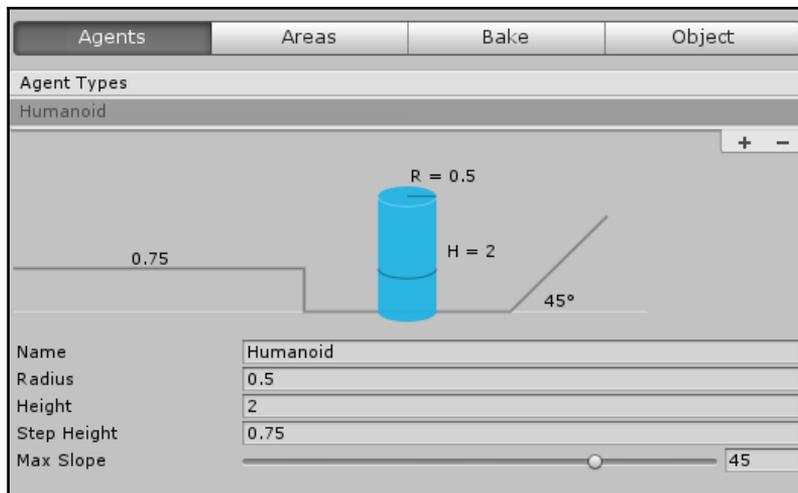
The scene view shows the NavMesh in blue when you have the Navigation Window open or, if it is tabbed like in the picture above, when it is focused

As you can see, the blue area is the walkable part, while the pink areas are the HeightMesh, which is optionally calculated to achieve a better placement of the AI characters. Navigation Static GameObjects carve holes in the NavMesh, to block the AI from using the path in that way.

For more info about the inner workings of Unity Navigation System, head to <https://docs.unity3d.com/Manual/nav-InnerWorkings.html>.

The Agents tab

The **Agents** tab section allows you to set up a series of NavAgent settings for different type of humanoids, creatures, or vehicles that might move and take different settings. In this way you will be able to use different groups of NavAgent with different base settings:



The Agents tab with the default Humanoid Agent settings shown in the Inspector

The Object tab

On the **Object** tab of the **Navigation** window, you can define:

- When a static object, such as a house or building, will be **Navigation Static**
- If this object should generate `OffMeshLinks` if needed

- The **Navigation Area** type this object is up to:



The Object tab showing the prisonWalls GameObject navigation settings



The **Object** will always show details of the object you have selected.

The Areas tab

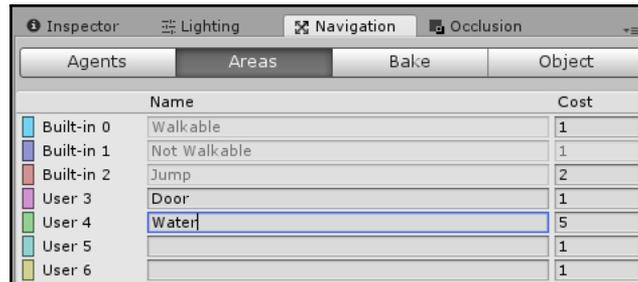
In the **Areas** tab, you can define additional areas (**Walkable**, **Non Walkable**, and **Jump** are built-in, which means you can't edit or remove them because they are meant for basic navigation tagging) for your very specific level-design purposes.

From the official Unity manual:

The Navigation Areas define how difficult it is to walk across a specific area, the lower cost areas will be preferred during path finding. In addition each NavMesh Agent has an Area Mask which can be used to specify on which areas the agent can move.

Area Types and Navigation cost

The following screenshot displays the **Navigation** window's **Areas** tab docked with other tools together with the **Inspector**:



In the preceding screenshot, we added two custom area types that can be used for two common use cases:

- **Water** area: It is made more costly to walk by assigning it a higher cost, to deal with a scenario where walking in shallow water is slower than dry land.
- **Door** area: It is made accessible for specific characters, to create a scenario where, for example, the player and NPC can walk through specific doors, while enemy guards cannot.

From the Unity Manual:

The pathfinding cost allows you to control which areas the pathfinder favors when finding a path. For example, if you set the cost of an area to 3.0, traveling across that area is considered to be three times longer than alternative routes.

The cost per area type can be set globally in the Areas tab, or you can override them per agent using a script.

The area types are specified in the Navigation Window's Areas tab. There are 29 custom types, and 3 built-in types: Walkable, Not Walkable, and Jump.

- **Walkable** is a generic area type which specifies that the area can be walked on.
- **Not Walkable** is a generic area type which prevents navigation. It is useful for cases where you want to mark a certain object to be an obstacle, but without getting NavMesh on top of it.
- **Jump** is an area type that is assigned to all auto-generated Off-Mesh Links.

If several objects of different area types are overlapping, the resulting navmesh area type will generally be the one with the highest index. There is one exception however: Not Walkable always takes precedence. Which can be helpful if you need to block out an area.

Area mask

Each agent has an *Area Mask* that defines which areas it can use when navigating.

The area mask is useful when you want only certain types of characters to be able to walk through an area. For example, in our evasion game, you could mark the area under the old man's hut door with a *Door* area type, and uncheck the Door area from the guards character's Area Mask to avoid them walking in that specific building.

For additional info about Areas and navigation costs, head to <https://docs.unity3d.com/Manual/nav-AreasAndCosts.html>.

In this last section, we briefly looked at how to prepare the level for the AI to walk around. For a deeper look at the Navigation System and how NPC and AI will use it through the NavAgent component, just dive into the next chapter.

Summary

In this chapter, we explored three key methods to detect interactions between objects in 3D games: Collision detection, Ray casting and Trigger collision detection. These three techniques have many individual uses, and they are key skills that you should expect to reuse in your future use of Unity. We have also learned how to use the basics of the Navigation System and how to set up the level to optimize AI pathfinding with Unity's built-in Navigation System and its most common features and settings.

In the next chapter, we will explore how to implement a simple **non-playing character (NPC)** and a simple enemy AI through coding and through the Unity Navigation System. We will take advantage of the work done with the `ThirdPersonCharacter` player to drive the AI with the `AICharacter` class. We will learn how to make the NPC interact with the Player character and how to build a simple dialogue system displayed through the Unity UI Canvas and design an Advanced AI class for the enemy guards.

8

AI, NPC, and Further Scripting

In this chapter, we'll expand our 3D game with a **non-playing character (NPC)** and with enemy AI characters that will be hunting the player to arrest him again! We will take a deeper look at the Unity Navigation System, a built-in path finding system based on a generation (baking) of a **Navigation Mesh (NavMesh)**, an optimized low poly invisible mesh needed to make AI path finding fast and performing. This data will be stored, together with lighting data, in a sub folder with the same name as the relative scene. This system facilitates the calculation of a *walkable* mesh over the existing terrain and geometry. Baking a NavMesh will enable **Navigation Mesh Agent (Nav Mesh Agents)** components on AI characters, to walk a path and find the way to a given spot in the level in a fast and optimized way. This will result in better performance compared to a generic real-time path finding solution, as it is prepared at editing time, and in general will give us more possibilities for managing AI behaviors at runtime.

In detail, we are going to:

- Create an NPC prefab and its Animator Controller and other required components
- Use Animator's `BlendTrees` to control the old man's speech set of animations
- Use Unity UI, Triggers, and some scripting to make the player interact with the NPC
- Set up the Navigation System for baking the NavMesh

- Create a `SimpleAI` basic class for managing the NPC and the guards AI
- Prepare a quick and dirty AI-test scene and bake its NavMesh
- Create the enemy guard prefab and its Animator Controller
- Discover Nav Mesh Agent settings and properties to achieve a solid base for our AI
- Create an `Advanced AI` class for the enemy guards that uses the Nav Mesh Agent and implements the field of view
- Make the components look better in the Inspector easily thanks to `Property Drawers`

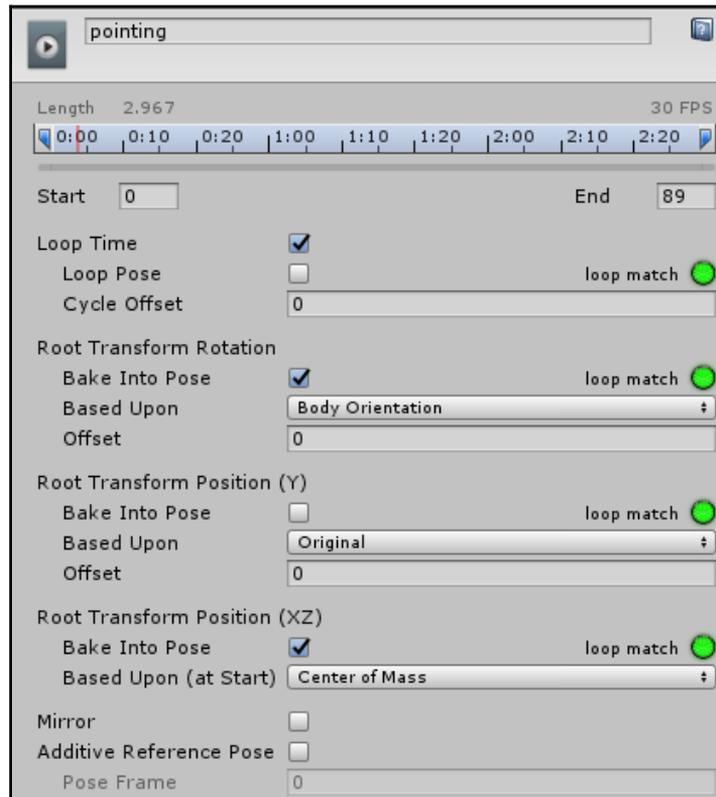
Creating an NPC

Creating a complex non-playing character can be a hard task. You can find the ready-made `oldman` NPC in the code book folder, but we will practice the Editor, by deconstructing the AI character prefab from Unity Standard Assets to build our own, and modify the structure and the code for our needs.

We are going to use a copy of our `Chapter6-start.unity` scene that we edited in the previous chapter, where the terrain is half-done, for simplicity, as this scene doesn't contain any additional buildings or other game elements. Open the scene and save it as `Chapter8-NPC.unity`. In this scene, you will find some of Unity's standard assets, such as the fire particle system that we will explore later, and terrain features such as trees and grass in a specific spot of the island in the scene. We will arrange our actors here, in this corner of the terrain, along with some trees and grass.

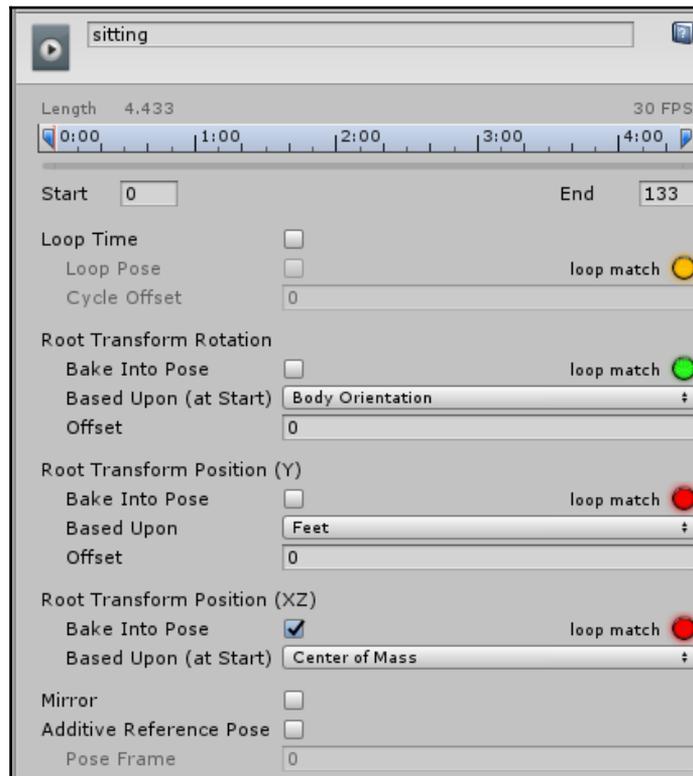
First of all, we want to make sure that the `oldman` rig is correctly imported as a **Humanoid** rig and that the animation clip included in the book's code is imported and set for the correct root motion animation, as we did in the final part of the previous chapter for the player hero.

We should also remember the avatar reference for all of the clips, which should be the avatars of the non-animated rig we did for our hero model back in Chapter 5, *Character Animation with Unity*:



All the animation clips that we need for the NPC will be set like the preceding screenshot, with the exclusion of the `walk_in_circle` and the `sitting` clips, the former will have the **Bake Into Pose** box unchecked in the **Root Transform Position (XZ)** position panel, while the latter will have the **Bake Into Pose** option checked, will be based on the **Center of Mass**, and the animation will not be played in a loop. The same goes for the `standup` animation clip.

They will all need the **Root Transform Position (Y)** set on the feet in the original location and will need the option: **Bake Into Pose** disabled to avoid movement from the GameObject's original position:

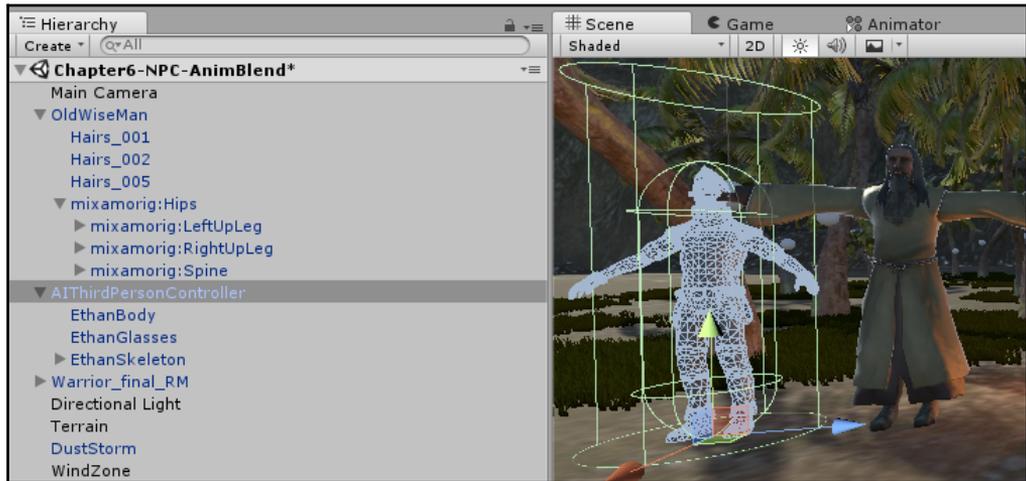


You can check the ready-made settings on this animation clip in the assets folder, be careful when playing with this settings, as a wrong setting may result in unwanted unexpected results.

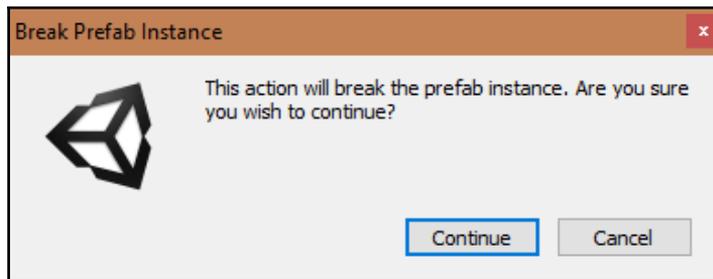
We will create the basic NPC, starting with the `AIThirdPersonController` prefab. You can do this in the **Scene** view, or you can switch to a front orthographic view to better check its body measures. Let's follow these steps:

1. Drag in the scene the `AIThirdPersonController` prefab located in the `Assets\Standard Assets\Characters\ThirdPersonCharacter\Prefabs` folder.

2. Drag our OldMan character model from the folder situated in `Assets\Chapters7-9-10\Models\Characters` and check its measures with the `AIThirdPersonController` prefab measure. Scale it accordingly to have the same proportion as the **Ethan** rig from the Standard Assets Characters package.
3. Then we will drag our OldMan in the **Hierarchy** so that it becomes a child of the `AIThirdPersonController` `GameObject`:



4. We will **DELETE** the `EthanBody`, `EthanGlasses`, and `EthanSkeleton` `GameObjects` from the **Hierarchy**.
5. When prompted by Unity with the following popup, choose **Continue**:



The prompt asking if you are sure you want to lose connection with the stored prefab

The link for the old prefab will break, but this is okay, as we are building a brand new prefab for the old man NPC. The `GameObject` in the **Hierarchy** has now turned to black and is not linked to any saved prefab in the project anymore. So finally:

1. Rename the `GameObject` from `AIThirdPerson` to `OldMan (NPC)`.
2. Check the size twice by looking at the characters from various angles in order to be sure that we have used the correct scale. Using orthographic front, top, or side views to check sizes can help, see the next screenshot.
3. Remove the `AICharacterControl` component to leave some space for our own component.
4. Create a new component and call it `SimpleAI`. This class will be the base to manage the NPC and the enemies AI.

Now we are ready to store the prefab we created. Drag the `GameObject` into a `Prefab` folder in your project, and you should see that the `OldMan (NPC)` `GameObject` in the hierarchy turns from black to blue.

Initial code and Animator Controller

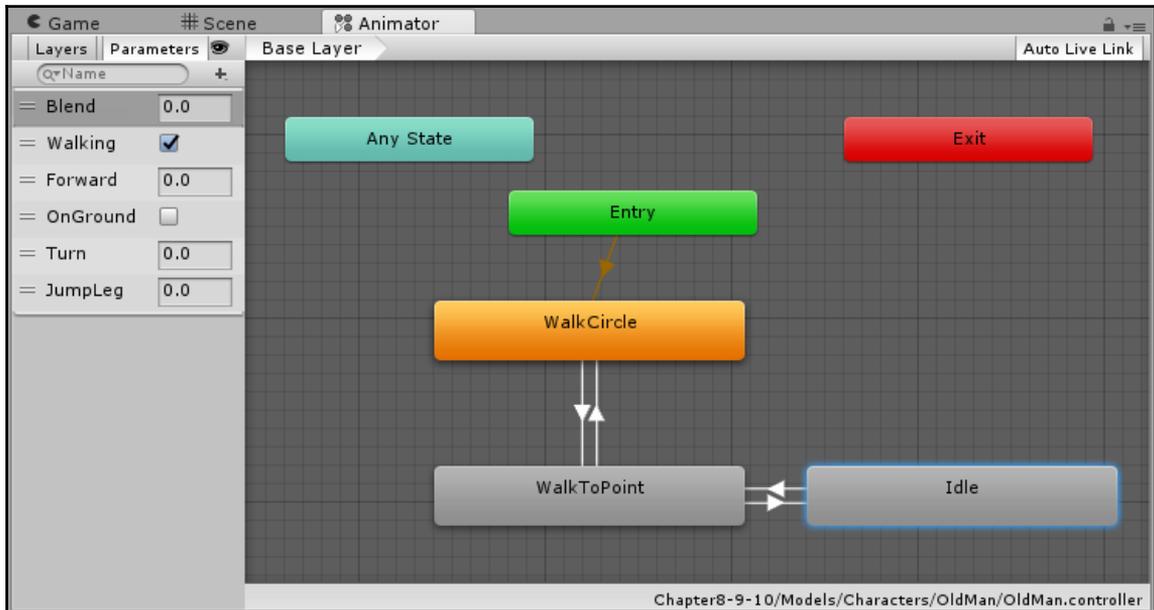
We will take advantage of our knowledge of the Mecanim system to manage the animation states for the NPC:

- Create an Animator Controller and assign it to the Animator component.
- Create two states: `WalkCircle` and `WalkToPoint`.
- Create a Blend Tree from state: `Idle`. We will use the `Idle` Blend Tree to regulate the various talking animations we have available, for when the NPC talks to the player.

The `WalkCircle` state will be a simple Root Animation of a walk loop in a circle, which is actually the Default State in which the NPC will find itself when the game starts. The `WalkToPoint`, which once again shows the power of Mecanim, will be a Standard Assets generic Mecanim walk animation applied to our rig character and used when we want to move the NPC from one spot to another. We will also create the same parameters we did in the standard assets `ThirdPersonAnimatorController` in the previous chapter.

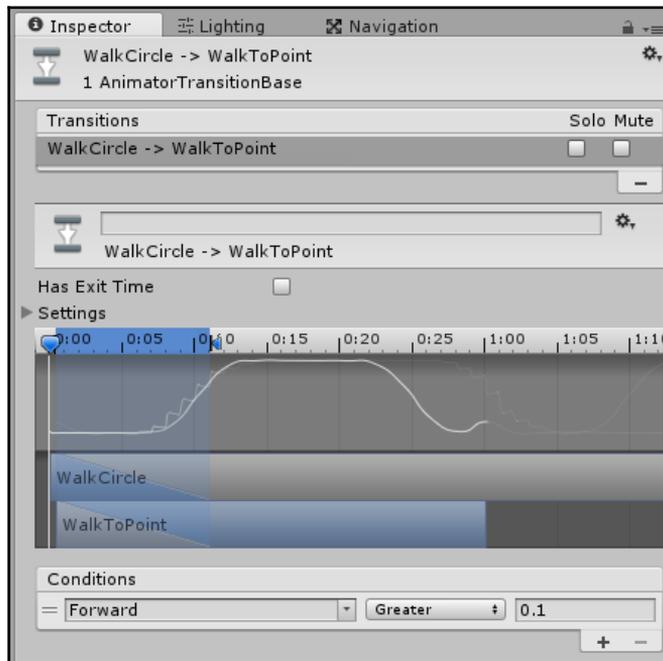
Additionally, we will create a Boolean parameter and call it `Walking`. This flag will indicate that the NPC is walking in a circle, hence, not speaking with the player. When this variable is set to `false`, it will indicate the start of the dialog and will drive our state machine into or out of the `WalkCircle` state, which is the default state.

Look at the following screenshot:



Animator transitions

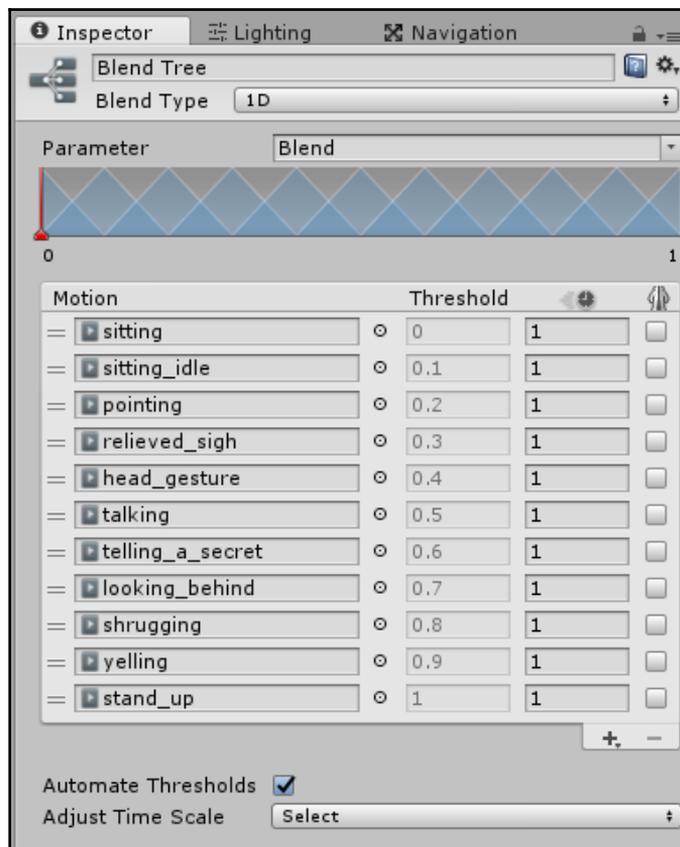
Our mission will be to make the NPC able to walk to a sit spot near the rock where he will sit down. Wait for your first answer before making him enter the third **Blend** animation with a value of 0.3 in the Idle Blend Tree. The **Forward** speed of the character will be the parameter for exiting the `WalkCircle` to `WalkToPoint` state, which is when the code will start to move the NPC toward the sitting spot, as well as from the Idle state to the `WalkToPoint`, when the `OldMan` will stand up and walk back to his meditation spot. We will need to edit the transitions (the arrows that connect the various states in the state-machine) one by one, to slightly tweak certain aspects of them:



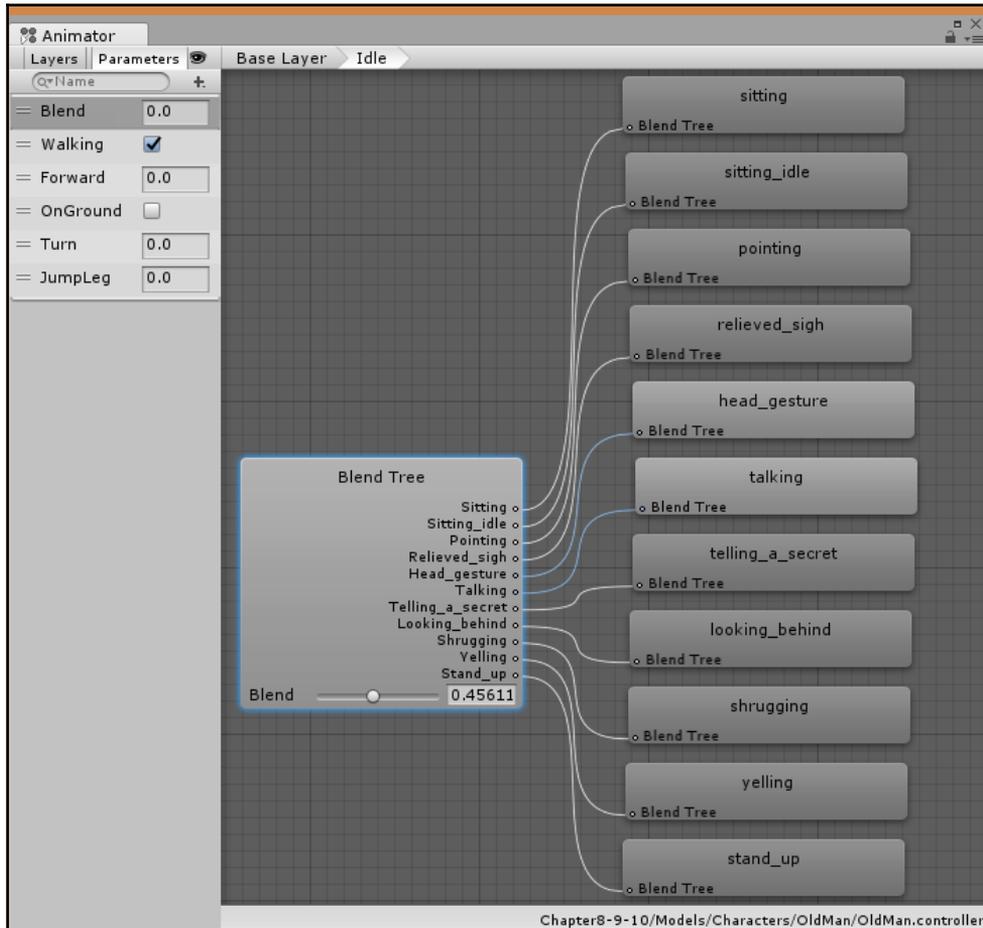
We want to move the starting point of the transition to **0:00** and the final to **0:10** in order to ensure the fastest transition as possible, and we also will uncheck the **Has Exit Time** option for the `WalkCircle -> WalkToPoint` transition, as in the previous screenshot.

We will add a condition with the parameter `Forward`, which must be greater than `0.1`. This will happen when the character starts to move. For the **WalkCircle -> WalkToPoint** transition, we will repeat the same steps, except we will add the condition on the `Walking` parameter if set to `true`. For the **Idle -> WalkToPoint** and backward transitions, we will instead use the `Forward` parameter that will be greater or less than `0.1` in its respective directions.

The Blend Tree, when selected in the **Inspector**, will show the 11 motions (animation clips) contained into it using a **one-dimension (1D)** linear interpolation between the states:



To implement this, we will take advantage of the Unity Navigation System and script some special methods to trigger the two states correctly in the `SimpleAI` class. The `SimpleAI` class we are going to write is a modification of the `AICharacterControl` class, and replicates the `Move()` and `UpdateAnimator()` methods. The Animator states' transitions will be described after the class code has been input, because these transitions need additional code in order to set the `Walking` parameter and Navigation Agent destination and vectors accordingly:



The Animator window showing NPC Idle Blend Tree, sit down, stand up, and the many kind of talking animation clips

We will need to manually rotate the character to orientate him in the correct direction, and we will do this while the character is sitting down with code. The correct way to do this is to use a combination of some simple AI and blend tree animation, together with Unity's built-in Navigation System.

Navigation setup

As we explored in detail in the previous chapter, Unity provides a simple and powerful system for AI path finding called Navigation System. When doing level building/level design, or prototyping your game, it is common to leave the **Navigation** window open, perhaps docked, either side of the **Inspector** and the **Lighting** panels. To open it, go to **main menu | Windows | Navigation** and simply drag it near the **Inspector**. It will automatically dock inside of it. This layout is very handy, but has the downside that you can't show both the **Inspector** and the **Navigation** window, which can be useful sometime.



You can also add the **Occlusion** window to update your occlusion data whenever you move a tree, a building, or anything that can occlude the view in your level. To open **Lighting**, **Navigation**, and **Occlusion** windows, look in the top menu: **Windows**.

Baking the NavMesh

Before hitting the **Bake** button, we need to prepare our scene by choosing what models, in addition to the terrain, will be static objects that will actually block the AI path. Select all the visuals and check their static checkboxes in the **Inspector**. In the **Object** panel of the **Navigation** window, you can do the same, as well as choose whether this mesh will be *walkable* or not, or even reachable with a jump or an off-mesh link. Now you can switch to the **Bake** tab and hit the **Bake** button.



It's important to choose the correct Nav Mesh Agent radius and height in the **Bake** panel, to be sure that the navigation mesh will be good for the Nav Mesh Agent(s) that will use it.

Unity will warn you if you choose an **Agent Radius** and **Agent Height** that are not suitable for the selected **Max Slope** angle.

Adding the rock stone and NPCsitSpot and NPCStartSpot points

Search the **Project** view for the word *rock* and drag in the scene and the rock mesh from the Unity terrain assets. Place it on the left-hand side of the fire, in an appropriate spot. Mark this `GameObject` as Navigation Static (or everything Static). Now, rebuild the NavMesh by pressing the **Bake** button again. Notice how the NavMesh has been carved in proximity to the stone mesh. As you can see, the Navigation System doesn't need a collider to calculate the NavMesh, and any Mesh object in the scene marked as **Static** will be considered in for the calculation:

1. Create an empty `GameObject` and rename it in `NPCsitSpot`.
2. Drag this object as a child of the rock mesh and position it in front of the stone.
3. Select the **Navigation** window to see the NavMesh in the **Scene** view and optimize its position by putting the empty `GameObject` where the NavMesh is walkable, otherwise, the Nav Mesh Agent will never be able to reach that point.
4. Create another empty `GameObject` and rename it in `NPCstartSpot`. Position it at the NPC's feet. This will be the point where the NPC will walk to when it has finished speaking, before once again starting the Walk in Circle loop animation.

Writing the Simple AI class

What is needed now is some scripting for the NPC. Let's dive into the `SimpleAI.cs` class that we created earlier on the NPC and break down the whole code step-by-step.

We encapsulate the class into the same

`UnityStandardAssets.Characters.ThirdPerson` namespace, to be able to access other classes in it:

```
using UnityEngine;

namespace UnityStandardAssets.Characters.ThirdPerson
{
```

We will use the `[RequireComponent]` statement to be sure that our AI GameObject will include `NavMeshAgent` and `ThirdPersonCharacter` components:

```
[RequireComponent(typeof (NavMeshAgent))]  
[RequireComponent(typeof (ThirdPersonCharacter))]
```

We will extend the `MonoBehaviour` class instead of extending the `AICharacterController` class because we are going to re-write it from scratch and define other variables and methods:

```
using System.Collections;  
using UnityEngine;  
  
namespace UnityStandardAssets.Characters.ThirdPerson  
{  
    [RequireComponent(typeof (UnityEngine.AI.NavMeshAgent))]  
    [RequireComponent(typeof (ThirdPersonCharacter))]  
    public class SimpleAI : MonoBehaviour  
    {  
        // the navmesh agent required for the path finding  
        public UnityEngine.AI.NavMeshAgent agent { get; private set; }  
        // the character we are controlling  
        public ThirdPersonCharacter character { get; private set; }  
        // if this is set to true, character won't engage the player  
        // for a fight  
        public bool isNPC;  
        // target to aim for enemy Simple AI, NPC instead will assign  
        // this by code  
        public Transform target;  
  
        // When the distance is below this threshold enemy will start  
        // chasing  
        public float distanceForEngage = 15f;  
        public GameObject dialogueTrigger;  
  
        // Enemy AI variables  
        // The target is in line of sight  
        private bool targetInSight;  
  
        // NPC Variables  
        private bool startedTalkingPhase;  
        private bool reachedStartPoint;  
        private bool startWalkBack;  
  
        private void Start()  
        {  
            // get the components on the object we need
```

```
agent = GetComponentInChildren<UnityEngine.AI.NavMeshAgent>
();
character = GetComponent<ThirdPersonCharacter>();
agent.updateRotation = false;
agent.updatePosition = true;
if (isNPC) agent.SetDestination(transform.position); else
targetInSight = true;
}

private void Update()
{
Quaternion rotation = Quaternion.Euler(new Vector3(0f, 10f,
0f));
// Simple AI Logic for NPC / AI
if (targetInSight)
{
if (target != null)
{
if (Vector3.Distance(transform.position,
target.position) <= distanceForEngage)
agent.SetDestination(target.position);
else
agent.SetDestination(transform.position);
}

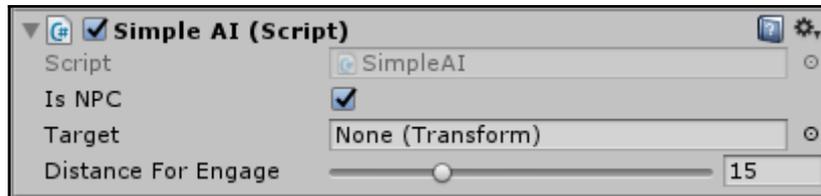
if (agent.remainingDistance > agent.stoppingDistance)
{
if (agent.remainingDistance > 7) agent.speed =
0.8f; else agent.speed = 0.4f;
character.Move(agent.desiredVelocity, false,
false);
}
else
{
// Execute when destination is reached only for NPC
if (isNPC)
{
if (!startedTalkingPhase)
{
startedTalkingPhase = true;
dialogueTrigger.SendMessage
("ShowAnswerButtons");
reachedStartPoint = false;
startWalkBack = false;
}
// adjust rotation to face camera/player
transform.rotation =
Quaternion.Slerp(transform.rotation, rotation,
```



```
        agent.CalculatePath(target.position, path);
        agent.SetPath(path);
    }

    // enabling NPC to seek a destination
    public void SetTarget(Transform target)
    {
        targetInSight = true; this.target = target;
        startedTalkingPhase = false;
        GetComponent<Animator>().SetFloat("Blend", 0.0f);
        agent.SetDestination(target.position);
        character.AnimatorSetWalkCircle(false);
    }
}
```

This is how the component will look in the **Inspector** after coding and setting up the NPC:

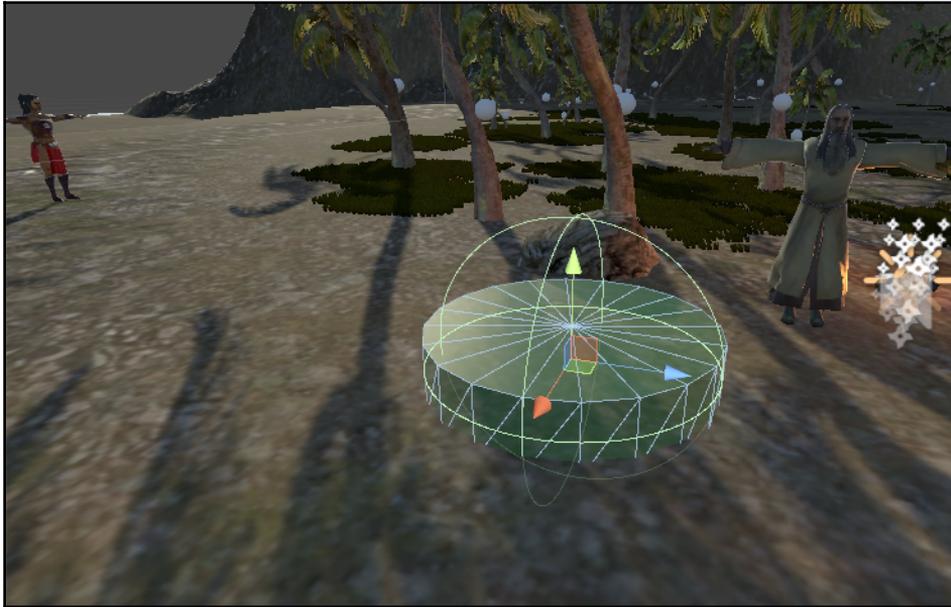


Let's see how to make the player interact and have a talk with the non-playing character.

NPC interaction

It is essential to set up an area around the non-playing character where the player will be able to interact with him. For this task, the best option is to use a Sphere Collider as a Trigger. For this book, I decided to use a Cylinder mesh using a transparent shader with a Sphere Collider component attached.

You can customize the look of your trigger or leave it invisible by using an empty GameObject with just the collider component attached:



The trigger area GameObject created near the rock, the sit place for the NPC

Triggering the dialogue

1. Create a new sphere GameObject from the top menu: **GameObject 3D-> | Sphere.**
2. Optionally remove the Mesh Filter and the Mesh Renderer components from the GameObject (alternatively, you can give assign to the Mesh Renderer a transparent shader material).
3. Resize the **Transform** scale to **1.0, 0.25, 1.0** in the **Inspector.**
4. Set the **isTrigger** option to **True** for the Sphere Collider.
5. Add a new component to the GameObject and call it `DialogueTrigger`.

Let's see how to write the C# class in order to manage the player entering the area, as well as managing and activating the dialogue.

The `MonoBehaviour` class we are going to write is peculiar due to the absence of any `Start/Awake` methods and `FixedUpdate /Update` methods. Because this is an event listener class, it will listen only to the trigger events and will not need any special initialization or update at every frame. Instead, it will start with a bunch of public variable declarations and references to `GameObjects` in the scene that we will have to fill in later for the inspector of the component: the two cameras, the player character, the NPC, the `NPCstartSpot` and `NPCsitSpot` `GameObjects`, the `DialogueUIPanel` and `PromptUIPanel` `GameObjects`, and the two UI button `GameObjects` responsible for choosing an answer during the dialogue:

```
using UnityEngine;

public class DialogueTrigger : MonoBehaviour
{
    // Player Character GameObject
    public GameObject PlayerCharacter;
    // Player Character GameObject
    public GameObject NPC;
    // Primary Camera (multipurpose rig usually)
    public GameObject Camera1;
    // Special handy camera for Dialogue phase
    public GameObject Camera2;
    public Transform NPCsitSpot;
    public Transform NPCstartSpot;
    public GameObject PromptUIPanel;
    public GameObject DialogueUIPanel;
    public GameObject TruthButton, LieButton;
```

We will have two public methods for starting and stopping the dialogue from external classes:

```
// Public methods to be called from UI prompt panel or external classes
public void StartDialogue()
{
    DialogueUIPanel.SetActive(true);
    PromptUIPanel.SetActive(false);
    NPC.SendMessage("SetTarget", NPCsitSpot);
}

public void StopDialogue()
{
    PromptUIPanel.SetActive(false);
    DialogueUIPanel.SetActive(false);
    Camera1.SetActive(true);
    Camera2.SetActive(false);
    GetComponent<Collider>().enabled = true;
```

```
    NPC.GetComponent<Animator>().SetFloat("Blend", 1f);
    NPC.SendMessage("StandUpAndWalk", NPCstartSpot);
    PlayerCharacter.SendMessage("SetTalk", false); // unlock player
}
```

And we will also write some custom methods to show/hide the UI parts interested in the dialogue, as follows:

```
// public methods used by the Dialogue Manager for the UI
public void ShowAnswerButtons() {
    TruthButton.SetActive(true);
    LieButton.SetActive(true);
}
public void HideAnswerButtons() {
    TruthButton.SetActive(false);
    LieButton.SetActive(false);
}
public void UpdateAnswerButtonsText(string str1, string str2){
    LieButton.GetComponentInChildren<Text>().text = str2;
    TruthButton.GetComponentInChildren<Text>().text = str1;
}
public void ShowAnswerButton() {
    AnswerButton.SetActive(true);
}
public void HideAnswerButton() {
    AnswerButton.SetActive(false);
}
public void UpdateAnswerButtonText(string str1) {
    AnswerButton.GetComponentInChildren<Text>().text = str1;
}
public void ClosePrompt()
{
    PrompUIPanel.SetActive(false);
    GetComponent<Collider>().enabled = true;
    DialogueUIPanel.SetActive(false);
    Camera1.SetActive(true);
    Camera2.SetActive(false);
    // set the StandUp animation
    NPC.GetComponent<Animator>().SetFloat("Blend", 1f);
    NPC.SendMessage("StandUpAndWalk", NPCStartSpot);
    PlayerCharacter.SendMessage("SetTalk", false); // unlock player
}
```

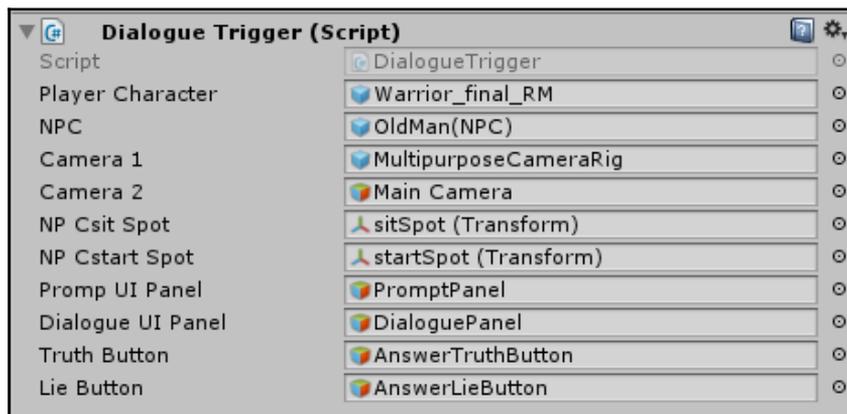
Finally, we will add a boolean variable, and check the `OnTriggerEnter` to activate the prompt to start the dialogue. We will check the `OnTriggerExit` events to reset `Renderer` component state and `dialogueIsActive` to false:

```
private bool dialogueIsActive;

void OnTriggerEnter(Collider col)
{
    if (col.gameObject.tag == "Player" && !dialogueIsActive)
    {
        PlayerCharacter.SendMessage("SetTalk", true); // lock it
        Camera1.SetActive(false);
        Camera2.SetActive(true);
        PromptUIPanel.SetActive(true);
        dialogueIsActive = true;
        PromptUIPanel.SetActive(true);
    }
}

void OnTriggerExit(Collider col)
{
    GetComponent<Renderer>().enabled = true;
    dialogueIsActive = false;
}
```

Save the script and then go back to the editor to see the effects in the **Inspector**. They should look like the following image:



The component look after saving and after Unity compiled the code and assigned the references

To complete the setup we will write two custom methods that will be called by the `DialogueManager` each time it is needed. This will work as an expression when the NPC *speaks* by performing a parameter change on the NPC's Animator to play the animations. We do it in this way because the `DialogueTrigger` class has a reference to the NPC `GameObject` already and this lets us easily access its Animator component. The floating number we pass in will be used to set the Animator's parameter we have set up to manage the Blend States. We will declare this `public` to let it be accessible from external classes. As a side note, the value should not be just passed in, but slightly moved to one value from an older value with a `Mathf.Lerp()` instruction to achieve better animation blending results (see `DialogueManager` class later in this chapter).

Alongside the `SetBlendNPC` method we will add a `GetBlendNPC` public method to retrieve the current parameter value:

```
// public method to force/set the Blend parameter value
public void SetBlendNPC(float blend){
    NPC.GetComponent<Animator>().SetFloat("Blend", blend); }
// retrieve actual blend state value
public float GetBlendNPC(){ return
    NPC.GetComponent<Animator>().GetFloat("Blend"); }
```

Writing the DialogueManager class

We will write a simple class to manage the UI and *hard-code* the text content required for the dialogues in two string arrays, one for NPC speech, and one for the player. Hard coding means to have contents like strings embedded in the code rather than utilize some more engineered methods that, for example, read the content from files. For simplicity, we are going to build a linear dialogue.



Even though this approach can be good while prototyping your game, I wish to invite you to consider Unity Editor scripts and XML for managing complex dialogues and scenes, or even better, if you want to work at a professional level, dive into products such as *Articy Draft* at: <http://www.articydraft.com>, and see the amazing tools they offer for writing complex videogames. These can also integrate easily in Unity to enrich your RPG or adventure games.

Let's take a look at the code. First of all, we will add the `UnityEngine.UI` framework so that we can access UI components:

```
using UnityEngine.UI;
```

Then, we will define two public variables for later specifying the dialogue UI panel and the `DialogueTrigger` references:

```
public class DialogueManager : MonoBehaviour {
    public GameObject DialogueLog;
    public DialogueTrigger dialogueTrigger;
```

We will define two private array variables to store the real text of the dialogues to be able to later decide what sentence should be picked up for the two characters:

```
private string[] OldManSpeech = new string[9];
private string[] PlayerSpeech = new string[9];
```

We will use an enum type to describe the dialogue's steps to make our code more readable. We will define it as a public enum of our custom `DialogProgress` type, a public variable for storing the current progress status and a private variable to store the previous status:

```
public enum DialogProgress
{
    Approach,          // 0
    Bye,               // 1
    Interest,          // 2
    Knowledge,         // 3
    Story,             // 4
    Deal,              // 5
    Quit,              // 6
    StartAdventure,   // 7
    ThanksBye         // 8
}
private DialogProgress progress; // dialogue status progress
private DialogProgress oldProgress; // store the older status
```

We will then add two arrays with 10 slots each, to hold the dialogue sentences of the two characters:

```
// Define 2 arrays for storing the speeches of the two characters
private string[] OldManSpeech = new string[9];
private string[] PlayerSpeech = new string[9];
```

We will initialize the two string arrays in the `Start ()` method, executed when the script starts, with the speech dialogue strings between the two characters:

```
OldManSpeech[0] = "Oldman:I can't believe my eyes, but it's true, you are
not a guard..";
PlayerSpeech[1] = "Hero: I am just a native from this island";
```

To check the full string collection of sentences, look at the ready-made class source code.

We will use the `Update` method to print out the correct sentences of the dialogue progress; the code will be executed only if the progress has changed:

```
// Update is called once per frame
void Update () {
    if(progress!=oldprogress)
    {
        string contentLog = DialogueLog.GetComponent<Text>().text;
        contentLog = OldManSpeech[(int)progress] + "\n" +
        PlayerSpeech[(int)progress] + "\n" + contentLog + "\n";
        DialogueLog.GetComponent<Text>().text = contentLog;
    }
}
```

For commodity we could fold this code into a private method, we could call: `ProgressAdvance ()`, and call it in the update when the `progress!=oldprogress` condition is met, making the code a lot simpler to read:

```
// Update is called once per frame
void Update () {
    if (progress != oldprogress)
    {
        oldprogress = progress;
        ProgressAdvance();
    }
}
```

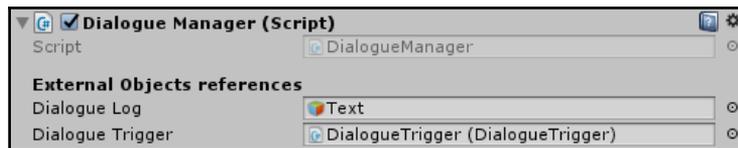
The `ProgressAdvance()` method will be also useful later on, when we will extend our code to implement NPC animation blending:

```
// Folding part of update code
private void ProgressAdvance()
{
    string contentLog = DialogueLog.GetComponent<Text>().text;

    // reset interpolation to start a new blend
    previousBlendValue = dialogueTrigger.GetBlendNPC();
    animationBlendTarget = (0.1f + (float)progress * 0.1f);
    interpolation = 0.001f;

    // update
    contentLog = OldManSpeech[(int)progress] + "\n" +
    PlayerSpeech[(int)progress] + "\n" + contentLog
    + "\n";
    DialogueLog.GetComponent<Text>().text = contentLog;
}
}
```

Saving now the script and going back to the editor should show this for the component in the **Inspector** after finish compiling:



The Dialogue Manager component in the Inspector after saving the declaration part and filled the references in its slots

The `IncProgress` (short for increment progress) method will take care of increasing the state of the dialogue by a given number and manage the result of this increment by changing the UI and populating the message log window. It is a very simple and rough logic, but it will make simpler the implementation of something that is not to easy and quick. While there are plenty of ready made dialogue systems on the Assets Store, we want to implement this ourselves for the sake of learning. This method will be called by the UI when the player starts the dialogue or after the player gives an answer.

The logic is implemented through a quite big `switch()` statement in the body of the method:

```
// public method called by the UI buttons, increments the progress and
acts accordingly
public void IncProgress(int q)
{
    // this line to support increment parameter
    if (q > 1) progress += q; else ++progress;

    switch (progress)
    {
        case DialogProgress.Approach:
            // we don't need any specialcode for the initial status
            break;
        case DialogProgress.Bye:
            dialogueTrigger.HideAnswerButtons();
            dialogueTrigger.StopDialogue();
            break;
        case DialogProgress.Interest:
            dialogueTrigger.HideAnswerButtons();
            dialogueTrigger.ShowAnswerButton();
            break;
        case DialogProgress.Knowledge:
            dialogueTrigger.UpdateAnswerButtonText("Merchant?");
            dialogueTrigger.ShowAnswerButton();
            break;
        case DialogProgress.Story:
            dialogueTrigger.UpdateAnswerButtonText("How?");
            dialogueTrigger.ShowAnswerButton();
            break;
        case DialogProgress.Deal:
            dialogueTrigger.HideAnswerButton();
            dialogueTrigger.UpdateAnswerButtonsText("Accept",
            "Decline");
            dialogueTrigger.ShowAnswerButtons();
            break;
        case DialogProgress.Quit:
            dialogueTrigger.HideAnswerButtons();
            dialogueTrigger.StopDialogue();
            break;
        case DialogProgress.StartAdventure:
            dialogueTrigger.HideAnswerButtons();
            dialogueTrigger.UpdateAnswerButtonText("Thanks him");
            dialogueTrigger.ShowAnswerButton();
            break;
        case DialogProgress.ThanksBye:
```

```
        dialogueTrigger.UpdateAnswerButtonText("Greet him and  
        go");  
        break;  
        case DialogProgress.FoundPieces:  
            dialogueTrigger.StopDialogue();  
            dialogueTrigger.EndSuccessful();  
            break;  
    }  
}
```

Its public methods are called to choose what dialogue UI GameObject to show hide or update:

- The `HideAnswerButtons()` and `ShowAnswerButtons()` methods will hide/show the correct buttons for answering questions or simply speaking (note the plural and singular version of the methods).
- The `UpdateAnswerButtonsText()` method changes the caption of the buttons when needed.
- The `StopDialogue()` method will stop the dialogue and go back to the initial status where the player can move; the following camera is restored and the NPC will go back to walking in a circle.
- The `EndSuccessful()` method will take care of marking this dialogue concluded and heal the player from his wounds.

Tying up the logic and UI events

The `ResetDialogue()` public method will be called by the UI `OnClick` event of the `PromptPanel` button and will reset the progress, the log message window, and the button captions at their initial state:

```
public void ResetDialogue()  
{  
    Debug.Log("dialogue reset");  
    DialogueLog.GetComponent<Text>().text = "";  
    dialogueTrigger.UpdateAnswerButtonsText("Truth", "Lie");  
    dialogueTrigger.UpdateAnswerButtonText("Hungry");  
    progress = 0;  
}
```

We will add a public method to the `ThirdPersonCharacter` class used by the NPC to call from our `DialogueTrigger` to start or stop the dialogue sequence for NPC's animation:

```
// Chapter 8 NPC
public void AnimatorSetWalkCircle(bool flag)
{
    m_Animator.SetBool("Walking", flag);
}
```

This method will take care of initiate or stop the Blend Tree State in the NPC's Animator logic when the dialogue starts or ends.

Press Play to finally test your work. You should see the hero character and the NPC at a fire and his hut. Walk toward the rock near the fire, and you should see the `PromptPanel` dialog asking to start the dialogue. If you press the **NO** button, you will return to free roaming, pressing **YES** will start the dialogue and the player will be locked from movements until the dialogue is terminated.

Driving Animator Blend Tree with scripting

To give some color to our dialogue, we want to smoothly step from one animation clip to another by slightly changing by scripting the parameter that drives our Idle Blend Tree we have setup earlier in this chapter when another part of the dialogue is initiated.

To do this we will add four private float variables first at the end of our declaration part:

```
private float animationBlendValue, animationBlendTarget,
previousBlendValue, interpolation;
```

In the `Update()` method after the block that calls the `ProgressAdvance()` method we will add:

```
if (interpolation > 0)
{
    interpolation += 0.5f * Time.deltaTime;
    animationBlendValue = Mathf.Lerp(previousBlendValue,
    animationBlendTarget, interpolation);
    if (interpolation >= 1) interpolation = 0;
}
```

As you can read in the comment we are using the decimal values to choose animation clips, but we need to change the `animationBlendValue` variable slightly with a `Mathf.Lerp` function to see a smooth result when passing this value to the Blend Tree parameter. We alter the `interpolation` variable adding a constant value multiplied by `Time.deltaTime` to keep a constant framerate on all the type of hardware, and finally, we set the Blend Tree parameter with our `SetBlendNPC` method we added earlier to the `DialogueTrigger` class.

The last step will be to reset the `interpolation`, `previousBlendValue`, and `animationBlendTarget` variables for the next blend to happen when the dialogue progress status change:

```
// reset interpolation to start a new blend
previousBlendValue = dialogueTrigger.GetBlendNPC();
animationBlendTarget = (0.1f + (float)progress * 0.1f);
interpolation = 0.001f;
```

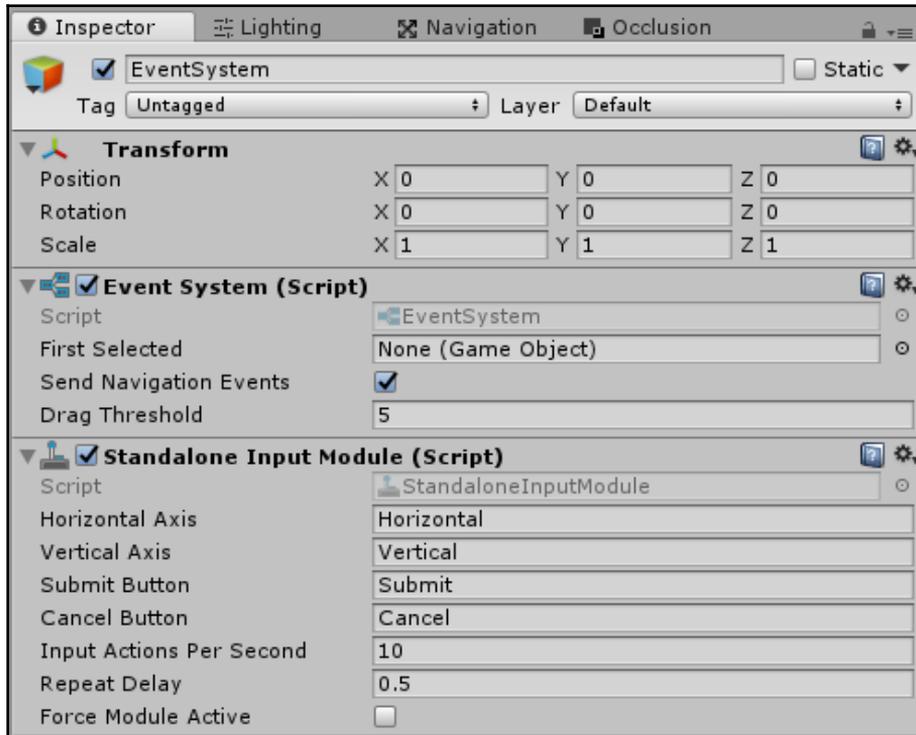
Press Play to finally test your work. You will see the NPC changing animation in the different phases of the dialogue; sitting down at the beginning and standing up when starting to talk, moving the arms, talking, and other gestures with the next steps of the talk.

Creating a basic UI for displaying the dialogue

We will create a very basic UI for managing the dialogue for later continuing the exploration of the UI system in [Chapter 9, AI, NPC, and Further Scripting](#), and [Chapter 12, Designing Menus with Unity UI](#). We will keep it very simple and create the necessary UI elements in order to display the text of the talk and make the player able to answer some of the questions.

Creating the Canvas

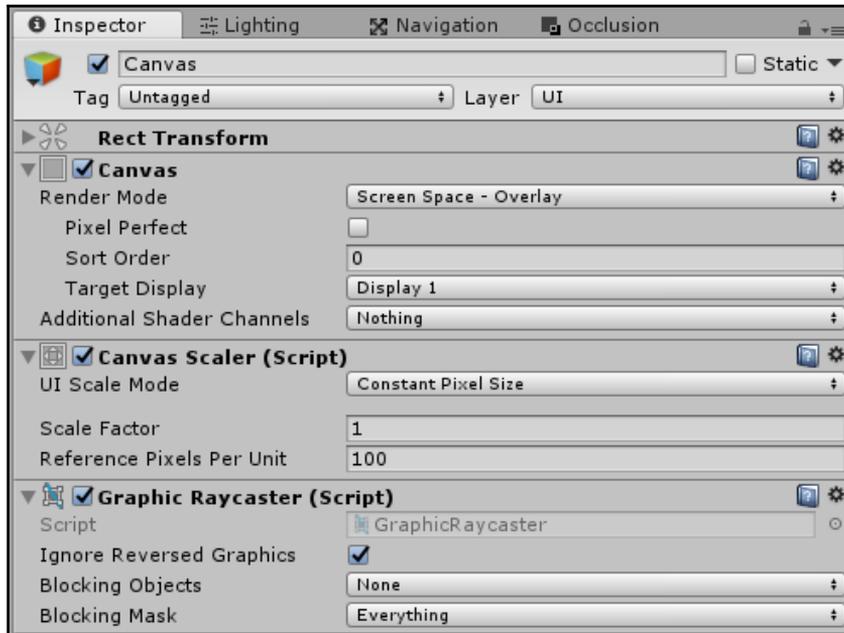
Create a new Canvas from the top menu: **GameObject** | **UI** | **Canvas**. This operation will automatically create an `EventSystem` `GameObject`, which has attached special components needed to take care of all UI events and input:



The `EventSystem` `GameObject` shown in the Inspector

If a `Canvas` and `EventSystem` are not present in the scene, and you create the first UI `GameObject`, Unity adds them for you. While it can coexist more than one `Canvas` in a scene, only one `EventSystem`, needed for managing multiple `Canvas`.

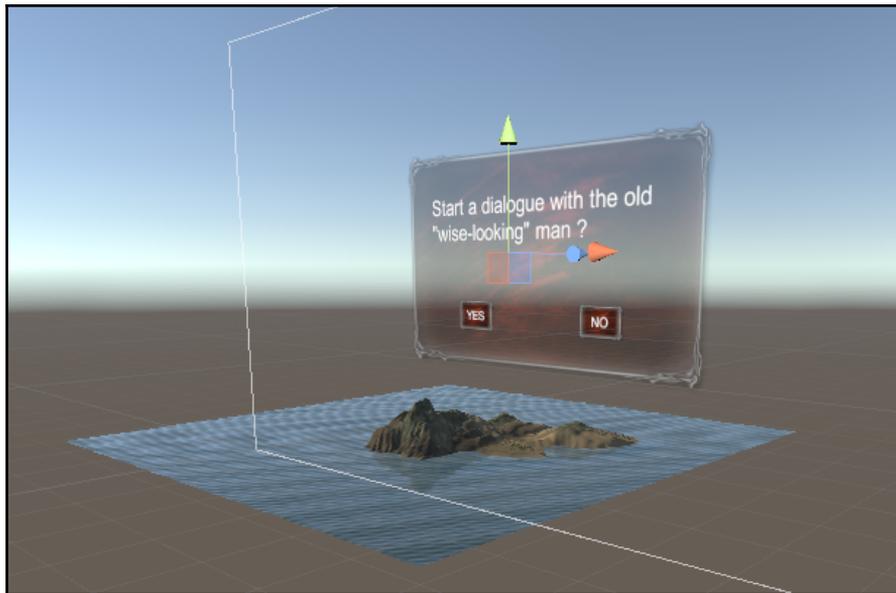
For managing the dialogue UI and later the HUD, we will use a single `Canvas` choosing **Screen Space - Overlay** for the render mode. This mode renders UI objects over the rest of the 3D scene:



The Screen Space Overlay Canvas in the Inspector

Creating the start dialogue prompt

We will need a small box displaying a message prompting the user to start the dialogue with the NPC or leave. To do that, we will use one UI panel and two UI buttons. Call the UI panel `PromptPanel` and assign it to the `DialogueTrigger` component we prepared. On the **No** buttons, we will add an `OnClick` event that will hide the UI prompt panel and return to exploration mode, while on the **Yes** button, we will trigger the dialogue with a series of instructions:



The `PromptPanel` with buttons seen in the Scene view

As you can see there is no need to put this panel in front of the camera at a small dimension because we are using a `Screen Space - Overlay` render mode for the Canvas. In this mode, the UI is automatically shrunk to screen size.

Creating the dialogue window

The dialogue window will be a UI panel containing a UI `ScrollView`, with a UI `Text` inside the content viewport of the scroll view. To quickly create that, let's start with creating the UI Panel and rename it `DialogPanel`. Leave it selected in the **Hierarchy** and add the three answer buttons: two for when the player has choices and a neutral one for progressing with a simple answer.

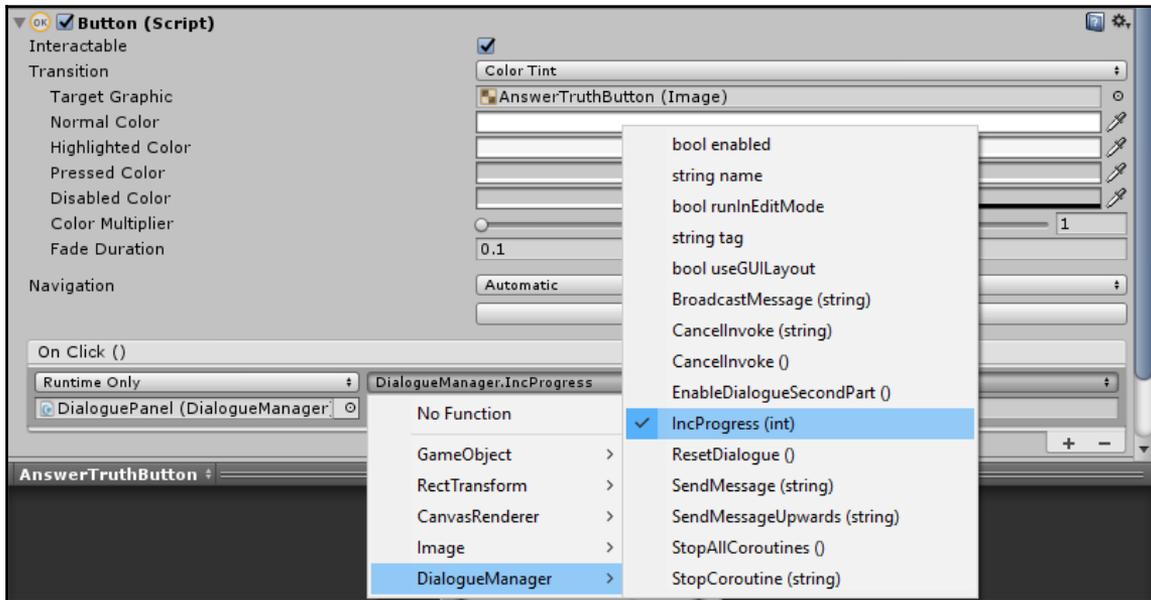
Creating the answer buttons

These buttons will be displayed when the player has two choices, and the text of the buttons will be updated from truth/lie to accept/decline depending on the dialogue progress.

To create the button:

1. Select the Canvas `GameObject` from the main menu **GameObject | UI | Button**. Define the `Text` child of the button that will display the word **TRUTH** and set for the background of the `Image` component the green color, then, on the `Button` component, add the `OnClick` event and an event with the + button.
2. Drag into the slot the `DialogPanel` `GameObject`, and from the list of methods in the pull-down menu, choose `IncProgress`, which will specify that we must increment the dialogue status value. Enter a value of 2 in the value slot to have an increment of two steps at a time. We do this because at each step, we will write the old man's sentence and the player's answer in the dialogue scroll view's text field.
3. Place it on the left-hand side and change the anchor to top-left. When finished, clone this button, change its color to red and place it on the right, changing the anchor of course to top right instead.
4. In the **Runtime Only** slot, drag the `DialogPanel` `GameObject`, then from the methods list choose **`IncProgress (int)`**. After, enter a value of 1 in the value slot.

5. Repeat these steps for the `AnswerNeutral` button; this will be a green color again and positioned at the center of the canvas.
6. When finished, deactivate the `DialoguePanel` as well as the `PromptPanel` GameObjects:



The custom public method we specify visually in the Editor on the `OnClick ()` event method on the button

Making enemy AI

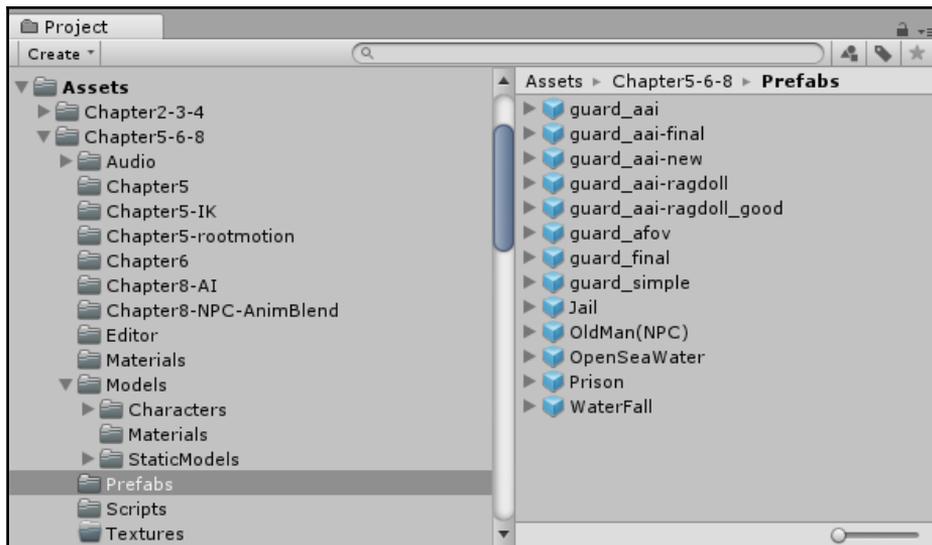
We will make the AI prefab with the same guidelines from Chapter 5, *Character Animation with Unity*, for the hero character, with some differences to the Animator Controller state machine. This prefab will be a modification of the `AIThirdPersonController` class in Unity standard assets, which uses the same class from where we extended our: `ThirdPersonCustomCharacter`, the `ThirdPersonCharacter` class.

The `AICharacterControl` class of the original prefab controls the character movement through the `ThirdPersonCharacter` component. The class can be used to control a player *unit* in a RTS/RPG game, or for controlling enemy in a third person/first person game. Here is where we will start at, extending this class and renaming it into our new `AdvancedAI` class, which will regulate the behavior of the guards in our game and, in the same way, will manage the character movement through the `ThirdPersonCustomCharacter` component.

Using Unity Standard Assets in our favor

If you want to skip the prefab creation use the ready-made prefab, which has basically a very similar structure to the `AIThirdPersonController` prefab in the Unity's **Standard Assets | Characters** package and will have a copy clone of the Animator Controller with some change and adjustment.

These prefabs are located in the folder `Chapters5-6-8/Prefabs/` of the book's codes:

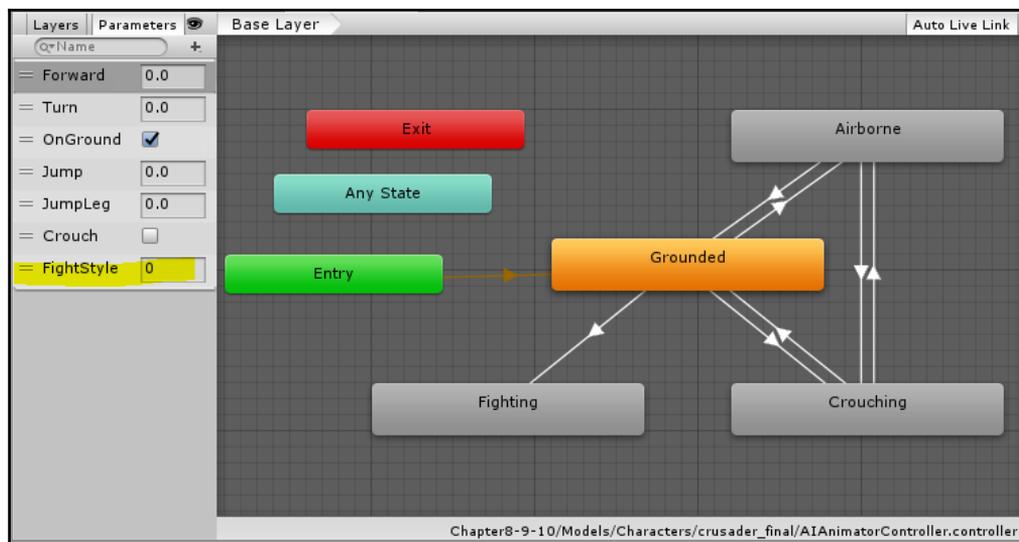


In this folder, you will find different versions of the AI with various steps and evolutions, from `SimpleAI` class driven (`guard_simple`), to the `AdvancedAI` class driven `guard_aai-final` with ragdoll physics, footsteps sounds, `FieldOfView`.

The only major differences with the Standard Assets Animator Controller will be:

- A new state in the guard's Animator Controller compared to the original, the Fighting State (see next screenshot).
- The AdvancedAI class that we are going to write, that will replace the standard one and an additional sphere collider trigger, will be used optionally for adding depth to let AI chase or find the player when this goes out of the trigger.

This approach will be perfect for making our SimpleAI or AdvancedAI classes both rely on the ThirdPersonCharacter class from the Standard Assets:



The AIAnimatorController layout in the Animator window

Making the AI smarter

As you can see, both the Unity Standard Assets AI example and our SimpleAI modified version we used for the NPC are very simple. The target to follow is set in advance in the component slot at edit time and the guards will always chase the player, no matter what, even if they are far away or out of sight! Let's kick in something more advanced for our AI.

The ready made prefab we are going to use for the guard AI can be easily done from scratch, with the same technique we used for the old man NPC. It is basically an `AIThirdPersonController` prefab with a Rigidbody, a Capsule Collider, and a Nav Mesh Agent component attached to the main `GameObject` and the crusader skinned mesh character as a child of the main object:



The guard_aai-final prefab in the Hierarchy

Duplicate ThirdPersonCharacter class

From now on we will work with a new class, a duplicate of the Standard Assets class: `ThirdPersonCharacter` that we will call: `ThirdPersonCustomCharacter` mainly for two reasons:

- We want to keep older prefabs we worked on as well as Standard Assets prefabs working in the old fashion
- We want to keep the implementations separate so as to avoid an overwrite of the files when you update Standard Assets, or eventually decide to not import them and start from scratch

To do so, open with a double click on the script slot in the component of the `ThirdPersonCharacter`, in Windows 7/10 if you installed Unity Visual Studio component and Microsoft Visual Studio Community 2017 the **Integrated Development Environment (IDE)** will open with the actual project solution loaded with this file open.

In the class header for now, just change the Class name from `ThirdPersonCharacter` to `ThirdPersonCustomCharacter` then, finally, save the file as `ThirdPersonCustomCharacter.cs`.

Go back to the editor and wait for the code compiling ends, you should see no errors in the console.

Creating the Advanced AI Controller class

In this class, some concepts of co-routines execution as well as ray casting and triggers will be used.

Let's break through the whole class step by step:

First, we will encapsulate the class into the same `UnityStandardAssets.Characters.ThirdPerson` namespace to be able to access other classes in the same. But also for making the code compatible with the various step of the book and its prefabs, where different classes were used in together with new ones until the end of the book, where the classes will be stand alone and be eligible of being removed from the namespace.

Also, we will add `System.Collections` to be able to use co-routines:

```
using System.Collections;
using UnityEngine;
namespace UnityStandardAssets.Characters.ThirdPerson
{
```

We will `[RequireComponent]` a `NavMeshAgent` and a `ThirdPersonCustomCharacter` component to be added to the same `GameObject` that carries this class:

```
[RequireComponent (typeof (NavMeshAgent))]
[RequireComponent (typeof (ThirdPersonCustomCharacter))]
```

We will simply extend a `MonoBehaviour` class instead of the `AICharacterController` to re-write all the methods from scratch and define all the variables:

```
public class AdvancedAI : MonoBehaviour
{
    public NavMeshAgent agent { get; private set; }
    public ThirdPersonCustomCharacter character { get; private set; }
}

public Transform target;           // target to aim for
// Advanced AI variables
public bool advancedAI;
public enum State
{
    ROAM,
    CHASE,
    WONDER
}
public State state;
private bool alive;
private Vector3 lastPlayerSeen;
public GameObject[] waypoints;
private int waypointInd = 0;
public float roamSpeed = 0.7f;
//Chase-run
public float chaseWaitTime = 2f;
private float chaseTimer;
public float runSpeed = .75f;
//Wonder
private Vector3 wonderPosition;
private float wonderTimer = 0;
public float WonderWait = 5;
//Sight
public float heightMultiplier;
public float sightDST = 10;
```

At the `Start` method of the class we will initialize the components variables, allowing caching, then we initialize AI variables, and finally, we start the infinite state machine routine:

```
private void Start()
{
    // caching agent and character components
    agent = GetComponentInChildren<NavMeshAgent>();
    character = GetComponent<ThirdPersonCharacter>();
    agent.updateRotation = false;
    agent.updatePosition = true;
```

```
// Set up AI variables
state = State.ROAM;
alive = true;
heightMultiplier = 1.36f;

// Instead of using the Update method
// We will start a coroutine for executing AI state-machine
logic
StartCoroutine("ISM");
}
```

Custom AI state machine

The Infinite State Machine method is a simple co-routine that runs forever and executes code when the `alive` Boolean variable is `true`, which means that the AI has not been killed or, in our case, stunned for a while:

```
IEnumerator ISM()
{
    while (alive)
    {
        switch (state)
        {
            case State.ROAM:
                Roam();
                break;
            case State.CHASE:
                Chase();
                break;
            case State.WONDER:
                Wonder();
                break;
        }
        yield return null;
    }
}
```

Waypoints roaming

When the AI is not alerted, it roams through waypoints transform positions, waypoints are *empty* GameObjects with just the transform component, placed around in the game area, and were specified (dragged) in the component waypoints list slots:

```
void Roam()
{
    agent.speed = roamSpeed;
    if (Vector3.Distance(this.transform.position,
        waypoints[waypointInd].transform.position) >= 2)
    {
        agent.SetDestination(waypoints[waypointInd].transform.position);
        character.Move(agent.desiredVelocity, false, false);
    }
    else if (Vector3.Distance(this.transform.position,
        waypoints[waypointInd].transform.position) <= 2)
    {
        waypointInd += 1;
        if (waypointInd >= waypoints.Length)
        {
            waypointInd = 0;
        }
    }
    else
    {
        character.Move(Vector3.zero, false, false);
    }
}
```

Chasing the target

We will write a `Chase()` method that will be executed when the enemy AI spots the player. We will use a timer to prevent the AI to start immediately the run, then, after this given time, it will start to chase the player:

```
void Chase()
{
    chaseTimer += Time.deltaTime;
    if (chaseTimer < chaseWaitTime)
    {
        agent.SetDestination(this.transform.position);
        character.Move(Vector3.zero, false, false);
    }
    else
    {
```

```
        agent.speed = runSpeed;
        agent.SetDestination(target.position);
        character.Move(agent.desiredVelocity, false, false);
    }
}
```

Back to waypoints roaming

When the AI is in `WONDER` state, a timer is used to unlock it from staying idle and is alerted for who might be around, if nothing is seen the AI will go back to the `ROAM` state:

```
void Wonder()
{
    wonderTimer += Time.deltaTime;
    agent.SetDestination(this.transform.position);
    character.Move(Vector3.zero, false, false);
    transform.LookAt(wonderPosition);
    if (wonderTimer >= WonderWait)
    {
        state = State.ROAM;
        wonderTimer = 0;
    }
}
```

Enemy's Field Of View (or line of sight)

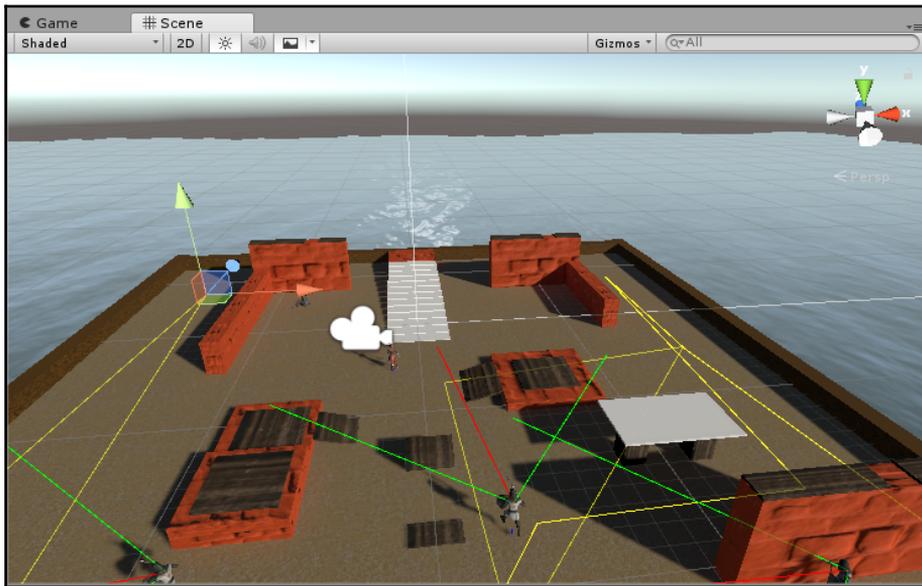
We are going to use the `FixedUpdate` `MonoBehavior`'s standard method to manage raycasting. `FixedUpdate` method is where the engine updates physics and raycasting is part of the physic framework. The first statement creates an empty `RaycastHit` variable to use for storing the result of the cast of the ray:

```
void FixedUpdate()
{
    RaycastHit hit;
```

The first lines are just the debug draw in the **Scene** view of the 3 rays that makes the AI field of view. The forward one in front of the AI and two side ones show the angle of their field of view:

```
Debug.DrawRay(transform.position + Vector3.up * heightMultiplier,
transform.forward * sightDST, Color.red);
Debug.DrawRay(transform.position + Vector3.up * heightMultiplier,
(transform.forward + transform.right).normalized * sightDST, Color.green);
Debug.DrawRay(transform.position + Vector3.up * heightMultiplier,
(transform.forward - transform.right).normalized * sightDST, Color.green);
```

This approach lets us debug visually a bit what's going on behind the scenes with our AIs, to note that the `Debug.DrawRay` instructions will draw the rays only in the **Scene** view:



The **Scene** view showing the AI moving around and their debug rays representing their field of view

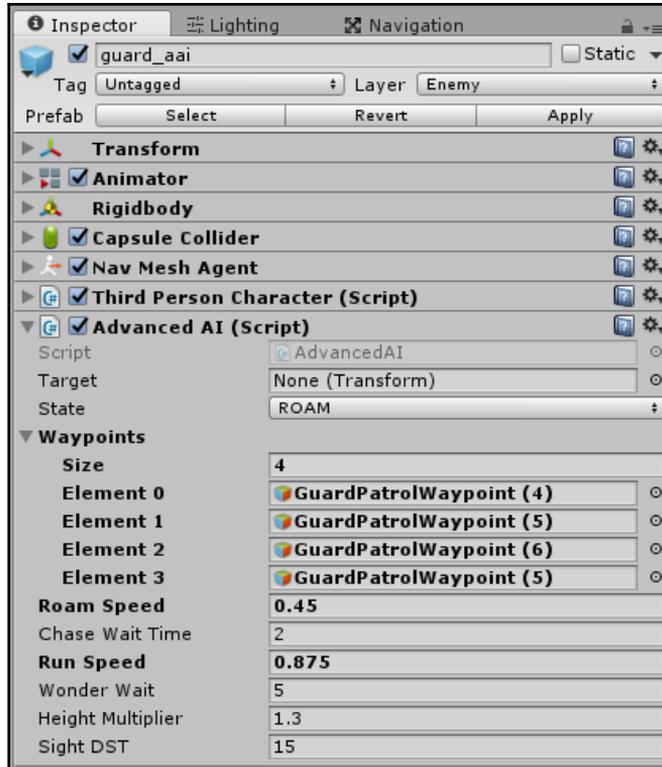
The next lines, after the `Debug.DrawRay` statements, are the real code needed to have the AI check if the player hero is in their view. A ray is cast from the AI position to the straight forward direction for `sightDST` meters (unity units) and checks if it intersects a `GameObject` collider tagged: `Player`, in that case, put the AI in `CHASE` mode so the guard will start chasing the hero:

```
    if (Physics.Raycast(transform.position + Vector3.up *
        heightMultiplier,
        transform.forward, out hit, sightDST))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = State.CHASE;
            target = hit.collider.transform;
        }
    }
    if (Physics.Raycast(transform.position + Vector3.up *
        heightMultiplier,
        (transform.forward + transform.right).normalized, out hit, sightDST))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = State.CHASE;
            target = hit.collider.transform;
        }
    }
    if (Physics.Raycast(transform.position + Vector3.up *
        heightMultiplier,
        (transform.forward - transform.right).normalized, out hit, sightDST))
    {
        if (hit.collider.gameObject.tag == "Player")
        {
            state = State.CHASE;
            target = hit.collider.transform;
        }
    }
}
```

Assuming you have deconstructed `AIThirdPersonControl` prefab instead of using the ready-made one, drag in the scene one then rename it as: `guardaai_final`. Remove the basic Ethan character and add the Crusader model like we did for the NPC earlier in this chapter. Add an `Audio Source` component as well as a `Sphere Collider` component that you will set as a trigger with the **isTrigger** option. Then remove the `AICharacterControl` component and create a new `C#` component called: `AdvancedAI`.

Remove also the `ThirdPersonCharacter` component and add the newly duplicated `ThirdPersonCustomCharacter`.

Now fill the **Third Person Character (Script)** and the **Advanced AI (Script)** components options and be ready to test the scene:



The guard enemy AI GameObject with all its components and our AdvancedAI component



You can tweak Roam Speed and Run Speed differently on different guards to make them a little different from each other.

Player presence awareness

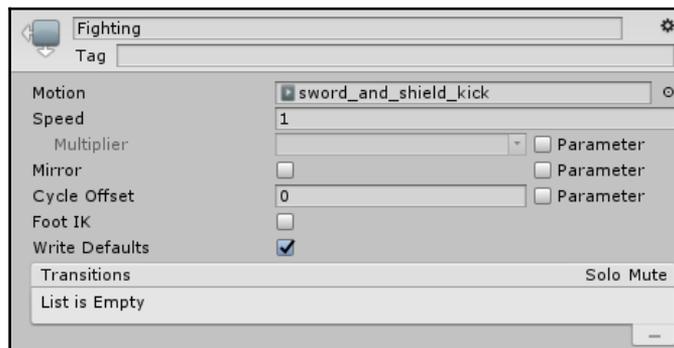
We want to simulate the guard's ability to listen to player presence and noises. To do this we will use the Sphere Collider trigger on the GameObject to check if the player entered this area, in that case, the AI will change its state from ROAM to WONDER with a wonder position assigned, which will be the position of the player when the OnTriggerEnter event method will be fired:

```
// Check AI distance and awareness triggers for player
void OnTriggerEnter(Collider coll)
{
    if (coll.tag == "Player" && currentState != State.CHASE )
    {
        currentState = State.WONDER;
        wonderPosition = coll.gameObject.transform.position;
    }
}
```

We are checking if the object that enters the trigger is tagged Player and that the current state of the AI is not already in CHASE mode, to avoid stopping an already chasing enemy and forcing it in the WONDER state. Finally, we will tweak the Sphere Collider component size to around 5.0 meters (units) around the AI and test the scene to find the best value in terms of gameplay.

Fighting the player

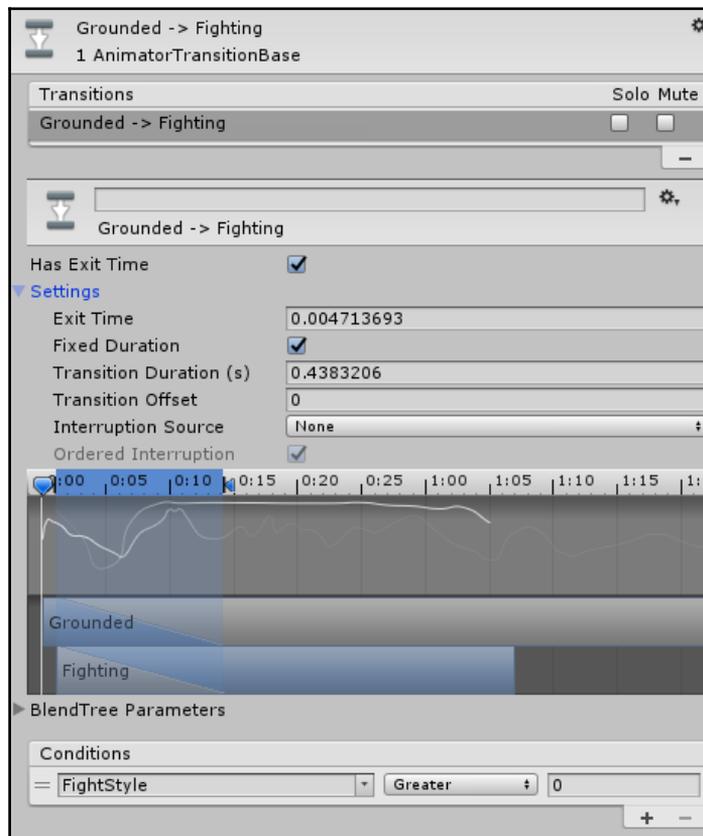
We want the AI attack the player when they get in range, just once, then they will just try to stay close for the arrest:



If the player moves away, the guard will restart chasing, and when the player is close enough, will give another attack hit.

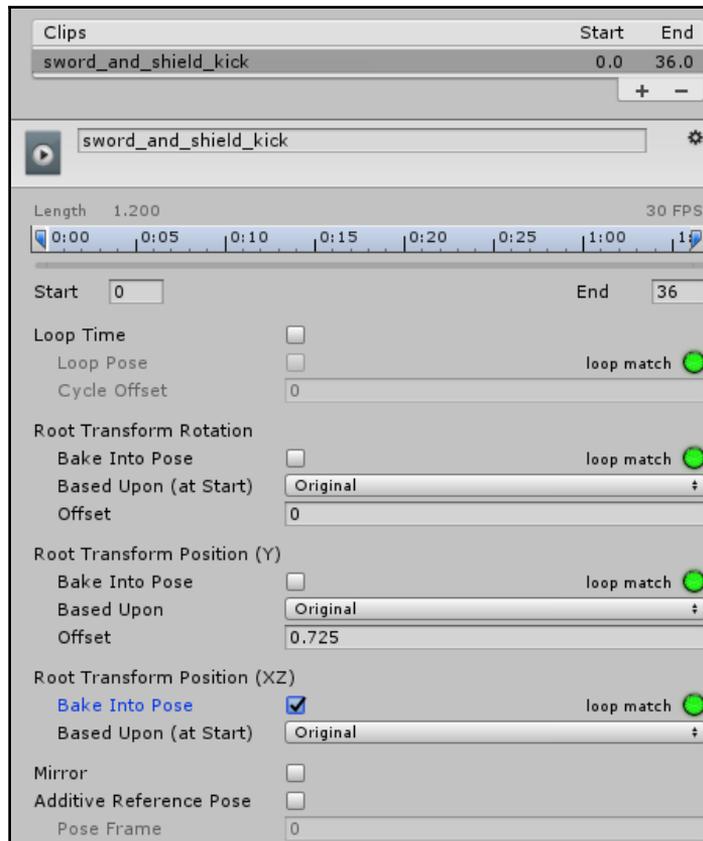
We will design the game in a way that, if the guard is just close to the player, it will remove a certain amount of points, when hit the amount will be bigger.

To implement such a feature, we will need to setup a collider in the guard feet that performs the kick and check collisions against the player to remove more points. First of all, we will set a condition for the fighting animation to happen in the Animator, the `FightStyle` parameter **Greater** than 0, like in this screenshot:



We will add a return transition as well, and set the inverse condition: the **FightStyle** parameter equal to 0. This will help the character to return to the idle animation after he performed the Fighting state animation.

Finally, we will set the animation clip parameters, selecting the clip in the **Project** view and in the **Inspector**, setting its options like in the next screenshot:



As an exercise, add a custom component to the right foot/leg that performs the check of enemy -> player collisions and perform a major loss of energy for the player. You could add the AudioSource for the impact on the same GameObject, where you check the collision on the enemy and play it once when the player is hit.

Modifying the chase method

To enable the Fighting state in the Animator we need to set the **FightStyle** parameter to 1. We will do that by calling the `Fight()` method in the `ThirdPersonCustomController` class, that is also used by overloading the method to `Fight(int FightStyle)` in our player character implementation.

We will need also to write a `StopFight()` method in the same class, to be called when the AI goes back to chase the player who in the meantime has escaped far. In this way, the Animator state will go back to the Grounded blend state.

In the `ThirdPersonCustomController` class, add these two public methods:

```
public void StopFight()
{
    m_Animator.SetInteger("FightStyle", 0);
}

public void Fight()
{
    m_Animator.SetInteger("FightStyle", 1);
}
```

We need to modify the logic of the `Chase()` method in the `AdvancedAI` class a bit in the following way:

```
void Chase()
{
    chaseTimer += Time.deltaTime;
    if (chaseTimer < chaseWaitTime)
    {
        agent.SetDestination(transform.position);
        character.Move(Vector3.zero, false, false);
    }
    else
    {
        agent.SetDestination(target.position);
        float distance = Vector3.Distance(transform.position,
            target.position);
        if (distance > 2.5f)
        {
            character.StopFight();
            agent.speed = runSpeed;
            character.Move(agent.desiredVelocity, false, false);
        }
        else
        {
            if (distance > 1.5f)
            {
                character.StopFight();
                agent.speed = roamSpeed;
                character.Move(agent.desiredVelocity, false,
                    false);
            }
        }
    }
}
```

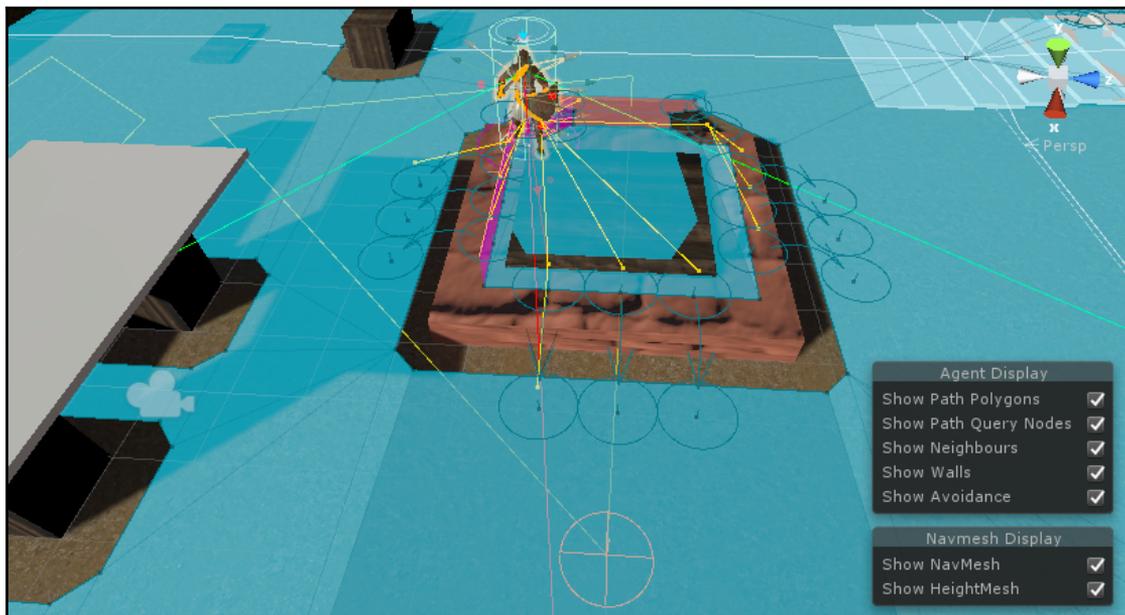
```
        else
        {
            character.Move(Vector3.zero, false, false);
            character.Fight();
        }
    }
}
```

And that's it! Press Play and test the mechanic by rushing away from the guards and see how they slow down when they reach the player before then kicking him.

Debugging the Nav Mesh Agent

The Nav Mesh Agent component of the guards will use Unity navigation system to reach a given target point.

Generating Off Mesh Link will ensure that AI will be able to climb or jump down a ladder or height, the Areas and the costs will establish which one of the possible paths is the best to reach a given target point:

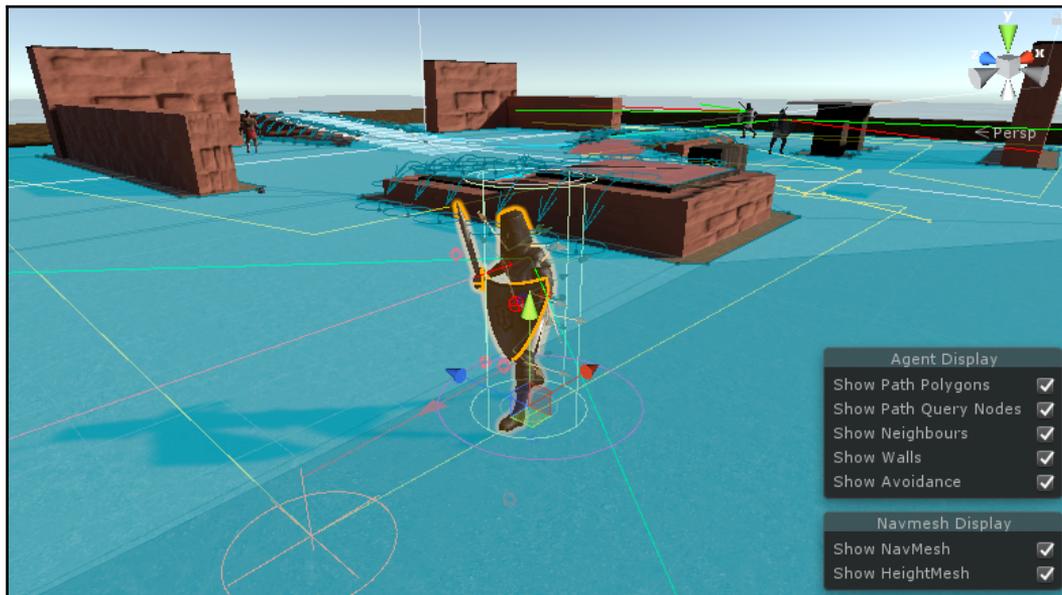


In this image, the guard has a series of choices to evaluate to reach the point

It is important to properly set the quality and the priority of the Agent, but even more important is the radius and the height, which determines the actual size of the Nav Mesh Agent in the world. These settings must reflect the settings you have chosen in the NavMesh bake settings, otherwise there might be spots where a path is narrower than the radius of the Nav Mesh Agent itself which will make it impossible to pass through.

To debug the AI Nav Mesh Agent properly, we will run the scene in the editor and watch the **Scene** view instead of the **Game** window.

Show the **Navigation** window, then choose one of the `guard_aai-final` prefabs we have prepared in the **Hierarchy**. Now, press Play and then switch back to the **Scene** view to see what the AI actually is doing, like in the following screenshot:



Selecting an AI GameObject in the Hierarchy and with the Navigation window open you will see more data in the Scene view

In Chapter 13, *Optimization and Final Touches*, we will place a dozen guards around the island at specific spots and create predefined routes made of custom waypoints for them to follow to make the game more interesting.

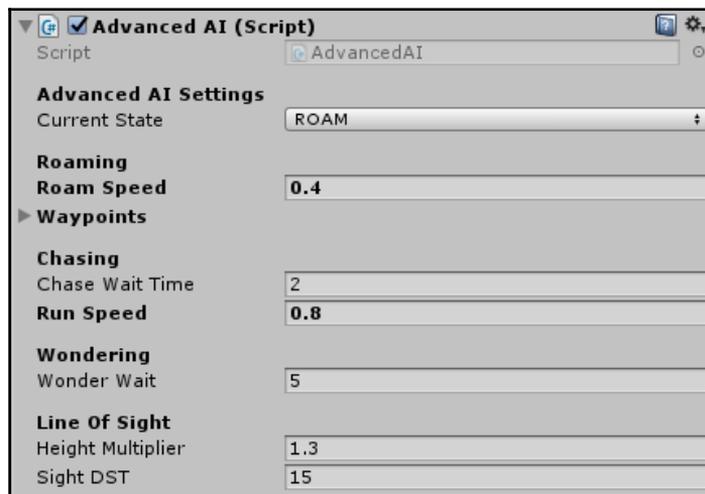
A better look for our custom components with PropertyDrawers

To give a better look to a component with a lot of public references and variables, we can use two standard property drawers to draw section headers and have tooltips on variables in the **Inspector** when the mouse rolls over.

This is the syntax of the two properties:

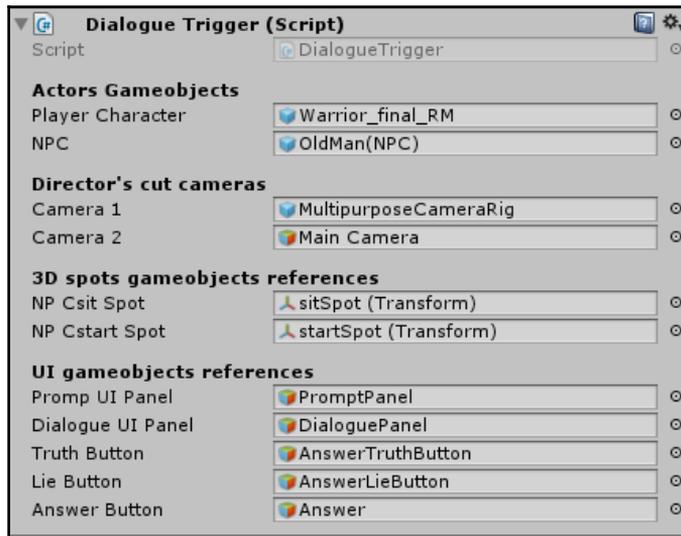
```
[Header("Section A of my component")]
[Tooltip("External reference..specify this and that")]
```

This code needs to be put before the public (or [Serialized]) variable to have it compiled. You can have just one `Tooltip` for each variable and one or more header for grouping them in the **Inspector**:



Our AdvancedAI component after a restyling through Tooltip and Header property drawers

Having a clear reading of custom components in the **Inspector** and increasing a user friendly experience is very important when working a lot of hours per day on Unity Editor. It lets you better focus on the changes to make and the game editing in general, so when a coder delivers their source to the designers, it would help their productivity too:



How the DialogueTrigger component could look after adding headers and tooltips

Further lectures and ideas

To make a better guard in AI, you could add an awareness *circle*, a smaller sphere collider-trigger, when the player enters this trigger, he will be so close to the guard that he could smell you/hear you breathe. You could define a float variable of *awareness* determining the size of this trigger. If you feel brave, you could even try to implement new features—here are some ideas.

Tips for enhancing the AI

Here are a few tips and tricks you can keep up your sleeve while developing:

- Additionally, you could change the `AI` class to implement a feature that will trigger the `CHASE` state for the AI whenever they see other guards chasing.
- A better field of view, possibly drawn at screen, with or without stencil buffer techniques; a ragdoll for guards that is hit by the player. (You will actually find this asset and source code in the project as a bonus, ready to be used).
- Make better enemy *ears*, aware of surrounding noises with triggers and distance calculations. For example, a stone launched near a guard would alert him even if he doesn't see the player.

Summary

Most of the interactive elements of our game have been created now, our player character, an NPC to talk to, and enemy guards AI, ready to roll their part on the beautiful, lonely, scary *Devil Island*, a prison island created to keep heretics like you captive.

In the next chapter, we'll make further use of triggers and collisions. We will create an object-collection game in which the player must find artifact pieces in order to please the old man and have the keys for the hidden boat for leaving the island. We will write the code to disallow entering the hidden place unless the old man is happy and the HUD for displaying game status.

9

Item Collection, Player Inventory, and HUD

Working with a similar approach to the preceding chapter, we will continue our use of trigger collision detection in this chapter, using it to pick up objects this time. We will then move on to look at creating parts of a 2D **Heads Up Display (HUD)** and controlling these, as well as the environment, through code. A HUD in video games varies between differing genres; in a racing game, for example, the HUD will be elements such as your speed, position, remaining laps, and score:



In a first-person shooter, your HUD is more likely to be made up of elements such as health, ammunition, and inventory items, like in this *Crysis 3* screenshot:



Crysis 3 Crytek 2013

As we have already set up an old man's hut with a door that can be opened and closed, we will now restrict player access to the inside of the hut by shutting the door, and let the player open it again only after he finds some specific items.

Inside the hut, the player will find the last piece of the puzzle in order to please the old man and finally have his hidden boat. By showing on-screen instructions when the player enters the hut door's trigger zone, we will inform them that the door requires the four missing pieces of the five to be able to enter the hut. We will then add a 2D HUD of an empty carved space for the artifact pieces on screen.

This will prompt the player to look for more artifact pieces that we will scatter nearby, so as to charge up enough power to open the door. By creating this simple puzzle, you will learn how to do the following things:

- Collecting objects with prefabs and triggers
- Learning how to manipulate game state based on tracked variables
- Working with the UI Image component to make a HUD

- Displaying on-screen text with the UI Text component
- Controlling game textures and lights using scripting
- Creating an inventory and controlling the HUD using an array

Creating the ancient artifact piece prefabs

In this section, we will take the ancient artifact model from the `Book Assets` folder that we imported previously, modify it in the **Scene** view, and turn it into a prefab. In this case, we will take the four children of the model to create four different prefabs; each of them with a trigger, a point light, and a script component to manage player collection.

Downloading, importing, and placing

To begin creating the puzzle, you'll need to locate the ancient artifact asset package within the `Book Assets` folder in the **Project** panel. You are provided with the following resources:

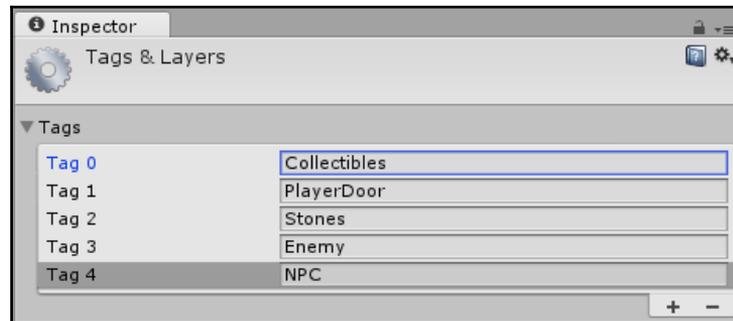
- An `artifactPiece` model in the `Models/static` folder.
- Five texture files for the HUD of the four artifact pieces filling a background, called `hud_quest`, in the `Book Assets/UI` folder.
- An audio clip to be played upon collection of a piece by the player. We will use the audio clip `spell13` in the `Assets/Chapter2-3-4/Sounds` folder.

Drag and drop the `artifactPiece` model from the `Models` folder into the **Project** panel and onto the **Scene** view. Then, hover your cursor over the **Scene** view and press *F* to focus the view on it. Your artifact piece may be placed at an arbitrary position; we will reposition it once we finish making our prefab. Obviously, if you have dragged `artifactPiece` into the scene and it is intersecting the floor, simply use the Translate tool (*W*) to move it to a position where you can see it well, remember that you can always press *F* to refocus the **Scene** view on your selected object.

Tagging the artifact piece

As we need to detect a collision with the `artifactPiece` object, we should give it a tag to help us identify the object in the scripting that we will write shortly. Objects can be identified by their name in the **Hierarchy**, but tagging items of a similar type can be helpful for game mechanics, such as collections. Click on the **Tag** drop-down menu and select **Add Tag** at the bottom of the menu.

The **Inspector** panel then switches to display the **Tag Manager**. If you are not shown the list of tags immediately, simply expand the area by clicking on the gray arrow to the left of the **Tags** title. In the next available element slot, add a tag called `Collectibles` if it is not already there (we created that in [Chapter 3, Creating and Setting Game Assets](#), and [Chapter 4, Player Controller and Further Scripting](#), on 2D development), as shown in the following screenshot:



Press *Enter* to confirm the tag name, then reselect the `artifactPiece` object in the **Hierarchy** panel, and choose the new `Collectibles` tag from the **Tag** drop-down menu at the top of the **Inspector** for that object.

Collider scaling and rotation

Now, we will prepare the `artifactPiece` as a prefab by applying components and settings that we'd like to feature on each instance of the `artifactPiece`.

Enlarging the artifact piece

We are creating something that the player is required to collect, so we should ensure that the object is of a reasonable size for the player to spot it in the game. As we have already looked at scaling objects in the **FBX Importer** for the asset, we'll take a look at simple resizing with the Transform component in the **Inspector**. With the `artifactPiece` object still selected in the **Hierarchy**, change all the **Scale** values in the Transform component of the **Inspector** to 1.6.

Adding a trigger collider

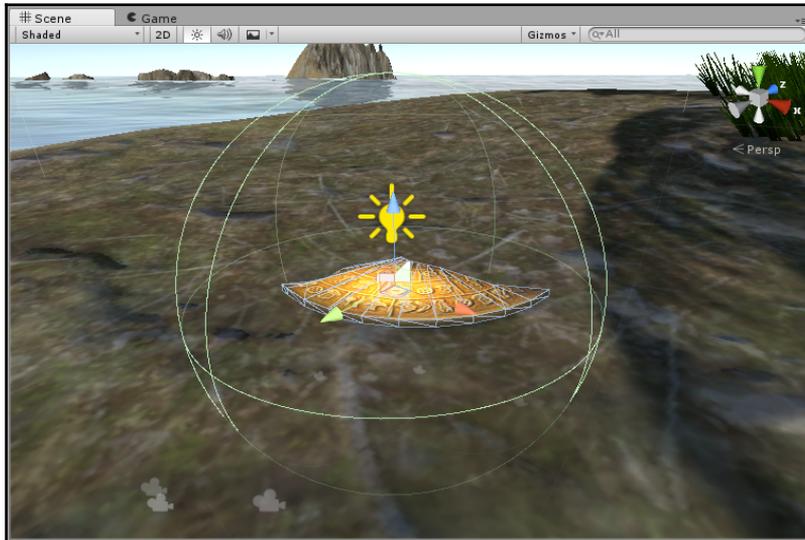
Next, we should add a primitive collider to the `artifactPiece` to enable the player to interact with it. Go to **Component | Physics | Sphere Collider**. We have selected this type of collider as it is of the closest shape to our `artifactPiece`. As we do not want the player to bump into this object while collecting it, we will set its collider to trigger mode. So, on the newly added Sphere Collider component, check the box to the right of the **Is Trigger** setting to enable it.

Collider scale and custom point light

Any primitive collider placed onto an object in Unity need not be of a particular size or position. Both of these properties can be adjusted by dragging the boundary dot handles in the **Scene** view after selecting the **Edit Collider** button in the Collider component on the **Inspector**. In the case of the Sphere Collider, due to its shape, there are **Radius** settings that will allow us to adjust the collider to better fit the artifact piece. In the field for **Radius**, type in the proper value. Make it a bit larger than our artifact piece so that it will allow the player to pick up this object more easily.

Now, we will create a point light and choose a warm yellow color. Set the intensity to 2.0.

A point light for each piece will be heavy for the GPU if we cast the light on the whole world. We will optimize the performance by changing the culling settings so that the light affects only the artifact piece: pick the mask setting of the light and select **Nothing**, then select **Collectible**. You will see that only the piece is getting light from the source:



A single piece of the artifact model will be the base for the prefab, complete of collider, trigger and a custom point light

Creating the artifact piece collection script

Now, we will write a script to handle the several actions we need to perform on our artifact piece during runtime:

- A rotation effect to make the artifact piece more noticeable to the player
- An `OnTriggerEnter()` method to detect the player collecting the piece, which sends a message to update an `Inventory` script attached to the player

On the **Project** panel, select the `Scripts` folder to ensure that the script we are about to create is created within that folder. Go to the **Create** button on the **Project** panel, and choose **C#**. Rename the newly created file from `NewBehaviourScript` to `ArtifactPiece` and double-click on its icon to launch it in the script editor.

Making it spin

Rotate the object so that its up vector (axis) is facing upward. We want to do this, to obtain a better view for the object, and to be able to spin it around its *y* axis correctly, as illustrated:



At the top of your new script, above the opening of the `Update()` method, create a floating-point public variable called `rotationSpeed`, and set its value to `100.0f`:

```
public float rotationSpeed = 100.0f;
```

We will use this variable to define how quickly the `artifactPiece` object rotates. As it is a public member variable, we will also be able to adjust this value in the **Inspector** once the script is attached to the `artifactPiece`. So, as a developer, or if you are working with an artist, they can change this value after the script is written. Within the `Update()` method, add the following command to rotate our `artifactPiece` around its *y* axis with a value equal to the `rotationSpeed` variable:

```
transform.Rotate(new Vector3(0, rotationSpeed * Time.deltaTime, 0));
```

The `Rotate()` command expects a `Vector3 (X, Y, Z)` value, and we provide values of 0 for X and Z, feeding the `rotationSpeed` variable's value into the `y` axis. As we have written this command within the `Update()` method, it will be executed in every frame, and the object will be rotated by 100 degrees in each frame. However, we also have `Time.deltaTime`, which means that the rotation will not be frame rate-specific, smoothing the motion to the way we need it to behave. In the script editor, go to **File | Save** in your favorite code editor and switch back to Unity.

To attach this script to our `artifactPiece` object, simply drag and drop the script from the **Project** panel onto the object's name in the **Hierarchy** panel, remembering to select the object to verify that it has been added in the **Inspector**.

Click on the Play button at the top of the interface and watch the `artifactPiece` in the **Scene** view (or Game view if your third-person camera is facing it) to ensure that the rotation works. If it does not work, then return to your script and check for any mistakes, double check that you have applied the script to the correct object. Remember to click on Play again to end your testing before continuing.

Adding trigger collision detection

Now, we will detect whether this object's trigger collider is intersected by the `Warrior_final_RM`'s collider. Return to your script, and beneath the closing right curly brace of the `Update()` method, add the following code:

```
void OnTriggerEnter(Collider col){
    if(col.gameObject.tag == "Player"){
        col.gameObject.SendMessage("PiecePickup");
        Destroy(gameObject);
    }
}
```

Here, we see the familiar `OnTriggerEnter()` method that we used on the trigger zone for our hut door in Chapter 7, *Interactions, Collisions and Pathfinding*. We are establishing the method and then using an `if` statement to check for a collided-with object tagged with the word `Player`. If this condition is met, we send a message to the collided-with object (our `Warrior_final_RM` hero player), calling a method we are yet to write, called `PiecePickup()`. We will write this into an `Inventory` script for the player in a moment.

Finally, the `artifactPiece` object is removed from the scene using the `Destroy()` command. Here, note that this command is always the last command carried out in order to ensure that the object is not destroyed before other commands are executed. Save your script and switch back to Unity. Click on Play to test your game and walk the `Warrior_final_RM` into the `artifactPiece` object. You should be shown the following error in the Console bar at the bottom-left of the Unity editor interface:



This is to be expected as we have not yet written a script that contains a method named `PiecePickup()`, so Unity warns us that although the code is correct, the sent message is not being received by any object. Now, we will save this object as a prefab before moving on to write an `Inventory` script so that the player object has a place to store the information each time a new piece is collected.

Saving as a prefab

Now that the `artifactPiece` object is complete, we'll need to clone the object three times, giving us a total of four artifact pieces. The best way to do this is with Unity's prefab system. As we already have a folder named `Prefabs` to store our prefabs in, let's store it there. Drag and drop the `artifactPiece` object from the **Hierarchy** to the `Prefabs` folder in the **Project** panel. This will save the object as a prefab, and it also means that the object in the **Hierarchy** is now an instance of that prefab. This means that any changes made to the prefab in the **Project** panel will be reflected in the instance in the scene. Objects in the scene that are instances of prefabs or models in the project are shown in the **Hierarchy** panel with blue text as opposed to the standard black text of scene-only objects.

Placing the artifact pieces

Now that we have our artifact piece object stored as a prefab, when we duplicate the object in the scene, we are creating further instances of the prefab. Ensure that you still have the `artifactPiece` selected in the **Hierarchy** and then duplicate the object three times so that you have four in total; this can be done either by going to **Edit | Duplicate**, using the keyboard shortcut `command + D` (on Mac) or `Ctrl + D` (on PC), or by right-clicking on the object in the **Hierarchy** and choosing **Duplicate** from the drop-down list.



When objects in the scene are duplicated, the duplicates are created at the same position. Don't let this confuse you; Unity simply does this to standardize where new objects in the scene end up, and this is because, when an object is duplicated, every setting is identical, including the position. Moreover, it is easier to remember that they are in the same position as the original and simply need to be moved from that position.

Now, select each of the four `artifactPiece` objects in the **Hierarchy** panel and use the Translate tool (*W*) to reposition them around the island. Remember that you can use the view gizmo in the top-right of the **Scene** view to switch from **Perspective** view to top, bottom, and side views. Ensure that you place the artifact pieces at a height at which the player can pick them up, so do not set them too high to reach or so low that it looks like the player's legs are collecting them!

In the following screenshot, you can see where we decide to place the four pieces, given that the last one is in a fixed place because it is closed up in the old man's hut. Once you have placed the three artifact pieces around the island, your output should look like the one shown in the following screenshot.

Note that, in the image, all four of the objects have been selected in the **Hierarchy** in order to help show their location:



As you can guess from the preceding image, we will place one in the old man's hut, one in the abandoned hut in the village, one under water in the lake we built in Chapter 6, *Creating the Environment*, and another two at your choice, where some guards are around, or in a lonely place.

Player inventory

In this section, we will establish a script for the player that stores information on what the player has collected in the game.

During the game that we are creating, the player will need to collect artifact pieces to power the door and a box of matches to light a campfire. The player is the best object upon which to store this information as it will allow other objects to query this `Inventory` script for information. For example, later, we will add a 3D model of a generator, which will display the current charge state of the door based on the information in this script.

Select the `Scripts` folder in the **Project** panel and click on the **Create** button, then choose the relevant scripting language you have been working in. Rename your new script `Inventory` and launch it in the script editor.

Saving the artifact collected piece count value

As we are establishing an inventory of collected items, we should make particular values of the inventory available to all scripts. The number of artifact pieces collected will be referred to as a variable, called `artifactPieceCount`, and to ensure that it is easily available to other scripts, we will make it a static variable, a global variable easily accessible by other scripts. Begin by establishing the following public static variable in your script; as usual, this means after the opening of the class declaration for C# users, and at the top of your script for JavaScript users:

```
public static int artifactPieceCount = 0;
```

This integer (whole number) variable will be set by the script itself, and ordinarily we would not make variables set by the script public as this would show them in the **Inspector**; however, static variables are not shown in the **Inspector**, so this is not a concern. The advantage of using a static variable here is that the information stored in them is considered global; that is, globally accessible, which means that in other scripts, we can refer to this value simply by stating, for example, the following:

```
if(Inventory.artifactPieceCount == 4){
```

This will allow us to check the `artifactPieceCount` value in our `DoorManager` script, as shown earlier, in order to deny entry to the hut until the user has collected the three missing pieces of the four to be able to find the last one inside the hut. We will add this to the `DoorManager` script once our basic `Inventory` script is complete.

Setting the variable start value

Ensure that when the scene loads, the charge is set to zero by setting it in a `Start()` method after your variables. When testing and reloading the same scene, variable values may be retained, and setting up a variable's default starting value in a `Start()` method such as this will avoid that issue:

```
void Start () {  
    artifactPieceCount = 0;  
}
```

Audio feedback

In addition to the `charge` variable, also add the following public variable as a reference to a sound effect to play when the player collects an artifact piece:

```
public AudioClip collectSound;
```

Adding the PiecePickup() method

Now that our inventory has an integer to store how many artifact pieces we have, we will write the `PiecePickup()` method that is called through `SendMessage()` in our `artifactPiece` script each time we pick up a new piece.

Add the following method to your script:

```
void PiecePickup(){
    AudioSource.PlayClipAtPoint(collectSound, transform.position);
    artifactPieceCount++;
}
```

This method, called `SendMessage()` in the `artifactPiece` script, will play our `collectSound` audio clip, and then add 1 to the value of `charge`. We are using `AudioSource.PlayClipAtPoint()` because there is no audio source on our player, and this command spawns a new temporary `GameObject` with an audio source on, plays the clip, and then removes itself. The second argument of `PlayClipAtPoint`, where we stated `transform.position`, is the location at which we create this temporary sound object, and we use `transform.position` (that is, the position of the player at that moment) as we simply need the sound to play near the player. This is also useful as it means that the sound will fade if the player continues to walk away from the location of collection. Save your script and switch back to Unity now.

Adding the inventory to the player

Attach the script you just wrote to the `Warrior_final_RM` object by dragging it from the `Scripts` folder to the **Project** panel onto the object in the **Hierarchy**.

Your `Warrior_final_RM` player object will now have an `Inventory` (script) component, and you should note that our public audio clip variable, `collectSound`, is shown in the **Inspector**. Expand the `Book Assets/Sounds` folder in the **Project** panel and drag and drop the `spell3` audio clip onto this variable to assign it.

Now, let's test our script! As we have not set our trigger zone to be dependent upon the charge value yet, all you will see so far is that when collecting artifact pieces, we no longer receive the `SendMessage` has no receiver error message, and you should also hear the artifact piece collection sound effect.

Press Play and ensure that this is working correctly; as always, press Play again to stop testing once you are done.

Restricting hidden piece spot access

Now that we have an inventory keeping track of the artifact pieces that we have collected, let's set up a game mechanic that forces the player to collect all the four artifact pieces before they are granted access to the hut. We will begin by achieving this in code, then add a visual indicator for the player.

Restricting dialog access with a piece counter

In this section, we will write code into our inventory that checks whether the player has enough artifact pieces to open the door. We currently have a trigger zone in charge of opening the door, so we will amend this to query the value of the charge in our new inventory whenever the player enters the trigger. First, let's ensure that our door only opens when the player has collected four artifact pieces. Locate the `DoorManager` script in the **Project** panel and double-click on it to open it in your chosen script editor. Then, in the `OnTriggerEnter()` method, add the following if/else statement inside the existing if statement that checks the current count in the Inventory so that your code matches the following:

```
void OnTriggerEnter(Collider col)
{
    if (col.gameObject.tag == "Player")
    {
        PlayerInventory inventory =
            col.gameObject.GetComponentInParent<PlayerInventory>();
        if (inventory.artifactPieceCount == 3)
        {
            Door(doorOpenSound, true);
        }
    }
}
```

Here, we are ensuring that our static `artifactPieceCount` variable in the `Inventory` class is equal to three, if it is, we call the same `DoorCheck()` as before; if not, we prepare an else statement that we will use later to warn the player that they do not have enough pieces collected to unlock the door. We will add further code to this else statement later, once we have set up the visual HUD and UI Text objects we need to display such warnings and hints for the player. Save your script now and return to Unity so that we can test this restriction. In Unity, press Play and ensure that the door does not open until you have collected all the four artifact pieces.

Displaying the game progression status HUD

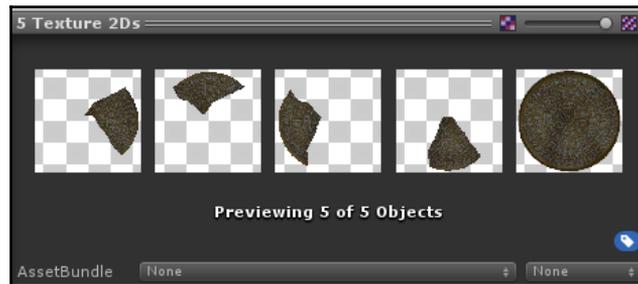
Now that we have our collectible pieces and inventory in place, we'll need to show the player a visual representation of what they have collected. The textures imported with the `Book Assets` have been designed to clearly show that the player will need to collect four artifact pieces to open the door and with the last piece inside, complete the artifact and finally end the game by escaping the island. These are the six stages we want to setup for the HUD indicator: an empty background one, and the five pieces, finally composing the whole:



Note that the last picture looks like it is not broken anymore, differently from the 4th image, this was a design choice, to show the artifact as a whole, not broken piece in the end. Using a series of images starting with an empty background image and, at the end, the full artifact, we can create the illusion of a dynamic interface element.

The `Book Assets/Textures` folder contains the image files that we need for this UI Texture-based HUD, one of an empty circular shape and the others of the five stages of the quest. Created with `The Gimp`, an alternative to `Photoshop`, these images have a transparent background and are saved in a `PNG` format.

The PNG format was selected because it is compressed but supports high-quality **alpha channels**. An alpha channel is what many pieces of software refer to as the channel of the image (besides the usual red, green, and blue channels) that defines transparency in the image:



The collection of 5 textures of the artifact pieces used over the background to create the indicator

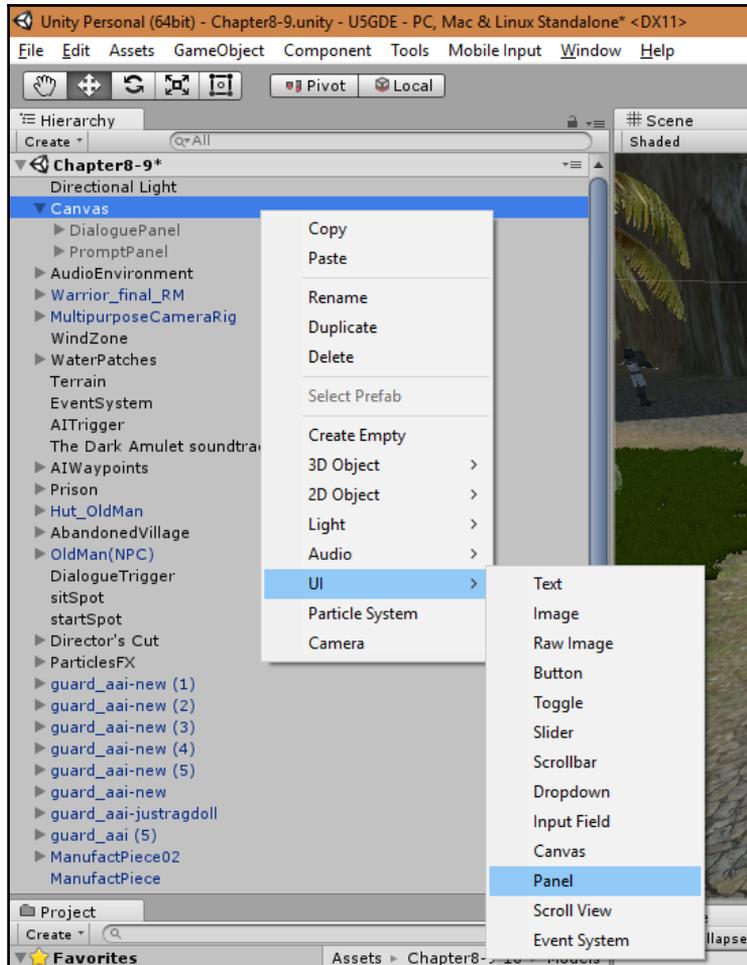
We will begin to create the UI showing the progress of the quest by adding the empty artifact graphic to the scene using Unity's UI Image component. In the **Project** panel, expand the UI folder within the `Book Assets` folder and select the file named `hud_background`. In a differing approach to our normal method of dragging and dropping assets into the **Scene** view, we need to specifically create a new object with a UI Image component and specify the `hud_background` texture for it.

Import settings for UI images

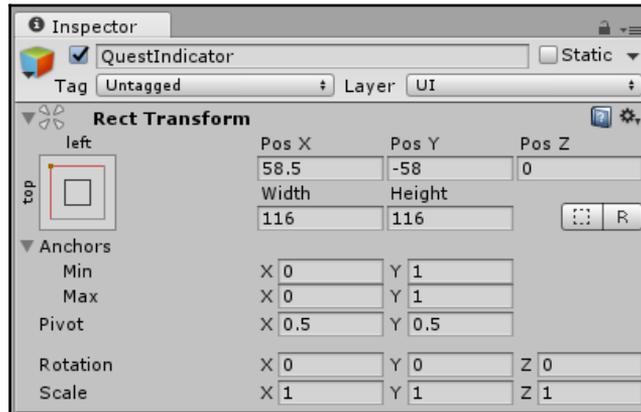
Before we set up this texture file as a UI Texture in our scene, we need to tell Unity how to interpret it. With the `hud_background` texture selected in the **Project** panel, take a look at the import settings for this texture in the **Inspector** under the **Texture Importer** heading. Begin by setting the **Texture Type** to **Sprite/UI** instead of **Texture**. This will force Unity to display the texture at its original aspect when using it for a UI element.

Creating the HUD panel and background UI image

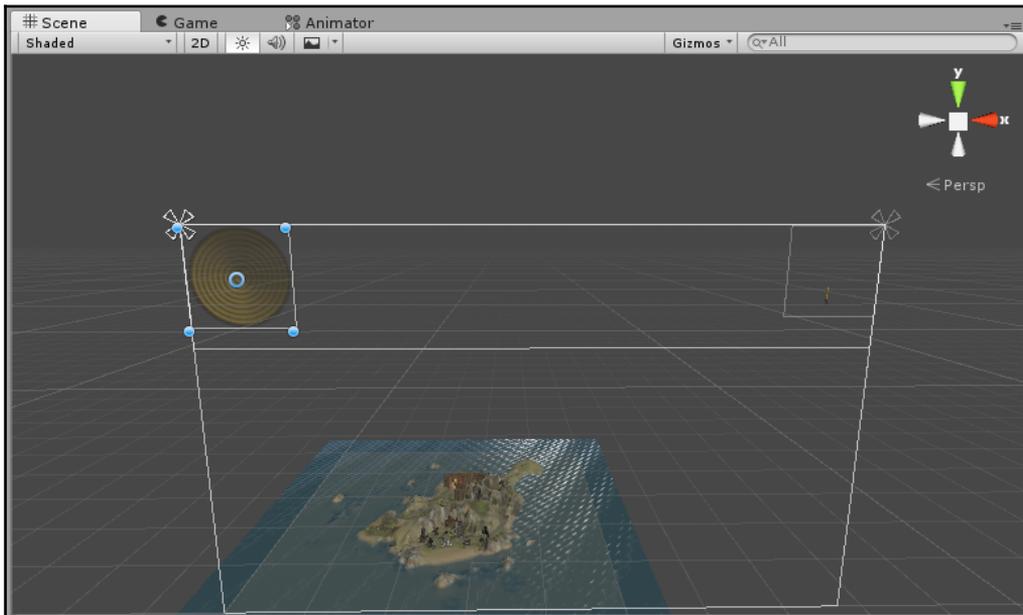
We will create the main parent object as a UI image containing our background image holder, select the Canvas GameObject, right-click on it in the **Hierarchy**, and choose **UI | Panel**, as shown in the following screenshot:



Then, selecting the HUD panel GameObject, create a **UI | Image** with the same procedure. Note that this will be created under the HUD GameObject in the **Hierarchy** as its child. Rename it `QuestIndicator` by pressing *Return* (Mac) or *F2* (PC):



Give **116 x 116** for width and height, as shown in the preceding screenshot, and position it manually with the 2D tool in the top-left corner of the panel, as in the following screenshot:

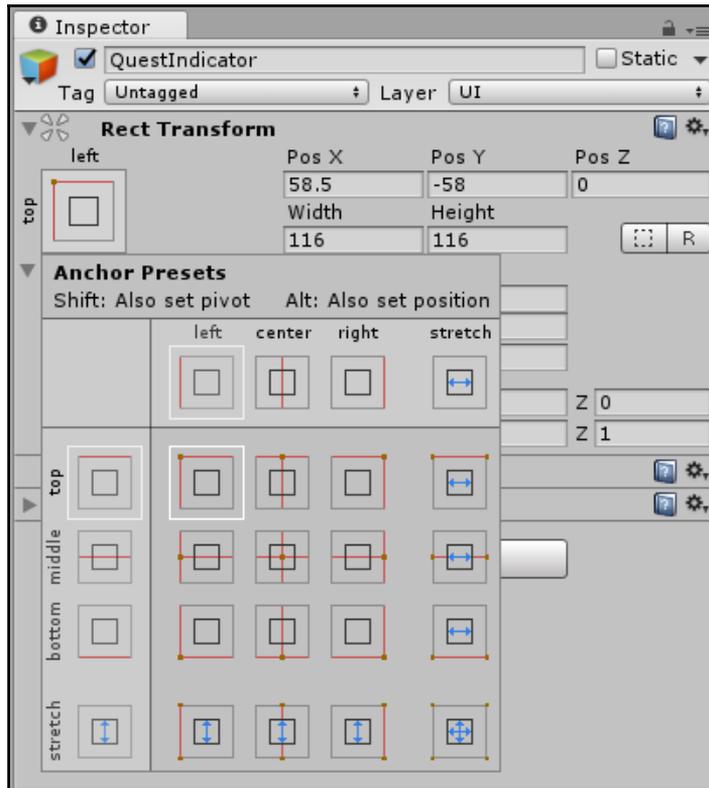


When dealing with UI elements, you will usually work in the scene view and focus on the Canvas GameObject with the *F* key. You will select the `QuestIndicator` object in the **Hierarchy** and the 2D tool in the toolbar:

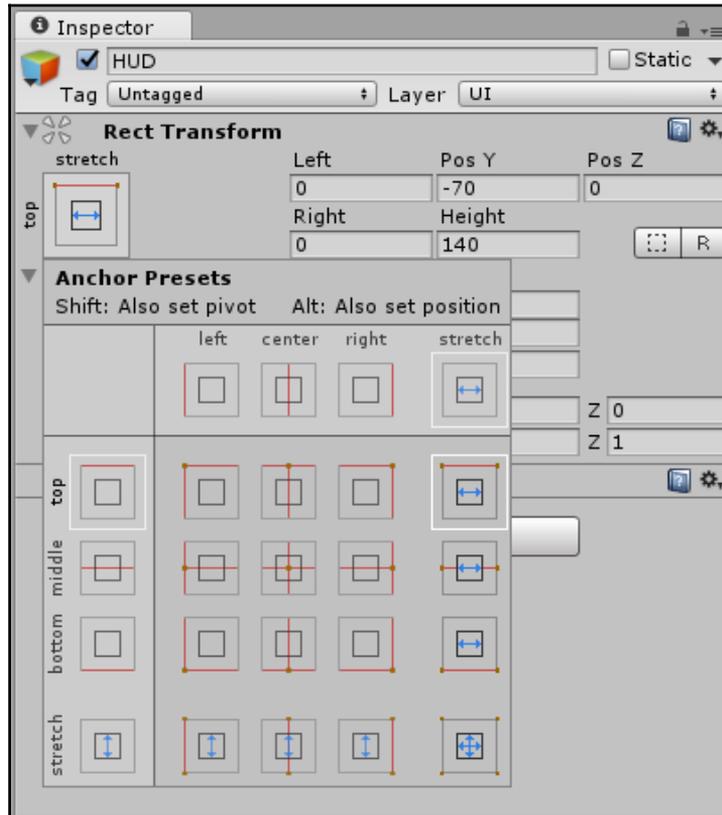


The 2D Tool in the main Editor toolbar

Then, move the object to the top-left corner of the canvas, aligning it with the top-left part of the HUD panel by dragging it. If it isn't automatically docked with the magnetic feature, you can choose the alignment/stretching/docking to top-left, as shown:



This ensures that the containing object with the background image is always the same size and that it's always aligned at the top-left corner. For the HUD panel container, we will set a *stretch horizontally* setting instead, as illustrated here:



After you set those bits you should now see the image displayed on the **Game** view, as shown in the following screenshot:



Scripting for UI image activation

Now that we have several artifact pieces to collect and our inventory is in place, we can use the `Inventory` script itself to control the status of the UI HUD panel children we just added. To control, we can use an array in scripting.

Understanding arrays

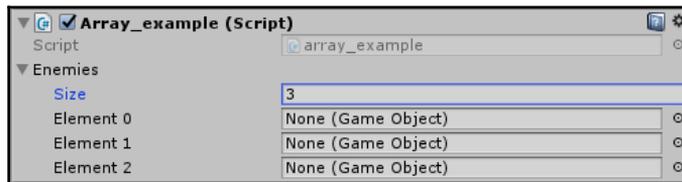
Arrays are data containers that can contain many values. Try to think of them as variables that contain many values or entries. The values in an array are organized by use of an index, a number of their entry into the array, much like the ID of an entry in a database table. The number of items stored in an array is referred to as the array length. Basic arrays can be resized in the **Inspector** using the `Size` parameter (we will use this shortly); in code terms, they can be resized through the `Array.Resize(ref element, 15)` method residing in the `System` namespace. Declaring an array is similar to the declaration of a variable: it needs a name and a type. This is the same procedure as when declaring a variable but in addition to the type, we also use square brackets to define it as an array.

For example, if we wished to make an array of enemy `GameObjects`, we might write the following code snippet:

```
public GameObject[] enemies;
```

We can then set the size in the **Inspector** and assign objects to the array.

The following screenshot shows what it will look like in the **Inspector** having filled in a **Size** (the length of the array) value of 3:



Drag and drop from the **Hierarchy** or **Project** panel can then be used to assign objects to these positions, but we can also use scripting to perform this task. Objects can be assigned to the array by stating a particular position within the array.

For example, if we wish to add a `GameObject` to the third position in the array, we would write:

```
enemies[2] = gameObject;
```

In this example, `gameObject` will ideally be a reference to a `GameObject` somewhere in the scene.

You should note that array indexes always begin at zero, so the third position is actually index number two in the preceding example. If you attempt to add an index that does not exist to an array, Unity will give an error stating that the index is out of range, which means that it is not a number within the current length of the array.

Adding QuestIndicator images

To create the five `QuestComplete` UI game objects, we will proceed by creating the first child of `QuestIndicator` of the same size as that of the parent. We do so for simplicity and quick development because the furnished images are single pieces of the artifact in a square, which is large like the full object; this is not the perfect way to proceed in terms of video memory usage, and could be optimized, especially for the mobile platform.

Ideally, the artifact pieces would be cut smaller and positioned discreetly into the parent frame. Keeping the `QuestIndicator` object selected, right-click on it in the hierarchy, and from the menu select **GameObject | UI | Image**. Rename this new created object `QuestCompleteFull`. Assign the full artifact sprite into the sprite slot of its `Image` component.

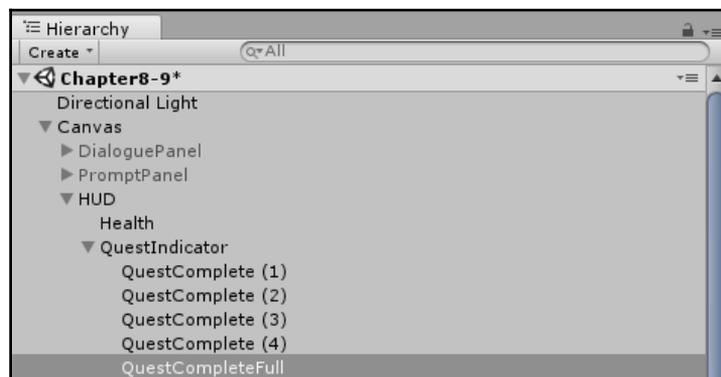
You will now see the whole artifact covering the background area. Now, clone this object four times and rename them `QuestComplete (1)`, `(2)`, `(3)` and `(4)`. Assign the four artifact piece sprites to the four corresponding component slots.

Finally, make inactive all the `QuestIndicator` children `GameObjects` from the **Inspector**.

Draw order of elements

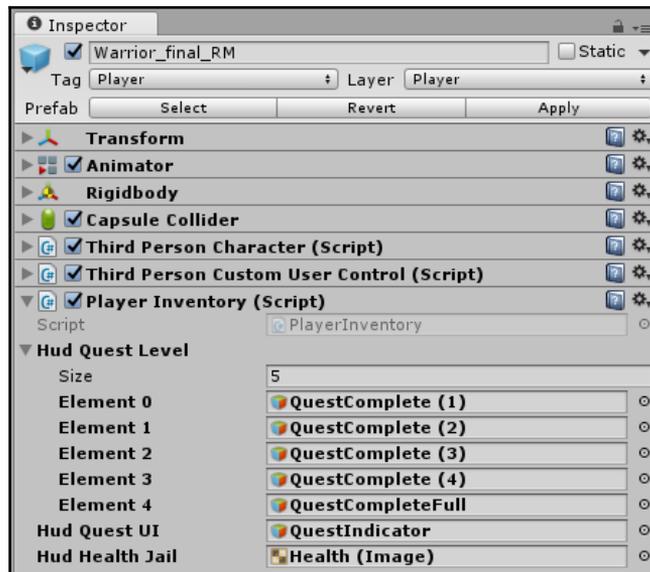
UI elements in the Canvas are drawn in the same order as they appear in the **Hierarchy**. The first child is drawn first, the second child next, and so on. To change the draw order of the elements, you simply have to reorder them by dragging them into place.

We will need to do this so that the final picture with the entire artifact stays under the other pictures. In the next screenshot, you can see how the `QuestComplete` series of UI `GameObjects` were ordered to obtain the desired effect:



In the `Book Assets/Chapter7-9-10/UI` folder, you will find textures with names beginning with `hudQuestLevel`.

After importing them as **Sprites/UI**, drag them to become children of the HUD panel in the UI Canvas, as shown in the preceding screenshot. Next, assign the `QuestIndicator` children `GameObject`s in the **Hierarchy** to the `hudQuestLevel` array variable by dragging one by one into them. The final result should look like the following image:



By creating an array of five UI Image objects, four pieces of artifact, plus the entire artifact that will be drawn over an *empty* `hud_background` texture, we can activate the relevant object on the HUD to a particular one in our project, depending on the current value of our `artifactPieceCount` variable. This means that whenever we pick up an artifact piece and increment `artifactPieceCount`, our HUD will automatically update!

Open the `Inventory` script now and place your cursor on a new line after the existing `collectSound` variable.

We will use an array for this task and a separate array to store the textures for our generator; so, to keep these elements of our script separate, we'll put in comments to accompany them. Add the following code to your script:

```
// HUD
public GameObject[] hudQuestLevel;
private int artifactPieceCount;
```

As the **Size** value defaults to 0, no elements exist in the array. Set the **Size** value to 5 in the **Inspector**, and you will see Element 0 to Element 4 listed with their `GameObject` type alongside them.

Now that our array is set up, let's return to the code and use the value of charge to choose a texture from the array for the `QuestIndicator`. Inside the `Inventory` script, add the following lines to the existing `PiecePickup()` method:

```
++artifactPieceCount;
for (int i = 0; i < hudQuestLevel.Length; i++)
{
    if( i<artifactPieceCount )
        hudQuestLevel[i].SetActive(true);
    else
        hudQuestLevel[i].SetActive(false);
}
```

Here, we are addressing the `Image` object (our `QuestIndicator`) that is assigned to the `hudQuestUI` variable. We are setting this to a particular image in the `hudQuestLevel` array. The particular index, and therefore the particular image object, is chosen by feeding in the value of the charge variable where we would ordinarily use a number, cool, huh?

Now, save your script and return to Unity.

Another approach would have been to switch the `Sprite` reference in the `Image` component and use only one `UI Image` object. This would mean work again on the texture source images to obtain all the artifact pieces adding to the empty one in a sprite sheet, but this would not allow a specific image to be displayed according to the collected piece, and for our design this would not be a valid option.

Let's try it out! Press Play now and start collecting artifact pieces. You will see that your `QuestIndicator` HUD now switches on one by one each of the textures after you collect each artifact piece. Press Play again to stop testing.

Disabling the HUD for game start

Now that we have created the HUD, and have its increment based upon the `Inventory`, we should consider that when the game begins, we should not show the player the artifact piece HUD. This is too much of a clue as to what they need to do. Instead, we should only display the HUD once the player attempts to enter the hut.

We should do this at the following two points during gameplay:

- If the player walks into the trigger zone without having picked up anything
- If the player picks up an artifact piece without having talked to the old man

We will work on the former situation first, where the player enters the trigger with no artifact pieces. This is where the else statement of the `DoorManager` script comes into play. Open the `DoorManager` script for editing.

When the player first enters the trigger but cannot open the door, they should be made aware that the door will not open without power. For this, we will play a sound clip to show that the door is currently locked without the generator to open it, and then enable the **QuestIndicator** to show that there is an empty artifact piece to fill.

Above the opening of the `OnTriggerEnter()` method in your script, add a public variable for the door locked sound to be assigned to later:

```
public AudioClip lockedSound;
```

Next, place your cursor inside the else statement within the `OnTriggerEnter()` method that we created earlier and add in the following lines:

```
transform.Find("door").audio.PlayOneShot(lockedSound);  
col.gameObject.SendMessage("HUDon");
```

Here, we are making use of two other objects, the door child object of the hut parent object and the `Player` object (third-person controller) as this is the object stored in the `col.gameObject` reference. We make reference to the door using `FindChild()` as before, and then use its `Audio Source` component to play our door locked sound. We then use `SendMessage()` to call a new method in the player **Inventory** called `HUDon()`. This new method will simply check whether the `QuestIndicator` panel is enabled, and if not, it will enable it by re-enabling the component. We are placing this method into the `Player Inventory` script because we already have a reference to the `QuestIndicator` within that script. Now, save your script and reopen the `Player Inventory` script, or switch to the tab it is open in within the script editor.

Enabling the HUD

Within the Inventory, we should enable the HUD during runtime both when the player picks up their first artifact piece (if they have not tried to access the door) and when the player tries to open the door without any artifact pieces. We will deal with the latter first as we have just created a call to a new method that we are calling from our `DoorManager` script.

After the closing curly brace of the `PiecePickup()` method, add the following:

```
void HUDon() {  
    if (!hudQuestUI.activeSelf)  
    {  
        hudQuestUI.SetActive(true);  
    }  
}
```

This simply uses the check on the `QuestIndicator` `GameObject` that we have already set up in the form of the `hudQuestUI` variable. The `if` statement here simply checks whether it is not enabled (or rather, whether the `activeSelf` property of this `GameObject` is *false*), and, in that case, sets its value to `true`. Now, let's take care of enabling the HUD when the player first picks up an artifact piece.

We know this is dealt with by our `PiecePickup()` method, so we will simply call the new `HUDon()` method rather than writing the same `if` statement again. Add the following method call to the beginning of your `PiecePickup()` method:

```
HUDon();
```

By doing this, the same check is performed when the player first picks up an artifact piece. Save your script now and switch back to Unity so that we can test the toggling of the HUD in both of the aforementioned circumstances.

Before testing, select the `outPost` parent object and assign the `door_locked` audio clip in the `Book Assets/Sounds` folder in the **Project** panel to the `Locked Sound` public variable in the `Trigger Zone` (script) component.

Press Play to start testing, and ensure that if you enter the trigger zone with no artifact pieces, the HUD is enabled and the locked door sound is played. Then, restart testing (switch off Play and press it once more) and this time, pick up an artifact piece without entering the trigger zone to ensure that the HUD is also enabled.

Save your script now and switch back to Unity. Select the first-person controller to see the Inventory (script) component in the **Inspector**. Press Play again to stop testing, and then save your progress in Unity by choosing **File | Save Scene** from the top menu.

Next, we will move on to give an additional hint to the player in the form of on-screen text. We will do this in the style of classic adventure games using inner monologue, with which the player character effectively speaks to the player.

Hints for the player

What if the player, less intrigued by the artifact pieces than the hut door itself, goes up to the door and tries to enter? It may seem a little cold to simply switch on the closed door itself, whereas having the player character speak to the player will be a much friendlier approach to the gameplay experience. Or even better, let's have the **non-playing character (NPC)** talk to him and give hints on what to do next, as we saw in the previous chapter.

At this stage, it is also important to think about phrasing - while we can easily say *Collect some more artifact pieces!*, it is much better in gameplay terms to provide hints, using inner monologue, by having the player character's thoughts relayed to the player, for example: *The old man won't open his hut until I find the first three pieces.*

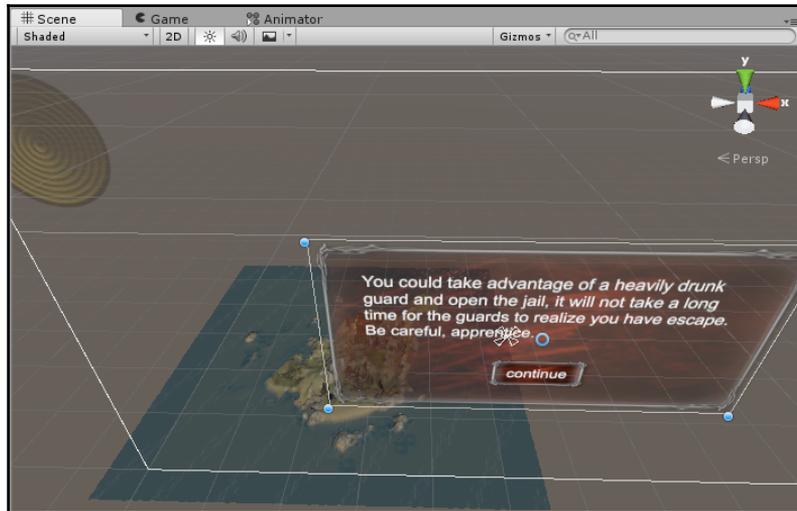
To show the text on screen, we will use Unity's UI Text component. There are various ways of displaying text on screen, but this is the simplest. We will go on to cover other methods, such as the Unity UI scripting class, later in this book.

Before creating the dynamic UI Text object, we will create another panel to contain it and an **Ok** button; but in this case, we will use a background for the panel.

Writing on screen with UI Text

Whenever you need to write text on the screen in 2D, the most straightforward way to do this is with a UI Text component, but it can also be done using a UI scripting class. By creating a new UI Text object from the top menu or by right-clicking in the **Hierarchy**, you will get a new object with both Transform and Text components. Create one of these now by choosing **UI | Text** from the **Hierarchy Create** button or by choosing **GameObject | Create Other | UI | Text** from the top menu. You should now have a new object in the **Hierarchy**, called **UI Text**, and some 2D text on the screen that we will fill for the start of the game with this phrase: *You could take advantage of a heavily drunk guard and open the jail, it will not take a long time for the guards to realize you have escaped. Be careful, apprentice.*

It will be displayed at the beginning, as illustrated in the following screenshot:



Rename this object in the **Hierarchy** by selecting it and pressing *Return* (Mac) or *F2* (PC). Name it `TextHintUI`.

In addition to the positioning of the entire element, UI Text also features an Alignment parameter that works in a similar manner to justification settings in word processing software. Click on the up/down arrows to the right of the Anchor parameter and choose middle center, then set the alignment to middle, this means that text will spread out from the center of the screen rather than starting in the center and then filling out to the right. While you can easily type what you wish this UI Text element to say into the Text property in the **Inspector**, we will instead control what it states dynamically using scripting. When you want to work with UI elements in your classes, you will add the following .NET library:

```
Using UnityEngine.UI;
```

Scripting for UI Text control

Our first hint about the door needing more pieces should be displayed when the player attempts entry without enough artifact pieces collected. For this reason, it makes sense to place our code into the `DoorManager` script and call a method within a script on the `TextHintUI` object we just created.

Open the `DoorManager` script now and, first of all, add the library to manage UI elements after the last `" "` directive, then add the following reference to the top of your script beneath the `lockedSound` variable:

```
public Text textHints;
```

This public variable of type `Text` will allow us to drag and drop our `TextHintUI` `GameObject` onto this public variable to refer to the UI `Text` component attached to it directly from this class. We used this data type in order to save storing a reference to the entire `GameObject`.

Now, to make use of this reference, we will send this object a message. Add the following line to your `else` statement within the `OnTriggerEnter()` method:

```
textHints.SendMessage("ShowHint", "The old man won't open his hut until I  
find at least the first three pieces");
```

Here, we are using `SendMessage()` yet again to call upon a method within our `TextHintUI` object. This time, we are calling a method with an argument; this is shown by the fact that not only the name of the method, `ShowHint()`, is within the `SendMessage()` parentheses, but there is also additional information, separated by a comma. This additional information is an argument of the method we are calling. Declared as a string data type, we will use this in order to set what our `TextHintUI` writes on the screen. Save your script now and return to Unity, and we will create the script that contains this method.

Before we write the receiver of this message, let's assign our new public reference to our UI `Text`, select the `hut` object in the **Hierarchy** to view the `Door Manager` component. Then, drag and drop the `TextHintUI` object from the **Hierarchy** to the `Text Hints` variable. Now, let's create the receiver of this message, a script to attach to the `TextHintUI` object. Create a new script file by first selecting the `TextHintUI` object in the **Hierarchy**; then, from the **Inspector**, click on **Add Component**, select your chosen script language as usual, rename the `NewBehaviourScript` in `TextHints`, and double-click on its icon to edit it. We will begin by establishing the `ShowHint()` method that our `DoorManager` script is calling. Add the following method to your new script:

```
void ShowHint(string message)  
{  
    GetComponent<Text>().text = message;  
    if (!this.gameObject.activeSelf) {this.gameObject.SetActive(true);}  
}
```

Here, we have added a method with a single argument (message) to receive the sentence from the `SendMessage()` method. We begin by setting the text parameter of this object's Text component to the sentence string received by our argument.

We then perform a similar task to the one we did for the `QuestIndicator`, checking whether the `GameObject` is active, and if not, activating it.

So, let's try this out. Save your script and return to Unity. Assign your new `TextHints` script to the `TextHintUI` object in the **Hierarchy** using drag and drop from the **Project** panel, and then remove the default words, UI Text, in the text parameter of the UI Text component for that object so that nothing is written on the **Game** view.

Now, press Play at the top of the interface and you will note that when you enter the trigger zone of the hut, text appears on the screen with the sentence that we specified in the `SendMessage()` call to our `TextHints` script.

This is all well and good, but hey, we need that text to go away at some point, right? Press Play to stop testing and return to your `TextHints` script.

Add the following variable to the top of your script:

```
float timer = 0.0f;
```

This variable is exactly what it sounds like, a timer. We will use the `Update()` method to increment this timer as soon as the UI Text component is enabled. When the timer reaches a certain value, we will disable it again to stop the text from obscuring the player's view.

Add the following code to your `Update()` method:

```
if(this.gameObject.activeSelf){
    timer += Time.deltaTime;
    if(timer >=4){
        this.gameObject.SetActive(false);
        timer = 0.0f;
    }
}
```

Here, we use the `TextHintUI` `GameObject` `isActive` in order to check whether the `GameObject` has been activated by our `ShowHint()` method.

If the `TextHintUI` object is not active, we increment our timer variable by adding the value of `Time.deltaTime` to it, a property that counts time in a manner that is not frame-rate-specific. After the timer increment, we have added a nested `if` statement that checks for our timer reaching four seconds. If it does, we disable the component by setting its `enabled` property to `false` and reset our timer to `0.0` so that it can start counting again the next time we use a hint. Save your script and return to Unity now. Play your game and you will note that when you enter the trigger zone of the hut, the following sentence appears on the screen: **This door seems locked... Maybe if I find those pieces that he wants...**

Now, thanks to our timer, the sentence should disappear after four seconds. Press Play again to stop testing. As usual, if you are receiving error messages at the bottom of the Unity interface, double-click on them and return to your script to fix them, checking it against the code in the book.

Now, save your scene in Unity by choosing **File | Save Scene** from the top menu.

Another way of creating this is using a UI button to pilot the dismissal of the Hints text panel. We can also use the timer and the button together so that the panel will hide after a certain time, if the user didn't dismiss it with the button.

This is what we will do in [Chapter 12, *Designing Menus with Unity UI*](#), to learn how to set up Unity UI events and how to manage doing things without coding.

Adjusting hints to show progress

Now that we have a hint shown on screen, we should ensure that the player knows that if they have started collecting artifact pieces, they are doing what the game requires of them. To do this, we should display a different message if they have begun to collect artifact pieces but do not have all four yet. Return to edit the `DoorManager` script.

Place the following `else if` statement into the existing `if/else` statement within the `OnTriggerEnter()` method, remembering that any `else if` statement must be placed before the `else` statement, which must be last:

```
else if (inventory.artifactPieceCount > 0 && inventory.artifactPieceCount < 4)
{
    TextHintsUI.SendMessage("ShowHint", "This door won't budge - collect more artifact pieces.");
    transform.parent.Find("DoorJoint").Find("DOOR").GetComponent<AudioSource>().PlayOneShot(lockedSound);
}
```

In the preceding code, we are checking there are more than zero pieces, but less than four. If these conditions are met, we are using `SendMessage()` to call the `ShowHint()` method on the `TextHintGUI` object, and playing the locked sound again as audio feedback.

`SendMessage()` works well for a quick prototyping, but it's better to use other design patterns because the object you broadcast your message to could change or be renamed or deleted, and this can lead to a lot of potential issues when working with a complex project. In this case, for simplicity, we will use it to broadcast the `ShowHint` command, a method in a script attached to the UI object. Your `OnTriggerEnter` method override should now look like this:

```
// Check collision with player's capsule collider
void OnTriggerEnter(Collider col)
{
    if (col.gameObject.tag == "Player" && doorIsOpen == false)
    {
        PlayerInventory inventory =
            col.gameObject.GetComponentInParent<PlayerInventory>();

        if (inventory.artifactPieceCount == 3) // 8 is the layer id
            for player
            {
                Door(doorOpenSound, true);
            }
        else if (inventory.artifactPieceCount > 0 &&
            inventory.artifactPieceCount < 4)
        {
            TextHintsUI.SendMessage("ShowHint", "This door won't
                budge - maybe more artifact pieces will help...");
            transform.parent.Find("DoorJoint").Find("DOOR").GetComponent<AudioSource>()
                .PlayOneShot(lockedSound);
        }
        else
        {
            transform.parent.Find("DoorJoint").Find("DOOR").GetComponent<AudioSource>()
                .PlayOneShot(lockedSound);
        }
    }
}
```

This gives variation to the feedback that the player is given and should help them feel that they are progressing, that the game is responding to their actions. Save your script now and return to Unity. Play test your game and pick up a single artifact piece, then try to enter the door. You will be greeted with the sentence we just added!

Go to **File** | **Save** in Unity to update your progress.

Summary

In this chapter, we successfully created and solved a game scenario. By assessing what your player will expect to see in the game you present to them, outside of your prior knowledge of its workings, you can best devise the approach you must take as a developer.

Try to consider each new element in your game from the player's perspective, play the existing games, think about real-world scenarios, and most of all, assume no prior knowledge from the player, even of the existing game traditions, as they may be new to gaming. The most intuitive gameplay is always found in games that strike a balance between the difficulties in achieving the tasks set and properly equipping the player for the task in terms of information and familiarity with the intended approach. Appropriate feedback for the player is crucial here, be it visual or audio-based, always consider what feedback the player has at all times when designing any game.

Now that we have explored a basic game scenario and looked at how we can build and control GUI elements, in the next chapter, we'll move on to solve another game scenario, knock down all the guards in order to get the artifact pieces. Let's dive into Rigidbody and Instances.

10

Instantiation and Rigidbodies

In this chapter, we'll expand upon the two crucial concepts in 3D game design that we looked at in [Chapter 2, *Prototyping and Scripting Basics*](#). We will take the abstracted game mechanic of aiming and throwing objects, and put it into the context of our island game by creating the ability to throw stones at the guards to distract them for a given time. When you begin building game scenes, you'll realize that not all the objects required within any given scene will be present at the beginning of the game. This is true for a wide variety of game genres.

Take our island exploration game as another example. In this chapter, we'll be taking a look at instances and physics by creating a simple stone launcher script but, as with the prototype we made earlier, the projectiles that will be thrown will not be present in the game scene when it begins. This is where prefab instantiation comes into play again. By specifying a `GameObject` stored as a prefab, as well as a position and rotation, objects can be created while the game is being played. This will allow us to create a new stone whenever we reach the correct frame of animation, after the user hits the right mouse button.

In order to tie this new part of our game to the game as it stands, we'll be placing three of the four pieces of a broken artifact hidden at specific spots on the island, and the fourth of them placed in the old man's hut. We will give the player a chance to find the lost artifact pieces across the island; the old man requires you to find them by stunning the guards and exploring the surroundings. The final piece of the artifact will require the hero to enter the old man's hut to find the last piece and finally solve the quest.

As part of this book's assets, you are provided with a lot of models, wooden boxes, barrels, and chests in particular, together with the stones to launch. You will need to create a prefab of a stone and control the animation of the player to implement a correct hand launch through scripting and animation events, detecting collisions between the stones and the moving targets, the guards.

We will be linking two separate game mechanics: the *piece collecting* game and the *avoiding or knocking out guards* game that will enable roaming the island to find the pieces without being put in a prison cell again... or worse.

In this chapter, you will learn about the following things:

- Using Rigidbodies and Prefabs in combination with the Instantiate command
- Sequencing hero animations with stone launch and how to spawn and impress force to the stone that has been launched
- Using a time-based script to destroy the instantiated stones after a given time
- Setting up a ragdoll simulation for the guards Prefab and scripting a Ragdoll Manager
- Triggering animations / AI states / ragdoll simulation as a result of collisions

Implementing instantiation

In this section, we will use the `Instantiate()` method that we saw in our earlier prototype again. This is a concept that is used in many games to create projectiles, collectible objects, and everything needed for the game to run smoothly without big hiccups. We will go deeper and further into game optimization later in the book.

Instantiation is simply a method of creating (also referred to as spawning) objects from a template (a prefab in Unity terms) during runtime, as opposed to those objects already present in the scene when it loads. This is basically what happens when you drag a prefab from the **Project** view to the **Scene** view or to the **Hierarchy**.

The approach when using instantiation will usually take this form:

1. Create the `GameObject` (that you wish to instantiate in your scene) manually, adding components as necessary.
2. Save this newly created `GameObject` as a prefab.
3. Delete the original object from the scene so that it is only stored as a prefab asset.
4. Write a script that involves the `Instantiate()` method, attach it to an active `GameObject` in the scene, and set the prefab you created as the object that `Instantiate()` creates using a public variable to assign a prefab to.

The prefab instance of our object, in this case, a stone, must be instantiated at a particular position within the 3D world and, when assigning the position of an object to be instantiated, you must consider where your object will be created and whether this position can be inherited from an existing object.

Consider these prototypes of the `Instantiate` method:

```
public static Object Instantiate(Objectoriginal);

public static function Instantiate(original: Object, parent: Transform):
Object;

public static Object Instantiate(Objectoriginal, Transformparent);

public static function Instantiate(original: Object, parent: Transform,
instantiateInWorldSpace: bool): Object;

public static Object Instantiate(Objectoriginal, Transformparent, bool
instantiateInWorldSpace);

public static function Instantiate(original: Object, position: Vector3,
rotation: Quaternion): Object;

public static Object Instantiate(Objectoriginal, Vector3position,
Quaternionrotation);

public static function Instantiate(original: Object, position: Vector3,
rotation: Quaternion, parent: Transform): Object;

public static Object Instantiate(Objectoriginal, Vector3position,
Quaternionrotation, Transformparent);
```

The more complete method clones the original and spawns it at position with a given rotation, parented with the parent object's Transform. There are many different combinations and you can use any of them to get what you need.

The `instantiateInWorldSpace` parameter explained on the official Unity manual:

Pass true when assigning a parent Object to maintain the world position of the Object, instead of setting its position relative to the new parent. Pass false to set the Object's position relative to its new parent.



Since the release of Unity 5.5, the way instantiated object inherits the transform position and rotation has changed in some function prototypes. Be careful with the earlier way of using it, as it might result in different behavior than expected if you were used to it with previous releases of Unity.

Check the official Unity manual for detailed usage, at <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>.

For example, when creating our stone prefabs, we'll be creating them at a given point in the world space defined by an empty `GameObject`, which will be a child of our player character's right hand `GameObject`. As a result, we can say that it will be created in local space - not in the same place every time, but relative to where our player character is standing and facing, because it is the camera that defines where our character is effectively looking. This decision helps us decide where to write our code, that is, which object to attach a script to. By attaching a script to the empty object that represents the position where the stones must be created, we can simply use dot syntax and reference `transform.position` as the position for the `Instantiate()` method. By doing this, the object created inherits the position of the empty object's `Transform` component because this is what the script is attached to. This can be done for rotation as well, giving the newly spawned object a rotation that matches the empty parent object. This will give us an `Instantiate()` method that looks like this:

```
Instantiate(myPrefab, transform.position, transform.rotation);
```

In the preceding code block, `myPrefab` is a reference to a `GameObject` stored as a prefab. We will put this into practice later in the chapter but, first, let's take a look at Rigidbody physics and its importance in modern video games.

Physics

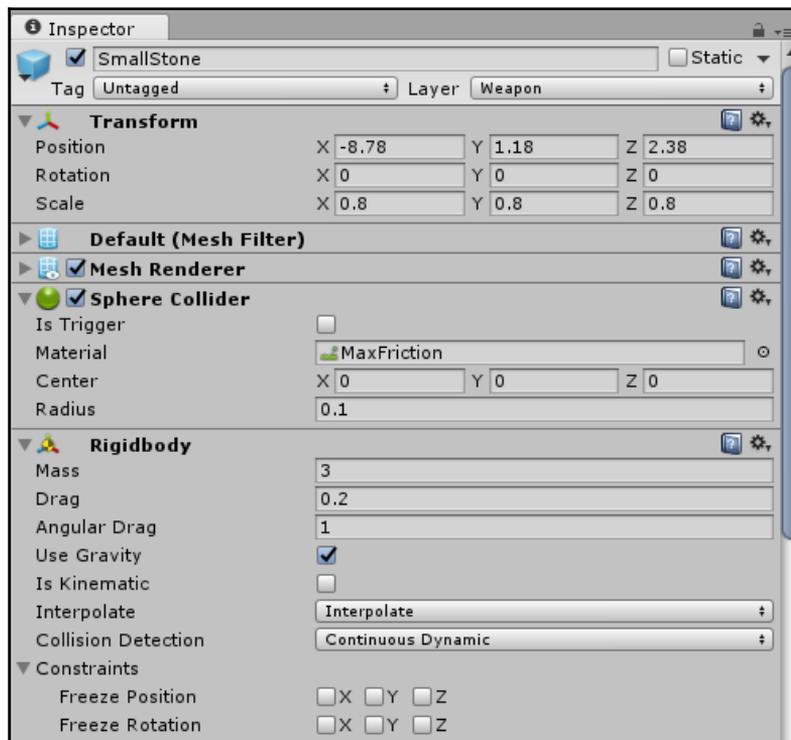
Physics engines give games a means of simulating realism in physical terms, and they are a feature in almost all game engines either natively or as a plugin. Unity uses the **Nvidia PhysX** physics engine, a precise modern physics engine that is used in many commercial games in the industry. Having a physics engine means that not only are physical reactions such as weight and gravity possible, but realistic responses to friction, torque, and mass-based impact are also available.

Forces

The influence of the physics engine on objects is known as **force**, and forces can be applied in a variety of ways through components or scripting. In order to apply physics forces, a `GameObject` must be what is known as a `Rigidbody` object.

The Rigidbody component

In order to invoke the physics engine on an object in Unity, making it a `Rigidbody` object, you must give it a `Rigidbody` component. This simply tells Unity to apply the physics engine to a particular `GameObject`. Having added a `Rigidbody` component, you will see the settings for it in the **Inspector** in the same way as any other object:



Rigidbody always works in conjunction with colliders; the latter must be not a trigger, and if it is a mesh collider, it must be marked as Convex as Unity at present doesn't support Rigidbody and non-convex mesh colliders. Rigidbody components have the following parameters that might be tweaked or controlled through scripting:

- **Mass:** This is the weight of the object in kilograms. Bear in mind that setting mass on a variety of different Rigidbodies will make them behave realistically. For example, a heavy object hitting a lighter object will cause the light object to be repelled further.
- **Drag:** This, as in real terms, is the amount of air resistance affecting an object as it moves. The higher the value, the quicker the object will slow when simply affected by air.
- **Angular Drag:** This is similar to the previous parameter, but angular drag affects the rotational velocity, defining how much air affects the object, slowing it to a rotational halt.
- **Use Gravity:** This does exactly as it states. It is a setting that determines whether the Rigidbody object will be affected by gravity or not. With this option disabled, the object will still be affected by forces and impacts from the physics engine and will react accordingly, but as if in zero gravity.



There are global settings as well for gravity in Unity - simply choose **Edit | Project Settings | Physics** and expand the **Gravity** settings by clicking on the gray arrow next to its title; this allows you to define the amount and direction. Note that the default is **Y-9.81** as the acceleration of gravity on earth is 9.81 m/s^2 . You might want to change this for a game that takes place in outer space, for example.

- **Is Kinematic:** This option allows you to have a Rigidbody object that is not affected by the physics engine. For example, if you wish to have an object repel a Rigidbody with gravity on, such as the trigger in a pinball machine hitting the ball, but without the impact causing the trigger to be affected, then you can use this setting. This setting should be used for any moving object that you wish to control the movement of through scripting or animation, as it means that, firstly, the physics engine is aware of the moving object and therefore will save on performance by not needing to compensate for its updated position and, secondly, the object can still interact with other moving objects without being controlled by the forces of physics directly.

- **Interpolate/Extrapolate:** This setting can be used if your dynamic objects are jittering. Interpolation and extrapolation can be chosen in order to smooth the transform movement based on the previous frame or next predicted frame, respectively. Interpolate is useful when creating characters that use a Rigidbody while the extrapolate is useful for fast-moving objects to ensure that collisions are detectable with other objects.
- **Constraints:** This can be used to lock objects so that they do not move or rotate as a result of forces applied by the physics engine. This is particularly useful for objects that need to use physics but stay upright or in a certain position, such as objects in 2D games that must be locked in the z (depth) axis, for example, or for biped characters that will have a Rigidbody that never rotates in the z and x axes.

Designing the gameplay

To put what we have just looked at into practice, we'll create a fight and run kind of game, full of physics implementations. From real-time physics stones to launch against guards to ragdoll simulation for enemies that are hit and stunned. In this game, the player will have to find the missing pieces around the island without being caught and put in jail again. By playing the game, the player will be rewarded with the artifact pieces required to enter the hut and collect the last one to complete the artifact.

Creating the heavy stone prefab

Now, let's begin our implementations by creating the projectile object to be thrown, that is, the small stone. Drag the `stone` model from the `Models/static/smallstone.fbx` folder directly into the scene. This creates a new stone object in the scene. While it may not be created close enough to the front of the editor viewport, you can easily zoom in to it by hovering your cursor over the **Scene** view and pressing *F* (focus) on the keyboard. Next, we'll make this object of a more appropriate size and shape for a stone by scaling it down and subtly extending its size on the z axis. In the Transform component of the **Inspector** for the `HeavyStone` object, change the **Scale** value for **x** and **z** to `0.5` and **Y** to `0.6`.

The texture to be used on the stone was downloaded with the rest of the book assets. Named `stoneTexture`, this asset can be found in the **Book Assets | Textures** folder in the **Project** panel. In addition to this file, you should also be able to find two audio clips with names `stoneimpact` and `stone_throw` in the `Book Assets/Chapter8-9-10/Audio` folder.

Adding physics to the stone prefab

Now, because our `HeavyStone` `GameObject` needs to behave realistically, we'll need to add a `Rigidbody` component in order to invoke the physics engine for this object. Select the object in the **Hierarchy** and navigate to **Component | Physics | Rigidbody**. This adds the `Rigidbody` component that will apply gravity. As a result, when the player throws the object forward, it will fall over time as we would expect it to in real life. The default settings of the `Rigidbody` component can be left unadjusted; you will not need to change them on this occasion. We can test whether the stone falls and rolls (if you have created it over land! If not, you can move it with the `Translate` tool) now by pressing the `Play` button and watching the object in the **Scene** or **Game** view.

Saving as a prefab

Now that our `HeavyStone` `GameObject` is complete, we'll need to store it as a prefab in our project to ensure that we can instantiate it using code as many times as we like, rather than simply having a single object.

To save your object as a prefab, simply drag and drop an object from the **Hierarchy** panel anywhere onto the **Project** panel. Drag the `HeavyStone` object we just made and drop it into the `Prefabs` folder we made earlier to save it there as a prefab.



Your prefab will be named the same as the originating object and you should see the text for the object in the **Hierarchy** turn blue to signify that this is now linked to a prefab asset. The advantage of this is that any instance of a prefab can be adjusted by altering settings on the prefab version in the **Project** panel.

For example, in a game scene with several buildings, all based on the same prefab, changing the value of an attached script or swapping a material could be done to the prefab instead of the instances in the scene. Select it in the **Project** view and change the values and settings in the prefab in order to affect all its instances in the scene.

Now that your `HeavyStone` is stored as a prefab, delete the original instance from the **Hierarchy** as we don't need it any more in the scene because it will be instantiated at runtime.

Throwing stones at guards

As we will allow our player to throw the stone prefab we have just created, we will need two things: a script to handle the instantiation and an empty `GameObject` to act as a reference point for the position where the `HeavyStone` objects in the world are to be created.

When we throw a ball (overarm style) in real life, it comes into view at the side of our head as our arm comes forward to release the ball. Therefore, we should position an object just outside of our player's field of view and ensure that it follows wherever they look. This is the reason we will set the stone spawn point at the character's skeleton right hand.

Next, we'll need to script for instantiating the stone and setting its propulsion when the player presses the fire button.

The StoneLauncher component

To make the player able to launch stones, we will add a script that implements three key steps:

1. Instantiation of the `HeavyStone` prefab referenced in the inspector upon right mouse button click.
2. Using the `rightHandRef` position to spawn the stone in the right hand.
3. Assigning a velocity to the `Rigidbody` component to propel the stone forward as soon as it is created.

In order to achieve this, select the `Scripts` folder in the **Project** panel and then create a new C# (or JavaScript) file from the **Create** button on the same panel. Rename the **New Behaviour Script** as `StoneLauncher` by pressing *Return* (Mac) or *F2* (PC), and then open this in the script editor by double-clicking on its icon. Replace the existing code with the following code:

```
using UnityEngine;
public class StoneLauncher : MonoBehaviour {
    public Rigidbody rockyStone;
    // Throw a stone from the right hand
    public void ThrowObject(Transform rightHandRef)
    {
        // instantiate stone object
        Rigidbody newStone = Instantiate(rockyStone, rightHandRef.position,
transform.rotation) as Rigidbody;
        newStone.name = "HeavyStone";
    }
}
```

```

        newStone.velocity = transform.forward * 30f;
    }
}

```

We will later assign player's RightHand transform in the **Hierarchy**, to StoneLauncher's rightHandRef variable:



The player in the hierarchy with the appropriate children expanded for reaching the right hand GameObject

Checking for player input

As the main logic for the player resides within the `ThirdPersonCustomCharacter` class, we will use our `StoneLauncher` class as a bridge with a single method that will be called at the correct time by the `ThirdPersonCustomUserControl` class by checking the inputs.

Given that we need to listen for player key presses in each frame, we need to write our code for the launcher inside the `Update()` method. Move the closing right curly brace (`}`) of this method down by a few lines, and then add the following:

```

// Prevent the fight when character is talking or when is in
the initial phase
if (!m_Talking && !is_Hurt)
{
    // fighting mode key trigger
    if (Input.GetMouseButton(0))m_Character.Fight(1);
    if (Input.GetMouseButtonUp(1) && canThrow)
        m_Character.Fight(2);
}

```

The `if (!m_Talking)` check will prevent starting a stone launch or a punch when talking to an NPC or when the action is paused for some other reasons. Then, it checks the user inputs and waits for the mouse buttons to launch the `Fight` method on the `ThirdPersonCustomCharacter` class with an option that will tell the method whether to throw a stone or give a punch.

Writing the Fight method

We will write the `Fight` method by declaring it `public`, to be able to call it externally. This method will catch the `style` parameter and, according to what is passed, will trigger the punch, or the stone launch animation for the hero:

```
// Called by the player controller
public void Fight(int style)
{
    if (m_Animator.GetCurrentAnimatorStateInfo(0).IsName("Grounded"))
    {
        if (m_Fighting_Style == 0 && m_Animator.GetInteger("FightStyle") ==
0)
        {
            m_Fighting_Style = style;
            m_Animator.SetInteger("FightStyle", m_Fighting_Style);
        }
    }
}
```

Instantiating the heavy stone

Next, we need to instantiate (also known as create or spawn) the actual stone itself, also within the current `if` statement. Switch back to the script editor now or relaunch the `StoneLauncher` script if you have closed it.

Given that we have created the stone and saved it as a prefab, we should establish another public variable so that we can assign our prefab to a variable in the **Inspector** later. Add another one below the last public variable we just added, as shown:

```
public Rigidbody rockyStone;
```

This adds a public variable of the `Rigidbody` type. Although our stone is stored as a prefab asset, we'll be creating a `GameObject` with a `Rigidbody` component in our scene when we instantiate it, and hence the data type. This ensures that we cannot drag a non-`Rigidbody` object to this variable in the **Inspector**. By strictly assigning the type `Rigidbody`, it also means that if, we wish to reference the `Rigidbody` component of this `GameObject`, we won't need to use the `GetComponent()` command to select the `Rigidbody` component first; we can simply write code that speaks directly to the `Rigidbody` class when referring to this variable name.

Now, inside the `if` statement in `Update()`, place the following line after the existing audio line:

```
Rigidbody newStone = Instantiate(rockyStone, transform.position,
    transform.rotation) as Rigidbody;
```

Here, we establish a local variable called `newStone`; it is said to be local because it is declared within the `Update()` method and therefore is not accessible outside this method. We are passing the creation (instantiation) of a new `Rigidbody` object into this variable and therefore we set that as the data type.



Remember that the three parameters of an `Instantiate()` method are the object, position, and rotation, in that order. You'll see that we have used the `stonePrefab` public variable in order to create an instance of our prefab and then inherited the position and rotation from the `transform` component of the object this script will be attached to: the `Launcher` object.

Now, let's check our progress in Unity, save your script now and switch back to the editor. Select the `Launcher` child object of `Main Camera` in the **Hierarchy** and you will see that there is a new public variable, called `Stone Prefab`, that needs assigning. Now, drag and drop the `Stone` prefab from the `Prefabs` folder to this variable to assign it.

Now, press `Play` at the top of the interface to test your game. Each time you press the **Fire1** button, you should now be creating a new instance of your stone prefab, causing objects, named `Stone (Clone)`, to appear in the **Hierarchy**.

However, the stones aren't going anywhere! At this point you will feel the need for speed, so let's add some velocity to our rocks! Press `Play` again to stop testing and return to your `StoneLauncher` script in the editor.

Naming instances

Whenever you create objects with `Instantiate()` during runtime, Unity takes the name of the prefab and appends `(Clone)` to it when creating new instances. As this is rather a clunky name to reference in code, which we will need to do for our targets later, we can simply name the instances that are created by adding the following line beneath the one we just added:

```
newStone.name = "HeavyStone";
```

Here, we have simply used the variable name we created, which refers to the new instance of the prefab, used dot syntax to address the name parameter, and a single equals symbol to set it.

Assigning velocity

While this variable will create an instance of our stone, our script is not yet complete as we need to assign a velocity to the newly created stone too. Otherwise, as we just saw, any stones will simply be created and fall to the ground. In general terms, velocity is regarded as speed plus direction; we will begin by addressing the former. To allow us to adjust the speed of the thrown stone, we can create another public variable above the `Update` method. In order to give us precision, we'll make this variable a float data type, allowing us to type in a value with a decimal place if we wish. Place the following code below the last public variable named `stonePrefab`, which we made earlier:

```
public float throwSpeed = 30.0f;
```

Now, you need to add a line beneath the following line in your if statement:

```
newStone.name = "HeavyStone";
```

The following line is the line to be added:

```
newStone.velocity = transform.forward * throwSpeed;
```

Here, we are referencing the newly instantiated `Rigidbody` object `NewStone` by its variable name and then using dot syntax to address the `Rigidbody`'s velocity property.

We have set the direction we need to throw using `transform.forward` as this command simply references the forward-facing direction of the launcher's `Transform` component, giving us its local direction without having to use the `transformDirection` command we looked at previously at the beginning of the book.

Using `transform.forward` is a shortcut to saying $(0,0,1)$, so using this by itself will only propel our stone by 1 on the z axis. Instead, this is multiplied by our `throwSpeed` value to give our Rigidbody stone a velocity.

Let's throw some stones! Save your script in the script editor now and switch back to Unity. Press Play and try firing stones again; this time, they should be thrown forward in the direction you are facing, and with the right hand! Let's continue to improve our script by adding some development safeguards, once you are done playing with your new stone throwing mechanic, stop the Play mode and return to the script editor.

Adding development safeguards

In this section, we will look at a few examples of ways we can ensure that our code does not cause problems for our development or the game itself. We have just created a mechanic in which the player character can throw stones. However, we need to ensure a number of things for this mechanic to not cause any problems before we continue:

- We need to ensure that the `HeavyStone` prefab we are throwing is a Rigidbody because the velocity we are setting refers to this class, so we will add a safeguard for this in the code.
- We should ensure that collisions never occur between the player character and the stones.
- We will look at two methods of doing this: through code and using layers and the collision matrix in Unity.
- Finally, we should have an audio clip that plays when the character throws, so we should ensure that the `GameObject` this script is assigned to has an audio source component attached.

Checking component presence

When developing, you will often need to ensure that an object has a component attached to it before addressing properties or commands related to that component. In this instance, we are creating our `HeavyStone` as a Rigidbody object so we already know that the prefab assigned to the `Instantiate` command will be a Rigidbody, otherwise this would have caused an exception (error) in the script when testing.

However, in order to show how to ensure that a component exists, we will briefly look at the following safeguard in order to help you learn how to add a component if it is not already present.

For example, with our instantiated stone, we can check the new instance's variable, `newStone`, for a `Rigidbody` component by saying the following:

```
if(newStone.rigidbody == null) {
    newStone.AddComponent(Rigidbody);
}
```

Here, we are saying that if there is no `Rigidbody` attached to this variable instance (if it is null), add a component of that type.

Safeguarding collisions

While we have set up our `StoneLauncher` on the player main `GameObject`, the stone can collide with the player character's capsule collider when throwing; we should still ensure that the `HeavyStone` itself never actually collides with the player. This can be done in two different ways in Unity: using ignore collision code, or placing objects on layers that do not interact. We will look at both the techniques to ensure that you are best equipped in your future development.

Using the `IgnoreCollision()` method

To force objects in our game to not interact with one another via code, we can include the following piece of code in order to safeguard against instantiating new stones that accidentally intersect with our player's collider. This can be done using the `IgnoreCollision()` command of the `Physics` class. This command typically takes three arguments, as follows:

```
IgnoreCollision(Collider A, Collider B, whether to ignore or not);
```

As a result, we simply need to feed it the two colliders that we do not want the physics engine to react to, and set the third parameter to true. Add the following line to your `StoneLauncher` class beneath the last line we added earlier:

```
Physics.IgnoreCollision(transform.root.collider, newStone.collider, true);
```

Ignoring collisions with layers

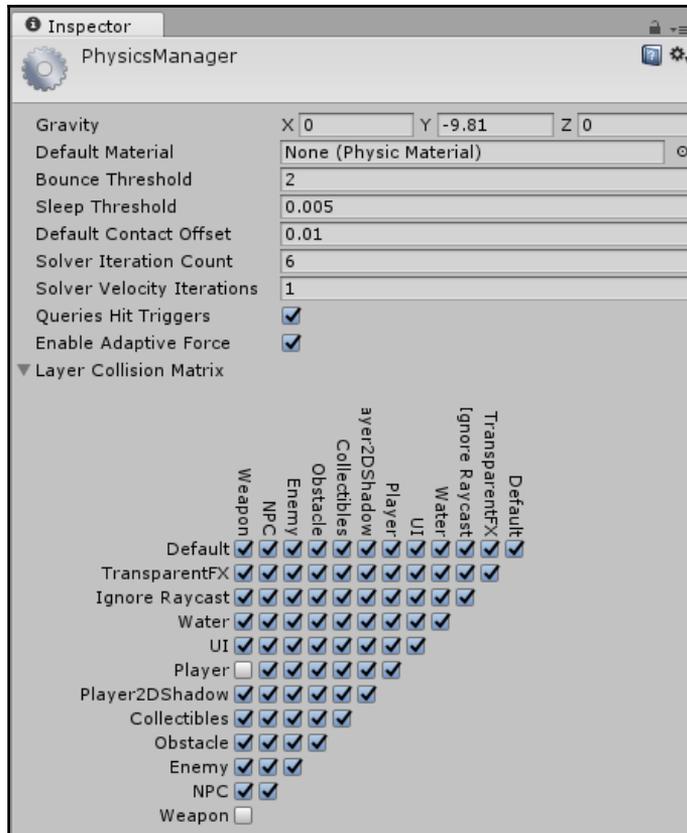
In addition to the `IgnoreCollision()` command of the `Physics` class, we can use the Layers built into Unity in order to tell the physics engine which objects should not collide with one another. By placing objects onto opposing layers and deselecting them within the **Physics** settings **Layer Collision Matrix**, we are telling Unity not to register collisions with objects added to those layers.

Let's begin by creating two new layers. Open the **Tag & Layers** settings by navigating to **Edit | Project Settings | Tags & Layers**. Beneath the existing **Tags** in your project, you will see a list of built-in layers and, beneath that, starting from **User Layer 8**:



Add a new user layer for `Weapon` as shown in the screenshot. Simply press *return* to confirm. We chose `Weapon` instead of `Bullet` or `Projectile`, because we will be using it also for the hand collider punch later in Chapter 13, *Optimization and Final Touches*. Next, to adjust the configuration of how these layers interact, go to **Edit | Project Settings | Physics**.

Expand the settings for **Layer Collision Matrix** at the bottom, and deselect where these two layers intersect:



This sets up the layers to ignore collisions with one another; now we can simply assign our objects to these layers. Select the `Warrior_RM` player hero in the **Hierarchy** and, at the top of the **Inspector**, select **Player** from the **Layers** drop-down menu. You will be prompted to assign the layer **Player** to children GameObjects, choose to do so. Now, select the `stone` prefab in the **Prefabs** folder of the **Project** panel, and use the same approach to set this prefab's layer to **Weapon**.

We now have two safeguards in place to stop our stone projectile from interacting with the player itself, finding solutions such as this is an important part of minimizing bugs toward the end of your game's development.

Final tweaks

Although the code we have just added is more for safeguarding against bugs than it is functional, we should always test when adding new parts to our script. Press the Play button now, check the **Console** bar at the bottom of the Unity interface for errors, and either left-click or right-click to throw stones! Press the Play button again to stop testing once you are satisfied.

If something does not work correctly, return to your script and double-check whether it matches the full script, which should be looking like the following:

```
using UnityEngine;

public class StoneLauncher : MonoBehaviour {

    public Rigidbody rockyStone;
    public Vector3 offset = new Vector3(0, 0.25f, 0);
    [Range (0f,1f)] public float launchPower = 0.5f;

    // Throw a stone from the right hand
    public void ThrowObject(Transform rightHandRef)
    {
        // instantiate stone rigidbody
        Rigidbody newStone = Instantiate(rockyStone,
            rightHandRef.position + offset, transform.rotation) as
            Rigidbody;
        newStone.name = "HeavyStone";
        newStone.velocity = transform.forward * 60f * launchPower;
    }
}
```

This class isn't ready to be tested yet as the public method will not be called by itself; it will be called by an external script, our `ThirdPersonCustomCharacter` class. The code will work when the modification we are going to make to our `ThirdPersonCustomUserControl` class calls the `Fight` method on the `ThirdPersonCustomCharacter` class.

Instantiation restriction and object tidying

Instantiating objects in this manner is the ideal use of Unity's prefab system, making it easy to construct any object in the scene and create many clones of it during runtime. However, creating many clones of a `Rigidbody` object can prove costly as each one invokes the physics engine and requires its own draw call from the GPU. In addition, as it negotiates its way around the 3D world, interacting with other objects - it will be using CPU cycles (processing power).

Now imagine if you were to allow your player to create an infinite number of physics-controlled objects; you will appreciate that your game may slow down after a while. As your game uses too many CPU cycles and memory, the frame rate becomes lower, giving a jerky look to your previously smooth-motivated game. This will, of course, be a poor experience for the player and, in a commercial sense, will kill your game.



In the previous version of Unity, a new and long-awaited feature was introduced, GPU instancing. This technique, that uses a special shader, will allow us to instantiate a large number of objects that share the same shader and geometry without overwhelming the CPU with a lot of draw calls, which usually results in bad performance or slowdowns.

Rather than hoping that the player does not throw many stones, we will do two things to avoid too many objects slowing down the game's frame rate:

- Allow the player to throw stones only when the throw animation is near to the correct point in time (with a safety check that restricts the game to instantiate just one stone for each launch)
- Write code to remove stones from the scene after a defined time since spawn

Implementing stone throw animation

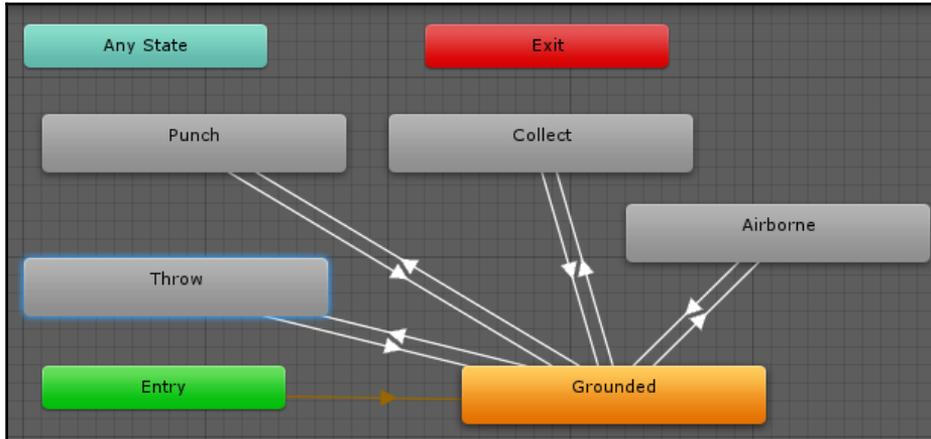
While we can simply activate the stone launch at key press, spawning the stone to launch at a player's generic discreet position and toward its forward direction, we want to add some realism in this chapter and, hence, we will spawn the stone exactly in the right hand, and we will throw the stone from there, toward the forward direction:



To achieve this, we will edit the player character **Animator** state machine we worked on in Chapter 5, *Character Animation with Unity*, so let's select the `Warrior_RM` `GameObject` in the scene hierarchy and open the **Animator** window if it isn't already open, then, as in the screenshot above, we will add a **Throw** state, linked to the main default **Grounded** state and create the two-way transitions.

For the **Throw** state, we will set the `goalie_throw` animation clip situated in the `Chapter5-6-7\Models\Characters\Warrior_Mecanim_RM` folder.

To do so, select the new **Throw** state in the **Animator** window with the `Warrior_RM` selected, as shown in the following screenshot:



In the **Inspector**, you should see the animation state information. Then, drag the `goalie_throw` animation nested within the `Warrior_final_RM @ goalie_throw.fbx` file from the project view into the **Inspector** in the **Motion** slot and set the playback **Speed** to 1.5 (150%):

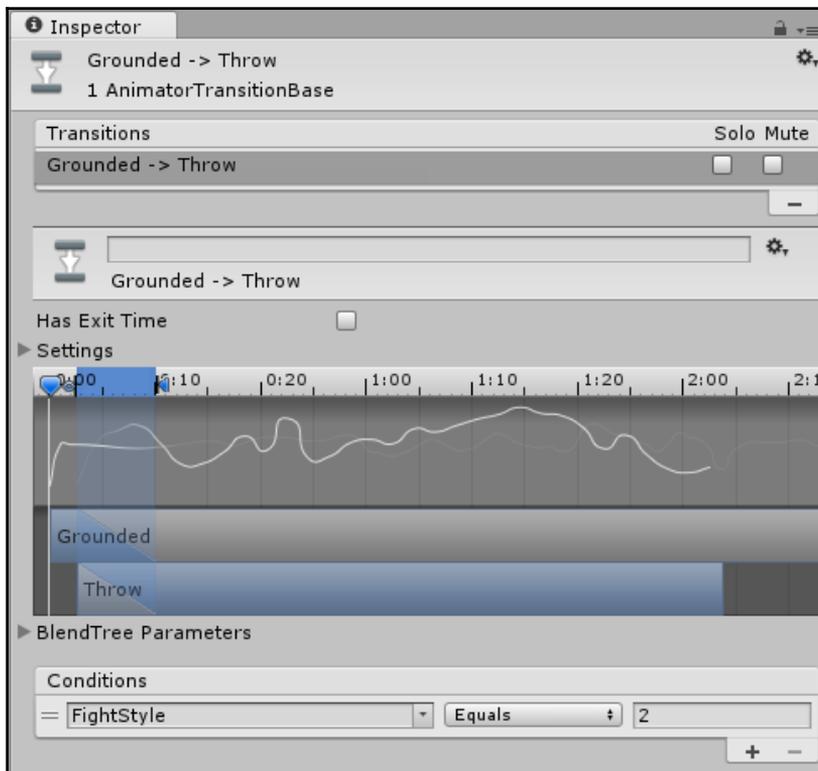


After doing this, we will switch back to the **Animator** window and select **Grounded -> Throw** transition (the white arrow connecting the two states).

Selecting the transition arrow in the **Animator** window will allow us to see transition options in the **Inspector**. To be able to drive from scripting the state change, we will:

- We will ensure that the **Has Exit Time** option is not checked and restrict the transition end to around **0:10** in the timeline, as in the following screenshot.
- Add a Condition, by clicking on the + (plus) button, set **FightStyle** for the parameter and **Equals 2** ($==2$) for the condition.

This will instruct **Animator** to switch to the **Throw** animation state when the **FightStyle** parameter becomes 2:



The Inspector showing the Animator states transition properly set, with the FightStyle condition set

Observe the following line of code:

```
Animator.SetInt("FightStyle",2)
```

This line of code is in the `Fight` method of the `ThirdPersonCustomCharacter` class and will be called by our user input code that we wrote earlier. In the same way, we will select the **Throw->Grounded** transition and set its **Exit Time** and a condition with the `FightStyle` parameter with **Equal** to 0 (zero) for the condition value. While this series of calls might sound complicated in the implementation, it will ensure component consistency and reusability. Let's start from the *two bridge methods* that will be called at the right time in the animation timeline by the events we are about to set. First, we want to declare a new public Transform variable to store the right-hand bone in the player character skeleton so that we can drag the object from the **Hierarchy** to the component slot after saving the script; let's call it `rightHandRef`:

```
public Transform rightHandRef;
```

Then we want to add these two methods to the `ThirdPersonCustomCharacter` class and declare them public to enable the possibility of calling them from the Animator events:

```
// Throw a stone from the right hand
public void ThrowStone()
{
    SendMessage("ThrowObject", rightHandRef);
}

// restore the fight style = 0, to go back to idle state
public void EndOfSlash() {
    m_Fighting_Style = 0;
    m_Animator.SetInteger("FightStyle", 0);
    rightHandRef.GetComponent<Collider>().enabled = false;
}
```

As you can see, the first method will send a message to the same `GameObject` that is carrying the `StoneLauncher` component we wrote and, hence, will be successfully receiving the message. The parameter for this message will be the right-hand reference (`rightHandRef`) that we previously declared and assigned dragging the Transform to the component slot. We have to pass a transform parameter because the method was declared with an input variable need:

```
public void ThrowObject(Transform rightHandRef)
```

It was declared public for your commodity because you might want to call this method from another class, such as the following:

```
GetComponent<StoneLauncher>().ThrowObject(spawnPoint);
```

At this point, the code is ready. How will we set up **Animator** to fire these two methods on a certain animation frame?

Activating the stone launch at the right frame

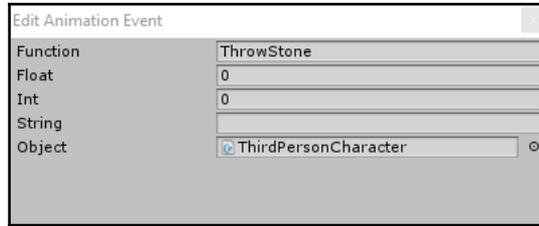
Select the `Warrior_final_RM @ goalie_throw.fbx` animation file in the **Project** view. Select the main object, not the animation clip nested into it. The **Inspector** will show the properties of the file.

Switch to the **Animation** tab then scroll down until you see a group of expandable options: **Curves**, **Events**, **Mask**, **Motion**, and so on:



Expand the **Events** section by clicking on the small arrow then use the animation scrub bar after the Play button to animate the preview and position the animation at the point where you want the stone to be launched. Now click on the button with a + to add an event at that point.

You should see a small white bar appear where the red cursor is in the timeline, so double-click on it to bring up the **Edit Animation Event** pop-up window:



The Animation Event pop-up window

In this window, it is possible to specify a class, a method to call in this class, and a float, int, or string parameter to send to the method, if required. In our case, we will simply call the `ThrowStone()` method on the `ThirdPersonCustomCharacter` class without any parameter because will be this method to take care of calling the `ThrowObject` method on the `StoneLauncher` with the right parameter. With the same technique, we will add another event in the `goalie_throw` animation timeline, around 0:070 when the launch is basically near to finish. We will use this event to track down the end of the slash of the right arm to be able to send a message when this event occurs, to inform **Animator** that it can go back to the **Grounded** state by setting the **FightStyle** parameter back to 0. In fact, we will use that **Animator** parameter for the two transitions between the **Throw** and **Grounded** states:



In the following screenshot, you can see the point for the second event that will fire our `EndOfSlash()` method

Removing stones in a smart way

As mentioned earlier, too many physics-controlled objects in your scene can seriously affect performance. Therefore, in cases such as this, where your objects are simply throwaway objects (objects that need not be kept, no pun intended!), we can write a script to automatically remove them after a defined amount of time.

Select the `Scripts` folder inside the **Project** panel, click on the **Create** button to make a new script. Rename this script `HeavyStone` by pressing *Return* (Mac) or *F2* (PC) and retyping. Then, double-click on the icon of the script to launch it in the script editor.

Remove the default `Update()` method from this script as we do not need it. To remove any object from a scene, we can simply use the `Destroy()` method and implement its second argument in order to establish a waiting period. `Destroy()` works as follows:

```
Destroy( object or component to destroy, optional time delay);
```

To make this script reusable, we will establish a public variable to allow us to specify the time delay. This is because we may wish to use this to remove a different kind of spawned object after a defined time, but not the same amount of time as for our stone prefab. Above the `Start()` method, add the following variable:

```
public float removeTime = 3.0f;
```

We will call the `Destroy()` method within the `Start()` method:

```
void Start () {  
    Destroy(gameObject, removeTime);  
}
```

Using the `Start()` command, we will call `Destroy()` as soon as this object appears in the world, that is, as soon as it is instantiated by the player pressing the fire button. By referring to `gameObject`, we are simply saying the object this script is attached to. After the comma, we simply state a time in seconds to wait until activating this command by using our `removeTime` float variable. As a result of this script, a stone will stay in the world for three seconds as soon as it's thrown, and then it will be removed. Go to **File | Save** in the script editor and return to Unity.

Previously, when we wrote scripts, we attached them to objects in the scene we were working on. However, in this instance, we have already finished working on our stone prefab, and we no longer have a copy in the scene. There are two ways of applying the script we have just written to the prefab. To do this the easy way, you can drag and drop the `HeavyStone` script from its position in the **Project** panel onto the prefab in the **Project**. Alternatively, select the `HeavyStone` prefab you made earlier in the **Project** panel and go to **Component | Scripts | HeavyStone**.

Alternatively, to take a more long-winded route, you can modify the prefab in the **Scene** in the following way:

- Drag the `HeavyStone` prefab to the **Scene** window or into the **Hierarchy**.
- Go to top menu: **Component | Scripts | HeavyStone**, when prompted with the warning: **Adding a component will lose the prefab parent**, click on the **Add** button to confirm.
- Save this change to the original prefab by going to **GameObject | Apply Changes to prefab** or by pressing the **Apply** button at the top of the **Inspector**.
- Delete the `GameObject` from the scene.



In this instance, it is recommended to use the former, that is, the single-step route, so let's do that now. However, in some instances, it can be useful to take a prefab back into the scene and modify it before you apply changes to the prefab. For example, if you are working on something visual, such as a particle system, then you will need to see what effect your adjustments or newly added components will have. Therefore, taking a prefab of such an object into the scene to edit will be essential.

Go to **File | Save Project** and save the **Scene** in Unity now to update your progress so far. Press Play to test, and you should now see that when standing on the mat and throwing stones, they only exist in the world for the default 10 seconds of the `Destroy()` command that we have specified for the `StoneLauncher` component in the **Inspector**. This means that the maximum amount of stones in the game at any time will be limited.

Finishing touches

We will refine some mechanics and do some more advanced steps in this section to make the fight with guards funnier and more realistic. We will create a new version of the Guard AI, the same as we created in [Chapter 8, AI, NPC, and Further Scripting](#), but with ragdoll physics simulation aboard. Finally, we are going to add a punch animator state and method for the hero, with precise collider on the right hand, which will collide with enemy colliders to save him when the guards are too close for a stone launch, make him able to stun the guards with a well-set punch packed in their face.

Ragdoll physics simulation

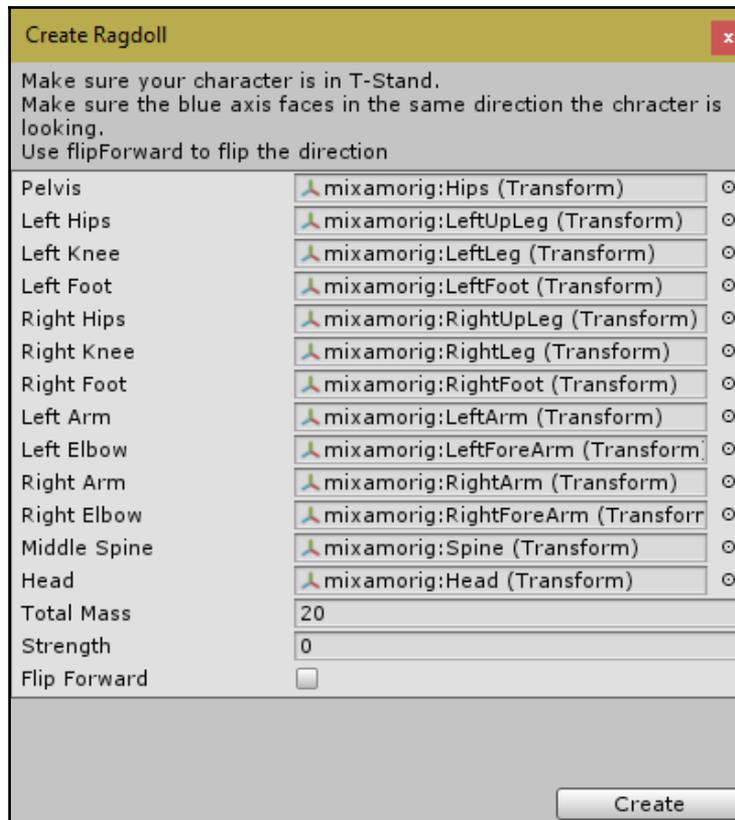
When you want to have some more realism, and/or don't want to spend effort animating many kinds of different death/dying animations, rigidbodies and physics come to the rescue. Unity has, since the earliest versions, a handy tool that allows you to specify the rigged character skeleton bones into ragdoll slots and then, with a single click, to create all the components necessary to have a realistic ragdoll simulation for that character.

We will use the harder approach this time, which will be *not* to have a copy of the guard without components with just ragdoll colliders and rigidbodies, and instantiating it, when needed, in the place of the guard. In this game we are not killing the guards by design, instead we are just temporarily stunning them, so, they will soon be back on their feet. There is no reason to write a special class to no point to use that we will write a special class to take care of enabling/disabling those components at runtime. First things first, we will create the ragdoll for the guard prefab.

Select one of the AI guards in the hierarchy, then open the main menu and select **GameObject->3D->Ragdoll**. This will bring up the Unity Ragdoll creation wizard.

As you can see from the wizard, it will be required that you drag the guard skeleton's corresponding bones to the wizard slots. For example, we will drag the `LeftUpLeg` child in the `Left Hips` slot, and we will drag the `Hips` child in the `Pelvis`.

Leave 20 for **Total Mass** parameter and 0 for **Strength**:



Create Ragdoll [X]

Make sure your character is in T-Stand.
Make sure the blue axis faces in the same direction the chracter is looking.
Use flipForward to flip the direction

Pelvis	 mixamorig:Hips (Transform)	○
Left Hips	 mixamorig:LeftUpLeg (Transform)	○
Left Knee	 mixamorig:LeftLeg (Transform)	○
Left Foot	 mixamorig:LeftFoot (Transform)	○
Right Hips	 mixamorig:RightUpLeg (Transform)	○
Right Knee	 mixamorig:RightLeg (Transform)	○
Right Foot	 mixamorig:RightFoot (Transform)	○
Left Arm	 mixamorig:LeftArm (Transform)	○
Left Elbow	 mixamorig:LeftForeArm (Transform)	○
Right Arm	 mixamorig:RightArm (Transform)	○
Right Elbow	 mixamorig:RightForeArm (Transform)	○
Middle Spine	 mixamorig:Spine (Transform)	○
Head	 mixamorig:Head (Transform)	○
Total Mass	<input type="text" value="20"/>	
Strength	<input type="text" value="0"/>	
Flip Forward	<input type="checkbox"/>	

Create

After pressing Create, you will see the guard GameObject filled up with colliders, rigidbodies, and physics joint that link rigidbodies. Test the ragdoll by pressing the Play button and observe the simulation.



We can tweak bone colliders if we like, as well as rigidbodies and joints properties to achieve an even better result for the physics evolution of the ragdoll.

Scripting the ragdoll simulation

To be able to deactivate and activate the ragdoll simulation on the guards, we will use the following class. For simplicity, we will keep this class in the `UnityStandardAssets.Characters.ThirdPerson` namespace.

This class will take care of enabling or disabling all the Rigidbodies of the guard `GameObject` children, avoiding touching the main `Rigidbody` that is still needed to keep the AI on the ground:

```
using UnityEngine;
using System.Collections;

namespace UnityStandardAssets.Characters.ThirdPerson
{
    public class RagdollCharacter : MonoBehaviour {

        // At Start, deactivate all components that serve ragdoll
        physics.
        void Start() {
            // Call DeactivateRagdoll method at start
            DeactivateRagdoll();
        }
        // A method that activates all components for ragdoll physics
        public void ActivateRagdoll() {

            // Stop the AI by putting as alive = false the character
            gameObject.GetComponent<AdvancedAI>
            ().SetActiveStatus(false);

            // Disable Animator component
            gameObject.GetComponent<Animator>().enabled = false;
            // Disable NavMeshAgent component
            gameObject.GetComponent<NavMeshAgent>().enabled = false;

            // Find every Rigidbody in character's hierarchy
            foreach (Rigidbody bone in
            GetComponentsInChildren<Rigidbody>()) {
                // Set bone's rigidbody component as not kinematic
                bone.isKinematic = false;

                //Enable collision detection for rigidbody component
                bone.detectCollisions = true;
            }

            // this rigidbody must not detect collision and be
            kinematic
        }
    }
}
```

```
GetComponent<Rigidbody>().isKinematic = true;
GetComponent<Rigidbody>().detectCollisions = false;

// Find every Collider in character's hierarchy
foreach (Collider col in
GetComponentInChildren<Collider>())
{
    // Enable Collider
    col.enabled = true;
}
// main collider must be disabled
GetComponent<CapsuleCollider>().enabled = false;
GetComponent<SphereCollider>().enabled = false;
}
}
```

Reviving the enemy

We will reactivate the enemy by making the inverse of the `ActivateRagdoll` method on our `RagdollCharacter` class; we will write an opposite method that will disable all the ragdoll physics components on the guard's skeleton bones that are involved and re-enable the `AdvancedAI` and `ThirdPersonCharacter` components on him.

We will write the method right after the `ActivateRagdoll` method's closing bracket:

```
// deactivate all components needed for ragdoll physics

public void DeactivateRagdoll() {
    // Enable Animator component
    gameObject.GetComponent<Animator>().enabled = true;

    // Enable NavMeshAgent component
    gameObject.GetComponent<NavMeshAgent>().enabled = true;

    // Find every Rigidbody in character's hierarchy
    foreach (Rigidbody bone in
GetComponentInChildren<Rigidbody>()) {
        // Set bone's rigidbody component as kinematic
        bone.isKinematic = true;
        // Disable collision detection for rigidbody component
        bone.detectCollisions = false;
    }
    // this rigidbody must detect collision and be not
    kinematic
    GetComponent<Rigidbody>().isKinematic = false;
}
```

```
GetComponent<Rigidbody>().detectCollisions = true;
// Find every Collider in character's hierarchy
foreach (Collider col in
GetComponentInChildren<Collider>())
{
    // Disable Collider
    col.enabled = false;
}
// main collider must stay enabled
GetComponent<CapsuleCollider>().enabled = true;
GetComponent<SphereCollider>().enabled = true;
}
```

Using coroutines to time game elements

In our hut door mechanic in Chapter 6, *Creating the Environment*, we used a trigger mode collider to detect the player's presence and open the door, and then set up a float-type timer variable in our `DoorManager` script that increments in an `Update()` method, counting up to a defined time before closing the door.

However, although the problem of resetting our targets is effectively the same as opening and closing a door, something happens, there is a timed delay, and then something else happens, in the case of our targets, we'll level up our programming knowledge and use something called a coroutine instead of incrementing a timer.

Setting values in `Update()` is costly in terms of performance, and it should be avoided if possible. However, for the purpose of learning Unity, we have opted to show you both the inefficient method of timing and resetting and, now, the more efficient method. Being able to write and increment timers is also useful in other contexts, so ensure that you don't forget it; it'll come in handy when making a timed game later!



A coroutine, although it sounds complex, looks and behaves like a standard method, but can use `yield` to pause until a defined time or condition is met. The main visible difference is seen in C#, where the method begins with a return type called `IEnumerator` instead of `void`.

Let's put this into practice; add the following code to our `RagdollCharacter` script to make the guards *wake up* after they have been stunned:

```
// A yield coroutine to restore the character after some time
IEnumerator Restore()
{
    // Wait for n seconds
    yield return new WaitForSeconds(60f);
    // Deactivate Ragdoll
    DeactivateRagdoll();
    // finally, reactivate the AdvancedAI component process
    gameObject.GetComponent<AdvancedAI>().SetActiveStatus(true);
}
```

Finally, we will modify the `OnCollisionEnter()` method we have just written on the `HeavyStone` class to start guard ragdoll simulation by adding the following:

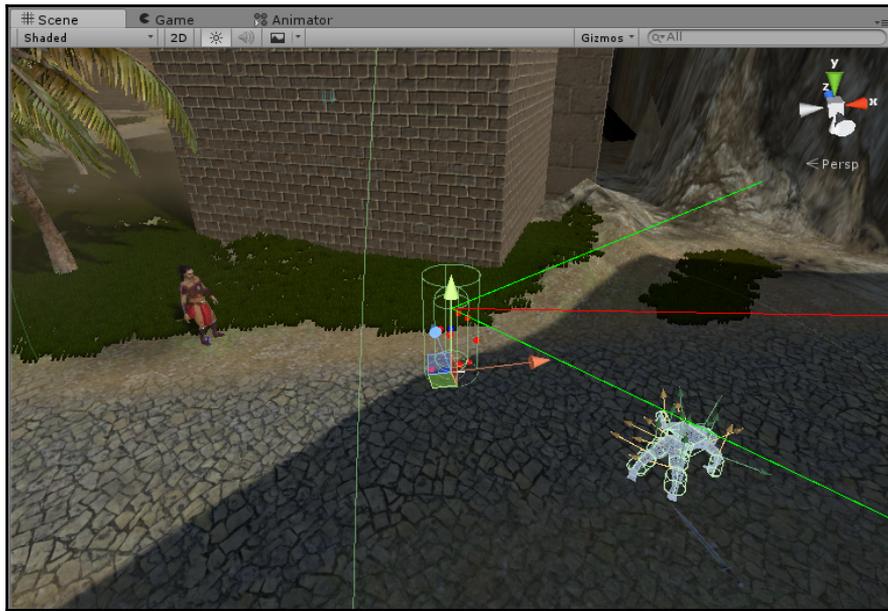
```
if (other.gameObject.GetComponent<RagdollCharacter>() != null)
{
    other.collider.gameObject.SendMessage("ActivateRagdoll");
}
```

We used what we have just learned about the component existence check; if the hit enemy has a `RagdollCharacter` script component attached, the `ActivateRagdoll` method will be called.

At this point, we just need to start the restore coroutine when the `ActivateRagdoll` method finishes its code, so, right after the capsule and sphere collider disabling, add the following:

```
// Start coroutine that will restore AI status after some time
StartCoroutine(Restore());
```

We need some hack to avoid the situation in the following screenshot, where the ragdoll evolution brings the guard away from its main gameObject, which had the main components disabled, resulting in a weird behavior, when the guard will wake up and appear in a different position than the one where he was lying down:



We will put this line of code, before anything, into the `DeactivateRagdoll` method:

```
GetComponent<Rigidbody>().position = new Vector3(  
    rightHandT.parent.parent.parent.position.x,  
    GetComponent<Rigidbody>().position.y,  
    rightHandT.parent.parent.parent.position.z  
);
```

This is because we don't need to move the main object right away; that can still result in a wrong position as the ragdoll simulation can be stronger and the body will move further away. This way, we move the main object into ragdoll position just right before the ragdoll deactivation code is executed. Save all will save all the unsaved scripts in Visual Studio; do so to update your progress so far.

Press Play to test the code. You should finally see that, after hitting a guard, when the enemy stunned wait is over, they will restart on foot at the same spot where the ragdoll body simulation brought it.

The guard who was previously hit, after the `Restore()` method is called, will finally wake up in the correct position:



Collision detection

As we need to detect collisions between the stones and the enemies, we'll need to add some code to our stone prefab. We will check for the target mesh being hit by the stone in it. Let's insert the following collision detection method in our `HeavyStone` class:

```
void OnCollisionEnter(Collision other)
{
    if (other.gameObject.layer == enemyLayerNumber)
    {
        if (other.gameObject.GetComponent<RagdollCharacter>() != null)
        {
            other.collider.gameObject.SendMessage("ActivateRagdoll");
            other.gameObject.GetComponent<Rigidbody>().AddForce(transform.forward *
2000f);
        }

        // We destroy the game object immediately after the sound has
finished playing
        Destroy(gameObject, audioSource.clip.length);
    }
}
```



Different from the `OnTriggerEnter()` methods that we've used previously, `OnCollisionEnter()` handles ordinary collisions between objects with primitive colliders, that are not colliders in trigger mode.

In this method, the `other` argument of type `Collision` is an instance of the `Collision` class, which stores information on velocities, `Rigidbody`s, colliders, transform, and contact points involved in a collision. Therefore, here, we simply check the information stored in this argument with an if statement, checking whether the layer of the `GameObject` is `enemyLayerNumber`, and in that case, activate the procedure to stun the hit enemy.

Punching the guards

Having all this prepared will make easy to add this feature to our game.

The `ThirdPersonCustomCharacter` class `FightMethod` takes a parameter which can be 0 (stop fighting), 1 (punch) or 2 (throw stone). For the punch, we will simply trigger the punch animation when the user press the left mouse button, and the collider placed on the `RightHand` `GameObject` will do the rest, thanks to this special class we are going to add to it.

This class will only check collisions and perform its action, only when the object the right hand will collide has the `Enemy` layer:

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;
public class AttackSensor : MonoBehaviour
{
    private int enemyLayerNumber = 12;

    // Check collision with enemy's capsule collider
    void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.layer == enemyLayerNumber)
        {
            float magn = other.relativeVelocity.magnitude;
            if (magn > 0.1f)
            {
                GetComponent<AudioSource>().Play();
                if (other.gameObject.GetComponent<RagdollCharacter>()
                    != null)
                {
                    other.gameObject.SendMessage("ActivateRagdoll");
                    other.gameObject.GetComponent<Rigidbody>
```

```
        ().AddForce(transform.forward * 2000f);
    }
}
}
```

As you can see from the code, upon collision, if the magnitude of collision is enough, will trigger the ragdoll for the hit enemy and stun him for 60 seconds. This magnitude check, will allow to give some dynamics to the fighting, requiring the player to have the space to punch the opponent, when they are stopped without any velocity, the punch will be less powerful, and, in that case, the player will need to run away for then turning and punching the foe properly. Press Play and enjoy testing your new power of throwing stones and assessing punches.

Playing stones collisions feedback sounds

We will use the `hitSound` reference variable to set `AudioSource` clip then play the sound, right after the layer check:

```
if (other.gameObject.layer == enemyLayerNumber)
{
    GetComponent<AudioSource>().clip = hitSound;
    GetComponent<AudioSource>().Play();
}
```

We will use the other audio reference stored in the variables of type `AudioClip` in our `HeavyStone` class to switch the audio clip to the generic collision sound, when the collision happens on generic objects, instead hitting an enemy, so, after the `if(other.gameObject.layer == enemyLayerNumber)` check statement, right after its closing brackets, add:

```
else
{
    GetComponent<AudioSource>().clip = collisionSound;
    GetComponent<AudioSource>().Play();
}
```

Adding force impulse to stones impacts

To add some realism to the action, we want to impress some force to the guard that has been hit, taking into account the force of the stone that hit them. To achieve this, we will simply add the following line of code, after the `ActivateRagdoll` statement we have used in the `OnEnterCollision` methods of both the `HeavyStone` and `AttackSensor` classes:

```
other.gameObject.GetComponent<Rigidbody>().AddForce(transform.forward *  
2000f);
```

Summary

In this chapter, we covered various topics that you will find crucial when creating any game scenario. We looked at implementing `Rigidbody` objects both for animated dynamic elements and instantiated projectiles. This is something you'll likely expand upon in many other game scenarios while working with Unity.

We have also continued the use of instancing from our prototype scene at the start of this book, and effectively created a game mechanic that ties in with the rest of our game, forcing the player to interact with another element in order to gain access to the old man's hut, adding depth to the game, and creating a simple puzzle for the player to solve, put the guards to sleep and find the artifact pieces needed to open the door.

We also took a look at how coroutines can be used to provide structure and added functionality to your scripting, something that you'll definitely need to use again in your future developments. These are all concepts you will continue to use in the rest of this book and in your future Unity projects. In the next chapter, we'll take a break from coding and take a look at more of Unity's aesthetic effects.

We'll explore the use of particle systems to create a custom waterfall to spring into our lake and see ready-made prefabs from Unity Technology in the Standard Assets collection, such as the `DustStorm` prefab and the realistic fire made with more particle systems to add some fire for chimneys and torches.

11

Unity Particle System

In this chapter, we will explore Unity Particle System (code name: Shuriken) to make our 2D game look better, and for making some very cool graphic effects for the 3D game.

For visual effects (VFX) such as moving liquids, rain, smoke, clouds, flames, laser blast, explosions, or magic spells, one or more particle systems can be used to achieve the expected result. We will build special particle effects for the good or bad ending of the 2D game, we will create a fantastic waterfall just over our small lake in our Devil Island, and we will use the ready-made prefab of `Fire` and `DustStorm`, to enrich our environment. In detail, we will be:

- Creating a fireplace using the `Fire-Complex` prefab
- Creating a series of wall torches from a `Fire-Complex` prefab but with some elements removed
- Writing a custom component with a user-friendly Editor UI to move the firelight around
- Creating court dust and sea breeze effects with the help of the `DustStorm` prefab
- Creating a beautiful waterfall made of some customized particle systems that will drop water into our small lake

What is a Particle System?

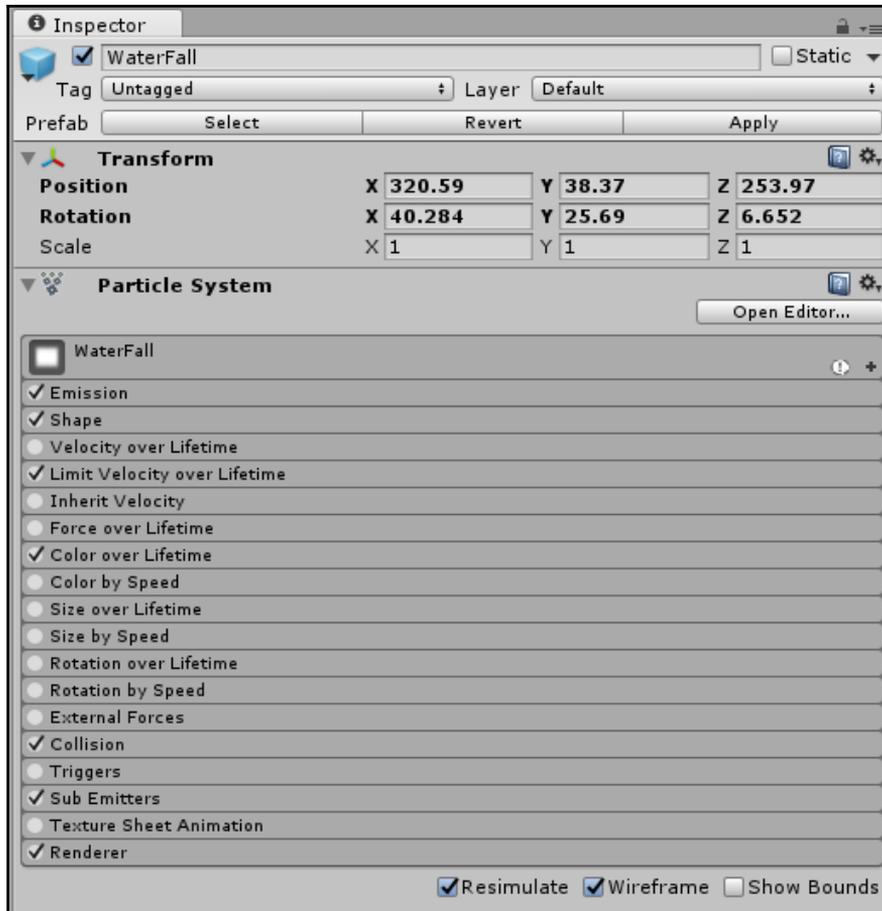
Particles are small images or meshes that are displayed and moved in great number by an emitter. Each particle represents a small portion of the fire, smoke, or water, and the effect of seeing all the particles moving together creates the illusion of the real thing. Using a smoke cloud as an example, each particle would have a small smoke texture resembling a tiny cloud in its own right. When many of these small images are arranged together in an area of the scene, the overall effect is of a larger volume-filling smoke cloud.



Read here for a more detailed overview: <https://docs.unity3d.com/Manual/ParticleSystems.html>.

Unity Particle System is a modular component made of a main module and many submodules. The main module in the following picture contains the main feature of the system. You can expand and collapse the modules by clicking the bar that shows their name for editing their properties. Use the checkbox on the left to enable or disable the module functionality. For example, if you don't want to vary the color of the particles over their lifetime, check off the Color over Lifetime module. Aside from the module bars, the Inspector contains a few other controls.

The **Open Editor...** button shows the options in a separate editor window that also allows you to edit multiple systems at once:



The shuriken particle system component of the waterfall we will build later in this chapter

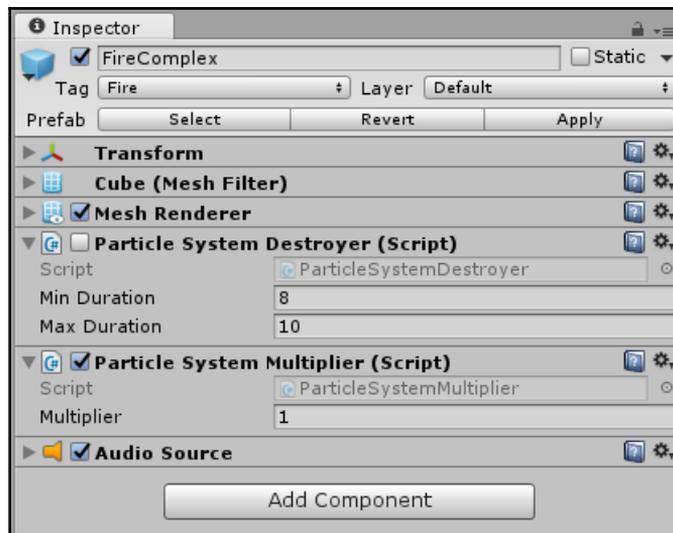
The **Resimulate** checkbox determines whether or not property changes should be applied immediately to particles already generated by the system (the alternative is that existing particles are left as they are and only the new particles will have the changed properties). The **Wireframe** checkbox will show particles geometry in wireframe. **Show Bounds** display the bounding volume around the selected Particle Systems.



You can find more information and a detailed guide of the main module and all the submodules here: <https://docs.unity3d.com/Manual/ParticleSystemMainModule.html>.

Now, if you haven't already, import the `Standard Asset: Particles`, by opening the main menu and going to **Assets | Import | Particle Systems**.

You will notice a new folder within the `Standard Assets` folder in your **Project** view, the `ParticleSystem` folder. This folder contains a `Prefab` subfolder, where you will find some cool ready-made prefabs based on the Shuriken particle system. We will start with the `Fire` complex prefab, which we will be using for making the fireplace and the wall torches. This prefab is called *complex* because it is a set of nested `GameObjects` that contain smoke, fire, glow, sparkles, and flames particles systems meant to work together:



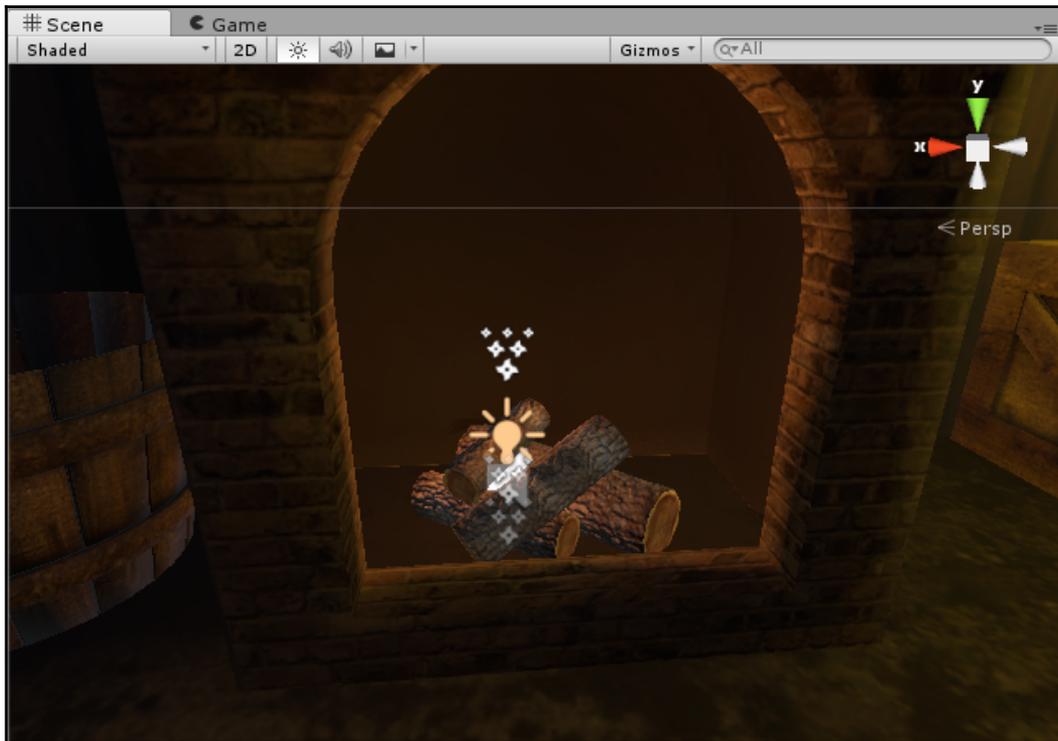
Particle System Destroyer component will stay disabled because we want it to work forever

We will be using the full set for the fireplaces and modifying the smoke position a little, to let it out from the top of the buildings, while we will use a restricted set, without smoke, for the wall torches instead. We will also write our own firelight mover, to animate the point light, and finally, use the smoke alone, on the top of the houses at the *abandoned* village, but first things first, let's build the fireplace!

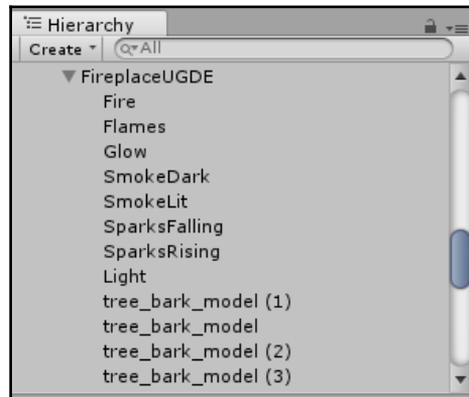
Creating the fireplace

Before you start work on the particle system, we will create a 3D visual for the fireplace.

We will use the `tree_bark_model.fbx` 3D model in the `Assets/Chapter8-9-10/Models/Static` directory. Drag the model prefab into the **Scene** view or in the **Hierarchy** and position it in the old man's hut, inside the fireplace. Scale it properly, to fit in the fireplace. Now clone four or five pieces and scale them a bit differently and also position them and rotate them a bit like in this picture:



Now, multi select all the `tree_bark_model` clones in the **Hierarchy** and drag them under the `FireComplex` **GameObject** as children. In this way, when you will try to move the main object around, you will also move the wood chunks. Verify this by dragging the main object around. Press `Ctrl + Z` to undo your last move. Looking at the **Scene** and **Hierarchy**, you should end with a tree like this:



Even if it seems more convenient to add a convex Mesh Collider and a Rigidbody to the wooden chunks, for them to be lying one on top of the other with the help of physics, unless the player would need some kind of interaction with the bark chunks, a game would never be made in that way. It would use a lot of CPU/PPU cycles, [**Physic Processing Unity (PPU)**] for physics calculations for no real reason. For our purpose, we should patiently position the wood to give them a realistic look.

Now select both the `SmokeDark` and `SmokeLit` GameObjects and move them a little bit higher, by dragging them along the y (green) axis only, on the top of the hut:



The fire got more complex, because it also contains the 3D models of the wooden chunk. Rename the main object into: `FireplaceUGDE` and drag this object from the **Hierarchy** into the `Prefabs` folder in the **Project** view to save the work we have just done.

Press Play to quickly test it, after the game starts, switch into the **Scene** view and focus on some object in the hut with the *F* key. You should see something similar to this picture:



You could also have simply pressed the **Play** button in the Particle System overlay in the **Scene** view, and see how it looks without having to press Play. These controls appear in the scene view when a `GameObject` carrying a particle system is selected. They can be seen, for instance, in the image in the preceding page, where the `Smoke` objects are selected, and the Particle Effect overlay pops up.



To really boot all the particle systems under the main object, because they are on separate `GameObjects`, you will need to select one after another pressing the **Play** button in the Particle System overlay in the **Scene** view to see the overall effect with all the systems running and emitting.

If we are satisfied with the look and the setup of the fireplace, we can go further.

If you made any modifications to the position of the object or on the particle systems setup, go to the top of the `GameObject` in the **Inspector**, and press the **Apply** button.

This will save the last modifications on the stored prefab. Differently, if you feel you touched something you didn't want to, you can go back to the first prefab you saved, by clicking the **Revert** button. The **Select** button, instead, will underline in the **Project View**, the origin prefab that the object in the scene is connected to.

To repeat these actions for a second `FireplaceUGDE` prefab, you drag into the scene from the **Project** view, which will be placed in the fireplace spot of the hut in the village:



The second hut, in the village, where one of the three artifact pieces is hidden

As a last step, we might clone some `Smoke Dark` and `Smoke Light` `GameObject` couples from one of the fires and drag them both as children of some of the chimneys at the top of the houses in the abandoned village to enrich the scene. Press Play and walk around the island with the player and/or check the **Scene** view to see your work spinning!

Creating wall torches

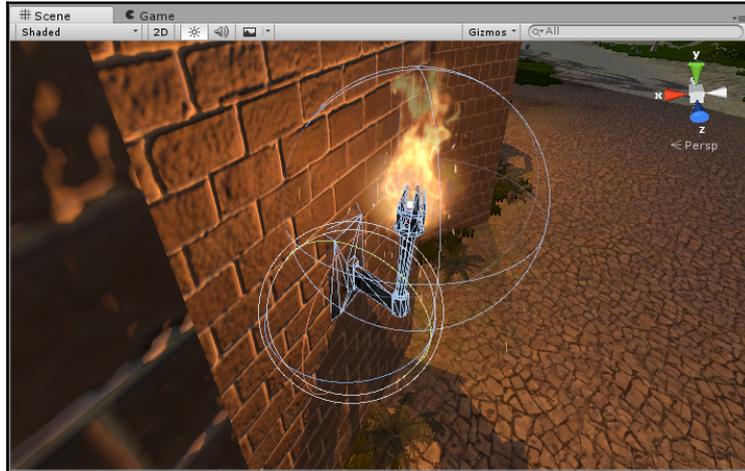
We will build the main origin prefab, then you will drag the stored prefab wherever you feel they might be needed around the scene. First of all, let's drag into scene the `walltorch.fbx` object that resides in this book's codes asset folder:

`Assets/Chapters11-12-13/Models/Static.`

Let's follow these steps:

1. Check the model scale compared to the prison walls and adjust the overall scale in the **Model Import Settings** to fit the correct size for the scene
2. Move the object near the wall

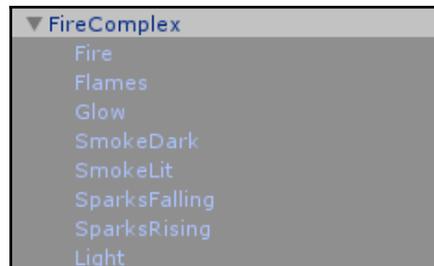
3. Rotate it carefully, aligning the wall torch model to the wall like in the next image:



4. Drag a new FireplaceUGDE prefab in the scene and rename it TorchFireUGDE
5. Delete Smoke Lit and Smoke Dark children objects (answer yes to loose the prefab link when prompted)
6. Make it a child of the torch object
7. Move the TorchFireUGDE object inside the top of the torch like in the next picture:



Now select all the children of the torch, basically the GameObjects that carry a particle system in the **Hierarchy**, like in this screenshot:



Pressing on the **Play** button in the Particle System overlay in the **Scene** view, will show the overall effect in the **Scene** view:



Press Stop and rename your new object as `WallTorch` and save your new prefab by dragging it in the **Project** view `prefabs` folder.

After you do this, you can duplicate the object in scene, or drag a new object. Do so and spread the wall torches in appropriate spots. Save the scene and press Play to see the results by walking around. It should be pretty satisfying, but still, we can do something to make it look even better.

Modifying the Fire Light component

Although we could have done this with RGB and movement animation, we will use what we have inherited from the `FireComplex` prefab to obtain a similar effect with programming. We will practice what we have learned about scripting by extending the `FireLight` component included in the `Particle System Asset` package to suit our needs, but most of all, to enhance the component editor's UI. Let's start by selecting one of the lights in any of the fire torches you just spread around the island, and open the script by double-clicking in the script slot of the attached `FireLight` component. As we can see in the script, there are no parameters that can be tweaked. There are just a couple of private variables. We can easily add depth in the final visual result features for the **Inspector**.

We will modify this script to enable some new parameter tweaking to make the light animation look even better and also add some cool `UnityEditor` UI features. First of all, right before the `Start()` method, add these public variables:

```
using UnityEngine;

////////////////////////////////////
/////
// Modified specifically for Unity 2017 Game Development Essentials codes
examples
////////////////////////////////////
/////
namespace UnityStandardAssets.Effects
{
    public class FireLight : MonoBehaviour
    {
        private float m_Rnd;
        private bool m_Burning = true;
        private Light m_Light;

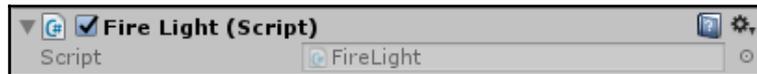
        // Added for UGDE
        public Color startColor = new Color(1f, .45f, 0f);    // Light
        starting color
    }
}
```

```

public Color endColor = new Color(1f, .95f, .55f);    // Light
ending color
public Vector3 mOffset = new Vector3(.5f, 1.5f, .5f); // movement
offset
[Range(0f, 2f)] public float mSpeed = 1f; // movement speed
[Range(0.1f, 1f)] public float mMultiplier = 0.5f; // movement
multiplier

```

This will result in the component changing its visual shape a bit, from a basic component with no public variables:



Into something more complex and user-friendly:



Tweak the component with these values for optimal results in our main scene.

Note: if you change the overall scale of the world and object, these values will not fit any more, of course.

As you can see, the new public color variables will result in two color pickers, very handy for quick tweaking at runtime. The Vector 3 variable has become a three float slots box for specifying the x and z position starting offset for the light movement.

The Movement Speed and Movement Multiplier float public variables have commuted into sliders with the `[Range (min,max)]` directive for additional comfort.

Let's see what we should do in the update method to apply our parameters.

Inside the if (`m_Burning`) check in the `Update` method, we will transform the first line of code:

```
m_Light.intensity = 2 * Mathf.PerlinNoise(m_Rnd + Time.time, m_Rnd + 1 +
Time.time * 1);
```

Into the following code:

```
// Oscillates the light intensity over time
m_Light.intensity = 2*Mathf.PerlinNoise(m_Rnd + Time.time, m_Rnd + 1 +
Time.time*mMultiplier);
```

Then we will add the color oscillation part:

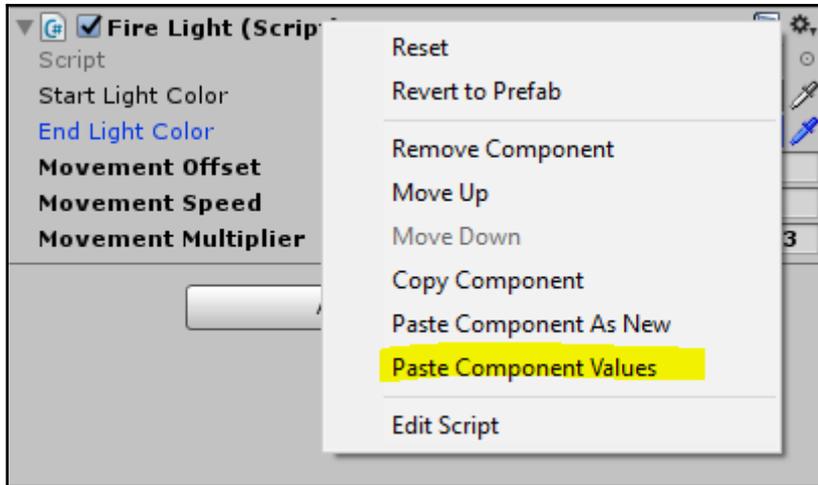
```
// Oscillates the light color between start and end light colours over time
m_Light.color = new Color(
    ( startColor.r + ((endColor.r - startColor.r)*.5f)) +
    (Mathf.Sin(Time.time*3f) *((endColor.r - startColor.r)*.5f)),
    ( startColor.g + ((endColor.g - startColor.g)*.5f)) +
    (Mathf.Sin(Time.time*3f) *((endColor.g - startColor.g)*.5f)),
    ( startColor.b + ((endColor.b - startColor.b)*.5f)) +
    (Mathf.Sin(Time.time*3f) *((endColor.b - startColor.b)*.5f))
);
```

And finally, we will change the other four lines that manage the movement into this:

```
// Move it around by changing the 3 coordinates for the final vector
float x = Mathf.PerlinNoise(m_Rnd + 0 + Time.time* mSpeed, m_Rnd + 1 +
Time.time* mSpeed) - mOffset.x;
float y = Mathf.PerlinNoise(m_Rnd + 2 + Time.time* mSpeed, m_Rnd + 3 +
Time.time* mSpeed) - mOffset.y;
float z = Mathf.PerlinNoise(m_Rnd + 4 + Time.time* mSpeed, m_Rnd + 5 +
Time.time* mSpeed) - mOffset.z;
// Apply finally the movement on the transform local position
transform.localPosition = Vector3.up + new Vector3(x, y, z) * mMultiplier;
```

This enables our movement multiplier and our movement speed parameter.

You can also tweak values at runtime, until the wanted effect is obtained. Then, before you hit Stop (the component parameters would return to their original values), you can Copy the component, by right-clicking the component title in the inspector, then after Stop, you will right-click the component you want to update and choose **Paste Component Values** as shown in the next screenshot:



Component default context menu with the Paste Component Values option(highlighted)

Enhancing environment ambience

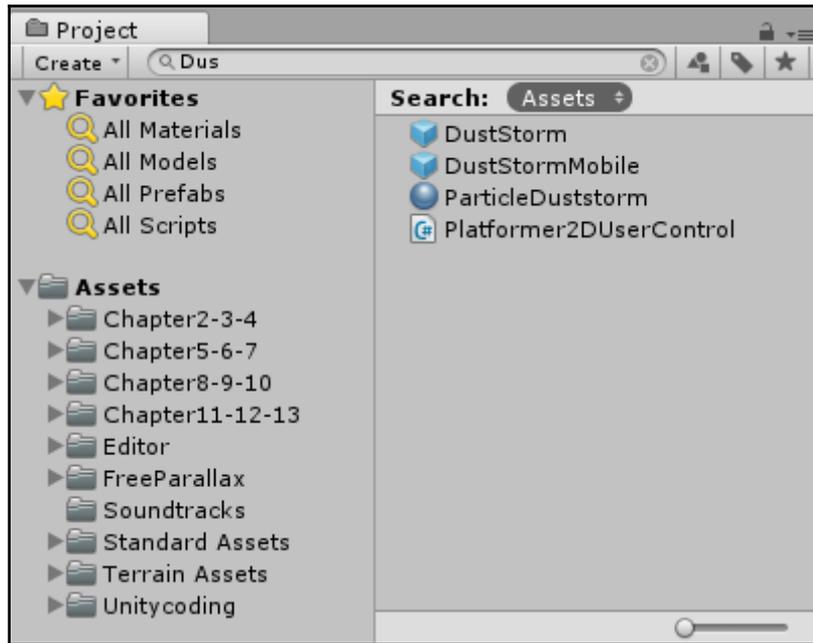
Now we will use the cool ready-made `DustStorm` prefab in the Standard Assets Particle System we have just imported, to create some cool ambient. We will create a very light sand/dust floor cloud in the village and we will use the same prefab, slightly customized, to obtain something like sea shore breeze.

Making the village ground dusty

How much better would the village court look if the sandy ground could *emit* some dusty cloud? It would definitely give more ambience and depth to that part of the scene.

We will quickly implement this with the help of the `DustStorm` ready-made particle system prefab, in the `Standard Assets/particle system/prefabs` folder.

We will drag the `DustStorm` prefab in the scene from the Standard Assets quickly this time by searching in the **Project** view for it, just type the letters `Dus` and it should come out in the search box. Choose the non-mobile version of it and drag it in the scene:



We will just need to customize the size of the emitter area to cover the zone we want to see with this effect. After you dragged in the `DustStorm` prefab scene, in the main module, halve the **max particle count** from 1000 to 500.

Open the Emission module and set the **Emission rate** to 50, then open the Shape module, choose Box shape, and set the box size of the area to something that will cover the village court surface. We will finish like in the preceding picture.

Now, move the `DustStorm` `GameObject` to the first part of the village like in the following picture. Try to cover the whole village area as well as the beach. We also need to increase the `Box` shape size to `70, 0, 70`:



You should see the square that represents the area of emission, drawn with white lines, when you expand the `Shape` submodule

We will now tweak the particle system a bit by doing the following in the main module:

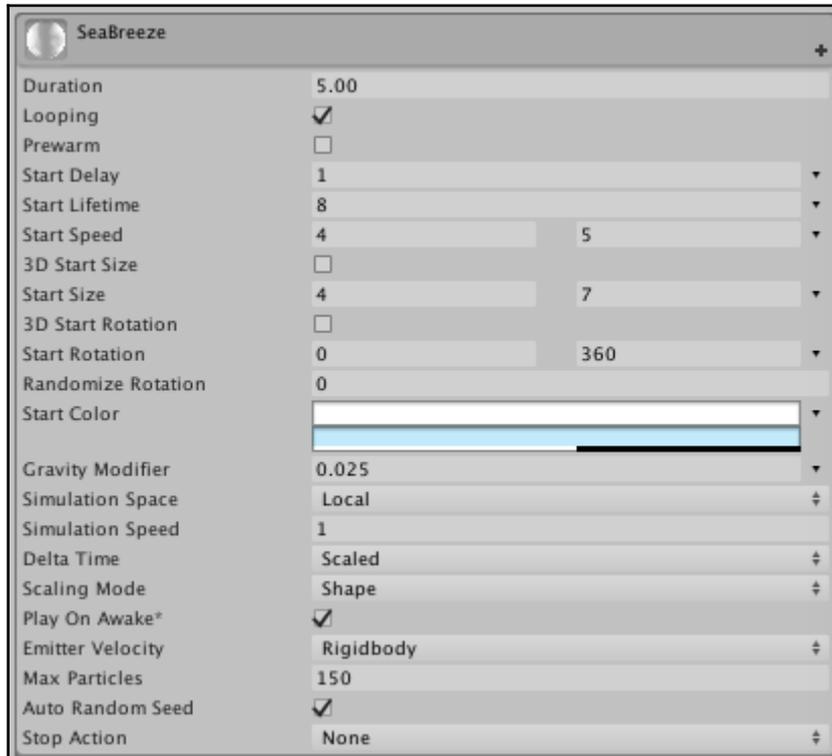
- Reducing the **start speed** to `0.5 - 1`
- Setting the **Gravity modifier** to `-0.01`
- Checking the **3D Start Rotation** checkbox

We will increase the minimum particle size in the `renderer` sub module to `0.1`.

Save the scene and press **Play** button in the Particle System overlay in the **Scene** view, and observe how the FX is performing and behaving in the village court, and eventually tweak until you reach a satisfying result.

Creating a sea breeze particle system

Drag a new `DustStorm` prefab in the scene and rename it `SeaBreeze`. To be able to save the new tweaks and changes and to be sure to create a new prefab and do not overwrite the `DustStorm` prefab, drag again from the **Hierarchy** to the `Project` view the new `Prefab`. Let's edit its basic settings, in the main module, set it up like in this image:

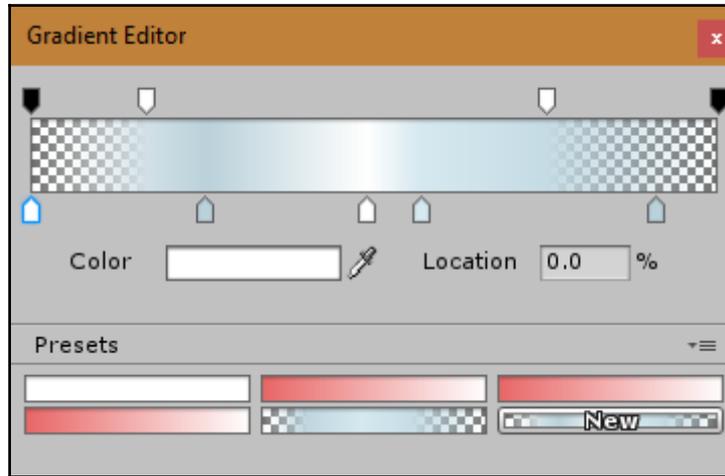


In the Emission module, put 50 for the **Rate** and leave the **Time** setting for the rate. In the Shape module, choose Box shape and set 60, 0, 10, for its **Size**. Press Play and you will see, as soon as you start to walk the hero out of the jail, that what was the `DustStorm` prefab is something different now.

Everything isn't done yet. We will also change the Color over Lifetime module by tweaking the gradient according to the new FX we are doing, we will click the small arrow on the right side and choose **Random Between Two Gradients**. Then, we can choose a previously saved gradient preset or edit your own, using the Gradient Editor.

The Gradient Editor

The **Gradient Editor** allows you to quickly edit a gradient made of more break points with more colors involved. It also managed the alpha transparency through the **Location** percentage value. After you have made your gradient you can quickly create a new preset by clicking on the **New** button:



The Gradient Editor with our new gradients for simulating the sea breeze water sprays

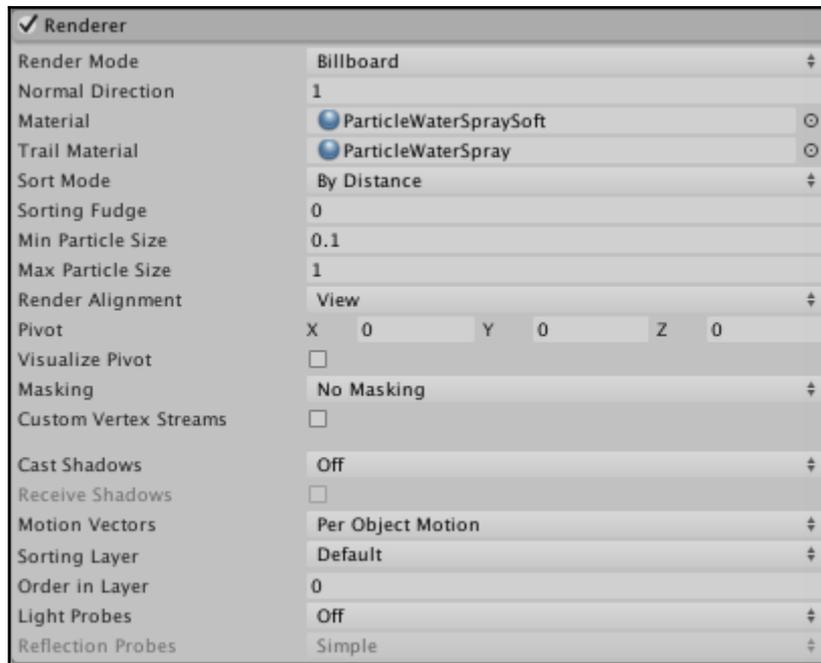
Create two slightly different gradients and choose them as the two gradients in the **Color** over Lifetime sub module. We will finish with something like in this picture:



We are going now to tweak the renderer sub module of the particle system, to tweak the visual aspect of the individual particle and give a better overall feeling to the seashore water breeze clouds:

- Change the actual material into the `ParticleWaterSpraySoft` material
- Set `0.1` instead of `0` in the minimum particle size

The final look of the first part of the Renderer submodule will look like the next picture:



The Particle System Renderer module settings for the SeaBreeze vfx

The last step will be to position and rotate the `SeaBreeze` `GameObject` in a way that will satisfy the final look. We could have used collisions at World level for each particle through the Collisions module, and that would look wonderful, but this is way too expensive for a game in most of the situations:



The `SeaBreeze` `gameObject` particle system simulating and showing the box shaped emission area

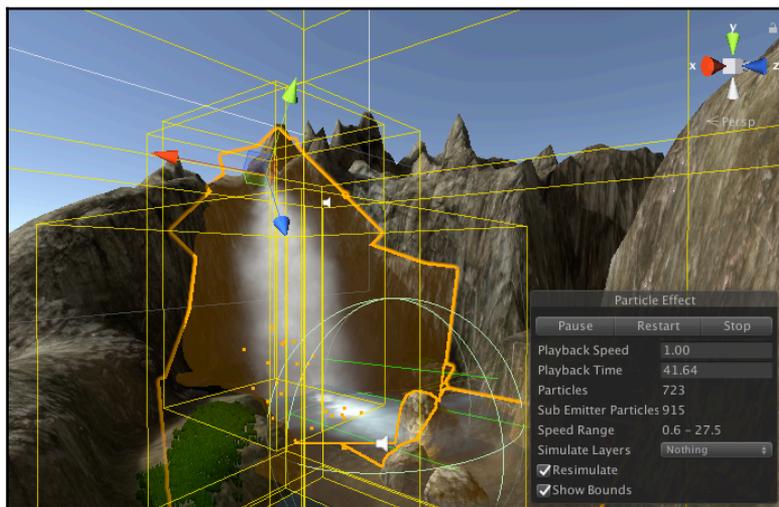
We can finally test our Particle System with the **Play** button in the **Scene** view, and see its overall behaviour in the **Scene** view, of course, consider the fact that the player camera will see the particles from a different angle, so here, hit Play and test the game to see the final result.

Creating a waterfall

We want to create a waterfall that will spring water from the top of the mountain over our small lake, and also create some effect for the water falling into the lake. To implement this we will use the ready-made Unity Standards Assets Hose that we can find in the prefabs folder. This prefab simulates the fire protection water hose. In fact, the main `GameObject` is an empty `GameObject` with two scripts: `hose.cs`, responsible for giving variable force to the water output, and `simpleMouseRotator.cs`, which enables the user to give a direction to the water flux with the mouse. Now, drag the `Hose` prefab into the scene. As we don't need the mouse and the power controls over the natural waterfall, we will parent out the `WaterShower` `GameObject` child, rename it `WaterFall`, and delete the parent, the `Hose` `GameObject`.

To quickly position it near the lake, we can use this four step shortcut:

1. Select one of the rocky big stones we positioned earlier inside the lake
2. Press the `F` key on the keyboard to focus **Scene** view camera on this object
3. Select the `Waterfall` `GameObject` in the **Hierarchy**
4. Press `Alt + Shift + F` to quickly move the selected `GameObject` into **Scene** view:

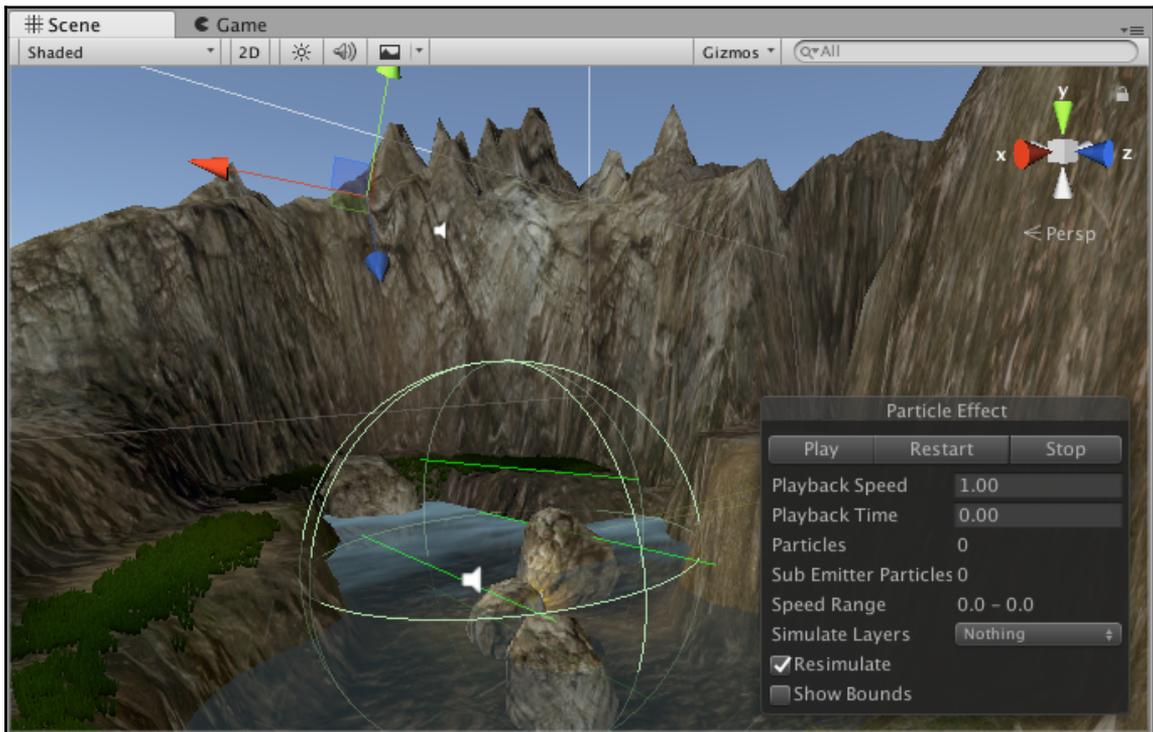


This quick four-steps trick will save you a lot of time when building or designing your game level because it can be quickly driven with keyboard shortcuts.

Now slightly move, using the move tool, the waterfall GameObject on the top of the mountain, like in the preceding picture.

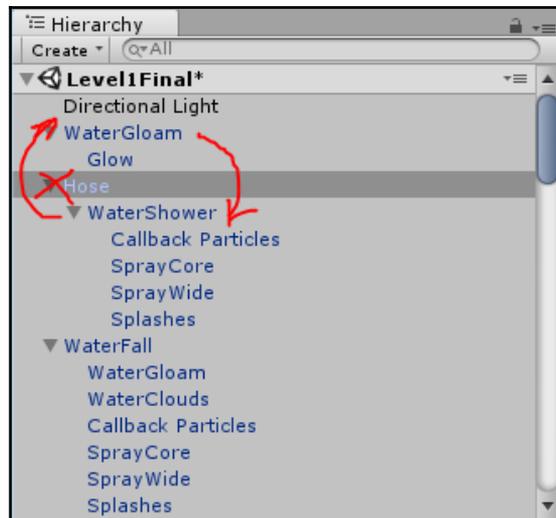
Press the **Play** button in the Particle System overlay in the **Scene** view, to see where the waterfall is directing the water spring. Adjust the rotation of the GameObject to fit your design needs (that is, with the emitter looking almost to the ground).

When you are happy with the position, Save your scene and proceed with the next step:



We will combine this cool prefab with another particle system that we are going to customize to implement the movement effect for the water falling in the lake, the *Afterburner* prefab from the Standard Assets Particles System package. So now drag it in the scene and rename it `WaterGloam`.

Make it a child of the `Waterfall` object:



Quick scheme to illustrate the steps just described

Move it lower, dragging the y axis arrow down, until it will be at water level.

Press the **Play** button in the Particle System overlay in the **Scene** view, to see the effect on the water from above. When you are happy with the result, save your scene and press Play to test it with the hero, or by going directly to the spot in the **Scene** view.

Summary

In this chapter we learned how to build some cool particle effects and how to customize the particle system for our needs. We learned about how multiple particle systems can play together and how to combine the same effect for different situations. We modified and adjusted the `FireLight` component, for moving around and changing the color/intensity of our fire *point lights*.

We also built our own Waterfall with the help of the `Hose` prefab, a cool Unity Asset, and by extending it with an additional modified particle system, the `Afterburner`.

In the next chapter, we will make our own menu for pausing/resuming or quitting the game, and a scene loader component that uses the new `SceneManager` namespace.

12

Designing Menus with Unity UI

In order to create a well-rounded example game, in this chapter, we will look at creating a separate scene to enable our existing scene to act as the main menu. In this chapter, you will learn the following things:

- New UI basic concepts: canvas and event system
- Understanding the main difference and usage of the three types of canvas available
- Preparing images as sprites to work with new Unity UI
- Loading scenes to navigate menus and loading the game levels
- Creating a simple audio options menu with music and general volume controls
- Creating a simple video options menu with general quality control
- Writing a listener script for UI Slider elements and exploring Dynamic Variables
- Learning the basics of post-processing image effects
- Creating an in-game/pause menu feature with World Space Canvas
- Sorting out rendering artifacts and depth bugs concerning World Space Canvas

This book will not go through a lot of examples of the existing game menus. Considering that fashion and tendencies change fast, I have personally seen a lot of AAA amazing games with poor main menus.

This seems to be the recent trend in many modern games. As a player, I usually tend to spend just a few seconds in the main menu and hours on the real game. Personally, I prefer a great game with a poor menu over a boring game with an astounding menu.

We will try to keep things as easy as possible and give some atmosphere to our Main Menu. We'll use the main character's 3D animated model and some special FX, and then apply some cool camera post-processing image effects to the final rendering. We will learn the basics of UI and experiment with this feature while designing our game menu. Since Unity version 4.6, this is the way to implement HUDS, Menus, and UI, in general this process has significantly improved. If you read the first edition of this book for Unity version 3.x, you will find a totally different system to implement user interfaces.

The older approach, still available for backward compatibility until the current 2017.x version cycle, was generally the more widespread method of creating full game menus, as it gave the developer more flexibility. The menu items themselves were established in the script, but styled using GUI skins in an approach comparable to HTML and CSS development in web design—the **Cascading Style Sheets (CSS)** controlling the form with the HTML providing the content—in this example, the GUI is the HTML and the GUI skin is the CSS.

This approach is more flexible but less performing and consumes much more video memory. Since 2D texture assets are not batched into a single atlas, this results in a lot of draw calls for the CPU, especially when you plan to port or design your game for mobile, web, or other low-end platforms.

Unity UI

Unity UI system that comes built-in with the editor since version Unity 4.6 takes advantage of a complex and powerful event system with a similar approach to other third parties 3D user interfaces extensions such as EZGUI or NGUI. Using Unity UI is really handy because it performs well when sprites are packed into texture atlases, it already supports multiple video resolutions and is cross-platform ready.



Historic consideration

Techniques for making in-game HUD and user interfaces changed a while ago, in 2003, when DirectX9 definitively abandoned the old raster software 2D rendering library (`DirectDraw`) that was used in conjunction with the `Direct3D` library and unified all the graphics under the `DirectGraphics` library. This provided a huge benefit to the performance and eventually engines and games evolved in this direction. Everything from buttons to images, text, menus, and HUD is made with a texture mapped on a simple quad, made of two triangles. If you are an experienced user who follows unity at least since version 3, you will surely remember the EZGUI or the NGUI Asset Store packages. Those libraries were introducing what Unity has built-in since version 4.6 and onward: a fully-fledged UI system based on 3D objects meant to work in a 2D coordinate system.

Unity UI works with two main objects: `Canvas` and `EventSystem`. While the `Canvas` will render the UI object, is used to arrange elements, and represents our screen, the `EventSystem` component is responsible for mouse and keyboard input (on PC/Mac/Linux) and on other input for other platforms. When you add any UI `GameObject` to the scene, a `Canvas` as well as an `EventSystem` `GameObjects` will be automatically created if not already present in the **Hierarchy**. While you can have more than one `Canvas` `GameObject` in a scene, only one `EventSystem` is necessary for the system to work.

Canvas render modes

There are three different render modes for the `Canvas` meant for different types of UIs, two are meant to work at screen-space level and will ignore the position and size of the `Canvas` in the 3D space, while one is meant for diegetic UIs.

Let's look at them quickly.

Screen space – overlay

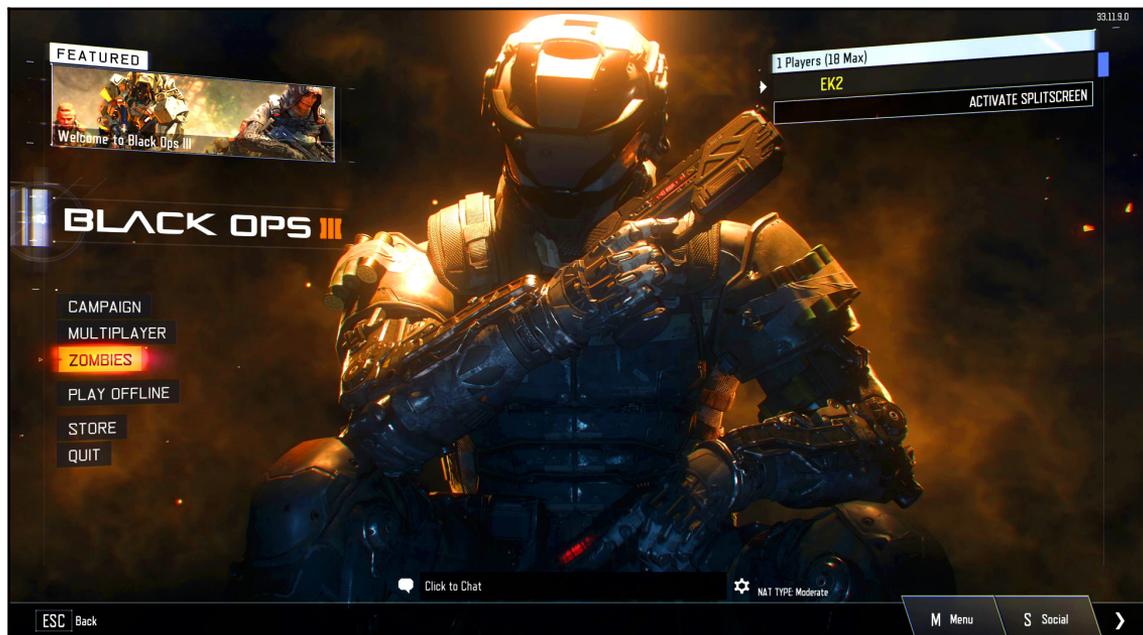
We already looked at the generic rules for a Screen Space Overlay canvas that we used in-game to render the HUD. With that render mode, UI elements placed on the screen are rendered on top of the scene. If the screen is resized or if it changes resolution, the Canvas will automatically adapt. We will use this kind of the Canvas for the game title in the main menu, and we are already using it for the in-game HUD:



An example of an overlay menu with a series of graphic 3D elements in the background, Deus Ex Human Revolution (Square Enix, 2011)

Screen space – camera

This is similar to Screen Space Overlay, but, in this render mode, the Canvas is placed a given distance in front of a specified camera. The UI elements are rendered by this camera, which means that the camera settings affect the appearance of the UI. If the camera is set to perspective, the UI elements will be rendered with perspective, and the amount of perspective distortion can be controlled by the camera field of view. If the screen is resized or changes resolution, or the camera frustum changes, the Canvas will automatically change size to match. This kind of camera allows us to have panels that rotate a bit on their y axis, and that's the main reason we are using this for our game instead of the previous method:



An example of the screen-space camera render type for canvas that would be needed to achieve this sort of menu, Call of Duty: Black Ops III (Activision , 2015)

World space

In this render mode, the Canvas will behave just like any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be part of the world. This is also known as a *diegetic interface*. This approach allows you to have the menus made with real 3D objects in the scene (for example, street, planks of a wooden wall, or whatever you can come up with). We will use it in a different manner though, for the in-game menu, just to practice, because our game doesn't actually need a diegetic interface. Diegetic comes from an ancient Greek word, *diegesis*, which means *narration*. While in cinema and music, diegetic means something a little different, in modern video games, a diegetic UI gives info to the user from the 3D world itself, and not in an overlay, hence, it requires a World Space Canvas to implement it:

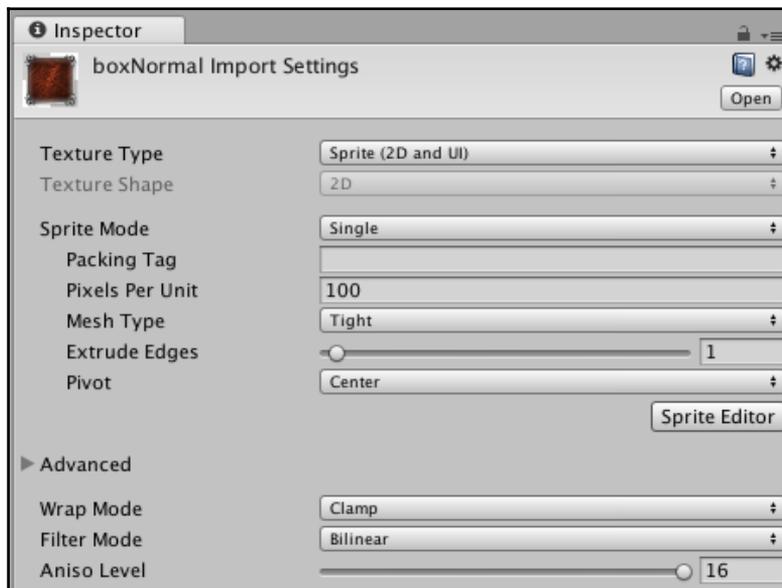


Far Cry 2 (Ubisoft, 2008) is an example of a good diegetic interface, where the UI renders on real 3D models in the scene

For more information about Canvas, head to <https://docs.unity3d.com/Manual/UICanvas.html>.

Preparing textures for UI usage

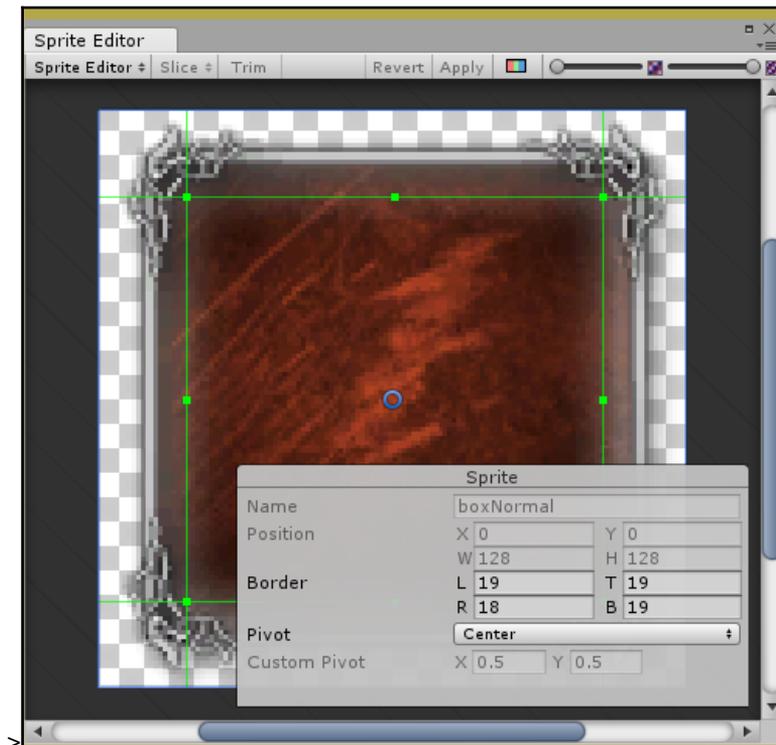
Like in [Chapter 3, Creating and Setting Game Assets](#), and [Chapter 4, Inspector and Scripting - Coding the Player Controller](#), when we import the images as Sprite/UI instead of as textures, the UI components all work with the same import setting: 2D Sprite/UI:



Let's import all the images in the `Chapters11-12-13/UI` folder in this way so that they are ready to work with Unity UI.

We will also need to edit the slicing crop for those images with the Unity **Sprite Editor**.

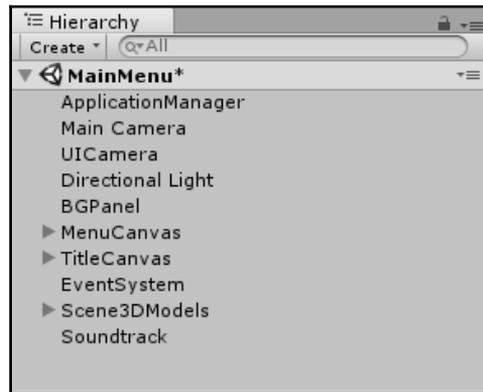
To do so, select one of the images in the **Project** view. Then, look in the import settings in the **Inspector**, and you will find the **Sprite Editor** button just before the advanced import settings section:



We will also set the import settings for the `cursor.png` file to `cursor` instead, set the application mouse cursor to this image, and leave the UI uncompressed for the game icon and splash images. We are going to look at image compression in more depth in Chapter 14, *Building and Sharing*.

Creating the main menu scene

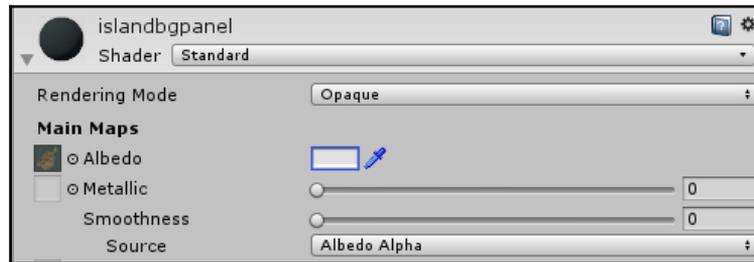
We will start by creating a new blank scene from the main menu, **File | New Scene**. If prompted, save the scene you were working on. In this simple scene, we will have just two UI Canvases, some additional 3D models, a **Main Camera**, a backdrop panel, and a **Directional Light**. At the end of this section, we will have a scene hierarchy similar to this one:



Let's create the 3D scene we want to keep in the background:

1. Drag a `FireTorch` prefab from the previous chapter into the scene, clone another one from it, and rotate one of them so that they are opposite each other.
2. Drag the `DarkWinds` audio clip, in the `Chapters3-4/audio/soundtracks` folder, into the scene (automatically, a `GameObject` carrying the `Audio Source` component with that `Clip` assigned will be created by Unity).
3. Check the **PlayOnAwake** and **Loop** checkboxes on the `Audio Source` component).
4. Create a 3D plane from the editor main menu, **GameObject | 3D | Plane**, and create a new material to apply to it with the standard Unity shader. Rename it `islandbgpanel`, and position it at `1, 1, 5.43`.

5. Drag the `islandbg.png` texture, which resides in the same folder (Chapters11-12-13/UI), on to the **Albedo** slot of the material, or press the tiny circle button, choose it from the assets explorer, and set color to 225,225,225:



The islandbgpanel material seen in the Inspector

6. Drag our model hero t-pose base character rig without animation into the scene
7. Create a new simple Animator Controller to control the hero's animations
8. Add an idle state in the state machine and specify the `Warrior_Idle` motion clip for it
9. Position the Main Camera at `0.5, 1, -1.5`
10. Position the `Warrior_RM_final` GameObject transform at `0.778, -0.333, 0.331` and rotate it 180 degrees on the *y* (Up) axis
11. Ensure that the Main camera has an **Field Of View (FOV)** set to 25
12. Ensure that the Main camera has set `0.01/10` as the near/far values for the clip planes of the frustum cone



You can create an empty GameObject called `3Dscene` to hold the 3D scene objects for commodity.

This will set up our 3D part of the scene, let's now dive into the UI part by looking into Unity's UI features and capabilities, as well as how to manage UI with scripting.

Pressing Play in the editor should show something like this:



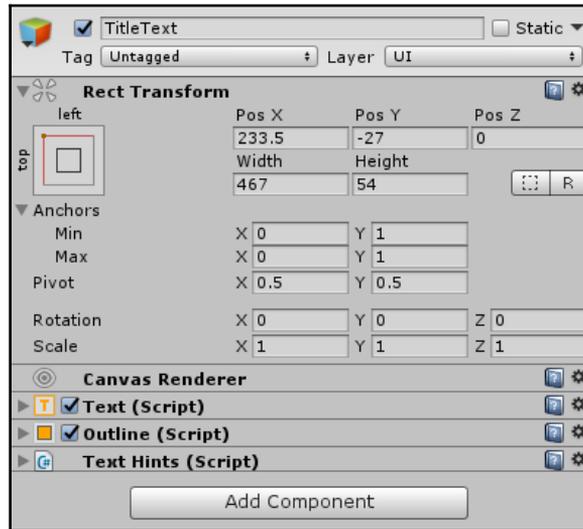
Finally, we will add a background soundtrack to play in a loop in the scene: drag the `Soundtrack` audio clip from the **Project** view directly in the **Hierarchy**. Unity will create a `GameObject` with the same name on the audio clip with an `Audio Source` component attached and that audio clip specified automatically.

Just check the `Loop` option in the **Inspector** for the `Audio Source` component on this `GameObject`, save the scene, and test that it plays and loops correctly.

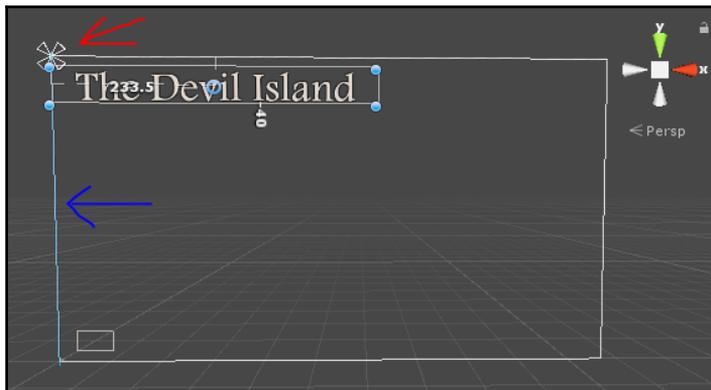
Adding the game title

We will add our first `Canvas`, which will be responsible for rendering `Title Text`, and we want this title to stay in the top-left corner and resize with the resolution. For this reason, when we add the `Text UI` object from the main menu, or by right-clicking the hierarchy, a `Canvas` containing it will automatically be created for you. The `Canvas` will be set as a `Screen Space Overlay` render mode canvas. Rename the canvas in `TitleCanvas`. Let's type `The Devil Island` in the `Title Text` component.

Then, to be able to see the UI Text GameObject handle, the pivot and the anchor points visually in the scene editor, you must ensure that you have the **Rect Transform** component expanded in the **Inspector**:



Now, using the **Rect tool**, align and anchor the Text component in the Canvas by dragging the Text GameObject to the top-left corner until you see the sides of the canvas turning blue:





You can also use the arrow keys to move the UI object.

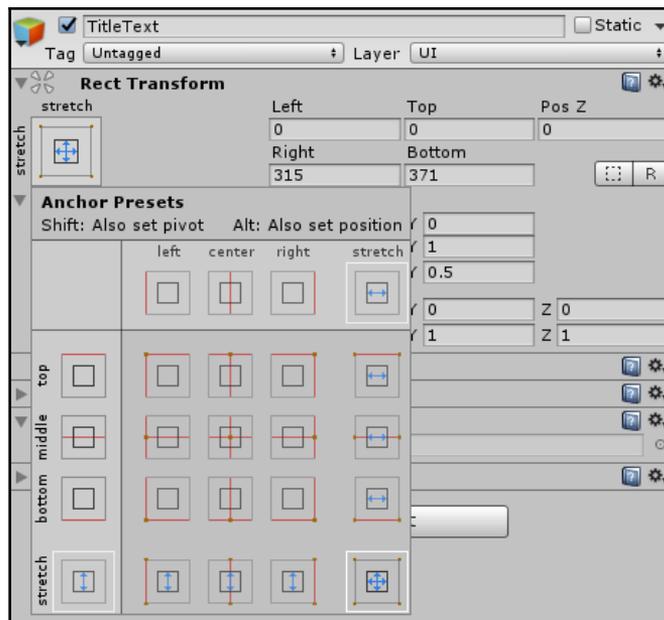
When positioning the title `GameObject` in the Canvas, we make the final touches by adding a UI Text effect; select the `TitleText` `GameObject`, and then go to top menu: **Component** | **UI** | **Effects** | **Outline**. We should see a result similar to the preceding screenshot.



The Rect tool selected in the main toolbar

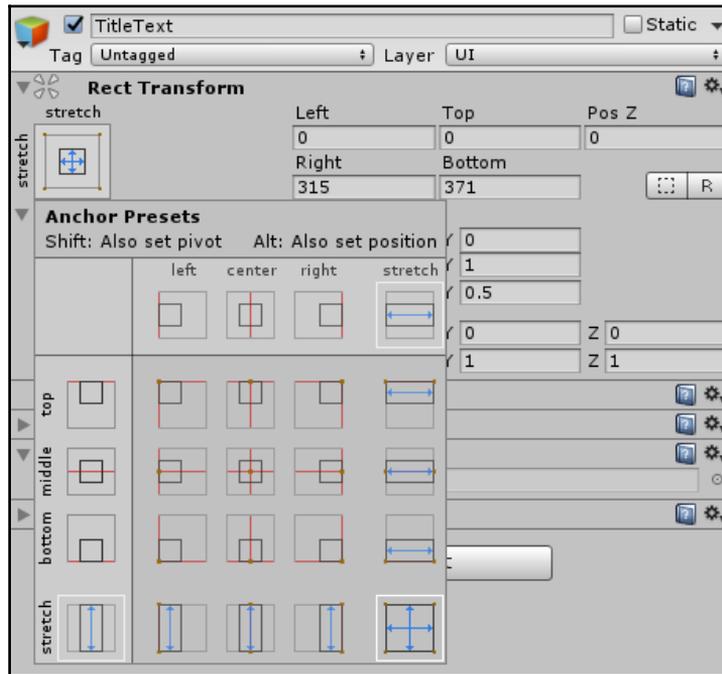
Manually anchoring the title to the Canvas

To anchor the UI `Text` `GameObject` title, we can use the **Scene** view handles visible when you have the 2D tool enabled and a UI object selected, or you can use the Anchor presets table shown in the **Inspector**, obtained by clicking the stretch button on the Rect Transform component on the object:



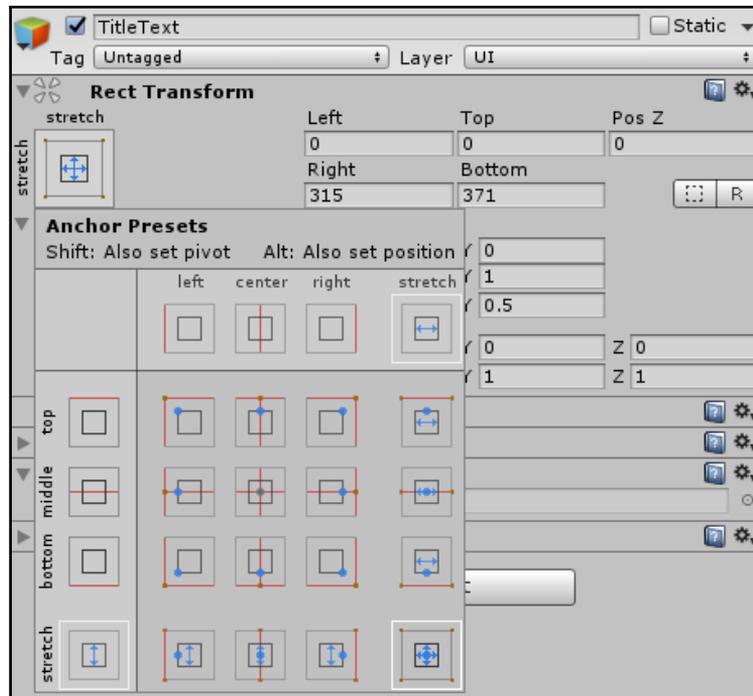
If you are not comfortable auto-setting these properties by dragging the element around the canvas, you could always set the same pivot, anchors, and alignment properties manually.

By holding the *Alt* key, you will be able to set the position of the anchor:



This tool will give you maximum flexibility to arrange your controls within a Panel or a Panel within a Canvas. You can always make the same adjustments directly in the scene, by using the **Scene** view handles and 2D pivots. This widget is very similar to other IDE for other frameworks and is quite handy and to the point.

Holding the *Shift* key, you will also be able to set the Pivot position that determines the anchor spot:



Let's take care of populating this Canvas and creating the first menu panel and the first buttons.

Creating the main menu panel

We will use a secondary Canvas with a secondary camera for rendering the main menu to avoid rendering artifacts and conflicts with Camera Image Post Effects. For this reason, we will add a new Canvas with a Screen Space Camera selected in the Render Mode, and we will rename it `MenuCanvas`. This will be the main canvas where all the menu panels will be contained. Then, we create a UI Panel for holding button controls and such by right-clicking the `MenuCanvas` `GameObject` in the hierarchy, and choose **UI | Panel**. This will create a panel as a child.

A UI Panel is nothing more than a `GameObject` with a `Rect Transform` component and a `UI Image` component, which can also be left empty (in the case we just want a virtual box holder and not a real displayed box).

Adding buttons

Before we proceed, to simplify the visual, we will rotate the main panel back to zero degrees so that it is 100% flat. Instead of using the main menu to create the buttons, we will right-click the **MainMenu** panel object in the **Hierarchy**, and add **UI | Button**. As you already know, a UI Button `GameObject` is made by a `UI Button` component `GameObject` with a `UI Text` component `GameObject` as its child. Place this first button on the top of the panel:

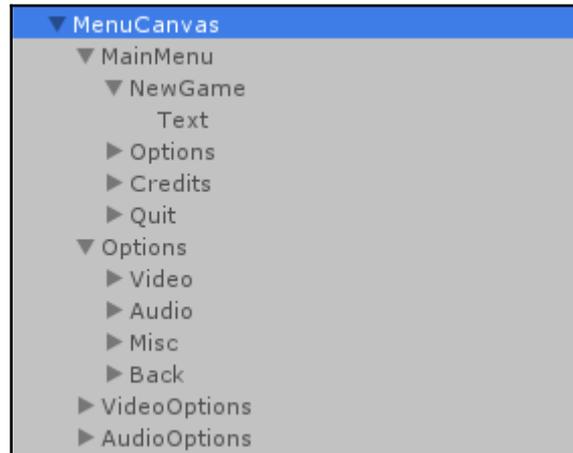


As the button element is a child of the main panel, you will see anchors appearing while you drag around the button, lock it on the top anchor, and you will not even need to set their anchor properties like we just did for the `Title` `GameObject` within the `TitleCanvas`. Repeat the option for the other three buttons, rename their labels respectively: **New Game**, **Options**, **Credits**, and **Quit**.

Clone the MainMenu to obtain the Options menu

We will clone the `MainMenu` `GameObject` along with its children and rename the clone `GameObject` to `Options`.

We will change the buttons: New Game, Options, Credits Text component string respectively into **Audio**, **Video**, **Game**, and the **Quit** button with Back. We should end with a scene **Hierarchy**, such as this:



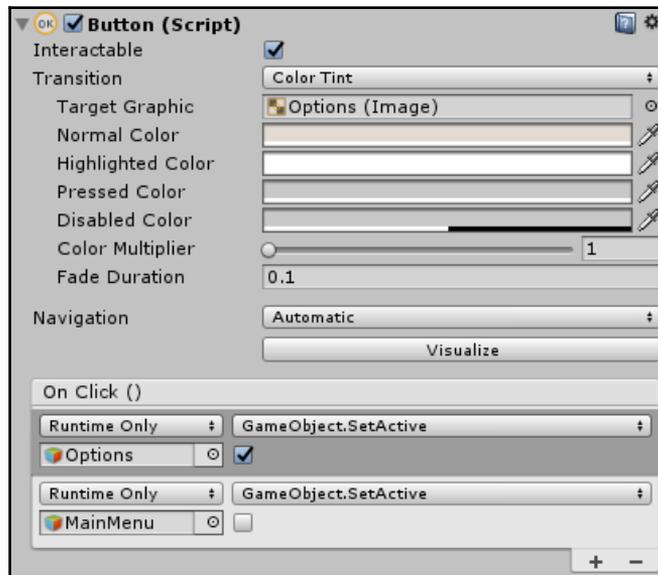
We will now add the code and the **Inspector** settings necessary to let New Game start, for managing Video, Audio, and the Options submenu as well as for the **Quit** button functionality.

Configuring UI buttons to show/hide menus with the `OnClick()` event method

To drive the buttons to open a new menu panel and hide the previous one and so forth, we will just use the UI `OnClick()` event method found on the inspector for the Button component.

For example: to show the Options menu and hide the `MainMenu` GameObjects, we will set these two actions for the `OnClick()` event method of the Button component of the `Options` GameObject child of the `MainMenu`.

Look at this image:



For the **Back** button in the Options panel, we will make a similar thing, but inverting the activation of the two GameObjects panels; this will deactivate the `Options` GameObject and activate the `MainMenu` GameObject to bring it in back in view.



It's important to leave the children controls under a panel as Active in the **Inspector** and deactivate the parent panel GameObject. This will insure that when a panel is shown, all the controls will be active and available for the user.

Creating an audio options menu

Select the `MenuCanvas` GameObject and add a new UI Panel GameObject to it, like we have just done for other panels, and add the usual background image for its Image component. Rename the GameObject to `AudioOptions`.



Alternatively, you can do what we have done for the `OptionsMenu` and clone that panel then delete the unwanted controls in it, and then rename the clone in `AudioOptions`.

To keep things simple, we will just add two simple sliders to control **Music Volume** and **General Volume**. For learning purposes, we will use two different techniques for managing the two sliders.

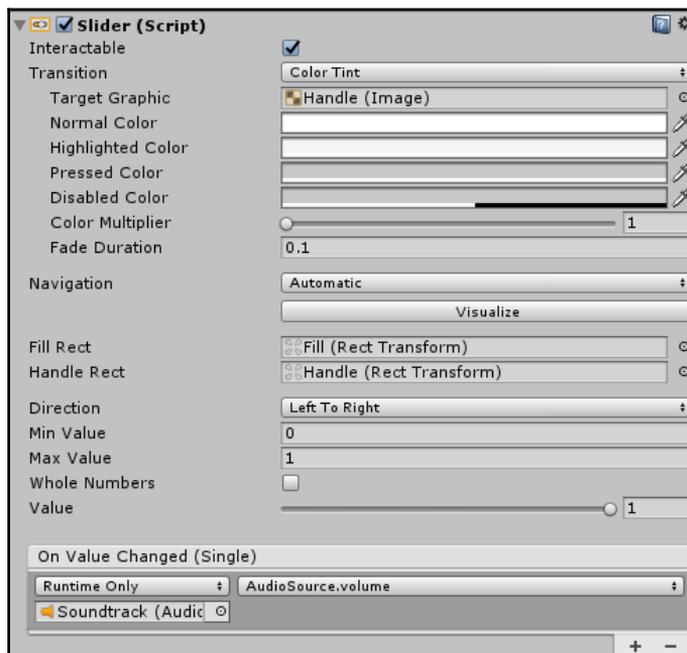
Adding music volume control

We will add a `Slider` `GameObject` as a child to `AudioOptions` by right-clicking the `AudioOptions` `GameObject` and choosing **UI | Slider** from the drop-down menu.

We will then set 0 for min value and 1 for the max value, and leave deselected the **Whole Numbers** option to use float number in the range of 0.0 to 1.0. The Value slider can be set there and will change while our UI drives the change.

For this slider, we will make it super simple: we will specify the `Soundtrack` `GameObject` in the **On Value Changed (Single)** method and choose from the available object list: `AudioSource` and `volume` for its property.

Every time the slider will change its value by user input, the background music will adapt its volume accordingly:



The Slider component in the Inspector with the Soundtrack audio clip GameObject specified in the On Value Change event slot and the AudioSource.volume call set

Adding general volume control

We will add another `Slider` GameObject to the **AudioOptions** panel **Hierarchy** by right-clicking the panel and choosing **UI | Slider**.

In the Text we will type `General Volume`, and for the implementation we will try something different from what we did for the `Music Volume`.



Beware: don't add the slider GameObjects from the Editor's top menu. Even if the parent is selected, the Slider will be created outside, resulting in a wrong relative position/rotation. This will force you to fix the values in the **Inspector** after you have manually dragged the slider under the parent panel. Instead, selecting the parent and right-clicking to create the Slider from the context menu is the correct choice as it is already parented to the panel with the correct orientation and position.

Writing a listener script for UI slider elements

To practice scripting and see the power of the UI system, we will control this slider in a different way.

Instead of adding the `OnValueChanged` listener in the **Inspector**, like we did for the **Music Volume** slider (see the preceding image), we will add the listener by code at runtime when the `Start()` method is called. This listener will trigger a custom function whenever the slider changes to set the volume of the general Audio Listener, which is always placed on the Main Camera.

Select the **General Volume** slider GameObject and add a new component with the button beneath the last component in the **Inspector**. Call the new behavior `VolumeHandler.cs`.

We will write the following code:

```
using UnityEngine;
using UnityEngine.UI;
[RequireComponent(typeof(Slider))]
public class VolumeHandler : MonoBehaviour
{
    // Use this for initialization
    void Start ()
```

```
{
    // Adding an event listener on the slider UI component
    GetComponent<Slider>().onValueChanged.AddListener(SetVolume);
}

// method called by the listener when the slider changes
void SetVolume(float volume)
{
    AudioListener.volume = volume;
}

// we need to override the OnDestroy monobehavior method to remove the
listener once this object is destroyed
void OnDestroy()
{
    GetComponent<Slider>().onValueChanged.RemoveListener(SetVolume);
}
}
```

As you may recall from the earlier chapters, the `[RequireComponent]` directive ensures that the `GameObject` we are attaching this script to contains a `UI Slider` component.

For the general volume, instead of acting on the volumes of all the `AudioSource`'s in the scene (and in the game), we will just act on the `AudioListener` volume, ensuring that the output is made louder or for all the active `AudioSource` components.

The single line of code on the `Start` method adds an event listener with code. This is the same technique as specified in the `Slider` UI component in the Inspector. We have already seen some actions for the `OnChange` event in the NPC dialogue-making back in Chapter 8, *AI, NPC, and Further Scripting*.



We could have used a similar technique to the one we used for controlling music volume, by dragging the main camera in the slot and controlling `AudioListener.volume` from there.

For compactness and usability, we will manage all the sliders by code, inside an `ApplicationManager` class, and, in the next chapter, we will extend this class to be able to save the preferences to the local drive and to be able to restore them when a user starts the game again. Through this class, we will also manage asynchronous scene loading with a dynamic progress bar and some of the menu logic.

For the **Back** button, we will use the same technique used for switching the menus: add two actions to the Button component and put the **AudioOptions** panel and the **Options** panel inside it. The former will be hidden and the latter will be shown when the user clicks the button.

User interaction

To manage all the other controls, we will create an `ApplicationManager` class, to be able to catch and drive user interactions as well as manage global aspects of the application.

This class will be able to manage sliders and toggles, drop-down menus, and scene loading, as well as storing the user settings to registry keys (on Windows) through the `PlayerPrefs` Unity API. We will start writing the code with the creation of the **VideoOptions** menu.

Let's dive into this class and the building of the **VideoOptions** menu.

Creating a video options menu

Until now, we didn't need a class for managing the whole game, even if you already noticed that the regulated audio volume settings are not kept when you quit the game or stop the editor.

First of all, to allow the application to save the audio settings and the video settings before quitting, and also to manage scene loading, the progress bar from asynchronous scene load, and all other video options panel UI controls, we will write an `ApplicationManager` class. This class will take care of managing all the UI controls in the menu as well as loading the game level and quit the program with the `Application.Quit()` instruction written inside a custom public `ApplicationQuit` method that we will add to our class.

In the next chapter the same class will be used to restore the settings in the game level.

Note that the `Application.Quit()` instruction is ignored in the editor. Because of this, if we want to manage the action assigned to the **Quit** button while we are testing in the editor, we will need to add some pre-processor directives.

For more info about pre-processor directives and their use, refer to the next chapter, and for a full directives list, visit <https://docs.unity3d.com/Manual/PlatformDependentCompilation.html>.

In this case, we will add: `#if UNITY_EDITOR`, `#else`, and `#endif` to manage both cases (build/editor), and put the editor on pause, instead of just doing nothing when in the editor:

```
// This method will be called by the UI quit button
public void ApplicationQuit()
{
    PlayerPrefs.SetInt("QualitySettingsIndex",
QualitySettings.GetQualityLevel());
    #if UNITY_EDITOR
    EditorApplication.isPlaying = false;
    #else
    Application.Quit();
    #endif
}
```

This method must not be confused with `OnApplicationQuit`.

When the application quits because the user killed the process, or closed the window, or because your code launched the `Application.Quit()` method, the `OnApplicationQuit` event is triggered and sent to all `GameObjects` before the application quits.



Hence, we might have written the `PlayerPrefs` code inside an `OnApplicationQuit` method, if we don't have any function in our app, to let the user close the application with a button of the UI. This is the case in many mobile platforms, where closing the application should be left to the user with the OS way to do it, instead of closing it from within the app. In the editor, this is also called when the user stops play mode.

The following code is the code for the `ApplicationManager` class fully commented:

```
// Unity Engine main API namespace
using UnityEngine;
// Add this namespace to access SceneManager
using UnityEngine.SceneManagement;
// For accessing the UI API
using UnityEngine.UI;
// For accessing coroutines IEnumerator type
using System.Collections;
// For accessing the Standard Assets post-processing Image Effects from
scripting
using UnityStandardAssets.ImageEffects;
// For accessing the UnityEditor API
#if UNITY_EDITOR
using UnityEditor;
#endif
```

```
public class ApplicationManager : MonoBehaviour {

    public GameObject MenuCamera;
    public GameObject mainMenu;
    public GameObject title;

    private Toggle PostEffectsToggle;

    void Start () {
        Application.targetFrameRate = 400;
        StartCoroutine( FadeIn() );
    }

    IEnumerator FadeIn()
    {
        while (true)
        {
            Camera.main.GetComponent<VignetteAndChromaticAberration>
                ().intensity -= 0.01f;
            if
                (Camera.main.GetComponent<VignetteAndChromaticAberration>
                    ().intensity <= 0.349f)
            {
                mainMenu.SetActive(true);
                title.SetActive(true);
                break;
            }
            yield return new WaitForSeconds(0.001f);
        }
    }

    public void LevelLoader(int levelnum)    {
        SceneManager.LoadScene(levelnum);
    }

    // This method will be called by the UI quit button
    public void ApplicationQuit()
    {
        PlayerPrefs.SetInt("QualitySettingsIndex",
            QualitySettings.GetQualityLevel());
        #if UNITY_EDITOR
            EditorApplication.isPlaying = false;// instead we stop the
            playmode if this is played from the editor
        #endif
    }
}
```

```

    #else
    Application.Quit();           // Quit the application
    #endif
}
}

```

Add this component to a new empty `GameObject` that we will name `ApplicationManager`. We will now clone the `AudioOptions` panel and rename it in `VideoOptions`, then we will delete all the buttons labels and controls contained in the copy, but we can keep the **Quit** button and rename its text label to **Back**:

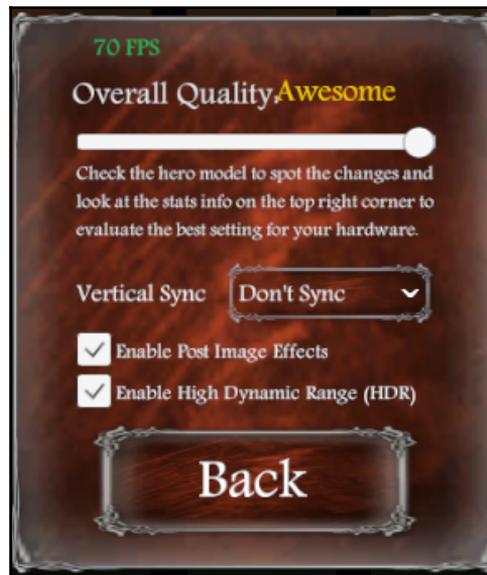


The `VideoOptions` panel with a `Text` label, a `Slider` the first of the two toggles

With the `VideoOptions` `GameObject` selected, perform the following steps:

1. Add a **UI | Text** control to display actual frames per second. This will be useful for users to check the performance of their hardware with the Unity engine at different quality levels.
2. Add one new **UI | Slider** control for managing the **Quality Settings** and two toggle controls for managing the **HDR** and **PostFX** settings.
3. Add a **UI | Toggle** to let the user enable/disable Image Effects.
4. Add a **UI | Toggle** to let the user enable/disable **High Dynamic Range (HDR)**.
5. Create a **UI | Dropdown** to let the user choose the desired vertical sync.

This is how the panel will look with all the controls in place:



The power of UI dynamic variables and UI events

Unity UI can use dynamic variables. Dynamic variables are variables declared public in your classes of type String, Int or Float that will be exposed in the **Inspector** for accessing them with UI components events for managing UI logic. Let's see how to use them for managing our graphic settings in the **VideoOptions** panel.

First of all, we will add two public methods to set what we need in `QualitySettings.vSyncCount` and the main camera `allowHDR` property (settings that allow High Dynamic Range rendering), with a float input parameter for the former:

```
public void SetVerticalSync(int param) {
    QualitySettings.vSyncCount = param;
}
```

And a Boolean variable that can be true or false for the latter:

```
public void SetHDRenabled(bool flag) {
    Camera.main.allowHDR= flag;
}
```

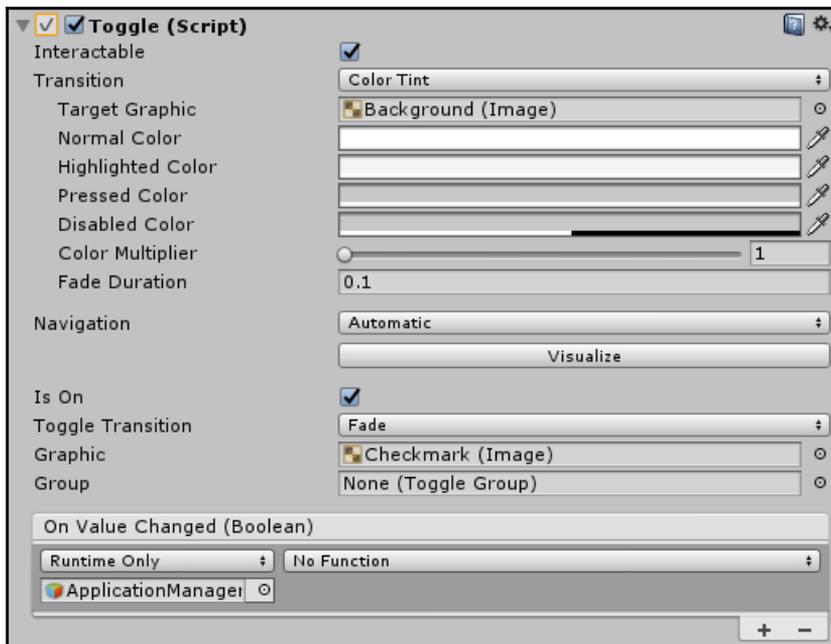
It gets a little more complicated for the post FX switch code, because the camera may contain several different Image Effects components, and it would be tricky to iterate all the different types and check whether they exist attached to the main camera.

This is how we can iterate all of them and set them as enabled or not, according to the parameter we will pass to them.

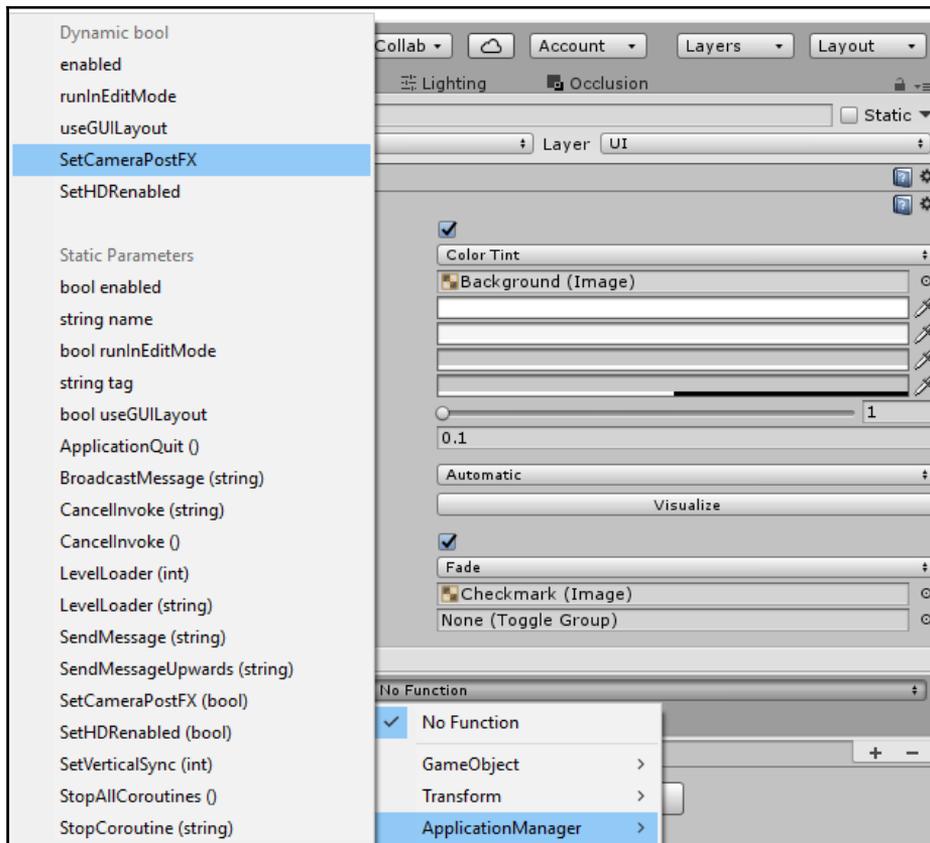
The `SetCameraPostFX` method will get the Boolean input parameter flag and apply it to each post effect present on the main camera with this simple `foreach` statement:

```
public void SetCameraPostFX(bool flag) {
    PostEffectsBase[] CameraEffects =
    Camera.main.GetComponents<PostEffectsBase>();
    foreach (PostEffectsBase postfx in CameraEffects)
    {
        postfx.enabled = flag;
    }
}
```

On the `postfx Toggle` component in the inspector, we will drag the `ApplicationManager` `GameObject` to be able to access its public methods:



By declaring this method public with a `bool` input parameter, this will be seen by the UI event system as a dynamic Boolean. Hence, we can select our Toggle GameObject, and in the component in the **Inspector** set its `OnChange` event to **ApplicationManager's** `SetCameraPostFX` method, like in the following screenshot:



The flag parameter will be set directly when the Toggle component value changes and is passed to the `SetCameraPostFx` method

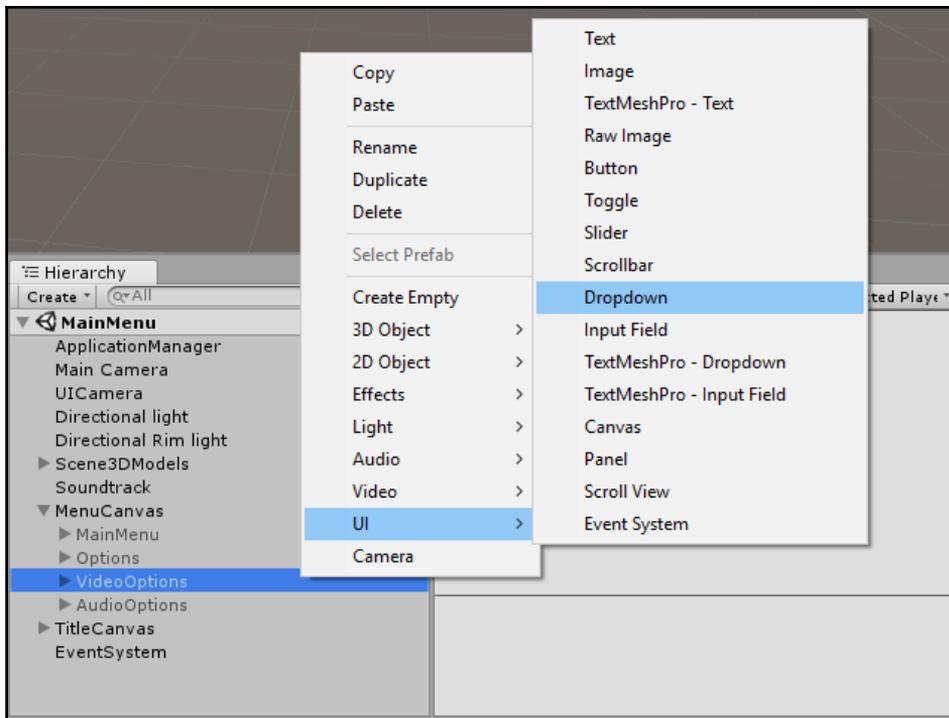
Creating a drop-down menu

For managing the `vSynch` settings instead, which is not just two values, 0/1 (`true/false`), but can be 0, 1, or 2, we will use a drop-down menu control. The three values `vSynch` can assume mean:

- No vertical synch (all the frames the computer can do)

- Double buffer (this caps the frame rate to 60 frames per second)
- Triple buffer (this caps the frame rate to 30 frames per second)

To create a drop-down menu, select the `VideoOptions` `GameObject`, right-click it, and choose **UI | Dropdown**:

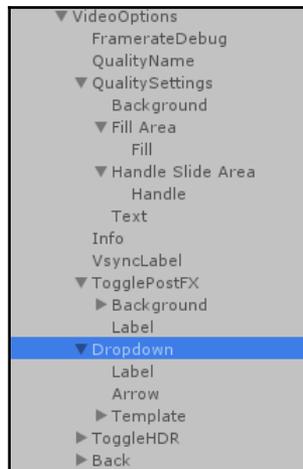


Yet another slider controller

We will implement this in the quickest way possible to enable the game to save user preferences later in the next chapter.

Create a new UI Slider `GameObject` and rename it: `QualitySettings`.

The final scene **Hierarchy** of the `VideoOptions` menu expanded in the **Hierarchy** window:



The `QualitySlider` class attached to the `QualitySettings` `GameObject` will take care of listening to the **Quality Settings** slider changes, as well as changing the label text with the current quality name:

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Slider))]
public class QualitySlider : MonoBehaviour {

    public Text QualityNameLabel;

    // Use this for initialization
    void Start()
    {
        // Adding listener on the slider UI component
        GetComponent<Slider>().onValueChanged.AddListener(SetQuality);
    }

    void SetQuality(float val)
    {
        Debug.Log("change into:" + (int)val);
        QualitySettings.SetQualityLevel((int)val);
        QualityNameLabel.text =
            QualitySettings.names[QualitySettings.GetQualityLevel()];
        // Save the preferences
        PlayerPrefs.SetInt("QualitySettingsIndex",
```

```

QualitySettings.GetQualityLevel());
// Read the Vsync from preferences and over rides preset
settings
QualitySettings.vSyncCount = PlayerPrefs.GetInt("vSyncCount");
}

void OnDestroy()
{
    GetComponent<Slider>().onValueChanged.RemoveListener(SetQuality);
}
}

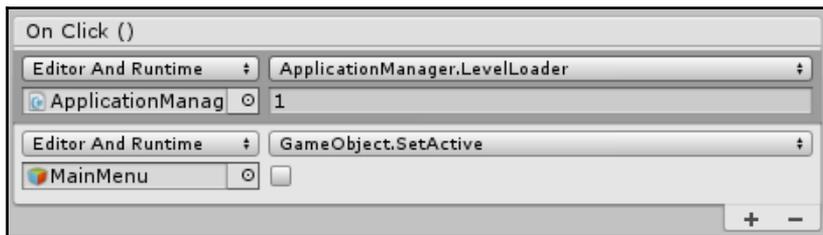
```

Loading the game

We will use the `OnClick()` event of the `Button` component attached to the `NewGame` `GameObject` to start the loading of the game level scene.

Before we do so, we need to add a public method to call from the UI in our `ApplicationManager` class.

Select the `NewGame` `GameObject` in the hierarchy, then click the `+` button twice to create two event actions. We will drag the `ApplicationManager` `GameObject` into the first slot. Unity will automatically recognize the `ApplicationManager` monobehavior attached to this object, and in the second pull-down, you will find and choose the `LevelLoader` method (which takes an integer for parameter) and, after having done so, put `1` in the value field:



The first action will call the `LevelLoader(int levelnum)` method on our class and initiate the loading of the scene number specified in the parameter. In the second action we will specify the `MainMenu` `GameObject`, and `SetActive(false)` on the right, to hide the menu when the load starts.

In our case, we will see in Chapter 14, *Building and Sharing*, the indices of the scenes are used by Unity for loading them by simply specifying their index number. Slower but still effective, you can still use the string name of a scene to load it.

Final touches

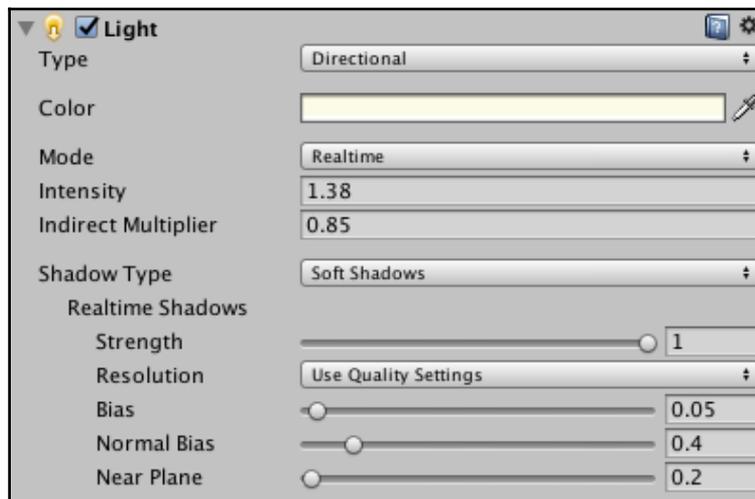
We will take care about the final galore for the main menu scene by properly lighting the scene and adding Realtime Soft Shadows, as well as adding the **HDR (High Dynamic Range)** option on the Main Camera and some cool post effects.

We will also explore the technique to split the rendering of UI elements on another camera to avoid visual issues with the use of the post effects on the Main Camera.

Lights and shadows

First, we will slightly change the default white color into a very light yellow: 251,251,233. Then, we will ensure that soft shadows are enabled on the directional light in the scene.

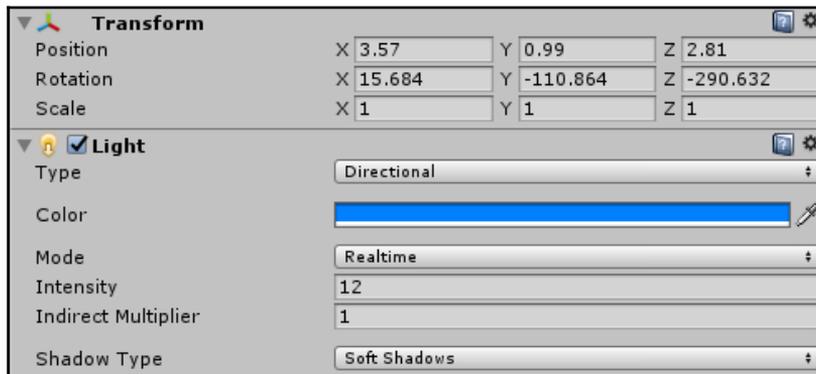
For the **Intensity** and **Bounce Intensity** values, we will set 1.38 and 0.85, and leave all the rest untouched, like in this screenshot of the **Inspector**:



We want to add more atmosphere to the scene so we will try to work with an additional Rim Light.

Adding a Rim light

We will add also another additional **Directional** light and rename it: `Rim light`. We will assign a light blue/purple color to it and we will rotate it for looking from the bottom to the top with a very large cone angle. Boosting the **Intensity** to 12 and enabling real-time shadows on this light will help to create more atmosphere and give that cool, artistic look:



In photography, Rim Light is a lighting technique where the subject image is backlit, and the image is exposed to hide the features in shadow. The technique gets its name from the fact that lighting a subject in this way produces a thin line or *rim* of light that appears to cling to the subject's outline. Using Rim Light lifts the subject from the background in images rendered predominantly in shadow. In more complex situations, using extremely technical lighting setups, Rim Light can be applied to one area of an otherwise well lit image, for example to the hair of a model being shot against a dark background.

Real-time shadows

We will switch on shadows on both of the Directional Lights in our menu scene, to add some more realism. Set the Light component's **Shadow Type** parameters as shown in the previous image for both of the directional lights. Take your time in tweaking the shadow strength and light intensity.

Finally, we are ready to add a combination of Post Processing Image Effect components to the main camera.

Post-processing image effects

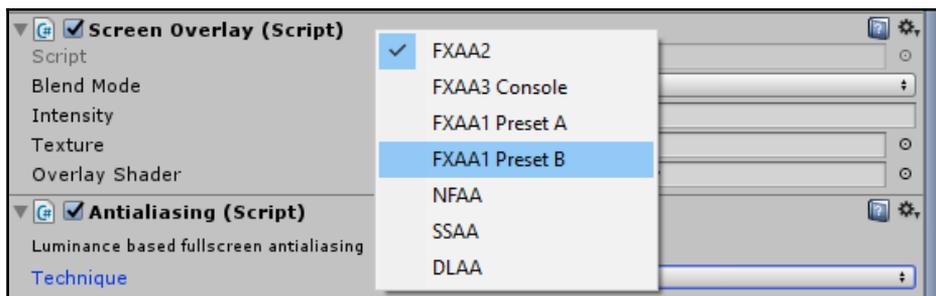
Image effects are components that are added to the main camera to obtain a deeper feel against the *flat look* that old games pre-**Shader Language Model (SML)** used to have. Let's import the Effects Unity Standard package from the top menu: **Assets | Import**.

To add depth and an original look to our menu, we will use some post-processing image effects for the camera.

These effects will be a subset of the same bunch of effects we will apply to the main camera in the game. In particular, Global Fog and Depth Of Field effects will not be used in the main menu scene. Instead, our subset will be made of four effects: **Full Screen Anti Aliasing (FSAA)**, which runs much faster than hardware **Multi Sample Anti-Aliasing (MSAA)**, Optimized Bloom, Screen Overlay, and Vignette and Chromatic Aberration.

FSAA (Full Screen Anti-Aliasing)

This effect is meant to substitute (older) hardware MSAA Anti-aliasing. Anti-aliasing is an image processing algorithm used to remove the pixelation from the lines. In short, it makes the final image look much better:



the Anti-aliasing component series of presets for the technique used to obtain it

There are many different FSAA techniques you can choose from the component drop-down menu; the first ones are the most CPU-intensive, so, while the FXAA2 is the hardest to compute/better looking, the DLAA is the quickest but less nice-looking. FXAA3 is meant for console platforms, so you should ignore it when building for PC/MAC/Linux or the mobile platforms.

Bloom and HDR

The bloom and optimized bloom image effect is a cool post-render effect that brings up the whites in the final image on certain conditions. Some good examples are shiny metal reflections, light reflections in general, and sun-burning effects.

Screen overlay

The screen overlay image effect is, simply put, a fullscreen image overlay that is drawn on the top of everything using a blending method that can be specified. In our case, we will use a small gradient texture and choose Multiply in the blending options.

Vignette and chromatic aberration

There are many uses for this image effect, which is a combo of vignetting and chromatic aberration effects. For our purposes, we will only use the vignette effect for fading in/out of the screen in a cheap (in terms of efforts) but solid way. We will use this method for fading in/out the scene in our `ApplicationManager` class in the next chapter.

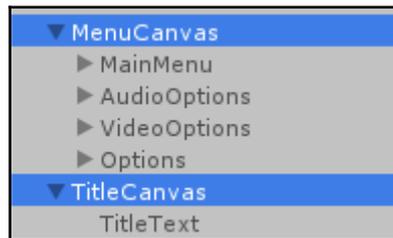
We will also take a look at the new Post Processing Stack, an optimized version of the Post Processing Image Effects that batches all the render passes required for all the effects into a single one to maximize the performance.

Splitting the render on two different cameras

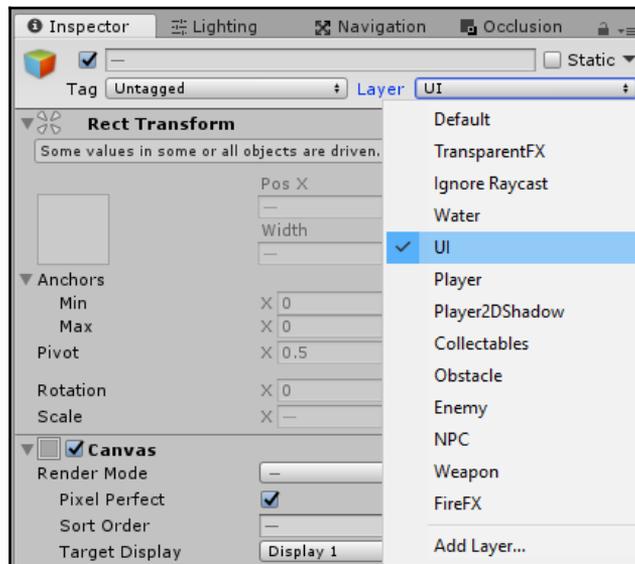
To avoid visual issues with the camera effects and also for performance purposes, it might be useful to split the render of a UI Canvas and its children elements to be rendered by a secondary camera. This technique will use the `layer` set on `GameObjects` to filter out the rendering on a specific camera through the **Culling Mask** setting.

First of all, we will create a new camera from the top menu, `GameObject | Camera`, and renaming it `UI camera`.

Now we will make sure that both the Canvas we have created and all of their children have the **UI** layer. Select both the `Menu` and `TitleCanvas` objects in the **Hierarchy** with the help of the *Ctrl* key (*Cmd* key on the Mac):



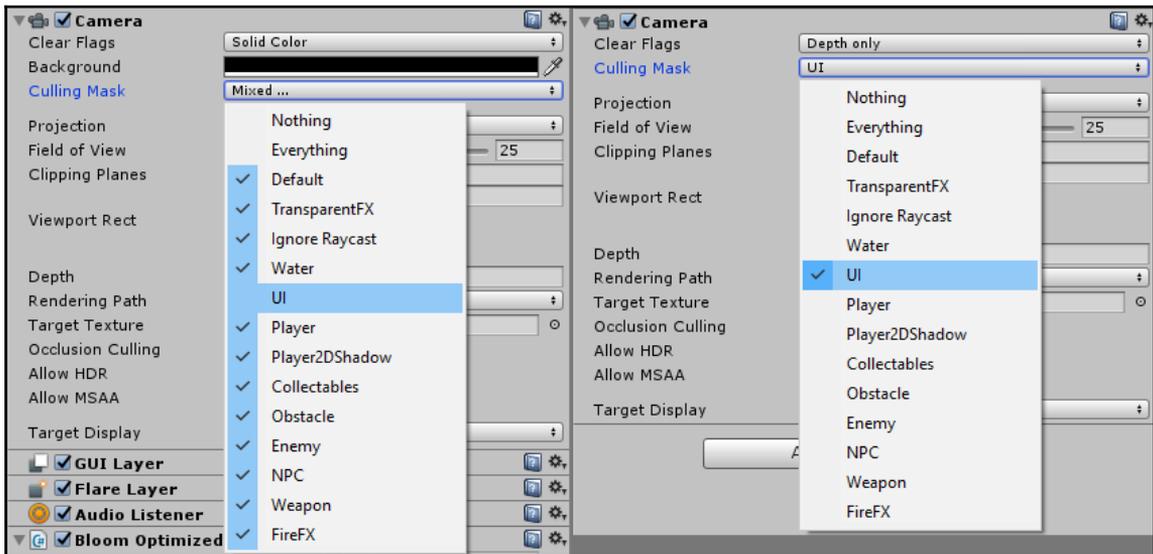
In the **Inspector**, change the layer to **UI** and, when asked, answer Yes for all the children:



Now, the final touches would be to edit both the Main Camera and UI Camera Culling Mask settings.

For the Main Camera, we will click the **UI** layer in the **Culling Mask** drop-down menu list to exclude it. For the **UI Camera**, we will choose first **Nothing** from the list, to exclude all the layers, then we will click **UI** to add it.

In the next image, you can see both the cameras in the Inspector for a quick comparison of the different **Culling Mask** settings for each of them, on the left the Main Camera, on the right the UI Camera:



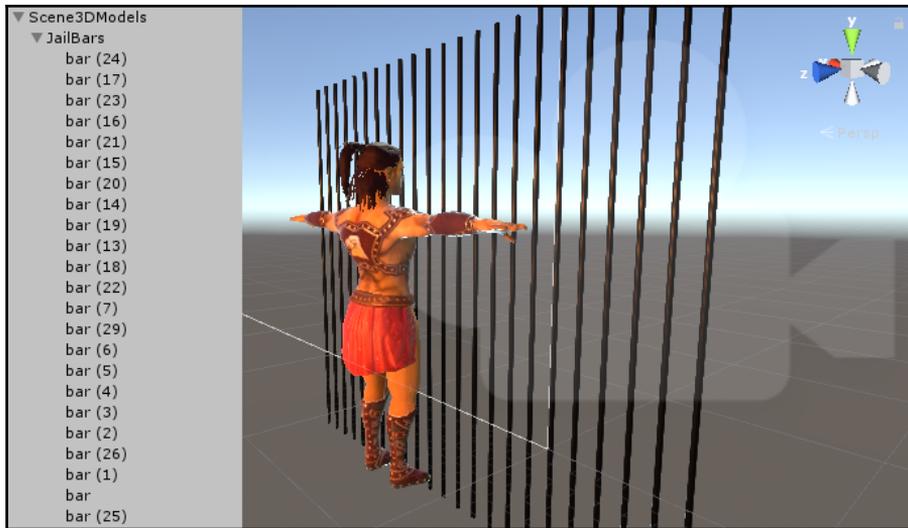
Adding jail bars to the scene

Adding a series of jail bar models used to build the prison in our game level will be super quick.

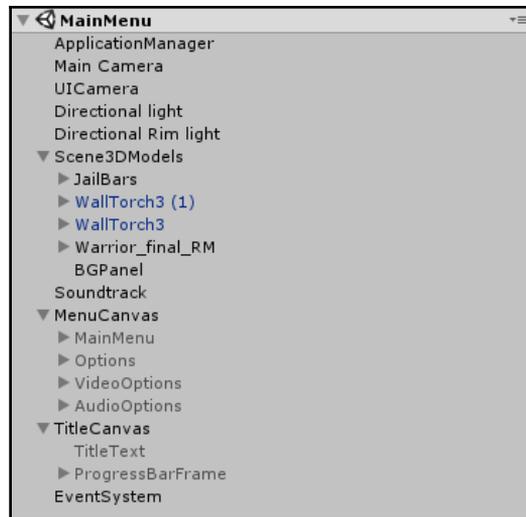
Create an empty `GameObject` and call it `JailBars`. Drag it under the `Scene3DModels` `GameObject` container.

Use the prefab: `Chapter5-6-8\prefabs\bar.prefab`, which is the base for the jail bars we used to build the prison cellar, by dragging one directly into the `JailBars` empty `GameObject` and moving it in front of the character.

Then, duplicate 25 of them and align them on the x axis, placing them at the same distance one from each other, like in this picture:



As you can see from the next image of the **Hierarchy**, the scene became complex enough, and we tried to keep it as neat as possible by grouping GameObjects in parents holders:



The final Hierarchy of the scene with the main objects expanded

Conclusion

Working with Unity UI is pleasant and satisfying, simple but powerful enough to manage any kind of HUD, menu, or diegetic user interface that you might want to design for your game. Lighting, shadows, and camera image effects are playing a decisive role in the atmosphere you are creating. It is important to remember to pack all the small sprites used for your UI with the Sprite Packer to be sure to obtain a performance gain before you build your release.

Testing screen sizes

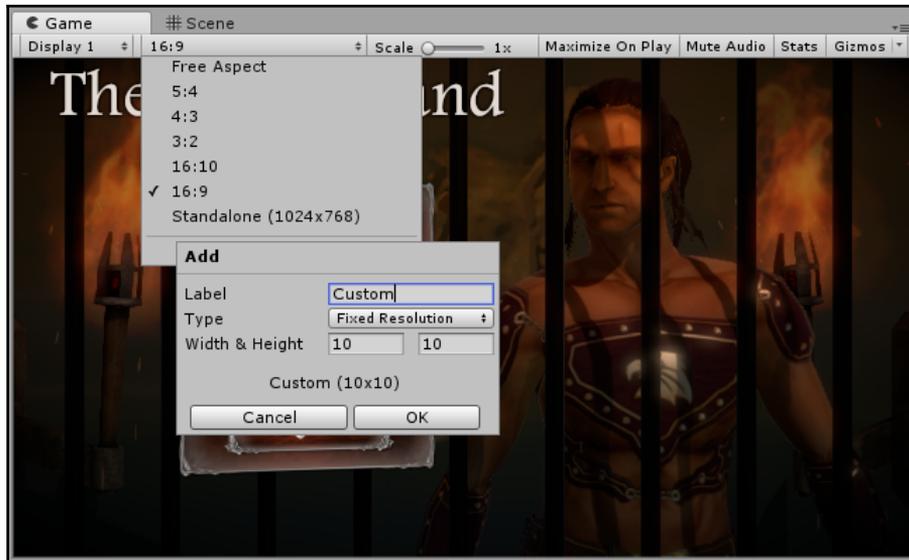
If we try to scale down the game view or launch the game with a very low resolution, we can note that the game title and the main menu are still in the correct position. This is one of the more important matters when you design a UI, you don't have to worry about realigning all the controls and panels via code, Unity UI takes care of it.

Scaling down the **Game** view will result in something like this screenshot:



To test different screen sizes for different devices and computers' video resolutions, you can use the drop-down menu in the top-left corner of the **Game** window to instantly change the size of it. You can create as many video resolution templates as you like by clicking on the + button.

You can give a name to the template and choose the ratio or the pixel size, like in the next screenshot:



If you choose a resolution bigger than the **Game** window can hold, the Scaling slider will automatically scale down the view for you to fit the area size.

Finally, our scene is ready! Enjoy it by starting Play mode and verify that everything works as expected.

Further looks

You can look at other kind of approaches with Unity Menu and take a look at the Game Jam Menu Template asset, which can be downloaded from https://www.assetstore.unity3d.com/en/?_ga=1.141556466.1110926299.1479406081#!/content/40465.

The video of the live training can be watched at <https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/game-jam-template>, and it can be read at <https://unity3d.com/learn/tutorials/topics/user-interface-ui/creating-main-menu>.

Summary

In this chapter, we created the main menu scene where users will start or quit the game.

We learned how to use animations on UI panels and how to catch mouse events and execute procedures upon firing an event. We wrote our own scene loader component for the menu, which makes use of the new `SceneManager` namespace.

We also learned how to use the World Space Canvas in 3D space and how to filter out graphics artifacts using a secondary camera to render just the UI, creating an in-game/pause menu.

We created an additional graphic HUD, displaying the *health* in the form of a *being catch again* amount using UI mask and 360 circular filling UI images.

We also gave the player further feedback by reusing the `TextHintUI` object we created in Chapter 9, *Items Collection and HUD*, and worked across scripts in order to send the information to this object.

In the next chapter, we will finally test the final demo, optimize the performances, tweak the effects, and add some other cool effects!

13

Optimization and Final Touches

In this chapter, we will take our game from a simple example to something we can deploy by adding some finishing touches to the island. As we have looked at various new skills throughout this book, we have added a single example at a time. In this chapter, we'll reinforce some of the skills that we have learned so far, and also look at some final effects that we can add in more detail; these aren't crucial to the gameplay, but add a professional quality to your work, which is why it's best to leave them until the end of the development cycle. For the purpose of the book, we will assume that our game mechanics are complete and working as expected. Now, let's turn our focus to what we can add to our island and the game in general to add a defining flair.

Understanding the basic concepts and getting the most out of all the features mentioned in this chapter, such as terrain and tree systems, image effects, batching, and occlusion, may require a while. This is an overall guide that should be integrated by reading the official Unity documentation as well as by experimenting with the editor.

We'll look at the following tasks in this chapter:

- Terrain tweaks, splat maps, grass details tips, and using SpeedTrees
- Quick level design, placing guards at specific starting points
- Game design, using a simple waypoint system to route AI surveillance rounds
- Adding realism with Enlighten, the new GI-capable Unity 2017 lighting system
- Baking lightmaps with Enlighten and using mixed lights to achieve the best results

- Global fog and other post-processing image effects to add depth and realism to the scene
- Adding a soft trail for our launched stones to show their trajectory
- Frustum culling and basic camera settings
- Graphic rendering performance optimizations, Static and Dynamic Batching, and Occlusion Culling
- Further optimizations: GPU Instancing, LOD, and camera Layer Culling
- Creating a final ending scene with Timeline and Cinemachine together with the new Post-Processing Stack

Tweaking the terrain

We will give the island terrain a better look by exploring the features that the built-in terrain system gives the developer. To make our game appealing to players when they begin, let's take some time to wander around the environment, improving the detail on the terrain, and enrich the player starting point with some additional fire torches and particle systems. We will now return to the terrain tools to improve the level of detail of our island. This will be done to the terrain in our `Level1Final` scene, so ensure that you have the correct level scene open in Unity now.

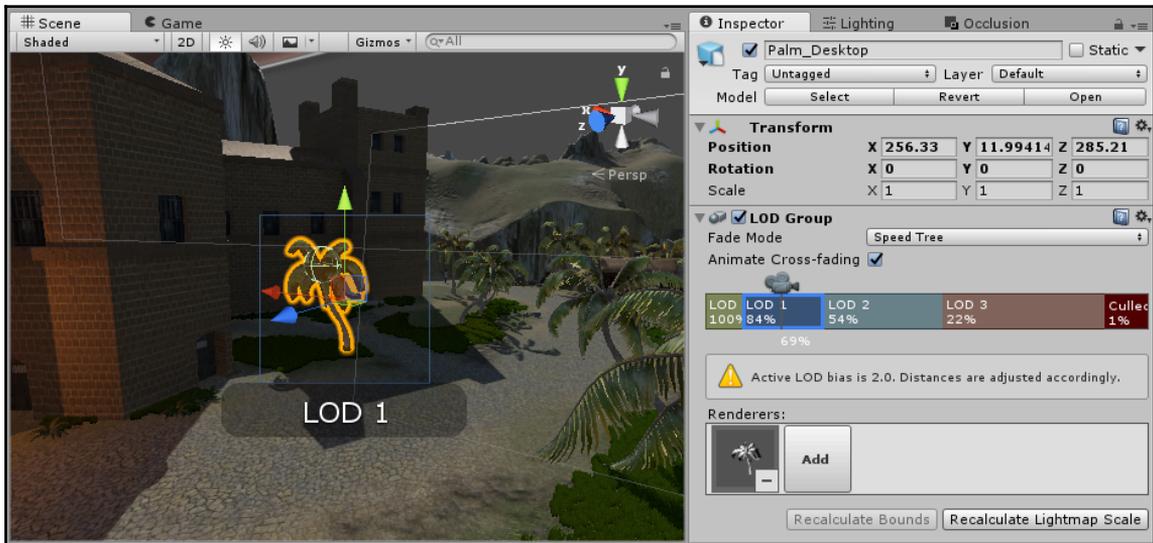
Using SpeedTree

To get a natural look at the trees that are placed on the island, you should attempt to think logically about where they may grow. In simple terms, a tree spreads seeds that cause others to grow around it over many years, while man-made forests may be created in a uniform order of rows of trees that are spaced evenly apart.

This system includes the nature shader feature, already seen in the previous version of the engine, which will bend tree leaves and grass when a `Wind Zone` component is blowing the wind in the scene, but also implements a bone system that allows the artist to quickly implement tons of tree variations with a few mouse clicks.

SpeedTree can be used in two ways: by dragging prefabs into the scene as individual `GameObject` instances or by specifying a SpeedTree prefab as the tree prefab that will be created on terrains with the **Place Tree** brush.

Here, you can see a SpeedTree prefab instance dragged into the scene and its relative LOD settings:



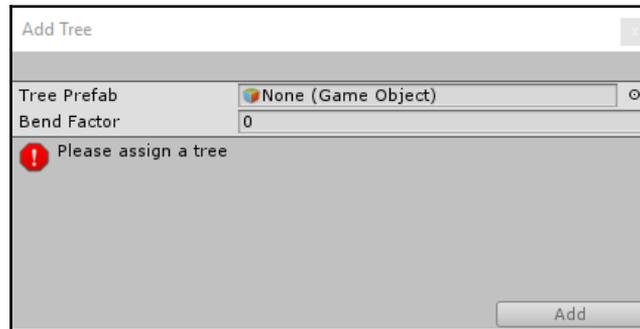
For more info about the SpeedTree system, check out the official SpeedTree website: <http://www.speedtree.com>. To learn more about Unity SpeedTree integration, visit <https://docs.unity3d.com/Manual/SpeedTree.html>.

Our island is supposed to be a relatively untouched environment, so let's take time to place some more trees on the island in small groups. In the latest versions of Unity, this new kind of tree can be placed on the terrain by painting areas with them.

Ensure that you have imported the environment package internally; you can always check by re-importing it, and look at the checklist to see whether something was updated or is missing in your project for that package. If you have already done so, you should have a `SpeedTree` folder under `Standard Assets/Environment`.

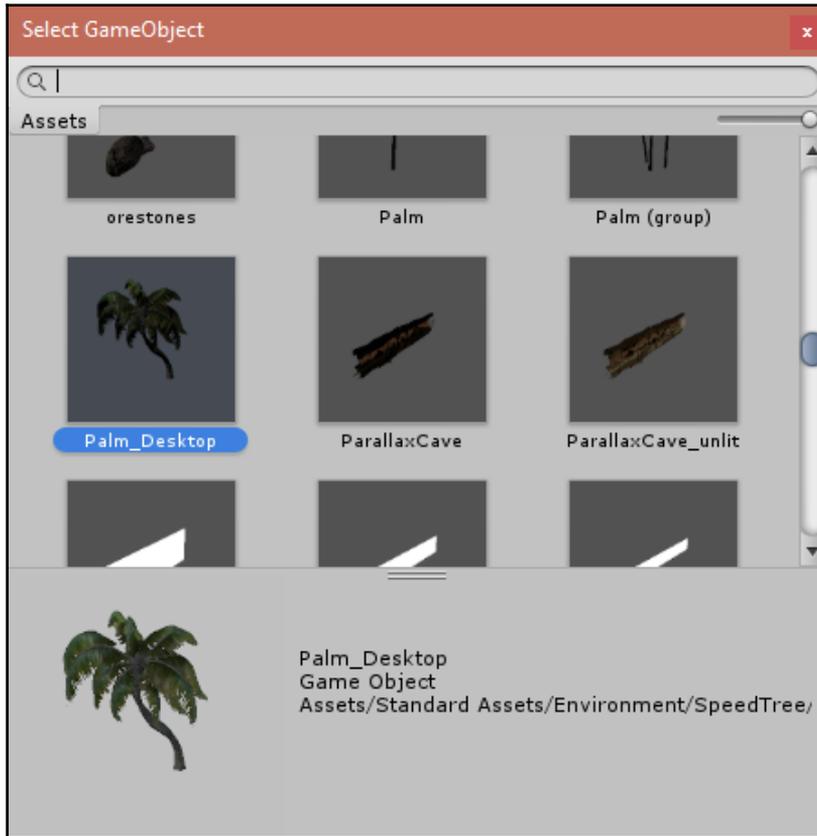
Select the **Terrain** object and then in the **Inspector**, choose the **Place Trees** section tab on the **Terrain** component in the **Inspector**.

To be able to use this feature and use the brush to paint a tree, we must have at least one tree in the **Trees** list. Add one by clicking the **Edit Trees** button and selecting **Add Tree**, this will open the **Add Tree** window shown here:



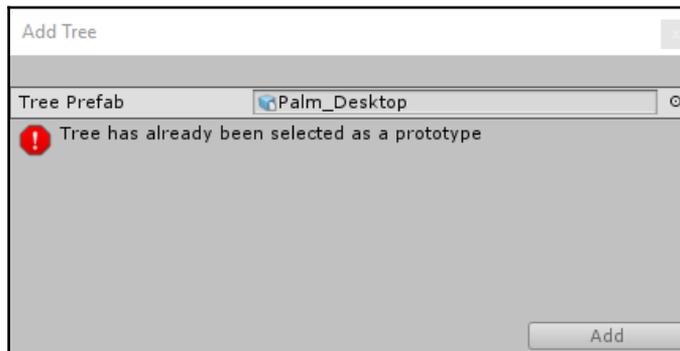
Clicking the circle icon on the **Tree Prefab** selection will pop up the `GameObject` selector window. You can filter results by typing a string, in this case, *palm*, in the search box at the top of it. Results will be filtered and you can choose the `Palm_Desktop` prefab, which is located in the `Standard Assets/Environment/SpeedTree` directory.

You can ignore the **Bend Factor** settings when using *SpeedTree*; these trees will have their own factor of bending with the wind, as opposed to the trees in older standard assets you may have used with the previous version of the game engine:



The GameObject selector window with the palm_desktop prefab selected

When you click a prefab, you can visualize a bigger preview below the selector grid, check the origin path of the asset. When you double-click it, you should see your **Add Tree** window again, with the palm prefab selected:



The **Tree has already been selected as a prototype** error is shown if you try to add a **Prefab** that already exists on the tree list for the terrain.

Then, go around the environment placing trees in groups. Try to create a treeline just before the edge of the coast to make the environment feel dense. As you do this, bear in mind that we will add some hills and further texturing afterwards to help the placement of the trees seem more natural.



In a production environment, it is more convenient to use an instance placement tactic despite the slower procedure, as compared to using the *Terrain Place Trees* brush. That way, you have more control over tree layout by using the move tool on each single tree instance, setting a detailed LOD and LOD Group according to level design, and so on.

Check out this historic video where Speed Trees were introduced for the first time:

<https://www.youtube.com/watch?v=fk1jOk7Ix0A>.

Hills, troughs, rocks, and texture blending

Now, we will refine our terrain shape and look at some editing tips. Bear in mind that the strategy we used for the trees is convenient for grass detail areas as well.

It is better to paint grass on the terrain in areas where you want to place grass:



As we can see in the preceding image, grass clumps should never be placed alone, because they will look weird. Instead, green spots should have a certain density, and it is even better when they are mixed with different grass textures.

Depending on whether your grass textures are lighter or darker than the ground terrain texture, a proper match should be created. For example, in our work, we are using a base light color for the sand of the terrain and a darker one for the grass patches.

It is appropriate to use the grass mold texture where we plan to place grass detail.



In this way, the overall look will be much better than seeing grass coming out of the sand! Of course, it is up to the artist and the designer. This is just a quick tip that you may generally find useful when editing your prototype terrains.

Smoothing and painting

Try and make use of the Smooth Height and Paint Height tools; for example, you can create a beach by making a plateau on one edge of the island. To achieve this, try using the Paint Height tool with **Height** set to 9, and then use the Smooth Height tool to soften the transition between beach, sea, and land. Finally, add some trees and paint texture details to polish off the transition between differing topographies.



Remember that the more you practice, the more you will get used to how the brushes behave, and the more creative you can be with your environment. Don't be afraid to try things out, there's always undo!

Keep on the right path by drawing path details with splat maps

In order to let us place the player on a part of the island that will lead them to our buildings, create a path going from the beach you just created that leads across the island to where the buildings are located. This will encourage the player to explore the island, ultimately leading them to where they need to go. For this, try using the Paint Height tool, starting at a height of 12, and slowly increasing this height as your path leads from the beach to the location of your buildings. To avoid harsh inclines leading down to the path, use the Smooth Height tool to create a smooth transition between high steps. Paint over this with the rock and sandy textures to create a path that looks something like this:



Go around the island and keep working on it as much as you like to create interesting features. Just remember to try and maintain the path so that the player makes their way to the various areas of the level intuitively.

The level was designed to have four interesting points, which are also, respectively, the spots where we will place the four artifact pieces needed to complete the quest: the old man's hut, the village, the hidden lake, and the far south beach. Except for the last one, which will require the player to jump down the usual path to access it, we will try to draw the roads to those three areas: one main road, starting from the prison, following the old man's hut, then inside the canyon down to the west side of the island. At this point, we will make two roads: one will lead to the village, and the other to the top of a guard's house, from where the player will be able to spot the far south beach.

The following is the two-way split piece of road at the end of the canyon seen from the west:



Toward the end of the path, create a hill area around the buildings so that the player takes the hint that they have arrived at the desired point: a small hidden lake. This lake will be reachable both from the path around the guard house and a hidden wooden plank that acts as a bridge, right after the old man's hut. In an adventure game, it is always good to create more possible paths to reach a point or achieve something. Usually, different paths mean different speed/difficulty; the game designer can choose what is best.

At this spot, you will need to arm yourself with patience and carefully edit the terrain shape with the Paint Height and Smooth Height tools to help NavMesh Baking a bit, to help the AI Nav Mesh Agents to climb their way toward that path full of slopes! Use the prison walls as a reference and flatten the terrain around the walls so that one of the guards can walk around the area.

Another idea is to leave another way to the right of the prison wall open, to join the mountain road that departs from the guard house and follow the mountain. In this way, the player will be really free to choose whatever route they want to solve the quest. We placed the artifact pieces in three ad hoc spots, and there will not be a specific order to be followed to solve the quest and collect the first three of them.

Another good idea would be to add some trees and grass in specific spots and paint over the mold and sand texture layers to obtain a result that is visually better. To reach this spot from the old man's hut, we use a long wooden board to fill the gap of what can be a *death hole*. We place a cube trigger in this hole to let the player *die* whenever he might fall down the board, adding some gameplay. A nice addition, though the visuals are not included in the book, would be to have some skeletons, bones, and skulls—signs of the fact that you are not the first who ended up there for the rest of their days:



The back of the prison walls reveals a hill path toward the hidden lake

Here, the challenge will be to cross the two guards and let them walk the same waypoints route, but it can be really tricky to reach the top because of the slopes, especially when climbing up. The AI can get stuck. For the purpose of this book, we made two different routes for the two guards, and neither of them climb up or down the hill, only the player will be able to do so. You can try to change this behavior whenever you like, and try different AI waypoints designs, but pay attention and remember to always recalculate (bake) your **NavMesh (Navigation Mesh)** when you edit the terrain! Also pay attention to the Nav Mesh Agent component Radius and Speed property that they use for walking, as well as the NavMesh baking parameters to tweak eventual AI path-finding issues.

Some level design: placing the guards

You may also want to add a couple of AI guards and add the AI waypoints to walk the way. We will also add an AI for the guard's house, two in the village entrance, another two roaming the village, plus one in the hut where we will put a piece of the artifact and one or more in the far south beach:

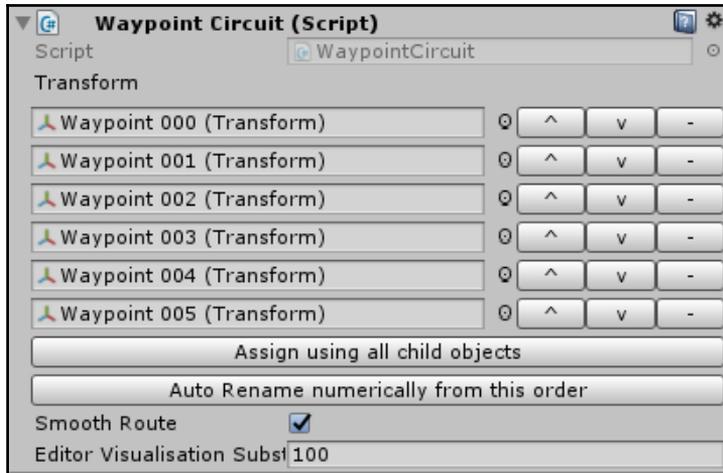


This picture shows guards' positions on the island

You can add as many waypoints as you want and also share them among different AI; the order they follow is specified in the advanced AI component. You can change the number of waypoints, and you have to specify one transform object for each slot. To make things clearer in the Editor, we can take advantage of the waypoint circuit component from the standard assets AI car example. The circuit will be visualized in the scene editor as splines or as straight-line polygons, depending on the settings you give. In our case, we will use the polygon, simpler and less CPU-hungry. To implement a circuit visualization in the Scene editor is very simple:

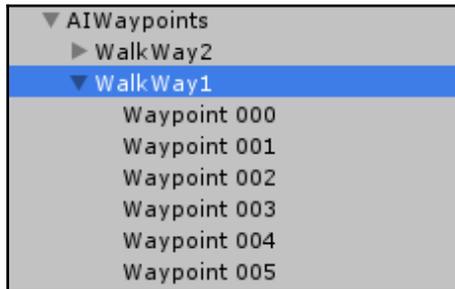
- Create an empty **GameObject** and place it as a child of the **AIWaypoints** **GameObject** (the one that contains all the actual waypoints for the level) and name it `WalkWay1`.
- Select all the waypoints you associated to a guard patrol route and drag them to become children of the `WalkWay1` empty **GameObject**.

- Select the `WalkWay1` GameObject and click the **Add Component** button. In the search field, type *circuit* and select the **WaypointCircuit** C# component.
- Look in the Inspector, find the **Assign using all child objects** button, and click it to let the component automatically assign (the order in the hierarchy matters) the children to the component items:



The Waypoint Circuit in its initial shape

Now, deselect the **Smooth Route** checkbox. I suggest you don't automatically rename the waypoints for the next `WalkWay` circuit you may want to set up. If you do so, it will be confusing when looking at the Advanced AI components of a guard to understand which waypoint they are using, because all circuits will have waypoints children with the same name:



The hierarchy of the AIWaypoints GameObject after autorename

As you can see in the previous image, using **Auto Rename numerically from this order** on all the walkways will let you have a lot of waypoints with the same name. Luckily, we can modify this script to control the way the component renames its children objects if we need it:



The Scene view shows the walkway for a guard at build time

Now, let's focus on something very important in developing a third-person adventure game—the scene lighting.

Unity lighting

From version 2017.1 and onward, the new lighting system can be of two types: Enlighten or the new, still experimental, Progressive Baker. While we are not going through the new experimental feature in this book, you should know that this was something requested by many developers for making the process of lightmap baking smoother and quicker, and will soon be available with the 2017 version. Even so, the lighting in Unity 5.6 and the 2017 version has changed so much that Unity Technology wrote a guide for the upgrade and is updating the lighting documentation.



For more information, take a look at the upgrade guides list for various versions at

<https://docs.unity3d.com/Manual/UpgradeGuides.html>.

The **Lighting** tool has three main sections:

- **Scene:** Scene deals with the main settings of real-time and baked lighting and the process of baking itself.
- **Global maps:** Global maps will show all the lightmaps for a scene; it will be empty if, for example, all the lights are set as realtime (nothing to bake, hence no lightmaps).
- **Object maps:** Object maps show a selected object lightmap only. This is useful when you want to understand what's behind a specific object illumination.

Switching to one of them lets you access different options.

Scene setup

The **Scene** tab in the following image is divided into five sections for setting scene environment, GI and the lightmaps baking:

- Environment
- Global Illumination
- Mixed Illumination
- Lightmapping
- Other settings

The Environment settings

In the **Environment** settings in the **Scene** tab, you can also set the **Environment Lighting** source of the scene. Set the **Skybox Material** to the **Default-Skybox** material and the **Sun Source** GameObject (typically the **Directional light**).



The Lighting settings are per scene, that means that each scene will have its own settings.

The **Environment Lighting, Source** can be the **Skybox** itself or a gradient or a single color. The **Intensity Multiplier** will increase the power of the environment lighting source. While our **Directional Light (Light)** handles the main lighting-acting as the sun in this example, the ambient light will allow you to set a general overall brightness, which means you can create scenes that look like a certain time of night or day. Try adjusting this setting now by clicking the color block to the right of the setting and experimenting with the color picker's color and alpha settings. If you choose to adjust either the skybox or sun source, or the **Environment Lighting** or **Environment Reflections** settings after a bake, in order for this change to affect static objects, you will need to rebake your scene by opening the **Lighting** window then choosing **Baked** for the **Ambient Mode**:



The Environment section of the Scene tab in the Lighting window



When you do so, **Realtime Global Illumination** is deselected automatically. Similarly, when you change Realtime GI setting, the **Ambient Mode** will change accordingly.

Global illumination is a lighting feature that also takes into account indirect light that bounces on surface for illuminating other surfaces. This feature delivers a lot of realism because it emulates how light works in the real world. When game engines implement this feature, games became more and more realistic.

Realtime lighting and Mixed Lighting

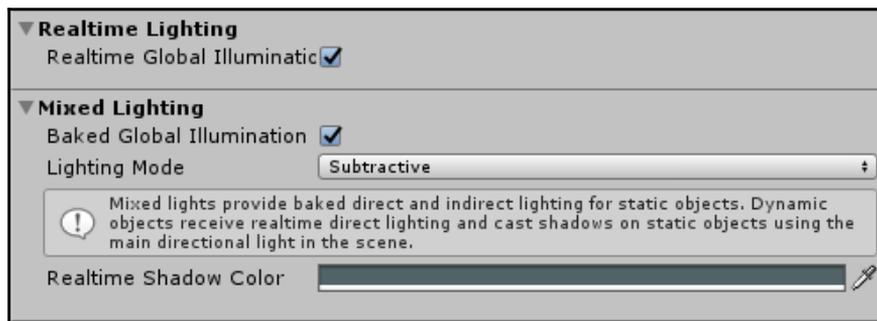
This controls whether real-time lights in the scene contribute indirect light. If enabled, real-time lights contribute both direct and indirect light. If disabled, real-time lights only contribute direct light. This can be disabled on a per-light basis in the light component Inspector or in the **Light Explorer** by setting the **Indirect Multiplier** value to 0.

It controls whether mixed and baked lights will use **Baked Global Illumination**. If enabled, mixed lights are baked using the specified **Lighting Mode** and baked lights will be completely baked and not adjustable at runtime.



For more information about global illumination in Unity 2017, visit <https://docs.unity3d.com/Documentation/Manual/GIIntro.html> and for a complete tutorial (based on 5.5 though) head to <https://unity3d.com/learn/tutorials/topics/graphics/introduction-precomputed-realtime-gi?playlist=17102>.

The **Realtime Global Illumination** and **Baked Global Illumination** set for a fully-featured **Mixed Lighting** setup where both real-time lighting with GI and real-time shadows will play together with baked lightmaps with GI:



Realtime only versus Baked only versus Mixed lighting

Unity leaves to the designer the choice of whether to use real-time illumination with global illumination support and real-time soft shadows at maximum quality, which is the choice for high-end PC and Console platforms, or bake all the lights with lightmaps with global illumination and not use shadows or real-time lights at all.

Nowadays, we see a lot of typology of games and there are a lot of platforms out there that differ in terms of performance and GPU capabilities. This choice must be made at design time. This means that before you proceed, you should already know what you are doing, and in which way to do it.

It is not just a matter of platform you should consider, but also the design of each level and the game itself. For example, your game might be all indoor-based, with only artificial lighting; in this case, you might want to bake everything in the lightmaps and use only some real-time light and real-time shadows for the characters and dynamic objects.

In an open-air only game, you might want to consider baking no lightmaps at all, at least on PC and console platforms, against a combined or lightmap-only solution for the mobile platforms, depending on the level detail of the overall environment, characters, and so on. Many modern games combine both these techniques to achieve the best result because they have many kinds of different scenes: open-air only, partially open/partially indoor, and just indoor levels.

You should also take into account whether your game has a real clock behind the scene, that will change the time of day by moving and rotating the sun (Directional Light) properly. No matter with what scale you progress the time, quick, real-time, or slow, if the time is supposed to change by design, you should consider baking lightmaps *only* for the indoor parts without windows, such as cellars, undergrounds, and dungeons. Otherwise, the move of the sun and the change of the light of the day will make the static lighting look wrong. There is no specific rule; the designer should accurately study the requirement of the final result, and opt for the best solution in terms of realism and/or performances.

Lightmaps and baked Global Illumination

Lightmapping is the method of baking, and rendering the lighting that happens to be affecting a rendered 3D object to a texture file. This can be done in modeling applications, but Unity also allows you to lightmap all of your lights and environment elements in one, alter light, and bake the lightmaps again.

Why do we need to lightmap? Well firstly, lights placed in your game scenes makes your **Graphics Processing Unit (GPU)** work harder, and by saving the effect of lighting into a texture, we can boost performance and improve the look of our game environments at the same time rather than having lights affect parts of our game dynamically.

Nowadays, on desktop platform, lightmapping is generally reserved for indoor environments, mixed with real-time shadows and real-time GI. On mobile platforms though, the GPU does not perform like desktop graphic cards and avoiding real-time shadows and lighting might be a must for the game that you are designing to be able to obtain acceptable performances.

Lightmapping must be typically done only to objects that will never move in the game; this is why Unity includes a **Static** checkbox next to every GameObject name at the top of the **Inspector**.

To prepare 3D objects for lightmapping, before you hit the **Bake** button, we have to typically perform three preparatory steps:

1. Ensure the 3D asset that the lightmapped GameObject originates from has **Generate Lightmap UVs** checked in the **FBXImporter** component in the **Inspector**.
2. Check the **Static** checkbox in the **Inspector** on the GameObject once it is in a scene to tell Unity to include this in our lightmapping.
3. Set the options for baking in the **Lightmap Setting** section.

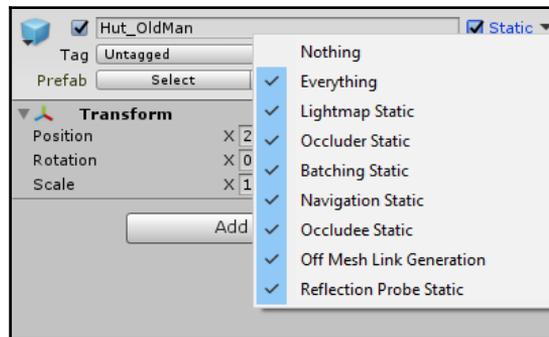
Baking the scene

Before we bake our scene, we will set up our hut, village buildings and prison walls, wooden barrels, static rocks, and all the static objects in the scene to be enabled for baking (**Lightmap Static**).

Preparing for lighting

The first object we will set up is the `prisonWall` object. As we expect to see differing lighting indoors compared to outdoors, we need to light the interior of the building. However, instead of casting light onto the lamp to make it appear lit, we will use a **Self Illuminated** shader on the Material of the **lamp** part of the model to give the illusion that it is lighting the room; the actual lighting will be done by a baked point light.

The old man's hut GameObject in the Inspector will be a full static object:



You want to dock this window as a panel with part of the interface, for example, in the same position as the **Inspector**. Drag the title tab of the lighting window and drop it to the right of the **Inspector** title tab; the window will snap into place.

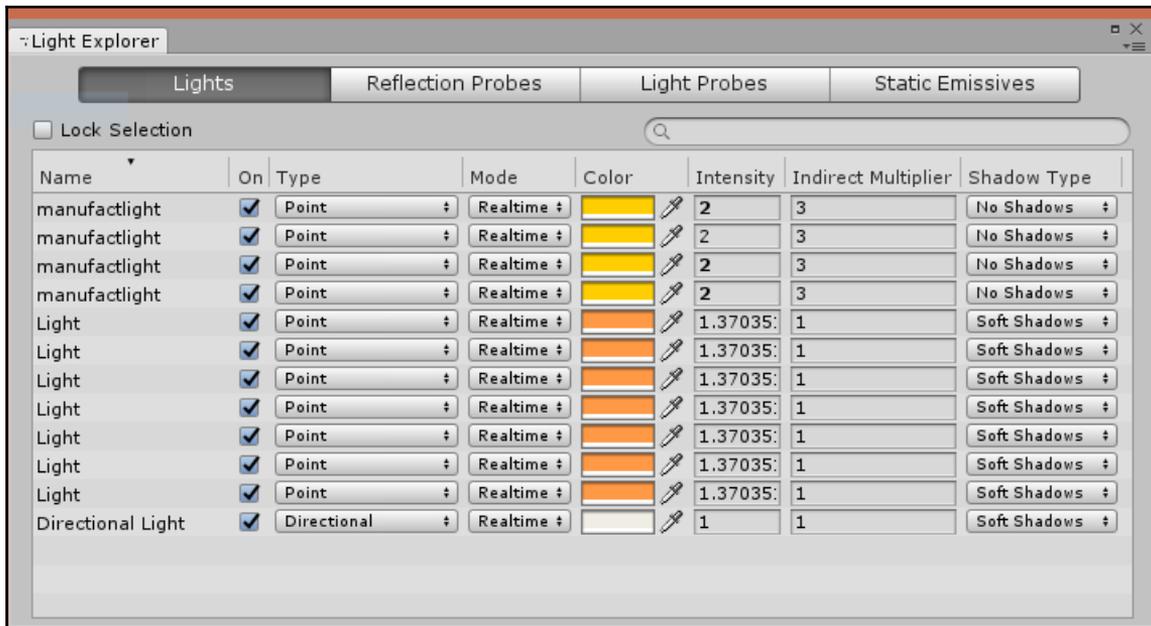
Now, we need to set the **prisonWall(s)** to be included in our lightmap bake—select the `prisonWall` parent object and check the **Static** checkbox next to its name in the Inspector—again choose **Yes, change children** when prompted. Do so for all the `prisonWall` objects. We would also like our village houses and hut to cast a shadow on the terrain and receive light from our **Directional Light** in the bake; select the campfire object in the **Hierarchy**, and check the **Static** checkbox next to its name in the **Inspector**. As earlier, confirm that child objects should also be made static when prompted.

Including or excluding lights from the baking

Before we continue with our Bake, we will set two of the lights in our scene: our **Directional Light** (that is, the sun) and our **Point** light in the old man's hut, also to be static. Our Door Light GameObject cannot be static because we do not wish to bake its default red color onto the outpost, as this will remain there even when it changes to green when the player unlocks the door. This is because the lightmap is stored as a texture that, once baked, remains unchanged at runtime.

While it is not necessary to include their light in the bake, marking these objects as static makes our game run more efficiently. Remember that lights can also be forced to only function as part of **Lighting** by setting them to **Baked Only**, as we did for the **Point Light** earlier. Another advantage of making these objects static is that it will help remind you not to move these objects.

Select the **Directional Light** in the **Hierarchy** now; you should recall that it is set as a child of the **Environment** empty parent object, so expand the parent object to reveal the **Directional Light** object if necessary. Check the **Static** checkbox in the Inspector and then switch to the **Lighting** window if it is not already open. In the **Object** section of the **Lighting** window, set **Baked Shadows** to **On** (realtime: **Soft Shadows**) using the drop-down menu, leaving the other settings that appear at their default values:



Since the latest Unity versions, a new standard handy panel has been added--the **Light Explorer**. This panel helps when your scene is huge and full of many lights to manage, often nested in other objects that are part of the scene. In this panel, you will just see the lights with their most used and important settings. This is very useful when you want to quickly change your setup on certain parts of a level or globally; it gives you the control you need. Now, to ensure that our lights are only part of our lightmapping (and do not attempt to try and light objects dynamically) choose **Baked** for the lighting setting or do the same with the **Light Explorer**.



For these lights, a good idea is to disable shadows.

Leaving soft shadow, even with the movement part of the script disabled, will make the final result of lighting look wrong.

Excluding GameObjects from the bake

While lightmaps will make our scene look a lot better and improve performance, we must take care to exclude certain objects from being baked. For example, we don't want to bake lightmaps for the rocks dynamic object we left on the terrain, or the `SpeedTree`. If you go for a totally non-realtime solution, which means you also disable realtime shadows for any of the lights of the scene, you also want to remove or disable the `WindZone` GameObject from the scene to avoid seeing trees oscillating in the wind with their shadow on the terrain standing still.

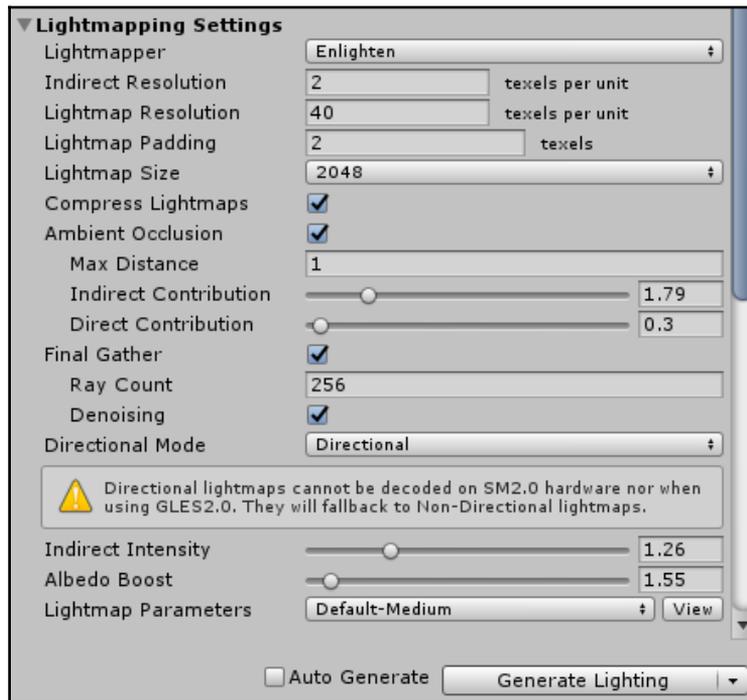
To exclude these objects from the baking, you can switch off their `Static` property or if they are supposed to be static but you don't want the lightmaps baked on this object, you may set the object as static, but deselect the **Lightmap Static** checkbox in the items list.

Baking the scene

Now that all our static objects and lights are prepared for the lightmapping, it's time to get to grips with the tool itself. Switch back to the **Lighting** window. If it was closed, open it again from main menu—**Windows** | **Lighting** | **Settings**.

Lightmapping Settings

Now that our scene is prepared, note that the `Terrain` object itself is marked as static by default, we are ready to make the last tweaks and bake the scene:



The Lightmapping Setting section of the Lighting window

The **Lightmapping Settings** section of the **Scene** tab in the **Lighting** window is where the lightmap baking is set up. Set the **Resolution** value to 40; this sets the overall quality for the bake, and if you decide that you prefer a higher quality after the bake is complete, you can always return to this value, increase it, and then rebake.

At the bottom of the section, there are the **Generate Lighting** buttons. If you click the arrow to its right, you can choose to clear any existing lighting data you have baked previously.



Beware, too-big values for the resolution may result in a very long calculation time as well as gigabytes of data for the scene to take into account. Play with these values gracefully. Also take into consideration the amount of time that may be needed to calculate all the lighting for this scene, especially if you have chosen to bake Global Illumination (Ambient Mode: Baked).



Flat shaded scene, no realtime lighting, no baked lightmaps versus same scene with realtime GI and realtime shadows

Lightmap baking is often a long process and will depend greatly on the speed of the computer Unity is running on; as many calculations are performed, baking may take anywhere from a few seconds to hours, depending upon the complexity of a scene. In this instance, we do not have many objects to include in the bake; as such, it should not take too long, but maybe prepare something else to do while you wait! The great thing about the lightmapping tool in Unity is that you can continue to view your open scene while baking is completed in the background; naturally, you cannot change the scene as this will invalidate the bake. This is one of the reasons Unity is implementing a second choice, which is the progressive lightmapper. Now, click the **Generate Lighting** button at the bottom of the **Lightmapping** window to start. A blue progress bar will appear, showing you calculations of the various parts of the lightmapping process. Be patient! It will be worthwhile once the bake is finished!



For additional information, head to <https://docs.unity3d.com/Manual/LightmappingDirectional.html>.

Following you can see a series of screenshots, from the flat shaded scene with no lighting at all, to the final result, and the intermediate passages. After quite a long calculation (it could take hours on an average gaming machine) you can see the result of the very long and CPU-intensive baking: baked lightmaps and baked Global Illumination plus real-time lighting and real-time shadows playing all together very nicely.

The following screenshot shows a good example on how a well-balanced use of mixed lighting baked shadow color and the shadow strength setting on the Directional Light can seamlessly melt the lightmaps baked for the shadows cast from the prison walls on the terrain and the real-time dynamic shadows cast from the trees that move with the wind, both coming from the same (mixed) directional light:



When you work on complex scenes, the process of baking global illumination and use mixed lighting can bring long waits; ready your coffee or favorite beverage and be prepared to wait quite a bit.



You can still use Unity Editor and/or write code and compile new saved edits while the lighting calculation takes place.

You can even start play mode and the baking will pause, then resume when the editor stops.

For best results, in term of the time required to perform the baking, we suggest to leave your PC alone, even just moving the mouse or browsing the internet can steal a lot of CPU power.

While you search photo-realism, such as what an interactive movie or a cartoon looks like or anything in between, lighting must be carefully chosen. Even with soft real-time shadows, sometimes it is better to go for mixed lighting and baking the lightmap, this often depends on the fact that your game design requires the time of the day changing while playing. In this case, of course, baking the lightmap on outdoor levels is a huge mistake!

Again, in the following figure, we can see the same spot before and after the baking. On the left-hand side, only realtime illumination and shadows take place while in the second (right-hand side), the terrain and geometries have lightmaps as well. The final result is way smoother and feels more realistic:



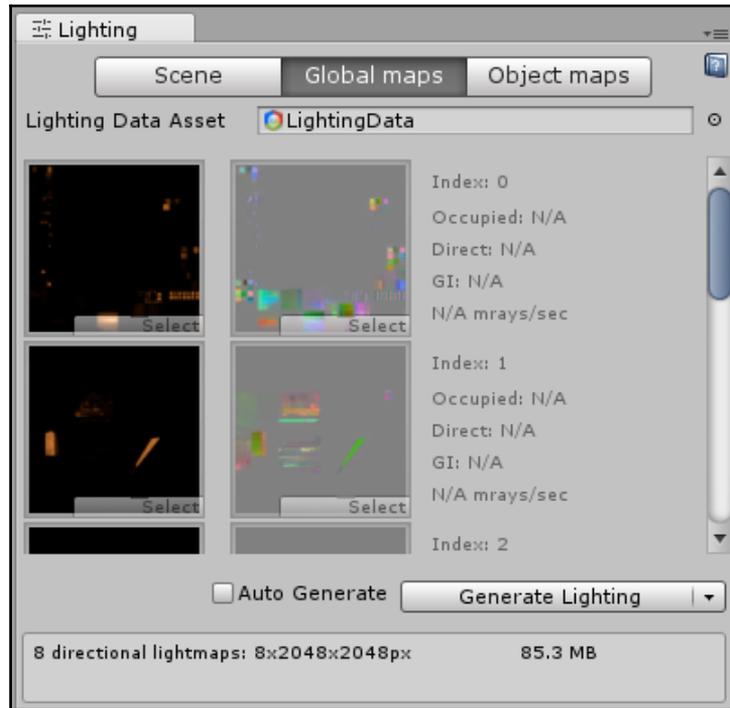
Even though the best results are achieved with a combination of mixed lighting, especially for outdoor scenes, if we try to switch the directional light to mixed and bake the whole scene, we may find some artifacts on some of the models. If you see artifacts, this is due to a wrong or non-existing second UV channel on these meshes.

Global maps

Global maps will show the baked data. When only GI is baked, **LightingData** will be calculated and show up in the **Lighting Data Asset** slot. The **Global maps** tab is empty if there are no baked lights in the scene at all:



Once we have finished the baking, we should see the real-time GI precomputed `LightingData` file but also a list of all the lightmaps textures generated by Unity.

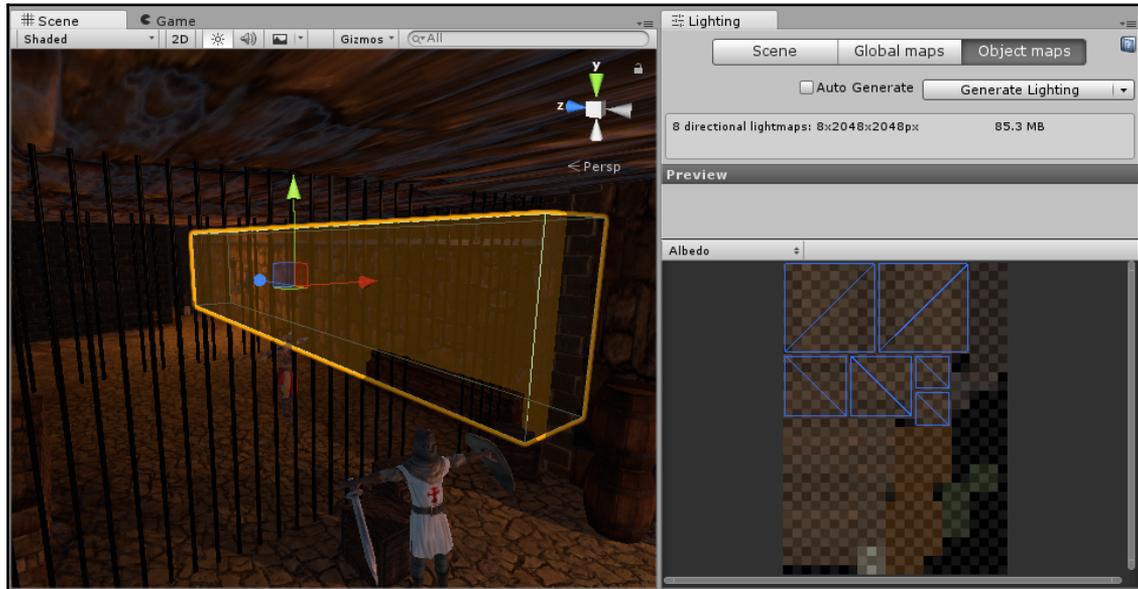


You can access the created lightmap image files once they are baked and saved, the Global maps section allows you to locate the texture files and edit them using an application such as Photoshop or The Gimp. Click a preview to find the Intensity or the Directionality lightmaps in the Project view. They are always stored in a subfolder where your Unity scene resides, this folder will have the same name of the Unity scene.

While this section gives an overall look at the lighting data and lightmaps of the last baking in the Object maps tab, you can see the lightmaps for a given object.

Object maps

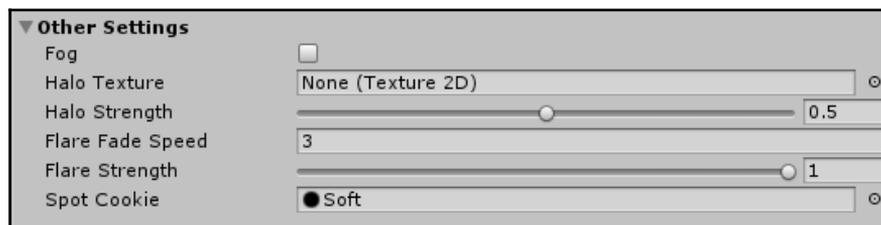
The **Object maps** tab will show something when a baked object is selected in the scene; if the lightmaps were baked for this object, you will find a view of the texture and its UV coordinate in the **Inspector**:



The Object maps show the selected object lightmaps

Other settings

The last section, right before the **Generate Lighting** button, is the **Other Settings** section that contains the fog settings—flare and halo settings; these are covered in the next paragraph. Flares are special textures added to lights that act as glow blink for that light. They can be set individually and can have different shapes. Halo is drawn around the light center and here, you can specify your custom halo texture:





Enabling **Fog** in **Other Settings** will enable the hardware fog in Forward Rendering and will make the Global Fog of the new Processing Stack work. If you enable the Global Fog effect in the Stack, you will not see any fog rendered until you enable this option.

The terrain and other objects were shaded gracefully, but some models we provided got a really weird lightmapping result. We can see an example of how bad baked lightmaps with a wrong or missing secondary UV channel look, such as for the model in this screenshot:



This kind of artifact is due to a wrong or missing secondary UV channel on some of the models we imported.

We will fix this to proceed to a general baking, on the top of which we will add real-time lighting and shadow for maximum result. We can do this by ticking the **Generate Lightmap UVs** option in the mesh import setting and applying re-importing for each of the static models we have used in our level, which might show artifacts after baking.



For more information on lighting, refer to the Unity manual page at <https://docs.unity3d.com/Manual/LightingInUnity.html>.

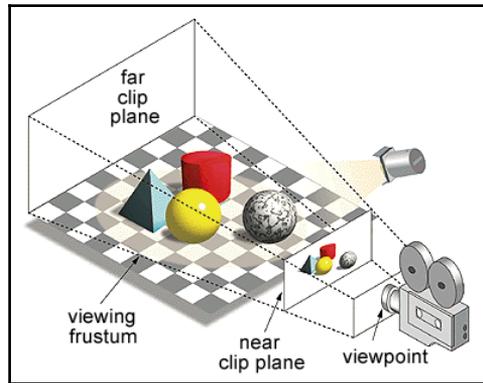
Optimizing performance

We will go through a series of game performance tweaking and knowledge, needed when you are ready to bring your game to the next level, from a quick prototype to something really playable on most computers. Most of these features are built into the engine, and you don't have to do anything special to use them, sometimes just tick a checkbox. You should, however, carefully read through this section to understand how and when they can be used, and in what part of the hardware (CPU, GPU, and so on) your optimizations are taking place. In this section, we will look at ways in which you can boost the performance of your game as an end product. Also known as optimization, this process is crucial once you have ensured that your game works as expected, ensuring that your player has the best possible experience from your game. Here, we cover some of the basics you should be aware of, but you should also understand that optimization is a broad topic that will be covered at the end of this chapter.

Camera clip planes (frustum culling)

The frustum is the cone that starts its pick from the camera center and is oriented where the camera is oriented, which contains the collection of everything that should be rendered by the camera. Everything outside this cone is excluded from the rendering (though it's still calculated and taken into account by the CPU). To determine the size of the frustum, the camera will clip near and far planes' distances and FOV angles are used. We will look at a picture of the visualization of occlusion culling (disabled) revealing still only a part of the map geometries as to what is inside the frustum. The frustum far clip plane should be set just a little farther than where fog starts and not too far from the camera. The near clip plane should be set close enough to render close pixel; for example, when the player is close to the camera or anything that might come close to the view.

Frustum culling is explained here with a single picture:



This schematic picture illustrates what is the viewing frustum and how the frustum culling works thanks to far and near clip planes Image courtesy of TheFreeDictionary

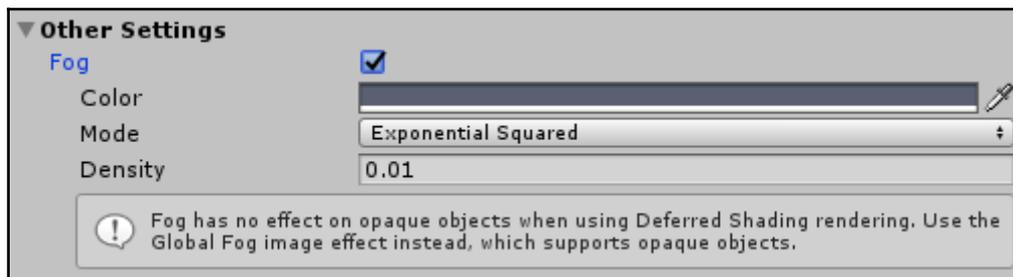
The longer the distance between near and far clip planes and the wider the **FOV** angle, the larger the number of object geometries that will be picked up for rendering within the frustum. Depending on the type of game and type of camera used in the game, it must be tweaked carefully to obtain the best performance.

Camera settings show our near and far clip planes values, 0.2 and 500, and a **Field Of View (FOV)** of 60°:

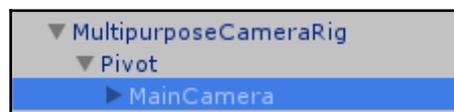


Standard fog versus Global Fog post effect

To add a nicer visual appearance to our island, we will enable fog. In Unity, fog can be enabled very simply and can be used in conjunction with the camera's `Far` clip plane setting to adjust draw distance, causing objects beyond a certain distance to not be rendered; this will improve performance. By including fog, you will be able to mask the cut-off for rendering distant objects, giving a less clunky feel to exploring the island. This means that we are able to boost the performance by reducing the draw distance, while still appearing to the player as if it is an effect intended by the game developer:



The final section of the **Lighting** panel's **Scene** tab is the **Other Settings** section. In this section, you specify which kind of method to use for hardware fog, whether it's enabled or not, and its color and density. In second-generation graphics video games, hardware fog was widely used. On modern hardware, it is better to take advantage of the global fog image post effect instead, also because we can do much better fog visual effect than the standard fog predecessor. The color is still important for the global fog post effect, and to set it, you must keep the fog hardware enabled. While the former is intended for use with **Forward** rendering, the latter is intended for **Deferred** rendering. We will learn what these different rendering modes for the engine are and where to set them globally in the next chapter. To add the global fog effect, select the **MainCamera** object under the **MultipurposeCameraRig** GameObject:



Set the property as you see fit on the **Inspector** and eventually, test the game to find the correct values:



Unity 2017 now manages fog with Forward and Deferred rendering in a more structured way, with the fog effect in the Post Processing Stack reacting only when the **Fog** option is enabled in the Lighting settings panel and automatically switching on one of the two according to the rendering mode of a camera.

Occlusion culling

One of the most important features Unity offers in term of the optimization of complex scenes with a lot of geometry is occlusion culling. Different from simple frustum culling, where the objects outside the **Field Of View (FOV)** angle are not rendered by the engine, occlusion culling will cut out objects and geometry that are in the frustum cone, but not *seen* by the camera at a given position/orientation. When a scene gets really complex, static batching and automated intrinsic frustum culling are just not enough. With Unity's occlusion culling, you can calculate (at editor time) what static objects will be seen by the camera according to its position and rotation. Setting up the correct values for the occlusion baker can be tricky, especially if you have used your own world scale, far from the 1 Unity unit = 1 meter standard. Given that you kept this scale throughout this book, you can leave the settings or try to tweak them to cover smaller objects, resulting in many more occlusion cells to be calculated (more time waiting for completion in the editor) but also a discreet additional precision when the game runs in slicing and cutting off parts of your 3D world. The occlusion window, which was docked as a panel right beside the **Scene** view for our comfort, has three main tabs:

- Object
- Bake
- Visualization

Let's take a look at each of them.

The Object tab

In the **Object** tab, you will see something when a **GameObject** with a **Mesh Renderer** component is selected in the **Scene/Hierarchy**. Select one of the prison walls' building mesh, and open the **Occlusion** window. You should see something like this:



In the **Object** tab, we can set whether this object should be an occluder, which means it will occlude the vision of other objects behind it, and whether it should also be an occludee, which means its sight can be occluded by other objects.

The Bake tab

The **Bake** tab allows us to finetune the settings for the occlusion calculation and should be modified carefully. They strongly depend on the scene scale and should be set opportunistically. Click **Clear** to delete the previous data, and **Bake** to start a calculation.

Depending on the size of the level and the number of static objects as well as how low the **Smallest Occluder** and **Smallest Hole** values are will determine the occlusion calculation grid size. Occlusion culling plays its top role on indoor-based maps, with a lot of building geometry.



After some tests, it turns out 1 and 0.05 was a bit high-quality given that the static models are bigger than that, but given the best result in terms of performance and weight in megabytes for the precalculated occlusion data.

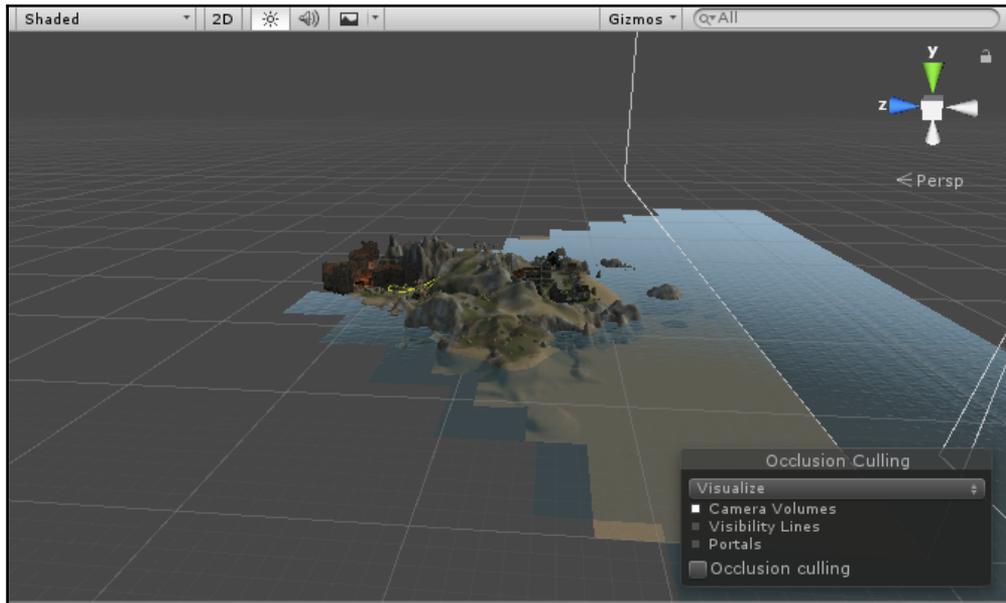
The following screenshot represents the Bake tab with occlusion settings, below the total amount of data from the last bake:



Beware, too-small values for the **Smallest Occluder** and for the **Smallest Hole** settings may result in a very long calculation time as well as gigabytes of data for the scene to take into account. Selecting the **Bake** tab while processing the occlusion culling data or after the baking is done will let you observe the **Scene** view in the grid size and how the level is subdivided according to the values we have put in the settings.

The Visualization tab

As we can see in the next image, you can see what will be rendered by the main camera in the **Scene** view. Only a portion of the terrain is rendered, like if it was sliced on the needed angle, and other geometry of the level is well-hidden. In addition, you can visualize **Visibility Lines** that are the actual possible sight lines of the main camera at the current position and orientation:

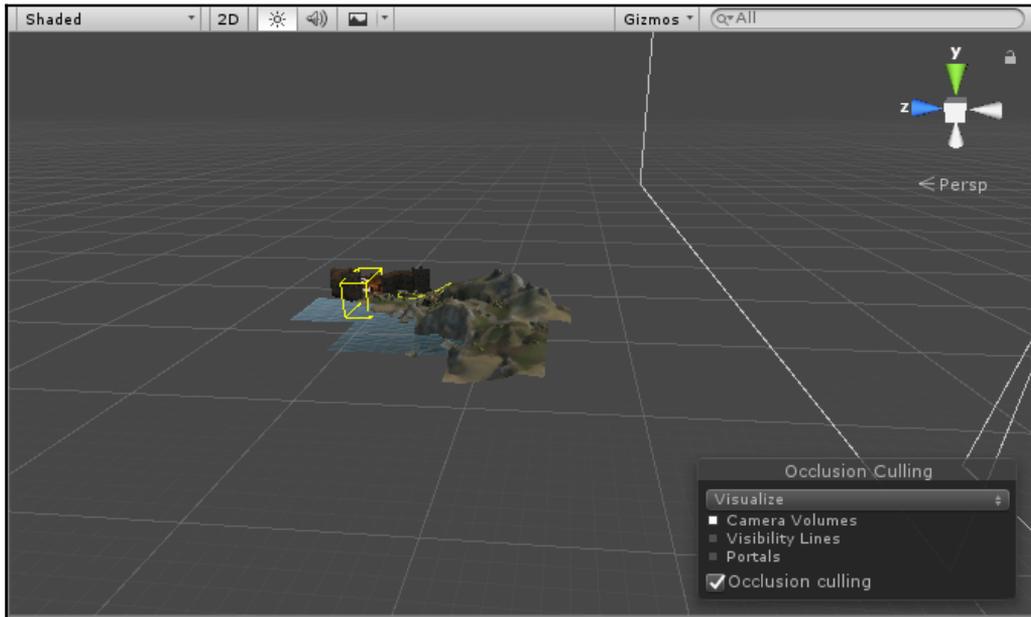


Frustum culling only in action: the geometries outside the field of view cone of the camera are not rendered.

Understanding how occlusion culling works is crucial to optimize a game that is supposed to *hit the shelves*. While it may seem easy to get it kicked, you may have to experiment and try different baking values before you find the optimum.

Ticking the **Portals** visualization option will also show calculated portals, areas that determine when a previously-hidden part of the level should be shown. You can see the difference between *plain Frustum Culling* and **Occlusion Culling** by selecting and deselecting the **Occlusion culling** option. In the preceding screenshot, you can see how the **Frustum Culling** removes the unnecessary parts of the level.

In the next similar image, with **Occlusion culling** enabled on the Camera component, you can finally see the result of the occlusion and how much geometry has been cut out compared to the previous image:

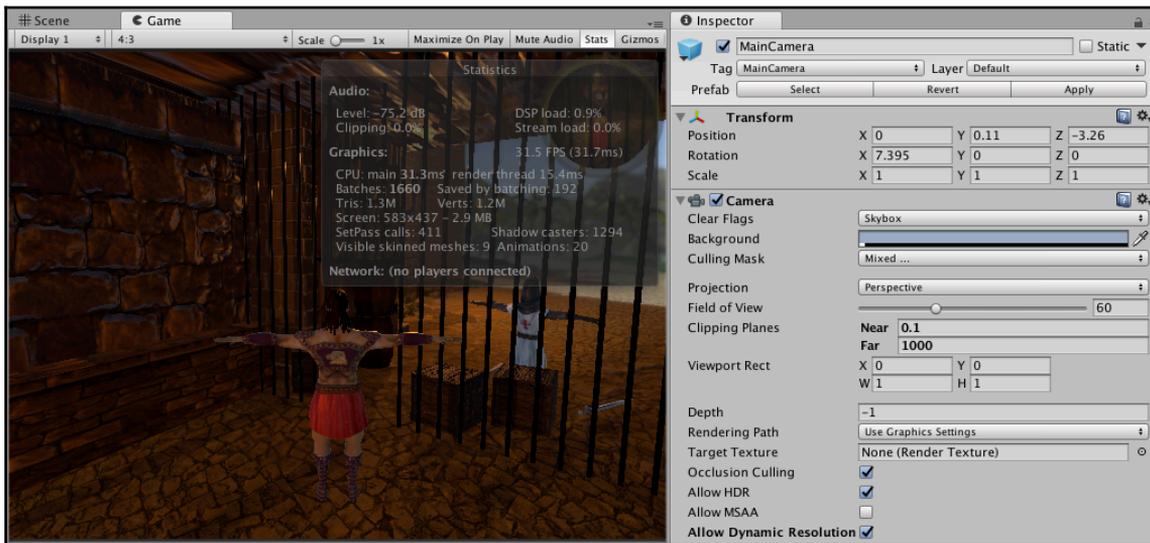


The results in terms of number of rendered triangles will be substantially different with Occlusion culling enabled on the camera.



If no occlusion data was calculated for the scene, enabling **Occlusion Culling** on the camera will not have any effect.

In the next screenshot, we can observe from the Statistics overlay the number of draw calls (Batches) and the number of triangles and vertices rendered with **Occlusion Culling** enabled on the camera:



2.2 million triangle with 1.8 million vertices drops to 1.4 million triangles with 1.2 million vertices to render the scene with Occlusion Culling enabled

Wrapping it all up

In this section, we will take care of the final look of the rendering of the main camera, we will see in-depth the difference between Deferred and Forward rendering and the Gamma and Linear color space, we will learn about Physical-Based Rendering, two older image effects, and the new Post Processing stack where we will replace all the old effects on the camera with the exception of the `SunShafts` effect, which is not replaceable with the new stack. We will also see how *GPU instancing* works and the difference between the **Dynamic Batching** and **Static Batching** features.

Rendering paths

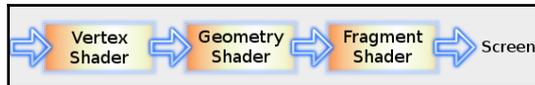
Unity supports two rendering paths techniques that are very different and that you need to understand to be able to choose which to use when designing your game.

On the official online manual, the table at the end of the page shows the various platforms and the compatibility with the two rendering techniques: <https://docs.unity3d.com/Manual/RenderingPaths.html>.

Graphics pipelines

To begin, we need to understand a little bit about programmable graphics pipelines.

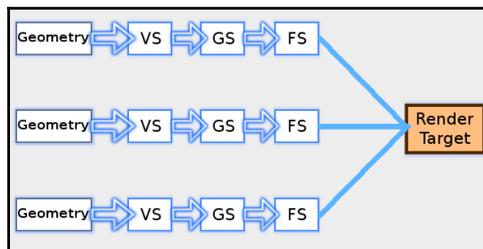
Before the third generation of accelerated graphics hardware, we were limited in what the video card graphics pipeline had at its disposal. We couldn't change how it drew each pixel, aside from sending in a different texture, and we couldn't warp vertices once they were on the card. But times have changed, and we now have programmable graphics pipelines. We can now send code to the video card to change how the pixels look, giving them a bumpy appearance with normal maps, adding reflections and a great level of realism. This code, written with CG or HLSL (high level languages for the SML: Shader Model Language), is first compiled and then given in machine language to the graphic card, in the form of geometry, vertex, and fragment shaders. They essentially change the way the video card renders your objects on screen:



Simplified view of a programmable graphics pipeline

Forward Rendering

Forward Rendering is the standard rendering technique that most engines use. You supply the graphics card the geometry, it projects it and breaks it down into vertices and then, those, are transformed and split into fragments or pixels, that get the final rendering treatment before they are passed to the screen raster:



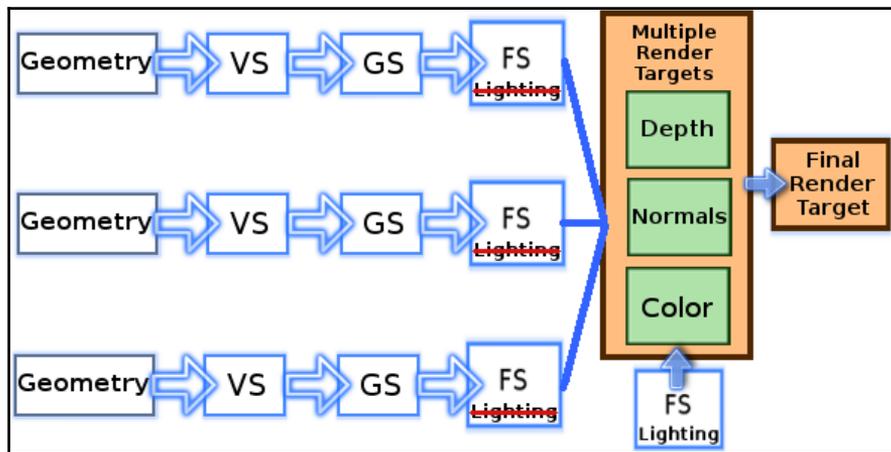
Forward Rendering: Vertex shaders to Geometry shaders to Fragment shaders

Each geometry is passed down the pipeline one at a time to produce the final image.

Deferred Rendering

In Deferred Rendering, the rendering is deferred a little bit until all of the geometries have passed down the pipeline; the final image is then produced by applying light shading at the end. When should we use this rendering technique?

This scheme below should help you understand how the Deferred Rendering technique works



Deferred rendering: Vertex shaders to Geometry shaders to Fragment shaders passed to multiple render targets, then shaded with lighting

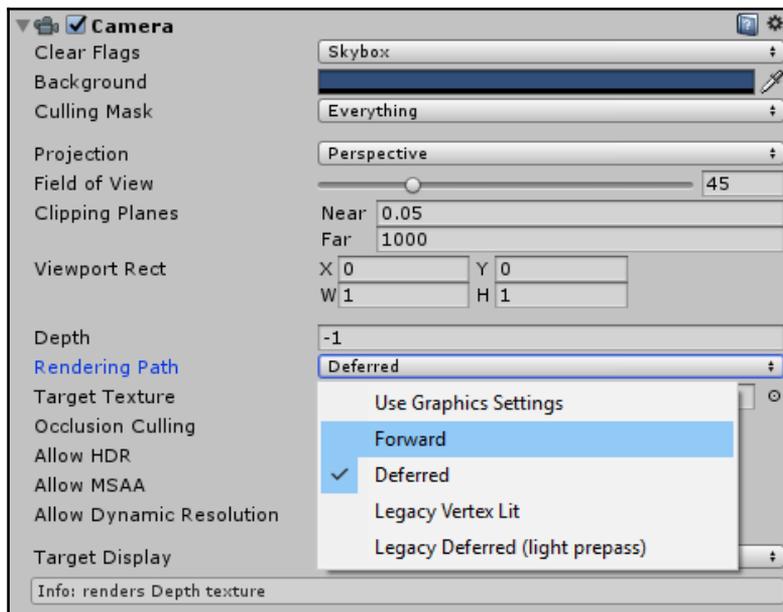
Lighting performance

Lighting is the main reason for choosing one route over the other. In a standard forward rendering pipeline, the lighting calculations have to be performed on every vertex and on every fragment in the visible scene, for every light in the scene.

If you have a scene with 500 meshes, and each of them has a geometry of 1,000 vertices, then you could easily end up having 200,000-300,000 polygons (approximate numbers). Video cards can handle this pretty easily nowadays, but when those polygons get sent to the Fragment Shader, that's where the expensive lighting calculations happen and the real slowdown can occur.

Developers try to push as many lighting calculations into the Vertex Shader pass as much as possible to reduce the amount of work that the Fragment Shader will have to do. You can optimize this by cutting out lights that are far away, or using lightmaps. But if we want to have many dynamic lights, we need a better solution, this is where Deferred Rendering comes into play.

The complexity of Deferred Rendering is given by the screen resolution (number of pixels) multiplied by the number of real-time lights. You can see that it doesn't matter how many objects you have on the screen that determines how many lights you use, so you can happily increase your lighting count:



The general setting for the render path is in the **Use Graphic Settings** tiers, but you can always override them on a camera basis by simply choosing a different **Rendering Path**.

When you are ready to release your game though, you want to leave the **Use Graphic Settings** option on the main camera of your scenes and let Unity decide what approach to take, depending on the platform and the hardware **Tier**. Tiers are defined to support different generations (and hence power) of the same platform. This will leave the developer the freedom to create while leaving the targeting of specific graphic features and settings automatized by Unity.

Should I use Deferred?

The short answer is, if you are using many dynamic lights, then you should use Deferred Rendering.

If you don't have many lights or want to be able to run on older hardware, then you should stick with Forward Rendering and replace your lights with static light maps. The results can still look amazing.

Physical-Based Rendering: Unity Standard Shader

Often you see graphic assets entries on the Asset Store, or developers and artists speak about PBR ready scene and PBR textures (materials). PBR stands for Physical Based Rendering and is a modern technique where the final look of a scene will be hit not just by a generic directional light in one direction but also take into consideration real physics light phenomena such as light refraction, specular reflection, and lights rebounding. PBR is achieved in Unity through real-time illumination and GI with lights hitting meshes in a scene that use the **Standard Shader**.

Unity's **Standard Shader** is very flexible and you can achieve almost any kind of surface by properly setting the Albedo texture, which is the color texture and, when needed, a normal map texture for bumped surfaces and you can specify Opacity maps, metallic or alpha shining parameters, and much more. Usually artists will provide meshes for Unity that are ready for these kind of materials along their artwork. For example, a modern character model would usually come with Color, Normal, and Specular texture maps to allow the proper rendering of all parts of the character: the skin, where it shines or is more opaque, the clothes, metallic parts on their gear, clothes, rubber, leather, and so forth.

There is also a specular version of the same shader called Standar Shader (specular) available for rendering special shining objects, such as metal, glass, and light-emitting objects, in a different manner. Check out both shaders and try one of your materials with each to see the difference. With the last 2017.3 release, Unity introduced two new shaders meant for particles: the Standard Surface and the Standard Unlit shaders.

Official manual link: <https://docs.unity3d.com/Manual/shader-StandardShader.html>.

With the requirements for photo-realism and special surfaces, such as semi-transparent rocky surface (salt), gel, or ice/snow, Unity's Standard Shader might be not enough. You might want to check out the Uber Standard Shader ultra, which is like Unity's Standard Shader on steroids. You can find it on the Asset Store at <https://assetstore.unity.com/packages/vfx/shaders/uber-standard-shader-ultra-39959>.

Image effects

Image effects are components that use special shaders to add to the main camera that will add a final pass of modification to the whole screen. We have initially seen them in the previous chapter while working on our main menu.

They are too many to enumerate and describe, so we will leave them for you to explore; we don't want to miss discussing what we intend to use for our game level: Sun Shafts and the Depth of Field effects.

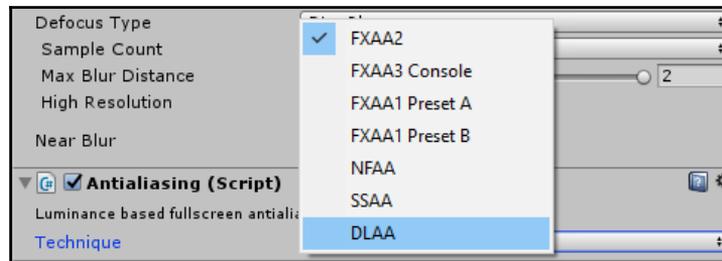
There are more useful effects for our cause, such as the Screen Space Ambient Occlusion/Screen Space Ambient Obscurance once, the bloom effect, and the Vignette effects, which we will explore later in this chapter because they are supported in the new stack, but before we look at these effects, I want to explore Anti-Aliasing.

Hardware-based anti-aliasing (MSAA) versus shader-based anti-aliasing(FSAA)

Something that video hardware has been trying to achieve for many years is a smoother vision of 3D graphics. In the history of real-time 3D, OpenGL, and, later, Direct3D, implemented features, such as shaded surfaces and smoothing groups, to make the 3D geometry look better. Something that was missing was antialiasing. As images and geometries are rendered with pixels on screens, no matter how small the pixels are and how large the video resolution is, the eye can see the pixel. While with 2D games and graphics, you can add antialiasing on the final images that you will import with Photoshop, *The Gimp* or such, it's another story with 3D graphics.

Here's where MSAA came in on modern graphics cards. MSAA can usually be set at different quality levels: single, X2, X4. The higher you go, the better the final quality of the rendering, and the more your GPU will sweat to make it. While graphic cards have all the MSAA hardware antialiasing features on board, it's always better to leave the choice to the end user, especially on PC platforms, where the hardware capabilities are so fragmented and different. For example, one user with top-notch hardware would like to be able to use it, while some other users, with less GPU power, or to obtain a higher framerate, would rather smooth the final image with **Full Screen Anti-Aliasing (FSAA)**, one of the many image effects you can throw in play on the main camera.

Despite what the user will choose in the settings, we need to add this component to the main camera, and eventually, disable it if the user enables **MSAA**. You can choose one of the available **FSAA** techniques to smooth the pixel feeling away; the top one—**FXAA2**—is the more GPU-hungry, while **DLAA** is the simplest and the lightest:



The list of available techniques for fullscreen antialiasing post image effect

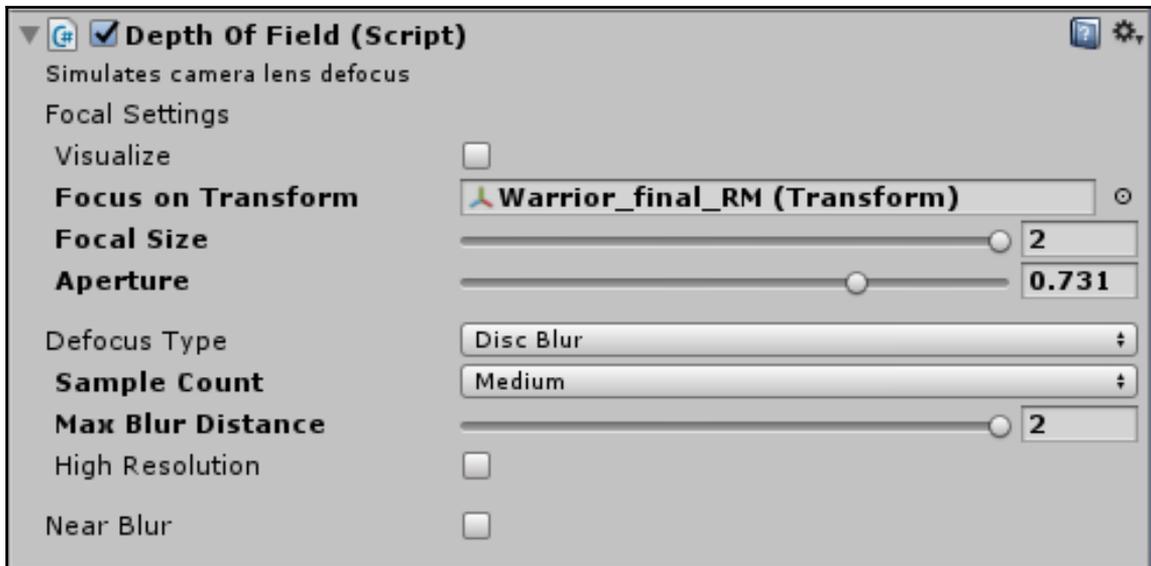


For an optimal final look and for the elimination of rendering flickers and camera scattering, we suggest you to don't use the older Image Effects **Antialiasing** component but the new Antialiasing module of the new Post Processing Stack profile set to Temporal Antialiasing. We will look into this right away later in this chapter.

Depth of field

Depth of field (DOF) is a common visual effect in real-world vision where things that are far away seem blurred, compared to the place where the eye gets the focus. More more information, refer to https://en.wikipedia.org/wiki/Depth_of_field.

DOF is intended to emulate this by blurring distant objects with the **Focal Size** and **Aperture** parameters:



Working especially well with DirectX 11 implementation (opposed to standard Disc Blur) and with HDR enabled, even though the result will come at a cost on the GPU.

Debugging depth of field

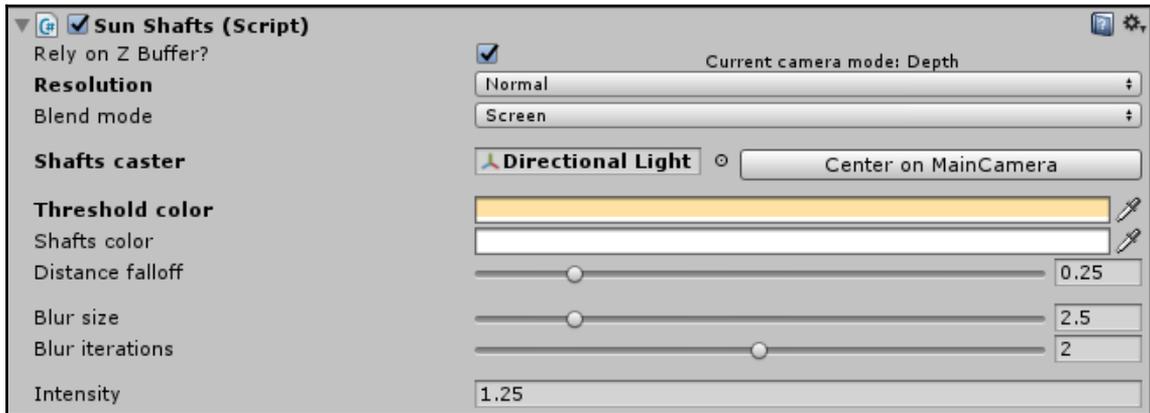
By selecting the **Visualize** option in the component, you can enter the debug mode for this effect to understand clearly when it comes into play with the distance.

Crepuscular sun rays through Sun Shafts effect

This image effects emulates the sun's crepuscular rays. It requires the current scene main directional light (the sun) to be specified in the Shaft Caster slot of the component and some fine-tuning of the other values.

We will attach this component to our Main Camera GameObject in our game level to obtain more atmosphere and a cool effect when the camera sees the sun and when some geometry occludes its rays.

We will set 1.25 for the **Intensity**, choose a warm yellow color for the Threshold color, and set the rest like in the following image:



This short video tutorial explains the Sun Shaft effect parameters and how to achieve the desired effect: <https://www.youtube.com/watch?v=seC3kHWExv4>.

The previous version documentation is available here: <https://docs.unity3d.com/550/Documentation/Manual/script-SunShafts.html>.

If you want to play around with another great third-party package about Volumetric Lighting, you may want to check out this GitHub space: <https://github.com/SlightlyMad/VolumetricLights/>.

Instead of going deeper into each of the older **Image Effects** collections, we are going to explore what's new in Unity 2017, a new and optimized post-processing image effects stack.

The Post Processing Stack

With the arrival of the VR platform, which demands a lot more GPU power than rendering just on the PC screen, but also with the more and more demanding necessity of today's games, last year Unity started to work on a new Post Processing Stack for rendering effects for the camera in a single step.

Using this package instead of the older Standard Assets Image Effects, which are still supported in Unity 2017, is mandatory for VR development for many technical reasons, but it is still better to use it because the older image effects will be dropped soon.

The Post Processing Stack package allows a strong performance gain compared to the old system, because a set of multiple effects is managed by a single final shader and will be performing way better than the previous solution, which is done with many different shaders playing simultaneously through different camera components. The new Post Processing Stack works by defining a profile that can be then used (tweaked or switched at runtime with Cinemachine Virtual Cameras) on the Post Processing Behaviour component that will be attached to the main camera.

The real advantage of this approach at edit time is that you look more at profiles that define the mix of effects you want to use to change the final image output instead having to remember the number of components to attach to a camera each time. This is really handy. The performance, the stability on more recent hardware, and the updated shader code make this system a must for any game developer on almost any platform.

There is so much to say about this package that is impossible to cover everything here. That being said, we will extend our knowledge by using it again in the next chapter in conjunction with Cinemachine.

Check out this video to recap all what we have said about making your game look better and also illustrate a good use of some of the effects of the stack: <https://youtu.be/owZneI02YOU>.

Post Processing Stack V2 and Utilities

At the time of writing, the official version of the package on the Asset Store is the V1, but we can still download V2 beta from this GitHub repository: <https://github.com/Unity-Technologies/PostProcessing>.

This asset package provides a utility to control the stack effects at runtime from the inspector, from scripting, and from the animation without messing with the saved profile.

With the utilities, you will have all the important stuff exposed into public variables shown in the **Inspector** so that will be possible to access them by scripting, by animation, and to be controlled with the new Timeline feature.

The use of the `Utilities` package is strongly recommended for obvious reasons, one of these is the Focus puller facility for the **Depth Of Field** effect. If we want to port **Depth Of Field** to the new stack as well, we need to do the following:

- Change the project assets for switching from Post Processing Stack V1 to Post Processing Stack V2

- Download and import the Post Processing Stack Utilities additional package

Focus puller

The focus puller utility for the processing stack is a utility intended for the Depth Of Field effect of the stack V2.

It will automatically recalculate parameters following the given transform target. (Just like the Standard Assets version is doing.)

This will help in always having the correct camera focus when the target is moving around. If you don't use Focus puller, or you are on the V1 of the stack, the Depth Of Field effect must be performed with the older Disc/DirectX11 image effect.

More info about the utilities can be found on this GitHub page: <https://github.com/keijiro/PostProcessingUtilities>.

An alternative to this approach, if you want to use the Depth Of Field effect from the Stack, is to use V1 and a Cinemachine camera for your gameplay. Because the Post Processing Stack V2 is still beta, in the next chapter, we will use this approach instead to fix the camera focus dynamically as the target moves away in our final cut scene.

Advanced rendering features

These are some of the more interesting advanced rendering features of Unity engine, they are meant for advanced users and here we will just overview them. I suggest you have a deeper look at them on the Unity official online documentation.

Level of Detail (LOD)

When a GameObject in the scene is far away from the camera, the amount of detail that can be seen on it is greatly reduced. However, the same number of triangles will be used to render the object, even though the detail will not be noticed at all.

A great optimization technique called **Level Of Detail (LOD)** rendering allows you to reduce the number of triangles rendered for an object as its distance from the camera increases. As long as your objects aren't close to the camera, LOD will reduce the load on the hardware and improve general rendering performance.



SpeedTree, the tree system we saw earlier, uses the same technique without any action required by the developer.

For other GameObjects, such as buildings, AI, or other objects with a complex geometry, we would use the **LOD Group** component to set up LOD rendering for an object.

Check out the official documentation: <https://docs.unity3d.com/Manual/LevelOfDetail.html>.

High Dynamic Range

High Dynamic Range (HDR) was seen in the previous chapter where we set the chance for the user to enable in the video options menu. We chose a Deferred Rendering path for the game hence no hardware anti aliasing for many reasons, one of this is the ability to have a good realism with the use of HDR and other post effects.

For a full look at the official manual page, visit <https://docs.unity3d.com/Manual/HDR.html>.

Asynchronous Texture Upload

Asynchronous Texture Upload enables asynchronous loading of Texture Data from disk and enables a time-sliced upload to GPU on the Render-thread. This feature reduces the wait for GPU uploads in the main thread. It will automatically be used for all Textures that are not read-write enabled, so no direct action is required to use this feature. You can, however, control some aspects of how the asynchronous upload operates and some understanding of the process is useful to be able to use these controls.

For more info, head to: <https://docs.unity3d.com/Manual/AsyncTextureUpload.html>.

Graphic Command buffers

Command buffers give the possibility to extend Unity's rendering pipeline with so-called *command buffers*. A command buffer holds a list of rendering commands (set render target, draw mesh, and so on), and can be set to execute at various points of the pipeline.

For example, you could render some additional objects into the G-buffer after all regular objects are done. For a high-level overview of how cameras render scene in Unity, images, and code examples, go to <https://docs.unity3d.com/Manual/GraphicsCommandBuffers.html>.

Unity Engine automated optimizations

Unity brings automated optimizations available by simply checking an option in the player settings like we will see in the next chapter. Static Batching, Dynamic Batching, and GPU Skinning are automatically performed when you enable them in **Player settings**. Some others optimizations, such as GPU Instancing, which is performed on a per-material basis or layer-based camera culling, require some other bits to be set. These are the most important, in terms of draw calls reduction and graphic optimization in general. Let's look at what they do in detail.

Static and dynamic batching

These two engine features are different. Both are intended for improving game performance, but play a role in different areas. While static batching is born for optimizing the static-level geometry, dynamic batching is meant for optimizing particles and small objects rendering. We will modify the `SmallStone` prefab, implementing GPU instancing to analyze the performance of the various implementations.

Static batching

Static batching is a process automated by the engine at build time that will assemble static meshes geometry that shares the same material into a bigger mesh. This will save a lot of calculation, especially in huge maps with both indoor and outdoor environment and a lot of buildings; in short, it will help make the frame rate better.



Beware: If your game uses simple frustum culling or a combination of frustum and occlusion culling with a top-down view, it may probably better not to let the objects merge; in this and other very special cases, you may want to disable this feature.

Dynamic batching

Dynamic batching is a process automated by the engine at runtime that will try to draw a large amount of identical objects on screen with fewer draw calls. The typical cases are bullets, particles, stones, and so forth. The first level of optimization for our stone is to use a simple second-generation shader that supports dynamic batching for the stone and ensure that the mesh geometry doesn't overpass the limit of 300 vertices.

To make this test, we will disable the destroy on timer feature on the stone prefab, in this way letting them always onscreen, they will grow in number and we will see the difference with the optimization running and without.



Currently, only Mesh Renderers, Trail Renderers, Line Renderers, Particle Systems renderers, and Sprite Renderers are batched. This means that skinned Meshes, Cloth, and other types of rendering components are not batched. Renderers only ever batch with other Renderers of the same type. Semi-transparent Shaders usually require GameObjects to be rendered in back-to-front order for transparency to work. Unity first orders GameObjects in this order, and then tries to batch them, but because the order must be strictly satisfied, this often means less batching can be achieved than with opaque GameObjects.

Often, manually combining GameObjects that are close to each other can be a very good alternative to draw call batching.

You can do so in your digital content creation tool or with dedicated Asset Store extensions.

Activating the statistics overlay in the **Game** view, while testing the scene, will now show fewer draw calls after 100 stones were instantiated and left on screen. **Batch** indicates the number of effective draw calls needed to render the scene, while **Saved by batching** indicates the number of draw calls saved by the batching.



This approach works well in **Forward** mode.

GPU instancing

GPU instancing is an advanced rendering feature but was included in this section for its vicinity with the Dynamic Batching concept. It was introduced some time ago with version 5.6 and we will see how it works with our rocky stones.

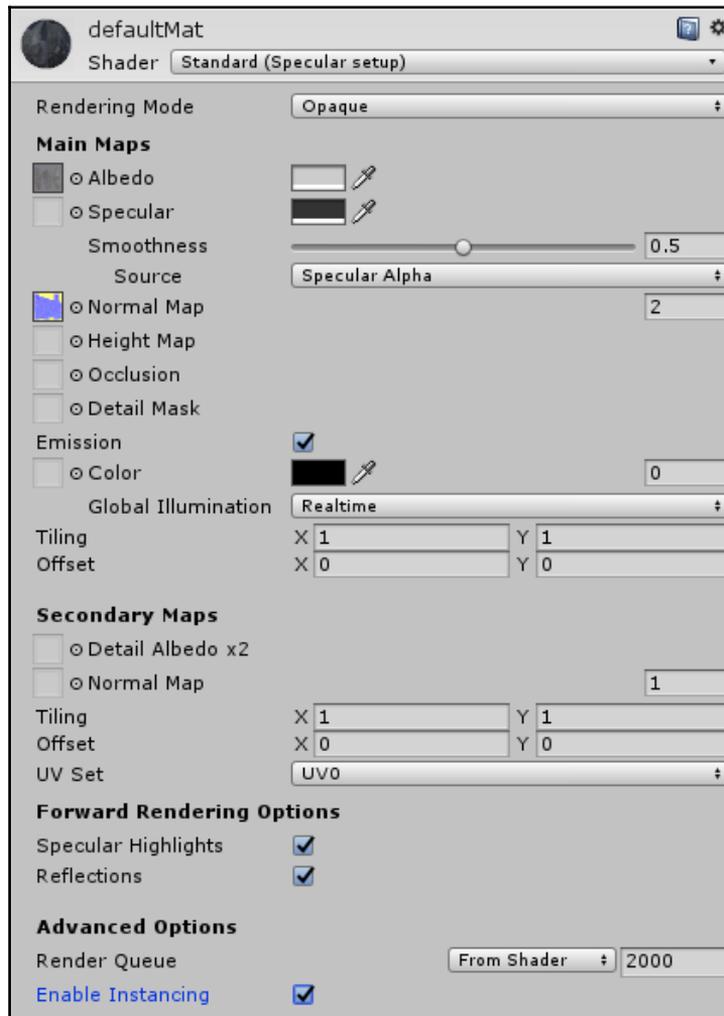
Let's take our `SmallStone` prefab as an example. We wish to render as many stones as we like to give the game a bit of an arcade feeling, but we are afraid of the final poly count and the many draw calls the engine will need to perform when hundreds of them are eventually lying on the ground and within the camera frustum cone, hence rendered.

In [Chapter 8, AI, NPC, and Further Scripting](#), we saw how to destroy the object after a desired time, to avoid frame-rate drops after many stones were thrown and hence instantiated. By design, we may want to increase that time for our game or even leave them all on the ground forever and this is possible, thanks to GPU Instancing.

GPU Instancing is a technique that uses a particular shader that will compute the geometry on the hardware, resulting in a much faster rendering of thousand objects with the same shader. There are some considerations to make, but essentially, this plays where the horsepower of modern graphics cards lie, your identical objects must share the same mesh and the same material.

To achieve this on our stone prefab in Unity 2017 is really simple, similar to changing an option on the stone material. Search for `SmallStone` in the **Project** view and select the prefab with that name (the `SmallStone.prefab` file) in the inspector, find the Mesh Renderer component and click the material assigned in the slot. Doing so will point out the file in the **Project** view, making it blink for a second and put the focus on the container folder.

Now, select the `defaultMat.mat` file in the project view, and then check the **Enable Instancing** option at the end of the panel in the inspector, as you can see in the next screenshot:

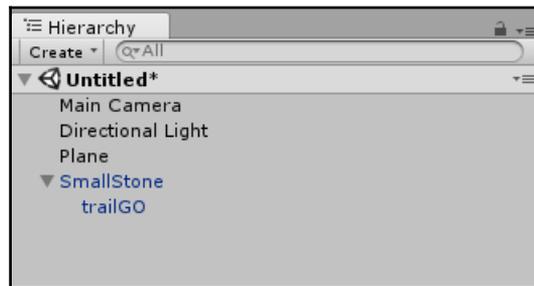


To make our implementation even better, we want to have the minimum number of draw calls for those stones. Imagine a multiplayer game where each player launches a multitude of stones. Each `SmallStone` prefab requires two draw calls: one (or more, depending on the shader) for the rock material, and one for the Trail Renderer material.

In the first implementation, we can use the `autodestruct` parameter in the Trail Renderer component to also destroy the stone after a given time.

This parameter will destroy the `GameObject`, not the component. We want to control the trail destruction separately from the stone itself, which is now kept on screen for a long time. To test all this, create a new test scene; we can call `GPUStressTest`, add a `Plane` in the scene, scale it at `10,1,10`, and place it at `0,0,0`. Add our `shooter.cs` script component from Chapter 1, *Entering the Third Dimension*, to the camera. Drag the `SmallStone` prefab in the component slot, and then click play and verify that you can shoot stones with the left mouse button and move with the arrow keys as we could do in the test scene we made back in that chapter.

Click **Stop** and then drag a `SmallStone` prefab in the hierarchy. Select it in the **Hierarchy** and from the main menu, select **GameObjects | Create Empty Child** to create an empty child; finally, rename it to **trailGO** (trail `GameObject`), as in the following screenshot:



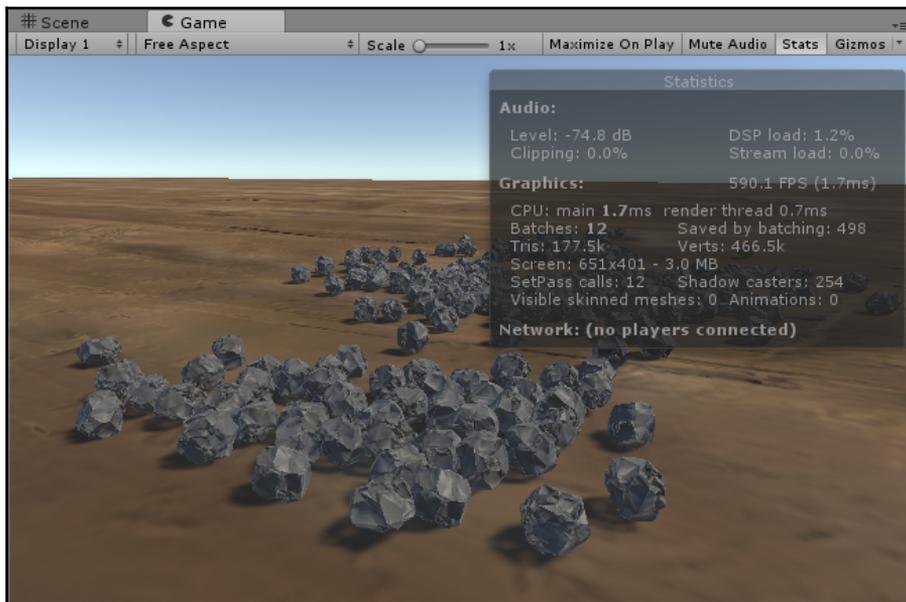
Copy the Trail Renderer component from the parent by right-clicking the component and selecting **Copy component**; then, select the **trailGO** child and, in the Inspector, right-click the Transform component and select **Paste Component As New**.

Now go back to the parent, and remove the Trail Renderer component by right-clicking it and selecting **Remove Component**. Finally, check the **Autodestruct** option in the `trailGO` `GameObject`'s Trail Renderer component.

Now, we can click **Apply** on the parent to save our edit on the prefab, and then safely delete the instance from the hierarchy and click **Play** to test.

Before we launch, show the Statistics overlay on the **Game** view, and then start throwing stones; throw a lot of them and see how after half a second, each of them in the hierarchy will lose the children (due to the auto destruct option on the Trail Renderer component of the `trailGO` GameObject). Keep on shooting, clicking very quickly, to spawn a lot of them. You should see a very large number of batched draw calls and a very small number of draw calls now.

The statistics overlay shows **12** batches and **498** saved by batching, occurring because we enabled GPU Instancing for that material, and the meshes are identical:



Let's run the same test, but before clicking the Play button, disable the GPU instancing flag on the `defaultMat` material.

The difference is quite visible on the statistics overlay. This technique can be useful for many purposes and many different types of games; it's up to the game designer to choose when to use this feature and how deeply to use it in a game.

If you run the same test with *GPU instancing* disabled on the material, you will notice that in the Statistics overlay **516** batches with no **Saved by batching** is displayed.

Now that the prefab is optimized and ready to play in the game, go back to the main level and test the improvement yourself.

To get a deeper look at GPU Instancing, head to Unity official online documentation at <https://docs.unity3d.com/Manual/GPUInstancing.html>.



This technique is meant to be used in deferred rendering.

Cull and cull more!

One of the interesting things that the Unity engine can do, which is often underestimated, is the possibility to filter out the rendering from a camera of specific layers at a given distance.

Or, again, the possibility to filter out the impact of a light, and having to hit only the `GameObject` of certain layers.

Used proficiently, this combination of optimizations will benefit the final framerate.

An example is the reflection camera for the water around our island. We could filter out from the rendering, for example, if we had manually dragged and positioned the Speed Tree prefab one-by-one, instead of painting them on the terrain. In that case, we could have set a layer for the `GameObject` of the tree and avoid rendering all that beauty in the sea reflection. But again, this is a design choice, a game might want to have the trees reflected in the water for some reason, so this optimization cannot be implemented. In the same way, though, you could set a layer to buildings, trees, or anything that is far from the seashore, because it would be never reflected in the water. Still, the engine will also take those objects into account if you don't set the camera cull mask properly.

One of the more GPU-intensive objects we added in our map is the multiple particle system fireplace and fire torch prefabs. It is GPU-intensive, especially if we had chosen Forward Rendering instead of Deferred, because each fire has a light that also moves around and cast shadows. It would be ideal to hide them when the camera is far from them, considering a minimum given distance for activating them back.

Camera culling distance

Another interesting optimization and automation of this process is a feature on the underestimated `Camera.Layercull` API.

Setting a specific layer or a group of specific layers assigned to the culling and giving a linear floating point precision distance to the camera. The official manual states:

Normally Camera skips rendering of objects that are further away than `farClipPlane`. You can set up some Layers to use smaller culling distances using `layerCullDistances`. This is very useful to cull small objects early on, if you put them into appropriate layers.

When assigning `layerCullDistances`, you need to assign float array that has 32 values. Zero values in cull distances means "use far plane distance". By default, per-layer culling will use a plane aligned with the camera. You can change this to a sphere by setting `layerCullSpherical` on the Camera to true.

And this is exactly what we will do!

Create a new component on the Main Camera GameObject and call it `LayerDistanceCulling`. This class is fairly simple, and use only the `Start()` method to set the `layerCullingDistance` on a specific layer (15) which is defined as the 15th element in the **Tag & Layers** settings (from the top menu: **Edit | Project Settings | Tag & Layers**):



The Layers section of the Tag & Layer settings

The class code will look like this:

```
using UnityEngine;

public class LayerDistanceCulling : MonoBehaviour {

    void Start ()
    {
        Camera camera = GetComponent<Camera>();
        camera.layerCullSpherical = true;
        float[] distances = new float[32];
        distances[15] = 35;
        camera.layerCullDistances = distances;
    }
}
```

Press play and test the game. You should notice that when your hero is at a distance of more than 35, the fire torch will disappear. The downside is that the light will be still there, enlightening and shadowing the surroundings!

Modifying the FireLight class

A GameObject with a Light component will not be culled by the rendering engine with the layer distance culling method, so we need to disable the light manually.

While fires particles systems and the torch mesh itself are batched by the engine, the real-time point lights used by the fires are the real GPU-hungry buddy here when using Forward rendering. With Deferred lighting, the framerate would not be affected that much, like we explained earlier, but the difference with many fire lights active in the scene, will still be noticeable at high resolutions.

We will modify the already existing script we introduced back in Chapter 11, *Unity Particle System*, that manages firelight color and movement animation to implement a similar feature that will also work with lights.

We will add a new private float variable, `linearDistance`, to hold the calculation of the distance from the camera and add two new public variables to expose in the **Inspector**:

```
public float renderDistance; // after this distance disable the light
public bool startBurning; // should this fire light start enabled ?
```

At the very start of the `Update ()` method we will add:

```
linearDistance = Mathf.Abs(Vector3.Distance(transform.position,
Camera.main.transform.position));
```

For calculating the linear distance between the fire and the camera and right after that line a simple check, to switch the fire on and off:

```
if (linearDistance > renderDistance) Extinguish();
if (linearDistance <= renderDistance) SetOnFire();
```

Add the following methods at the end of the class:

```
public void Extinguish()
{
    m_Burning = false;
    m_Light.enabled = false;
}

public void SetOnFire()
{
    m_Burning = true;
    m_Light.enabled = true;
}
```

We can optimize the code a bit, again by using a single method for the switch:

```
public void SwitchFire(bool switchFlag)
{
    m_Burning = switchFlag;
    m_Light.enabled = switchFlag;
}
```

Pass true/false to it from the code in `Update()` that we will change accordingly:

```
if (linearDistance > renderDistance) SwitchFire(false);
if (linearDistance <= renderDistance) SwitchFire(true);
```

This will cull the lights as well if the same distance given for `Camera.layerCullDistances` is set for the `renderDistance` variable. In this case, we will put 35 in the **Inspector**, to reflect what we wrote in the previous component.

Let's dive into some code to optimize other aspects of the game.

Further optimizations

Let's take a deeper look at how we can make the general performance of our game better through some essential tips and modifications. We will look at physics limits and optimize the starting number of AI active thanks to special trigger areas that will activate sleeping AI and in the end, add a graphic trail to the `SmallStone` prefab.

Physics optimizations

We will take a look at another important optimization that makes strong use of physics calculations.

The number of concurrent rigidbodies you can have alive on screen is unlimited. With the exclusion of Kinematic (the `isKinematic` property) rigidbodies, you have to consider the limits of how many objects you can have on scene with an active physics simulation running. The following table shows different physics setups with different numbers of rigidbodies with different values and options and the impact they will have on the framerate.

The table linked in this [google docs spreadsheet](https://docs.google.com/spreadsheets/d/12xfrcmWhFJfIyQFjjtjAlbVo_C9wz61jozi3Pkfo5Y) shows how much the number of rigidbodies impacts the performance with various different settings: https://docs.google.com/spreadsheets/d/12xfrcmWhFJfIyQFjjtjAlbVo_C9wz61jozi3Pkfo5Y.

By looking at this data, you can understand why, if we keep launching stones that are never destroyed, the framerate will drop pretty soon. To avoid this situation, we must think in two possible directions:

1. Limit the number of rigid bodies present concurrently in a scene by design
2. Write a manager class that takes care of *capping* the number of active rigidbodies concurrently

While the first option seems easy to achieve, it might be not an option for some games. In this case, a good bet would be to write your own physics manager, that performs the following:

- Disable ragdoll's rigidbodies after a certain time
- Disable stones' rigidbodies after a certain time or when their rigidbody is sleeping
- Disable any rigidbody attached to objects that are not in view, or far from the camera

Careful with the third point, because it might lead to objects magically appearing or magically taking lives.

We are not going that far in this book, but we will doing the following in the same direction:

- Create a new class for disabling redundant rigidbodies, such as the ragdoll simulation, and we will attach it to the `GameObject` of the fainted guard near the jail and apply a similar approach for enabling/disabling the ragdoll of the guards.
- Change our `HeavyStone` class to don't destroy the entire `GameObject` but just the `Rigidbody` and `Collider` components after a certain time instead. This could be cool with GPU instancing for making the environment more variable while still preserving a good framerate.

Writing the `DeactivateRagdollTimer` class

This class will be fairly simple. We will use the `Invoke` statement at the start of the script to call the `DeactivateRagdoll()` synchronous method after a given time (`delay`):

```
using UnityEngine;

public class DeactivateRagdollTimer : MonoBehaviour {

    public float delay;

    void Awake () {
        Invoke("DeactivateRagdoll", delay);
    }
}
```

This method will take care of setting all rigidbodies in the children of the guard `GameObject` `isKinematic` property to true, to disable calculations and disable collisions for all of them. Also another iteration will take care of disabling all the colliders attached to the same objects. Here follows the rest of the code for the class:

```
// A method that deactivate all components needed for ragdoll physics
private void DeactivateRagdoll()
{
    // Find every Rigidbody in character's hierarchy
    foreach (Rigidbody rb in GetComponentsInChildren<Rigidbody>())
    {
        // Set bone's rigidbody component as kinematic
        rb.isKinematic = true;
        // Disable collision detection for rigidbody component
        rb.detectCollisions = false;
    }
    // Find every Collider in character's hierarchy
    foreach (Collider col in GetComponentsInChildren<Collider>())
    {
```

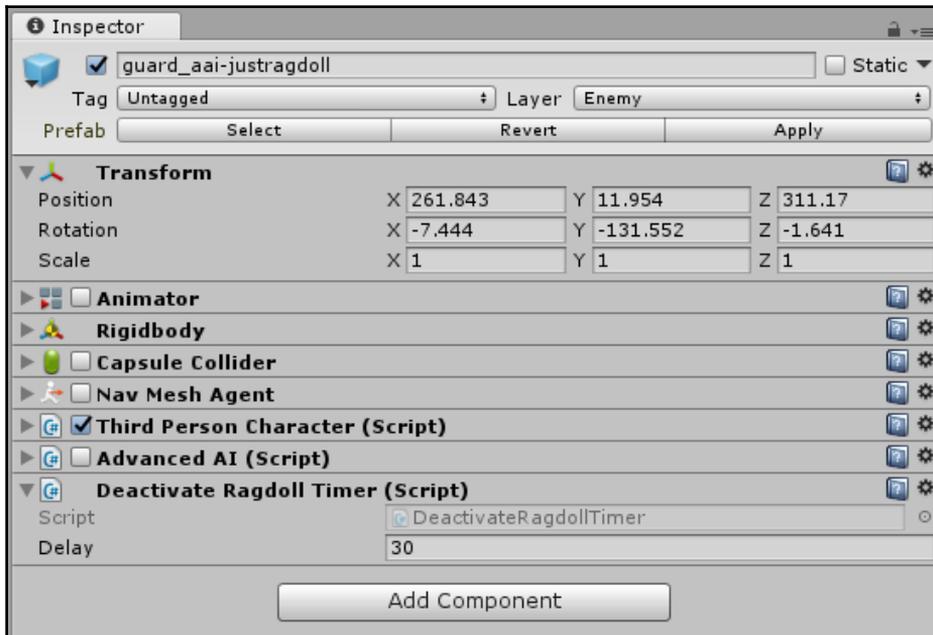
```

        // Disable Collider
        col.enabled = false;
    }

}
}

```

The `DeactivateRagdollTimer` component attached to the `guard_aai-justragdoll` seen in the **Inspector**. All Animator, Capsule Collider, Nav Mesh Agent, and Advanced AI components attached to it will be disabled at the start to let this AI act as a feinting guard:



We had kept the class generic and chose to simply set the `isKinematic` property of the `Rigidbody` to skip physics calculations, but for our dead guard out of the jail, we could have also destroyed the `Rigidbody` components and `Collider` component collections.

Let's modify the code to achieve this. First of all, we want to have the option to destroy the components instead disabling them, to be able to reuse this class for different purposes, so we will add a public bool variable: `destroyComponents`.

We will check against what we set in the inspector for this variable, to decide what to do in the `DeactivateRagdoll` method for rigidbodies and colliders:

```
// A method that deactivate all components needed for ragdoll physics
private void DeactivateRagdoll()
{
    // Find every Rigidbody in character's hierarchy
    foreach (Rigidbody rb in GetComponentsInChildren<Rigidbody>())
    {
        if (destroyComponents)
        {
            Destroy(rb);
            // Set bone's rigidbody component as kinematic
            rb.isKinematic = true;
            // Disable collision detection for rigidbody component
            rb.detectCollisions = false;
        }
        else
        {
            // Set bone's rigidbody component as kinematic
            rb.isKinematic = true;
            // Disable collision detection for rigidbody component
            rb.detectCollisions = false;
        }
    }
    // Find every Collider in character's hierarchy
    foreach (Collider col in GetComponentsInChildren<Collider>())
    {
        if (destroyComponents)
            // Destroy Collider
            Destroy(col);
        else
            // Disable Collider
            col.enabled = false;
    }
}
```

And that's it! Save the script and test the game level. You should see the guard perform the ragdoll evolution and, after the given number of seconds (I chose 30 seconds but it could be less) you should be able to walk into the guard body without moving it after that time. This will save a lot of CPU cycles for other live rigidbodies needed by the game.

Changing the HeavyStone class

To be able to leave thrown stones in the scene, but disable or destroy only its Rigidbody and Collider components after a given time, we will proceed in a similar way.

Open the `HeavyStone` class for editing. We will change the name of the public float `destroyTime` variable to something more appropriate for the scope:

```
public float interactionTime;
```

Then we will also add three private variables, to cache the components to optimize the script:

```
private Rigidbody rb;  
private AudioSource audioSource;  
private SphereCollider sphereCollider;
```

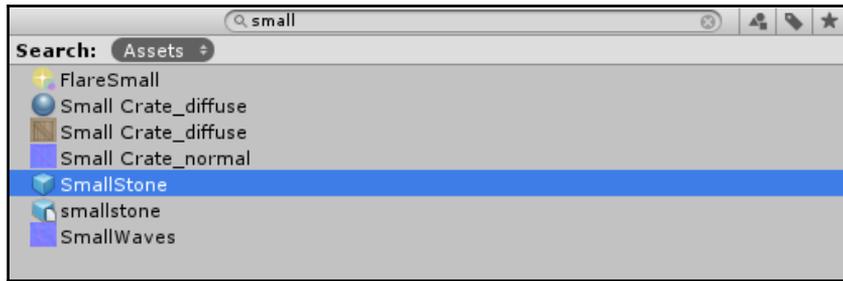
We will remove the `Required` directive to require the `AudioSource` component, if we want to destroy that as well.

Then, we will change the `Start()` method in this way:

```
// Use this for initialization  
void Start()  
{  
    // we initially cache Rigidbody and AudioSource components  
    audioSource = GetComponent<AudioSource>();  
    rb = GetComponent<Rigidbody>();  
    sphereCollider = GetComponent<SphereCollider>();  
  
    // then set the collision sound  
    if (audioSource != null) audioSource.clip = collisionSound;  
    // destroy rigidbody after a given time, to save CPU  
    if (rb != null) Destroy(rb, interactionTime);  
    if (sphereCollider != null) Destroy(sphereCollider,  
    interactionTime);  
    if (audioSource != null) Destroy(audioSource, interactionTime);  
}
```

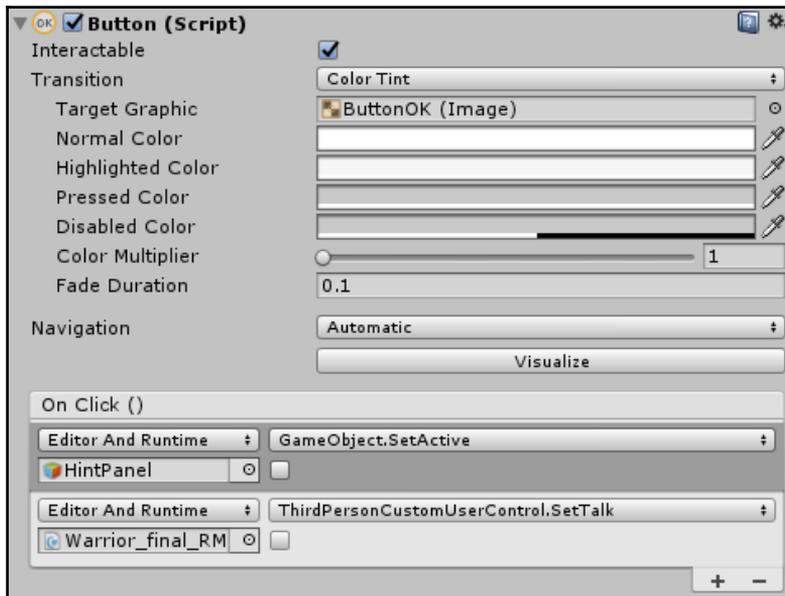
This change will destroy the components instead of the whole `GameObject` after the given time. Using a fairly large time for the interaction to stop is crucial, because we want the stones to finish their evolution when launched, before we stop the physics simulation by destroying the `Rigidbody`.

Save the script and go back to the editor, then, search the `SmallStone` prefab in the **Project** view, select it and in the **Inspector**, put 5 (seconds) in the **Interaction Time** field, so it will be set for each future instance of it:



The search in the Project view of the word small reveal all the assets with that word in the name

We want to test the results, but is really annoying to repeat the initial dialogue sequence every time. To be able to run and launch stones immediately at the start, the hero must be healed without the need to talk to the old man. To perform this, we will add an action on the `OnClick()` event of the Button component that dismisses the initial hint panel:

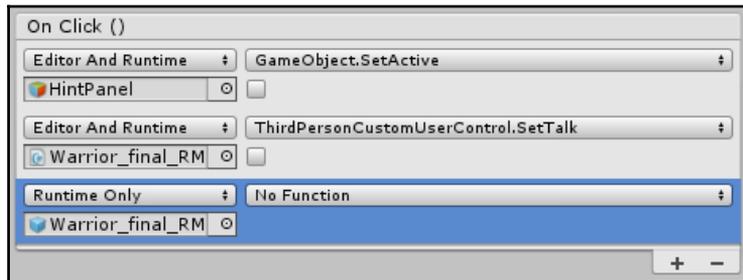


The HintPanel dismissal button component in the Inspector, before adding this testing-oriented feature

This Button already has two actions: the **HintPanel GameObject.SetActive** (false) called on the `GameObject` itself and `SetTalk` (false) call on the `ThirdPersonCustomUserControl` component attached on the hero that locks/unlocks the player's movement.

Add a new Item with the + (plus) button at the bottom. Then, drag the `Warrior_final_RM` `GameObject` into its slot. Note that the icon, differently from the previous call, is the prefab icon, because the system doesn't know what you are going to do with it.

Finally, from the drop-down menu, we choose `ThirdPersonCustomUserControl.HealAndHelp`. Note the icon in the slot changed into the C# document one:



The new action for the `OnClick()` UI event shows the prefab icon for the `Warrior_final_RM` `GameObject` in the Inspector

Now we are able to press play and immediately test the new behavior of the stone, when launched in the middle of nothing or at the head of a guard, without the need to walk slowly to the old man and have the dialogue.

Remember to remove that action when you want to test the real thing. You could also duplicate the button `GameObject` and call it `ButtonOK_TEST`, so that you can quickly switch from one test mode to the other.

Exercise proposal: a rough save game-status feature

The testing feature we implemented opens up the need of a basic save system for our game.

As an additional exercise, you could implement a series of `PlayerPref.SetInt("Checkpoint", GameState.currentGameState)` calls where you save the game status at each important point of the game for then restore the status when the game starts another time. `GameState` would be a struct, a class, or a simple enumerator of the `GameState` type, which will enumerate various points in the game. We would ideally add this feature in the `PlayerInventory` class, for example:

```
private enum GameState
{
    GameStart,
    Healed,
    FirstPiece,
    SecondPiece,
    ThirdPiece,
    FourthPiece,
    GameEnd
}

private GameState gameState;
```

In this way, your game will be able to read and write game status easily from anywhere in your code through the `PlayerPrefs` API. In the end, you should take care to save the player position (for example: with `PlayerPrefs.SetFloat(x)`, `PlayerPrefs.SetFloat(y)`, `PlayerPrefs.SetFloat(z)`) and the number of artifact pieces that were already collected. Because the guards are never killed, just stunned, you can void recording the status of the AI.

To restore the saved point, we will use `PlayerPref.GetInt()` calls to retrieve the saved data and apply it to the scene at start by restoring player position, assign the already-collected number to the `PlayerInventory`, and eventually disable the `GameObjects` of already-collected pieces (this requires more player prefs to be saved to remember which one of them was collected).

Optimizing AI impact on the CPU

AI Nav Mesh Agents, along with their Skinned Mesh Renderer components can use a lot of CPU and GPU cycle time to calculate, not only because of NavMesh path finding, but also for physics collision and rendering.

The island is quite big, there is no point in having all of the AI active at the beginning if the camera doesn't even see them!

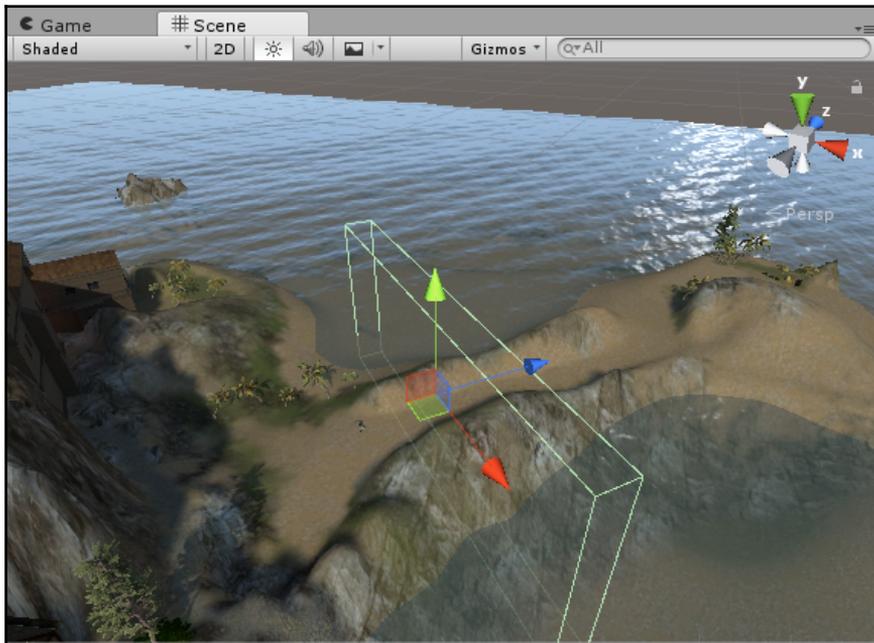
Setting up AI Trigger areas

AI trigger areas will allow us to switch on (activate as in `SetActive(true)`) specific enemy AI when the player enters a specific trigger.

There are different approaches to solve these problems; we will keep it simple and create big trigger areas that, when entered, enable a list of given AI instantly. It's important to place these areas in a way that will prevent the player from spotting the AI spawn:

- Create a new Cube primitive `GameObject` and rename it `AITrigger`
- Delete the mesh renderer and mesh filter components of the cube
- Scale it and position it as shown in the next figure

When we have finished, we can resized our `GameObject`, or the collider itself, to obtain something like in the following picture:

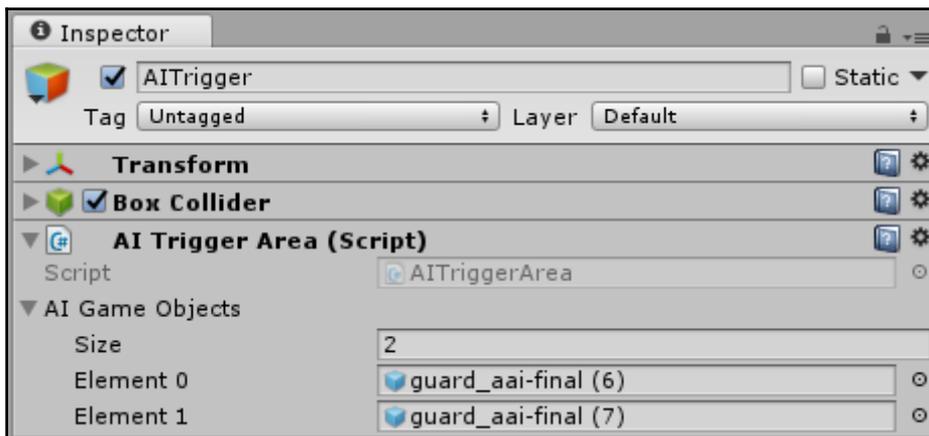


Add a new component for the cube trigger and call it `AITriggerArea`. Open the new script in the Editor and replace the existing code with the following:

```
using UnityEngine;
public class AITriggerArea : MonoBehaviour {

    public GameObject[] AIGameObjects;
    // Check collision with player's capsule collider
    void OnTriggerEnter(Collider col)
    {
        if (col.gameObject.layer == 8) // 8 is the layer id for
            player
        {
            foreach (GameObject AIGO in AIGameObjects)
            {
                if (!AIGO.activeInHierarchy)
                    AIGO.SetActive(true);
            }
        }
    }
}
```

The code is self explanatory: when the player enters the trigger, all the `GameObjects` in the `AIGameObject` array will be activated if not active already (`if (!AIGO.activeInHierarchy)`). Save the script and go back to the **Inspector**. Put the 2 in the **Size** field in the **Inspector** and then drag the two AIs we have put in the beach area near the second artifact piece into the **Element 0** and **Element 1** slots of the array:



The Inspector will now allow you to choose any number of AI enemy you want to activate

Repeat this for the abandoned village enemies and for the ones on the top of the mountain near the lake for some practice and finish the job. As an exercise, you could set up another kind of area trigger to deactivate the AI again when you exit a certain area and you don't need to see them anymore.

Other ideas

You want to go further in the implementation of AI Trigger areas and also create triggers for disabling the AI. Another idea is to have only four guards in the whole scene, and recycle them in the various spots of the level, but four may also not be enough if some of them follow you until a new checkpoint. This is all about game design and level design, which are the most exciting parts of game development because the game satisfaction will be strongly influenced by the decisions taken in these two fields. Another good call would be a custom LOD solution where not only the model quality will decrease with the distance, but also the quality of path finding and the RigidBody will be disabled.

Summary

In this chapter, we learned how to optimize the scene for final rendering and how to obtain better performances and a higher frame rate. Frustum culling, static batching, occlusion culling, dynamic batching, and GPU Instancing are all concepts that you may want to investigate further while you go through your Unity projects!

We have added some cool final touches exploring the new `Post Processing Stack` package from Unity Technology aimed for replacing the older `Image Effects`. We added also another bit and wrote some code for destroying rigidbodies, when not needed, to optimize the physics calculations in a scene such as AI trigger zones.

Now that we have completed the game, we'll spend the next chapter looking at building, testing, rebuilding, and the implications of deploying your game. We'll take a look at graphic and build options for different platforms and discuss getting your game seen around the world as an independent developer.

14

Building and Sharing

We have learned about Unity in almost all its aspects with what aims to be a full guide to an engine always in expansion and running towards the latest emerging technologies. Now it is time to finalize our work and get it ready for publishing. As the platforms vary a lot in terms of hardware and hence performance, you will find a corresponding section for each platform. This book should have given you enough to start exploring the amazing world of developing computer games.

In order to take our game from a simple example to something that we can share with play testers, we need to consider various platforms of deployment and how we can adapt the game to be exported to the WebGL versus exporting it as a standalone desktop game. The best way to start working in a fashion that will help you grow as a developer is to share your work; one of the most important elements of creativity is being able to accept other viewpoints and allow these to enrich you as a creative developer. This is why Unity's ability to export to the web is so important, and why we will look at how to export for both platforms in this chapter.

Unity allows for various scaled qualities of the final build of your game and will compress textures and various other assets as appropriate for the platform you choose. You should also be aware of the need for some platform detection for WebGL builds, as you may wish to alter your game slightly when deploying online because the hardware out there is very fragmented, as opposed to a full standalone desktop build, scale down the shadows, or disable them, change LOD and general details, especially if you plan to bring your games to mobile platforms such as iOS or Android where the graphic hardware could be even weaker than middle-end PCs.

The personal, free-to-use* version of Unity 2017 offers you the opportunity to build your game for the following platforms:

- PC Windows, Linux, and Mac standalone
- WebGL (cross-platform) for any WebGL-compliant browser
- Facebook platform, both Gameport (Windows) and WebGL versions
- Android and iOS mobile platforms
- Xbox One console, Samsung TV, Apple tvOS, and Windows Store



* means it's free to use unless you or your company overtake the \$100k cap per year of gross income from your games. When you overtake this limit, you or your company must buy a Plus or Pro license.

The Plus, Pro, and Enterprise versions of Unity can be purchased to build to all of the preceding, but also include paid and custom support, cloud computing, multi-platform builds, and team-cooperation tools; all that is needed for a middle or big sized company to manage a large number of projects for many different platforms. The add-ons of developing games for consoles will also require the use of hardware development kits from manufacturers and/or special licensing.

In this chapter, we'll look at how to customize assets to create a WebGL and a standalone desktop; we'll also take an overview of mobile builds, covering the following topics:

- Working with **Build Settings** to include scenes and choose a platform
- Set up **Player Settings** carefully to prepare for building web player, mobile, and standalone versions of your game
- Platform detection to add/remove elements from specific builds
- Where to share your games with other developers or with gamers around the world
- Where to get further help with Unity development

Unity offers you an array of differing build platforms to choose from, and when considering your work as a Unity developer, you should rest assured that as a product, Unity will continue adding new build options as different hardware and software platforms emerge. The current build options are contained in Unity's **Build Settings**, accessible from **main menu | File | Build Settings**.

Supported platforms

Let's start with a brief overview of the main platforms Unity supports. There are many other platforms available and next ones to come, especially after the latest explosions of the **virtual reality (VR)**, **augmented reality (AR)**, and **mixed reality (MR)** platforms, but they are definitively the more common ones.

PC (Windows), Linux, or Mac standalone

Standalone or desktop builds are fully-fledged applications that are delivered as a self-contained executable. Building your game for macOS X standalone will build a single (.app) (application) file with all the required assets bundled inside, while building for Windows PC standalone will create an .exe (executable) file and a dynamic linked library (.dll) file and the associated assets required to run the game in a folder with the same name of the executable plus `_data` appended. Building a standalone is the best way to ensure maximum performance from your game as the files are stored locally and your user is unlikely to have other applications open while running the game, which may be the case with a WebGL deployment.

Android platform

The Android platform covers thousand of device types built from a wide range of companies. While Samsung and HTC have proven to be able to build cutting-edge devices, not second to Apple's iPhone, there are many other producers and manufactures who install Android on their device for many reasons. This is one of the reasons that have brought billions of customers and, of course, thousands of developers as well to Android in the past five years. Whenever you want to deploy an Android application on a device, there are a few things you need to consider:

- On an Android device, you will run OpenGL ES 2 or 3, which are compact and reduced GL meant to run on micro GPUs. This means a lot things, one of which is that, for example, real-time soft shadows will not run smoothly on most devices for performance reasons.
- Preparing a texture atlas (or using the Sprite Packer in the case of 2D games) instead of many single texture files is a must. Where as you can almost ignore this on modern graphics cards mounted on PCs, the best practice on mobile is to always batch all the textures used in the game in a big (2048) texture atlas. With some exceptions for terrain and nature textures, all static objects of the level and all UI sprites should be packed into atlases.

- Texture compression should be chosen carefully. When you select the Android platform in **Build Settings**, you can choose the type of texture compression. ETC (Ericsson Texture Compression) compression is the default choice, and is good for almost any recent hardware out there, based on GL ES 2. If your game should run only on newer devices with GL ES 3, you can freely choose ETC2.
- Keep input controls in mind; you should plan the user interface in a way that will make the experience fluid and friendly. Always consider a custom interface and custom input controls for the mobile release; otherwise, it may be just a waste of your time to release a great game with poor user controls.

iOS

On the **iOS** platform, many of the optimizations you make for the Android mobile platform will be the same. Additionally, there are some extra bits regarding the iOS environment and its new native `Metal` graphic library.

Also, the graphic hardware on iOS devices is different, generally more advanced than the most of the Android models out there, and mount a PowerVR GPU which has really horse power to spare and with the latest `Metal` libraries become more powerful than ever. This is a reason for changing the textures import format. Luckily, when you switch platforms, Unity does this for you automatically.

WebGL

Not everybody has a tablet or a mobile phone powerful enough to play modern videogames; billions have a computer and an internet connection. The web browser is still the most widely used platform all over the world. This makes the web the platform where you will reach your wider audience. Whether you decide to develop for distributing a pay-per-play web portal such as Kongregate or on your dedicated web portal, or distribute it as free-to-play on Facebook, these are cases where we will use the WebGL build. Since Unity took the step toward WebGL versus the binary file read by the WebPlayer plugin, the end user should not worry about having something installed on their PCs. Instead, any HTML5 WebGL-compliant web browser will be able to play the content.

The WebGL build can then be taken and embedded into a web page of your own design. As WebGL builds rely on the web browser to load the HTML page that calls the final export, any computer running your game as a WebGL web page is already using up processing power on the browser, and some of this cost is caused by the dimensions of the web player embedded on the page. With this in mind, it can help to provide your game at a lower resolution than you would for a desktop build. We designed our game for an entry-level desktop resolution of 1024 x 768 pixels. However, when deploying into a web build, the screen size should be reduced to something smaller, such as 960 x 600 pixels. This makes the load on the GPU less intensive as lower-resolution images are being drawn at each frame, thus giving better performance. We will look at how to build your game at this resolution and how to alter settings for the player later in this chapter. This choice also depends on what kind of game you are making and which channel you choose to distribute it. For a puzzle game a fixed, low resolution is always recommended, while third-person or first-person games may start at that, but have the ability to choose a different video resolution in the game.

Virtual, augmented, and mixed reality

In the past four years, Oculus Rift and the mobile companion from Samsung, the Gear VR, and the Vive HTC from Valve and later Microsoft Hololens, as well as other players in this always growing market, expanded the commercial and home usage of virtual reality, augmented reality, and mixed reality devices for working, entertainment, leisure, and education.

Unity followed strictly the evolution of all these platforms, and today offers a perfect solution for developing cutting-edge applications for this amazing platform.

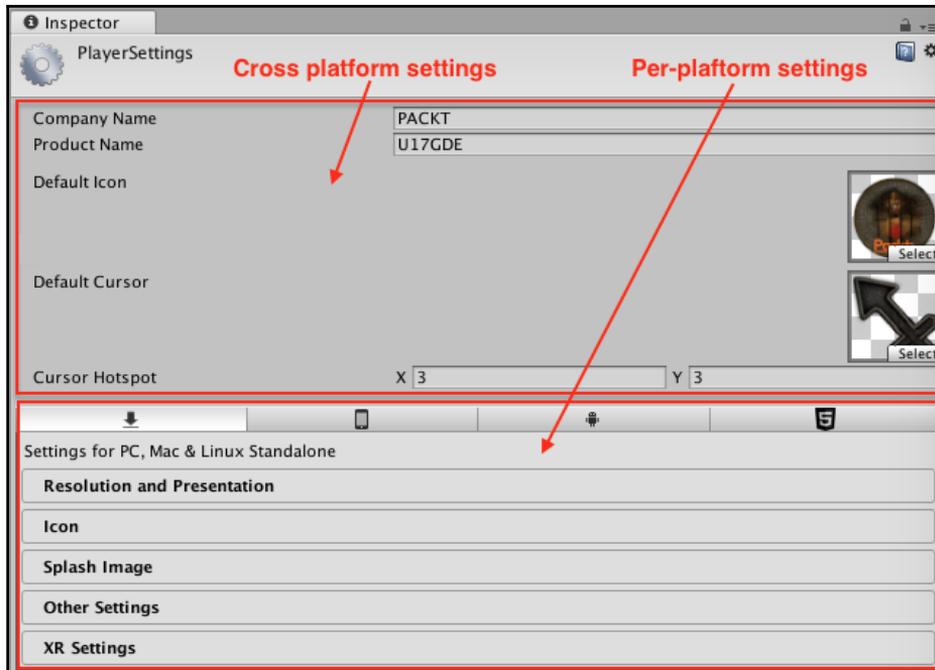
Let's look at customizing our build further by adjusting the **Player Settings...** to suit our *Devil Island* game for Mac/PC/Linux.

Player settings

In software development terms, an exported version of your project is known as a **build**. In Unity, when exporting a build, you are effectively placing your content into what is known as the Unity player. In a standalone build, the player is part of the packaged executable PC, Mac, or Linux game. In **Player Settings**, you can specify important options, such as resolution, icons, graphics API, and rendering settings, for the player to use. To adjust settings such as these, we'll need to look at the **Player Settings**; from the top menu, go to **Edit | Project Settings | Player**.

PlayerSettings are split into two main sections: cross-platform settings and per-platform settings. Cross-platform settings will be valid for all your build targets and are situated at the top of the **PlayerSettings**.

Other settings in the per-platform settings, which will be shared among the various platforms, will be marked with an asterisk (*):



The cross platform settings head section and the following per-platform settings section

Cross-platform general settings

The first part of **PlayerSettings** is cross-platform, and they are required for all kinds of builds. They will simply ask you to provide a **Product Name** for the project as well as a **Company Name** and a **Default Icon** image. This icon can be overridden by the *per-platform settings* where required, but it is generally acceptable to provide a higher-resolution image for this setting (such as the 128x128 screenshot provided for this book), and allow Unity to scale where appropriate. The other option is to provide an icon you have designed yourself for each icon resolution within the settings for individual platforms; this approach ensures total control for you, but it is best to try the former approach in case it *just works* for you.

Now, take a second to fill in a title and company name, and then locate the **icon_large** texture inside the **Book Assets | Textures** folder. Select this, set **Texture Type** to GUI in the **Texture Importer** component of the Inspector, and click on **Apply**.

Now, return to the **Player Settings (Edit | Project Settings | Player)** and drag it onto the **Default Icon** setting; if you need a custom cursor, you may specify one by dragging a custom image previously imported as a cursor in the import settings. If the cursor is not an arrow shape pointing north-west, for example, a cross hair, you can adjust the mouse position origin by specifying the **X** and **Y** values (in pixels) where the **Cursor Hotspot** will be the offset. In our case, we will specify the 3 and 3 pixels offset, to match the exact point for the arrow pick, which is not exactly touching the image borders.

Let's take a look at all the important settings sections that are per-platform.

Per-platform player settings

These settings for your chosen platform are divided into four categories:

- Resolution and presentation
- Icon
- Splash image
- Other settings
- XR Settings (this section is not available for the WebGL build)



Some settings may vary from platform to platform; when a setting is shared among platforms, it's marked with an asterisk: *.

Mac - PC - Linux - standalone build

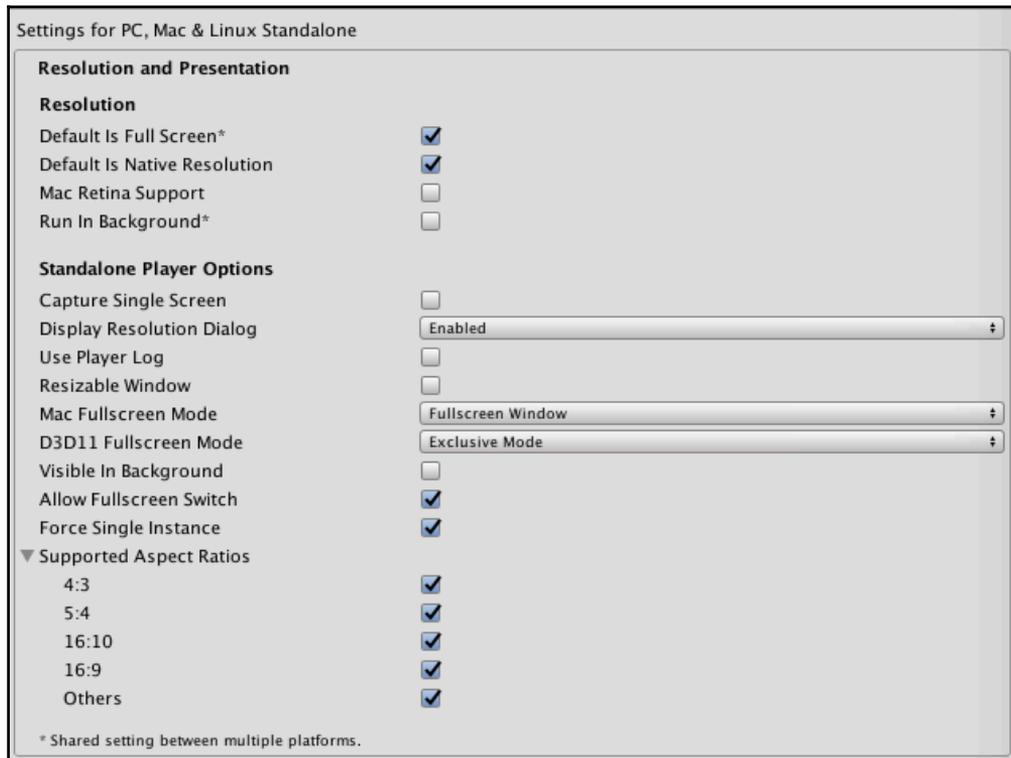
First, we will see the **Player Settings** for the standalone build that includes the Microsoft Windows PC platform, the Apple MacOS platform, and the Linux platform. Here's an illustration of the **Resolution and Presentation** section of the **Settings for the PC, Mac & Linux Standalone** platforms.

Resolution and Presentation

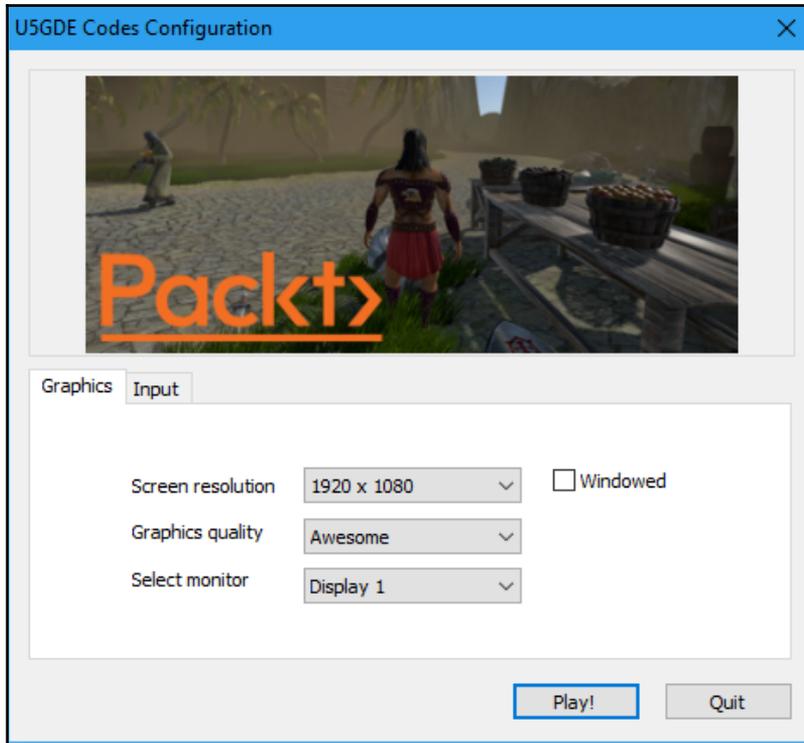
The settings in the **Resolution and Presentation** section of the menu allow you to customize what the player will see when they experience your game, and are especially crucial for setting up the resolution that the game is exported at.

The configuration options in this section vary a lot with the corresponding platform.

This section contains a **Resolution** section and a **Standalone Player Options** section; both of them change when changing the platform:



When loading a standalone copy of your game, the player can be presented with a **Display Resolution Dialog**, a startup window showing options for resolution, control input, and quality settings at which to play your game. However, you have the choice of disabling this option from being shown; the **Display Resolution Dialog (Enabled)** in the **Resolution and Presentation** settings (seen in the preceding screenshot) toggles this option:

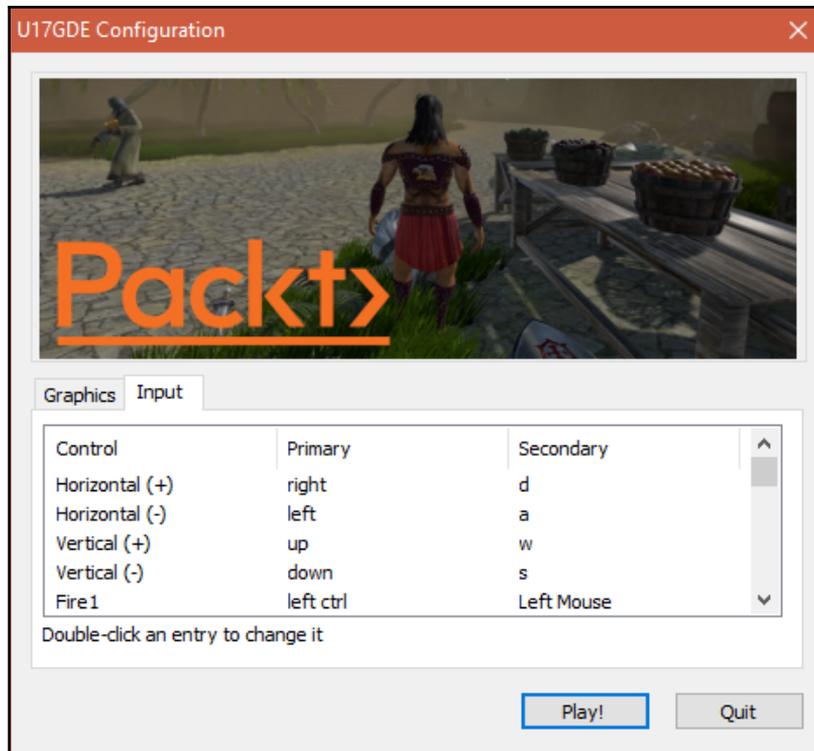


The Resolution Dialog (PC)

User input bindings

The **Input** tab shows the default controls for your game. The controls are mapped on keyboard, mouse, and gamepads/joysticks. Like we have seen in the previous chapter, you can configure these inputs to fit your game needs in the Editor Input settings.

When you do so, it's then easy for the end user to configure alternative controls and bind them directly in the **Input** tab before playing the game:



To read more about standalone player settings, head to <https://docs.unity3d.com/Manual/class-PlayerSettingsStandalone.html>.

Icon

This setting is not applicable for WebGL deployment, as the only icon in web terms is the one seen as the HTML page favicon (the icon that you will see in your browser favorites bookmarks), which is defined by the website on which your game content is embedded. The **Icon** settings shown in the next screenshot for the standalone build have been taken from our default icon and are scaled appropriately.

This explains the benefit of creating a default icon larger than 48x48, the one provided as part of this book's assets is designed at 128x128, but it's best to design it at least 1024x1024 and then let Unity scale it down for you at various sizes:



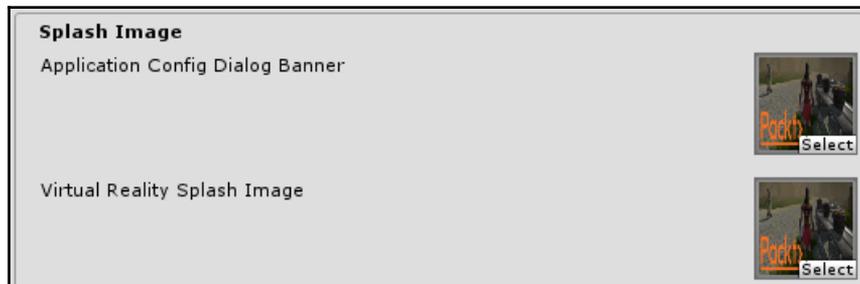
The initial part of the Icon settings section

As you can see from these settings, you are able to provide different textures for various scales of texture, allowing you to create a simpler design for smaller scales if you wish. On the mobile platform especially, you might need to override these settings because each platform has its own icon guidelines.

Splash image

If you enabled the **Display Resolution Dialog** in the **Presentation and Resolution** settings, here you will specify the image to display in the dialog. For standalone deployed games, the **Splash Image** setting gives you a slot to assign a **Config Dialog banner**. This banner is shown in the preceding screenshot as part of the **Resolution Dialog** window that loads when the player first launches your game, if this has not been disabled under **Resolution and Presentation**, as discussed previously.

A splash image for Survival Island has been designed for you. Simply select the texture called `splash_image` in the **Book Assets | Textures** folder in the **Project** panel, and set its **Texture Type** to **GUI** in the **Texture Importer** component of the **Inspector**; click on **Apply** to confirm. Return to **Player Settings (Edit | Project Settings | Player)** and drag the `splash_image` texture onto the empty **Config Dialog Banner** slot, where it currently shows **None (Texture 2D)**. This is not applicable to the WebGL build:



The Splash Images for the standalone dialog and the Virtual Reality Splash Image



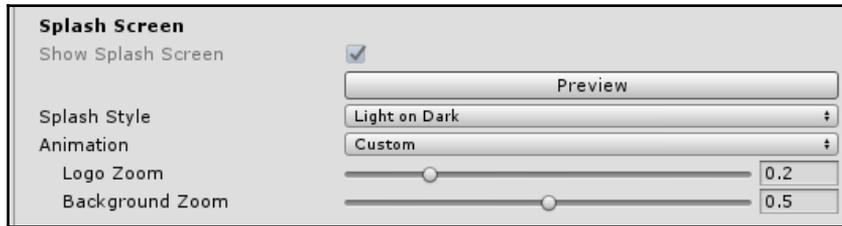
If you wish to design your own **Application Config Dialog Banner** texture for your games, you should create a file in your preferred art package with a resolution of 432x163 pixels. This will adapt at the dialog space reserved for the Splash Image. Import the file as GUI legacy non compressed.

Splash screen

The rest of the section allows you to set up an animated introduction screen to be seen after dismissing (if enabled) the resolution dialog as the first thing you will see before your starting scene (scene 0). Unity Personal Edition users will not be allowed to disable the **Show Splash Screen** option, as a Unity logo will always be on screen for them.

The **Splash Screen** for your game will be designed as part of the first scene you load into the game, perhaps making use of an animation to introduce the developer's logo, as you've likely seen in commercial games.

Let's see how to configure it:



The **Splash Style** setting specifies the style of your splash screen. With **Light on Dark**, you will be using a white Unity logo over a dark background; this can be set in two ways: **Light on Dark** or **Dark on Light**.

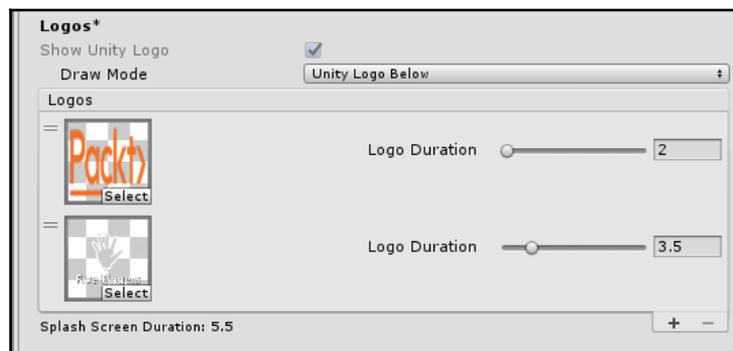
The **Animation** setting can be set as **Static**, **Dolly**, or **Custom**:

- **Static**: The one or more logos you specified in the list (see the next paragraph) will just fade in and then out, one after another, in the order shown in the **Logos** section
- **Dolly**: The logos will alternate with a similar zoom, and the background will almost stand still (zoom slightly in)
- **Custom**: You specify the logo(s) zoom transition time and the background zoom transition time individually

You can preview the animation anytime by pressing the **Preview** button.

Logos

In the **Logos*** section, you can specify a list of images that will alternate one after another with the parameters chosen in the **Splash Screen** section:

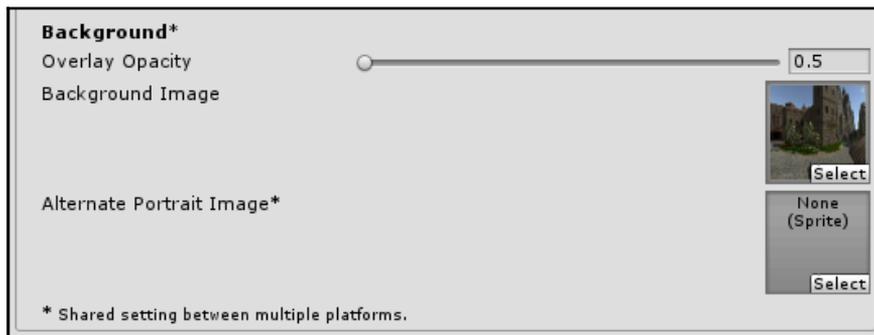




Personal versus Plus versus Pro and Enterprise Unity editions: when building your game with the personal version of Unity, you should be aware that a "Made with Unity" splash screen will be shown before your game loads. Unity Plus, Unity Professional, and Unity Enterprise users will be able to set up their own graphics for the animation, without the Unity logo, and, if they want, disable the intro completely by deselecting the **Show Unity Logo** checkbox.

Background

The background section allows you to set an image to be faded with the rest of the logo animation in a semi-transparent overlay shape. The **Overlay Opacity** setting can tweak the hardness of the effect. Also, this option is a cross-platform setting that is shared between multiple platforms, and also goes for the **Alternate Portrait Image** that can be specified in case the device is a phone or tablet that can run the game on a portrait landscape screen orientation:



The Background section of the Splash Image settings

Other settings

The other settings focus on **Rendering** specifics, scripting **Configuration**, and various other **Optimization(s)** and **Publishing** options.

These settings should be left at their default for the WebGL build, but it is worth being aware that the **Rendering Path** can be switched to **Forward** or **Vertex Lit** if you are working on a project that is designed with older hardware in mind. Ideally, deferred lighting should always be used to get the best visual performance, although it does not support anti aliasing (softening harsh edges in your game); so, it is best to try out these two options to balance performance and quality.

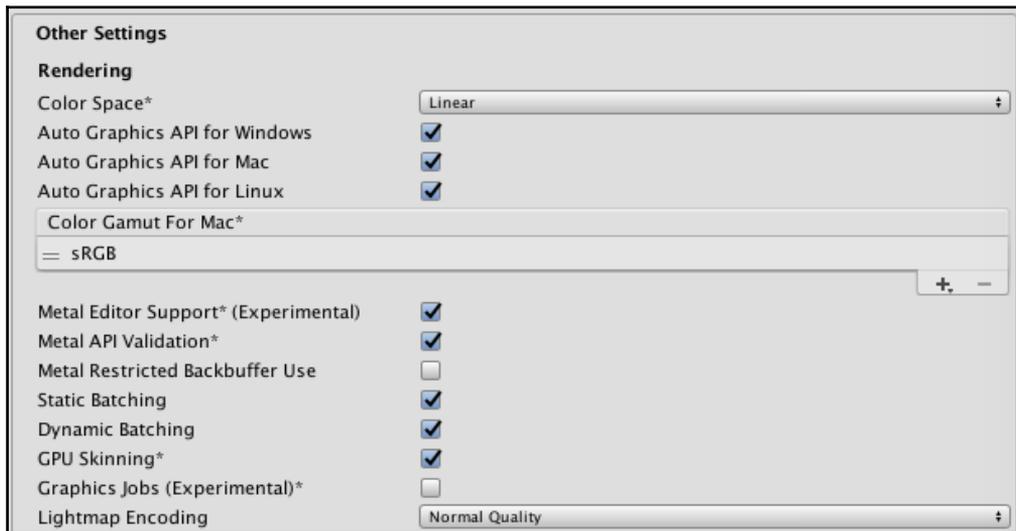
Let's take a look at the four main parts that fill the **Other Settings** section.

Rendering

Color Space* should always be **Gamma** for mobile and WebGL builds, but can be switched to **Linear** for a better color space on high-end computers and console platforms. Beware that the **Color Space*** option is a cross-platform option (asterisk), so it is shared between platforms.

This means that you cannot select a different color space for each platform at runtime, but you can always switch before building the application. A practical example would a PC/mobile game. You should put **Linear** Color Space to support modern videocards, but a **Gamma** Color Space when you export the mobile-targeted version of your game.

The next screenshot shows the **PlayerSettings** for the exported build where you can set **Color Space*** and other important bits:

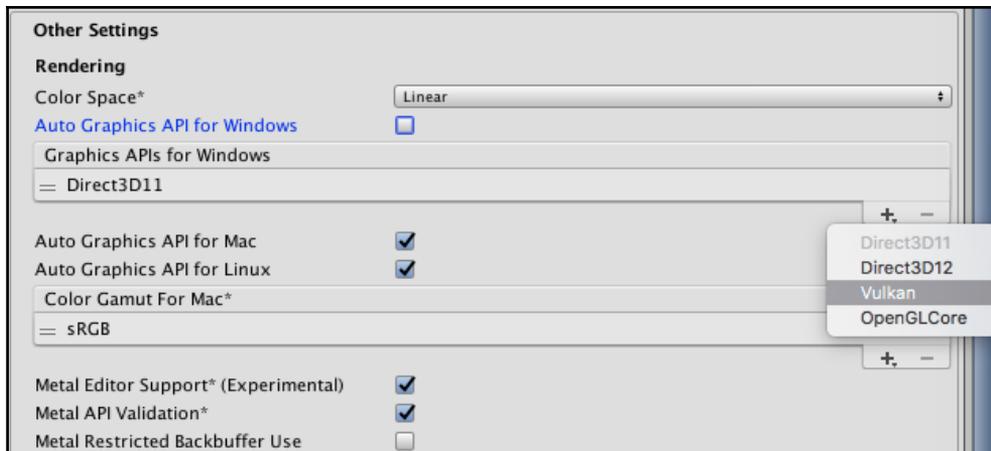


This can be seen as non-important at first glance, but the **Color Space** setting will change the look of your scenes a lot. While **Gamma** is intended for older hardware and is suggested for mobile/slower platforms, the **Linear** color space is far better-looking and more high-end in terms of GPU effort. This is a compile-time setting and cannot change during execution.

You should decide carefully whether your game should be rendered with **Linear** or **Gamma**.

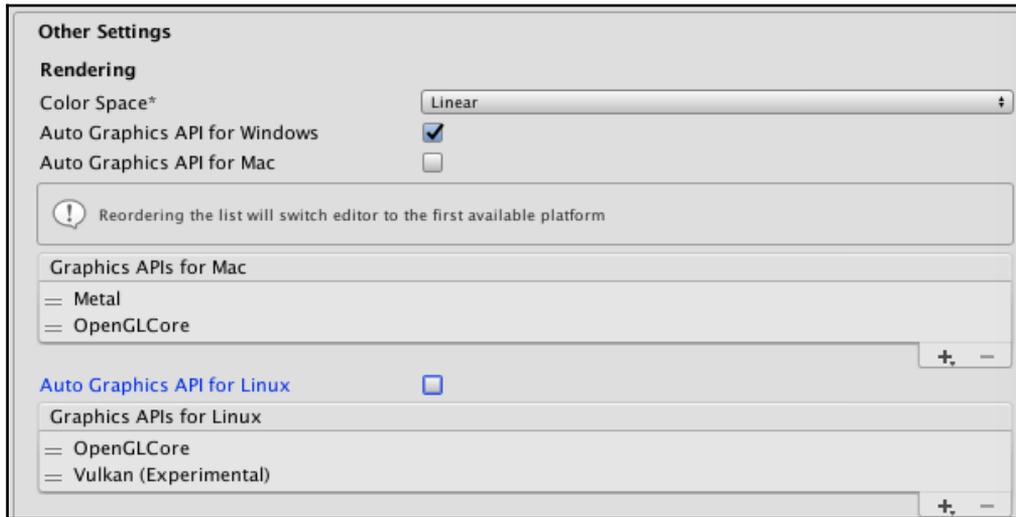
Color Space* isn't only property that's important and platform-relevant; we will now explore other options meant for the final build, such as the **Auto Graphics API** feature. This option allows us, platform by platform, to include or exclude certain versions of the core graphic library used for your application.

When you deselect the **Auto Graphics API for Windows** checkbox, new options will appear to give you the ability to include support to only one or more of the listed graphic libraries. For example, on Windows you could select **Direct3D12** only if needed, or **Direct3D11** only, because it might happen that some of your shader code doesn't support **Direct3D12**, or for any other technical reasons:



In the preceding screenshot, the available choices on the Windows platform seen in the **Other Settings - Rendering** section are **Direct3D11**, **Direct3D12**, Vulkan, OpenGLCore. On the Mac, can choose from **OpenGLCore** or **Metal** graphic library, while for Linux, you can choose among **OpenGLCore** and the experimental **Vulkan** support. When you are working on a given platform, you may consider also switching the rendering for the editor renderer itself.

You can see from the information box in the next screenshot that, for example, switching from **Metal** to **OpenGLCore** by reordering the supported libraries on the Mac will switch editor rendering too:



We have seen how Static and Dynamic batching works in the previous chapter; here is where you set these automated optimizations on and off. GPU Skinning, instead, is a feature that moves calculation efforts for the skinned meshes to the GPU instead of the CPU.

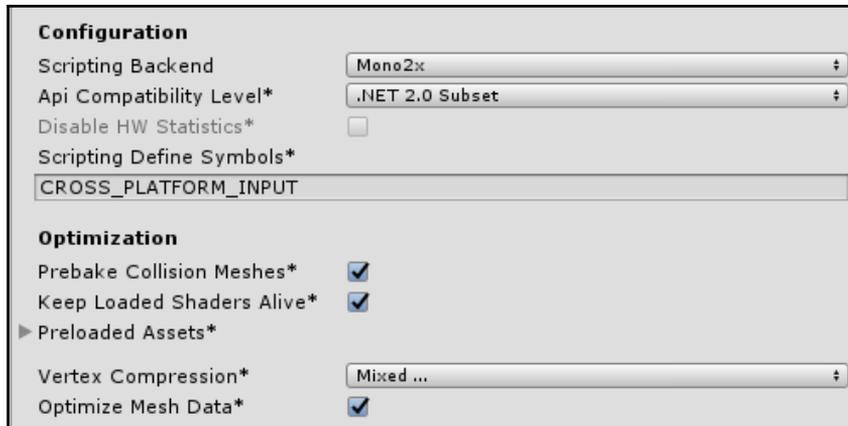


Batching: Unity utilizes dynamic batching—a technique that batches objects together (grouping the render) with a shared material and low mesh detail into single draw calls. This means far better performance when creating multiples of the same object; for example, multiples of props such as buildings. Dynamic batching is done automatically, provided that the object's mesh does not exceed a vertex count of 300. The addition of static batching is a feature that does a similar thing for non-moving objects that you have marked as static.

Configuration and optimization

The **Configuration** section defines the **Scripting Backend** as well as eventual **Scripting Define Symbols*** you might need to compile your source.

The **Optimization** section, instead, will operate optional automated optimization on meshes and collision data:



Logging

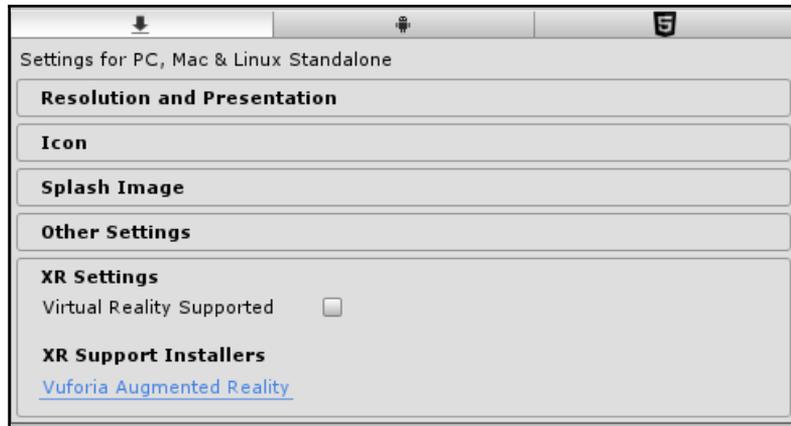
Note that most projects will be fine using the default settings for **Rendering** and **Optimization** when building a standalone game.

Let's see now where Player Settings differ the most when building for other platforms.

XR settings

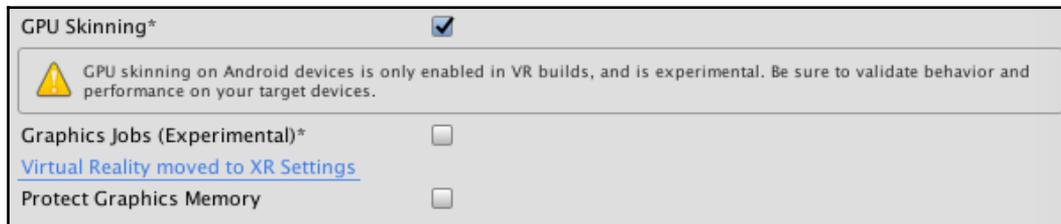
The XR settings section is present for standalone and mobile platforms, and it includes a checkbox to enable virtual reality support (hence the stereo rendering of the two eyes on potentially independent cameras) and open to the support of the VR platforms out there: Oculus, Vive, GearVR, iPhone VR.

Additionally, a list of augmented reality and mixed reality framework installers supported by Unity follows:



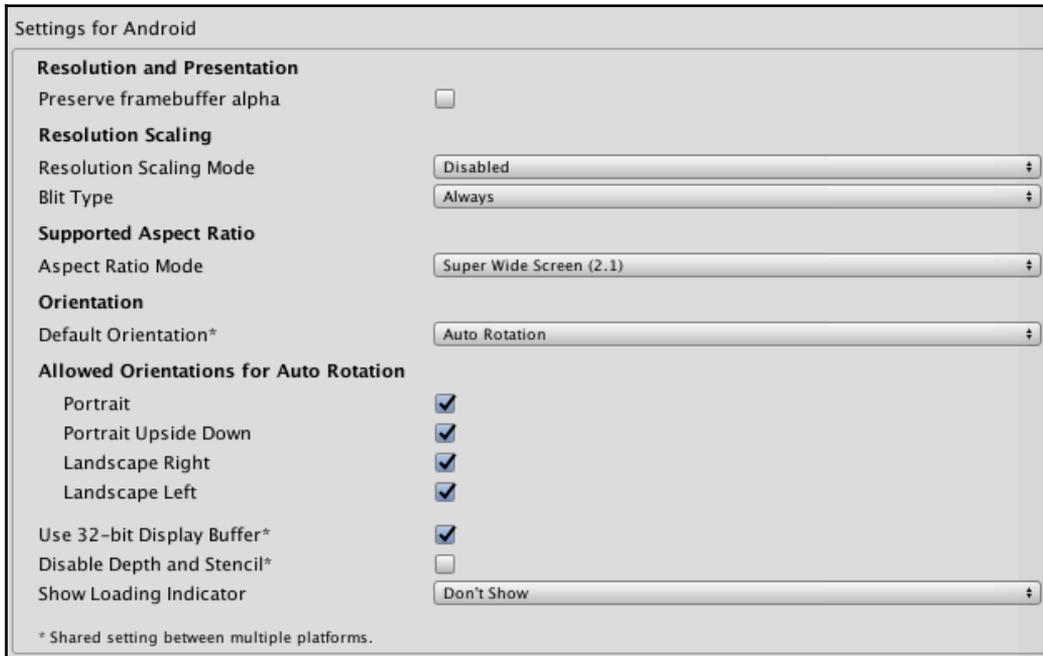
Android

While GPU skinning is supported on most platforms, Android is an example where this support is still experimental:

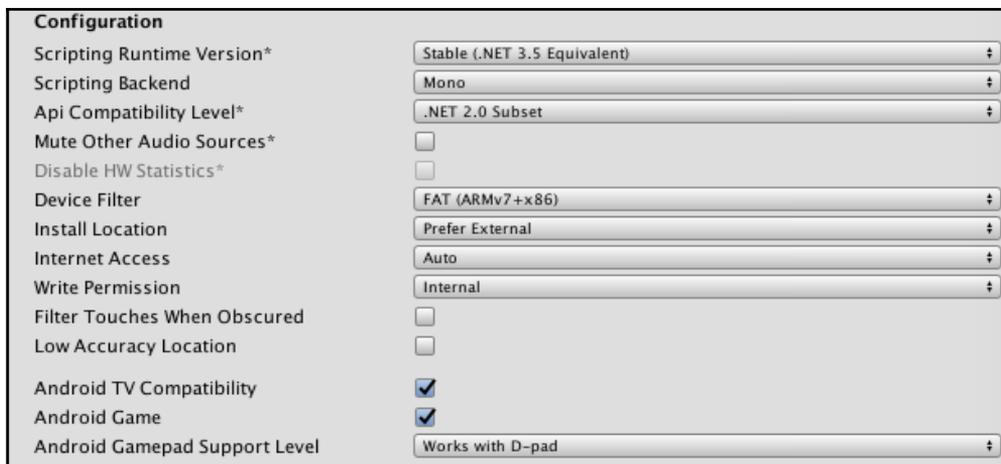


On Android, things are quite different. It is a mobile platform and we don't have any initial dialog to show.

In the next screenshot, you can see Android's **Resolution and Presentation** settings:



On the **Configuration** section, you can set important bits such as the device filter platform, the install location, write permissions, and other important Android settings, TV and game related:

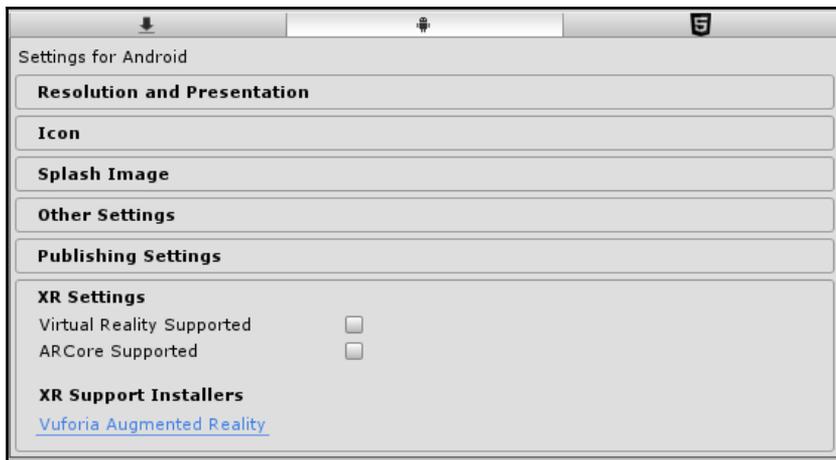


In the identification section, you set important bits of the build such as the minimum and target API level, the version number, and the package domain name of your application:

Identification	
Package Name	<input type="text" value="com.Company.ProductName"/>
Version*	<input type="text" value="1.0"/>
Bundle Version Code	<input type="text" value="1"/>
Minimum API Level	<input type="text" value="Android 4.1 'Jelly Bean' (API level 16)"/>
Target API Level	<input type="text" value="Automatic (highest installed)"/>

Finally, in the **Publishing Settings** section, you can set important bits of the Android build that are related to publishing on Google Play Store, as we will see later in this chapter.

Note that the XR settings are available on Android too, to support platforms such as Samsung's Gear VR for example:



For more information about the Android player settings, head to <https://docs.unity3d.com/Manual/class-PlayerSettingsAndroid.html>.

iOS

If you are using a MacOS, you can build for iOS directly on the device. Unity will export an XCode project for you and will try to auto-launch it to build the app on the device when you choose **Build && Run**.

To set up the iOS build, you should take care of the Metal graphic API settings:

Force hard shadows on Metal*	<input type="checkbox"/>
Metal Editor Support* (Experimental)	<input checked="" type="checkbox"/>
Metal API Validation*	<input checked="" type="checkbox"/>
Metal Restricted Backbuffer Use	<input type="checkbox"/>

See the official manual page for detailed iOS player settings at <https://docs.unity3d.com/Manual/class-PlayerSettingsiOS.html>.

WebGL

Most differences from the standalone settings lie in the **Publishing Settings** section, where you can set important bits of the WebGL build:

Publishing Settings	
WebGL Memory Size	256
Enable Exceptions	Explicitly Thrown Exceptions Only
Compression Format	Gzip
Name Files As Hashes	<input checked="" type="checkbox"/>
Data caching	<input type="checkbox"/>
Debug Symbols	<input type="checkbox"/>
WebAssembly (Experimental)	<input type="checkbox"/>

See the official manual page for detailed iOS player settings at <https://docs.unity3d.com/Manual/class-PlayerSettingsWebGL.html>.

Player input settings

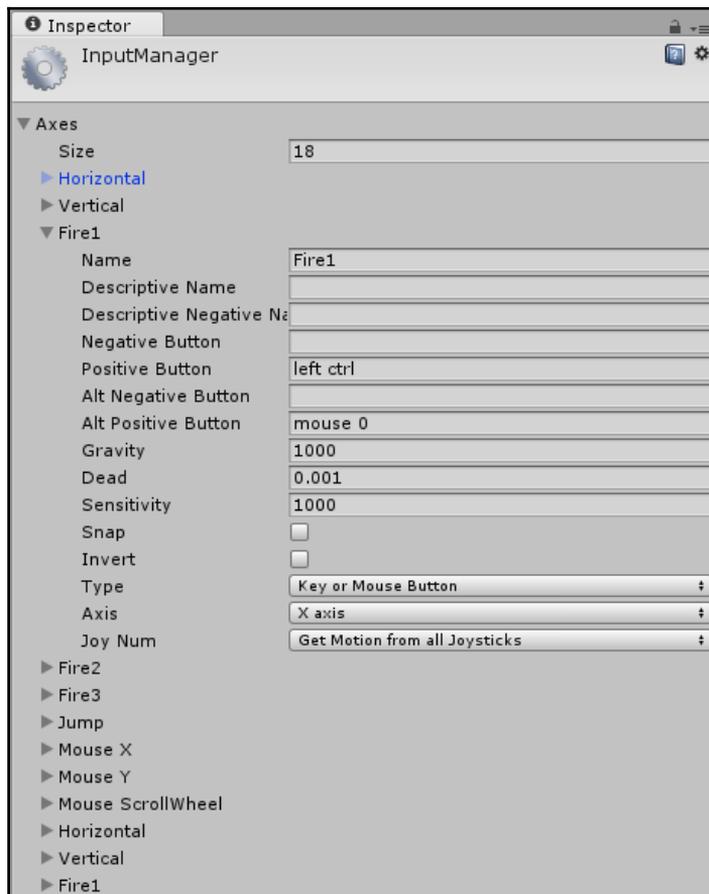
While the **Resolution Dialog** window gives the standalone build player the ability to adjust the input controls of your game in the **Input** tab (see the following screenshot), it is important to know that you can specify your own defaults for the control of your game in the **Player Input** settings.

This is especially useful for web builds, as the player has no ability to change control settings when they load the game. Therefore, it is best that you set them up sensibly and provide information to the player through your in-game GUI.

In Unity, navigate to **Edit | Project Settings | Input** to open **InputManager** in the **Inspector** part of the interface. You will then be presented with the existing axes of control in Unity. The **Size** value simply states the number of existing controls. By increasing this value, you can build in your own controls; alternatively, you can simply expand any of the existing ones by clicking on the gray arrow to the left of their name and adjusting the values therein. Now, click on the arrow to the left of the **Fire1** entry to expand it. Looking at this setting, you can see how this ties together with the code we wrote earlier; when looking at the `StoneThrower` class, we wrote the following code:

```
if (Input.GetButtonUp("Fire1"))
```

Here, the **Fire1** axis is referenced by its name. By changing the **Name** parameter in the input settings, you can define what needs to be written in scripting:

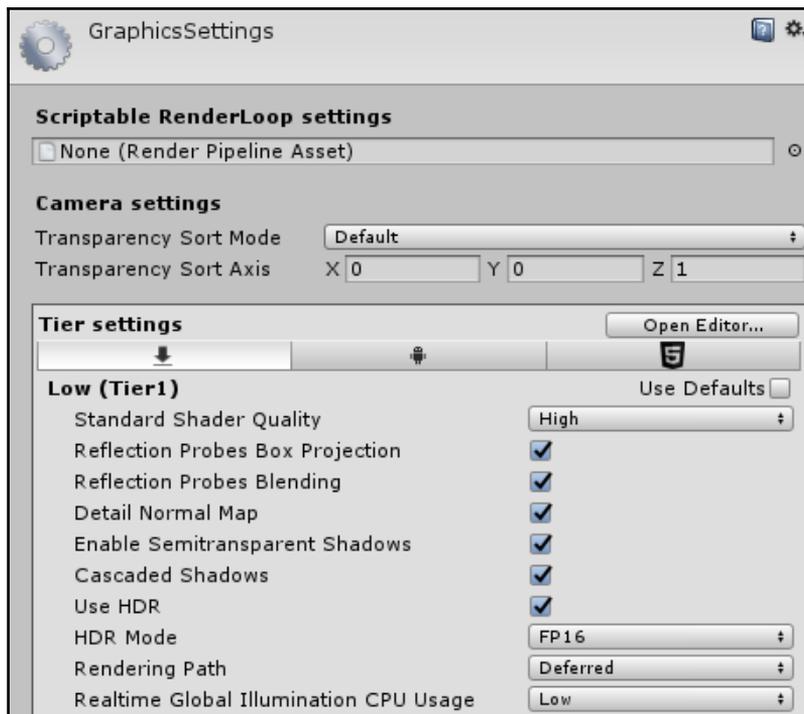


For more information on the keys you can bind to in these settings, refer to the **Input** page of the Unity manual at <http://unity3d.com/support/documentation/Manual/Input.html>.

Graphics settings

GraphicsSettings is a new window for managing different tiers of graphics caps for different families/generations of devices.

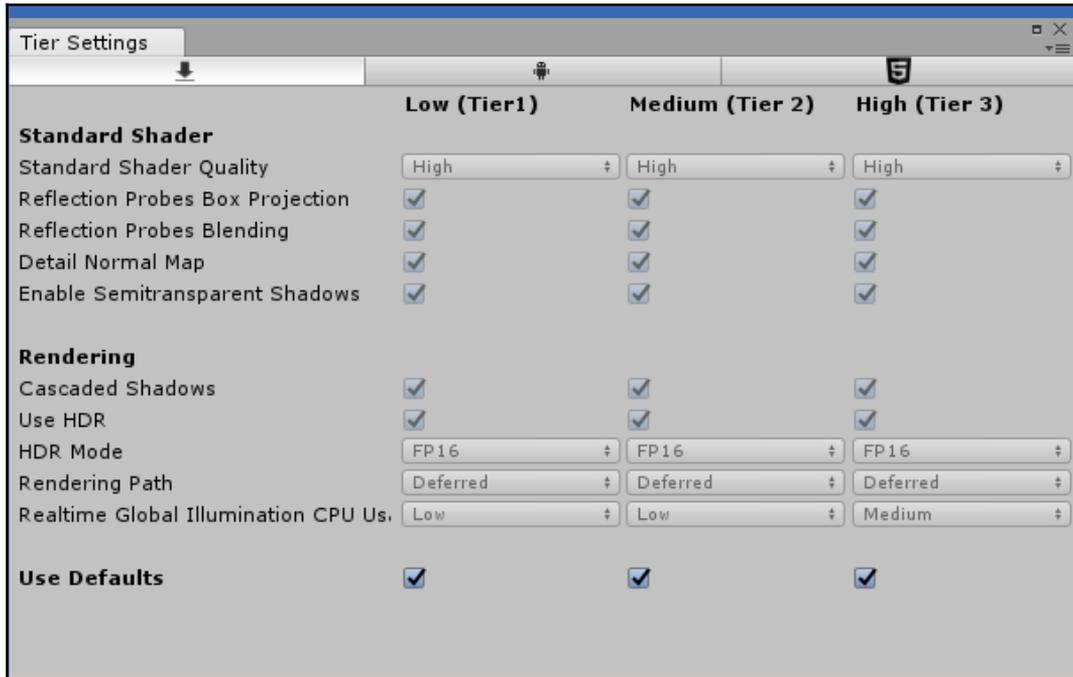
In this way, developers have the ability to customize even the Render Pipeline according to different platforms and tiers, by specifying a **Scriptable RenderLoop settings** profile. Here is where you define per-tier engine settings in detail:



This approach, new in Unity 2017, is especially useful when planning to use the cloud building system, but in particular for those platforms that have very fragmented hardware capabilities among the various generations that year after year come out and replace the previous ones. To read more about *Scriptable pipeline*, head to <https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html>.

Tier settings

By clicking on the **Open Editor...** button in the **GraphicSettings** panel, we can open the **Tier Settings** editor. This is where you can define, platform by platform, three different tiers for the quality settings. The low tier is intended for older devices, while the medium and the high tiers are for stronger PC gaming machines and recent consoles:



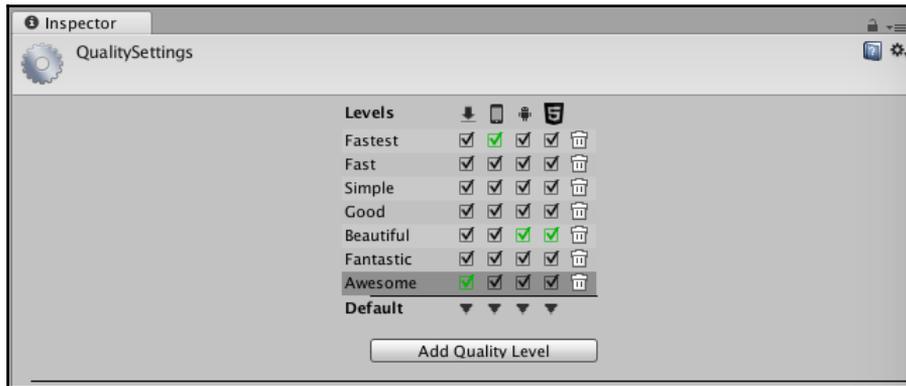
Read more about **GraphicSettings** and **Tier settings** at <https://docs.unity3d.com/Manual/class-GraphicsSettings.html>.

Quality settings

When exporting from Unity, you are not restricted to any single level of quality. You have a lot of control over the quality of your output, which comes in the form of **Quality Settings**. Open this now in the **Inspector** part of the interface by selecting **Edit | Project Settings | Quality** from the top menu.

Here, you'll find the ability to set your three different builds to one of the six different quality presets, **Fastest**, **Fast**, **Simple**, **Good**, **Beautiful**, and **Fantastic**. You can then edit these presets yourself to achieve precise results, as you need. To understand the range of potential quality that Unity can produce, let's take a look at opposite ends of the scale, comparing **Fastest** with **Fantastic**.

For our project, we have added a new quality level **Awesome**, that tries to boost the settings even higher:



The quality level selected in the list will be the actual quality of the editor itself. Sometimes it's useful to reduce the quality of the editor to allow smooth editing of complex scenes that may result in being too choppy and slow at a high-quality level, depending on the hardware you have at your disposal.

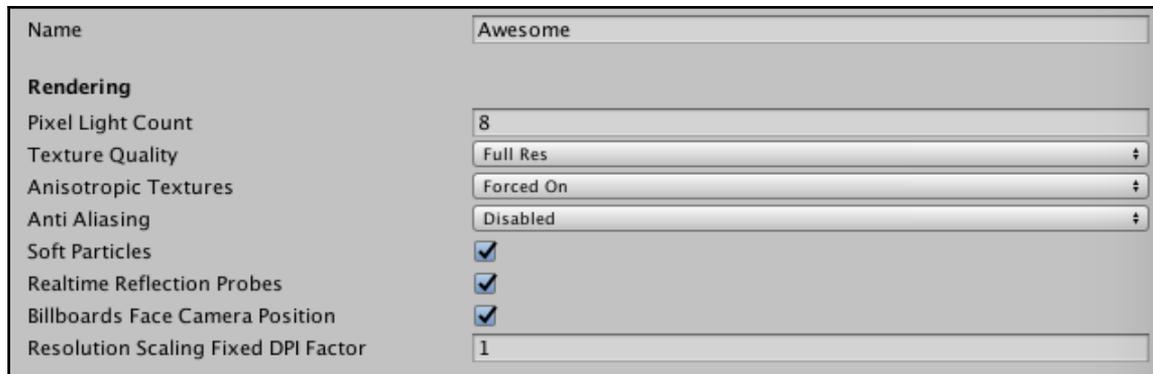
For our project, we will add a new quality level for high-end, top-level gaming computers. Click on the **Add Quality Level** button to do so. Rename the quality level to **Awesome**. On this setting, we will set **Pixel Light Count** as 8, which is the maximum number of real-time lights you can have in a scene, and a very high resolution for the default **Shadow Resolution**, with a **Shadow Distance** of 200 [very far when using a 1:1 world scale (1 Unity unit = 1 meter)] and **Four Cascades** for the **Shadow Cascades** setting.

Of course, we will have **Soft Particles**, **Realtime Reflection Probes**, and **Billboards Face Camera Position** checked for our project.

Let's look in detail at the **QualitySettings** where a certain quality level is described in its three sections, **Rendering**, **Shadows**, **Other**.

Quality settings - Rendering

As you can see, these settings are vastly different at each end of the scale, so let's take a look at what the individual settings do. Click and select the newly created **Awesome** quality level, and make changes to the values, like in the following screenshot:



The Awesome quality level preset with its Rendering settings

Let's take a deeper look at some of the more important options:

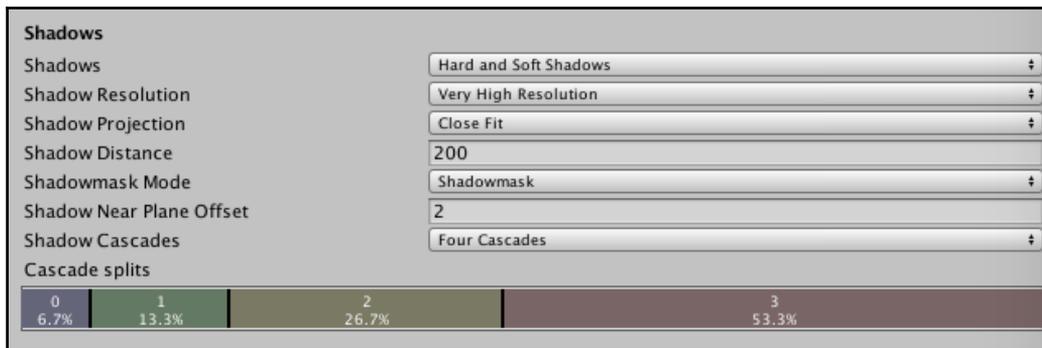
- **Pixel Light Count:** The number of pixel lights that can be used in your scene. Lighting in Unity is calculated either per-vertex or per-pixels; usually per-pixels lighting provides a more accurate result at the cost of more computation. With this setting, you can allow a certain number of real-time pixel lights, with the rest being rendered as vertex lights. This is why the Fastest quality preset has the **Pixel Light Count** set to 0 by default.
- **Texture Quality:** Exactly as it sounds, the amount to which Unity will compress your textures: 1 = no reduction, 4 = 25%.
- **Anisotropic Textures:** Anisotropic textures filtering can help improve the appearance of textures when viewed at a steep angle, such as hills, but is costly in terms of performance. Bear in mind that you can also set up this filtering on an individual-texture basis in the **Import Settings** for assets.
- **Anti Aliasing:** This setting softens the edges of 3D elements, making your game look a lot better. However, as with other filters, it comes at the cost of performance.

- **Soft Particles:** Indicates whether soft blending should be used to render particles.
- **Realtime Reflection Probes:** This option indicates whether reflection probes should be updated during gameplay.

Quality settings - Shadows

In the Shadows section, you can set the quality of the shadows system for the selected quality level.

We will set our **Awesome** quality level **Shadows** setting in the following way:



The Awesome quality level preset Shadows settings

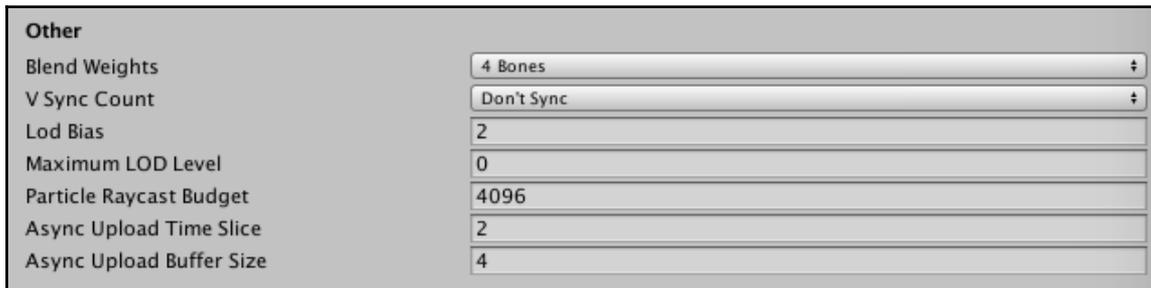
Descriptions of some of the settings are as follows:

- **Shadows:** This feature allows you to specify no dynamic shadows, hard shadows only, or hard and soft shadows. However, as we have seen, shadows can be baked as part of the lightmapping process in the free version of Unity.
- **Shadow Resolution:** This setting allows you to choose a quality setting specifically for the shadows being rendered. This can be useful to save performance when you have multiple objects with dynamic shadows in your scene; setting them to a low resolution can mean the difference between switching them off and keeping shadows entirely during optimization.
- **Shadow Distance:** Similar to the optimization of restricting the camera's far clip plane, this is another level of detail tweak. It can be used to simply set a distance after which shadows are not rendered.

- **Shadow Cascades:** A higher number of shadow cascades can improve the appearance of shadows on directional lights.

Quality settings – Other

In the **Other** section of the quality settings where we fine-tune of the engine in miscellaneous areas:



The Awesome quality level preset Other settings

- **Blend Weights:** This setting is used for rigged characters with a boned skeleton, and controls the number of weights (levels) of animation that can be blended between. Unity Technologies recommend two bones as a good trade-off between performance and appearance.
- **V Sync Count:** This setting, covered in [Chapter 12, Designing Menus with Unity UI](#), sets the vertical sync from none (don't sync) to double and triple buffer.
- **Lod Bias:** LOD levels are chosen based on the on-screen size of an object. When the size is between two LOD levels, the choice can be biased toward the less detailed or more detailed of the two models available. This is set as a fraction from 0 to +infinity. When it is set between 0 and 1, it favors less detail. A setting of more than 1 favors greater detail.
- **Maximum LOD Level:** The index of the starting LOD level, which will be the detail for a camera close up.
- **Particle RayCast Budget:** The maximum number of raycasts to use for approximate particle system collisions (those with **Medium** or **Low** quality). For more information, see the Unity online Official Manual: [Particle System Collision Module](#).

You should use these presets to set options that will benefit the player, as they will have the ability to choose from them in the **Resolution Dialog** window (refer to the **Player Settings** section) when launching your game as a standalone, unless you have disabled this. However, it is fairly safe in most instances to use Unity's own presets as a guide, and simply tweak specific settings when you need to. The settings at the top of the Quality settings that offer defaults are also useful as they will allow you to set the editor itself to a particular quality, giving you a more realistic representation of your game's final look as you work.

For additional in-depth information about Unity Quality Settings, head to <https://docs.unity3d.com/Manual/class-QualitySettings.html>.

Now that our environment is looking polished, let's take a look at how we can improve it further!

The first streamed level setting allows you to specify an index number of a scene that you want to be loaded first through streaming if your game is set up to work that way. If this is anything other than your first level (index number 0), you may specify the index number of the scene, which can be found on the right-hand side of the list of scenes in **Build Settings**.

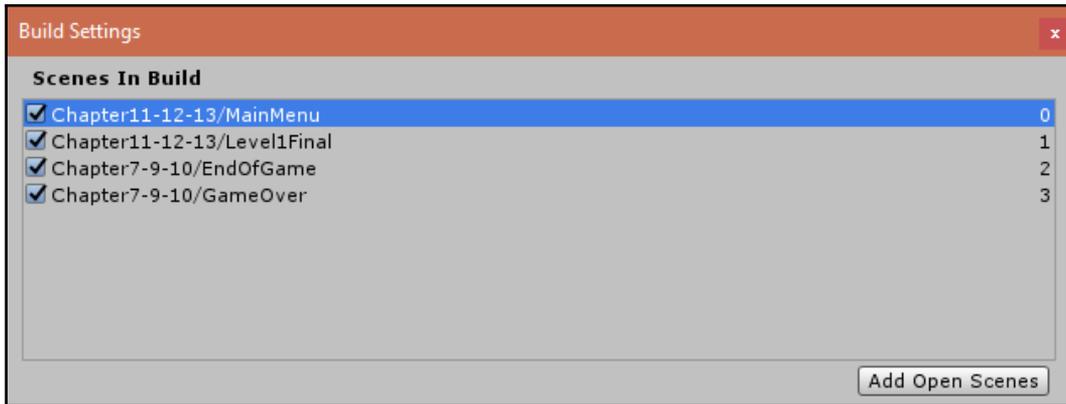
Building the game

Now that we are nearly ready to build the game, you need to consider the varying deployment methods discussed previously, and adapt the project to be built for the web player as well as a standalone game.

Build settings

In Unity, choose **File | Build Settings** from the top menu, and take a look at the options you have. You should see the various options mentioned previously. In the **Build Settings** window, each platform gives additional options in the form of drop-down menus or checkboxes on the right side when selecting a platform on the left side.

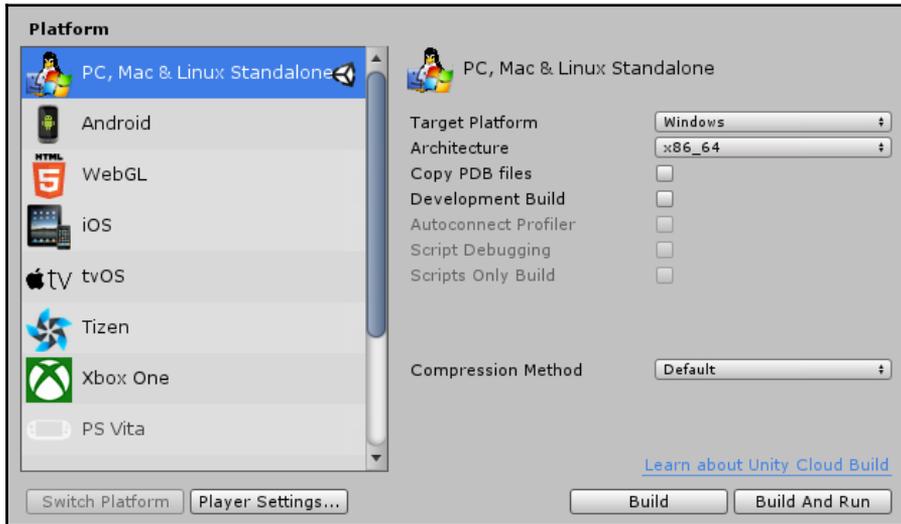
At the top of the **Build Settings** window, you are shown a list of scenes added to our project so far, beginning with the **Menu** scene. It is important to have the first scene you would like your player to see as the first item in the **Scenes In Build** list. If your menu or first scene is not first in the list, then you can simply drag and drop the names of the scenes to reorder them. Also, note that scenes are indexed here, and are given a specific number starting from 0, shown on the right-hand side of the scene name in the list. These index numbers can be useful when loading scenes, but you may also refer to scenes by name (using a string):



The **Add Open Scenes** button will automatically add any open scene in this list

Below the scene list is the **Platform** list, where all the available platforms are listed. Available for Unity doesn't mean available for building; this list includes all the platforms available for Unity, but you should also install their modules in the first place. When you launch Unity's installation setup, you have a list of modules enable for install. Be careful though, in some cases; you may miss an installed SDK (Android SDK or XCode with iOS or MacOS SDKs on Mac computers) you'll need to be able to fully deploy your game on that platform.

While you can export a build of your project by selecting (highlighting in blue) your desired target and choosing **Build**, in order for Unity to be best set up for your intended platform, you should select it and click on the **Switch Platform** button in the bottom-left of **Build Settings**. This effectively sets up Unity to work best for your intended target platform. This is especially important when working with mobile development, as Unity will reinterpret the assets you are working with to suit the chosen mobile platform, automatically choosing compression settings in the **Inspector** for assets already in your project:



The bottom part of the Build Settings window

The other key advantage of selecting your platform is that it causes the **Game** view to show the chosen platform and its available pool of video resolutions as a setting. This allows you to test your game with the correct resolution, before you deploy, which helps show you realistic positions for 2D elements and so forth.

Quit button with platform automation

When deploying as a web build, having a **Quit** button as part of the menu is meaningless because the `Application.Quit()` commands do not function when a Unity game is played through a browser; instead, players simply close the tab or window containing the game or navigate away when they are finished playing. We need to exclude this button from our web menu, but we do not want to delete it from our script because we still want the script to render the **Quit** button in a standalone build.

In the **Platform** area of the **Build Settings**, the Unity logo next to a particular platform indicates that you have selected it as your intended platform to build for. If it is not currently set to PC and Mac standalone, highlight **PC and Mac standalone** in blue and then in the bottom right-hand side of the window, click on **Switch Platform**.



We are switching platforms here for consistency, and to allow you to see the resolution in the **Game** view list of preview dimensions, but it is not necessary to switch platforms for building if you are working on a game and regularly switching platforms. Be aware that Unity switches for you when building, depending on which platform is currently highlighted, anyway.

The Unity logo should now appear on the right-hand side of **PC Linux and Mac standalone**. Now, simply select which **Target Platform** you'd like to build for:

- Windows
- Windows 64-bit
- Mac OS X Universal
- Linux

Click on the **Build** button at the bottom of this dialog window, and you'll be prompted for a location to save your game. Navigate to your desired location and name your built game in the **Save As** field. Then, click on the **Save** button to confirm.

Wait while Unity constructs your game! You'll be shown progress bars showing assets being compressed, followed by a loading bar as each level is added to the build. Once building is complete, your game build will be opened in an operating system window to show you that it is ready. On MacOS, double-click on the application to launch it, on Windows, open the folder containing the game and double-click on the `.exe` file to launch your game. Bear in mind that if you build a PC version on a Mac or vice versa, they cannot be tested (launched) natively.

Adapting for the web

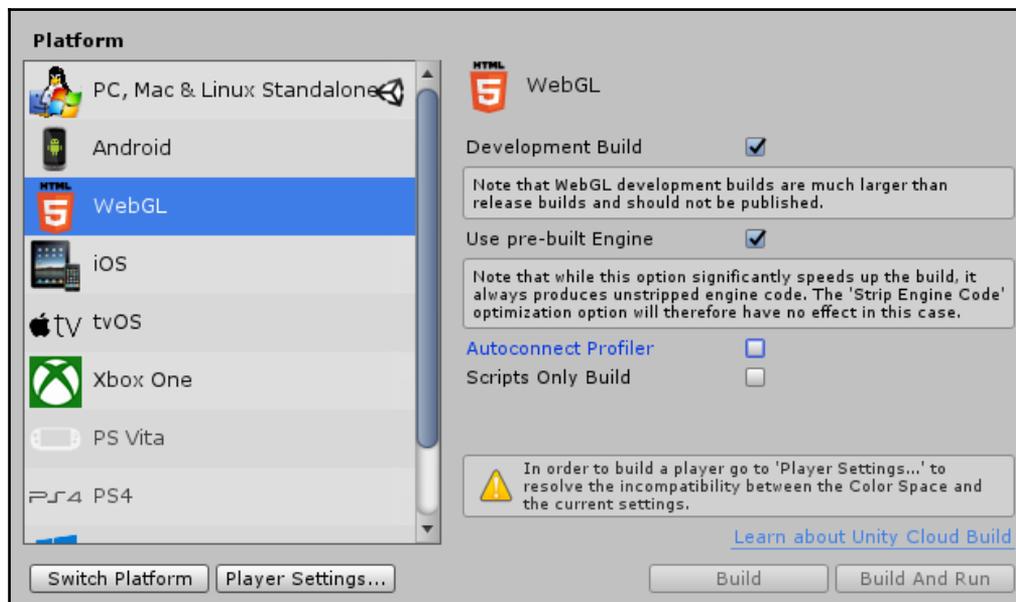
The WebGL build should be at a smaller resolution than a standalone build for performance reasons, the smaller the screen real estate you are rendering, the better performance you can expect. Despite this smaller scale, it is possible for the user to right-click on games deployed in the web player in-browser to switch to the fullscreen mode.

As the WebGL build does not have the exact same features as the standalone and because computers with the internet and a browser may vary from the very slow ones to super fast gaming machines, you should pay attention and take some steps to make your game playable on all those different hardware configurations:

- Build a game menu that allows the users to manually vary the video resolution as well as the quality of the engine, and other CPU/GPU-intensive options such as post-processing camera effects or HDR.
- Test carefully on different hardware and eventually take decisions such as using different LOD, POSTFX quality, and general engine quality for different tiers of hardware. Use the tiers in the **Graphic Settings** along with those tweaks.
- Use all the optimizations techniques that Unity has to offer for your project: Occlusion Culling, Baked Lighting, Static and Dynamic batching, GPU Instancing, and special Post Processing FX pipelines.
- Prefer Forward over Deferred rendering and **Gamma** over Linear **Color Space**.

Building for the web

In Unity, open the **Build Settings** by navigating to **File | Build Settings**, and highlight **WebGL Player** in blue under **Platform**:



As stated in the preceding information box, there is no need to hit **Switch Platform**, as Unity will switch while building to whatever platform is highlighted in blue.

As you can see, the build buttons are grayed out and you cannot build the game at present. The Build Settings are warning you that you should switch player's rendering **Color Space** from **Linear** to **Gamma**. Press the **Player Settings...** button to quickly access player settings again, and if you forgot to do so, change this setting. Now you can build.

Click on the **Build** button. You will be prompted for a location for the build. Unity will handle the relevant conversions and compressions necessary to create a build that will run well on the web, and create this to the specifications we made earlier in the **Player Settings**. You'll be prompted to specify a name and location to save the file. Enter the filename after having carefully chosen the folder destination and then press the **Save** button at the bottom to confirm.

When the build is complete, the operating system will switch to the window your build is saved in, showing you the exported content along the `index.html` file containing the embedding code required to load up the WebGL build. To play the game, open the HTML file in a WebGL-compliant browser, such as Edge, Firefox, Safari, or Chrome.

Adapting and building for the mobile platform

While we suggest you to try to port a 2D example of the first chapters to mobile platform as a first attempt, you can try porting the 3D adventure as well. Of course, you will have to tear down general graphic appeal by using a different approach to lighting and also extra special care in optimizing AI, LOD, and so forth.

The mobile platform is perhaps the less performing platform; the hardware is much smaller and compact than on PCs/consoles, hence the degree of optimizations and extra steps to take is larger. In the last decade, mobile device hardware has increased its power significantly, hence you can achieve pretty good results with your 3D game on the mobile platform too.

For more information about mobile optimizations, read this page of the official manual:

<https://docs.unity3d.com/Manual/MobileOptimisation.html>.

Adapting for Android

The Android platform is for sure the more fragmented platform in terms of hardware performance.

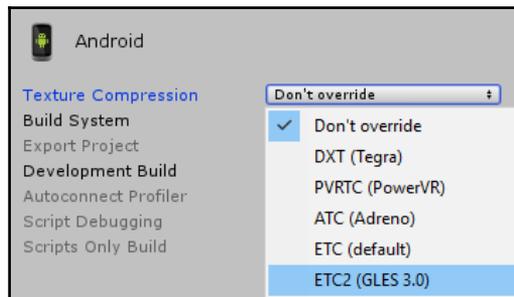
You can have users with very old Android 6, 5, or even version 4 devices with low hardware capabilities. At the time of writing, almost every Android device still actively supports OpenGL ES 2.0, and this is good news; we can at least support bump mapping and some image effect as well as other cool features! Almost every new-generation (2015 and later) Android device supports OpenGL ES 3.0, which opens support for a lot of new cool features.

The downside of this hardware fragmentation might lead the developers to have the need for building different .apk for a specific family of devices or specific producers, according to their hardware specifics. Luckily, the Google Play developer console supports the submission of a multiple number of builds of the same application, and restricts each version to a group of devices the developer will carefully set up.

For a deep optimization guide, look at the following pages: <https://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html>.

Texture Compression formats

Another very important setting to look at for the Android build, depending on the device family you are targeting, is the **Texture Compression** format. The Google Play Store backend allows us to upload a different APK for different producers or family of devices as well as one different APK for each device type, if you need to. Thanks to this possibility, you can target each mobile hardware producer using the more appropriate texture compression method for the targeted GPU:



The Texture Compression list of options

For example, some producer hardware prefer the DXT (mostly the older NVIDIA Tegra CPU for mobile) compression format, while another producer may mount the GPU of Apple devices, the PowerVR GPU; in that case, you want to choose, like you would do on iOS, the PVRTC compression format. In most cases, you will choose ETC, the newer and better compression format for the OpenGL ES 2 platform, or ETC2 if you target devices with an OpenGL ES 3 capable GPU. The **Don't override** option will leave the textures imported as they were imported the first time.

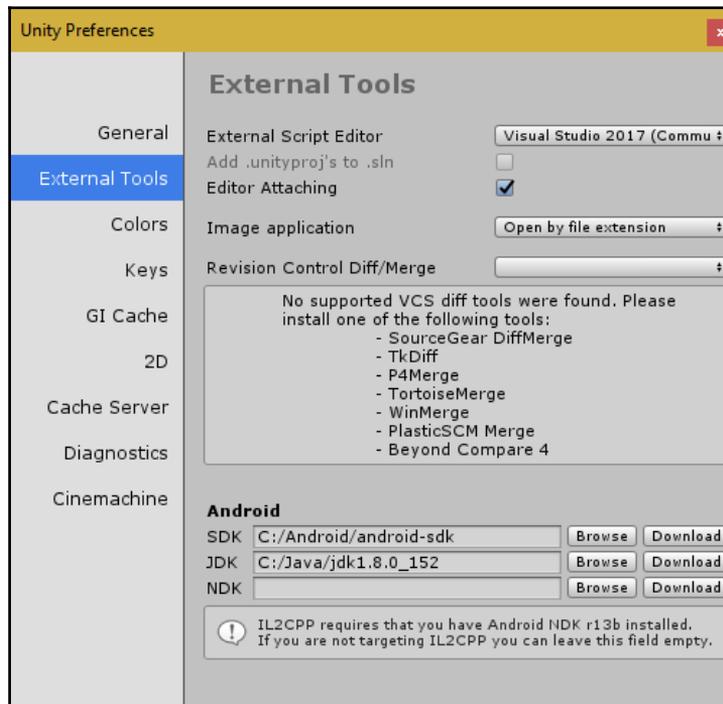
Building for Android

Building for the Android platform requires you to install the Android SDK on your system. At the time of writing, Android SDK comes in two flavors: with or without Android Studio. If you plan to export a full Android project that you can open in Android Studio, you can then choose to export a **Gradle** project (which will be opened with and compiled from Android Studio) or to create an APK (Android Package) archive directly, which can be ready for submission.

Because the Java environment can be used also for other things from the Android ecosystem, Unity can specify what should be the exact path for the three important components of Android development; the first two are mandatory for deploying to the device:

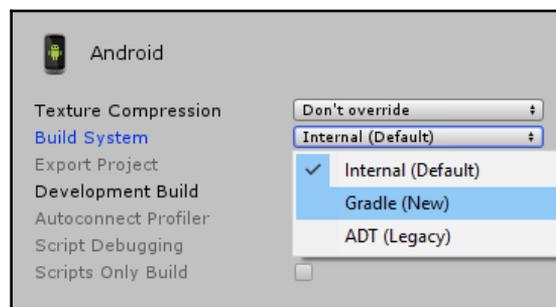
- SDK, Android Software Development Kit path
- JDK, Java Development Kit path
- NDK, Native Development Kit path

You can find these settings in the **Unity Preferences**, as shown in the following screenshot:



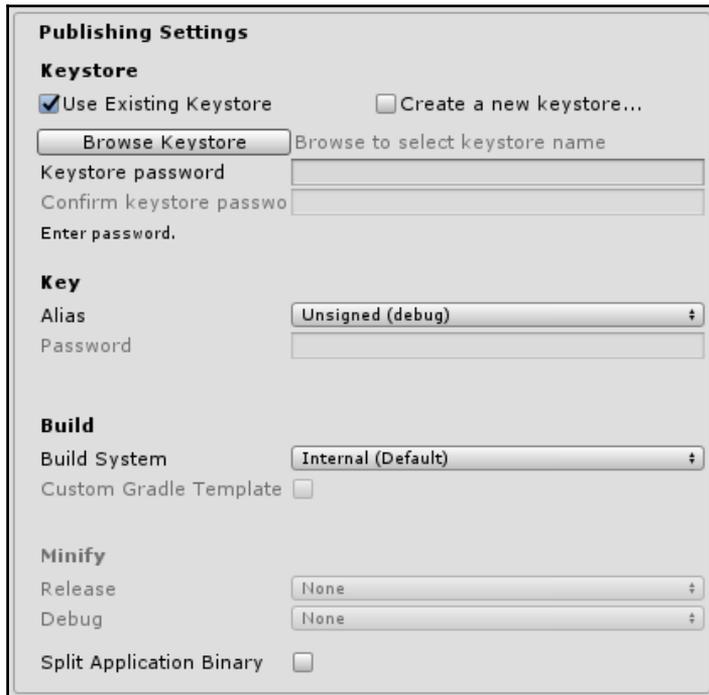
Choosing the preferred build system

You can choose the build system by changing the **Build System** option in the **Build Settings**, among the three available options: **Internal (Default)**, **Gradle (New)**, and **ADP (Legacy)**:



Publishing settings

The **Publishing Settings** section of the **Player settings** contains the data you will embed as metadata in your application to validate any submission to the store. To build a development release to be run on your device, you don't need to specify anything here:



The screenshot shows the 'Publishing Settings' dialog box. It is divided into several sections: 'Keystore', 'Key', 'Build', and 'Minify'. In the 'Keystore' section, 'Use Existing Keystore' is checked, and 'Create a new keystore...' is unchecked. There is a 'Browse Keystore' button and a text field for 'Keystore password' with a 'Confirm keystore passwo' field below it. The 'Key' section has a dropdown menu for 'Alias' set to 'Unsigned (debug)' and an empty 'Password' field. The 'Build' section has a dropdown for 'Build System' set to 'Internal (Default)' and an unchecked 'Custom Gradle Template' checkbox. The 'Minify' section has dropdowns for 'Release' and 'Debug' both set to 'None', and an unchecked 'Split Application Binary' checkbox.

To be able to distribute your game on Google Play Store, you will have to create a Google Developer account.

This will cost 25\$ lifetime.

For more information about Android Studio and Android SDK and how to install them on Mac/Windows systems and get your build ready for device deployment, go to <https://developer.android.com/studio/index.html>.



Building for iOS

Building for iOS can be expensive, especially if you don't have a Mac computer already. First of all, you need a Mac computer with the latest macOS to test an Apple mobile device such as the iPod, iPhone, or iPad. There are many models of each type around, and you can go as low as <300\$ for a new iPod (the same features as that of an iPhone, but without the phone) or, if your game is for the bigger screen, you can find an iPad Mini for less than 400\$. Of course, ensuring that all devices of all generations are running your game properly can be really expensive: iPod, iPhone, iPhone SE, iPad Mini, iPad, and iPad Pro all have different hardware and screens, and buying them all can empty your wallet pretty soon; so, if you are low on budget, you might want to consider external beta tester people who already own the devices for the testing phase.

Furthermore, you will need to sign up for an account to become an approved Apple Developer. The Apple Developer Program for iOS costs 99\$ per year. When these steps are accomplished, you will need to set up and download development and distribution certificates, and install them properly in your Mac Keychain. After the certificates are correctly installed in the Apple Developer Portal, you will create applications or team provisioning profiles that will be automatically installed on your test devices connected to your development MacOS-based computer when you build with Xcode.

You can read more at <https://docs.unity3d.com/Manual/iphone-accountsetup.html>.

Xcode is the Apple **Integrated Development Environment (IDE)**. These profiles will enable test applications to run on developer-enabled Apple mobile devices. Unity will export a ready-to-build Xcode project for you.

You can read more at <https://docs.unity3d.com/Manual/StructureOfXcodeProject.html>.

Without these certificates installed on your Mac KeyChain and Provisioning Profiles installed on the testing devices, you will not be able to deploy a debug release on your phone/tablet. For further information about XCode and iOS SDK and how to install them on the Mac and get your build ready for device deployment, visit <https://developer.apple.com/programs/>.



Some time ago, Apple acquired TestFlight, a quick method to let external and remote coworkers test your application without the hassle of manually sending and installing a provision profile and automatically adding the device UniqueID to the list of devices eligible for testing, by installing it along with the Profile by simply clicking on a link in an email. You can also use this method on your devices, instead of going through application profile creation on the Apple Developer Portal.

There are several iOS-specific options in the **Publishing Settings** section of the **Player Settings** that will not be covered on this book; instead, head to <https://unity3d.com/learn/tutorials/topics/mobile-touch/building-your-unity-game-ios-device-testing> for an in-depth tutorial on iOS deployment.



Official documentation about the building process is available at <https://docs.unity3d.com/Manual/iphone-BuildProcess.html>.

For a fully-fledged iOS building tutorial, head

to <https://unity3d.com/learn/tutorials/topics/mobile-touch/building-your-unity-game-ios-device-testing>.

For more information, check the official documentation

at <https://docs.unity3d.com/Manual/iphone-GettingStarted.html>.

A guide to the various generations and family of Apple iOS devices can be found at <https://docs.unity3d.com/Manual/iphone-Hardware.html>.

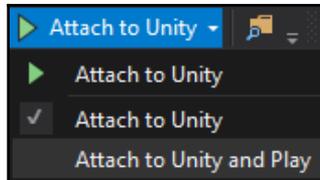
We covered almost everything important for these platforms, which are the more diffused around the world. You can also check out the Unity official documentation about publishing builds for those platforms at <https://docs.unity3d.com/Manual/PublishingBuilds.html>.

Testing and debugging

We will now face one of the most important and delicate parts of game development—when the game seems ready, but it is not, and you need to perform a series of tests and debug your code, to find bottlenecks, solve issues, and improve the performance. Unity Editor and Visual Studio are your friends, and both have valuable tools to help you perform these steps.

Debugging with Visual Studio 2017

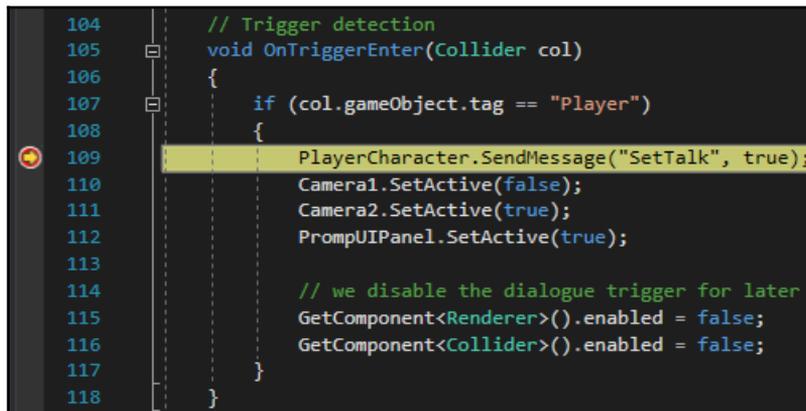
To enable debugging of your work with Visual Studio, you have two choices. In Visual Studio, when you open a Unity project, you will see a pull-down menu/button choice action at the top of the interface with *Attach To Unity* and a green arrow:



By clicking on this button, you will attach VS debugging to the Unity Editor. When you start that option, Visual Studio will start the debug mode, and wait for Unity Editor to start the **Play** mode to attach. You can force Unity Editor to automatically go into the **Play** mode when pressing that button by choosing the other option in the menu: **Attach To Unity and Play**:

```
104 // Trigger detection
105 void OnTriggerEnter(Collider col)
106 {
107     if (col.gameObject.tag == "Player")
108     {
109         PlayerCharacter.SendMessage("SetTalk", true);
110         Camera1.SetActive(false);
111         Camera2.SetActive(true);
112         PromptUIPanel.SetActive(true);
113
114         // we disable the dialogue trigger for later
115         GetComponent<Renderer>().enabled = false;
116         GetComponent<Collider>().enabled = false;
117     }
118 }
```

When doing so, you are able to put breakpoints at a specific point in the code, to have the execution stopped; Visual Studio gain the focus, enabling the developer to perform step-by-step debugging and understand exactly what is going on:

A screenshot of the Visual Studio code editor showing a C# method. The code is as follows:

```
104 // Trigger detection
105 void OnTriggerEnter(Collider col)
106 {
107     if (col.gameObject.tag == "Player")
108     {
109         PlayerCharacter.SendMessage("SetTalk", true);
110         Camera1.SetActive(false);
111         Camera2.SetActive(true);
112         PrompUIPanel.SetActive(true);
113
114         // we disable the dialogue trigger for later
115         GetComponent<Renderer>().enabled = false;
116         GetComponent<Collider>().enabled = false;
117     }
118 }
```

A red circular breakpoint icon is placed on the left margin next to line 109. The line of code on line 109 is highlighted in yellow.

This practice is very useful to find bugs and code misbehavior, but cannot help when the issue is a graphic or CPU bottleneck due to the nature of the game, or due to bad design.

With the internal Unity **Profiler** window open and while in recording mode and the game in play mode, you can understand what is going on in the hardware, memory, and CPU and GPU load, and check methods execution timing, to check what exactly is slowing your game down.

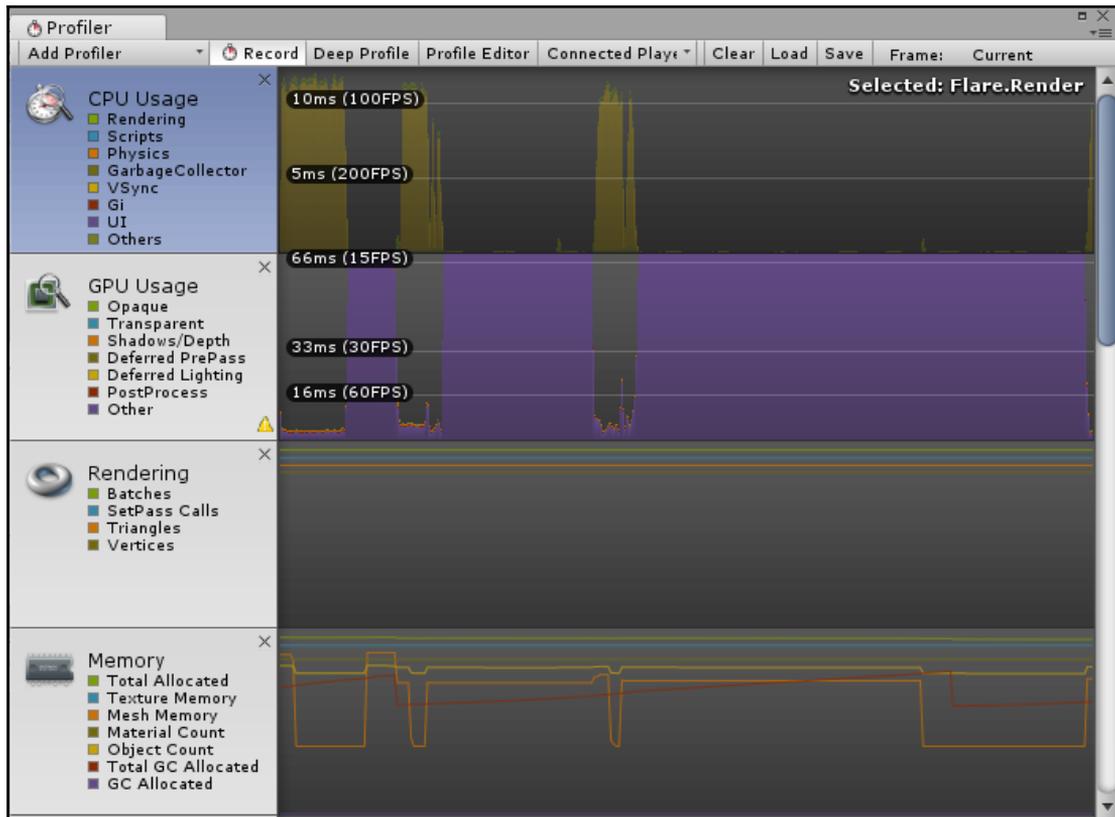
Testing and profiling with Unity Editor

There are various tools to debug and/or have an overview of your game performance in the various situations your game may step through its execution or debug a specific object draw call. The physics debug will help you debug your Rigidbody components while the game is running. The frame debugger will help you understand the passes the GPU takes to render an object. The Unity internal profile will help you measure the performance of the various areas of the game against the hardware and the time used by the CPU to perform a specific method or routine.

Unity Profiler

Unity (internal) **Profiler** is the best way to detect hiccups and memory leaks, CPU or GPU over usage, and other important performance factors. You can profile CPU, GPU, the memory, and one or more of them can run together to monitor the execution of your game. The next screenshot illustrates the **Profiler** while the game is running.

You can easily add or remove profilers with the **Add Profiler** menu:



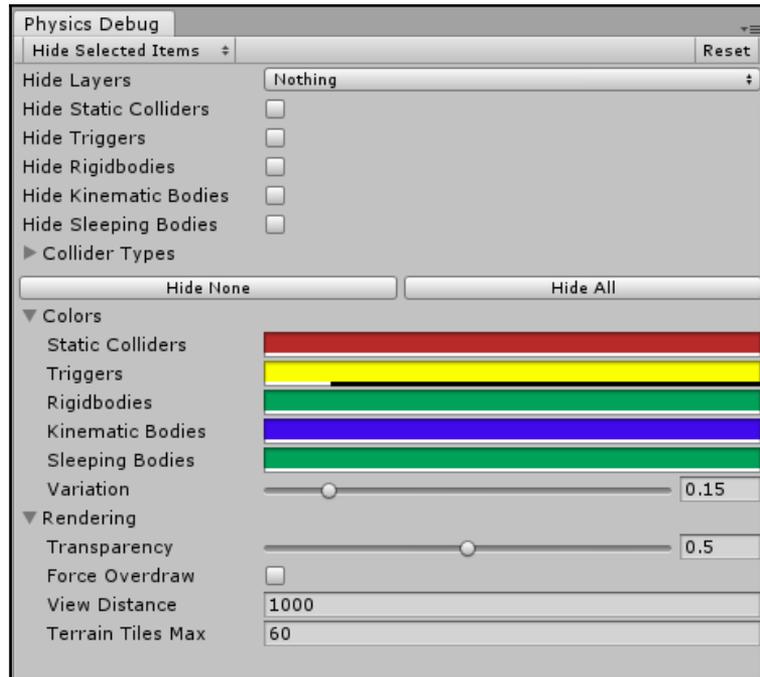
You can even know how many milliseconds a specific internal or scripted method will take to execute, and the CPU usage and garbage collector memory allocation:

Hierarchy		CPU:11.71ms GPU:3.52ms		No Details		
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ Initialization.PlayerUpdateTime	95.3%	0.0%	1	0 B	11.17	0.00
▶ Camera.Render	2.7%	0.2%	1	0 B	0.32	0.02
Overhead	0.4%	0.4%	1	0 B	0.05	0.05
Profiler.FinalizeAndSendFrame	0.3%	0.3%	1	0 B	0.03	0.03
▶ EarlyUpdate.UpdateMainGameViewRect	0.1%	0.0%	1	0 B	0.01	0.00
▶ PreUpdate.SendMouseEvents	0.0%	0.0%	1	0 B	0.00	0.00
▶ PostLateUpdate.UpdateAudio	0.0%	0.0%	1	0 B	0.00	0.00
▶ GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
▶ PostLateUpdate.UpdateCustomRenderTextures	0.0%	0.0%	1	0 B	0.00	0.00
▶ FixedUpdate.Physics2DFixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
▶ FrameEvents.NewInputSystemBeforeRenderSen	0.0%	0.0%	1	0 B	0.00	0.00
▶ PostLateUpdate.UpdateAllRenderers	0.0%	0.0%	1	0 B	0.00	0.00
EarlyUpdate.UpdateInputManager	0.0%	0.0%	1	0 B	0.00	0.00
Camera.FindStacks	0.0%	0.0%	2	0 B	0.00	0.00
▶ FixedUpdate.ScriptRunDelayedTasks	0.0%	0.0%	1	0 B	0.00	0.00
▶ FixedUpdate.NewInputEndFixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
▶ PreUpdate.IMGUISendQueuedEvents	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.NewInputBeginFrame	0.0%	0.0%	1	0 B	0.00	0.00
GUIUtility.SetSkin()	0.0%	0.0%	1	0 B	0.00	0.00
▶ PostLateUpdate.PlayerUpdateCanvases	0.0%	0.0%	1	0 B	0.00	0.00
▶ PreUpdate.AIUpdate	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.PlayerCleanupCachedData	0.0%	0.0%	1	0 B	0.00	0.00
▶ FixedUpdate.PhysicsFixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
ReflectionProbes.Update	0.0%	0.0%	1	0 B	0.00	0.00
▶ UIEvents.CanvasManagerEmitOffScreenGeometr	0.0%	0.0%	1	0 B	0.00	0.00
▶ EarlyUpdate.DirectorSampleTime	0.0%	0.0%	1	0 B	0.00	0.00
▶ FrameEvents.OnBeforeRenderCallback	0.0%	0.0%	1	0 B	0.00	0.00

I suggest you look at this excellent series of videos from Joachim Ante about profiling Unity applications: <https://youtu.be/0969La1B7vw>.

The Physics Debug window

The **Physics Debug** window is a new feature that lets you debug the behavior of a given Rigidbody's physics simulation:

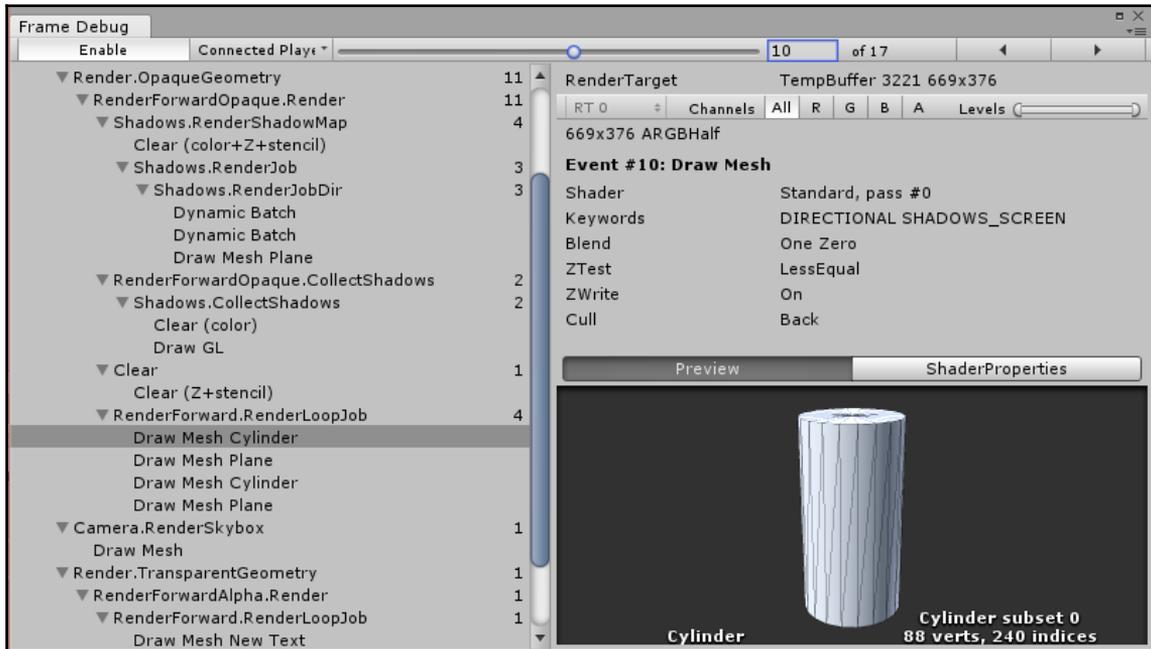


With this window open, you can better inspect the physics simulation in general and set the colors for each object to recognize them better in the **Scene** view.

The Frame Debug window

The **Frame Debug** window allows us to debug the several steps taken by the GPU to obtain the final rendering. This is especially useful when you are debugging visual artifacts or testing the Z buffer of a shader. In general, it lets you inspect an object in all the steps required by the GPU to actually draw the final thing; for example, you can understand the reason why a single object requires eight draw calls to render.

Here, we have the **Frame Debug** window inspecting a simple cylinder:



A draw call is performed by the engine when it needs to draw an object on screen. It will show all the *render passages* the GPU pipeline will go through, step by step, to finally render the object. This is especially useful when trying to optimize WebGL or mobile builds, but also in general.

Sharing your work

It is important that you share your work with others, not only to show off your development skills but also to get feedback on your game and allow members of the public with no prior knowledge of your project to test how it works. Also, consider the value of social media in sharing your work; a strong community for Unity exists on both Facebook, through various groups, and on Twitter, using the **#unity3d** hashtag.

Unity 2017 Personal Edition and Visual Studio 2017 Community Edition are the perfect solution for learner, indie, and startup developers as they have all the important features in place and a cutting-edge script editor used by the industry.

The only limitation will be how well your future business will be with game development with Unity. When you overtake the \$100k gross income per year limit Unity Personal will not be enough, and a Plus or Pro license must be acquired.

Over the course of this book, we covered the essential topics to get you started in development with the Unity game engine. In working with Unity, you'll discover that with each new game element you develop, new possibilities open up in your knowledge. Fresh ideas and game concepts will come more easily as you add further Unity development and scripting knowledge to your skill set. In this chapter, we'll conclude your introduction to Unity by looking at the following:

- Sharing your game
- Measuring performance and optimize more
- Approaches to testing and finalizing your work
- Where to go for help with Unity and what to study next

Sharing WebGL builds

In addition to sharing your game with your audience on your own website, there are also several independent game portal sites available that act as a community for developers sharing their work. Here are some recommended sites you should visit once you are ready to share your work with the online community:

- www.kongregate.com
- www.facebook.com (as a mobile app linked to Facebook or as a WebGL app that Facebook users can play within the website)
- www.tigsource.com (The independent gaming source)
- <http://forum.unity3d.com> (showcase area)

While Facebook requires its own monetization payment system to be implemented in your game, Kongregate and other websites pay according to the time your game is played, thanks to the ads they implement on their website.

Remember to search the internet from time to time; there might be new websites that accept Unity WebGL builds and pay per play.

Publishing on mobile stores

Mobile releases will be brought to these official stores:

- Google Play
- App Store

This book is not meant to give complete information about the path to follow to successfully publish your game on various stores, because it would make this chapter really huge. Instead, recently Unity's official manual included precise guides for each single platform, so I will provide here only external links for each platform store guidelines:

- More information about Google Play Store submission:
<https://developer.android.com/studio/publish/index.html>.
- More information about iOS App Store submission:
<https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html>.

Publishing for the desktop

Desktop standalone releases can be release to these official stores:

- Windows Store
- Mac App Store
- Steam (multi-platform)

For more information about publishing your Unity game on the Mac App Store, read <https://developer.apple.com/macos/distribution/>.

For more information about publishing your Unity game on the Windows Store, start with the Unity official manual at: <https://docs.unity3d.com/Manual/windowsstore-gettingstarted.html>.

Recently, Steam changed its policy for app submission, so, instead of asking for votes on Greenlight, now, you have to pay a single fee for each game you want to submit: <http://store.steampowered.com/sub/163632/>.

Digital Content Creation tools

This is a partial list of some of the most used Digital Content Creation packages, those in bold are the ones used for the book:

- **The Gimp / Adobe Photoshop**: Image editing and texture creation
- **Maya / 3D Studio Max / Blender**: Modeling, UV mapping, and animation
- **Ultimate Unwrap 3D**: Model format converter and cutting-edge UV mapping tools
- **Fuse**: Character design, auto-rig, and character animation with Mixamo
- **Soundforge/Audacity**: Sound editing/design
- **Articy Draft**: Screenplay, team story writing, and quest and dialogs integration



Mixamo has recently changed its policy and workflow; check its website for more information:

<https://www.mixamo.com/>

<http://www.adobe.com/it/products/fuse.html>

Future of Unity and MonoBehaviour

We have seen recently a lot of **Entity Component Systems (ECSs)** written in plain C#, specifically for Unity, as well as Inversion Of Control and Dependency Injection design patterns; all these have the aim of becoming independent from the MonoBehaviour-centric approach Unity has always had. Recently, after reading Unity Technology's road map, I was able to plan a very similar approach from being released by Unity Technology itself: an ECS for Unity built on the top of Unity API. Keep an eye on all that, as this will mean changing the coding style a lot, aiming to make the code easier to maintain and more powerful in its implementations, but still, different from what we are used to. For a deeper look at this kind of approach in programming, I suggest you to watch the latest Unite 2017 video from CTO Joachim Ante: <https://youtu.be/tGmnZdY5Y-E>.

To mention one ECS implementation to use with Unity that is actually available for free, I feel to suggest *Sebastiano Mandala's Svelto ECS*, a fully-fledged Entity Component System and its `Svelto.Task` class, meant to replace Unity's co-routines.

Have a look at his GitHub repository page: <https://github.com/sebas77/Svelto.ECS>.

To better understand the system and its approach, you should read also his series of articles: <http://www.sebaslab.com/ecs-1-0/>.

For a taste of some games made with Unity and *Svelto ECS*, you can download for free the Robocraft client for PC/Mac/Linux here: <http://robocraftgame.com/landing.php> or on Steam at: <http://store.steampowered.com/app/301520/Robocraft/>.

Testing and further study

By reading through the Unity manual, followed by the component reference and script reference, available both online and as part of your Unity software installation, you'll begin to understand what's the best way to create all types of game elements, which may not apply to your current project, but should flesh out your understanding to help you work more efficiently in the long term:

- **Component reference:**
<http://www.unity3d.com/support/documentation/Components>
- **Scripting reference:**
<http://www.unity3d.com/support/documentation/ScriptReference>
- **Unity manual:** <http://www.unity3d.com/support/documentation/Manual>
- **MSDN C# reference:** <http://msdn.microsoft.com/en-gb/vstudio/hh341490>

Learn by doing

In addition to referring to the Unity manual, component, and scripting references, one of the most valuable approaches that you can take to improve your game development skills is that of *rapid prototyping*. Rapid prototyping is the process of creating simplified versions of game ideas quickly and with basic visuals, focusing on the gameplay of the idea itself. Try to think of simple game mechanics, and then set yourself the task of finding out how to build them. By using idea generation to drive your learning, you will find that motivation will help you maintain your concentration, as you will be focused on goals instead of an overall intangible leveling up of your knowledge.

Testing and finalizing

When considering game development, you should be very aware of the importance of testing your game among users who have no preconceptions of it whatsoever. When working on any creative project, you should be aware that in order to maintain creative objectivity, you need to be open to criticism and that testing is just as much a part of that, as it is a technical necessity. It is all too easy to become used to your game's narrative or mechanics, and we're often unable to look at the game in a detached manner, similar to how a player might respond to it. Always try and place yourself in the position of the player, instead of looking at the game you're making as a game designer; there are bound to be parts of the game that are obvious to you, but may not make sense to a player.

Public alpha testing and open beta

When looking to test your game, try and send test builds to a range of users who can provide test feedback for you with the following variations:

- **Computer specification:** Ensure that you test on differently configured hardware, and get feedback on performance.
- **Format:** If working on a standalone build, try sending a build for both Mac and PC where possible.
- **Language:** Do your test users all speak the same language as you? Can they tell you if your game interface makes sense?
- **Experience:** What kind of games do your testers play ordinarily, are they casual or hardcore gamers, and how do they find your game in terms of difficulty? What parts of the game are causing issues for them?

After testing your game in **alpha** form, a test version that you and other developers test, you should hand your game over to a collection of public testers. You are handing them what is referred to as a **beta** test of your game. By formalizing the process, you can make the feedback you get about your game as useful as possible; draw up a questionnaire that poses the same questions to all testers, while asking not only questions about their responses to the game, but also asking for information about them as a player.

In this way, you can begin making assertions about your game, such as the following:

Players aged 18 to 24 liked the mechanic and understood the game, but players of over 45 years did not understand it without reading the instructions.

Casual gamers found that the game didn't have points of interest close enough together to keep them playing; consider level design changes.

All this can be done in addition to technical information such as the following:

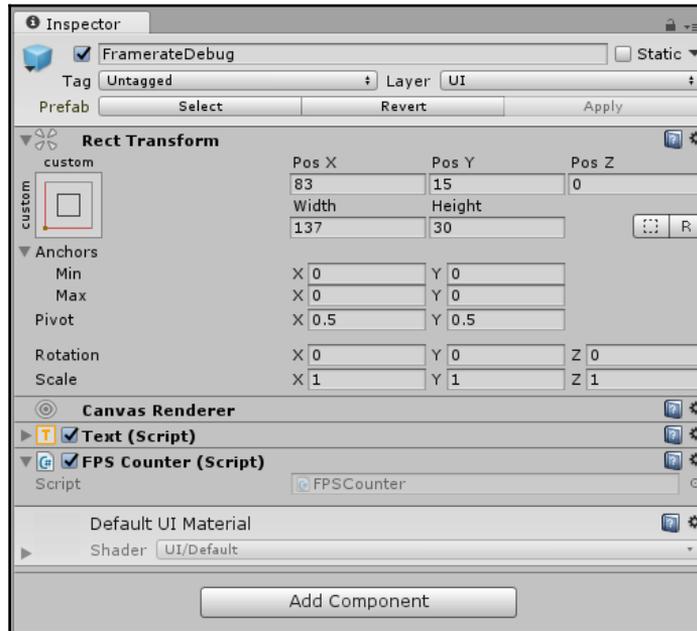
Players with computers under 2.4 GHz processing speed found the game responded sluggishly, averaging 15 to 20 frames per second; consider model detail and texture compression.

Frame rate feedback

As you test your game in the editor, you have access to statistics of performance using the **Stats** overlay on the **Game** view (click on the **Stats** button). However, like a player of your test build, a member of the public won't be able to see this, so we should consider building such tools into our game. You can enable the stats overlay in the **Game** window to know a lot more about just the overall frame rate of your game, but you won't have this chance in the built executable. In order to provide testers of your game with a means of providing specific feedback on technicalities such as frame rate (the speed at which game frames are drawn during play), you can provide your test build with a UI Label (Text component) element, telling them this information.

To add this to any scene, let's take a look at a practical example. Open the scene you wish to add a frame rate screen overlay to, in our example, the **Island** scene, and create a new UI Text object to display the information. Right-click on the Canvas game object to show the hierarchy drop-down menu and choose **UI | Text**.

Doing so will create the new UI Text gameobject as a child of the Canvas. Rename the newly created object as **FramerateDebug**, and then in the UI Text component of the **Inspector**, set the **Anchor** to **lower bottom corner**, and the text **alignment** to **center**:

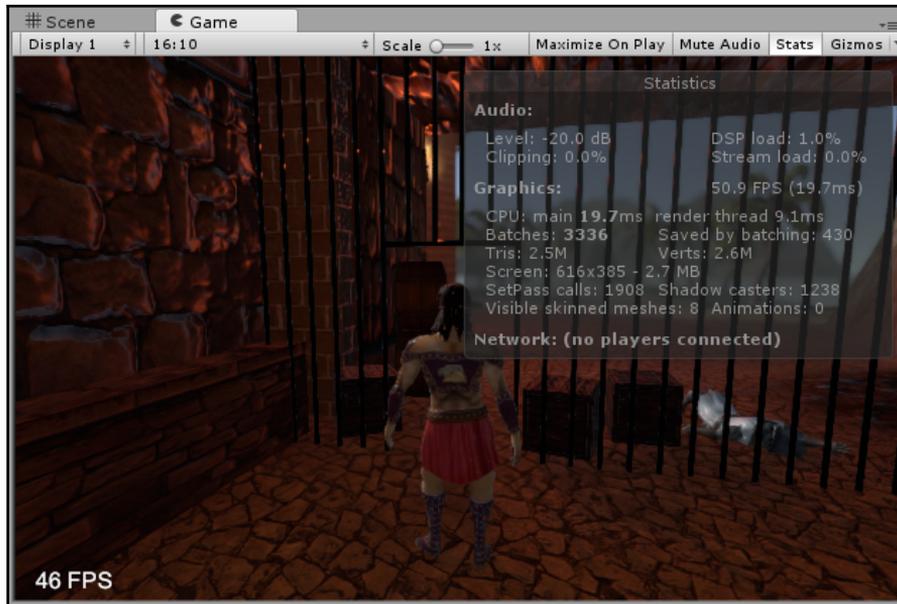


The Inspector for our GameObject holding the FPSCounter component

We will take advantage of the ready-made script in the `Assets/Standard Assets/Utility` folder. Add it to the inspector by adding a component with the **Add Component** button and search `FPSCounter` to find the C# script and add it to the game object.

As you can see, it starts by adding an additional using directive, `using UnityEngine.UI;`. This namespace will allow us to use a UI-related component API. Now, because we might wish to make this a script that we can reuse for other games, we should ensure that it can be dropped onto an object in another project and still work perfectly. Part of doing this is ensuring that the script contains the `[RequireComponent]` directive to ensure that a UI Text component is present. We have used these before, so hopefully you remember how; if not, you need to add `[RequireComponent (typeof (Text))]`.

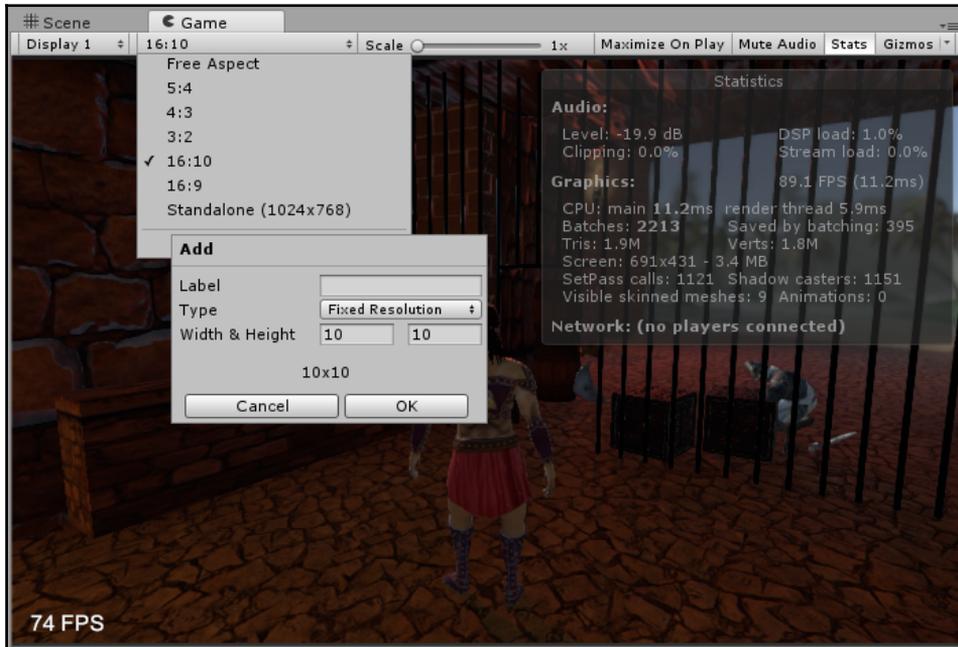
Now, press Play and test your scene; you will see the UI Text displaying the frame rate at the bottom-left corner of the screen:



Your game will perform differently outside the Unity Editor. As such, readings from this FPS display should only be noted once the game is built and run either as a web player or a standalone version; ask your test users to note the lowest and highest frame rates to give you a range of readings to consider, and if there are particularly demanding parts of your game, for example, complex animation or particle effects, then it can be worth asking for readings from these separately. As the game performs differently outside the Unity Editor, the player needs to play the game at least in full HD resolution for the higher resolution, which is 1920x1080 pixels, a resolution you will hardly have on the **Game** view. Unless you detach it from the editor and extend widely, the values from this frame rate display will be different from those given in the **Stats** tab of the **Game** view.

Testing different video resolutions

You can quickly test different aspect ratio video resolutions by adding presets to the **Game** window. Click on the drop-down menu before the zoom scale view slider to quickly choose among the widely diffused **4:3**, **16:9**, or **16:10**, or add a preset with a fixed custom video resolution or with a different aspect ratio with the *plus (+)* button, as illustrated:



In this way, you will have the chance to switch the final output video resolution while the game is running and test all possible cases. This is also very useful with the mobile platform when you don't have all the possible devices on the market available to be able to test all their different video resolutions, but you have to.

Optimizing more

Improving the performance of your game as a result of testing is easily an entire field of study in itself. However, to improve your future development, you'll need to be aware of basic economizing in the following ways:

- **Spotting bottlenecks:** Your game's performance may suffer due to a number of things; the key is to spot where this is happening, is a certain scene causing lower performance? Is the game only slower when encountering a high-detail model or high-output particle system? Try and be systematic when searching for what is making your game run slowly, and always ask your test users for a scenario you can reproduce in order to spot the problem.
- **Polygon counts:** When introducing 3D models, they should be designed with low polygon counts in mind. So, try and simplify your models as much as possible to improve performance.
- **Draw distance:** Consider reducing the distance of your far clip plane in your cameras to cut down the amount of scenery the game must render.
- **Occlusion culling:** Only available in the Pro version of Unity, occlusion culling helps avoid overdraw-rendering of unnecessary objects that are in front of one another in your camera view. There's more on this in the online Unity manual at <http://unity3d.com/support/documentation/Manual/OcclusionCulling.html>.
- **Texture sizes:** Including higher-resolution textures can improve the visual clarity of your game, but they also make the engine work harder. So, try and reduce texture sizes as much as possible, using both your image editing software and the Unity import settings of your texture assets.
- **Script efficiently:** As there are many approaches to differing solutions in scripting, try and find more efficient ways to write your scripts. Start by reading the Unity guide to efficient scripting online, available at http://unity3d.com/support/documentation/ScriptReference/index.Performance_Optimization.html.

Approaches to learning

As you progress in this book, you will need to develop an approach to further study, which keeps a balance between personal perseverance and the need to ask for help from more experienced Unity developers. Follow the advice laid out here, and you should be well on your way to helping other community members as you expand your knowledge.

Don't reinvent the wheel

This may seem incongruous in view of the previous point, but as you start tackling more complex tasks in your game development, you'll learn that there is little point in reinventing the wheel, recreating something that another developer has already solved and offered to the community. For example, if you are tackling the task of enemy AI or smart camera behaviors or anything you might need for your project, it is worth looking at the various solutions that might be available to you for this, rather than writing your own code from scratch. There are many plugins and ready-made assets for Unity available to solve almost any task, available primarily from the **Asset Store** window inside the editor (**Window | Asset Store**) and from various sites across the web.

I recommend that you check out the next two sections and the links for extensions and content that have already been created by Unity Technology and third parties, which can help a lot with your game development.

Editor extensions

Editor extensions are plugins that are written in Unity itself, taking advantage of the existing Unity Editor UI. It's incredible how many things can be done just by extending the Editor.

Some of them are paid, and some are free from Unity Technologies, but they all are worth checking out and using.

As the latest Unity release has integrated some of these packages to be built-in the editor, I will group them in: Recently built-in, on the Asset Store from Unity Technology and on the Asset Store from third parties.

Recently built-in:

- **Text Mesh Pro:** A new pixel-perfect geometric mesh-based UI solution to display text
- **Timeline:** A new fully-fledged animation system for the direction of complex cut scenes and movies

On the Asset Store from Unity Technology:

- **Cinemachine:** This virtual camera system allows developers to build a solid and compact cinema director setup that is perfect for storytelling, FPS, adventure games, and other titles. This package fits particularly well with the **Timeline** feature; get it from: <http://u3d.as/GJQ>.
- **PostProcessing Stack:** This extension allows you to *bake* multiple camera post-processing effects into one single pass, saving performance and taking your game to the next level. You can find it at <http://u3d.as/KTp>.

On the Asset Store from third parties:

- **PlayMaker:** This useful plugin allows you to design your game logic, or part of it, with a powerful visual node system. Rumors say PlayMaker might become built-in with future versions of Unity.
- **AQUA water/river set:** When you want to take your water simulation to the next level: <http://u3d.as/mVN>.
- **Enviro sky and weather:** For a dynamic sky and sunlight as well as weather conditions simulator, look at <http://u3d.as/cGY>.
- **Anima:** A fantastic package to add skeletal animation and physics to your 2D game characters: <http://u3d.as/GGE>.
- **Lips Sync pro:** This editor extension uses an external sound processor to optimize the speed of automatic speech detection calculation, and is the best solution out there to implement lips sync with speech audio in your game. It can use both Blendshapes or Bone animation. Also, check out their Eye Blink solution to let your characters blink their eyes (uses blendshapes as well).
- **Rewire:** Cross-platform advanced input system solution: <http://guavaman.com/projects/rewired/>.
- **Fabric:** Extends Unity's audio functionality and provides an extensive set of high-level audio components that allow the creation of complex and rich audio behaviors: <http://www.tazman-audio.co.uk/fabric>.
- **Streaming AudioSource for WebGL:** This package, available at <http://u3d.as/Dk2>, implements basic support for streaming audio sources in Unity WebGL. If you need to play background music or use some simple 3D sound, consider using this optional drop-in replacement for standard audio sources for Unity WebGL. It's designed to optimize memory consumption in your WebGL project.

In the Chapter 13, *Optimization and Final Touches*, addendum folder of the codes project, you will find a full ending cut scene made with Timeline Cinemachine and the new Post Processing Stack to load when the puzzle of the game level is completed.

Take your time looking at how this scene is prepared and be ready to learn how to do movies with Unity!

Complete projects

Complete projects are fully fledged assets you can download on the Asset Store that include not just an extension or a package to plug into an existing project, but full projects on their own, generally with the purpose of making the base for your game, or giving you instructions and inspirations for development:

- **Lighting Optimization Tutorial:** <http://u3d.as/BNc>
- **Adventure Sample Game:** <http://u3d.as/DRi>
- **Tactical Shooter AI:** <http://u3d.as/k5F>
- **Vehicle Tools:** <http://u3d.as/KJn>
- **Racing Game Template:** <http://u3d.as/i2s>
- **Viking Village:** <http://u3d.as/bqF>
- **Survival Shooter Tutorial:** a 2.5D game making tutorial <http://u3d.as/hCf>
- **Space Shooter Tutorial:** a 2.5D vertical scrolling space shoot them up: <http://u3d.as/66k>

This is just a very small partial selection of what you can find on the store, and much more is appearing every day.

If you don't know how to do it, just ask!

Another useful approach to learning is, of course, to look at how others approach each new game element you attempt to create. In game development, what you'll discover is that often there are many approaches to the same problem, as we learned in *Chapter 5, Character Animation with Unity*, while making our character open the outpost door. As a result, it is often tempting to recycle skills learned when solving a previous problem, but I always recommend double-checking that your approach is the most efficient way.

By asking in the Unity forum (forum.unity3d.com), [answers page \(answers.unity3d.com\)](http://answers.unity3d.com), or on the **Internet Relay Chat (IRC)** channel (irc.freenode.net, room #unity3d), you'll be able to gain a consensus on the most efficient way to perform a development task, and sometimes even discover that the way in which you first thought of approaching your problem was more complicated than it needed to be!

When asking questions in any of the aforementioned places, always remember to first search whether your question has been asked before. It most likely has been, but if not, when asking, remember to include the following points whenever possible:

- What are you trying to achieve?
- What do you think is the right approach?
- What have you tried so far?

This will give others the best shot at helping you out; by giving as much information as possible, even if you think it may not be relevant, you will give yourself the best chance of achieving your development goals. If asking for help with scripting, always remember to use a pasting site, such as <http://www.pastebin.com>, in order to show your code to others without taking up pages of chat room space.

The great thing about the Unity community is that it encourages learning by example. Online, you can find a wide range of examples of everything from scripts to plugins, tutorials, and more. Here, you'll find useful and free-to-use code snippets to supplement the examples of scripted elements within the script reference, and also find information on how to implement it with example downloadable projects. If you can't find your answer there, you can still try asking at Stack Exchange or look on the many other wikis/blogs or on the many GitHub repositories of Unity projects around the globe.

Summary

In this chapter, we looked at how you can export your game to the web and as a standalone project, as well as for the two main mobile platforms, Android and iOS.

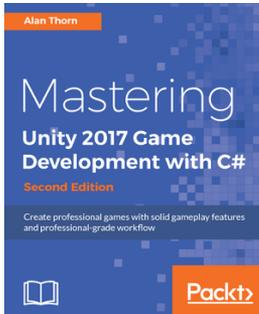
In conclusion, we'll look back at what you have learned over the course of this book and suggest ways in which you can progress further with the existing skills you have developed and where to look for continued assistance, help, and support with your Unity development. We have discussed ways that you should move on from this book, and how you can gather information from test users to improve your game.

Also, check my upcoming blog at tutorials.litobyte.com for addendums and tutorials on AI, VR, Timeline and Cinemachine.

All that remains is to wish you the best of luck with your future game development in Unity. From myself and everyone involved with this book, thanks for reading, and I hope you enjoyed the ride—it's only the beginning!

Other Books You May Enjoy

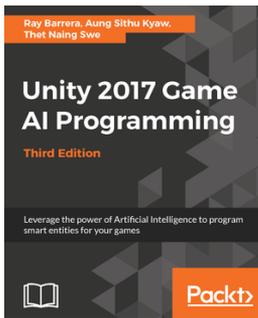
If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Unity 2017 Game Development with C# - Second Edition Alan Thorn

ISBN: 978-1-78847-983-7

- Explore hands-on tasks and real-world scenarios to make a Unity horror adventure game
- Create enemy characters that act intelligently and make reasoned decisions
- Use data files to save and restore game data in a way that is platform-agnostic
- Get started with VR development
- Use navigation meshes, occlusion culling, and Profiler tools
- Work confidently with GameObjects, rotations, and transformations
- Understand specific gameplay features such as AI enemies, inventory systems, and level design



Unity 2017 Game AI programming - Third Edition Ray Barrera, Aung Sithu Kyaw, Thet Naing Swe

ISBN: 978-1-78847-790-1

- Understand the basic terminology and concepts in game AI
- Explore advanced AI Concepts such as Neural Networks
- Implement a basic finite state machine using state machine behaviors in Unity 2017
- Create sensory systems for your AI and couple it with a Finite State Machine
- Work with Unity 2017's built-in NavMesh features in your game
- Build believable and highly-efficient artificial flocks and crowds
- Create a basic behavior tree to drive a character's actions

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

2

- 2D animation basics
 - about 93
 - animated Sprites, importing 93
 - Animator component 97
 - animator state-machine editor 98
 - implementing 102
- 2D character
 - anatomy 129
 - audio, inserting 142
 - collectible items, spawning 133
 - component's source code, analyzing 130
 - component's source code, modifying 130
 - difficulty level, creating 136, 138
 - features, implementing 141, 147
 - game logic, writing 129, 145
 - Layer Collision Matrix 140
 - namespace, defining 132
 - Platformer2DUserControl class, extending 129
 - player, creating 129
 - shadow, adding for character 141
 - sprites, shading with real-time lights 144
- 2D Freeform Cartesian 196
- 2D
 - axis, ignoring 20
 - creating, in 3D 19
 - Sprites 20

3

- 3D
 - about 7
 - cameras 12
 - coordinates 8
 - local space, versus world space 9
 - vectors 11

A

- alpha channels 377
- ancient artifact piece prefabs
 - artifact piece collection script, creating 367
 - collider scale 366
 - collider, rotation 365
 - collider, scaling 365
 - creating 364
 - custom point light 366
 - downloading 364
 - enlarging 366
 - importing 364
 - object, spinning 368
 - placing 364, 371
 - prefab, saving as 370
 - tagging 365
 - trigger collider, adding 366
 - trigger collision detection, adding 369
- Android, Player settings
 - about 586
 - URL 588
- Android
 - adapting 604
- animated Sprites
 - importing 93
- Animator component
 - about 97, 176
 - animation clips, importing for mecanim 170
 - avatar 157
 - avatar, configuring 158
 - hero, building 156
 - model, scaling 173
 - properties 176
- Animator controller
 - about 129, 135, 177
 - animation states 182

- Animator window 178
- animator, controlling through code with
 - parameters 187
- Any State state 183
- character behavior, controlling with state
 - machines 180
 - state machine 179
- animator state-machine editor
 - about 98
 - animation states, transitions 99
 - conditions 99
 - parameters 99
- answers page
 - URL 628
- Any State state 183
- AQUAS (Lite) package
 - reference link 246
- arguments 116
- arrays 382
- Articy Draft
 - URL 330
- Asset Store
 - using 80
- Asynchronous Texture Upload
 - about 546
 - URL 546
- audio options menu
 - creating 475
 - general volume control, adding 477
 - listener script, writing for UI slider elements 477, 479
 - music volume control, adding 476
 - user interaction 479
- Audio Source component
 - reference link 254
- augmented reality (AR) 570
- avatar
 - about 157
 - configuring 158
 - issues, fixing with default bone mapping 163
 - muscle actions, configuring 165
 - settings, configuring 165

B

- billboarding 236
- Boo 24
- build 572

C

C#

- about 24
- Base Monobehaviour methods 52
- behavior script 50
- class 50
- comments 53
- monobehaviour class 51
- scripting 50
- variables 53
- cameras
 - about 12
 - projection mode 12
- Canvas render modes
 - about 460
 - Screen Space Overlay 461, 462
 - world space 463
- Capsule Collider 19
- cartesian coordinate 8
- Cascading Style Sheets (CSS) 459
- Cloth component 19
- Colliders 18
- collision detection
 - about 276
 - animations, playing through animation controller
 - state machine 285
 - assets, creating 276
 - audio event, playing 281
 - character collision detection 278
 - code maintainability 289
 - colliders, extending 284
 - disadvantages 291
 - door status, checking 281
 - function, declaring 280
 - OnCollisionEnter, working with 278
 - OpenDoor() method, writing 280
 - PlayerCollisions component, removing 301
 - procedure, reversing 287
 - script, testing 282

- scripting, for trigger collisions 300
- trigger zone, creating 298
- trigger zone, scaling 298
- triggering 298

- collisions
 - overview 269, 272
 - raycasting 273

- comment 119
- complete projects
 - about 628
 - references 628

- Component reference
 - URL 619

- conditions
 - about 120
 - for loops 120

- control tools
 - control bar 28, 29
 - Hand tool [Q] 27
 - Move tool [W] 27
 - React Transform tool [T] 28
 - Rotate tool [E] 27
 - Scale tool [R] 27
 - scene navigation 28
 - Transform tool [Y] 28
 - URL 28

- coroutines 127
- crepuscular sun rays
 - through Sun Shafts effect 542

- culling
 - about 553
 - camera culling distance 554
 - FireLight class, modifying 555

- custom functions
 - arguments 116
 - calling 116
 - return type 115
 - writing 115

D

- debugging 280
- deep optimization
 - URL 604
- deferred rendering
 - about 537

- using 539
- Depth of field (DOF)
 - debugging 542
 - URL 541
- desktop
 - publishing for 617
 - references 617
- digital content creation applications
 - about 259
 - Animation Compression 262
 - Animations 261
 - material 261
 - meshes 260
 - models, import settings 259
- Digital Content Creation tools
 - about 618
 - references 618
- Dot Syntax 122, 124

E

- edges 13
- Editor extensions
 - about 626
 - Asset Store from third parties 627
 - Asset Store from Unity Technology 627
 - built-in 626
- enemy AI
 - Advanced AI Controller class, creating 345
 - chase method, modifying 355
 - creating 341
 - custom AI state machine 347
 - custom components, modifying with
 - PropertyDrawers 359
 - Enemy's Field Of View 349
 - enhancements 343, 360
 - Nav Mesh Agent, debugging 357
 - player presence awareness 353
 - player, fighting 353
 - target, chasing 348
 - ThirdPersonCharacter class, duplicating 344
 - Unity Standard Assets, using 342
 - waypoints roaming 349
 - waypoints, roaming 348
- enemy
 - coroutines, using to time game elements 427,

- 430
- reviving 426
- Entity Component System (ECS)
 - about 618
 - URL 618
- environment ambiance
 - dusty village ground, creating 448, 450
 - enhancing 448

F

- Field Of View (FOV) 12, 467, 530
- Find() 123
- FindWithTag() 123
- Fire Light component
 - modifying 445, 447
- fireplace
 - creating 438, 440, 442
- force 400
- Forward Kinematics (FK) 201
- frame rate feedback
 - about 621, 623
 - video resolutions, testing 624
- Frames Per Second (FPS) 114
- FreeParallax 80, 90
- frustum culling
 - about 527
 - standard fog, versus Global Fog post effect 529
- Full Screen Anti-Aliasing (FSAA)
 - about 491
 - versus hardware-based anti-aliasing (MSAA) 541
- functions
 - about 113
 - OnMouseDown() 114
 - Update() method 114

G

- game assets
 - background layers, importing 77
 - background layers, placing 77
 - prefabs, pacing in game 76
 - setting up 73
- game development, elements
 - collision detection 258
 - raycasting 258
 - trigger collision detection 258
- game progression status HUD
 - arrays 382
 - background UI image, creating 378
 - displaying 376, 377
 - HUD panel, creating 378
 - HUD, disabling 386, 387
 - HUD, enabling 388, 389
 - order of elements, drawing 384, 386
 - QuestIndicator images, adding 383
 - settings, importing for UI images 377
 - UI image activation, scripting 382
- game title
 - adding 468, 470
- game, building
 - about 597
 - Build settings 597, 599
 - build, creating 600
 - quit button, with platform automation 599
 - standalone, building for Linux 600
 - standalone, building for Mac 600
 - standalone, building for PC 600
- GameObject
 - URL 57
- gameplay
 - designing 402
 - heavy stone prefab, creating 402
 - physics, adding to stone prefab 403
 - prefab, saving as 403
- General Public License (GPL) 80
- geometry 206
- GetComponent() 124
- global illumination
 - references 514
- GNU Image Manipulation Program (GIMP) 16
- GPU instancing
 - about 549, 552
 - URL 552
- Gradient Editor 452, 454
- Graphic Command buffers
 - about 547
 - URL 547
- Graphics Processing Unit (GPU) 515
- Graphics settings
 - about 591

tier settings 592
URL 591, 592

H

hardware-based anti-aliasing (MSAA)
 versus shader-based anti-aliasing(FSAA) 540
Heads Up Display (HUD) 12, 362
heightmaps 213
Heretics Island game
 asset package, importing 251, 252
 audio file formats 250
 audio, applying 249
 audio, settings 253
 colors, adding to trees 235, 238
 curves, setting up 253
 designing 207, 208, 209
 environmental sounds, using 250
 grass, adding 232
 Heretics Island game 254
 island outline, creating 225, 227
 island terrain, saving 256
 island, creating 224
 island, surrounding with sea water 242, 244,
 245, 246
 lake carving 227, 228
 lake, creating 247, 248
 Multipurpose Camera Rig, using 256
 painting procedure 230, 231
 positional 3D sound, enabling 253, 254
 positional audio, versus non-positional audio 249
 prefabs, using 255, 256
 procedural skybox, using 240, 242
 rock, adding 232
 sandy area, creating 231, 232
 scene illuminating 238
 sunlight, creating 239, 240
 Terrain Editor tool, using 210
 terrain, setting up 225
 textures, adding 229
 tree creator, using 233, 235
High Dynamic Range (HDR)
 about 482, 489, 546
 URL 546
hints, for player
 about 389

adjustments, for displaying progress 393, 394
screen, writing with UI Text 389
UI Text control, scripting for 390, 393
hut model
 adding 264
 audio, adding 269
 colliders, adding manually 266
 Physics Material 268
 positioning 265
 setting up 263

I

if else statements 118
image effects
 about 540
 crepuscular sun rays, through sun shafts effects
 542
 Depth of field (DOF) 541
 hardware-based anti-aliasing (MSAA), versus
 shader-based anti-aliasing(FSAA) 540
Inspector
 layers 107
 prefabs 108
 tags 105
 working with 104
instantiation
 implementing 397, 399
Integrated Development Environment (IDE) 25,
 344, 608
inter-script communication
 about 122
 comments 128
 coroutines 127
 Find() 123
 FindWithTag() 122
 GetComponent 124
 mobile, programming 125
 null reference exceptions 126
 objects, accessing 122
 SendMessage 123
interface
 about 26
 control tools 27
 Game view 33
 Hierarchy view 27

- Inspector 31
- Project window 32
- Scene view 27
- Internet Relay Chat (IRC) channel
 - URL 628
- Inverse Kinematics (IK) 150, 201
- iOS, Player settings
 - about 588
 - URL 589

L

- Layer Collision Matrix 140
- layers, Inspector 107
- level building 206
- Level Of Detail (LOD)
 - about 223, 546
 - URL 546
- light probes
 - URL 45
- lighting
 - about 511
 - baked global illumination 515
 - baked only, versus mixed lighting 514
 - Environment settings 512, 513
 - GameObjects, excluding from bake 519
 - global maps 523, 524
 - lightmapping, settings 520, 523
 - lightmaps 515
 - lights, excluding from bake 517
 - lights, including from bake 517
 - mixed lighting 514
 - object maps 525
 - preparing for 516
 - realtime lighting 514
 - realtime only, versus baked only 514
 - scene, baking 516, 519
 - scene, setting up 512
 - settings 525
 - URL 526
- lights
 - about 489
 - Rim light, adding 490
- local space
 - versus world space 9
- local variables 59

M

- main menu
 - audio options menu, creating 475
 - buttons, adding 473
 - clone, for obtaining Options menu 473
 - game title, adding 468, 470
 - panel, creating 472
 - scene, creating 466, 468
 - title, anchoring manually to Canvas 470, 472
 - UI buttons, configuring to show/hide menus with
 - OnClick(0 event method 474
 - video options menu, creating 479
- materials 15
- Mecanim
 - animation clips, importing 170
 - animations, setup for looping 171
- Mesh Colliders 13, 18
- Mesh Filter 44
- Mesh Renderer
 - about 45
 - anchor override 45
 - cast shadows 45
 - dynamic occluded 46
 - light probes 45
 - lightmap static 45
 - materials 46
 - motion vectors 45
 - prefab, creating 46
 - receive shadows 45
 - reflection probes 45
 - URL 46
- meshes 13
- MIT License
 - URL 80
- mixed reality (MR) 570
- mobile optimizations
 - URL 603
- mobile platform
 - adapting for 603
 - Android, adapting 604
 - Android, building 605
 - building for 603
 - iOS, building 608
 - preferred build system, selecting 606

- Publishing settings 607
- references, for iOS 609
- Texture Compression formats 604

mobile stores

- publishing on 617
- references, for platforms 617

models

- placing 302

MonoBehaviour

- future 618

MSDN C# reference

- URL 619

Multi Sample Anti-Aliasing (MSAA) 491

N

Navigation Mesh Agent (Nav Mesh Agents) 309

NavMesh (Navigation Mesh) 508

non-playing character (NPC)

- about 189, 207, 309, 389
- Animator Blend Tree, driving with scripting 335
- Animator Controller 314
- animator transitions 316, 319
- answer buttons, creating 340
- Canvas, creating 337
- creating 310, 314
- dialogue window, creating 340
- dialogue, triggering 325
- DialogueManager class, writing 329, 334
- interaction 324
- navigation setup 319
- NavMesh, creating 319
- rock stone, adding 320
- Simple AI class, writing 320
- sitSpot points, adding 320
- start dialogue prompt, creating 339
- startSpot points, adding 320
- UI events, using 334
- UI, creation for displaying dialog 336

non-positional audio

- example 250

null reference exceptions 126

Nvidia PhysX 399

O

Object Oriented Programming (OOP) 196

occlusion culling

- about 530
- Bake tab 531
- Object tab 531
- Visualization tab 533, 535

one-dimension (1D) 317

open beta 620

optimizations

- about 556
- AI impact, optimizing on CPU 564
- AI trigger areas, setting up 565, 567
- bottlenecks, spotting 625
- distance, drawing 625
- enhancements 567
- occlusion culling 625
- physics optimizations 557
- polygon counts 625
- scripting 625
- texture sizes 625

P

parallax scrolling

- Asset Store, using 80
- collectable items, spawning 90
- death zone, creating 88
- discrepancies, avoiding on ground collider 86
- elements, setting up 83
- foreground, putting 87
- FreeParallax component 82
- holes, avoiding on ground collider 86
- implementing 79
- middleground 85
- references 79
- special layers, putting 87

Particle System

- about 435, 437
- URL 436

performance

- frustum culling 527
- occlusion culling 530
- optimizing 527

Physics Material 22

- physics optimizations
 - about 557
 - DeactivateRagdollTimer class, writing 558, 560
 - HeavyStone class, modifying 561, 563
 - save game-status feature 563
 - URL 557
- physics
 - about 399
 - forces 400
 - Rigidbody component 400
- Player input settings
 - about 589
 - URL 591
- player inventory
 - about 372
 - adding, to player 374
 - artifact collected piece count value, saving 373
 - audio feedback 374
 - dialog access, restricting with piece counter 375
 - hidden piece spot access, restricting 375
 - PiecePickup() method, adding 374
 - variable start value, setting 373
- Player settings
 - about 572
 - Android 586, 588
 - background 581
 - configuration 584
 - cross-platform general settings 573
 - icon 577
 - iOS 589
 - logging 585
 - logos 580
 - optimization 584
 - per-platform player settings 574
 - presentation 575
 - rendering 582, 584
 - resolution 575
 - splash images 578
 - splash screen 579
 - standalone build 574
 - URL 577
 - user input bindings 576
 - WebGL 589
 - XR settings 585
- polygon count 15
- polygons 13
- positional audio
 - example 250
- Post Processing Stack
 - about 543
 - focus puller 545
 - Post Processing Stack V2 544
 - URL 544, 545
 - utilities 544
- post-processing image effects
 - about 491
 - bloom 492
 - chromatic aberration 492
 - FSAA (Full Screen Anti-Aliasing) 491
 - HDR 492
 - screen overlay 492
 - vignette 492
- prefabs
 - force, adding to Rigidbody 67
 - Instantiate(), using to spawn objects 66
 - projectile, firing 66
 - projectiles, clearing 69
 - saving 65
 - wall, resetting to initial state 69
- Primitive Collider 18
- private variables 59, 111
- projectile
 - creating 63
 - material, applying 63
 - material, creating 63
 - physics, adding with Rigidbody 64
 - prefab, creating 63
- PropertyDrawers
 - custom components, modifying 359
- public alpha testing 620
- public variables 111

Q

- Quality settings
 - about 592
 - other section 596
 - rendering 594
 - Shadows 595
 - URL 597

R

- Ragdoll physics simulation
 - about 423, 424
 - force impulse, adding to stone impacts 433
 - scripting 425
- raycasting
 - about 107, 269, 272, 273, 292, 295
 - code, refactoring 293
 - collider, resetting 297
 - collision detection, disabling 292
 - PlayerCollisions, tidying 294
 - predictive collision detection 274
- reflection probes
 - URL 45
- render
 - jail bars, adding to scene 494
 - splitting, of 3D and UI on different cameras 492, 494
- rendering features
 - about 545
 - Asynchronous Texture Upload 546
 - Graphic Command buffers 547
 - High Dynamic Range (HDR) 546
 - Level of Detail (LOD) 545
- rendering paths
 - about 535
 - deferred rendering 537
 - forward rendering 536
 - graphics pipelines 536
 - URL 536
- Rigidbody component
 - about 400
 - angular drag 401
 - constraints 402
 - drag 401
 - extrapolate 402
 - interpolate 402
 - Is Kinematic 401
 - mass 401
 - Use Gravity 401
- Rigidbody physics
 - about 17
 - collision detection 18
- Rigidbody

- URL 57
- Root Motion 151
- root-motion animations
 - about 189
 - cloth simulation 198
 - enhancing 198
 - Inverse Kinematics 201
 - transitions 192
 - transitions, setting between Grounded state and Airborne state 193
 - Unity standard assets classes, modification for importing playing character 196

S

- Scene view, control bar
 - create button 30
 - draw mode 29
 - gizmos 30
 - search box 30
 - toggle 2D 30
 - toggle scene lightning 30
- scene
 - setting up 73
- scripting reference
 - URL 619
- scripting
 - statements 109
 - URL 108, 128
 - variables 109
 - with Unity 108
- sea breeze particle system
 - creating 451
 - Gradient Editor 452, 454
- SendMessage 123
- shader 15
- Shader Language Model (SML) 491
- shadows
 - about 489
 - real-time shadows 490
- Shooter class
 - camera, moving 57
 - local variable 59
 - private variables 59
 - projectile, creating 63
 - public variables 59

- public variables, declaring 55
- scripts, assigning to objects 55
- Translate 60
- Translate, implementing 60
- writing 54
- skybox 240
- Softbody physics
 - about 19
 - Cloth component 19
- SpeedTree
 - URL 501, 504
 - using 500, 504
- Sphere Collider 18
- splat maps
 - about 212
 - path details, drawing 506, 508
- Sprites 20
- Standard Shader
 - about 46, 539
 - references 540
- state machine
 - about 179
 - character behavior, controlling 180
- StoneLauncher class
 - collisions, ignoring with layers 411, 413
 - collisions, safeguarding 410
 - component presence, checking 409
 - development safeguards, adding 409
 - Fight method, writing 406
 - heavy stone, instantiating 406
 - IgnoreCollision() method, using 410
 - implementing 413
 - instances, naming 408
 - object tidying 414
 - player input, checking for 405
 - stone launch, activating at animation frame 419, 420
 - stone throw, implementing 415, 419
 - stones, removing 421, 422
 - throw stones, scripting 404
 - velocity, assigning 408
 - writing 404
- StoneSpawner 138
- Sun Shafts effect
 - crepuscular sun rays 542

- URL 543
- supported platforms
 - about 570
 - Android platform 570
 - augmented reality 572
 - iOS 571
 - Linux 570
 - Mac standalone 570
 - mixed reality 572
 - virtual reality 572
 - WebGL 571
 - Windows 570

T

- tags 105
- Terrain Editor tool
 - 2D heightmaps, exporting 213
 - 2D heightmaps, importing 213
 - about 210, 213
 - detail prototypes 222
 - edit details 222
 - Edit Trees tool 220
 - features, setting 210
 - Flatten Heightmap tool 217
 - Paint Height tool 216
 - Paint Texture tool 218
 - Place Trees tool 220
 - Raise Height tool 215
 - resolution, setting 211, 212
 - setting area 223, 224
 - size detail, setting 211, 212
 - Smooth Height tool 217
 - Terrain component 213
 - tree prototypes tool 220
 - treas, mass placement 221
 - using 210
- terrain
 - about 206
 - guards, placing 509, 511
 - painting 505
 - path details, drawing with splat map 508
 - path details, drawing with splat maps 506
 - refining 504
 - smoothing 505
 - SpeedTree, using 500, 504

- tweaking 500
- Texture Compression formats 604
- textures 15
- Translate
 - about 60
 - game, testing 61
 - implementing 60
 - URL 60
- tree creator
 - about 233
 - reference link 233
- trigger collision detection
 - adding 369

U

- Ultimate Unwrap 3D
 - URL 150
- Unity audio
 - reference link 249
- Unity Editor
 - Frame Debug window 614
 - Physics Debug window 614
 - profiling with 611
 - testing with 611
 - Unity Profiler 612
- Unity Engine automated optimizations
 - about 547
 - culling 553
 - dynamic batching 547, 548
 - GPU instancing 549, 552
 - static batching 547
- Unity forum
 - URL 628
- Unity Legacy Animation System
 - about 151
 - animation, setting up 154
 - animations, importing 152
 - animations, importing with multiple model files 153
 - character models, importing 152
- Unity manual
 - URL 619
- Unity Navigation System
 - about 302
 - Agents tab 305
 - area mask 308
 - Area Types 307
 - Areas tab 306
 - Bake tab 303
 - Navigation cost 307
 - Object tab 305
 - URL 305
- Unity project
 - about 36
 - empty objects, duplicating with 48
 - empty objects, grouping with 48
 - light, adding 39
 - master brick, building 41
 - master brick, creating 41
 - prototyping environment 37
 - scene, setting 38
- Unity Standard Assets
 - using 342
- Unity UI
 - about 459
 - canvas render modes 460
 - conclusion 496
 - exploring 497
 - references 497
 - screen sizes, testing 496
 - textures, preparing for UI usage 464
- Unity water system
 - reference link 246
- Unity, concepts
 - about 20
 - assets 22
 - components 24
 - example 22
 - GameObjects 23
 - prefabs 25
 - scenes 23
 - scripts 24
- Unity
 - future 618
 - scripting with 108
- UnityScript 24
- User Interface (UI) 7, 323

V

variables

- about 109
- data types 110
- declaring 112
- private variables 111
- public variables 111
- using 110

vectors 11

vertices 13

video options menu

- creating 479, 480
- drop-down menu, creating 485
- dynamic variables, using 485
- game, loading 488
- slider controller 486, 487
- UI events, using 483, 485

virtual reality (VR) 570

Visual Studio 2017

- debugging 610, 611

Volumetric Lighting

URL 543

W

wall torches

- creating 442, 444
- Fire Light component, modifying 445, 447

Water4Simple 247

waterfall

- creating 455, 457

WebGL builds

- references, for work sharing 616
- work, sharing 616

WebGL, Player settings

- URL 589

WebGL

- Build Settings 602
- hardware configurations 601

What You See Is What You Get (WYSIWYG) 71

wheel

- reinvention, avoiding 626

world space

- versus local space 9