Community Experience Distilled

Unity iOS Essentials

Develop high performance, fun iOS games using Unity 3D

Robert Wiebe

[PACKT] PUBLISHING

Table of Contents

Unity iOS Essentials
Credits
About the Author
About the Reviewers
www.PacktPub.com
- Support files, eBooks, discount offers and more
  - Why Subscribe?
  - Free Access for Packt account holders

Preface
- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
  - Downloading the example code and colored images
  - Errata
  - Piracy
  - Questions

1. Planning Ahead for a Unity3D iOS Game
- iOS device differences
- Unity support for multiple iOS devices
  - Terrain and trees
  - Cut scenes
  - Audio
  - Lighting
  - Shadows
  - Modeling and Animation
  - Cloth
  - Code stripping
  - The iPhone classes and global enumerations
- Understanding iOS SDKs and target platforms
- Set up an iOS App for multiple target building
- Planning efficient levels
  - Well then, let's get started!
    - First consideration: Is my game 2D or 3D?
    - Second consideration: How will I structure my levels?
      - Is my game extensible?
      - A few pointers
      - How to set up unloading?
    - Third consideration: How can I make scenes realistic?
      - A few pointers
      - Easter Eggs
    - Fourth consideration: Embellishment
      - Fog: Not such a great idea
      - Particles make everything better
      - Atlas your textures: Share those materials

            - Water makes things look bigger
            - You can fly
        - Fifth consideration: Teleportation
            - The warp gate
            - The airship
            - The spell
            - The world map
            - The train
        - To summarize:
    - Culling is important
        - Near Clip
        - Far clip
    - Distance culling
    - Occlusion culling
- Summary

2. iOS Performance Guide

- Choose the right game
    - Stay grounded
    - Choose first person
    - Avoiding third person
    - Things to avoid
        - Open planes
        - Speed
        - Flight
        - Artificial Intelligence (AI)
    - Things to include
        - Skybox
        - Rolling our own terrain
        - Mountainous terrain
            - Dungeons
            - Secrets
        - Using urban terrain
        - Adding life
            - Secrets
        - Using suburban terrain
            - Adding life
            - Secrets
        - Using platform terrain
            - Adding life
    - Mobile game genres
        - Platformer basics
            - Pickups
            - Chests
            - Restorers
            - Mechanics
        - First Person Shooter basics
            - Items
            - Vehicles
        - Puzzle basics
            - Items
            - Mechanics

- Arcade basics
  - Mechanics
- Unified graphic architecture
  - Shared memory
  - Shared processing
  - Vertex Processing Unit (VPU)
  - Advanced RISC Machine (ARM) Thumb
- Dos and Don'ts
  - Programming the main loop
    - Model View Controller (MVC)
    - Springboard of death
    - Cache it: Awake() and Start()
    - Coroutines, not Update()
    - Time
    - Make it static, why instantiate?
    - Use hashtables
    - Triggers and collisions
    - OnBecameVisible()/OnBecameInvisible()
    - ApplicationWillTerminate() or ApplicationWillSuspend()
    - Strict compilation
    - Compilation order
  - Design to load additively
  - Artwork
    - Vertex count
    - Skinned meshes
    - Alpha testing
    - Lights
    - Post processing
  - Physics
    - FixedUpdate()
    - Collision detection
    - Deployment
- Culling is important
  - Frustum culling
    - Camera clipping planes
  - Camera layer culling
  - Occlusion culling
- Summary

3. Advanced Game Concepts
- Engaging menus
  - Mouse over
  - Mouse click
  - Screen size
  - Shake
  - Think outside of the computer
- Dealing with device screen resolutions
  - Convert to pixels in scripts
  - Scale bitmaps in scripts
- Testing iOS games in the editor
  - Simulating the accelerometer
- Using shaders to solve real-world problems

- Shading/Lighting models
  - Phong
  - Constrained (Blinn-Phong)
  - Flat (Gouraud)
- Applying shaders to solve problems
  - Z-Fighting
  - Back face rendering
- Organizing game objects for easy asset sharing
- Summary

4. Flyby Background
- Set up a background scene
- Set up the camera path
- Set up the main menu
- Testing the scene in the editor
  - Enable the Stats option
  - Choose Graphics Emulation
  - Click the Play button and repeat
- Setup for multiple iOS device deployment
  - Build Settings
  - Select iOS
  - Cross-Platform Settings
  - Resolution and Presentation
  - The Icon
  - The Splash Image (Pro feature)
  - Other Settings
    - Rendering
    - Identification
    - Configuration
    - Optimization
- Deploy the App to multiple devices
- Task: create the background scene
- Task: create a path to follow
- Task: putting it all together with the menu
- Challenge: flipping the iOS device
- Summary

5. Scalable Sliding GUIs
- Think about resolution not pixels
- Separating dialogs from contents
  - The buttons
  - The hook
- Dialog location, animation, and speed
  - Dialog location
  - Dialog sliding
  - Dialog fade
  - Dialog speed
- Task: script a scalable dialog
  - Set up the OptionsDialog
  - Set up the OptionsDialogContent and OptionsDialogControl
  - Set up the messages
- Task: script some dialog contents
- Challenge: making things flexible in the editor

- Challenge: creating custom GUI elements
- Summary

6. Preferences and Game State
- Introducing .NET
  - System.Environment
  - System.Collections
  - System.IO
  - System.xml
- Understanding property lists
- Handling different game players
- Deciding where to put files
- Task: load and save the players' preferences
  - Using the Globals.js script
  - Creating the defaultoptions.plist file
  - Provide a GUI dialog for the options
- Task: create a GUI to get the players' names
  - Add the player name variable to Globals.js
  - Create the new GUI Dialog game objects
  - Add the GUI dialog scripts
- Challenge: load and save the player's game state
- Summary

7. The Need for Speed
- Adding a four-wheeled vehicle
  - Unity3D Car Tutorial
  - JCar
    - A JCar vehicle model
    - Enabling JCar steering
    - Connecting a Joystick to JCar
- Managing physics
  - Large objects
  - Mid-sized objects
- Realistic and fast foliage
  - A typical tree
  - Foliage for iOS devices
    - Foliage that does not use transparency
    - Foliage that uses a single plane with transparency
    - Foliage that uses multiple planes with transparency
  - Foliage colliders
- Culling made easy
  - Distance culling
  - Occlusion culling
- Task: build your track
- Task: create a death zone
- Challenge: eliminate Z-Fighting
- Summary

8. Making it real
- Lighting and lightmapping the track
  - Preparing for lightmapping
    - Lightmap UVs
    - Static objects only
    - Triangulate objects before import

- Adjust ambient light
- Lighting the track
- Lightmapping the track
- Particles that pop
  - Renderer
    - Trail
    - Line
    - Particle
      - Particle system
        - Particle emitter
        - Particle animator
        - World Particle Collider
  - iOS recommendations
  - Creating particles in your particle system
  - Animating particles
    - Emission velocity versus added force
    - Color
    - Growth
    - Autodestruct
  - Particle example: Weather
- Shaders in the game world
  - Anatomy of shaders
    - Geometry and its properties
    - Other passed values
      - Declaring and referencing properties
      - Advanced lighting
    - Shading approximation
    - Depth
    - Platform capability and subshaders
    - Begin the pass
    - Alpha testing
      - Premultiplied alpha testing
    - To the screen (BLIT and Rasterization)
    - Fallback
  - Fun examples
    - Gems
    - Simple metal
- Water that works
- Task: The final track
- Task: Create a rich and engaging scene
- Summary

9. Putting it all together
- Ending or suspending the game under multitasking
  - OnApplicationPause
  - OnApplicationQuit
- Top scores board
  - Setting up the database
    - Create a MySQL database
      - Log in to phpMyAdmin
    - Create player and game achievement tables
  - Writing the server PHP scripts

    - Add a new player
    - Update the player's score
    - Retrieve the top score
- Task: Publish results online
- Task: Build and deploy the finished game to multiple iOS devices
- Summary

Unity iOS Essentials

Copyright © 2011 Packt Publishing

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

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

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

First published: December 2011

Production Reference: 1011211

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-182-6

www.packtpub.com

Cover Image by Parag Kadam (<paragvkadam@gmail.com>)

Credits

Author

Robert Wiebe

Reviewers

Fabio Ferrara

Karl Henkel

Marcin Kamiński

Clifford Peters

Alejandro Martínez Pomès

Acquisition Editors

Steven Wilding

Robin De Jongh

Development Editor

Susmita Panda

Technical Editors

Ankita Shashi

Llewellyn Rosario

Sakina Kaydawala

Copy Editor

Leonard D'Silva

Project Coordinator

Joel Goveya

Proofreader

Mario Cecere

Indexers

Monica Ajmera Mehta

Tejal Daruwale

Graphics

Valentina D'Silva

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Born in 1961, **Robert Wiebe** has more than 30 years experience in designing, implementing, and testing software. He wrote his first game in 1979, as a high school student, using the 6502 assembler code on an Ohio Scientific C2-4P computer with 8k RAM. More recently, he has focused on developing games and utilities for Mac OS X.

His interests include collecting vintage computers, which include many pre-IBM PC era microcomputers, Apple Macintosh computers starting with the SE/30, running Mac OS 7 through to the Macbook Pro running Mac OS X that he uses today, and both 2D and 3D game engines including Cocos2D, Torque, Cube 2, Dim 3, GameSalad, and Unity.

In addition to writing games, he has experience developing software in a number of industries, including mining, finance, and communications. He has worked for a number of employers, including Motorola as a Senior Systems Architect developing 2-way wireless data systems, and Infowave Software as the Software Development Manager for their Imaging Division. After working for other people's companies, he founded his own companies, Mireth Technology and Burningthumb Software, which are his primary interests today. This is his first book.

I would like to thank my son, Abram, who not only inspired me to pursue writing, but also has been instrumental in researching, reviewing, and revising the content of this book. I would also like to thank my wife, Donna, for not only encouraging me, but also for making it possible for me to pursue the things I want to do. And finally, I would like to thank my daughter, Amanda, who keeps me focused on the things that really matter in life.

About the Reviewers

Fabio Ferrara is a trainee at Kiwari, which is a company for software specializing in e-mail marketing. His tasks include application SQL, debugging web applications, and the creation of mini websites.

He is also a trainee at Shangrilabs S.r.l, Milan (entertainment company). His tasks include designing graphics and programming.

He is a registered student at Brera Academy, Milan. He is pursuing his first degree course in "New Technologies for Arts". He has passed all his exams for the first two years, with a grading average of 29/30. Now, he's starting with his third and final year.

He has a high school Diploma di Maturità Scientifica - from Liceo Luigi Cremona, Milan.

Karl Henkel is the founder of Parabox LLC, a company specializing in games and simulation development. Prior to forming Parabox, Karl studied Digital Media at Ohio University.

Marcin Kamiński is a programmer with 12 years experience. For the past eight years, he has been working in the games industry as an independent game developer, and is hired for outsourcing. His main fields of expertise are AI, network programming, debugging, and optimizing.

In 2005, with two friends, he created Mithril Games, a company that was creating audio games for blind people. In 2011, he founded the company, Digital Hussars, mainly to provide game programming services for others. The biggest Polish game developer and publisher, City Interactive, is using his services.

I would like to thank my wife and my daughter for supporting me in making my dreams come true.

Clifford Peters is currently a college student, pursuing a degree in Computer Science. He enjoys programming and has been doing so for the past four years. He enjoys using Unity and hopes to use it more in the future.

Clifford has also helped to review the books Unity Game Development Essentials, Unity 3D Game Development by Example Beginner's Guide, and Unity 3D Game Development Hotshot.

Alejandro Martinez Pomès is a Spanish developer, currently developing software for mobile devices. Now working on Sixtemia Mobile Studio.

He has participated in some game development, and is passionate about technology and the Internet. He has for some time written in his blog (www.elmundoexterior.es) and on his personal website (www.alejandromp.com), in addition to collaborating on other sites.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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

http://PacktLib.PacktPub.com

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

Why Subscribe?

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

Free Access for Packt account holders

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

Preface

Several years ago, before the iPhone existed, I began researching game engines. I looked at several engines, including the Sauerbraten Cube 2 engine, the Torque Game Engine, the Cocos2D engine, and others. I developed some game prototypes using each of these engines and found that all of them had issues that hindered game development. Then I tried a piece of software called Unity from a developer named Over the Edge Software, and found that, while it too had some limitations, it provided the most comprehensive set of features for independent game developers.

Since then, Unity has undergone many changes, as has the software industry, and my focus has changed from strictly desktop game development to also including mobile gaming. The Unity game engine has made that transition easy. The latest version of Unity for iOS makes developing games for both desktop and mobile platforms fun.

This book is intended to help anyone using Unity for the desktop, extend their game development target to also include mobile deployment on the iOS platform of devices.

What this book covers

Chapter 1, *Planning Ahead for a Unity3D iOS Game*, covers some important things that need to be done before starting game development. By completing these tasks prior to developing for the iOS platform, we can improve the final game result.

Chapter 2, *iOS Performance Guide*, explores the performance limitations of mobile devices in general, and the iOS family of devices specifically. It provides tips on how to get the best performance from the limited (compared to desktop computers) hardware in mobile devices.

Chapter 3, *Advanced Game Concepts*, explores some advanced iOS concepts that require us to change our approach from a traditional desktop gaming perspective. There are several key hardware and software differences in mobile gaming that we need to address.

Chapter 4, *Flyby Background*, moves from concepts to the hands-on work of building the initial scene for our mobile game, while taking the unique requirements of mobile platforms into account.

Chapter 5, *Scalable Sliding GUIs*, takes a closer look at the Unity Graphical User Interface (GUI) system and provides a hands-on approach to developing an advanced GUI framework that will work on any iOS device, regardless of the device's screen resolution.

Chapter 6, *Preferences and Game State*, introduces the .NET framework and how it can be used to manage player preferences and game state by creating property lists, and both writing and reading them from permanent storage in a platform-independent manner.

Chapter 7, *The Need for Speed*, looks at speed on mobile platforms. It introduces the concept of a four-wheeled vehicle that travels more quickly than a grounded player, and how working with physics and developing a scene is both engaging and efficient.

Chapter 8, *Making it real*, introduces the advanced features of Unity for iOS that we need to bring a static scene to life. It includes lighting and lightmapping, particle effects, shaders, and realistic water on a mobile device.

Chapter 9, *Putting it all together*, looks at how we can connect our game to the real world without tying it in to a third-party server. It walks through the steps needed to connect our mobile game to our own Internet services to record the players' high scores and achievements.

What you need for this book

This book requires Unity3D v3.4.2 or later. Some of the concepts (like Beast Lighmapping) require Unity Pro, but it is not essential to the book. In addition to Unity, we need a 3D modeling tool like Blender or Cheetah3D and a bitmap editing tool like Gimp or Photoshop.

Much of our emphasis has been on cross-platform development, so the majority of the scripts work on Mac and Windows, but because iOS devices require Apple's XCode to build programs, we need a Mac OS X development system to explore many of the concepts in this book.

Who this book is for

This book is for people who want to plan, develop, and deploy Unity 3D games on iOS mobile platforms, including the iPhone, iPod Touch, and iPad. Anyone who has experience with the free desktop version of Unity 3D can pick up this book and learn how to take the desktop skills and optimize them to work on the mobile iOS platforms. Some of the features in this book discuss the Pro features of Unity 3D for iOS, so a Pro license is required to use some of the features (notably, Occlusion Culling and Beast Light mapping).

Conventions

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

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
//This variable controls the
//execution of the coroutine loop
//When true the loop executes
//When false the loop does not execute and
//the coroutine exits
var go : Boolean = true;
function Awake()
{
// Start a coroutine
MyCoroutine();
//Do some more things
//…
// The end of Awake
Debug.Log("Awake is finished");
}
function MyCoroutine()
{
var myVar : int = 0;
//Do this important stuff
//every 5 seconds
while(true == go)
{
//Execute my code
Debug.Log("The count is: " + myVar);
myVar = myVar + 1;
// Wait for 5 seconds
yield WaitForSeconds(5);
}
}
//Execute this function to prevent
//the coroutine from running and cause
//it to exit
function ToggleFunction
{
go = !go;
}
```

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

```
//This variable controls the
//execution of the coroutine loop
//When true the loop executes
//When false the loop does not execute and
```

```
//the coroutine exits
var go : Boolean = true;
function Awake()
{
// Start a coroutine
MyCoroutine();
//Do some more things
//…
// The end of Awake
Debug.Log("Awake is finished");
}
function MyCoroutine()
{
var myVar : int = 0;
//Do this important stuff
//every 5 seconds
while(true == go)
{
//Execute my code
Debug.Log("The count is: " + myVar);
myVar = myVar + 1;
// Wait for 5 seconds
yield WaitForSeconds(5);
}
}
//Execute this function to prevent
//the coroutine from running and cause
//it to exit
function ToggleFunction
{
go = !go;
}
```

Any command-line input or output is written as follows:

```
NSMutableDictionary *myDictionary = [[NSDictionary alloc] init];
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

> **Note**
>
> Warnings or important notes appear in a box like this.

> **Note**
>
> Tips and tricks appear like this.

Reader feedback

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

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

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail< suggest@packtpub.com>.

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

Customer support

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

Downloading the example code and colored images

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

Errata

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

Piracy

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

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

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

Questions

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

Chapter 1. Planning Ahead for a Unity3D iOS Game

Before we start to develop our game, there is some planning that needs to be done. We want to do this planning, because it's a lot easier to plan first and develop second than it is to rework and rework again as we develop our game.

> ### Tip
>
> This book has been tested with Unity3D version 3.4.

A good example of this is the relative sizes of our 3D objects. It is really important to scale objects to the correct size. If we simply started developing a game and making 3D game assets without scaling them correctly, we may later find out that our physics are not working as expected. This is because the physics engine expects objects to be of the correct size. To get the physics to work correctly, we would need to rescale every object in our game. It brings a whole new meaning to the phrase "measure twice, cut (model in this case) once". Using Unity3D's default unit scale, the PhysX engine used by Unity3D expects one unit in your model to represent one meter.

This chapter will walk through the important things we need to consider when planning our game, so that we get them right the first time and save ourselves a lot of time and trouble on the backend of releasing our game.

In this chapter, we will learn the following:

- The differences between the capabilities of the various iOS devices (iPhone, iPhone with Retina Display, iPod Touch, iPad)
- Desktop functionality that is not supported on iOS devices and how to work around the limitations it imposes
- Which iOS Software Development Kit (SDK) to use, regardless of the iOS version being targeted
- How to set up Unity3D to target multiple iOS devices with a single application (universal binary for iOS)
- How to plan game levels that are efficient on iOS devices

iOS device differences

Before developing our game on different iOS devices, it is important that we understand the different capabilities of the devices. For the purposes of this book, we recommend using the latest iOS device available. We may choose to use more than one iOS target platforms. However, it should be noted that our game may need to configure itself at runtime to facilitate the capabilities of the individual devices.

The following is a brief overview of the capabilities of each device. The columns indicate capabilities, while the rows indicate the specific device:

Device	Graphics* Normal					
Vertex-Lit	**Diffuse**	**Decal**	**Detail**	**NormalMap**	**Specular**	
iPhone	1	1.5	1.5	2.5	2	2
iPod Touch	1	1.5	1.5	2.5	2	2
iPad	1	1	1.5	2	2	2

Device	Graphics* Complex			
Parallax	**NormalMap Parallax**	**Reflective**	**Other**	
iPhone	2.5	3	2.5	2.5
iPod Touch	2.5	3	2.5	2.5
iPad	2.5	3	2.5	2

Note

Graphics* capability varies by the device's iOS generation; we recommend deploying it for the latest iOS devices only if we believe your game will be graphic intensive. The ratings, given previously for shaders, are based on the assumption that you are using the latest software.

If we consider shader support (note that Unity3D does include a number of shaders optimized for performance on mobile devices), the numbers in each box indicate the degree of support on each device, where the following numbers have the associated meanings:

- 0: Not supported
- 1: Supported
- 2: Use sparingly

- 3: Supported but not recommended. This means that the shader will work, but the performance cost for using it is high and so it should be used only if there is no better option available.

If we plan to use transparency, then we need to add 0.5 (which is the rating in the previous table).

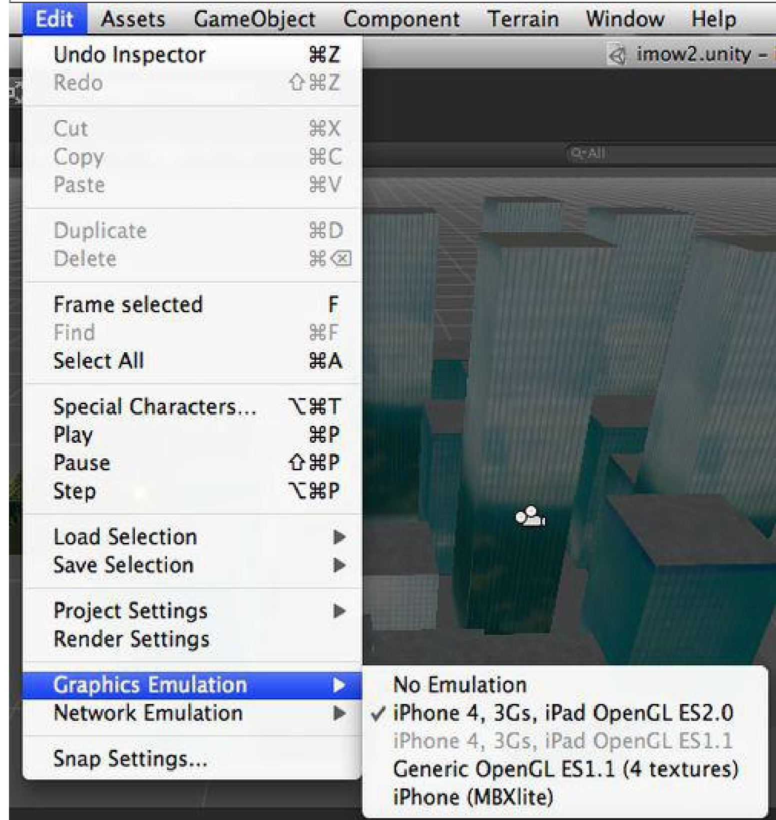
Graphics are by far one of the most important considerations, because they will, most likely, be the primary limitation that we will encounter during development.

It is understood that the shaders discussed here are using OpenGL ES2.x. Devices or deployment settings that use OpenGL ES1.x will not support certain shaders, such as, those that use cube-maps or normal-maps. To test shader support in the editor switch, we can switch the graphics emulation under the **Edit** | **Graphics Emulation** menu, as shown in the following screenshot:

Tip

Graphics Emulation menu

The contents of the **Graphics Emulation** menu will change based on the platform build settings that you have selected. Because we have selected iOS as our build platform, we see only the iOS-specific **Graphics Emulation** options.



We also need to consider the technical specifications shown in the following table when we are:

- Designing the graphics and levels for our game
- Deciding which devices we want to target for our game deployment

Device	**Technical Specifications Base**						
CPUMHz	**GPU**	**Memory**	**Screen**	**GPS**	**Mic**	**OpenGL ES2.0**	
iPhone 4	800	See CPU	512 MB	326 ppi	YES	YES	YES
iPhone 3GS	600	SGX	256 MB	163 ppi	YES	YES	Emulated
iPhone 3G	412	MBX	128 MB	163 ppi	YES	YES	NO
iPod Touch	533	MBX	128 MB	163 ppi	Emulated	NO	NO
iPad	1k	See CPU	256 MB	132 ppi	Emulated	YES	YES
iPad 2	1k Dual	See CPU	512 MB	132 ppi	YES	YES	YES

Device	**Technical Specifications Auxiliary**				
Vibration	**Compass**	**3G**	**Silent Switch**	**Lock Orientation**	
iPhone 4	YES	YES	YES	YES	NO
iPhone 3GS	YES	NO	YES	YES	NO
iPhone 3G	YES	NO	YES	YES	NO
iPod Touch	NO	NO	NO	NO	NO
iPad	NO	NO	NO	NO	NO
iPad 2	NO	YES	YES	YES	YES

You can find detailed information on all this at http://unity3d.com/support/documentation/Manual/iphone-Hardware.html.

Unity support for multiple iOS devices

There are several Unity3D functions found in desktop and console games that are not, for performance reasons, supported on iOS platforms. This section will outline the functionality that is unsupported and, where possible, the ways that we can work around these platform limitations.

Terrain and trees

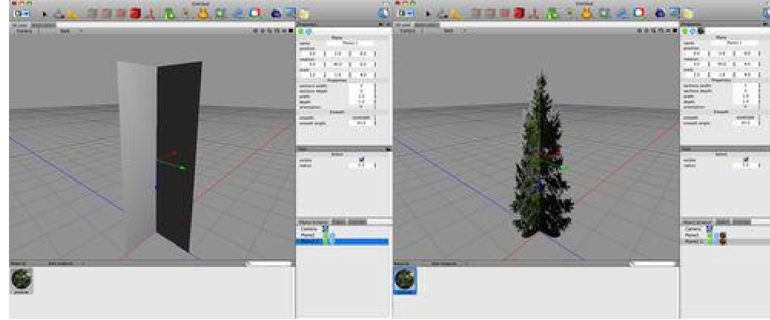
The Unity3D terrain engine is designed to work with platforms that can push high numbers of vertices. Soft bodies that would be used to simulate wind blowing in the trees and waving grass are available, but not recommended on the current generation iOS platforms with the exception of the iPad 2 (and later).

We can still create our own terrain using a 3D modeling program. If the 3D modeling program can programmatically create terrain, we need to make sure that it uses a relatively low number of polygons. If the 3D modeling program does not allow us to control the number of polygons, we can create the terrain object and then remove edges to reduce the triangle count.

An easy way to create a low polygon count terrain is to do the following:

1. Draw an aerial view of the level
2. Map the drawing to a plane
3. Cut the major shapes out of the plane
4. Extrude the major shapes

Trees can be modeled quite easily in a cartoon or semi-cartoon style. If more realistic trees are desired, then we can use two intersecting (non-backface-culled) planes with an image mapped onto the faces, as shown in the following screenshot:



Tree and grass movements can be simulated on iOS using **baked** animation. Methods for creating foliage will be covered in more detail later.

It may sound easy, but we don't jump into making our level just yet; first, we need to review the information contained in the section, *Planning efficient levels*.

Cut scenes

On the desktop PC platforms, Unity3D usually handles cutscenes through the use of a camera, facing a movie texture. However, movie textures are not supported on iOS devices. Instead, fullscreen streaming playback is provided. Unfortunately, the streaming playback suffers from a number of limitations, the worst of which does not support device orientation, so your cutscene may begin playing upside down.

This is clearly one area where Unity3D for iOS requires more work on the part of the Unity3D development team.

A potential alternative to using a movie for cutscenes is to use **Machinima**, which refers to the queuing of animations with the engine, rather than playing a pre-rendered clip. This is definitely not something for the faint of heart, as an effort was started in 2009 as a **Summer of Code** project to create a Machinima editor for Unity3D, but that project has definitely been stalled and may never be completed.

At this stage of Unity3D for iOS, we recommend using cutscenes only in games, where the addition of the cutscenes outweighs the cost of creating them.

Audio

iOS devices can play back only one compressed audio clip at a time. This means that if you want to play more than one sound simultaneously, you need to decide which of the clips should be uncompressed.

Given the limited memory available to games on iOS, it is likely that large uncompressed audio clips will cause our game to get memory warnings and even be forced to terminate by iOS. This kind of application crash can be very difficult to track down, so it is better to do some judicious planning and avoid it altogether.

Therefore, it is recommended that the compressed stream be used for the longest audio clip in a scene, such as background music, and all other audio clips be kept to the shortest duration possible.

The exception to this rule would be when doing Machinima (see cutscenes), where the entire sequence of audio will be consolidated into a single compressed track, where all of the events are queued to match.

Lighting

iOS devices only support the forward rendering path. The Unity3D manual describes this path as follows:

> Forward rendering
>
> "Forward is a shader-based rendering path. It supports per-pixel lighting (including normal maps and light Cookies) and real-time shadows from one directional light. In the default settings, a small number of the brightest lights are rendered in per-pixel lighting mode. The rest of the lights are calculated at object vertices."

As such, lights are relatively expensive to render.

Note

In scenes with several lights, some lights will only be calculated at object vertices. Vertex lighting, combined with the low polygon nature of models on iOS devices, can leave our game with light patches in areas rather than fully lit areas.

For dynamic lighting, in the order of rendering cost from least to most expensive, we have the following:

- Directional: This is a type of lighting that is indefinitely far away and affects everything in the scene, such as sunlight
- Point: This is a type of lighting that affects everything within the range of its position in the scene
- Spot: This is a type of lighting that affects everything in a cone shape, defined by its angle and range from its position in the scene

In terms of shader lighting, in order of rendering cost, from least to most expensive, we have the following:

- Un-lit: This type of lighting is not applied, and the object's illumination depends only on the textures applied to it
- Vertex-lit (generally but not always): This type of lighting for all lights is calculated based on the vertices and applied once, so the objects are only drawn one time
- Pixel-lit: This type of lighting is calculated as each pixel is drawn and needs to be recalculated for each light in the scene that affects the pixel. As a result, the object is drawn multiple times, once for each light in the scene that affects it

Lights can be configured as either **lightmapped-only** or as **realtime-only** with a configured degree of importance.

Configuring lights to lightmapped-only will cause them to appear only in the lightmap bitmap, and it will not have any dynamic lighting. This kind of lighting is appropriate where there is a requirement for an area to be lit, but not dynamically, such as those instances where a single dynamic light is the focus, or if we will be baking a scenery in the background (this methodology will be explained later).

Tip

It is also important to note that lightmapped lights will not display light cookies. Because of that limitation of lightmapping, lights with cookies should be set to realtime-only with a degree of importance set to important or automatic to achieve the desired effect.

Configuring lights to realtime-only is useful if you want to apply lighting to an object that will always be moving around. Moving objects should not be included in the lightmap, and they will never be marked as static.

Realtime lights can be assigned a degree of importance. They are as follows:

- Automatic (the default)
- Important
- Unimportant

Important lights will always be rendered when visible. Unimportant lights are given lowest priority for rendering and may not be rendered. Automatic lights will also not always render, but are given higher priority than unimportant lights.

Rather than using dynamic lighting, lightmapping should be used, when possible, to render much higher quality effects with less bandwidth.

Shadows

On desktop platforms, shadows would also be an important consideration, but on mobile platforms, dynamic shadows are supported. On mobile platforms, you will need to use lightmap textures.

Modeling and Animation

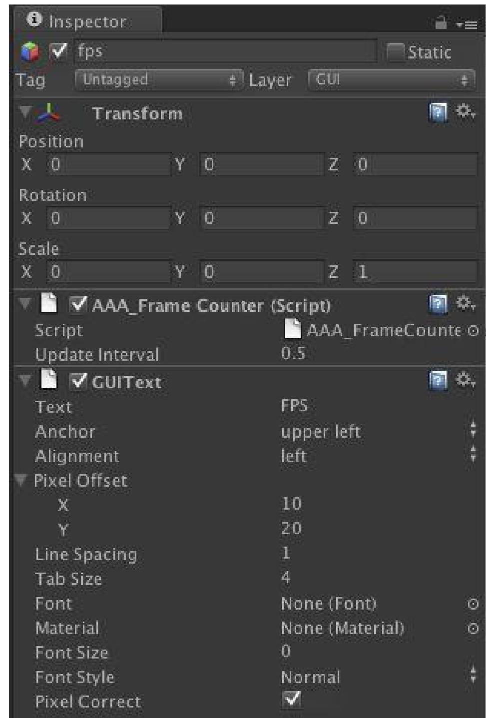
Depending on their age, iOS devices can push between 10k and 30k vertices. The maximum number of vertices that the original iPhone can push is 10k, and the iPhone 4 can push 30k vertices. In addition, devices will slow down depending on the number of polygons that they need to render. Trying to push too many vertices or render too many polygons will result in degraded frame rates and choppy gameplay.

The `FrameCounter, js` script, shown as follows, can be attached to a `game` object that contains a `guiText` item to display a `Frames Per Second` (FPS) counter that can be used to gauge the performance of our game. Typically, gameplay is smooth until the `FPS` counter falls below 20. This is shown in the following code:

```
// Attach this to a GUIText to make a frames/second indicator.
//
// It calculates frames/second over each updateInterval,
// so the display does not keep changing wildly.
//
// It is also fairly accurate at very low FPS counts (<10).
// We do this not by simply counting frames per interval, but
// by accumulating FPS for each frame. This way we end up with
// correct overall FPS even if the interval renders something
// like 5.5 frames.
//
// You need to create a game object, add a GUIText to it
// and set the Pixel Offset so that the text appears on
// the screen
// This is the frequency at which the FPS are displayed
varupdateInterval = 0.5;
// This is the number of FPS accumulated over the interval
privatevaraccum : float = 0.0;
// This is the number of frames rendered over the interval
privatevar frames : int = 0;
// This is the time left time for current interval
privatevartimeleft : float;
function Start()
{
// Make sure a guiText exists on the game object
if(!guiText)
{
print ("FrameCounter needs a GUIText component!");
enabled = false;
return;
}
// Set the time remaining equal to the interval time
timeleft = updateInterval;
}
function Update()
{
// Subtract deltaTime from the time left
timeleft = timeleft - Time.deltaTime;
// Accumulate the FPS over the interval
accum = accum + Time.timeScale/Time.deltaTime;
// Add one to the number of frames rendered
frames = frames + 1;
```

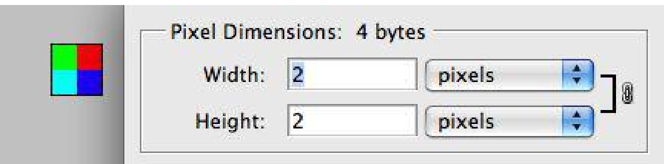
```
// The interval ended, display the FPS and reset the
// counters
if( timeleft<= 0.0 )
{
guiText.text = (accum/frames).ToString("f2");
timeleft = updateInterval;
accum = 0.0;
frames = 0;
}
}
```

The following screenshots show how the game object will appear after you have added the **Frame Counter (Script)** in the Unity3D editor and what it would look like in a running game:

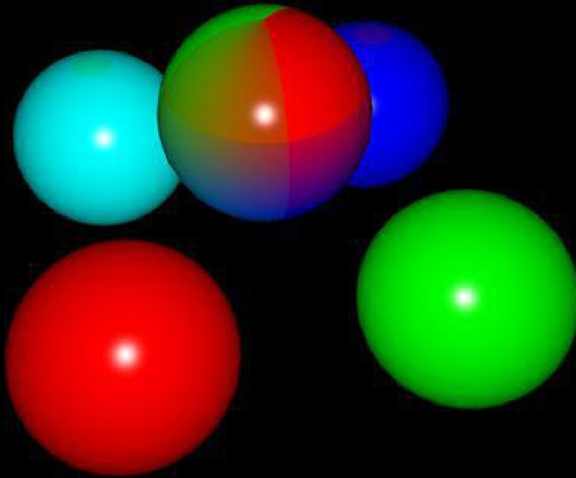


Characters animated using bones (skinned meshes) should have between 1 and 30 bones. A good compromise is to let minor characters have 10 to 15 bones, while major characters (such as the player character) have 20 to 25 bones. If only one skinned mesh is used, Unity3D has optimizations related to visibility culling and bounding volume updating. These optimizations are useful for games, where only one skinned mesh is needed for a player character (which is often the case).

Rendering performance can be optimized by atlasing textures that will commonly be rendered together. The following screenshot shows the image settings for a very small texture:



The small texture is used on all the spheres, as shown in the following screenshot. The spheres use different meshes with **UVW-maps** that project different sections of the same texture onto each sphere. In this case, the mapped texture would be referred to as the **texture atlas:**



This only works with static geometry or objects that are of the same mesh. Also, as previously mentioned in the section, *Lighting*, for pixel-lit models, if there are multiple lights illuminating the object, the number of draw calls required to render the objects will increase.

Cloth

Cloth is unsupported on iOS devices, it can, however, be simulated through a **baked** animation. For example, using a reactor in 3D Studio Max will allow you to make a flowing cape for your character, but this cape will not collide with Unity3D rigid-bodies.

Using cloth on unsupported platforms is usually more trouble than it's worth the time spent figuring it out and adds only a marginal visual effect.

Having said that, one practical time to use a cloth animation is while making a flag blow in the wind, in an area unreachable by the player.

Code stripping

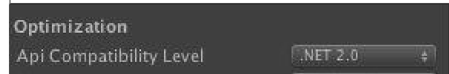
Unity3D can use code stripping to remove unwanted libraries from the engine. However, code stripping should be used with care. Stripping code may break your code, if you use certain functionality. Unity3D is not particularly careful when it comes to stripping, and it is assumed that you know what you are doing. If you set your stripping level too high, your game may simply crash when you run it on the iOS device.

The purpose of code stripping is to reduce the size of the application generated by Unity3D. However, using code stripping may limit your ability to use the .NET framework.

Because the built-in Unity3D support for writing to disk is limited, we make extensive use of the .NET framework for reading and writing game preferences. Therefore, we limit stripping code to strip byte code. If we re-implemented our preferences in a different manner that did not use .NET, then we could take advantage of additional stripping.

Note

In the player settings, found under **Edit** | **Project Settings** | **Player**, under **Optimization, API Compatibility Level**, we need to select **.NET 2.0**, rather than the **.Net 2.0 Subset**. Failure to do so will result in .NET issues that manifest themselves as a crash, when we deploy our game to an iOS device.



The iPhone classes and global enumerations

For legacy reasons, the classes and enumerations to access or describe specific iOS functionality begin with the prefix iPhone. It would be reasonable to expect that a future release of Unity3D will deprecate these items and replace them with iOS equivalents.

These classes and global variables are quite useful for determining which specific functionality is available on the device on which your program is running.

Class or Variable	Description
iPhoneGeneration	A global variable that lets you determine the generation of the iOS device on which your application is running.
iPhoneInput	Use location services to provide the last measured device's geographical location. All other forms of input use the more generic "Input" class.
iPhoneKeyboard	Interface for the software keyboard. We need to test to make sure that our GUI is positioned correctly, so that it is not covered by the keyboard when it is displayed.
iPhoneMovieControlMode	An enumeration used to describe options for movie playback controls.
iPhoneMovieScalingMode	An enumeration used to describe scaling modes for displaying movies.
iPhoneNetworkReachability	An enumeration used to describe network reachability options.
iPhoneSettings	Interface access to iPhone-specific setting information.
iPhoneUtils	Interface to access functions that play movies, vibrate the device, and check if our binary is genuine.

Understanding iOS SDKs and target platforms

When we install an iOS SDK from Apple, it's a unique experience. Apple has decided that they only want us to have the latest SDK on the most recent version of Mac OS X. This is far from the typical experience you may have had with other platforms, or multiple SDKs being installed simultaneously is the norm.

It is possible, and supported by Apple, to have multiple SDKs, but it is not very practical. For example, we could do the following:

1. Use System Update to install the latest version of Mac OS X.

   > **Note**
   >
   > If we skip the step of installing the latest version of Mac OS X, it is more than likely that the iOS SDK installer will refuse to install the latest SDK until we have installed the latest version of Mac OS X, so we'll just make that assumption and do it.

2. Download the latest iOS SDK.
3. Install the latest iOS SDK.
4. Notice that the installer removed the previous iOS SDK.
5. Use out Time Machine backup to restore the previous SDK.

But that is not what Apple want us to do, and really, it's not what we want to do either. There may be some exceptions, but for the most part, we want to jump on the Apple bandwagon and recognize that deploying our game using the latest and greatest SDK is indeed the correct thing to do.

So, what we want to do is this:

1. Use System Update to install the latest version of Mac OS X.
2. Download the latest iOS SDK.
3. Install the latest iOS SDK.
4. Notice that the installer removed the previous iOS SDK.
5. Don't worry about it; after all, that is what we wanted it to do.

The reason for this cavalier attitude is that the version of iOS on which your game runs has little to do with the iOS SDK that you are using. The version of the iOS on which your game runs is called the **Target Platform**, and it's perfectly acceptable to have a target platform of iOS 3.0 with the iOS 4.3 SDK.

It took the folks at Unity3D a while to catch on to, and fully support, this idea. Prior to Unity3D, you would need to go into your project settings after each iOS SDK install and update them to reflect the latest version of the SDK (or use Time Machine to restore the removed previous SDK).

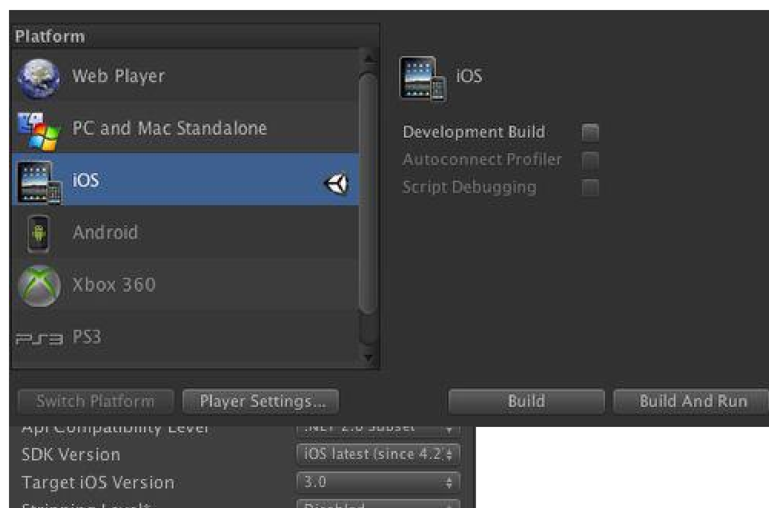
Thankfully, those days are gone. Now, setting up our Unity3D project so that it automatically uses the

latest SDK, which for the most part is what we want to do, has been greatly simplified.

What we need to do in Unity3D 3.x is this:

1. Open our Unity3D project.
2. From the **File** menu, choose **Build Settings**.
3. Under **Platform**, select **iOS**.
4. If required, click the **Switch Platform** button and wait for our assets to be re-imported.
5. Click the **Player Settings** button.
6. In the **Inspector** panel, click **Other Settings**.
7. Under **Optimization:SDK Version**, choose **iOS latest (since 4.2)**, where 4.2 is an iOS SDK version number.
8. Under **Optimization:Target iOS Version**, choose the version of iOS, upon which we wish our game to run.

And that is all there is to it. Now, when we install the latest iOS SDK version and re-build our project, the generated XCODE project should be correctly configured and built. It doesn't get any easier and it looks like the following screenshots:



Set up an iOS App for multiple target building

When the first version of Unity3D was released for the iPhone, it was called Unity iPhone. Though the first version of Unity iPhone was released a few years ago, the technology used in mobile devices has changed dramatically since then.

It should be clear that Apple renamed the iPhone OS to iOS for a reason. And it should be clear that when we are developing a Unity3D game for iOS, we want to target that game to a large audience. The way we do that is by targeting our Unity3D game to build for both the iPhone and the iPad as well as for the different versions of the ARM CPU used in those devices.

If there is one area where we may want to limit, rather than expand, the size of the market for our game, it is by selecting the latest version of OpenGL rather than limiting ourselves, and our game, to the limits of previous OpenGL versions.

Some developers prefer to take it one step further and limit the target market for their game to the latest and greatest iPhone and iPad platform. And that is something well worth considering, since building a game that runs on older hardware will not only limit what we can do in our game, but also dramatically limit the performance of our game.

Regardless of the generation of device that we will target though, we will always want to target our game for both the iPhone (and by extension, the iPod Touch) and the iPad.

Setting up Unity3D to build our game for these two (three if you include the Touch as a separate platform) is easy. These are the steps that we need to perform. The first step is to set up the Splash Images as follows:

1. From the **File** menu, choose **Build Settings**.
2. Make sure iOS is selected as the target platform.
3. Click the **Player Settings** button.
4. In the **Splash Image** section, as shown in the following screenshot, assign the Texture2D images to be used for each target platform.

The following screenshot shows our Splash Image settings in the Unity3D editor:

The second step is to set up the Target information:

1. In the **Other Settings** section, choose **Target Device** and click the **iPhone + iPad** button.
2. In the **Other Settings** section, choose **Target Platform** and click **Universal arm6+arm7 (Open GL ES 1.1+2.0)**.
3. In the **Other Settings** section, choose **Target Resolution** and click **Native (default device resolution)**.

The following screenshot shows our Target settings in the Unity3D editor:

Planning efficient levels

The virtual world hierarchy can be imagined using several key words. You may have noticed that these words have been used at earlier points in the book without rigorous definitions. However, they do have specific and precise meanings, which we will be using from this point onwards. We need to take some time to review and understand the following important terminology:

Key Word	Definition
World	The root of the game hierarchy, which refers to the entire virtual reality that we will be designing.
Level	A level is an assembly of different scenes to form a larger structure. An example of changing levels would be going from one virtual country to another, where reaching another country cannot be achieved instantaneously and would typically involve travel by land, sea, or air. Because of the potential load times between levels (if we are very good, our game should have no load time) these transitions are usually facilitated by a transition level (like riding the train) or a cutscene.
Mode	Levels may be reused as the setting for multiple events. Certain areas of a level can be made inaccessible, or accessible, by events in the progression of the story. One minute a level could be a peaceful town and the next moment it could be the site of a final battle. Mode refers to the use of the same setting for a different purpose in the progression of the story.
Scene	A scene is defined using the Unity3D definition of a scene. Levels are assembled by loading one scene and then, optionally, additively loading a number of additional scenes.
Section	Sections are usually differentiated by how terrain is organized or split. Sections can be easily confused with scenes, if each separate scene contains only one terrain section. Sections do not possess the same capabilities as scenes, because separate scenes can contain separate lightmapping and occlusion data.
Area	Sections can be further divided into smaller parts, which would be beneficial for occlusion culling.
Room	If you don't want to deal with large expanses, levels or even the world can be room-based. Rooms are like sections separated by doors. If the next scene has not yet loaded, the transition door remains locked or takes longer to open.

Well then, let's get started!

You can't have a game without at least one scene, even if you plan to make a very simple game such as a board game. Levels and their respective scenes should be planned ahead of time; nothing is more annoying than trying to retrofit sections into scenes.

First consideration: Is my game 2D or 3D?

Will our game be occurring in all three dimensions? Yes, of course it will, we're using a 3D game engine, but how will the player see it? If we lock the player in the X-Y axis, we can easily make a side scroller, and even add depth when we need it to spice things up, because our geometry is in a 3D space. We add the depth by changing the camera view from 2D to 3D, when we want to show 3D depth in the spiced up parts of our 2D game. The same goes for an overhead view (the X-Z axis), we can see our world from the sky, but if something explodes, the flames can fly towards us.

If we are locking the X-Z axis, not much can be done to improve the performance of our game, except reducing the field of view and far clip (after all, if we're always looking down, there is no reason to keep rendering after we hit the bottom of the level).

Side scrollers are more straightforward to implement, while some optimization may be required (for example, keeping the use of textures reasonable), typically, there is always plenty of time and processing power to load the next scene or section. We can spend less of our time optimizing the game, and more of our time developing content.

If we create a full 3D game in the first or third person view, we will need to think about a lot more in terms of what to load when, how to ensure the player passes the load trigger, how to store the state of objects in the scene, and much more.

Users have come to appreciate environments that are both large and seamless. However, we shouldn't be afraid to put in some load screens if we don't want to think about such things; this may drastically reduce our development time. We need to try to keep loads short and/or infrequent to optimize the player's experience.

Second consideration: How will I structure my levels?

Levels for iOS devices should be planned in such a way that they can be split up and then additively loaded using the `Application.LoadLevelAdditiveAsync ("sceneName")` function.

This stage will take the longest period of time to get through, because if careful consideration is not taken to figure out where scene transitions will occur, terrain geometry will have to be restructured and re-UVW-mapped in order to accommodate new additions.

Is my game extensible?

Do we intend to add more to our game in later updates? Do we want to sell the user more levels? If our game is to appear seamless, we need to leave space for a few *locked doors* (they may take the

form of any barricade).

> ## Note
>
> Apple prefers developers to include all their content in a single Application Bundle and upload it in its entirety, rather than provide downloadable content. If we include all of our assets in the bundle, we can unlock the bundled content, rather than download additional content.

A few pointers

We need to place trigger areas that are used to load and unload scenes at strategic locations.

If we cannot load a scene quickly enough, we need to consider placing a large *airlock* in the way, the player will be temporarily stuck while they wait for the next door to open, which we will do only once the scene loading is complete.

When breaking levels into pieces, unless we have tools to create global UV maps, the transition between sections will be obvious. We can use clever overlapping of door frames, rock formations, foliage, sidewalks, or doors to mask the texturing error between the separate meshes.

We can make our levels have long curvy areas and bottlenecks when we want to load a big scene or for any transition area.

How to set up unloading?

When setting up a scene, all the contents of that scene should be placed inside a single game object with a name identical to that of the scene.

Using a script, we can then use trigger areas (using the string, that is, its name) to both load the scene when we need it and then destroy it (by finding the `game` object with the name equivalent to that string), when it is no longer needed.

Scene splitting is required to properly load and unload while maintaining occlusion culling and lightmap date. If we do not use occlusion culling or lightmapping, we may simply load prefabs into the scene at their required positions.

Third consideration: How can I make scenes realistic?

If our levels appear linear, the player will get bored easily. Since iOS devices are limited in their capacity, there are a few ways in which we can make the scenes appear more realistic.

A few pointers

Carefully plan out scenes on paper before implementing them in Unity3D or designing them in a 3D modeling application. We can also effectively use the borders of a sheet of graph paper to indicate

a transition zone or the impassible edge of a scene that we implement as a large cliff, pit, or body of water.

A world looks bigger and more believable if we put islands in our oceans. Bay and cove sections appear larger, even if most of the section is not accessible by the player. Cliffs can be used to end the play area with an abyss or steep drops to the land below. Canyon and mountain levels will always make worlds using the cliff technique look much bigger.

We will always include some areas that are on the other side of large gaps, but are not accessible by the player. These gaps give the player a sense of intrigue. If we really want to tease the player, we can put a fancy sculpture or other artifact that is visible on a distant section, so that it seems, to the player at least, to be an important component of the game. We will put a collider around this area so it is not accessible, even if the player goes to extraordinary lengths to try to reach it. If, by some miracle, the player did actually manage to reach this area, we could congratulate them with an **Easter Egg**.

Easter Eggs

Apple may reject our application if we put a secret in it and don't tell them. Apple has this to say about Easter Eggs:

> "Easter Eggs, little programs hidden inside other programs, have long been part of the programming universe, most often as jokes or ways to sign otherwise uncredited work for those able to find them. If you want to add an innocuous Easter Egg to your application for that purpose, just use the demo Account Field to let the review team know the unlocking steps. Apple considers this information confidential and will not reveal those steps or their existence.
>
> On the other hand, not telling the review team about an Easter Egg in your code, in order to circumvent the review process is not allowed. When its existence becomes known, as it inevitably will, our first step will be to remove the offending application from the App Store."

Fourth consideration: Embellishment

Since, we want to keep the player interested, we can take advantage of several classic methods that require relatively little implementation effort. A great way to see how game developers use these classic methods on limited hardware platforms is to play older games.

Fog: Not such a great idea

Fog is one of the oldest tricks in the book. It is a classic that is used when game developers don't want to render something because it's too far away. It is also used when it may be obvious to the player that an object has vanished. The solution has been to obscure objects with fog.

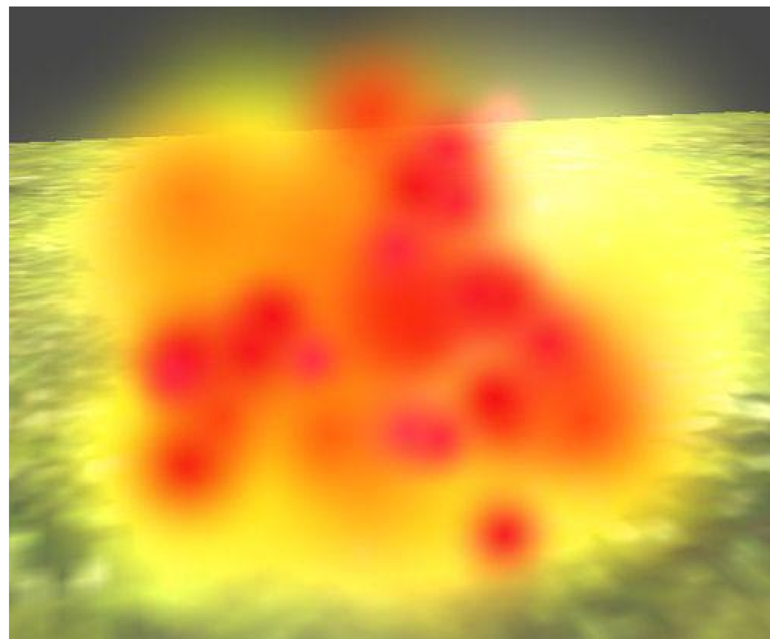
Another factor that has led to the use of fog as an embellishment device is that fog adds ambience.

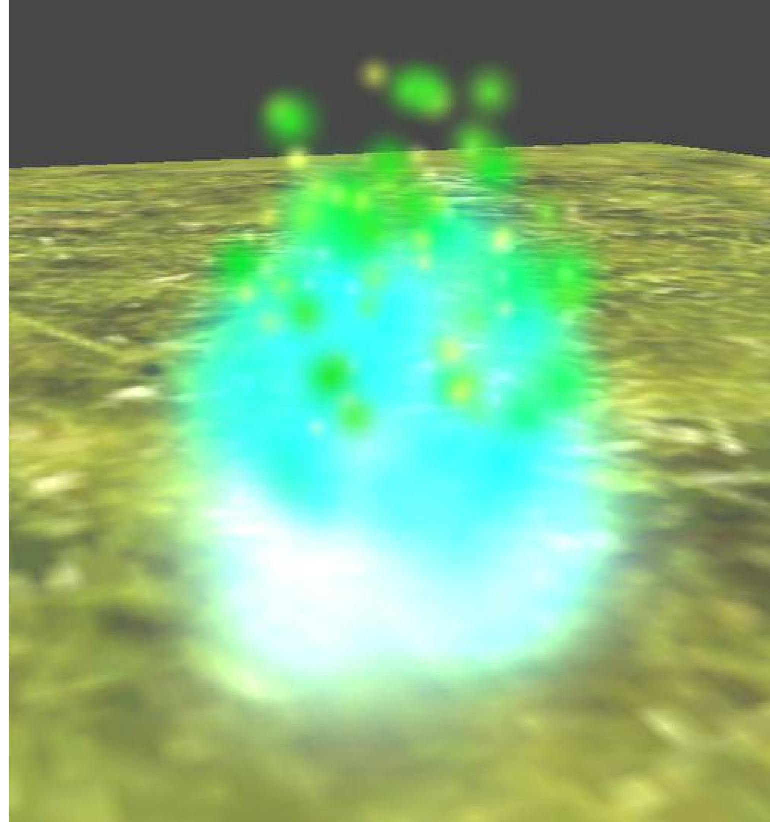
Particles make everything better

Particle effects will make your game seem more vibrant. They're great to:

- Add ambience
- Flag objectives
- Add sparkle
- Create explosions, vents, jets, even lasers

The following are two examples of how vibrant we can make our particles on iOS:





For performance reasons, there are several important things to note about particles on iOS platforms, which are as follows:

- There should be no more than 50 particles per system
- There should be no more than 50-200 particles on the screen at a time
- Particles should be small (because particle shaders use alpha testing, which itself has performance issues). For very small particle textures, it's best to use no alpha channel at all. We should also avoid additive shaders on particles as they contain large numbers of multiply operations
- Particle collisions should be limited; we can even write a script to disable the particle collider if the distance from the player is too far away for a collision to be noticeable or if there are too many particles on screen.

Atlas your textures: Share those materials

Textures that will be frequently rendered together should be consolidated into a single texture atlas.

A texture atlas is simply a single image file that contains multiple sub-images, such that different parts of the large image can be UV Mapped to texture different objects.

Typically related objects that should be atlased include terrain textures, foliage textures, and building textures.

Unity3D's static batching will result in meshes-sharing materials to be combined, so they can be rendered in one pass. In order to enable static batching, we need to make sure that the objects don't move and mark the objects as static in the editor.

The following is an example of a texture atlas that contains flowers and tree bark consolidated into a single image stored in a single file. This means that if trees and flowers that use this texture atlas are visible together, they will require only a single draw call to render. This assumes that they use the same material, which means they also share a shader:

We may also opt to add solid texture and alpha tested textures to larger atlases. This is practical if we have a building with lots of static windows, but also textures for the walls and floors that can all be rendered in a single pass.

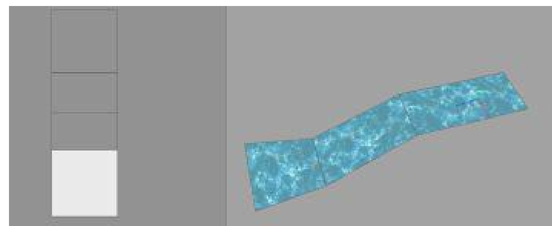
Water makes things look bigger

Water features can be added quickly and easily.

The only thing better than water is water in motion, for example, waves on a beach or a flowing river.

To make a non-linear body of water (such as a river) flow, we simply need to give it a linear UVW

map, as shown in the following image:



Once the texture is mapped, we can apply a UV offset to the texture of the object and the water will make it appear to flow smoothly, even around curved areas.

If an area looks bland, try adding a fountain or waterfall for some extra color. If water doesn't fit the level theme, we can add lava or radioactive ooze (or anything else that we can think of that flows).

You can fly

Giving the player the illusion of flying, makes your game environment more interesting. We could decide to grant the player a flying power with a time limit (there should always be a time limit, so that the power stays elusive and interesting). One way to make a power temporary is by adding trigger areas that cancel flying in zones that we have not prepared for flying. Using such trigger areas is ideal if there is something in the game story line that would cause the player to lose this special power (for example, the spell has only a small area of effect). Players always want to fly, but our game becomes too simple if they can do it all the time. So, it's a good idea to put a time limit on flights (15-40 seconds), and to only grant flying powers when the player reaches certain tokens.

In addition, the height at which a player can fly needs to be limited, such that they do not achieve a bird's eye view that would result in the frame rate dropping due to the amount of rendering needed or the scene being clipped due to the far clip plane. If we really want the player to achieve a great height with a bird's eye view of the world, we are going to need to implement some pretty fancy scripts to substitute parts of our scene, when the player achieves those lofty heights.

Broad views of great expanses are definitely not a strong point of Unity3D on iOS platforms.

Fifth consideration: Teleportation

If we have large distances between important levels, there should be a means to quickly jump between them. We can implement this many ways, but there are, of course, classic ways of doing it.

The warp gate

Once a new destination is reached, the warp gate in the new location may be opened, allowing quick passage to any other open warp gates.

The airship

The airship is usually used in fantasy games. The airship resembles a seafaring ship, but floats in the air. By boarding the airship, the player can fly to a new location.

The spell

If our game is a **Role Playing Game**, this is pretty much obligatory. The player will be expected to have an inventory of spells and a teleport spell is something that can be activated when enough mana is available and the player is not in combat.

The world map

If the player's movement is to be unrestricted, then they can simply pull up the pause menu, open the world map, and choose a new location.

The train

It's not glamorous to travel by car or bus, but the train is an excellent touch in steam punk style games (a game in which technology is advanced, yet appears antiquated). If we are doing a mystery game, the train adds atmosphere as the protagonist can think things over on the train, or the train could be attacked (by a mysterious stranger). We could also select the modern alternative to the train, the subway, for urban environments.

To summarize:

The following are the most important things that we need to remember to do:

- Split levels into sections, so they can be loaded additively
- Split scenes into blocks of terrain meshes
- Put curves in our levels to facilitate occlusion culling
- Atlas textures that will be rendered
- Plan ahead: retrofitting levels is a pain
- Levels should appear bigger than they actually are
- Water is always nice
- It's not always what the player can reach, but rather what they can't that intrigues them
- Some low polygon background art makes the world look vast
- Particles add realism and shine
- Give the player limited freedom with the illusion of great freedom

Culling is important

Enough cannot be said about culling. Culling is so important that there are multiple ways to cull different objects and sophisticated software tools that include pre-processing of scenes to achieve maximum culling have been integrated into Unity3D.

Near Clip

The near clip plane refers to the minimum distance away from the camera origin that an object must be before being rendered.

If the near clip plane were zero, then objects right on top of the camera origin would be rendered, which is usually distracting and not visually appealing. If the near clip is set to far away, then objects you don't want clipped will vanish, and if the player stands too close to an object, they may be able to see right through it.

Far clip

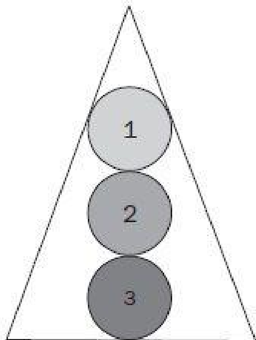
The far clip plane refers to the maximum distance away from the camera an object can be, before it will be clipped (not rendered).

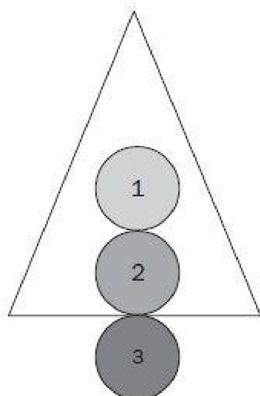
It is important to adjust the far clip, so that it is far enough away that it does not clip your objects, however, it should be close enough that it does not do needless depth testing, which would slow down your processing speed.

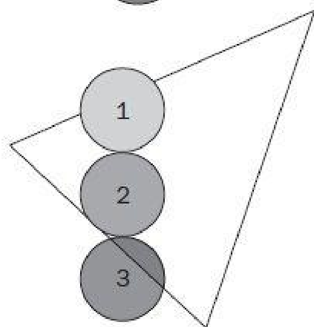
Arguably, there is no reason to have a far clip plane because distance culling should take care of it but because, distance culling doesn't have to apply to every object in a scene, the far clip plane is the default.

The drawing library (OpenGL ES) also requires a far clip plane in order to process the image properly, an attempt to set the far clip to zero will result in only the background rendering. Conversely, setting the far clip to the constant INFINITY will result in a strange, cryptic, and unnerving internal error. It is, therefore, essential that the far clip be farther than the distance culling distances, but still reasonably close.

In the following image, items within the field of view are rendered. When item 3 exceeds the distance specified by the far clip plane, it is not rendered. If item 3 is viewed at an angle and part of it exceeds the far clip distance, part of item 3 will render and the rest of it will display the camera background. Imagine a parabolic dish where the center of the dish exceeds the far clip, the edges of the dish will still render, but the center will be clipped and show a background texture, color, or skybox as follows:


1
2
3

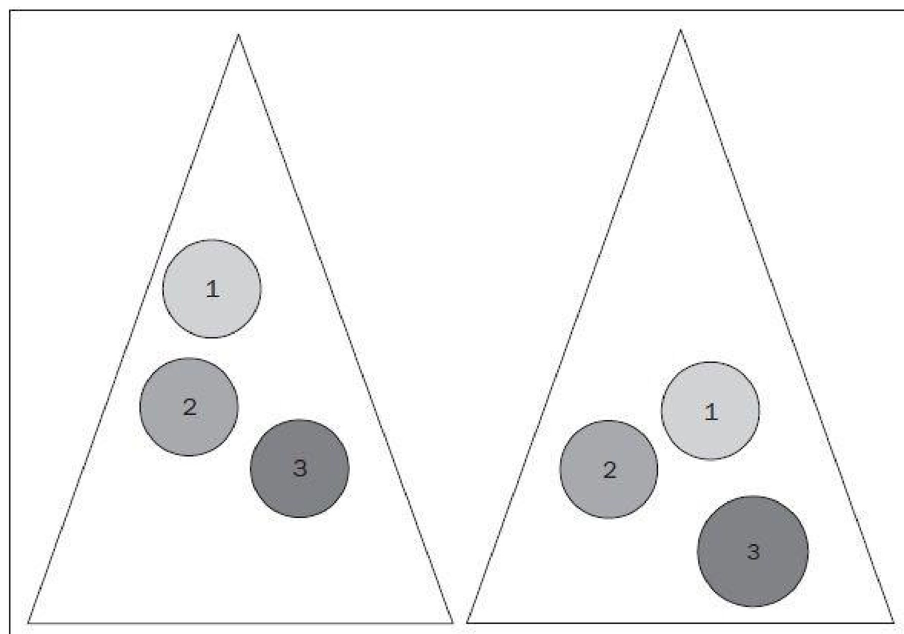

1
2
3


1
2
3

Distance culling

Distance culling is used to cull objects based on how far they are from the camera origin. Using layers in Unity3D, smaller, less noticeable objects can be culled before large objects. Distance Culling still complies with the far clip plane, so any culling distance farther than the far clip plane will always be culled. The far clip plane should be considered to be the absolute maximum distance any depth testing and geometry rendering should be done.

The following is a figure, demonstrating distance culling, when object 1 exceeds the culling distance set for its layer that it ceases rendering (It will not do anything fancy, it just vanishes. Therefore, it is crucial to make the distance far away, so the player doesn't notice the object disappear).



The following code should be applied to any camera we want to use with distance culling. Distances can be represented as any (floating point, >=0) number and should be less than the far clip plane: you cannot render any object farther than the far clip plane:

```
var distances = new float[32];
function Start ()
{
//Setting a value to zero will cause that layer to
// use the farclip
//User defined layers start at layer 8, which
// is distances[8]
this.camera.layerCullDistances = distances;
}
functionsetCullDistance(index : int, distance : float)
{
distances[index]=distance;
this.camera.layerCullDistances = distances;
```

```
}
```

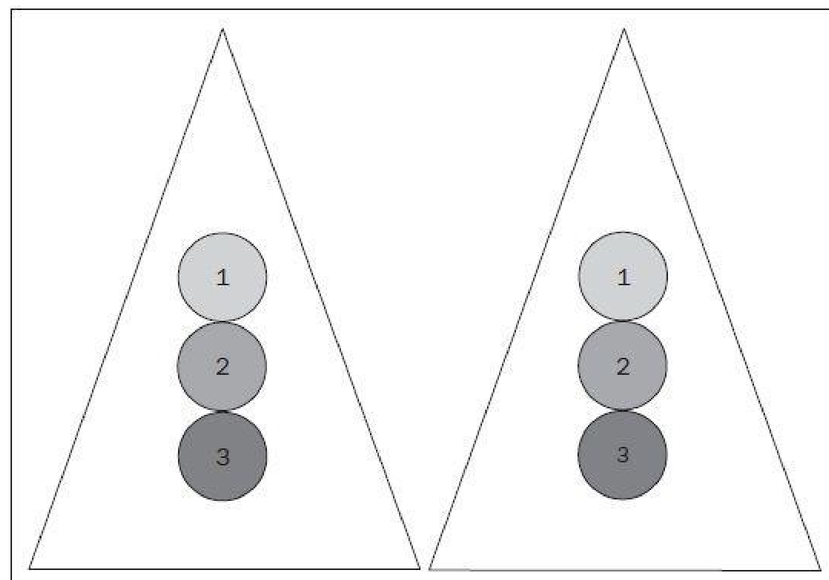
Putting an object on the layer associated with the clipping distance will cause the camera to cull it, when it's far away. It is recommended that you use at least three distance culling layers: **Near** for pickups and enemies, **Medium** for scenery, and **Far** for objectives and terrain chunks.

Occlusion culling

Occlusion culling reduces the amount of required rendering computation time by caching the visibility of objects. While iOS devices have built-in dedicated vertex computation, hardware occlusion culling can still be beneficial because Unity3D can avoid passing invisible geometry to said hardware.

The following image demonstrates how occlusion culling is beneficial.

Imagine object 3 is a section of terrain and object 2 is a large wall that blocks the player's view of object 3. Finally, imagine object 1 is a small tree in front of the large wall. Since the tree doesn't obscure the wall, the wall is still visible. However, the wall is obscuring the block of terrain, so even though the terrain is in memory, it is not being rendered.



Occlusion culling is achieved by pre-computing **PVS (Potentially VisibleSet)**, an array of bounds that cache the geometry, which should be rendered when a camera is facing a certain direction. Configuring and bounding zones and generating occlusion culling data should be the last step in making your game, as any change in the layout of your level will require that occlusion culling be computed again.

If you are experiencing problems with performance during testing, try reducing the far clip plane or adjusting your distance culling. You may also generate a quick preview of your occlusion settings by setting the view cell resolution very low.

Summary

In this chapter, we have covered the following:

- The important device differences across the iOS device family and between the desktop platforms and iOS platforms that we need to consider before we begin to create a Unity3D game for iOS
- How the iOS SDK setup differs from the Mac OS X desktop platform setup, when targeting different OS versions
- How to ensure that our game can be deployed on all the iOS platform devices
- The need to consider everything from the kind of game that we want to develop to the kind of level layouts and artwork that we will use to create the most efficient game possible

Chapter 2. iOS Performance Guide

iOS devices are a relatively new technology, but because of their compact form factor, they will perform at nowhere near the level of desktop platforms. Because of this, they are certainly nowhere near as powerful as today's modern dedicated game consoles. From a hardware perspective, iOS devices are not so much equivalent to a Playstation 3 as they are to an Original Playstation 1. It's not a perfect analogy, but the graphical performance of both platforms is relatively similar. Keeping that in mind, we understand that we will not be making the most stunning cutting edge game on the market (though we might pull off the most stunning mobile game).

The main thing we need to remember is that this is not a comparison to other mobile technologies (they are similarly limited), but rather meant to indicate that a modern mobile device is equivalent in capability to an older game console.

In this chapter, we will learn the following:

- The right kind of game to choose for deployment on iOS devices
- The important differences between iOS device hardware and desktop computer hardware
- The things to do, and the things not to do, in Unity3D for iOS game development
- The importance of culling on iOS

Choose the right game

> Don't expect to make Call of Duty

This section outlines a few good starting points for games and how to make our game look a whole lot cooler than it otherwise may have been. Remember, the iOS market is primarily for casual gamers, as is demonstrated by the popularity of games such as **Angry Birds**.

Stay grounded

If we sprout wings, we suddenly see a whole lot more of the level. This means that from a design, development, and testing perspective, we've got a lot more work to do, which translates into higher cost and a longer time to market. It also makes occlusion culling calculations take significantly longer, because we have to wait longer while Unity3D calculates another dimension (the Y-axis) of Potentially Visible Sets (PVS). This is not meant to imply that the Y-axis is never ignored, but rather to clarify that the BEAST occlusion culling engine will need to do more calculations, when the player is not grounded (assuming additional cells are stacked). So, for Unity3D on iOS, it is always better to keep the player on the ground.

Again, this doesn't mean that the player can never fly, and it doesn't mean we cannot make a flying game with Unity3D. It simply means that games in which the player is grounded have a lot of advantages and less for us to think about when using Unity3D.

Choose first person

If we use a first person camera, we don't need to develop a main character, and most importantly, the modeling and artwork for the main character. If we do not have dedicated and highly skilled 3D artists in our team, or if we have time constraints, then enemies such as, ghosts, zombies, automated turrets, and orb-shaped drones are all a good choice, because they require few polygons and can be animated with ease. Even simple animations go a long way to making this, otherwise static content, more interesting. The most efficient enemies are ones that need to be modeled, textured, and then just floated into place or along a pivotal path.

Avoiding third person

Using a third person camera requires the use of a player character model. Even using a simple player model means that there will be a 2000 vertex model on screen at all times. This will negatively impact our ability to render other things. If we plan on making platformer players, expect a player character model, consider developing a character that is very simple, and therefore easy to render. Unity has performance optimizations for animating skinned meshes, but your character design may only consist of a single skinned mesh for these optimizations to work.

In addition to developing the high polygon count character, we will need to provide all of the animations for that character. This can include walk cycles, run cycles, climbing, swimming, and so forth, all of which adds complexity to the game, which in turn increases the development and testing time.

Things to avoid

iOS devices are currently not ideal for several settings, here are a few things that will be a barrier to our development.

Open planes

Making an open level, instead of a level that simply appears open, is a big mistake. As soon as we make an open level, we allow the player to do several things: notice distance culling, reach areas without proper collisions, walk somewhere that has not finished loading. If we do plan to make such a level, extreme care must be taken if we don't want to break the illusion of our virtual world. It is very important in 3D gaming that we design the game structure to be well matched to the surrounding world, so that there is a balance between the two.

Speed

If we allow the player to rapidly traverse complex areas, we will have to load and unload objects (or perhaps entire scenes) frequently. If these transitions are complex, and cannot be accomplished quickly, the game will not flow. It is important to not allow the player to move quickly, for example, entering a vehicle in very populated areas could be a problem. We need to limit the player's velocity, when they approach load points and make areas designed to be race tracks using simple and low polygon objects. If a vast sandbox environment is available for the player to explore, it's important to provide the player with a mechanism to travel quickly (for example, teleport) between the important areas of the game, so that they do not become frustrated with the size of the game level and the distance between the important game elements.

Flight

Flight is challenging to implement in Unity3D on iOS platforms, unless the player's game is restricted, for example, in an arena. In general, basing a game on unlimited flight in an open area is a bad idea. However, limiting altitude and speed may allow us to base a game on unlimited flight in an open area.

Artificial Intelligence (AI)

Non-player characters often need an AI system to perform basic functions, such as patrolling an area or driving a car around a track. Often, this AI requires path finding. Complex AI, including sophisticated path finding, is CPU-intensive and as such is not well suited to mobile platforms. Keeping our AI as simple as possible will definitely boost the performance of our iOS game.

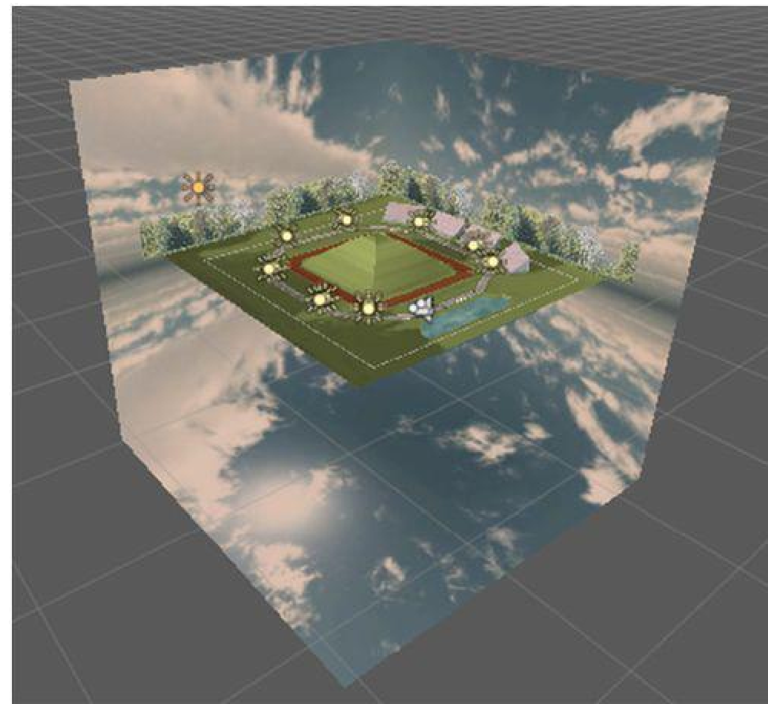
Things to include

Here are a few things that will help us as we develop our Unity3D game for iOS devices. It includes ideas on the kinds of terrains that work on mobile platforms as well as general guidelines for the various game genres that make sense on mobile platforms. Finally, it includes some general gaming components that can be used effectively on mobile platforms.

Skybox

While not technically a component of terrain, a skybox is an important aspect of terrain, since it is what is visible when our player looks beyond the terrain. While Unity3D provides support for skyboxes, a typical mobile game will use a cube with reversed normals (so that the texture renders inside the cube rather than outside the cube) instead of the built-in skybox.

The following screenshot shows how a skybox is implemented in Unity3D on an iOS device:



Rolling our own terrain

While the latest version of Unity3D for iOS does include the terrain engine, it is recommended by Unity Technologies that it should not be used! As a result, we need to create our own terrains using 3D objects. Typically, this means starting with a subdivided plane and extruding it to fit our game theme. Of course, there are exceptions for simple terrains, and we may find a way to use fewer

polygons or even take advantage of the built-in terrain engine. However, in general, a terrain tends to be a moderate number or a fixed set of extruded polygons.

Mountainous terrain

Mountains give us an excuse to put large cliffs in the way of the player. Cliffs require less smoothing (polygons) than hills and block more of the player's vision, thus requiring less rendering time. Mountains can frequently have dungeons in caverns, and dungeons are even easier to render because they are interiors, and so the number of visible polygons that need to be rendered are significantly reduced.

Dungeons

Dungeons, in this context, refer to labyrinth-type levels that may become progressively more difficult as the player descends. Dungeons are popular in older games, because the player's behavior is predictable, and it is therefore easy to calculate what should be loaded. Gamers have come to like dungeons as a side effect of their prevalence: they tend to present a greater challenge than other parts of the game. Casual gamers tend to be irritated by dungeons, so there should only be a few on the critical path. However, much more challenging dungeons may be present in side quests.

Secrets

If it makes sense within the context of our game, we can hide secrets in plane sight on a mountain (like on the other side of a fissure) and require the player to solve a puzzle to reach them. Secrets are usually for players who backtrack after attaining the objects required to reach a restricted area, but it's a good idea to include a few easy secrets that players of any skill level can obtain.

Using urban terrain

Urban areas have tall buildings. Tall buildings obscure other things. In other words, occlusion culling can be used extensively in urban environments. Of course, if we can get away with just using distance culling and still achieve reasonable performance, then we should go with that (occlusion culling is not a good idea if it's not needed, as it will just slow down our game).

In our urban environments, buildings that the player may enter and that have no windows such that the player cannot look inside, should have their contents disabled until the player reaches the door. Disabling content is very similar to occlusion culling, but also prevents script execution, which in turn improves game performance.

In addition, items that do not need to be simulated, seen when the player cannot see them, or for which the simulation can be easily adjusted based on the last time they were seen, may be disabled using the `OnBecameInvisible()` function to further improve performance.

Adding life

Because urban areas provide us with significantly better culling performance, we can add more detailed objects than we would use in an open environment. For example, a city seems rather empty without people walking around or cars driving past. In addition, when a car goes around a corner, we can wait until the player forgets about it (20-30 seconds) and then unload it.

Because urban environments tend to be particularly distracting, it is easy to have low polygon objects generated in the background, and then high polygon objects can be swapped in when the player approaches the object. These details, of course, add development time, so they need to be included keeping the overall project schedule in mind, and in some cases we may add the touches only in a sequel to a popular game.

Secrets

Urban environments have lots of room for secrets. We can put secret tokens in shops, alleys, dumpsters, hotels, on roofs, benches, billboards, or trains. One of the best places in our game to hide secrets is in the urban environments.

While secrets can be an interesting sideline to the main game theme, and fun for many players, they should not be over used. In this chapter, we will mention secrets in several contexts. This is not meant to imply that secrets should be used in all of those contexts, but rather that they may be appropriate in some of those contexts, provided they add an element of fun to the game.

Using suburban terrain

Suburban terrain allows for limited occlusion culling, because suburban buildings tend to be shorter, smaller, and more spaced out.

Having said that, one great benefit to spacing things is that it may allow us to use distance culling without needing to even think about setting up occlusion culling.

Large fences can be used to make it appear as if we could find a way to reach another part of a level, and when it is possible to reach those areas using a fence, we will force the player to take the long way around, which in turn gives us time to load the next scene.

Adding life

Because suburban terrain is less dense than urban terrain and things are more spread out, it is natural for the player to expect that there will be fewer cars and fewer people. Having less life in areas, where less culling can be applied, creates the balance that we need to keep the game running smoothly. A game that is set in an urban environment should have important (or just processor hogging) events set in suburban areas around the urban center.

Secrets

Dumpsters, alleys, derelict buildings, rooftops, and tunnels are good places for secret items or areas in a suburban environment.

Using platform terrain

Platform terrain consists of, as its name would suggest, platforms of variable sizes, usually accessible via jumping or bridges. The key things to consider for a platform terrain are as follows:

- Secrets are found on very small, out of the way, platforms
- Transition areas are located on long and thin platforms
- Important areas are large, sometimes segmented, platforms
- Boss battles take place in arenas with no exits

Adding life

When it comes to platforms, the player usually expects some enemies to plow through, a few jumping and switch puzzles.

While not everyone will agree, we think that platformers have some potential advantages over other genres, depending on your game idea:

- Several top gaming companies got started by developing platformers, simple shooters, arcade, or driving games.
- Many tutorials developed by game engine companies (including the Unity3D Lerpz tutorial) are for platformers.
- Many game engine companies and third-party developers produce Platformer Starter Kits.
- Platformers lend themselves, quite naturally, to the inclusion of puzzles and other challenges that may not integrate, as easily, into other game genres.
- Some really interesting games have combined the elements of a 2D platformer with 3D challenges. These games make use of the power of the 3D engine to seamlessly blend between the 2D and 3D content.

Of course, no game is easier or better to create than any other game, and while we believe there are advantages to the platformer game genres for some game ideas, that doesn't mean that platformers are always better or easier games to develop. We should be careful to not develop a game based on the idea that one genre or the other is better, but rather we should consider all of the advantages (and disadvantages) of the various genres, when deciding which we will ultimately pursue to bring our game idea to life.

Mobile game genres

Not all game genres are suited to mobile platforms. In this section, we list the kind of games that work well on mobile platforms and some of the basics of those game genres. This information can be used as a starting point, the beginning of our journey from desktop gaming into mobile gaming. These concepts are not a cookie cutter solution to implementing a final game concept, but rather a place to start thinking about the capabilities of mobile game platforms in a realistic way.

In many cases, mobile game developers can look at console games from the Playstation 1 to get an idea of the kind of things that work well on modern mobile iOS devices. The genres that we will consider are as follows:

- Platformer basics
- First Person Shooter basics
- Puzzle basics
- Arcade basics

Platformer basics

Platformer game controls usually consist of player movements along with primary and secondary action commands (in a combat platformer, these would, for example, be attacks). There are a certain number of basic, special, and objective pickups per level. In order to advance the story, only basic and objective pickups must be collected. However, all pickups must be collected to complete the game in its entirety. Usually, completing 100 percent of the game yields a special bonus section.

Pickups

The following table lists and describes the typical pickups that we would use in the development of a platform-style game:

Pickup	Description
Basic	This pickup appears frequently. It is to keep the players interested. It may be required to complete a level 100%, or it may have monetary value.
Special	This pickup appears rarely, is difficult to find, and unlocks special features.
Intermediate	After collecting several of these (usually five), an objective pickup appears.
Objective	In order to complete the level, the player must attain this item.
Key	Unlock chests or doors, may not always take physical key form.
Health	Health restores hitpoints to the player character.

Upgrade	The player may find or purchase upgrades to improve their hitpoints, armor, or attack damage.
Power up	Power ups differ from upgrades, in that they provide extraordinarily augmented abilities, usually defense, speed, or attack power for a brief period of time. Power ups may also provide the player with a means to reach otherwise unreachable areas. Power ups occur for a period of time, undisclosed to the player, but are usually accompanied by music that speeds up when the power up is reaching the time limit (the player, or a status indicator, may also begin blinking rapidly to indicate this).

Chests

The following table lists and describes the typical chests that we would use in the development of a platform-style game:

Chest	Description
Basic	This chest contains a set item. It will always dispense the same thing every time the player enters the level. This may be health, basic pickups, or even ammunition.
Help	This chest appears at strategic intervals. It checks the player's status and dispenses the most useful item.
Random	This chest dispenses a random item that may help or hinder the player.
Locked	This chest requires a specific key to open, yields an objective pickup, special pickup, or a large number of basic pickups.
Explosive	This chest yields basic pickups when destroyed, but also damages anything within the range when it explodes. It starts a destruction countdown (3-5 sec) when touched or attacked.

Restorers

The following table lists and describes the typical restorers that we would use in the development of a platform-style game. Restorers, usually appear immediately before a particularly difficult challenge or boss battle. They take the place of the otherwise many smaller help chests:

Restorer	Description
Health	Entering this zone restores health completely.
Energy	Entering this zone restores mana, ammo, or stamina.

Mechanics

The following table lists and describes the typical mechanics that we would use in the development of a platform-style game:

Mechanic	Description

Moving platform	A platform, usually floating by magical or technological means, allows the player to cross a gap.
Barrier	Barriers prevent the player from advancing, and usually require a key or a series of events to remove them.
Pressure sensor	Pressure sensors are a special kind of switch that toggles on when something, usually the player or a movable object is placed on top of them. If this object is removed, the switch will toggle off and the mechanism it operates will deactivate or toggle states.
Timed lock	Timed locks are usually switches activated by being attacked or touched, once activated a timer begins that is indicated by a countdown in the GUI or by a ticking noise in the background. The latter option tends to be more suspenseful, and the ticking should speed up near the end of the timer.
Enemies	Platformers become very boring without anything in our way. Harass the player with enemies of varying strengths and weaknesses. We may also create enemies that cannot be destroyed unless the player has powered up.
Bosses	Bosses are enemies that are encountered in arenas. They are usually quite large and cannot be defeated by normal means. They usually appear after a series of levels, before progressing from one home world to the next.

First Person Shooter basics

Shooters are pretty straightforward and usually progress in a linear manner. The only exception is if the player can enter vehicles, at which point they drive to their next destination. The player has several guns, and the game either pauses to allow the player to switch weapons or has quick weapon changing.

Vehicles typically have a primary attack weapon and a secondary defensive weapon. If the game theme is war, then the player is typically part of a group that includes AI-controlled NPCs (Non-player Characters) that are part of the players unit.

Items

The following table lists and describes the typical items that we would use in the development of a shooter-style game:

Item	Description
Ammunition pickup	This pickup appears frequently. It is to keep the players interested. It may be required to complete a level 100 percent or it may have monetary value.
Special pickup	This pickup appears rarely, is difficult to find, and unlocks special features.
Objective pickup	In order to complete the level, the player must obtain this item.
Upgrades	The player may find or purchase upgrades to improve their hit points, armor, or attack damage.
Weapons	Weapons come in packages sent by our allies and/or are dropped by defeated soldiers.

Vehicles

The following table lists and describes the typical vehicles that we would use in the development of a shooter-style game:

Vehicle	Description
Turret	Turrets are stationary (but other than that they are identical in nature to a vehicle, which is why they are listed here), but usually deal significantly more damage than average vehicles.
Car	Cars refer to civilian cars. They are not armored and possess limited or no defensive capabilities.
Armored car	An Armored Car is a military or paramilitary vehicle that may be boarded by the player with NPC passengers. NPCs man turrets, while the player drives. The player may also man the turret, though NPCs are usually poor drivers.
Tank/Personnel Carrier	A Tank or Personnel Carrier is a military vehicle that may be boarded by the player with NPC passengers. While NPC passengers are on board, they are protected but do not command weapons. Only the player maneuvers the tank and fires its weapons.
Aircraft	This is a flying vehicle. It may take the form of a transport, fighter gunship, or bomber. Transports move faster but are poorly armored and armed, fighter gunships are designed for air combat, and bombers have limited air attack power but deal significant damage to ground targets.
Alien craft	Science Fiction genre shooters usually contain enemy aircrafts. These aircrafts are more difficult to obtain than standard aircrafts. Because of the level of difficulty, the player is rewarded with superior capabilities.

Puzzle basics

Puzzles may refer to casual puzzle games or full 3D puzzle/mystery games. Because the main focus of this book is developing advanced 3D games, this section covers only full 3D games/mystery game concepts, and not casual puzzle game concepts.

Items

The following table lists and describes the typical items that we would use in the development of a puzzle-style game:

Items	Description
Lock and key	Puzzle games consist primarily of locks and keys. This genre can be defined by answering questions such as: which key goes in which lock? When? How? Where is the key? Where is the lock? Does opening this door lock another door?
Switch	Toggling switches activates or releases one or several barricades. Toggling switches in the correct order allows access to more switches. Toggling all the switches in the correct order allows the player to advance.
Objective pickup	In order to complete the level, the player must obtain this item. In puzzle games, this is optional.

Clue	Clues help the player to proceed or are a primary pickup in a mystery game.

Mechanics

The following table lists and describes the typical mechanics that we would use in the development of a puzzle-style game:

Mechanic	Description
Barricade	Barricades impede the player's movement, but are not necessarily stationary. Barricades may be dangerous. Examples of dangerous barricades are an electrified fence or a moving blade.
Timing	The player must toggle a switch at the correct time or the player has a limited time to run through a door after toggling a switch.
Pressure sensor	Moving an object on top of this will hold a door open. The player stepping on this will also open the door.
Movable block	Moving this block will block or open a path. It may also activate a pressure sensor.
Conveyer	The player is forced to move in the direction the conveyer is moving. Toggling a switch may reverse the conveyer's direction.
Lasers	Using a line render and a little collision logic, simple lasers can be made. Lasers may even have sweeping movements that require good timing to pass. Touching a laser usually results in the player being repelled and receiving damage.

Arcade basics

Arcade games are the simplest possible type of game, but they were also designed to be the most addictive. Because arcade games were developed to *collect* quarters from the unsuspecting masses, they are frequently difficult, but still incredibly fun. These days, we can ask for another 25 cents every time someone runs out of lives, but we can still manage to sell something highly addictive.

Mechanics

The following table lists and describes the typical mechanics that we would use in the development of an arcade-style game:

Mechanic	Description
Lives	When the player runs out of hitpoints (or gets hit, depending on the game), they will lose a life. Extra lives may be collected as a pickup.
	If the player loses all their lives, they will be presented with the continue screen. Traditionally, it is at this stage that arcade machines would demand more quarters. If the demands of the machine were not met within the time

Continue	limit (20-30 seconds) indicated by a countdown, the game would end and their final score would be recorded. Because the players caught on, games no longer demand quarters. However, if we were sneaky, we might get away with selling them credits via an app purchase. The usual work around for this inconvenience is to penalize the player by subtracting a large number of points, if they choose to continue.
Score	Whenever the player performs an action that allows the game to progress, they are granted points. Obtaining points and completing the game are the primary goals. Many times, when games could easily express points in lower terms, they choose to use a multiplier in order to give the player a greater sense of accomplishment. For example, defeating an enemy could easily be expressed as adding one point to the score. However, arcade games frequently grant the player 100 points.
Stages	Arcade games are divided into stages, if a player has to "continue", they are restored to the beginning of the stage or boss fight, regardless of their last checkpoint. Between stages, the player can play mini-games to earn bonuses.
High scores	In order to encourage the player to play often, they may compete against other players playing the same game.

Unified graphic architecture

iOS hardware is different from desktop hardware. iOS hardware has many major differences from modern desktop hardware. The most obvious is that mobile hardware, when compared to desktop hardware, would be considered underpowered. In addition, there are trade-offs between space and time and between computation and drawing. In some cases, these trade-offs can work to our advantage, but in other cases, they can be a significant disadvantage. The best advantage is, of course, mobility, which allows us to take gaming to places where we could never have gone to before, such as the line at a bank.

Shared memory

On iOS devices, memory is shared between the CPU and GPU, which means that if our scene is expensive to render, then the computation of our model may slow down. Conversely, if our model is computationally expensive, then rendering the scene may be slower, depending on which processor the controller (Unity3D Engine) has given priority. Because iOS, like MacOS is built on top of BSD UNIX, physical memory constraints are not an issue, unless disk space is low. This is because the virtual memory system will page out physical memory on demand. That being said, Unity3D for iOS has issues with loading large amounts of memory, and may slow down or even crash our program during large loads. Because of this issue, it is recommended that we load our levels in manageable chunks.

Shared processing

As it turns out, GPUs are ideal for processing a large number of parallel tasks, and Unity3D makes use of this functionality to calculate certain non-graphical operations. This method of processing boosts our maximum output, but it means that our graphics or computation slows down.

Vertex Processing Unit (VPU)

The VPU is a specialized piece of hardware that computes the visible geometry ahead of the Graphics Processing Unit (GPU), and then only hands visible data to the GPU. This is referred to as **deferred rendering**. While such computations eliminate the need to consider overdraw, it still means that any geometry intersecting the field of view must be computed before the next frame is rendered. It is therefore imperative that non-visible faces be eliminated if you are experiencing performance difficulties, as this means the VPU may be at its limit.

Advanced RISC Machine (ARM) Thumb

No, that's not a joke. Thumb is a set of processor instructions that will make our program more compact. However, Thumb is not ideal for larger quantities of floating point numbers, such as 3D graphic computations. If we link against a library not included in Unity3D, it may compile using Thumb. Unfortunately, Unity3D is incompatible with Thumb, so make sure that Thumb is disabled in the Xcode target info, and then perform a clean build.

Dos and Don'ts

This section is important, it will list the Dos and Don'ts in our program in chronological order, from the beginning of our program to its termination.

Programming the main loop

The following tips are included in the main part of the program's scripting logic.

Model View Controller (MVC)

Unity3D is object-oriented. Because of that, we will use the Model View Controller (MVC) terminology. In the case of Unity3D, we, as the programmers, will be manipulating the model, essentially modifying and passing data between the correct objects. Unity3D keeps track of most of the controller business. It keeps track of physics calculations and sending information to the view. Unity3D also manages the view for the 3D world, leaving us to develop only the view for GUIs.

Springboard of death

On the iOS platform, applications are given a limited period of time to load before they are forcibly terminated by the OS. To prevent our game from being unexpectedly terminated before it has had a chance to start, we need to load a small initial level. A good choice for that small initial level is the main menu of our game. However, in some cases, even the main menu can be a complex scene and so the only solution is to load a small empty level and have it, in turn, additively load the real first level.

> **Tip**
>
> Developers who are new to the Unity3D iOS platform frequently believe that they have exceeded the iPhone's capabilities when they observe the resulting crash that occurs when the iOS kills their application because it is taking too long to load their first level.

Cache it: Awake() and Start()

`Awake()`, `Start()`, and `Update()` are Unity3D functions that we can use in any script, and that can contain any program logic that we want to add.

`Awake()` is called when an object is instantiated. `Start()` is called after `Awake()`, if the object is enabled, just before the object begins rendering.

`Update()` is called every time a frame is rendered, so the rate at which it is called will depend on the frame rate of our game. In other words, it's not called at a constant rate but at a variable rate.

Many beginning developers put a lot of program logic in the `Update()` function that, as it turns out, should be put in a different function. Often, using the `Update()` function is the wrong solution, and when not to use the `Update()` function will be a recurring theme in this book.

On a desktop or console platform, there is so much processing power that we could be forgiven for

not knowing which function to use and when to use it. This is because there is so much processing power available on a desktop platform that wasting it will not slow down our game. But a mobile platform is very unforgiving and using the wrong function for the wrong purpose can significantly limit what we can do in our game.

If we want to get our game off the ground, we can't be constantly doing a bunch of calculations that we don't need. For example, when a script is loaded and we want it to find an array of objects, we could do it in `Update()`, but then we would be finding those objects at every frame. Instead of using `Update()`, we can populate a private array in the `Awake()` function, and then only reference items in the array, as needed in the `Update()` function.

> **Note**
>
> There are other, more advanced, functions, like `FixedUpdate()`, that need to be used to apply forces to rigid bodies, but we will talk about them later as they don't suffer the same abuse as `Update()`.

Coroutines, not Update()

Because it is typically used to keep track of timed operations, `Update()` is the most abused function in Unity3D games. Ask any new Unity3D programmer and they will tell you that timed events can be used by doing the following:

```
// Some important variable
var myVar : float=0;
function Update()
{
// increment myVar by step every second
myVar = myVar + step * Time.deltaTime
}
```

This is inefficient! Use Coroutines.

The advantages of using Coroutines are given as follows:

- Coroutines are our friends. It would be nice if Unity3D documented them better, because they are actually deceptively simple to use.
- Coroutines are generators. A generator behaves like functions, but preserves its state between invocations.
- Coroutines are not threads, rather they are interrupted loops running on the main thread. This becomes readily apparent if we accidentally put an infinite loop that does not interrupt itself in a coroutine.

> **Tip**

We need to use caution when writing coroutines that include loops, because a mistake will lock up the Unity3D editor, and we will have to force quit the editor, because the loop cannot be interrupted.

There are two important parts of a Coroutine that are as follows:

- A function which is called
- One or more yield statements somewhere in the function

When a yield statement is reached, the function will not return, but rather it will suspend its execution until the yield condition has completed. When the yield condition completes, the coroutine will resume from the point at which it paused, retaining all of its state (values of local variables).

The following is an example of how to make an operation occur every five seconds, without checking every frame in the `Update()` function:

Note

This is a very simple coroutine that could be implemented using a different function called `InvokeRepeating()`, but we want to see how coroutine works. So for this example, we are using a coroutine. The advantage of using a coroutine is that it can contain multiple yield statements and yield at different points for different amounts of time. If our coroutine only yields a single time for a fixed interval, then `InvokeRepeating()` is a good option.

```
function Awake()
{
// Start a coroutine
MyCoroutine();
//Do some more things
//…
//The end of Awake
Debug.Log("Awake is finished");
}
function MyCoroutine()
{
var myVar : int = 0;
// Do this important stuff
// every 5 seconds
while(true)
{
//Execute my code
Debug.Log("The count is: " + myVar);
myVar = myVar + 1;
// Wait for 5 seconds
yield WaitForSeconds(5);
}
}
```

Tip

Be careful not to create an infinite loop

When using coroutines, we need to make sure that a yield statement will be executed if there is an infinite loop in the script. If we forget to yield, then Unity3D will go into an endless loop in the editor, and we will need to force quit to regain control of our game.

Yield statements may use a number of conditions, including the following:

- `WaitForEndOfFrame`: This waits until the frame has rendered. This is good for screenshots.
- `WaitForSeconds(x:float)`: This delays re-execution for an arbitrary period of time.
- `WaitForFixedUpdate`: This waits for the next fixed-update step. This is useful for physics.
- Coroutine: This operates in one of the two ways. Either we can call the coroutine directly (like this: `MyFunctionName();)` and continue executing the calling function immediately, or we can use the `yield` instruction in front of the coroutine call (like this: `yield MyFunctionName();)` to pause the currently executing function, until the `MyFunctionName()` coroutine finishes executing.

Note

In `C#`, use `yield MonoBehaviour.StartCoroutine(MyFunctionName());` to explicitly start a coroutine: in JavaScript, the compiler will call the `StartCoroutine` function implicitly.

Tip

Challenge

How could the previous loop be broken if some condition is met?

```
//This variable controls the
//execution of the coroutine loop
//When true the loop executes
//When false the loop does not execute and
//the coroutine exits
var go : Boolean = true;
function Awake()
{
// Start a coroutine
MyCoroutine();
//Do some more things
//…
// The end of Awake
Debug.Log("Awake is finished");
}
function MyCoroutine()
{
var myVar : int = 0;
```

```
//Do this important stuff
//every 5 seconds
while(true == go)
{
//Execute my code
Debug.Log("The count is: " + myVar);
myVar = myVar + 1;
// Wait for 5 seconds
yield WaitForSeconds(5);
}
}
//Execute this function to prevent
//the coroutine from running and cause
//it to exit
function ToggleFunction
{
go = !go;
}
```

Note

Yielded functions become generators and cannot return a value.

Tip

`MonoBehaviour.StopAllCoroutines()`; will halt all Coroutine execution in this mono-behavior.

Time

The time step determines how frequently `Update()` and `FixedUpdate()` are called. The difference between `Update()` and `FixedUpdate()` is that `FixedUpdate()` will always run at the defined interval, while `Update()` will run based on best effort, which means the game engine will call our `Update()` functions periodically using different interval times. In fact, if we make heavy use of `FixedUpdate()`, we can actually reduce the frame rate and thus reduce the rate at which `Update()` runs.

Time scale allows us to reduce or speed up the time of our game. For example, we could put the player in slow motion for some specific part of a battle with a particularly challenging boss, if we noticed the player was having trouble defeating that boss.

If we want to pause our game, then we need to set time scale to a very low value, that is, nearly but not equal to zero.

Note

Setting the time scale to zero has caused problems, but if we set it to a very small value, but not zero, we may avoid potential future issues.

Make it static, why instantiate?

So we want to make a `SuperGlobal` class. Well, good object-oriented programming practice tells us that we should use `getter` and `setter` methods. Certainly, if we expect anyone else to interface with our class, we should get and set variables in our `Globals` class using `setter` and `getter` methods. If, however, we happen to be a very small company or even a single person, we probably know our classes backwards and forwards. In other words, we can directly access and change variables without doing error checking. The following is the right way to set a variable followed by the lazy way to set one:

	Globals.js	Other.js

Correct	var money : int;
	static function SetMoney (var i : int)
	{
	money = i;
	}
	
	static function GetMoney ()
	{
	return money;
	}
		function Awake()
		{
		var v: int;
		v = Globals.GetMoney();
		v+=5; //add five money
		Globals.SetMoney(v);
		}

Fast	static money : int;	Globals.money+=5;

Use hashtables

Hashtables are tables of key-value pairs, consisting of a key for access and entry, and a value associated with that key. Keys are strings and their associated values may be any data including other hashtables.

We will discuss hashtables in greater detail later, but it is important to make use of hashtables whenever we are dealing with large amounts of structured information for which speed of access is critical. Hashtables are optimized for this type of access and should be used instead of any other, self-implemented, mechanism for accessing complex sets of information in the program memory.

Note

A hashtable is the .NET equivalent of an `NSDictionary` in Apple's Objective-C framework.

The following table illustrates the JavaScript and Objective C code needed to access key-value pairs:

Description	JavaScript	Objective C
Create a hash table	`var MyHashtable : Hashtable = new Hashtable;`	`NSMutableDictionary *myDictionary = [[NSDictionary alloc] init];`
Check if a key exists in a hash table	`MyHashtable.ContainsKey("Key");`	`[[myDictionary objectForKey:@"key"] != nil]`
Add a key/value pair to a hash table	`MyHashtable.Add("Key",value)`	`[myDictionary setObject: value forKey: key]`
Access a value using the key string	`pulledValue = MyHashetable["KeyForValue"];`	`pulledObject = [myDictionary objectForKey:key]`

Triggers and collisions

When an object collides in Unity3D, the collision detection system will call a function on both objects. The function that is called depends on whether or not the collider is defined as a trigger. If the collider is not a trigger, then the `OnCollision` variants of functions are called. If the collider is a trigger, then the `OnTrigger` variants of the functions are called.

The fundamental difference between the two collision detection mechanisms is that the information gets passed to the functions. `OnCollision` is expensive due to the calculations that need to be made to pass in the Collision information, so it should only be used if you really need that information. If you just need to know that a collision occurred, then either use triggers on the `OnCollision` functions without a parameter.

There is a function called `OnTriggerStay()` that can be used to perform some actions, when an object remains in a trigger area. While this may seem like a useful function, it is highly inefficient unless we need to check every frame.

Typically, we only need to perform this check periodically. Given that there is no reason to use `OnCollisionStay()`. Instead, we can begin a coroutine in `OnCollisionEnter()` and use a flag to terminate it in `OnCollisionExit()`.

Note

If we are managing many objects and need to keep track of them, then this would be a great place to use a hashtable with the `unique id` of the object as its key and a second hashtable as the value. The second hashtable can contain the information that we need to track for that object.

OnBecameVisible()/OnBecameInvisible()

When an object becomes invisible, it will be sent the `OnBecameInvisible()` message. This can be

used to disable unnecessary calculations required only for the visible aesthetics of this object.

The function `OnBecameVisible()` is only called when a rendering component becomes visible, so completely disabling an object while it is not in view is not a good idea unless we have devised another method, using some kind of `manager` object to re-enable and disable items via an external message. In most cases, we will want to disable some, but not all, of the logic being executed on an object. So we need to design our scripts in such a way that it is easy to disable and enable the optional logic.

> **Tip**
>
> The Unity3D editor scene view camera counts as a camera for object visibility purposes (at least as far as the previous two functions are concerned, this is not so with occlusion or distance culling) and will prevent an object from being deemed invisible by the engine. As a result, when we are testing our game in the editor, we need to move away from the desired object in the *Scene* viewport.

ApplicationWillTerminate() or ApplicationWillSuspend()

Make sure that you only implement these functions in one object. We typically implement them on our `Globals` object. These functions are called when iOS wants to end or suspend our program. These operations are time-sensitive, so make sure all the required data for your logic using them has been acquired prior to them being called. This way, you can be sure that information, such as game state, is written out as quickly as possible.

Strict compilation

When we are using a language, where dynamically typed code is perfectly fine (such as JavaScript), our program will compile and execute, and everything will work, but our laziness will result in the need to do some extra runtime compilation to figure out what we were implying. To force the compiler to check our code to see whether it is statically typed, put the strict pragma at the beginning of each script. Doing this will give our game that extra hit of speed.

The strict pragma looks in code as follows:

```
#pragma strict
```

In addition to improving performance, strict coding practices can prevent runtime crashes, so unless there is a good reason to not use strict coding (like porting a large but not performance-critical script from the Desktop to the Mobile platform) then it's always best practice to use `#pragma strict`.

Compilation order

We can't call or reference something before it is compiled.

Typically, interface definitions are included to provide hints to compilers about things that are

external, so that they can be compiled. In Unity3D, it's a bit more challenging and we need to be aware of the order in which things are compiled.

This means that we may need to move certain scripts into certain folders, so that they are compiled before other scripts that reference them.

Essentially, anything that is referenced by something else will need to be put into the `Standard Assets` or `iPhone Standard Assets` folder. There are some other less common places that we may need to put scripts, and we can see all these details on the Unity3D web page here:

http://unity3d.com/support/documentation/ScriptReference/index.Script_compilation_28Advanced29.html

Design to load additively

The best way to load in a scene is to use load additively and asynchronously. There are several mechanisms provided by Unity3D to facilitate additive loading. For example, we can load an included level additively, we can load resources from an included bundle, or we can load from the Internet. The mechanism that we choose will depend on the requirements of our game.

Putting corners in our level allows us to load in sections, while obscuring this operation from the player. If we cannot use the four corners approach, then we can use a door or other kind of transition area to cover the fact that something new is being loaded into the scene. If the next section is large, it will have a longer load time and we will need to create a padding area in the current scene for the player to walk along while the next scene is loading.

Another option that we can use to cover scene-loading is special time distortion effects, for example, we could reduce the players' speed while the next scene is loading or reduce the game time and create a slow motion special effect to cover the fact that a new scene is being loaded.

It's a good idea, in order to prevent a catastrophic failure, like the player falling into an endless abyss, because the scene was not fully loaded, to create an invisible barrier using a box collider. This barrier will prevent the player from proceeding, if they get too close to a partially loaded scene. We will leave the barrier in place, until we are done loading the next scene.

To create the illusion of a fast load, we may need to break things down into multiple additive loads, starting with the terrain and followed by the remaining scene objects. We will let the player proceed once the terrain is loaded, and continue with the rest of the loading even as the player enters the new area.

Artwork

When it comes to performance on an iOS device, our artwork will have a much greater impact on performance than it would on a Desktop or Console platform. The bottom line is simple — what works on other platforms does not work on iOS devices.

Vertex count

Unity for iOS is limited, depending on the device, to around 10,000 to 30,000 vertices per frame. This is quite different from the number of vertices in the scene. That is to say that this is the maximum number of vertices in the camera viewport that should be at one time within this range.

> ## Note
>
> This is a performance limit, not a technical limit. In other words, we can have more vertices, but the frame rate may decrease so dramatically as to make the game unplayable.

Additionally, this assumes that simple shaders, such as the diffuse and vertex-lit shaders, are in use. If more complex shaders are used, then the game frame rate may still become unacceptably slow, even if the vertex count is within the acceptable range.

In other words, the vertex count is a guideline, not a guarantee.

Skinned meshes

In the context of the vertex count, skinned meshes (such as those that animate characters) may report having upwards of 2,000 vertices each. This means that we will be quite limited with respect to how many characters can be seen at once. If we are using a third person view, such that our character is rendered at each frame, then we will be dramatically limiting what other objects can be displayed in the frame.

In an effort to address this, skinned meshes have special optimizations handled by Unity3D to reduce their performance impact. However, these optimizations are effective only if an animation consists of a single skinned mesh.

So, if we do decide to use more than one character, and we want to have the least impact on performance, then the characters should look very similar.

It's very important that skinned mesh has as few bones as possible. For very simple characters, it should be less than 10 bones. And for those requiring higher quality, it should not be more than 15.

Alpha testing

There is currently a bug in the Unity3D for iOS rendering pipeline that presents several problems

with alpha testing.

For starters, using any large geometry with alpha testing could cut our frame rate in half, so we need to avoid using large geometry with any shader that depends on alpha testing. Secondly, if we use an alpha shader, we may notice a white outline, because Unity3D assumes that the provided alpha texture is pre-multiplied. Some developers avoid this by using an alpha cutoff shader; however, this will incur a performance hit.

Making of an alpha map is discussed further in the *foliage* section of this book and in the Unity3D manual under the heading, *Graphics Questions.This* is a link to this section in the Unity3D manual: http://unity3d.com/support/documentation/Manual/Graphics%20how-tos.html

Lights

Real time lights require a lot of processing power. As a general rule, we need to limit, or even eliminate, the number of real time lights on iOS platforms, and use lightmapping whenever possible. Whenever we are considering using a real-time light, we need to ask ourselves if there is any alternative, because any alternative is a better choice than a real-time light.

If we have considered all the alternatives and have decided that a real-time light must be used, then we need to make sure that we do not place lights near scene loading boundaries, as this will cause abnormal behavior during additive loading.

The other thing to consider is the kind of real-time light that we choose. We need to consider using *cheap* lights rather than expensive lights. The following is the order in which we should consider lights, from least expensive to most expensive:

- Ambient lights
- Directional lights
- Point lights
- Spotlights

If we are not already familiar with the different light types, we can find a complete description of them here:

http://unity3d.com/support/documentation/Components/class-Light.html

Note

Typically, when discussing lighting performance, one would mention shadows. However, they are not supported on mobile platforms and thus are not discussed here.

Post processing

Post processing is called post processing for a reason. Post means last. This is important because we need to resist the temptation to do post processing tasks when there are still other tasks to be done. Post processing is time consuming and every time we change something in a scene, we need to re-do the post processing. So, the message is leave the generation of occlusion culling and light mapping data, until the level is finished.

Physics

Unity3D implements physics, and any object that has a rigid body attached to it will be affected by physics. It is important that objects are sized correctly, for example, our humanoid characters should be about 1.8 meters tall so that the physics interaction is correct. If, for some reason, our game objects are not the correct size, such that we need to resize them after the scene has been created, it can be a lot of extra work that could easily have been avoided by simply ensuring game objects were imported into Unity3D with the correct size prior to adding them to the scene.

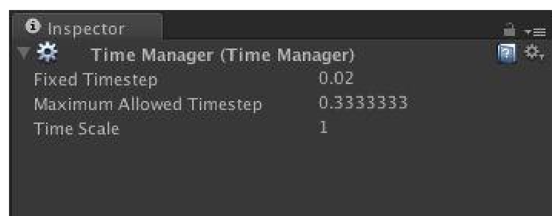
While Unity3D implements physics, there are a number of physics settings that need to be tuned to make sure that physics behave correctly. Special consideration is needed for both fast moving and small objects.

On mobile devices, it's always best to avoid physics unless it's strictly necessary for the gameplay.

FixedUpdate()

`FixedUpdate()` is a function called at an arbitrary time step, independent of `Update()`. Because `FixedUpdate()` is calculated in the model independently of being rendered, it is the correct place to include script logic that modifies the behavior of objects that are affected by physics.

`FixedUpdate()` usually has an interval more frequent than `Update()`, but if our game does a lot of calculations on objects that interact with the physic engine, we may need to modify the frequency with which it is called. We can find the setting called **Fixed Timestep** in the **Time Manager (Time Manager)** settings, under **Edit** | **Project Settings** | **Time**, as shown in the following screenshot:



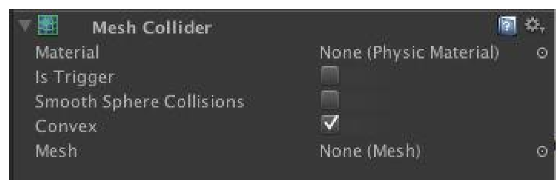
If physics accuracy is not an issue in our game, we can increase the **Fixed Timestep** value and reduce the CPU load. If physics accuracy is an issue in our game, then we will need to decrease the **Fixed Timestep** value, which will increase the CPU load. It's a balancing act and we may need to play with the setting to achieve the best balance between accurate physics calculations and CPU load.

Collision detection

Rigid bodies can be assigned different degrees of fidelity. If objects will be moving quickly, then we need to give them a higher degree of detection, while less important and slow moving objects may be given lower importance, reducing impact on performance.

It is important to remember that Mesh colliders will not collide with each other unless they are

convex, and that needs to be specifically enabled, as shown in the following screenshot:



Stairs and rolling hills should use smooth sphere collisions to prevent the player from *sticking*.

Typically, mesh colliders should be avoided, and instead we will add separate children to the game object forming the collisions out of primitive collider shapes rather than use a mesh collider. It would be impractical, for example, to assign and create a mesh collider, when a box collider will do. However, more complicated meshes can still use mesh colliders.

Mesh colliders are designed to make our job easier, so that we don't need to make complex terrain collisions or stair collisions out of primitives. Don't be afraid to use them for larger complex objects, but remember small simple objects sitting on top of them will fall through if they also use mesh colliders and are affected by gravity. We don't want to get carried away, so if performance is not an issue, we will use mesh colliders to speed up our game development, and only start rolling our own collisions if performance becomes an issue or if we know from the start that it will be an issue.

Deployment

There are a number of things that we need to consider before deploying our game, but we need to be careful about some of the methods that can be used to reduce the binary size, because they may conflict with the feature we are using.

The biggest *gotcha* in this category is the use of stripping combined with using the .NET framework (more on this later). The .NET framework provides us with the richness we may need, for example, hashtables, at the expense of size.

The following two articles go into both player size optimization and the .NET compatibility list. We need to be completely familiar with both of these articles prior to creating the final build of our game.

Player size optimization:

http://unity3d.com/support/documentation/Manual/iphone-playerSizeOptimization.html

The .NET compatibility list:

http://unity3d.com/support/documentation/ScriptReference/MonoCompatibility.html

Culling is important

The most important thing, from a performance perspective, to consider when developing a Unity3D game for iOS devices from a performance perspective is culling (not rendering objects that are not visible to the camera).

Put another way, it is not what the player sees, but rather what the player does not see that most affects the performance of our game.

There are several ways to achieve culling in Unity3D. They are given as follows:

- Frustum culling
  - Camera clipping planes
- Camera layer culling
- Occlusion culling

Frustum culling

Frustum culling is always on, and essentially, culls whatever is not inside the camera view. However, it does not cull things that are in the camera view, but hidden by other objects.

Camera clipping planes

Frustum culling makes use of the near and far clip planes. For a game that was nothing more than moving inside a maze, it can be sufficient to make the near clip plane as far out as possible, so that we cannot see through the walls and the far clip plane close enough that the length of the furthest tunnel is not clipped.

Camera layer culling

Camera layer culling allows us to define the distance at which objects, on that layer, are culled. Camera distance culling is perfect for small objects that are within the far clip plane, but that can pop in and out of view without causing a distraction. This works for rocks, plants, boxes, barrels, and so forth, but is not going to work for buildings.

Each instance of Camera has its own set of layer culling distances. This set is defined as an array of 32 floating point values, which represent the distance from the origin of the camera at which objects on the corresponding layer will be culled.

Note

Unity3D layers consist of built-in layers and user-defined layers. The user-defined layers start at index 8 of the distance culling array.

To enable culling distances on a camera, you can use the following JavaScript:

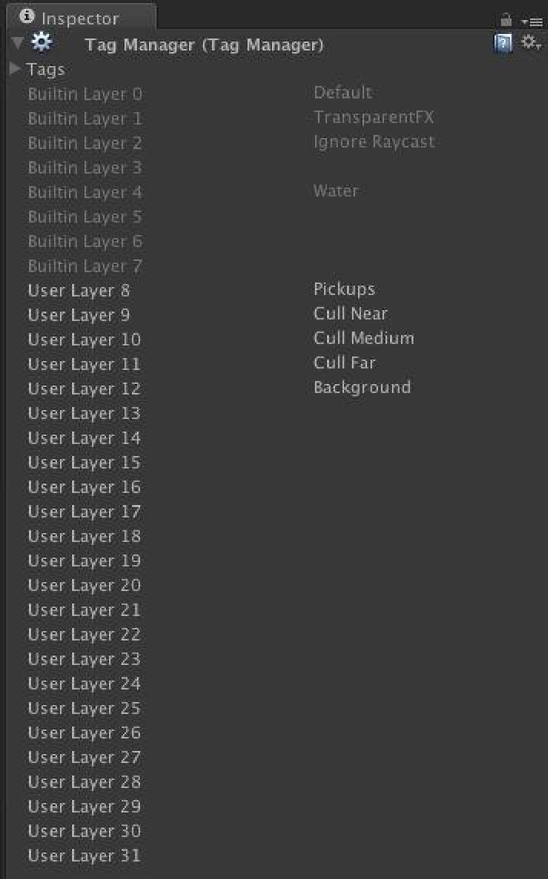
```
// The array of distances to cull
// objects on the corresponding layer
var distances = new float[32];
function Start ()
{
// User defined layers begin at index 8
// All layers with a culling distance of
// 0 use the far clip plane distance.
CullingDistanceUpdate();
}
//CullingDistanceUpdate is a seperate function
// so it can be called via message if the culling
// distances for a cameara are modified
function CullingDistanceUpdate()
{
// camera refers to the instance of Camera
// attached to this.GetComponent(GameObject)
camera.layerCullDistances = distances;
}
```

After attaching this script to a camera, we can set the culling distances in the Unity3D editor, as shown in the following screenshot:

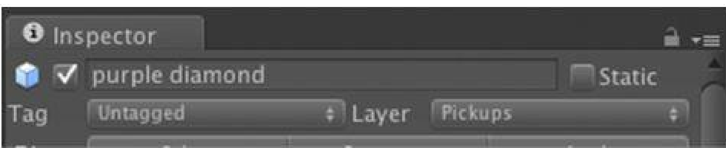
Culling Manager (Script)

Script	CullingManager
Distances	
Size	32
Element 0	0
Element 1	0
Element 2	0
Element 3	0
Element 4	0
Element 5	0
Element 6	0
Element 7	0
Element 8	30
Element 9	50
Element 10	100
Element 11	150
Element 12	300
Element 13	500
Element 14	0
Element 15	0
Element 16	0
Element 17	0
Element 18	0
Element 19	0
Element 20	0
Element 21	0
Element 22	0
Element 23	0
Element 24	0
Element 25	0
Element 26	0
Element 27	0
Element 28	0
Element 29	0
Element 30	0
Element 31	0

Once we assign non-zero values to the distance culling array, we need to give the layers reasonable names in the layer manager, as shown in the following screenshot:



And then whatever Game Objects we put on the corresponding layer will be culled at the corresponding distance, as shown in the following screenshot:



Occlusion culling

Occlusion culling is the best method for culling large objects that are in the Frustum but are obscured. iOS Pro users can use the built-in Umbra culling system to achieve fantastic results in hugely complex worlds.

> ## Note
>
> For iOS basic, there is a free third-party occlusion culling system, called H2H Culling, available in the Unity store.

Umbra Occlusion culling works by creating a model of the Possible Visible Set (PVS). We define a boundary area and a resolution, and from those values, Umbra decides what is visible, based on the camera location and orientation in that area.

In Umbra, view volumes should be used for areas with static objects, and target volumes should be used for areas with moving objects.

Since target volumes are more computationally expensive (they are calculated at runtime), they should not be combined with view volumes. Rather, target volumes should intersect view volumes, where moving objects should be occluded.

Target resolution refers only to the resolution of calculations in target volumes. It is set per instance of Target volume.

The view cell size refers to the number of subdivisions of a volume. During occlusion calculation, a virtual camera will perform visibility calculations for each cell and cache them for later use. Therefore, setting the view cell size is one of the most important considerations as higher resolutions will increase the calculation time and storage space required. It will also determine the accuracy of the occlusion with a greater resolution resulting in greater accuracy.

Under the bake tab, the near and far clips refer to the near and far clips of the virtual camera.

To visualize how changing these parameters will modify the calculation, imagine moving your player into each cell of the occlusion volume and looking in several directions. When looking in a direction, the virtual camera will consider occlusion from the near clip distance to the far clip distance.

To set up Umbra Occlusion culling, do the following:

1. Completely create your scene.
2. Mark objects that will undergo post processing and are stationary as static.
3. Under the **Window** menu, select **Occlusion Culling** to open the window.
4. Under the **Quick Select** drop-down box, select **Create New** to create a new bounding volume for the Occlusion calculation.
5. Scale the new bounding volume to enclose an area.
6. Repeat steps 3-4 for every area that requires culling.
7. Overlap bounding areas to avoid unexpected behavior around boundary transitions.
8. Either in the inspector or the **Occlusion Culling** window under object, check view volume or

target volume for each area.

9. In the **bake** tab, in the **Occlusion Culling** window, set the view cell size.
10. In the **bake** tab, in the **Occlusion Culling** window, set the near and far clip planes.
11. In the **bake** tab, in the **Occlusion Culling** window, set the quality to preview during development and production when you are ready to build a version of the game for release to the App store.

For a more detailed look at the setup of Occlusion Culling, we can look at the Unity3D documentation here:

http://unity3d.com/support/documentation/Manual/Occlusion%20Culling.html

Summary

In this chapter, we have learned the following:

- While performance tuning is important on iOS platforms, if we design our game levels and assets up front with performance in mind, we can save ourselves a lot of reworking in the later stages of our games, because the best way to address performance problems is to avoid them in the first place.
- We need to keep performance always in mind, since we don't want to find ourselves in the position of having developed a game that can never achieve acceptable performance on the target platform.
- The kinds of games that are suitable for iOS platforms: While iOS devices have achieved amazing performance specification, they simply are not capable of running every kind of game.
- The differences between iOS platforms and desktop computers and how those differences can affect Unity3D games.
- The most important things to do, and not to do, on iOS platforms when developing games with Unity3D, so that we don't get caught in common traps that occur when moving to mobile platforms.
- iOS culling is the most important aspect of achieving acceptable performance, and we need to employ culling in the most effective way possible.

Chapter 3. Advanced Game Concepts

When dealing with advanced technology like iOS devices, there are some concepts that require a game developer to think about games differently. When using an advanced game development platform, such as Unity 3D, there are some practical ways to create game assets that can be shared and reused in more than one game.

In this chapter, we will learn the following:

- Important ways to deal with menus on advanced iOS gaming platforms that take advantage of unique device characteristics
- How to implement screen resolution-independent menu systems that will work on iOS devices, regardless of the screen resolution
- How to test and the benefits of testing in the Unity3D editor for games that target iOS devices
- How shaders can be used to solve problems that are typically solved by geometry
- How to organize a Unity3D game project, so that game assets can be shared easily with different game projects

Engaging menus

We know that iOS devices are revolutionary, almost magical, in the hands of the people that use them. The reason for this is because they don't feel like computers. Instead, they feel like an extension of the of the user. The key is creating a magical feel of the integration of hardware and software that pulls the user in and makes the device feel like an extension of the person using it.

Our game menus need to pull people into the game and make them feel like they are a part of the game. We can create a game that, from the beginning to the end, completely immerses and engages players. We can create the kind of games that are simply not possible on any other platform.

Often, when we come from a desktop or web development environment, we are thinking in terms of the screen, the keyboard, the mouse, and perhaps a game controller. With an iOS device, we only have the touch screen, accelerometer, and perhaps gyroscope as input for the device, which together are essentially the game's controller.

Rather than thinking in terms of pointing, hovering, clicking, and typing, we need to start thinking in terms of touching, swiping, pinching, other gestures, and shaking. It is these latter concepts that make an iOS device an extension of the player. We need to think beyond a simple game menu system, where the player points and clicks.

Mouse over

The first thing that you need to completely forget about is moving the mouse pointer over something and having it react, for example, moving the mouse pointer over a menu item and having it highlighted. We need to forget about it, because that is simply not something that happens on an iOS device. Instead of thinking in terms of mouse over followed by a click, we need to think in terms of a finger sliding across the display being lifted. So the action, or response of the menu, happens not when a finger touches the screen, but rather when the finger stops touching the screen. This is called the end of the touch phase. For most small interface items, an aesthetic change in the item is impractical when a finger is touching the item. This is because the item is likely obscured completely by the finger. Instead, consider changing the GUI elsewhere to show this action, and play a sound to accompany it.

Mouse click

With an iOS device, a single finger touch can be equated to a mouse click. But the iOS device is a multi-touch display that can track multiple fingers touching different parts of the display (iOS can detect up to five fingers touching the screen) at the same time, and track them as they move. It would be like having more than one mouse pointer. The kinds of innovative menus that can be created with gestures, goes far beyond what can be created with a mouse click. For example, a player could pinch a part of the menu to select something.

Screen size

The screen size on an iPhone is much smaller than a typical computer display. The idea that your entire menu needs to fit onto the available display area, can really limit the menu information that we want to present in our game. However, if we think of the player as swiping one or two fingers across the display as a way to slide different areas of the menu onto the screen, we suddenly have an unlimited amount of area to display our menu items.

Shake

For some reason, that no one really understands, iOS users really like to shake their devices. So much so that shaking the device has become the standard *gesture* for undo. If we can think of a way to incorporate a shake into our menu system, perhaps as a method to navigate back from a sub menu, it will go a long way to give the player that feeling of being connected to our game.

Since Unity3D has no built-in support for shake, we need a script that detects when an iOS device has been shaken. There are several ways to do this, and just two of them are shown here.

This first script uses a very simple approach, using the accelerometer, to detect iOS device shaking, and makes use of the `Update` function, because there is no expectation that physics will be used. This is shown as follows:

```
// This editor variable can be adjusted to determine
// how vigorously the device needs to be shaken.
// Lower values make the script more sensitive
var threshold : float = 2.0;
// The Update function should be used only if no physics
// will be performed. For example to trigger a menu action.
function Update ()
{
// Get the accelerator value
var l_accel : Vector3 = Input.acceleration;
// Create a filter value to ignore small shakes
var l_filter : float = Mathf.Sqrt(l_accel.x * l_accel.x +
l_accel.y * l_accel.y +
l_accel.z * l_accel.z);
// Perform the desired shake action here
if (l_filter >= threshold)
{
Debug.Log(accel);
}
}
```

The second script, based on similar scripts you can find in the Unity forum or wiki pages, uses a more sophisticated approach, including a low pass filter over time to detect iOS device shaking, and makes use of the `FixedUpdate` function, because there is an expectation that physics will be used, which is shown as follows:

```
// This editor variable can be adjusted to determine
// how vigorously the device needs to be shaken.
// Lower values make the script more sensitive
var threshold = 0.25;
// This editor variable can be adjusted to change
// the rate at which the filtered value converges
// to the input sample.
// Lower values speed up the convergence
// Higher values slow down the convergence
var lowPassWidthSeconds : float = 1.0;
// Private variable for computing the low pass
// filter. They MUST be global to the script to
// work over time
```

```
private var firstTime : Boolean = true;
private var lowPassValue : Vector3 = Vector3.zero;
private var accelerometerUpdateInterval : float = 1.0 / 60.0;
private var lowPassFilterFactor : float = accelerometerUpdateInterval/
lowPassWidthSeconds;
// Low pass filter function to smooth convergence over time
function LowPassFilter(newSample : Vector3)
{
lowPassValue = Vector3.Lerp(lowPassValue,
newSample,
lowPassFilterFactor);
return lowPassValue;
}
// The FixedUpdate function should be used if physics
// will be performed. For example to roll a cube by
// applying a force
function FixedUpdate ()
{
if (firstTime)
{
// initialized with 1st sample
lowPassValue = Input.acceleration;
firstTime = false;
}
else
{
// Get the accelerator value
var acc : Vector3 = Input.acceleration;
// Converge to the sampled value over time
var deltaAcc : Vector3 = acc - LowPassFilter(acc);;
// Perform a desired shake action here
// It can be further filtered based on shaking
// along a specific plane
if (Mathf.Abs(deltaAcc.x)>=threshold)
{
Debug.Log("x: " + deltaAcc.x.ToString());
}
if (Mathf.Abs(deltaAcc.y)>=threshold)
{
Debug.Log("y: " + deltaAcc.y.ToString());
}
if (Mathf.Abs(deltaAcc.z)>=threshold)
{
Debug.Log("z: " + deltaAcc.z.ToString());
}
}
}
```

Think outside of the computer

iOS devices have become known as post PC devices for good reason. When designing a game from the very start, we need to consider all of the ways in which these post PC devices feel like an extension of the game player. Simply presenting the player with a flat menu system of buttons squeezed onto the screen is not going to achieve the feeling of being connected to the game. So, whenever we start developing a game, from the first menu onwards, we need to consider all of the ways in which we can make the game, through the facilities provided by iOS and iOS devices, an extension of the player.

Note

Unity3D does not currently provide support for the Camera or Microphone input, though a developer can create a native plugin to take advantage of those devices.

Dealing with device screen resolutions

Before talking about screen resolutions on iOS devices, it is important to understand the difference between a **pixel** and a **point**.

> ## Note
>
> A **pixel** is a measure of the size of the screen, or the size of the bitmap in an image (icon, button, and so on) that you create.
>
> A **point** is an area on the screen where drawing occurs.

For a long time, a pixel and a point were the same size, and very few game developers were concerned about them being different.

When Apple introduced the retina display in the iPhone 4, everything changed because with the retina display, **a point is two pixels (the resolution doubles in both the x and y directions)**. So, while the number of pixels on the display increased, the number of points did not.

The practical implication of that statement is that every bitmapped image in our user interface suddenly appeared on the retina display, at half the expected size in each direction. The GUI for our games became very small, and in some cases, smaller than the recommended minimum of 44x44 points (the smallest area that is easy for a finger to touch).

Then Apple introduced the iPad. On the iPad, one point is equal to one pixel (for now), but the number of points (and thus pixels) are more than double that of the iPhone and iPod Touch.

Apple, very quickly, moved game developers from having to deal with a single, simple, resolution, to having to deal with a complex mix of screen sizes and point to pixel variations. The current state of the number of points and pixels on iOS devices is summarized in the following table, but the one thing you can be sure of is that this too will change:

Table 1-1 Screen sizes of iOS-based devices:

Device	Portrait pixels	Portrait points	Landscape pixels	Landscape points
iPhone/iPod Touch	320x480	320x480	480x320	480x320
iPad	768x1024	768x1024	1024x768	1024x768
iPhone 4	640x960	320x480	960x640	480x320

For games that made any assumption about the screen size in iOS, these innovations created numerous problems and additional work that would not have been required if games had managed displays in a resolution-independent manner.

So, we need to be careful in our games to never make any assumptions about the number of points, and we need to make sure that wherever pixels are required (like bitmapped images), we

automatically scale to a reasonable size based on the screen size.

Convert to pixels in scripts

The first thing we need to consider is the placement of GUI items on the screen. We should never specify the location of items in pixels; instead, we should specify the location of items as a screen-relative fraction.

Let's examine the placement of a simple **GUITexture** in the center of the display. If we simply use the Unity3D Inspector and position the texture, it would look something like the following screenshot:

And it would look perfectly fine on the iPhone, prior to the introduction of the Retina Display, as shown in the following screenshot:

Unfortunately, on an iPhone 4 with Retina Display, the same game would look like the following screenshot:

Clearly, something is wrong and we need a better way to manage the screen that is resolution-independent.

> **Note**
>
> Unity3D did manage to adjust some of this scene both automatically and in a resolution-independent manner. The camera view was adjusted to show the scene's 3D objects correctly.

We can easily fix the `GUITexture` position by putting a script on the game object to automatically place the texture using a fractional value of the screen size to represent the desired placement on the screen. Our script is shown as follows:

```
// This script will position a GUITexture on the
// screen using numbers that represent its position
// as a fraction of the screen size rather than
// absolute pixel positions. For example to put
// a texture in the center of the screen you would
// use 0/5, 0.5 as its position for halfway across
// and half way down
// This editor value is the fractional position
// on the screen where the GUITexture should be place
var guiPosition : Vector2;
function Start ()
{
// This is the original placement rectangle
var l_rect : Rect = guiTexture.pixelInset;
var l_menuHeight = Screen.height / 6;
var l_menuWidth = Screen.width / 2;
// This converts the fraction values to
// absolute pixel values
guiPosition.x = Screen.width * guiPosition.x;
guiPosition.y = Screen.height * guiPosition.y;
// This makes sure the transform is located at
// position (0,0,0) with a localScale of (0,0,0)
transform.position = Vector3.zero;
transform.localScale = Vector3.zero;
// This places the GUITexture at the correct
// pixel relative position
guiTexture.pixelInset =
Rect ( guiPosition.x - l_rect.width / 2,
guiPosition.y - l_rect.height / 2,
l_rect.width, l_rect.height);
}
```

After applying this script to the `GUITexture` game objects, the game will look like the following screenshot on the Retina Display. Notice that while the GUITextures are in the right relative place, they are not the right size. We need to fix that too:

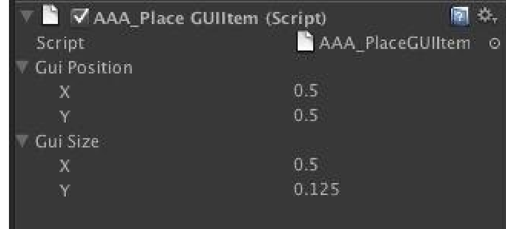
Scale bitmaps in scripts

In order to deal with the size of the GUITextures, we need to modify the script that places them, based on a fractional location on the screen to also scale them based on screen size. Our modified script is shown as follows, and has been changed to accept a new `Vector2` that tells the `GUITexture` what size it should be as a fraction of the total screen size.

Again, we convert that fraction to pixels based on the actual screen size, and apply the new size to the GUITextures `pixelInset` rectangle. We have added a check to make sure the resulting number of pixels is not less than 44 (the number recommended by Apple as the smallest size for a finger to comfortably touch) to ensure we don't end up with tiny textures on the screen. The changes to the script are shown as follows:

```
// This script will position a GUITexture on the
// screen using numbers that represent its position
// as a fraction of the screen size rather than
// absolute pixel positions. For example to put
// a texture in the center of the screen you would
// use 0/5, 0.5 as its position for halfway across
// and half way down.
// In addition, the size of the GUITexture is provided as a
// fraction so that an image that we want to be half the
// size of the screen will have a size of 0.5, 0.5
// This editor value is the fractional position
// on the screen where the GUITexture should be place
var guiSize : Vector2;
function Start ()
{
// This converts the fraction values to
// absolute pixel values but uses the original size if
// the one specified is too small
if (Screen.width * guiSize.x < 44.0 ||
Screen.height * guiSize.y < 44.0)
{
guiSize.x = guiTexture.pixelInset.width;
guiSize.y = guiTexture.pixelInset.height;
}
else
{
guiSize.x = Screen.width * guiSize.x;
guiSize.y = Screen.height * guiSize.y;
}
// This places the GUITexture at the correct
// pixel relative position and scales it to the
// correct size
guiTexture.pixelInset =
Rect ( guiPosition.x - guiSize.x / 2 ,
guiPosition.y - guiSize.y / 2 ,
guiSize.x, guiSize.y);
}
```

A typical `GUITexture` now has the properties shown in the following editor:

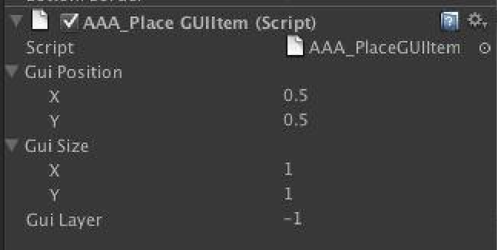


And the display, on a Retina Display, now looks like the following screenshot. An interesting thing has happened. All of the GUITextures are scaled and positioned correctly, but the order in which images are layered on the screen is not correct, and some GUITextures are being covered by what is supposed to be a background image, shown as follows:

So, we need to do one final thing in our script, and that is to set the layer, relative to the other textures, that we want each texture to be on. This is controlled by the Z attribute of the `GUITexture` game object. Our final script that implements layering is shown as follows:

```
// This editor value is the layer
// on the screen where the GUITexture should be place
var guiLayer : float;
function Start ()
{
// This makes sure the transform is located at
// position (0,0,layer) with a localScale of (0,0,0)
transform.position = Vector3(0,0,guiLayer);
}
```

The following screenshot shows the settings in the editor:



And this final screenshot shows the final screen image with everything positioned, scaled, and layered in a resolution-independent way that will work on all iOS devices:

Testing iOS games in the editor

Games developed for iOS devices are played using gestures, not mouse clicks. This means that game testing becomes quite a long and cumbersome process of continually building and deploying the game on a real device, or messing around with a real device and Unity Remote to test even the minor changes.

Wouldn't if be nice if you could run your game, for the most part, on the desktop computer, and even in the Unity editor to test it without the need to constantly go through the process of building the game.

It turns out, that for many things, it is possible to do that with a little bit of extra scripting that makes our game responsive to not only mobile device input, but also keyboard and mouse input.

This is a tremendous time saver during the game development cycle and can also be used to create an iOS game that also runs as a desktop application on a Mac and a PC.

Simulating the accelerometer

One of the easiest things to simulate is the accelerometer. Many games that use device tilting as the primary method for gameplay can simply use the arrow keys/joystick to allow the game to be tested in the editor.

The following script fragment demonstrates how to get accelerometer input, but it will only work if you use a physical iOS device:

```
function FixedUpdate ()
{
// This variable will contain the accelerometer values
var dir : Vector3 = Vector3.zero;
// get accelerometer input
dir.x = -Input.acceleration.y;
dir.z = Input.acceleration.x;
// Do something with the values here
//
}
}
```

One approach to simulating accelerometer input is to expose a flag in the editor, and set or unset that flag depending on whether or not you want to check for arrow key/joystick input or the accelerometer input. The advantage of using a flag is that the script will always work when iOS is selected in your **Build Settings**. The disadvantage is that it requires a check at runtime, and so it introduces a small delay in the execution of your game. The updated script that demonstrates using an editor flag is shown as follows:

```
// This variable will appear in the Editor as a checkbox
// If you set it, arrow key / joystick input will be used
// If you unset it, accelerometer input will be used
public var simulateAccelerometer:boolean = false;
// Other functions will appear here
//
// Always apply physics forces in FixedUpdate
function FixedUpdate ()
{
// This variable will contain either the arrow key /
// joystick values or the accelerometer values
var dir : Vector3 = Vector3.zero;
// The Simulate Accelerometer property must be set or
// unset in the editor to change the behavior
if (simulateAccelerometer)
{
// use arrow key / joystick input instead of
// accelerometer
dir.x = Input.GetAxis("Horizontal");
dir.z = Input.GetAxis("Vertical");
}
```

```
else
{
// use accelerometer input instead of arrow key /
// joystick input
dir.x = -Input.acceleration.y;
dir.z = Input.acceleration.x;
}
// Do something with the values here
//
}
```

Another approach to simulating accelerometer input is to make use of **platform-dependent compilation**.

> # Note
>
> Platform-dependent compilation is essentially the same as conditional compilation, and it allows you to divide your scripts in a way that sections of code are compiled and executed only on a specific hardware platform.

The advantage of using platform-dependent compilation is that it does not require any check at runtime, and so it does not delay the execution of your game. The disadvantage is that you need to change the platform that is selected in your **Build Settings**, when you want to change the behavior of your game. The updated script that demonstrates using platform-dependent compilation is shown as follows.

If you set your **Build Settings** platform to iOS and run this script using a device, it will use the accelerometer. If you set your platform in the **Build Settings** to **PC** and **Mac Standalone**, it will use the arrow keys/joystick, which is shown as follows:

```
// Other functions will appear here
//
// Always apply physics forces in FixedUpdate
function FixedUpdate ()
{
// This variable will contain either the arrow key /
// joystick values or the accelerometer values
// Since it is a Vector3 it will have x, y, and z
// components. For example dir.x, dir.y, and dir.z
var dir : Vector3 = Vector3.zero;
// use arrow key / joystick input instead of accelerometer
#if !UNITY_IPHONE
dir.x = Input.GetAxis("Horizontal");
dir.z = Input.GetAxis("Vertical");
#endif
// use accelerometer
#if UNITY_IPHONE
dir.x = -Input.acceleration.y;
dir.z = Input.acceleration.x;
#endif
// Do something with the values here
// …
}
```

In order to decide which method to use, we need to consider our game, our development environment, and our personal preference and style. There is no correct way and there is no incorrect way; there is only the way we prefer, or our team prefers, to manage developing our game.

Using shaders to solve real-world problems

Shaders are a crucial part of 3D rendering. In Unity3D, lights, projectors, normals, tangents, and textures are all just data that will be passed to the shader. Once it has this data, the shader takes care of what is sent to the camera.

These examples illustrate the power of understanding the usage of a shader:

- If a shader simply maps a texture to geometry without using the scene lighting, it will return geometry that has brightness identical to that of the texture file. This means that as the lighting changes in a scene, say, to a dark night or a bright day, the geometry would look the same.
- If geometry has information about its normals, but the shader does not cull faces or use bump-mapping, then the normal will only be useful for lighting.
- If a texture map contains an alpha channel, but if the shader is not an alpha channel, the transparency will be ignored and a solid object will be rendered.

Before we can solve problems with shaders, we need to understand what properties of a shader are used to achieve the desired effect.

Shading/Lighting models

Shading models provide an approximation of smoothed normals (this essentially means that the visible faces of objects are made to appear to have smooth, rather than sharp, edges). Without them, all models would appear to be composed of many tiny flat polygons.

When importing meshes, Unity will use normals imported from the file but if the importer settings are changed, Unity can recalculate normals when it imports the meshes.

It is important to understand which normal approximations result in which changes in the appearance of the product geometry.

Note

Please note that while shading, models are often categorized as an aspect of the geometry of an object, they are still very important for achieving the desired effect. The following image, from the Cheetah3D modeling tool, shows the results of using the different shader models, so that we can see the differing levels of smoothing.

Phong

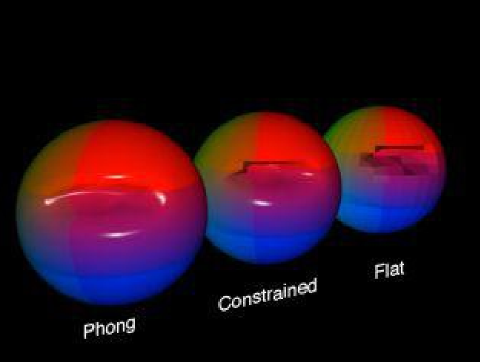
This model causes all faces and edges of the geometry to be smoothed. It is ideal for organic-looking art, such as character design. In the following image, the phong model is shown in the sphere to the far left.

Constrained (Blinn-Phong)

This model causes the edges to be crisp, while the faces appear smooth. It is desirable for use in most models. The crisp-edge effect is the result of an angle exceeding the constrained bounds, set to use the Phong shading (usually vertices are shaded via Phong, and in between pixels use less-expensive shading) at which point, Gouraud shading is used. In the following image, the **Constrained** model is shown in the middle sphere.

Flat (Gouraud)

The **Flat** shader model appears blocky and faceted. It is ideal for shapes like gems where sharp, faceted edges are desirable. In the following image, the **Flat** model is shown in the sphere to the far right:


Phong
Constrained
Flat

Applying shaders to solve problems

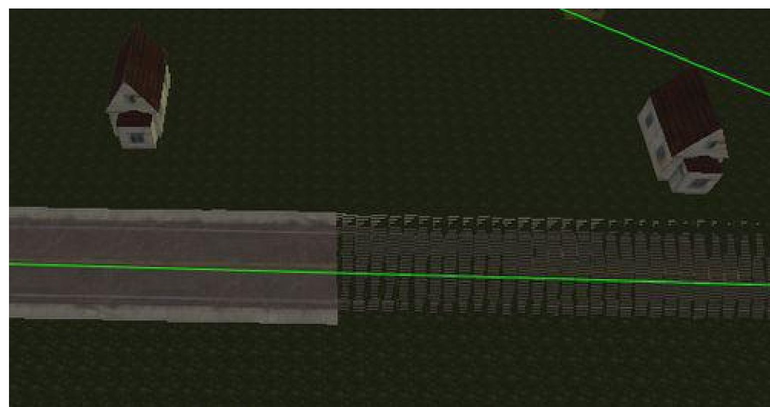
Now that we understand the properties of shaders, we can look at the real-world problems that can be solved by using a shader. The question that we need to consider is, should the problem be solved using geometry or shading. Often, there are solutions that can be solved either way; and in the end, we need to make sure that we are not using a shader simply because it is something we know, but rather because it makes sense to use a shader rather than using geometry.

Z-Fighting

Z-Fighting is a really good example of a problem that can be solved with geometry, but for which a shader solution is the better solution.

Z-Fighting occurs when two meshes are, as the name implies, on the same (or different, but with a small offset) Z-plane such that, depending on the distance and camera angle, different parts of each are rendered in such a way that they overlap each other (typically in rows or bands). This is difficult to say, but easy to see.

In the following image, the road segment on the right is Z-Fighting with the Terrain, because they are both on the same Z-plane.



We could use geometry to solve the Z-Fighting problem. By that, I mean we could position the road slightly above the terrain, and the distance between the objects would reduce or eliminate the Z-Fighting, but our player, if they got close enough to the objects, may notice the gap between the planes. Another option would be to make our road segments out of a cube and make it thick enough that it can sit above the terrain, perhaps even extending into the terrain a bit. Because it has sides, it would be less noticeable (other than its height) that the objects were not on the same plane. Of course, doing this would add a lot of triangles to our scene, since each road segment would contain a lot more faces this could dramatically impact rendering performance.

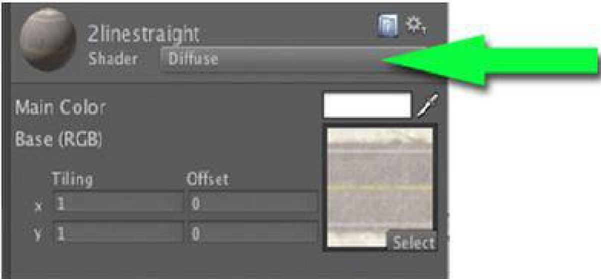
Or we could use a copy of the shader that draws the road at an `Offset` of `-1, -1`, so that whenever the two textures are drawn, the road is always drawn last.

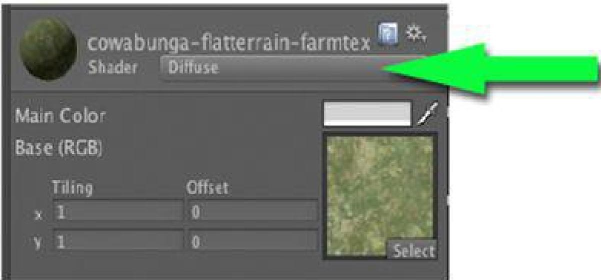
The following is a listing of our modified copy of the Normal Diffuse shader. Notice that we have

changed the name of the `Shader` to `DiffuseOffset`, the `Offset` to `-1, -1`, and also the `Fallback` to `VertexLitOffset`:

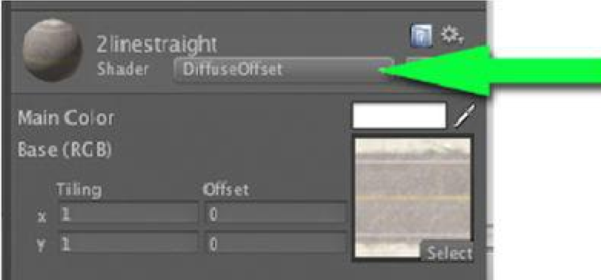
```
Shader "DiffuseOffset"
{
Properties
{
_Color ("Main Color", Color) = (1,1,1,1)
_MainTex ("Base (RGB)", 2D) = "white" {}
}
SubShader
{
Offset -1, -1
Tags { "RenderType"="Opaque" }
LOD 200
CGPROGRAM
#pragma surface surf Lambert
sampler2D _MainTex;
float4 _Color;
struct Input
{
float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutput o)
{
half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
o.Albedo = c.rgb;
o.Alpha = c.a;
}
ENDCG
}
Fallback "VertexLitOffset"
}
```

While examining these changes, the first thing we see is the name change. Notice in the following screenshot that both textures use the **Shader** named **Diffuse:**

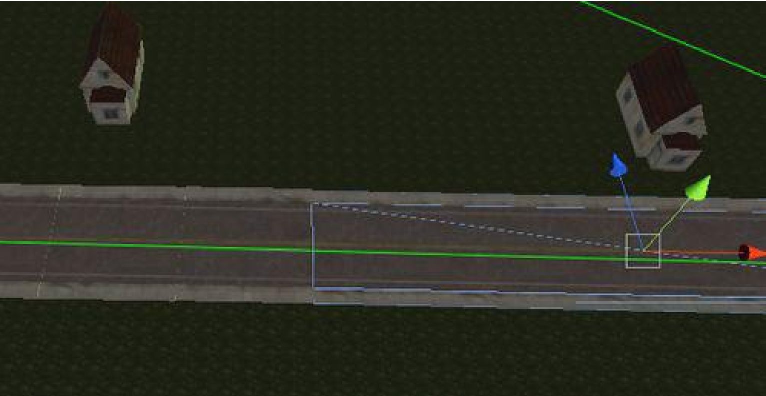




Notice in the following screenshot that we have changed the **Shader** for the road to be **DiffuseOffset:**



The next change is the Offset, and as we can see in the following image, changing the Offset has indeed fixed the Z-Fighting problem. Note that this is a Z-buffer Offset, not a texture Offset:



The final change is the Fallback shader. This is needed because on a platform like iOS, some iOS

devices will use the Fallback shader. This means that not only do we need to make an Offset shader for the Normal-Diffuse shader, but we also need to make a Fallback shader for the Normal-VertexLit shader. Making the changes to the second shader is left as an exercise for the reader with the big hint being the first SubShader needs an offset of -1, -1.

Note

In Unity3D version 2, the values for the Offset setting could be passed in from the Unity3D editor as variables. It appears that in Unity version 3, they must be hardcoded, so if you need multiple layers of Z-Fighting shaders, you need to make multiple copies of the shader.

Back face rendering

A triangle has two faces, namely, the Normal face and the Back face. By default, Unity3D shaders cull (remove or do not render/shade) the Back face. While this is what we want for most materials, there are exceptions, for example, glass and water.

Again, we can solve this problem with geometry by using two-sided planes to fill in, for example, windows. Alternatively, we can use a shader that does not cull the Back face, such that the material is applied to both the Normal face and the Back face. This instantly gives us a nice sheet of glass.

The following image shows the same model of a restaurant that uses a single-sided plane for the windows, the first image shows the standard transparent shader that culls the Back face and the second image shows the same model using a shader that does not cull the Back face for the glass material. Similar shaders are used for vegetation or fences, where the objects are made up of simple decorative planes, shown as follows:

The following listing shows the iPhone Standard Asset shader that has been modified to remove Back face culling; it has also been renamed. A similar change could be made to the desktop version of the shader:

```
Shader "iPhone/Transparent/Vertex Color 2-Sided"
{
Properties
{
_Color ("Main Color", Color) = (1,1,1,1)
_SpecColor ("Spec Color", Color) = (1,1,1,0)
_Emission ("Emmisive Color", Color) = (0,0,0,0)
_Shininess ("Shininess", Range (0.1, 1)) = 0.7
_MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}
Category
{
Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
ZWrite Off
Alphatest Greater 0
Blend SrcAlpha OneMinusSrcAlpha
SubShader
{
```

```
Material
{
Diffuse [_Color]
Ambient [_Color]
Shininess [_Shininess]
Specular [_SpecColor]
Emission [_Emission]
}
Pass
{
ColorMaterial AmbientAndDiffuse
Lighting Off
SeparateSpecular On
Cull Off
SetTexture [_MainTex]
{
Combine texture * primary, texture * primary
}
SetTexture [_MainTex]
{
constantColor [_Color]
Combine previous * constant DOUBLE, previous * constant
}
}
}
}
}
```

Organizing game objects for easy asset sharing

When we develop a successful game, we are more than likely going to want to reuse some of the components of that game in either a follow up game or in a similar game. If you look at the big names in the gaming industry, you will discover that they share game assets between companies.

Unity3D has an excellent asset server, but as a small developer, we need a lighter-weight mechanism to share our game assets. The key to this is organizing assets in our project, so that related items are, as much as possible, kept together.

The steps required to export an asset, or a group of assets, into a Unity Package that can be imported successfully into another project are as follows:

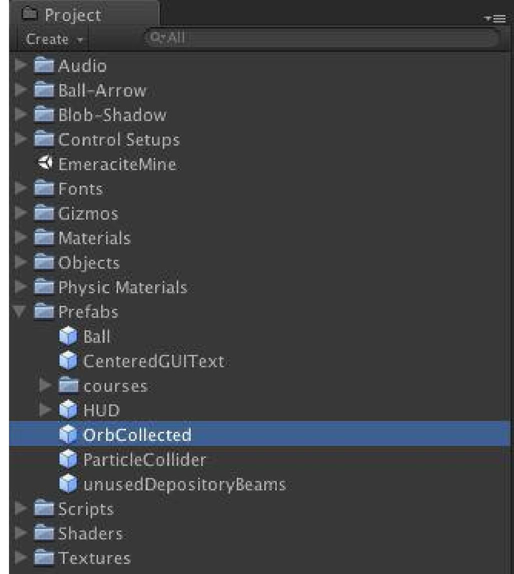
1. Select the **Asset Prefab** in the editor **Project** hierarchy.
2. From the **Assets** menu, choose **Select Dependencies**.
3. From the **Assets** menu, choose **Export Package**.

The first thing we notice when we do this is that selecting the dependencies will result in all kinds of items being selected all over the **Project** hierarchy. It can be a real mess, especially in sample code.

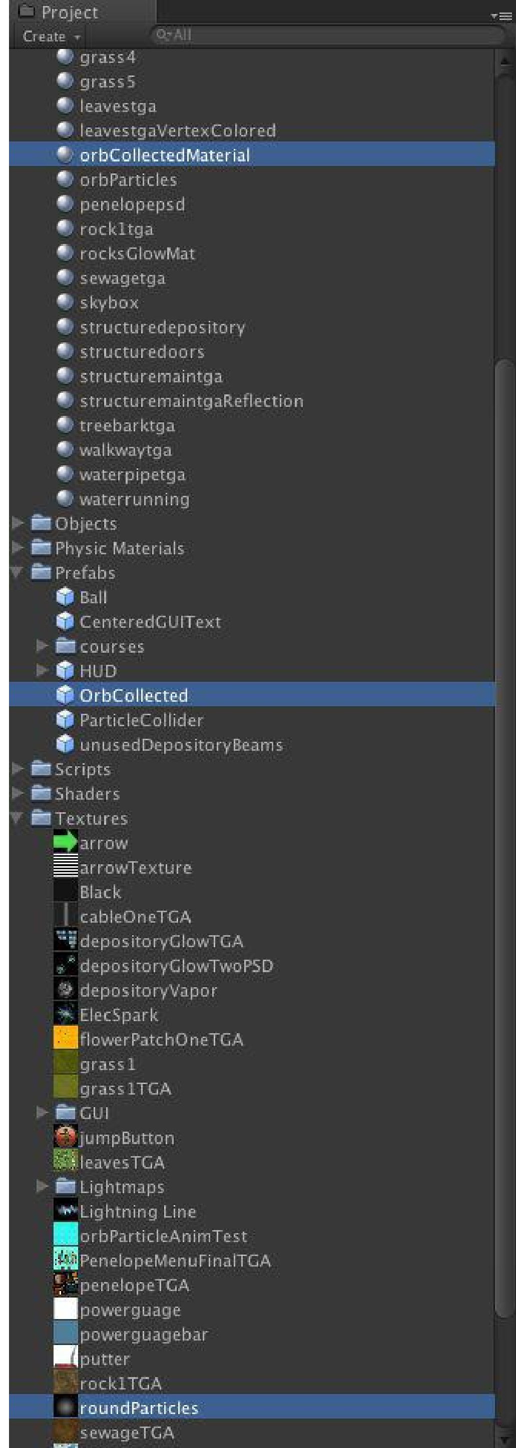
The second thing we notice is that when we import the Package into a new game, the same mess will be re-created, and we may also have conflicts where different packages contain the same items in the same places, and we end up overwriting, for example, one texture with another.

All these complications can be avoided by creating an organized hierarchy in the **Project** hierarchy for related prefabs. This is best illustrated with an example.

In the following unorganized project, we first selected the prefab named **OrbCollected**, and then from the **Assets** menu, we chose **Select Dependencies:**



When the dependencies were selected, as shown in the following screenshot, the **Project** window had to open multiple sub folders to display all the items upon which the prefab depends:

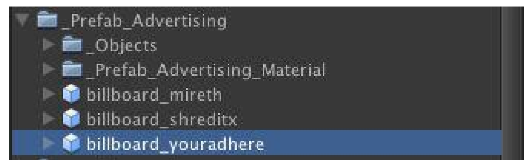


In the same way, after we export this prefab, when we import this Asset Package into a new project,

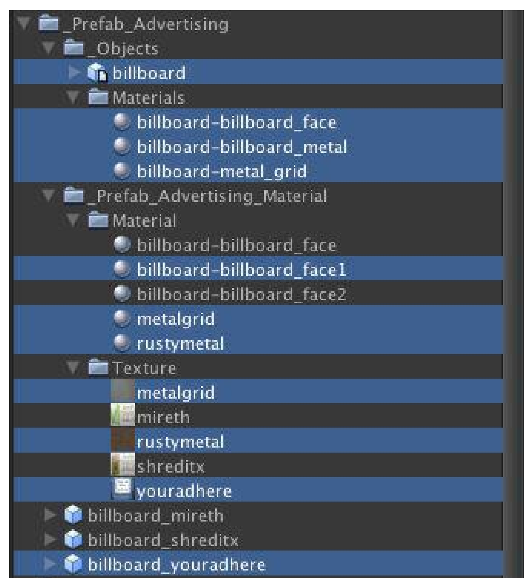
the same unorganized hierarchy will be created.

Instead, we can organize the objects, textures, scripts, and so on that we need for a particular prefab into a structure, such that everything is together. Again, this is best illustrated with an example.

In the organized project, as shown in the following screenshot, we first selected the prefab named **billboard_youradhere**, and then from the **Assets** menu, we chose **Select Dependencies:**



When the dependencies were selected, the **Project** window had to display only the contents of the folder named **_Prefab_Advertising**, to select all of the items upon which the **billboard_youradhere** depends:



Optimizing the hierarchy of an existing project, after a scene has been created, can take a lot of time. Therefore, this is something that needs to be done from the start of a new project. Of course in the real world, we may want to share materials, objects, or scripts, but it's important that when we make the decision to do that, we understand the complexity it introduces to the project hierarchy, and we make an informed decision with respect to having a second texture versus a shared texture. If we do plan to use shared items, we will organize our project by creating folders specifically for all the shared items, so that it is very clear where they are found and that they are shared. It also makes it clear to anyone reviewing our project hierarchy that we know about and intend to share assets, so that they can ask us our (no doubt very good) reasons for deciding to add complexity to the clear benefits of sharing.

Summary

In this chapter, we have covered the following:

- Advanced iOS gaming concepts that will allow us to develop engaging games that take advantage of unique iOS device features that run on all iOS devices, regardless of display differences.
- How to dramatically reduce game development time by creating scripts that work both in the Unity3D editor and on iOS devices.
- The basic properties of shaders as well as when and how to use shaders, rather than geometry, to solve problems on iOS and achieve better performance as a result.
- How creating organized Unity3D Prefab objects allows us to easily reuse game assets in another game. By taking some care in the beginning, we will make our games both entertaining and available across the widest array of iOS devices.

Chapter 4. Flyby Background

Our game menu system is a critical component for drawing players into our game. One of the best ways to make the menu system engaging is to have the player fly through one or more of the game scenes as a backdrop to the menu system.

Because mobile devices are limited in so many ways, it is important to keep the flyby as efficient as possible. We need to develop a menu with a flyby background using the real-world compromises required for a mobile computing platform to achieve a solid introduction to our game.

Some mobile games have no interface or menu system at all, and it's important to recognize that a solid interface and menu system significantly increase the player's experience in a mobile game.

In this chapter, we will learn the following:

- How to set up a background scene that gives the player a feel for the expansive nature of our game?
- How to create a path that a camera can follow?
- How to create a main menu that overlays the camera, flying through our scene?
- How to save time by creating a menu that can be tested as easily in the editor as on an iOS device?
- How to set up Unity3D for iOS build settings to create an App that will run on all iOS devices?
- How to deploy an iOS app on multiple devices?

Set up a background scene

Typically, the background scene for the menu system will also be a scene from the game. But because the player will not be interacting with the menu's background scene, they will not be able to see it from all angles, jump on objects, and so on. It is possible to strip out many of the components from the scene to optimize it for use as a menu background.

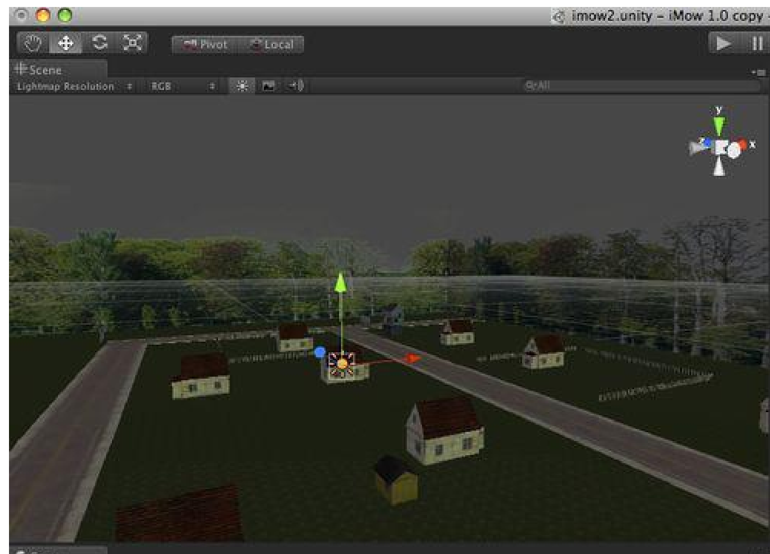
Another option is to create a brand new background scene that incorporates game assets from multiple game levels into a single menu background to give the player a full picture of the scope of our game. Again, we can achieve this by being selective about the game assets that we use and stripping out all but the most essential visual components of those assets.

The background scene can be a teaser, it can include Easter Egg hints, it can contain character cameos from other games, and it can contain parts of other games.

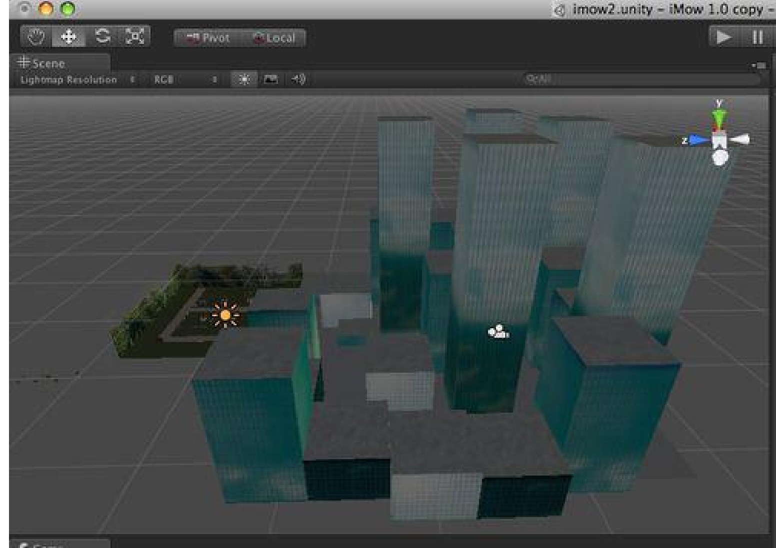
Finally, the background scene can be a selling opportunity. It can contain advertising or it can contain objects that the player may be interested in purchasing, using game purchase.

The contents of the menu background scene are limited only by what we can imagine.

In the following image, we show the original first level of a game that we have selected to be used as the background scene for our game:



The scene is adequate and we could use it as the background for our game menu, but we want to give the game player a better idea, right on the main menu, of just how large the game is. So instead, we add some buildings from a second level (we could do more; we could make our menu level huge, but this should do for now), and come up with the new, much larger scene, as shown in the following screenshot to use as the background for our main menu:


imow2.unity - iMow 1.0 copy -
Pivot
Local
Scene
Lightmap Resolution
RGB
y
z

Set up the camera path

Once we have decided on the final scene that we want to use as the background for our main menu, we need to decide on the path that the camera will follow as it moves through the scene.

Note

While it is not a requirement that the camera moves along a path, it adds interest to the menu, and so we will continue with that menu theme for our game.

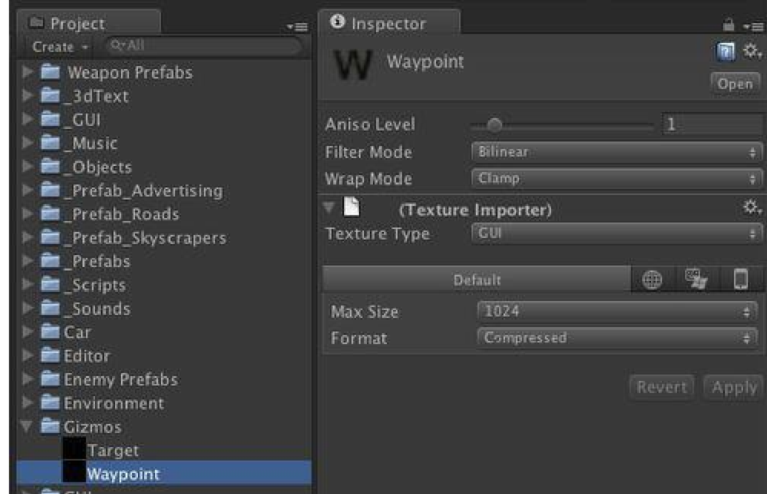
This is an opportunity for us to tease the player, since it may be some time before they arrive at a specific part of the menu scene during actual gameplay. We can flyby Easter Eggs or we can flyby specific objectives. Again, this is the part of the game where we get to be creative and draw the player into the game.

If we want to, we can even defy the physics of the game and fly through things, so that we don't give away the secret of how to achieve a game objective or how to obtain an Easter Egg, but we do let the player know that they are there. We need to be careful to do this in such a way as to fade through scenes rather than clip through the geometry, so that we don't disturb the immersive feel of the game in order to hide things from the player's view.

To set up a path, the first thing we need to do is create a **gizmo**. A gizmo is an image that appears in the Unity3D editor at the position of a game object's transformation, but otherwise will not be visible in the game. As a game designer, we use gizmos to arrange these game objects graphically in the editor, rather than having to find each object in the game hierarchy, and enter x, y, and z values to position the transformation. Gizmos, while not necessary, are very useful because they allow us to create a visual representation of an otherwise empty game object in the Unity3D editor. To create a gizmo, you need two things:

- A gizmo image
- A gizmo script

The following is an example of a gizmo image. It can be anything you want. In this case, we have simply chosen an image of the letter W for waypoint. The image file is named `Waypoint.tif`, and it has been imported into Unity3D as a standard `GUITexture` as follows:



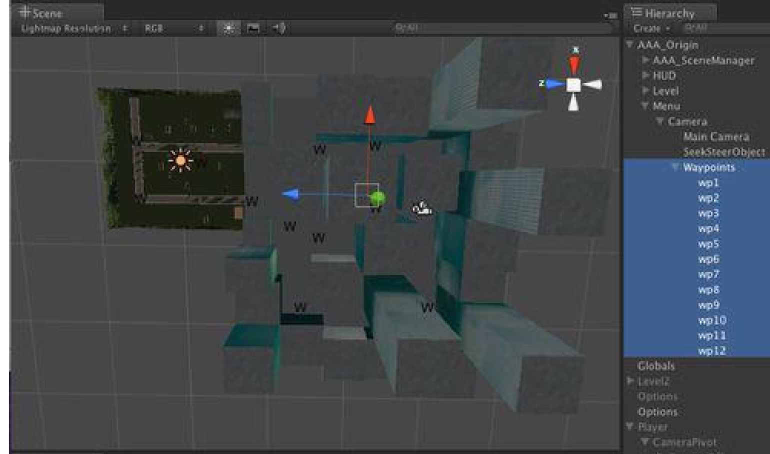
The following is an example of a gizmo script that draws a waypoint gizmo image at the location of the game object transform to which it is attached:

```
// Draw the waypoint gizmo at the Game Object's
// transform position.
// You can use any image that you want
// This gizmo will be pickable which means you
// can click on it in the editor to select
// the attached game object
function OnDrawGizmos ()
{
//The image must be in Assets/Gizmos the size of the image
//is how large it will be drawn in the scene view
Gizmos.DrawIcon (transform.position, "Waypoint.tif");
}
```

To use a waypoint in the editor, we need to do the following:

- Create an empty game object
- Attach the `waypoint.js` script to the game object

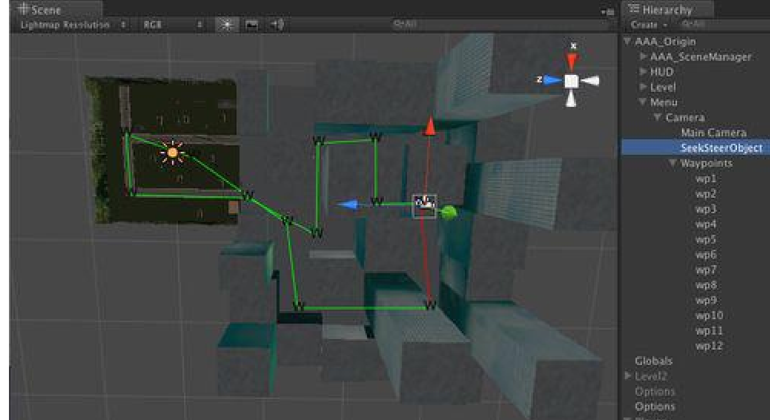
The following image shows our example level with a number of waypoints added to define the path that we want the camera to follow for our menu flyby. Everywhere you see the letter W in this image, denotes a waypoint along the path:



Adding waypoints and drawing a gizmo at a waypoint's location in the editor is helpful, but one more thing that we can do to make the order of the waypoints clear is to draw a line between the waypoints. So, for example, we would draw a line from wp1 to wp2, from wp2 to wp3, and so on. The following script fragment, which is part of the `SeekSteer` script, shows how to draw the lines between the waypoints:

```
// draws a line from waypoint to waypoint
public void OnDrawGizmos()
{
Vector3 l_prevWaypointPosition;
Vector3 l_currentWaypointPosition;
// Choose a color, it can be any color you like
Gizmos.color = Color.red;
// Draws a line between the last waypoint
// and the first waypoint
l_prevWaypointPosition =
waypoints[waypoints.Length-1].position;
l_currentWaypointPosition = waypoints[0].position;
Gizmos.DrawLine(l_prevWaypointPosition,
l_currentWaypointPosition);
// Choose another color, it can be any color you like
// except the first color
Gizmos.color = Color.green;
// For each remaining waypoint, in the waypoints array
// draw a line between the two points
for (int i=1;i < waypoints.Length;i++)
{
l_currentWaypointPosition = waypoints[i].position;
l_prevWaypointPosition = waypoints[i-1].position;
Gizmos.DrawLine(l_prevWaypointPosition,
l_currentWaypointPosition);
}
}
```

The following screenshot shows the new scene with lines being drawn between the waypoints. It's easy to identify the first and last waypoint, because they are the points with the red lines between them:



Once the waypoints have been set up, we need something to follow them. There are literally dozens of ways that we can use to follow a path, and many kinds of paths that we can follow. In fact, there are complete Unity3D projects devoted to path finding.

In this case, we have chosen to use the `SteerSeaker` method of following a path. The `SteerSeaker` creates an array (or list) of all the waypoints, and moves from one waypoint to the next in the same amount of time. In order to keep the time between waypoints constant, the `SteerSeaker` speeds up or slows down based on the distance between the waypoints, which makes it easy for us to predict the total time it will take to follow our path and create sections of both slow and fast movement. The rest of the `SteerSeaker` script (remember we looked at the previous image that draws lines between the waypoints) is shown as follows:

Note

This script is written in C# rather than JavaScript. While many people new to Unity3D prefer to work in either JavaScript or C#, it's important that we become familiar with both scripting languages, so that we can take advantage of all the open source resources available in the Unity3D community. While we don't need to be able to program in both languages, we do need to be able to read both languages

```
// SeekSteer.cs
// Based on the original SeekSteer by Matthew Hughes
// -- 19 April 2009
// -- Uploaded to Unify Community Wiki on 19 April 2009
// -- URL: http://www.unifycommunity.com/wiki/index.php?title=SeekSteer
//
// Changes by BurningThumb Software
// -- March 2010
//
using UnityEngine;
using System.Collections;
public class SeekSteer : MonoBehaviour
{
// This is the array of waypoints
public Transform[] waypoints;
// This is the radius, in meteres of a waypoint
public float waypointRadius = 1.5f;
```

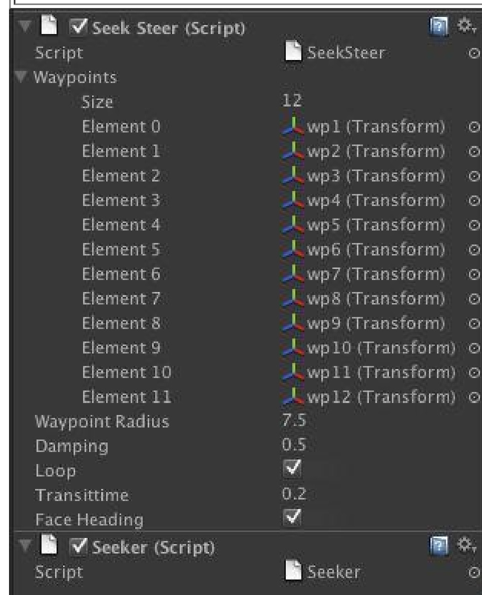
```
// Damping is used to limit the rate at which
// the object turns towards the next waypoint.
// Smaller numbers slow down turns, larger
// numbers speed up turns
public float damping = 0.1f;
// Set loop to true if the object loops
// continuously around the waypoints and
// to false if the object only goes around
// one time
public bool loop = false;
// The time between waypoints is constant
// so the object will speed up or slow down to
// achieve this time regardless of the distance
// between the points
public float transittime = 2.0f;
// Set faceHeading to true to make the object
// turn to face the forward direction and to
// false to not turn to face the forward
// direction
public bool faceHeading = true;
// The current heading of the object
private Vector3 currentHeading;
// The desired heading of the object
private Vector3 targetHeading;
// The array index of the waypoint that the
// object is heading toward
private int targetwaypoint;
// A reference to the transform of the object
// used to speed up the script by caching the
// reference which is used many times
private Transform thisTransform;
// If the object has a rigid body then this
// is a reference to the rigid body of the object
// used to speed up the script by caching the
// reference which is used several times
private Rigidbody thisRigidbody;
// Use this for initialization
protected void Start ()
{
// If the waypoints array is empty this script
// logs a message and disables itself. You need
// to add waypoints to the array in the Unity3D
// editor
if (waypoints.Length <= 0)
{
Debug.Log("No waypoints on "+name);
enabled = false;
}
// Cache a reference to the transform to speed
// up the execution of the script
thisTransform = transform;
// Set the current heading to be forward
currentHeading = thisTransform.forward;
// The first target waypoint, is the first
// one in the array
targetwaypoint = 0;
// Cache a reference to the attached Rigidbody to
// speed up execution of the script. If
// no Rigidbody is attached this will be null
thisRigidbody = rigidbody;
```

```
}
// calculates a new heading. This is done in
// fixed update just in case there is a
// Rigidbody attached and physics are
// involved
protected void FixedUpdate ()
{
// A simple Lerp with damping to adjust
// the heading towards the current target
// waypoint
targetHeading = waypoints[targetwaypoint].position -
thisTransform.position;
currentHeading = Vector3.Lerp(currentHeading,
targetHeading,
damping * Time.deltaTime);
}
// moves us along current heading
protected void Update()
{
// If a Rigidbody is attached then
// physics is in use so add velocity
if (thisRigidbody)
{
thisRigidbody.velocity = currentHeading *
transittime;
}
// Otherwise set the position directly
else
{
thisTransform.position = thisTransform.position +
(currentHeading *
Time.deltaTime *
transittime);
}
// If the object needs to face the heading,
// make it look that way
if (faceHeading)
{
thisTransform.LookAt(thisTransform.position +
currentHeading);
}
// Check to see if the object is inside the
// waypoint radius
if (Vector3.Distance(thisTransform.position,
waypoints[targetwaypoint].position) <=
waypointRadius)
{
// Add one to the target waypoint to select
// the next waypoint
targetwaypoint++;
// if the next waypoint is past
// the end of the array
if(targetwaypoint>=waypoints.Length)
{
// set it back to the beginning
targetwaypoint = 0;
// If the object is only supposed
// to transit the waypoints one
// time then disables the script
if (!loop)
```

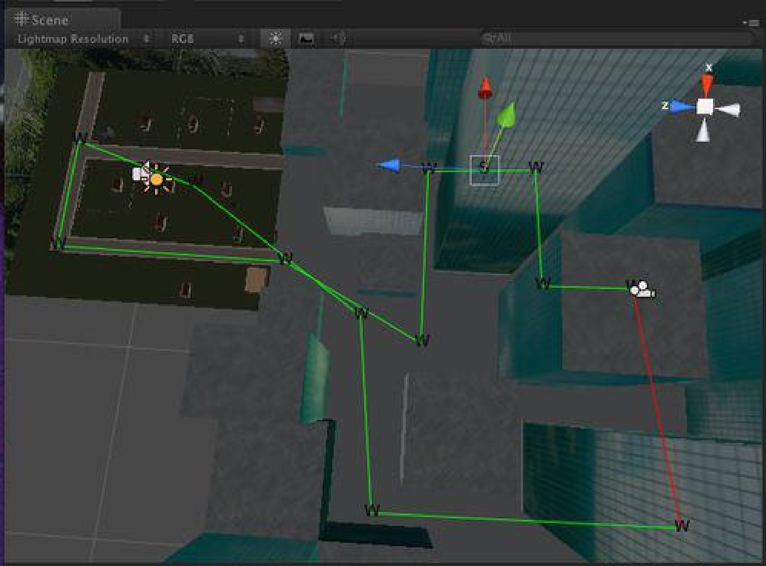
```
{
enabled = false;
}
}
}
}
```

With this final script, we create the `SteerSeeker` game object and attach not only the `SteerSeaker` script, but also a gizmo script to display the letter S. This is done so that we can see the position of the game object in the Unity3D editor. The following image shows the `SteerSeaker` object settings in the Unity3D editor:

> ## Note
>
> The variables declared as public in the script are the ones that will appear in the Unity3D editor.



The following image shows its position on the waypoint path, as we run the game in the editor:

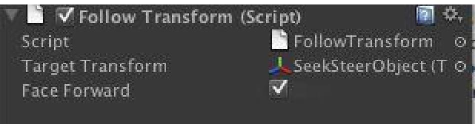


Finally, we need to have the main camera follow the `SteerSeaker` object as it moves along the waypoint path.

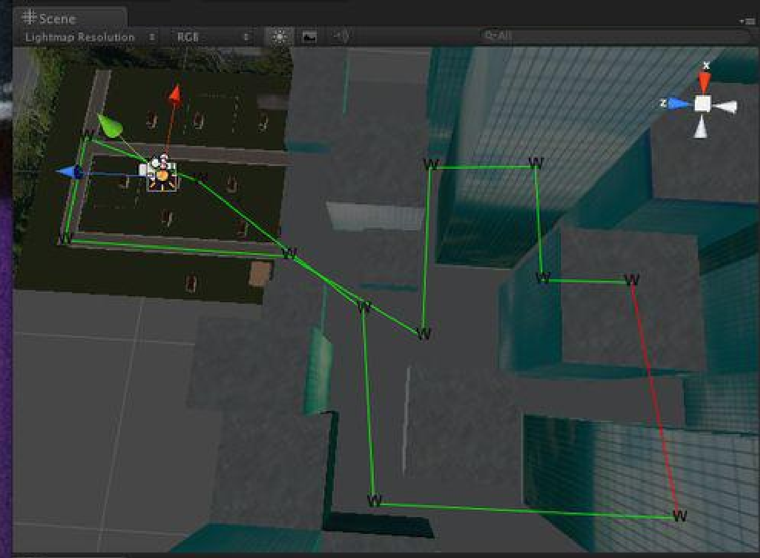
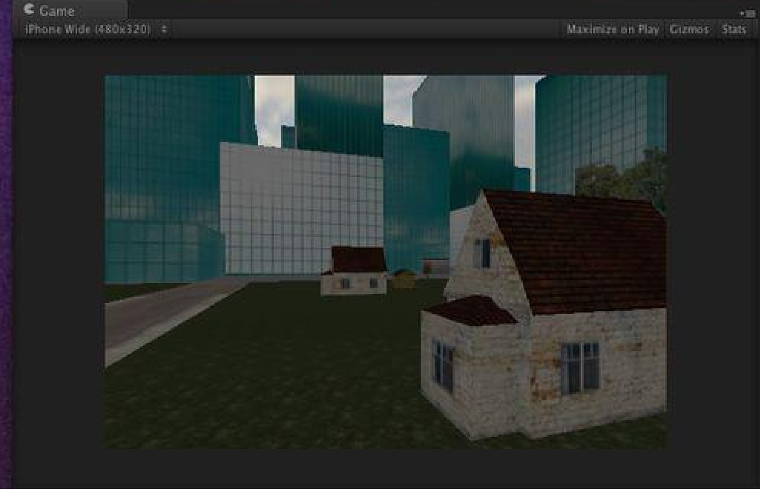
Note

The advantage of using a script, instead of a simple timeline, is that we can create more dynamic camera movement, for example, a rollercoaster effect where the camera changes angles as it moves, or a script that *gobbles* up jewels along the path as it moves. It's important to understand the concept of using a script and its benefits, rather than simply looking at what this specific example does. Often, as in this case, example code is kept simple to convey the concept or idea, and is not meant to be the final solution that we would deploy in a game.

This is done by attaching the `FollowTransform` script from the iPhone Standard Assets to the main camera, while assigning the **SeekSteerObject** to the **Target Transform** and checking the box **Face Forward** on that script. This is shown as follows:



The output is shown in the following image:


Scene
Lightmap Resolution
RGB
x
z

Game
iPhone Wide (480x320)
Maximize on Play
Gizmos
Stats

Set up the main menu

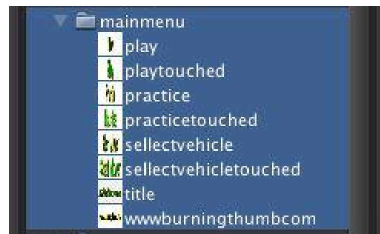
Now that our game has an interesting flyby background, we need a menu that allows the player to select some options. Because we are using iOS, we want the menu to have a touch interface where appropriate. First, though, we need to decide what options we want to present on the menu.

We will put the following things on our main menu:

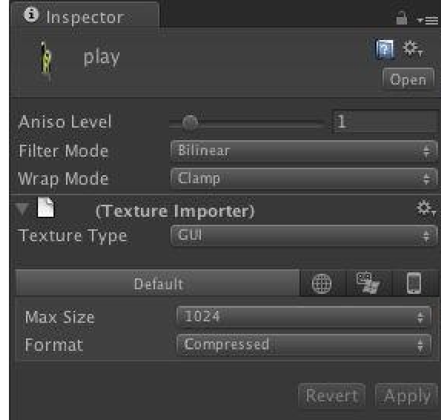
- The title of our game
- The URL of our website
- An option to choose from various available player avatars
- An option to practice playing
- And an option to play the game

We have also decided to use `GUITextures` for our menu items, so that we can use any kind of graphics that we want to represent these options. It is often the case that we would choose to use icons rather than text to represent these choices. However, for our prototype, we are going to use text that could easily be replaced with icons before releasing the game.

Because we will be touching the menu items, we need to create an image that will be displayed both when the menu item is touched and when it is not touched. The following screenshot shows the textures that we will use:



The following screenshot shows an example of how they will be imported into Unity3D using the built-in **Texture Importer**, where the **Texture Type** has been set to **GUI:**



Since each menu item will have a touched and untouched image, we need to create both images. The following image shows the untouched image for the **Play** menu item:

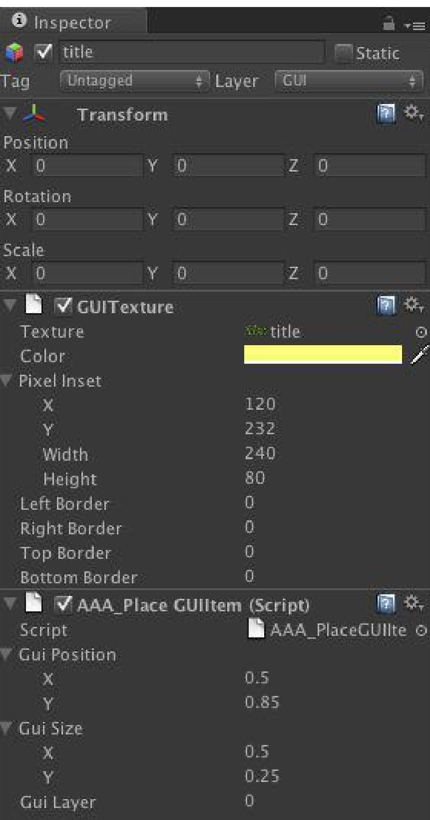
The following image shows the touched **Play** menu item that will be used by our menu script to make the item pop, when the player touches it:

When we place the `GUITextures` in the Unity3D editor, we need to pick one of the standard iOS device layouts. A good choice to work with is the original iPhone (in wide mode for this particular game) because it is the smallest display both in terms of points and pixels. So if our GUI looks good on it, then it will look good on the other displays.

It is important to remember that the final placement of the textures will be altered using the script `AAA_PlaceGUIItem`, so that items are placed on the screen in a display resolution-independent manner. In other words, the placement in the Unity3D editor is for reference during the design of our menu, and we will use the pixel placement to calculate the fractional representation of the script.

The following image shows an example of how we have placed our menu items:

The following screenshot shows the script settings in the Unity3D editor for one of the menu items:

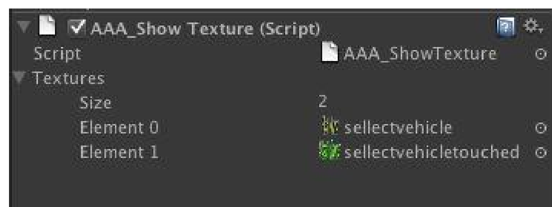


Once the menu `GUITextures` are placed, we need a script to display the correct texture based on a touch on the iOS screen. Conveniently, Unity3D will return a mouse down event when the player is touching the screen, so we can take advantage of that to create a script that will display the correct

texture for both a mouse over and a touch. The script is shown as follows:

```
// Display one of the two textures
// This is an array of textures. The script
// expects there to be exactly 2 textures.
// The first texture is displayed when the
// player is not holding the mouse over or
// touching the menu item.
// The second texture is displayed when
// the player is holding the mouse over
// or touching the menu item
var textures : Texture[];
function Update ()
{
// For iOS devices, the player must have a finger
// on the screen, which Unity3D will also report as
// Mouse Button 0 being held down. So we can use
// that to display the second texture only if the
// iOS device screen is being touched
#if IPHONE_UNITY
if (Input.GetMouseButton(0))
{
#endif
// If there is touch, which Unity3D also reports
// as mousePosition, then display the correct
// texture
guiTexture.texture =
textures[guiTexture.HitTest(Input.mousePosition)];
// This is needed to complete the conditional if given above
#if IPHONE_UNITY
}
#endif
}
```

An example of **AAA_Show Texture (Script)** attached to a menu item game object is shown in the following screenshot. Notice that there we have two textures assigned to the **Textures** array in the inspector. This is done by setting the **Size** element of the **Textures** array to **2**, and selecting the desired textures into **Element 0** and **Element 1**, which is shown as follows:



Finally, we need a script to determine if a texture has been touched or clicked. The way we want this to work is that a click in the Unity3D editor, or a finger on an iOS device, will be used to select a menu item.

This means that in the editor, when we mouse over a menu item, its second texture is displayed. Then when we click on it, the action will take place. Alternatively, on an iOS device, as we slide our finger on the display, the menu item will display its second texture, and when we lift our finger, the action will take place. The script that accomplishes both these tasks is shown as follows:

```
// This is the message that will be
// sent when the menu item is selected
// either by click in the editor or
// touch on the iOS device
var menuMessage : String;
// This is a variable that helps us
// identify if we are running in the
// editor or on a real device
enum myPlatform { Mobile, MacOSX, Windows }
// By default we assume the game is
// running on an iOS mobile device
private var currentPlatform = myPlatform.Mobile;
function Awake ()
{
// If the game is running in the OS X editor
// or as an OS X standalone deployment
// we consider it to be MacOSX
if ((Application.platform == RuntimePlatform.OSXEditor) ||
(Application.platform == RuntimePlatform.OSXPlayer))
{
currentPlatform = myPlatform.MacOSX;
}
// If the game is running in the Windows editor
// or as a Windows standalone deployment
// we consider it to be Windows
if ((Application.platform == RuntimePlatform.WindowsEditor) ||
(Application.platform == RuntimePlatform.WindowsPlayer))
{
currentPlatform = myPlatform.Windows;
}
}
function Update ()
{
// If the mouse button was clicked or the iOS
// device screen was touched during this frame
// the this will be true
var l_mbd : boolean = Input.GetMouseButtonDown (0);
// For iOS devices this will be the last touch
// event otherwise its always 0
var lastTouchEvent : int = 0;
// This block of code only runs on iOS devices
// since it has the touch interface
#if UNITY_IPHONE
// A temporary place to store the last
// touch event
var l_touch : Touch;
// If there was one or more touches
if (Input.touchCount)
{
// The last touch event is the only one
// we want to examine
l_touch = Input.GetTouch(Input.touchCount - 1);
// But we only want to know if the touch
// was not cancelled
if (l_touch.phase != TouchPhase.Canceled)
{
lastTouchEvent = l_touch.phase;
}
}
#endif
```

```
// This is the tricky part
// Basically in the editor a mouse click will be true
// and on an iOS device a finger up will be true
if ((currentPlatform != myPlatform.Mobile &&
l_mbd) ||
(currentPlatform == myPlatform.Mobile &&
lastTouchEvent &&
!Input.GetMouseButton(0)))
{
// So if there was a mouse click or a finger up
// and the mouse or finger is in the texture
// rectangle, then send the message
if (guiTexture.HitTest (Input.mousePosition))
{
SendMessage(menuMessage);
}
}
}
// These are the messages that are implemented
// for ALL the menu buttons
// They could be in separate scripts but for a simple
// menu system its fine to put them all here
// When the select menu is picked, load the
// scene to allow the player to select
// the player avatar
function selectMenu()
{
Application.LoadLevel("MenuSelect");
}
```

Testing the scene in the editor

Because we have created scripts that allow player controls directly in the Unity3D editor, testing in the editor will be very easy for us to do.

The steps are straightforward. They are as follows:

1. Enable the **Stats** option in the **Game** view.
2. Under the **Edit** menu, select the **Choose Graphics Emulation** option and select the appropriate emulator for the device.
3. Click the **Play** button.
4. Repeat steps 1-5 for different devices.

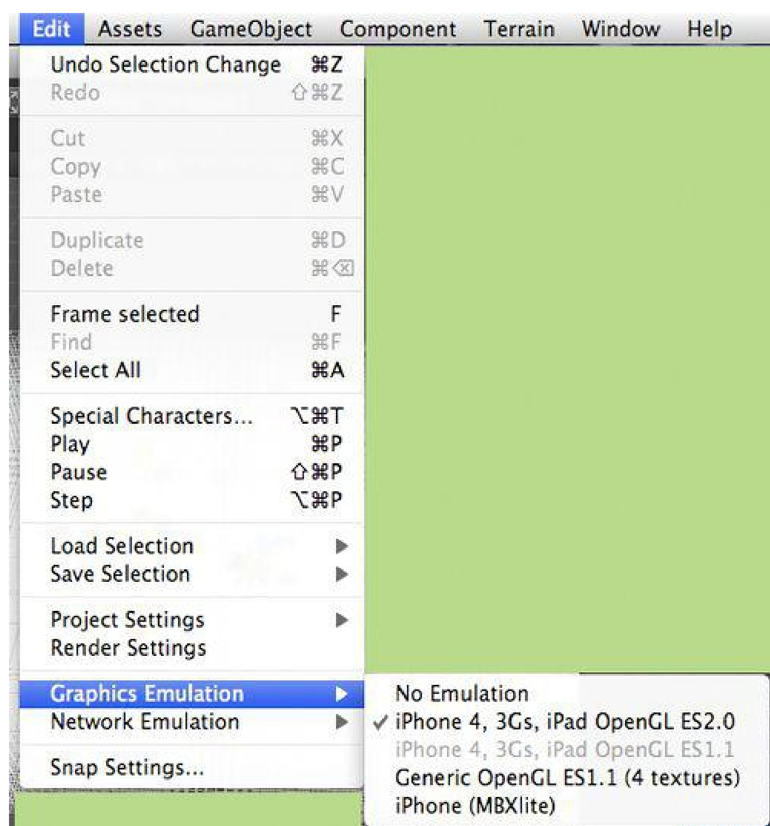
Enable the Stats option

The Unity3D editor's **Stats** option, shown as follows, is a good first step in analyzing the performance of our game on iOS. While some of the stats are not useful in the editor, others, most notably the number of **Draw Calls**, are very useful in understanding how our application may perform when deployed on an actual device:

One of the things we notice right away, however, is that the **Statistics** window is overlaying our game screen. This makes it difficult to see and can interfere with some of the game options that we may want to try.

Choose Graphics Emulation

The Unity3D editor with the **Choose Graphics Emulation** option, shown in the following screenshot, has the benefit of allowing us to test the functionality of any custom shaders in the editor, prior to deployment on an iOS device:



Click the Play button and repeat

It will always reduce our development time, if we can test in the Unity3D editor rather than deploying to a device. By developing games following these guidelines, we can dramatically reduce the amount of device debugging and speed up our time to market.

> ## Tip
>
> Unity Remote
>
> Unity3D does include an application called Unity Remote, which allows you to remotely control the editor from a device. While we do occasionally use Unity Remote (for example, for a quick test of accelerometer functionality), we primarily test it in the editor and on the devices. While Unity Remote is a good idea, it is not something we have found that we really rely on during game development.

Setup for multiple iOS device deployment

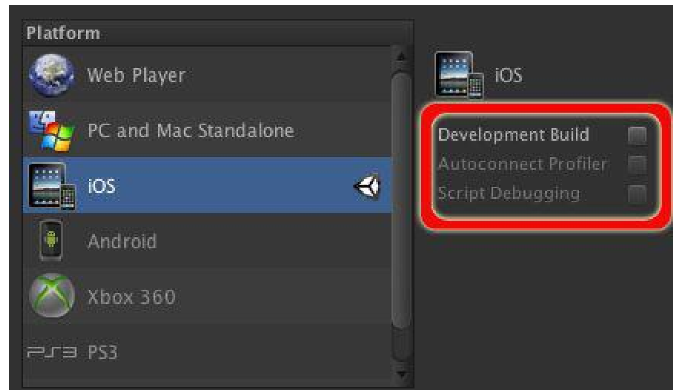
When we develop a game, we want to have as large a target market as we can. With iOS, we can target our game to run on iPhone, iPod Touch, and iPad. We can create a single application image that will run on all of these devices very easily.

The steps to set up for multiple device deployment are as follows:

1. From the **File** menu, choose **Build Settings**.
2. If needed, select **iOS**, and click the **Switch Platform** button.
3. Click the **Player Settings** button and set the **Cross Platform Settings**.
4. Under the **Per-Platform** settings, click the mobile phone icon, and set the **Resolution and Presentation** settings.
5. Set the **Icon** setting.
6. Set the **Splash Image** setting (Pro version required).
7. Set the **Other Settings** (as explained in detail as follows).

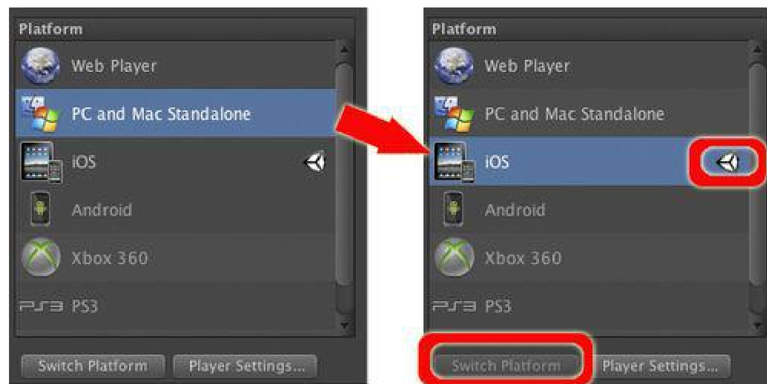
Build Settings

The most important thing we need to remember when setting up for deployment is that the **Development Build** settings are all unchecked, because we don't want to deploy a build that is intended for debugging. The following screenshot shows that all the settings are unchecked:



Select iOS

When we select **iOS** in the **Build Settings**, we need to click the **Switch Platform** button. If the button has been pressed, it will be grayed out, and the Unity3D icon will appear to the right of the **iOS** icon and name in the selection list. The following screenshot shows a correctly selected platform:

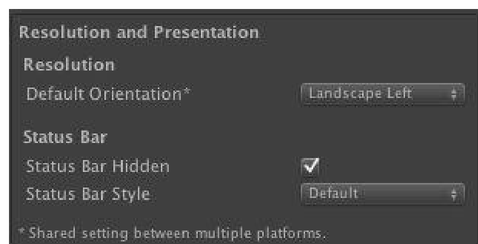


Cross-Platform Settings

The **Cross-Platform Settings** specify basics like our **Company Name**, the name of our game called the **Product Name**, and a 2D texture that will be used as the **Default Icon**. These settings, shown as follows, are required, with the exception of the **Splash Image**, regardless of the platform for which we are deploying our game:

Resolution and Presentation

The most important thing that we need to decide is the default orientation for our game. Typically, it is Portrait, Landscape, or Auto Rotation. The **Default Orientation** setting includes options for **Landscape Right** and **Portrait Upside Down**, mainly for completeness, but we will most likely never use those settings. There is also an option for the **Status Bar**, but again, we will probably always just select a **Status Bar Hidden** for our games. All of the other options, which again are provided for completeness, really don't apply to a game. The following screenshot shows a typical setting for a landscape game:



The Icon

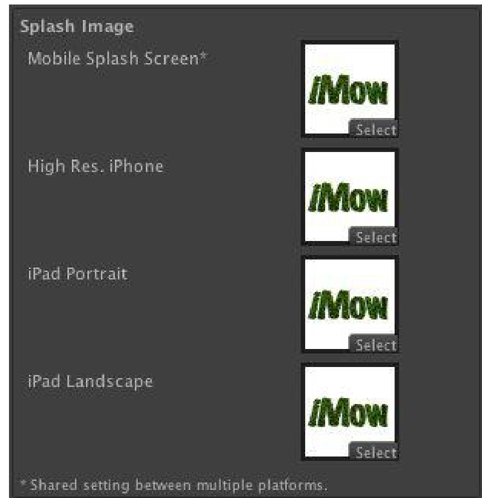
If, for some reason, we want different icons for the different icon sizes supported on different iOS devices, we can specify them. Typically, for a game, we just want to use the default icon, and so the **Override for iPhone** checkbox will be left unchecked. If we do check it, then we need to supply the custom 2D texture for each icon size.

The **Pre-rendered Icon** checkbox may be applicable. If we don't want the iOS to apply a bevel and sheen to our icon, then we need to check the box. Typically though, we do want iOS to apply the bevel and sheen, and so we leave that box unchecked.

The Splash Image (Pro feature)

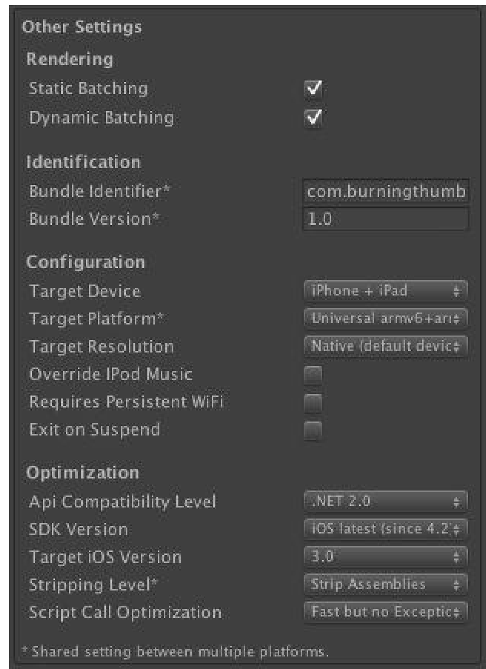
If, for some reason, we want different **Splash Image** sizes, depending on the different screen resolutions of iOS devices, then we can specify them. Typically for a game, we just want to use one **Splash Image**. However, it is possible that there could be a reason to provide different images for some games.

The following screenshot shows how we would typically set up the **Splash Image** using the same image across all device resolutions:



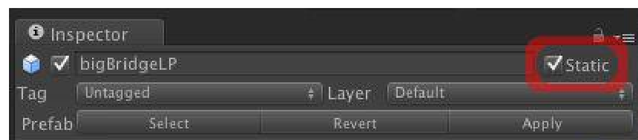
Other Settings

Oddly, the **Other Settings** are the most important settings in the group, when it comes to setting up for multiple device deployment. The settings that we would typically use are shown in the following screenshot:



Rendering

Typically, for rendering and batching (combining multiple objects into a single object, so that they can all be drawn with a single draw call) both **Static Batching** and **Dynamic Batching** should be enabled. There may be rare cases where performance or memory is an issue, and batching needs to be turned off. It is important to remember to mark objects that don't move as **Static** in the editor, as shown in the following screenshot, so that Unity3D knows that **Static Batching** can be applied to those objects:



Dynamic Batching refers to the batching of objects that use shared materials, which is generally a good thing.

Identification

The bundle identifier should use the reverse domain notation, `com.domain.gamename`, or sometimes `com.domain.mobile.gamename`. The bundle version is the version number of the game.

Configuration

This is the most important section when deploying for multiple devices. To include the greatest number of devices, the Target Device needs to be set to **iPhone + iPad** and the target platform needs to be set to **Universal arm6+arm7 (OpenGL ES1.1+2.0)**. The **Target Resolution** should be **Native** (default device resolution).

The remaining three settings depend on our game and need to be set based on our game's requirements. Typically, we do not override iPod Music (overriding prevents the user from listening to their tunes while playing). We enable persistent Wi-Fi only in network games, and exit on suspend for multitasking versions of iOS. There may be special cases where we want to use other settings, but those would be exceptional.

Optimization

Because we make extensive use of `Hashtables`, the API compatibility level needs to be set to .NET 2.0 (full not subset). Because we always want to use the latest SDK when building for iOS devices (this is different from the desktop Mac OS X where we would want to use the SDK version that matches the highest OS version that we support with our application), the SDK version will be set to **iOS latest** (since 4.2). The **Target iOS** version should always be set to the lowest version available. The **Stripping** level will be set to **Strip Assemblies**. We could use more stripping, but only if we are sure Unity3D will figure out what we need. Often, it's not the case without us providing hints to Unity3D. And finally, **Script Call Optimizations** should always be set to **Fast**, no exceptions when deploying to a device.

Deploy the App to multiple devices

Once we have configured for deployment to multiple devices, we need to actually deploy and test on multiple devices. This, in turn, means that we need to have multiple devices. Often it is possible to purchase older devices on eBay and new devices from the refurbished section of the Apple store, so that we can stretch our device-buying dollar. We absolutely need to test on as many different classes of physical devices as possible. Of course, it is not always possible to test on all the classes of devices, but we should at least have one device from each screen resolution. The moral here, though, is the more the better.

Note

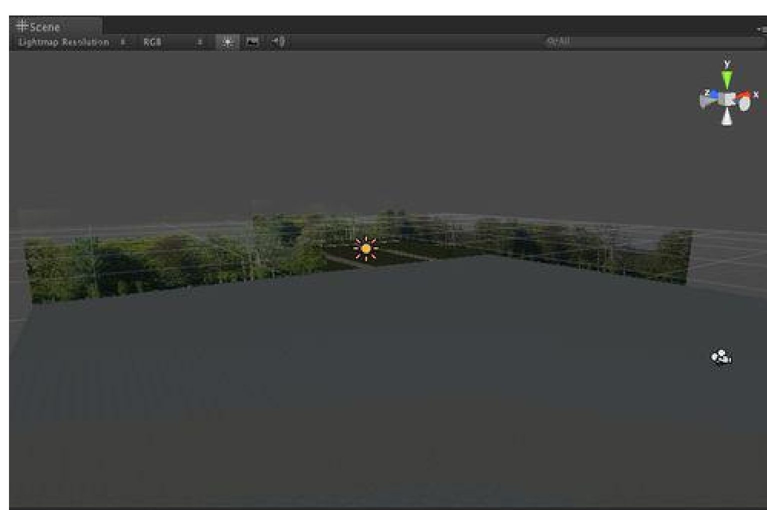
We are not going to cover the setup and provisioning of iOS devices in iOS Provisioning Portal, other than saying all your devices must be configured and you must have installed the appropriate profiles. Apple provides excellent step-by-step documentation on how to do these.

Once you have your devices provisioned and the profiles installed, you need to do the following steps in Unity3D for iOS:

1. From the **File** menu, choose **Build Settings**.
2. Make sure all the scenes you want, which are included in the final build, are checked.
3. If required, select the iOS platform and change to that platform.
4. Click the **Build** button.
5. Choose a folder in which to save the build.
6. Click the **Save** button.
7. Open the generated project in Xcode.
8. Physically connect your first device.
9. Build and run in Xcode.
10. Repeat steps 6-7 for each device.

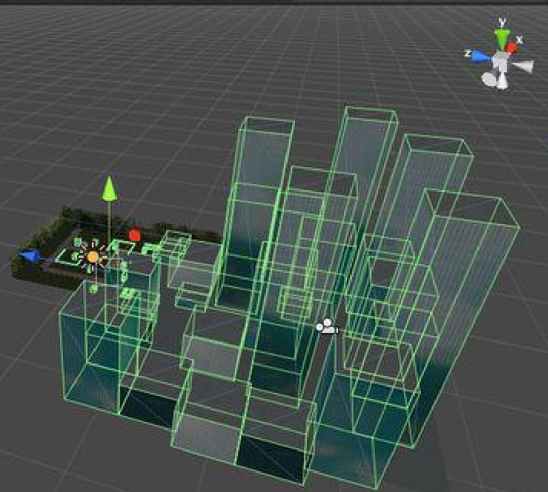
Task: create the background scene

In the Chapter 4, *Unity Project*, folder, open the `iMow.unity` file and you will see a pre-defined terrain that looks like the following image:



In the **Project** tab, you will find a number of prefab assets for houses and buildings. Use the prefab assets to create an urban and suburban scene. You do this by dragging the prefab building assets and placing them in the scene in the game object named **AAA_Create_The_Background_Scene**.

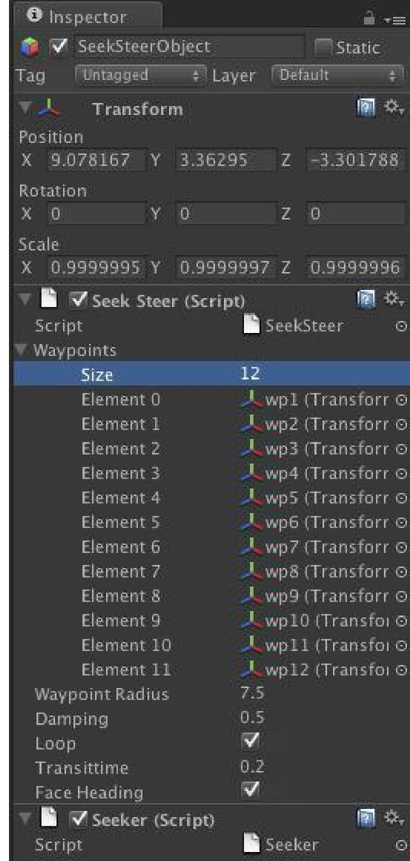
To see an example of the kind of scene that you need to build, enable the disabled game objects named **ZZZ_Answers** and its children. The example scene will look like the following image; of course, your scene will look different. Make sure to disable the answer scene and its children, so that you can build your own scene:


Create
y
x
z
AAA_Origin
AAA_SceneManager
HUD
Level
Advertising
Roads
Terrain
ZZZ_Answers
Menu
Globals
Player
CameraPivot
Mower
rideonmower-lowpoly

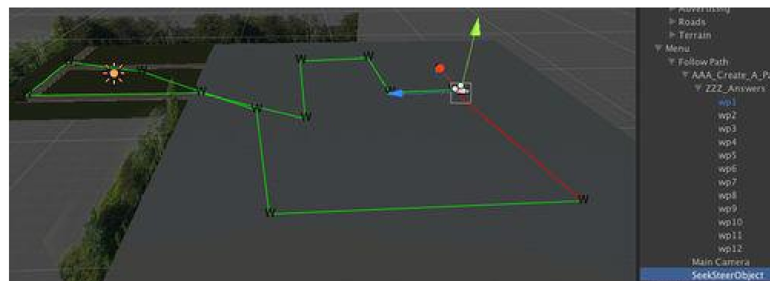
Task: create a path to follow

After you create your scene, you, for this example, need to create a path for the camera to follow. You do this by dragging the prefab named **waypoint** from the **Project** tab into the scene several times to define your path. Make sure to put the waypoints into the game object named **AAA_Create_A_Path_To_Follow**. There is an item named **ZZZ_Answers** in that folder with a predefined path that you can activate (used for seeing and for providing an example for reference) and then deactivate prior to building your own path. You need to put a waypoint at every point where you want the camera to change direction. The example waypoints, with the background scene buildings disabled, are shown in the following image:

Locate the **SteerSeakerObject** in the **Hierarchy** tab and select it. Then in the **Inspector** tab, change the **Size** of the **Waypoints** array to match the number of waypoints you have created, and drag each waypoint into an array element, making sure to place them in the array in the order that you want them followed. The following screenshot shows what the array looks like when populated by the waypoints from the **ZZZ_Answers** folder:



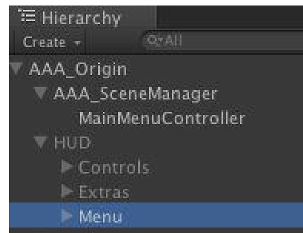
As soon as you drag in the final waypoint, the `SeakSteer` script will draw the lines between the waypoints in your path, and it will look like the following image (again, this image is shown with the background scene disabled):



Once you have completed this step, and of course, you will do it with the background scene enabled, pressing **Play** in the Unity3D editor will result in the camera following your path through the city.

Task: putting it all together with the menu

Finally, you need to enable the **Menu**, so that it is active in front of the scene, as the camera follows the path through the scene. You will find the **Menu** folder disabled in the **HUD** (Heads Up Display) folder in the **Hierarchy**, as shown in the following screenshot:

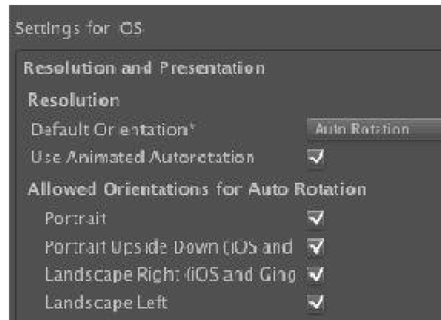


You need to select the **Menu** in the **Hierarchy** and enable it in the **Inspector** to complete the scene.

Challenge: flipping the iOS device

One of the big differences between iOS devices and desktop devices is the frequency with which the device will be rotated. Our game interface needs to rotate, when the device is rotated, to take full advantage of this difference. In fact, for the iPad, if our game does not support at least flipping of the device, Apple will not accept it in the App store.

In Unity3D v3.4, an option called **Auto Rotation** was added to the **Resolution and Presentation** present under the **Settings for iOS**. This setting, shown as follows, works effectively when rotating a game:



It is sometimes desirable to have a bit more control than this setting provides for optimizing our game, so that only the components that are interested in the rotation are updated when a rotation occurs. We can roll our own rotation code and take more control over exactly what happens during a device rotation.

The code needed to support flipping of the device is simple and is shown as follows as a fragment from a rotation script:

```
// Rotate the Screen to match the device rotation
if (Input.deviceOrientation ==
DeviceOrientation.Portrait)
{
Screen.orientation =
ScreenOrientation.Portrait;
}
else if (Input.deviceOrientation ==
DeviceOrientation.PortraitUpsideDown)
{
Screen.orientation =
ScreenOrientation.PortraitUpsideDown;
}
else if (Input.deviceOrientation ==
DeviceOrientation.LandscapeRight))
{
Screen.orientation =
ScreenOrientation.LandscapeRight;
}
else if (lastOrientation ==
DeviceOrientation.LandscapeLeft))
{
```

```
Screen.orientation =
ScreenOrientation.LandscapeLeft;
}
```

The problem is that it gets a bit more complicated in the real world than this code fragment implies. Typically, we need to do the following:

- Conditionally support portrait and landscape flipping
- Conditionally eliminate the black rotation borders that are on by default
- Send a message to one or more other objects when the device is rotated
- Optimize for performance

For example, our menu items need to know when the device is rotated, so that they can reposition themselves to the correct resolution-independent locations on the display. The following script is an example of how we can modify the `AAA_PlaceGUIItem` script, so that it can receive and act on a message from another script to reposition the item, if the display is rotated.

The important thing to notice is that the script uses private variables, so that the absolute pixel positions can be recalculated from the fractional positions, and the logic has been moved into a new function name, `RotateDevice()`, so that when a `RotateDevice` message is sent to the game object using `BroadcastMessage()`, the `GUITextures` will be repositioned on the screen. This is shown as follows:

```
// This editor value is the fractional position
// on the screen where the GUITexture should be placed
var guiPosition : Vector2;
// This editor value is the fractional position
// on the screen where the GUITexture should be placed
var guiSize : Vector2;
// This editor value is the layer
// on the screen where the GUITexture should be placed
var guiLayer : float;
// To handle device rotation we calculate
// GUI positions privately
private var calculatedGUIPosition : Vector2;
private var calculatedGUISize : Vector2;
function Start ()
{
// To handle device rotation we place the
// item in a function that will be called
// by another script
RotateDevice();
}
// The RotateDevice message is going to be
// sent by the RotateDevice script whenever
// the player changes the orientation of
// the iOS device
function RotateDevice()
{
// This converts the fraction values to
// absolute pixel values
calculatedGUIPosition.x = Screen.width * guiPosition.x;
calculatedGUIPosition.y = Screen.height * guiPosition.y;
```

```
// This places the GUITexture at the correct
// pixel relative position and scales it to the
// correct size
if (Screen.width * guiSize.x < 44.0 || Screen.height * guiSize.y < 44.0)
{
calculatedGUISize.x = guiTexture.pixelInset.width;
calculatedGUISize.y = guiTexture.pixelInset.height;
}
else
{
calculatedGUISize.x = Screen.width * guiSize.x;
calculatedGUISize.y = Screen.height * guiSize.y;
}
// This makes sure the transform is located at
// position (0,0,layer) with a localScale of (0,0,0)
transform.position = Vector3(0,0,guiLayer);
transform.localScale = Vector3.zero;
// This places the GUITexture at the correct
// pixel relative position and scales it to the
// correct size
guiTexture.pixelInset =
Rect ( calculatedGUIPosition.x - calculatedGUISize.x / 2 ,
calculatedGUIPosition.y - calculatedGUISize.y / 2 ,
calculatedGUISize.x, calculatedGUISize.y);
}
```

The challenge is to develop an optimized `RotateDevice` script that allows us to set values in the **Inspector** to enable and disable support for Portrait rotation, Landscape rotation, and for turning off the black border rotation animation. The script also needs to send a message to the `GUITextures`, so that they know they have been rotated. Finally, it needs to do all of these things in a performance-optimized manner.

You can find a sample solution in the script named `RotateDevice.js`, which is attached to the main camera in the `Chapter 4 Unity Project` folder.

Summary

In this chapter, we have covered the following:

- The development of a complete and engaging menu system that allows our player to fly through our game environment, and experience a level that truly gives a feel of the large scale of our games' many levels
- The ways in which we can manage the unique challenges and features of the different screen resolutions of iOS devices
- The ways in which we can save time by continuing to develop a robust set of scripts that work both in the Unity3D editor and on iOS devices
- How to configure Unity3D and deploy our game as a universal application that will run on all iOS devices

In Chapter 5, *Scalable Sliding GUIs*, we will discuss how to make a resolution-independent Graphical User Interface (GUI) system with sliding dialogs.

Chapter 5. Scalable Sliding GUIs

One of the most important parts of our game is the Graphical User Interface (GUI). The GUI is the part of the application that our game player will interact with to perform basic actions like setting options, saving preferences, and selecting levels. The Unity3D GUI system provides us with the basic building blocks for creating a GUI, but it is up to us to extend those basics into a more comprehensive GUI system.

The GUI that we provide needs to match the mood of our game and interact with the player by providing a polished interface that is as compelling as the game content itself. The GUI should enhance the game experience and stay out of the way of the actual gameplay, whenever the player does not absolutely need to interact with it. In this chapter, we will learn the following:

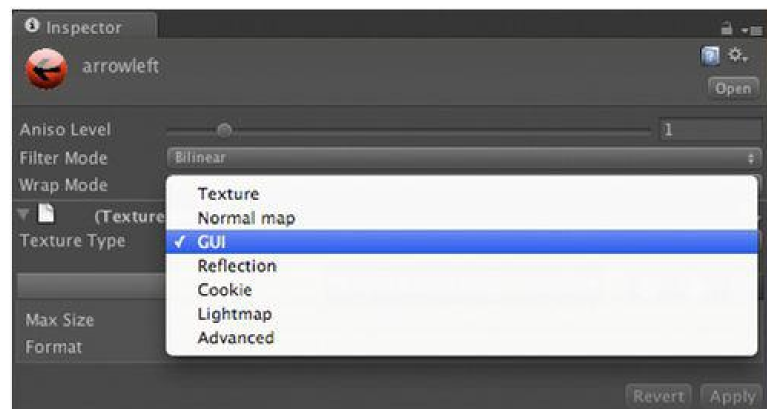
- How to scale a Unity3D for iOS GUI?
- How to make our GUI wrappers re-usable by separating the dialog function from the dialog content?
- How to animate GUI dialogs, so that they are more engaging?
- Look at some of the ways in which the basic GUI system provided in Unity3D can be enhanced to provide this interface for our game.

Think about resolution not pixels

As we have said before, it is important with Unity3D for iOS to think in terms of screen resolution, and not in terms of pixels. If we just think in terms of pixels, our GUI will not work correctly on the current crop of iOS devices.

Typically, we can create artwork that, when imported using a **Texture Importer** with **Texture Type** set to **GUI**, looks good when scaled to all of the iOS device resolutions possible. In fact, the same artwork will even look good on desktop platforms.

The following screenshot shows an example of using the GUI Texture Type to import a texture:



Whenever pixel art is used, we need to look at the final results on different-sized displays to make sure it looks good when scaled for each device resolutions. In most cases, we will find that Unity3D can scale images from larger sizes down to smaller sizes quite well, though it is possible that we will find some pixel art does not look good at some resolution, when the art is resized by Unity3D. In those cases, we may decide to include multiple copies of the same image at different resolutions (typically resized by the software in which the art was created), and have Unity3D display the image that looks the best for the resolution of the display.

The GUI scripts will then display the higher resolution art on higher resolution screens rather than relying on Unity3D to scale the artwork up or down.

Having said that, for our GUI system, we are going to use a single copy of the artwork, and rely on Unity3D to scale it for us. We can use a number of standard image formats for our GUI elements including, but not limited to, `.png`, `.jpg`, `.tiff`, and `.psd` file formats.

Because the Unity3D GUI system is pixel-based, in that the function we call expects us to provide absolute screen coordinates to place GUI items and absolute pixel sizes to scale items, we need to have some mechanisms to reposition and resize GUI items based on the actual screen resolution of the device on which our application is running.

Previously, we had accomplished this scaling by specifying the location and scale of GUITexture items as a fraction of the total screen size.

With the GUI system, we are going to approach things a bit differently. Instead of working with

fractions of screen size, we are going to assume a default screen size and create our GUI to fit that default screen size using absolute positions and sizes.

Then we are going to calculate, based on the actual screen size, a transformation matrix that we will apply to the GUI, so that Unity3D automatically scales the GUI to fit the screen resolution.

This approach allows us to scale an entire GUI dialog, rather than each individual GUI dialog item.

The following code fragments illustrate how this is going to work.

First, we create two inspector-visible values that we will set to the width and height of the iOS screen on which we designed our GUI. We will do it as follows:

```
// These two inspector variables are set to
// the screen resolution that our GUI is
// designed to fit without any scaling
var width : float = 480.0;
var height : float = 320.0;
```

It is important that our GUI fits onto this screen resolution without any scaling, as our script is going to calculate the scaling factor based on this screen resolution to fit the actual resolution.

> ## Note
>
> It does not matter if we choose 480x320 or 960x640, it only matters that we pick the resolution and use it consistently. Some people prefer the results when scaling down, and other people prefer the results when scaling up. There is no right or wrong answer beyond being consistent.

If, for some reason, we specified these values incorrectly, the final GUI will not scale correctly on any iOS device. This is illustrated in the following screenshots, where the GUI has been designed for an iOS screen resolution of 480x320, and when configured correctly, displays correctly, but when the inspector values are incorrectly set to 960x640, the GUI dialog items appear to be half the size that they should appear.

The first screenshot is as follows. It shows a GUI item in the Unity3D inspector with the **Width** set to **480** and the **Height** set to **320:**

The next screenshot shows the dialog contents displayed correctly in a running game scene:

The following screenshot shows a GUI item in the Unity3D inspector with the **Width** set to **960** and the **Height** set to **640:**

The following screenshot shows the dialog contents displayed incorrectly in a running game scene:

In order to calculate the transformation, we are going to need a matrix. The line of script code that declares the private matrix variable is shown as follows:

```
// this sets our "virtual" screen for the GUI.matrix stuff
privatevartMatrix : Matrix4x4;
```

> ## Note
>
> We could manually calculate the sizes of every GUI item, based on the actual screen size, but we would also need to adjust things like font sizes. It is much easier to simply use the transform matrix function, which automatically adjusts everything, including the font, in a single line of code.

The `Awake()` function is where we will calculate the actual transformation matrix. The calculation is done in `Awake()` so that it only needs to be done once. The script code that calculates the matrix is shown as follows:

```
// Whenever the device is rotated
// the matrix needs to be
// recalculated
functionRotateDevice()
{
// Calculate the transformation matrix
// for the actual device screen size
tMatrix = Matrix4x4.TRS(Vector3.zero,
Quaternion.identity,
Vector3(Screen.width / screenWidth,
Screen.height / screenHeight,
1.0));
}
function Awake ()
{
// There is lots of code not shown here
// for this example …
// The very first time the script
// runs the matrix must be calculated
// and since that is done in RotateDevice()
// we call it here
RotateDevice();
}
```

Finally, in the `OnGUI()` function, the transformation matrix is applied to scale the entire GUI dialog contents as follows:

```
functionOnGUI()
{
// There is lots of code not shown here
// for this example
// Apply the transformation matrix
// for the actual device screen size
GUI.matrix = tMatrix;
// There is lots of code not shown here
// for this example
}
```

To summarize, we need to design our GUI for a specific screen resolution. We can pick any resolution that we like, and in fact, it could be a desktop computer screen resolution or an iOS

device screen resolution. We should design and test our GUI using the selected resolution to make sure it works the way we would like it to. And finally, we will apply a transformation matrix in the `OnGUI()` function to scale the GUI automatically to the resolution of the screen on which the application is running.

Separating dialogs from contents

We want to develop the dialog system, or core dialog functions, independently of the individual dialog content, so that the script code that manages the behavior of dialogs does not get mixed with the script code that presents and manages specific dialogs.

We need to consider the parts of the dialog that can be the same for every dialog versus the parts of the dialog specific to that dialog and find a way to separate the implementation.

The things that the dialog wrapper will implement are as follows:

- Displaying the **OK** and **Cancel** buttons
- Providing a hook to allow another script to manage the dialog content
- Sending messages to other game objects, when the **OK** or **Cancel** button is pressed

The buttons

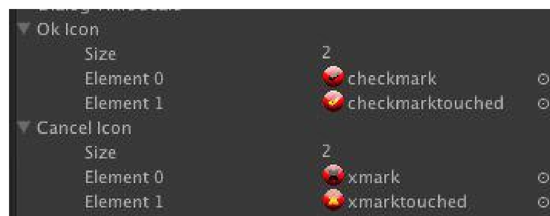
Our dialog wrapper assumes that there will be at least an **OK** button, and at most, an **OK** and **Cancel** button. The dialog wrapper will automatically display those buttons in the corners of the dialog. If we have a special dialog that, for some reason, does not want to use these buttons, it is of course, free to implement its own buttons in the dialog content script.

The dialog wrapper script fragments that implement the **OK** and **Cancel** buttons are shown as follows:

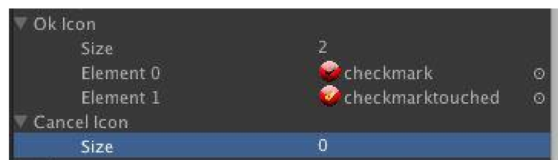
```
// Set two icons for the buttons
// The first icon is displayed
// when the button is not selected
// and the second icon is displayed
// when the button is selected
varokIcon : Texture2D[];
varcancelIcon : Texture2D[];
// Index of which icon texture
// to display
privatevargOKIconIndex : int;
privatevargCancelIconIndex : int;
function Awake ()
{
// There is lots of code not shown here
// for this example
// Show the unselected icons for the
// OK and Cancel buttons
gOKIconIndex = 0;
gCancelIconIndex = 0;
// There is lots of code not shown here
// for this example …
}
functionOnGUI()
{
// There is lots of code not shown here
// for this example
// Draw the OK button
// If two textures are provided the
// button will be drawn
if (2 == okIcon.length)
{
if (GUI.Button(okRect, okIcon[gOKIconIndex]))
{
gOKIconIndex = 1;
// There is lots of code not shown here
// for this example
}
}
// Draw the Cancel button
// If two textures are provided the
// button will be drawn
if (2 == cancelIcon.length)
{
if (GUI.Button(cancelRect,
cancelIcon[gCancelIconIndex]))
```

```
{
gCancelIconIndex = 1;
// There is lots of code not shown here
// for this example
}
}
GUI.EndGroup();
}
```

Using these scripts, the button will only be displayed if there are two textures provided in the Unity3D interface. If any other number of textures is provided, the button will not be displayed. The following screenshot shows what a dialog with both a **Cancel** and **OK** button will look like, when correctly configured in the inspector:



The following screenshot shows what a dialog with only an **OK** button will look like, when correctly configured in the inspector:



The hook

The way we accomplish this separation is by creating two scripts. The first script will be a generic dialog wrapper script and the second script will render the contents of the dialog. In fact, we will create a variable on the first script that allows us to assign in the Unity3D inspector, a second game object that is responsible for managing the content of the dialog. By making it a game object, we can reuse the wrapper with many different dialogs that contain different content.

The following script fragments show the implementation in the dialog wrapper script. The implementation of the content for any particular dialog would depend on what content that dialog needed to render and manage:

```
// This is the game object responsible
// for rendering the dialog content
varcontentGO : GameObject;
// This is the messages sent
// to the content rendering game object
// The rendercontent method must exist
// in a script on that object
varcontentRenderMessage : String = "rendercontent";
// The box size, is the size of the dialogs
// bounding box specified as a fraction of
// the screen size so that it is
// resolution independent. It will be
// converted to the correct number of
// pixels by the script
varboxSize : Vector2;
functionOnGUI()
{
// There is lots of code not shown here
// for this example
// Apply the transformation matrix
// for the actual device screen size
GUI.matrix = tMatrix;
GUI.BeginGroup(aRect);
if (true == contentGO.active)
{
contentGO.SendMessage (contentRenderMessage,
boxSizePixels,
SendMessageOptions.RequireReceiver);
}
// There is lots of code not shown here
// for this example
}
```

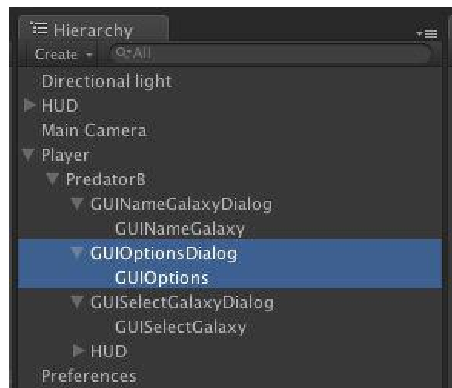
In addition, the content rendering script will implement the `rendercontent()` method, so that when the wrapper wants the content rendered, that message will be successful. The following script fragment shows how the function will look:

```
functionrendercontent(boxSize : Vector2)
{
// There is lots of code not shown here
// for this example
```

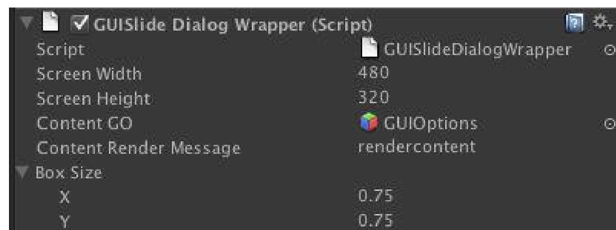
```
}
```

One of the things we may notice is that the `rendercontent()` function expects a `boxSize` function to be passed in to it. This is the size of the dialog in which the content will be rendered. The `boxSize` is set on the wrapper, and passed in to the content renderer.

The way these parts will come together in a Unity3D game is that we will create two game objects. The first will have the dialog wrapper script attached, and the second will have the dialog content rendering script attached. In the inspector, we will drag the second object into the first object, so the content rendering object is parented under the dialog wrapper object. In addition, in the inspector, we drag the rendering object into the `ContentGO` (note that GO is programmer shorthand for `GameObject`; rather than saying `ContentGameObject`, we just say `ContentGO)` variable of the wrapper script. An example of this, for our **GUIOptionsDialog**, is shown in the following screenshots. The first screenshot shows the **GUIOptionsDialog** game object and the **GUIOptions** content rendering game object in the Unity3D hierarchy:



The second screenshot shows the configuration of the **GUISlide Dialog Wrapper (Script)** on the **GUIOptionsDialog**, after it has been configured to reference the **GUIOptions** content rendering game object:



Dialog location, animation, and speed

Now that we have a framework for the separation of GUI dialog content from GUI dialog control, we need to look at the things we want to accomplish with our GUI dialog wrapper.

The following are the things that we want to create:

- Dialogs that can be located on the screen in a resolution-independent manner
- Dialogs that can slide onto the screen and off the screen
- Dialogs that can fade in and out
- Dialogs that move at different speeds

Dialog location

Before we can talk about where to put our dialog on the screen, we need to know how big the dialog is going to be. It simply would not make sense to put a dialog that is three fourth the width of the screen, and position it in a way that leaves part of the dialog content clipped.

So, the very first thing we need to decide is the size of the dialog, not in pixels, but as a ratio of the screen size. For example, if we want the dialog to be one half the width of the screen we would choose the value 0.5. We do this by providing a `Vector2` that is visible in the Unity3D inspector, so that we can create GUI dialogs of different sizes.

The following script fragments (recall that `screenWidth` and `screenHeight` are inspector variables, which are set to the screen resolution that our GUI is designed to fit without any scaling) show how the GUI dialog size will be implemented:

```
// The box size, is the size of the dialogs
// bounding box specified as a fraction of
// the screen size so that it is
// resolution independent. It will be
// converted to the correct number of
// pixels by the script
varboxSize : Vector2;
// The dialog box size needs to be
// converted to the correct number
// of pixels. The number of pixels
// is stored in this private variable
privatevarboxSizePixels : Vector2;
function Awake ()
{
// Convert the fractional representation
// of the box size to absolute pixels
boxSizePixels.x = boxSize.x * screenWidth;
boxSizePixels.y = boxSize.y * screenHeight;
// There is lots of code not shown here
// for this example …
}
```

Once we know how to specify the size of our box, we need to specify its position on the screen. We could simply create one variable to contain the position. However, thinking ahead, we know that we will want the dialog to slide on and off the screen. In order to handle dialog sliding, we need to know its starting position and its ending position, so that we can slide it from the starting position to the ending position, when it is coming in, and slide it from its ending position to its starting position, when it is sliding out. The following code fragment shows how the starting and ending positions will be implemented:

```
// The dialog needs to start in one
// place and end in another so that
// we can make it slide. Its important
// to note these positions can be off
// the screen so that dialogs can
// slide on and off the screen. Similar
// to the box size, the dialog positions
```

```
// are specified in fractions of the
// screen size and not in absolute pixels
//
// Note that we use Vector3 instead of
// Vector2 because the Z value is
// used to set the layer for overlapping elements
varstartPos: Vector3;
varendPos: Vector3;
// The dialog box positions need to be
// converted to the correct number
// of pixels. The number of pixels
// is stored in these private variables
privatevarstartPosPixels: Vector3;
privatevarendPosPixels: Vector3;
function Awake ()
{
// There is lots of code not shown here
// for this example …
// Convert the fractional represnetation
// of the dialog positions to absolute pixels
startPosPixels.x = startPos.x * screenWidth;
startPosPixels.y = startPos.y * screenHeight;
startPosPixels.x -= boxSizePixels.x / 2;
startPosPixels.y -= boxSizePixels.y / 2;
endPosPixels.x = endPos.x * screenWidth;
endPosPixels.y = endPos.y * screenHeight;
endPosPixels.x -= boxSizePixels.x / 2;
endPosPixels.y -= boxSizePixels.y / 2;
// There is lots of code not shown here
// for this example …
}
```

The following screenshot illustrates what the inspector settings, which we can change to any value that we like (these are just used as an example), for a typical dialog that slides onto the screen will look like:

Box Size	
X	0.75
Y	0.75
Start Pos	
X	0.5
Y	-1
Z	0
End Pos	
X	0.5
Y	0.4
Z	0

Dialog sliding

Now that we know how big our dialog will be, and both its starting and ending positions, we can implement the script code that will slide the dialog on and off the screen.

The first thing that we need to decide is whether the dialog will start in its start position, which would be typical, or its end position, which is desirable in some special cases. In this case, we will create an inspector variable, that is, a Boolean with a default value of false to reflect the typical case.

Then we need to decide if the dialog script should disable itself when the dialog is in its start position, which will typically be off the screen. The default case is that dialogs, which are in their start position, will disable themselves, though there may be special cases where we want a dialog to continue running even if it is in its starting position.

So in a typical dialog scenario, we will have a starting position that is off the screen, and ending position that is on the screen, and a reasonable amount of time to slide the dialog in to place. Also, so that a dialog does not impact performance when it is not being displayed, dialogs that are off screen (in their starting position) will automatically disable themselves.

Dialog fade

We can control the visibility of a GUI by setting the alpha channel of the `GUI.color`. When we slide our GUI dialog in, we may want it to fade in, and when we slide the dialog out, we may want it to fade out. Because we want to be able to turn fading on or off, we use a Boolean variable that can be set in the inspector. If the value is true, the dialog will fade as it slides.

Dialog speed

We need to decide how long it will take for the dialog to slide in to position and out of position. The speed at which our dialogs slide on to and off of the screen are an important component of setting the mood of our game. They may slide in slowly with grace or they may snap in with military precision. It is important that we have the ability to adjust the slide speed for any dialog. We achieve this by creating an inspector variable that lets us set the speed in seconds that it takes for a dialog to slide in and out.

The script fragments that implement these basic sliding and fading features are shown as follows:

```
// The dialog needs to start in one
// place and end in another so that
// we can make it slide. Its important
// to note these positions can be off
// the screen so that dialogs can
// slide on and off the screen. Similar
// to the box size, the dialog positions
// are specified in fractions of the
// screen size and not in absolute pixels
varstartPos: Vector3;
varendPos: Vector3;
// The dialog box positions need to be
// converted to the correct number
// of pixels. The number of pixels
// is stored in these private variable
privatevarstartPosPixels: Vector3;
privatevarendPosPixels: Vector3;
// The dialog needs to slide for some
// amount of time. Some dialogs may
// slide in quickly while others may
// slide in slowly. The amount of time
// depends on the dialog and the game
varslideTime: float = 3.0;
// Normally when a dialog is in the start
// position, we want to disable it so
// that it does not impact performance.
// There are special cases where we want
// the dialog to remain active even if
// it is in the start position
vardisableWhenInStartPos : boolean = true;
// This is a private helper variable that
// we set when we want the update function
// to disable the game object
privatevardisableThis : boolean = false;
// The dialog can optionally fade in
// as it slides in and fade out as it
// slides out. The fade option is
// controlled by the fade flag
var fade : boolean = true;
function Awake ()
{
// There is lots of code not shown here
// for this example
// Convert the fractional represnetation
```

```
// of the dialog positions to absolute pixels
startPosPixels.x = startPos.x * screenWidth;
startPosPixels.y = startPos.y * screenHeight;
startPosPixels.x -= boxSizePixels.x / 2;
startPosPixels.y -= boxSizePixels.y / 2;
endPosPixels.x = endPos.x * screenWidth;
endPosPixels.y = endPos.y * screenHeight;
endPosPixels.x -= boxSizePixels.x / 2;
endPosPixels.y -= boxSizePixels.y / 2;
// There is lots of code not shown here
// for this example …
// Since we cannot disable a game object
// from withing OnGUI() (it causes an internal
// error in Unity3D to do so), we do it in
// Update() based on a flag set in OnGUI()
function Update()
{
if (disableThis)
{
disableThis = false;
gameObject.SetActiveRecursively(false);
}
}
functionOnGUI()
{
// If the GUI is waiting to be
// disabled, do nothing
if (disableThis)
{
return;
}
// There is lots of code not shown here
// for this example
gTime += Time.deltaTime;
var t: float = gTime / slideTime;
// The corner of the dialog in
// screen pixels
var corner : Vector3;
// Default the alpha to 1 so that the
// dialog is fully visible
varl_a : float = 1.0;
// This is the part that does the actual
// sliding of the dialog
if (slide)
{
// This is slide out
// Linear interpolate from the END
// position to the START posiiton
corner = Vector3.Lerp(endPosPixels, startPosPixels, t);
// If fading, then fade out
// The dialog will fade out as it slides out
if (fade)
{
l_a = Mathf.Lerp(1.0, 0.0, t*2);
}
wasDisplayed = false;
// If the dialog auto disables
// after sliding out, set the
// flag so that Update() will
// disable the game object
```

```
if (disableWhenInStartPos)
{
if (
(Mathf.FloorToInt(corner.x) ==
Mathf.FloorToInt(startPosPixels.x)) &&
(Mathf.FloorToInt(corner.y) ==
Mathf.FloorToInt(startPosPixels.y)) &&
(Mathf.FloorToInt(corner.z) ==
Mathf.FloorToInt(startPosPixels.z))
)
{
disableThis = true;
return;
}
}
}
else
{
// This is slide in
// Linear interpolate from the START
// position to the END posiiton
corner = Vector3.Lerp(startPosPixels, endPosPixels, t);
// If fading, then fade in
// The dialog will fade in as it slides in
if (fade)
{
l_a = Mathf.Lerp(0.0, 1.0, t/1.25);
}
wasDisplayed = true;
}
// Set the transparency of the dialog
GUI.color.a = l_a;
// There is lots of code not shown here
// for this example
```

Task: script a scalable dialog

From the code bundle of Chapter 5, open up the `UnityProject` folder and then open the `SceneFirst.unity` file; you will see a pre-defined main menu that looks like the following image:

In that project, look in the game hierarchy and locate the game object named **Player**. Open **Player** and then open **Menu**. You should see an enabled game object named **Buttons** and a disabled game object named **ZZZ_Answers**.

Once we have located the correct game object in the hierarchy, we need to do the following:

1. In the **Menu** game object, create a new game object named **Dialogs**.
2. In the **Dialogs** game object, create a new game object for your **OptionsDialog** and a new game object for your **OptionsDialogController**.
3. Finally, create one more game object for your **OptionsDialogContent** and put it in the **Options** game object.

The hierarchy should not contain game objects nested like the following (your names may be different):

Player

Menu

Buttons

Dialogs

OptionsDialog

OptionsDialogContent

OptionsDialogController

ZZZ_Answers

Set up the OptionsDialog

The **OptionsDialog** needs to be set up, so that the dialog is positioned and slides as well as displays the correct button icons. The steps for this are as follows:

1. On the **OptionsDialog** game object, attach the script named **GUISlideDialogWrapper.js**.
2. Select the **OptionsDialog** game object.
3. In the inspector for the **OptionsDialog**, locate the **GUI Slide Dialog (Script)**.
4. In the **GUI Slide Dialog (Script)**, locate the **Content GO** variable and drag your **OptionsDialogContent** game object into that value.
5. In the **GUI Slide Dialog (Script)**, locate the **Box Size** and set the **X** value to **0.8** and the **Y** value to **0.7**.
6. In the **GUI Slide Dialog (Script)**, locate the **Start Pos** and set the **X** value to **0.5**, the **Y** value to **-1**, and the **Z** value to **0**.
7. In the **GUI Slide Dialog (Script)**, locate the **End Pos** and set the **X** value to **0.5**, the **Y** value to **0.4**, and the **Z** value to **0**.
8. In the **GUI Slide Dialog (Script)**, locate the **Skin** and set the value to **CustomSkin**.
9. In the **GUI Slide Dialog (Script)**, locate the **OK** icon and set the **Size** to **2**, the first element to the texture named checkmark, and the second element to the texture named checkmarktouched.
10. In the **GUI Slide Dialog (Script)**, locate the **Cancel** icon and set the **Size** to **2**, the first element to the texture named xmark, and the second element to the texture named xmarktouched.

Set up the OptionsDialogContent and OptionsDialogControl

Now, we need to create a new script to render the content of the **OptionsDialog**, which is as follows:

1. In the project hierarchy, create a new folder for the **OptionsDialog**.
2. In the **OptionsDialog** folder, create a new **Javascript** for the **OptionsDialogContent** and a second new **Javascript** for the **OptionsDialogControl**.

The project Hierarchy should now look as follows (your items may have different names):

OptionsDialog folder

OptionsDialogContent.js

OptionsDialogControl.js

1. Now, open the **OptionsDialogContent.js** file and add the following script code:

```
// The wrapper should send an ok
// message to this game object when
// the ok button is pressed
function ok()
{
Debug.Log("Options Dialog: OK");
}
// The wrapper should send a cancel
// message to this game object when
// the cancel button is pressed
function cancel()
{
Debug.Log("Options Dialog: Cancel");
}
// When enabled, the init function should
// be called
function init()
{
Debug.Log("Options Dialog: Init");
}
// Called automatically by Unity3D and
// used to initialize the dialog
// content
functionOnEnable()
{
init();
}
functionrendercontent(a_boxSize : Vector2)
{
varl_box_x : int = 0;
varl_box_y : int = 0;
varl_rect : Rect = Rect (l_box_x,
```

```
l_box_y,
a_boxSize.x,
a_boxSize.y);
GUI.Box (l_rect, GUIContent("Options"));
}
```

2. Now, open the **OptionsDialogControl.js** file and add the following script code:

```
varoptionsDialog : GameObject;
function show()
{
optionsDialog.SetActiveRecursively(true);
varl_dialogWrapper : GUISlideDialogWrapper =
optionsDialog.GetComponent(GUISlideDialogWrapper);
l_dialogWrapper.setSlideOut(false);
l_dialogWrapper.displaydialog();
}
```

3. Now back in the game hierarchy, open the **Player | Buttons**, and locate **GUIButtonOptions**.
4. Click on **GUIButtonOptions** to select it.
5. In the inspector, locate the **GUI Slide Button (Script)**.
6. In the **GUI Slide Button (Script)**, locate the **Message GO** variable, and set it to your **OptionsDialogControl** game object.
7. Finally, press **play** in the inspector and then click the **options** button. A dialog like the following image should slide onto the screen:

Set up the messages

If the dialog is still displayed, press the **stop** button in the Unity3D editor to stop playback. If you played with the dialog, you will have noticed that while the buttons indicate they have been pressed, not much else happens.

This is because we need to send messages to make things happen.

To make the dialog's **Cancel** and **Ok** buttons work, we need to add, at a minimum, the **Cancel** and **Ok** messages to the **GUISlideDialogWrapper**.

1. In the game hierarchy, select the **OptionsDialog** game object.
2. In the inspector, locate the **GUI Slide Dialog Wrapper (Script)**.
3. In the **GUI Slide Dialog Wrapper (Script)**, locate the **Ok Message, Ok Message Type, Ok Send Delay**, and **Ok Message GO** variables. Modify them all, so that the **Size** is **1**.
4. In the first element of the **Ok Message**, set the value to **ok**.
5. In the first element of **Ok Message Type**, set the value to **SendMessage**.
6. In the **GUI Slide Dialog Wrapper (Script)**, locate the **Cancel Message, Cancel Message Type, Cancel Send Delay**, and **Cancel Message GO** variables. Modify them all, so that the **Size** is **1**.
7. In the first element of the **Cancel Message**, set the value to **cancel**.
8. In the first element of **Cancel Message Type**, set the value to **SendMessage**.
9. Now press **play** again; this time the **OK** and **Cancel** buttons will cause the dialog to close.
10. Press **Stop**.
11. In the **GUI Slide Dialog Wrapper (Script)**, locate the **Ok Message, Ok Message Type, Ok Send Delay**, and **Ok Message GO** variables. Modify them all, so that the **Size** is **1**.
12. In the first element of **Ok Message GO**, set the value to your **OptionsDialogContent** game object.
13. In the **GUI Slide Dialog Wrapper (Script)**, locate the **Cancel Message, Cancel Message Type, Cancel Send Delay**, and **Cancel Message GO** variables. Modify them all, so that the **Size** is **2**.
14. In the first element of **Cancel Message GO**, set the value to your **OptionsDialogContent** game object.
15. Now press **play**.
16. Press the **options** button to display the options dialog.
17. Press the **cancel** button
18. Press the **options** button to display the options dialog.
19. Press the **Ok** button
20. Now look at the game console; it should look like the following screenshot:

> ## Note
>
> If we enable the **Clear on play** option in the console, it will help us to see the results, since the console messages prior to playing will be cleared.

Console

Clear | Collapse | Clear on play | Error pause

Options Dialog: Init
UnityEngine.Debug:Log(Object)

Options Dialog: Init
UnityEngine.Debug:Log(Object)

Options Dialog: Cancel
UnityEngine.Debug:Log(Object)

Options Dialog: Init
UnityEngine.Debug:Log(Object)

Options Dialog: OK
UnityEngine.Debug:Log(Object)

Task: script some dialog contents

Dialog content can be anything. We can look in the Unity3D manuals and find a myriad of sliders, buttons, and images that you can put in a GUI. What we decide to put in our dialog really depends on our game.

This is an opportunity for us to look into the Unity3D GUI system, and add some things into the **GUIOptionsDialogContent.js** script to see what kind of results can be achieved.

We can find some GUI basics on the Unity3D website at:

http://unity3d.com/support/documentation/Components/gui-Basics.html

As an example, you can find the game object named **ZZZ_OptionsDialogFull** in the game hierarchy. If you assign that game object to the **OptionsDialog** variable of the **AAA_OpitonsDialogControl**, you will see an **Options** dialog as shown in the following image. But this is just one example of what can be done:

Challenge: making things flexible in the editor

One of the things about using an array of messages, message types, and message game objects is that it allows us to be very flexible in the editor without the need to be changing scripts. One of the things we have noticed is that when a dialog is displayed, the GUI interface is still active. It can be distracting to have menus responding to events when a dialog is displayed.

The way to manage this is to have those items disabled when a dialog is displayed and enabled when a dialog is closed.

Considering each of the buttons and the messaging system available, devise a way in which the editor flexibility of the messaging system can be used to enable and disable the rest of the user interface.

The following are some hints:

1. **GUISlideButtons** have a flag that automatically disables them after they are pressed.
2. **GUISlideButtons** respond to `disable()` and `enable()`.
3. **GUISlideButtons** respond to `hide()` and `show()`.
4. The **AAA_ShowTexture** script responds to `enable()` and `disable()`.
5. You can send a **BroadcastMessage** to the parent of a group of game objects and they will all receive that message.

Challenge: creating custom GUI elements

Unity3D provides us with some very basic GUI elements, but there are a lot of ways in which we can make our GUI dialogs more interesting by creating some custom elements. Try and spice up the GUI by taking advantage of Unity3D's flexibility.

The following are some hints:

1. **GUISlideButtons** accept two textures for the icon.
2. **GUISlideDialogWrapper** can use any **GUISkin**, and there are lots of skins to be found on the Internet. Some extra GUISkins can be found on the Unity3D website here: http://unity3d.com/support/resources/assets/extra-gui-skins.html
3. **GUISlideDialogWrapper** accepts two textures for the icon.
4. The `rendercontent()` function of the game object that renders the dialog content can do anything, and it is limited only by our imagination.
5. Sounds make dialogs interesting. Consider how a sound could be played when a dialog slides onto the screen, and what kind of sound may be interesting for game players.

Summary

In this chapter, we have covered the following:

- How to make GUIs that work on Unity3D for iOS devices, regardless of the device screen resolution
- How to create flexible scrolling GUI dialogs that can send any number of messages to any number of game objects to flexibly implement anything we would need in a GUI dialog system
- How to set up and configure the flexible dialog (as in dialog boxes or windows, not as in player conversations) system
- Some ideas on how the dialog system can be expanded to suit the needs of our games using more custom dialog content

In Chapter 6, *Preferences and Game State*, we will explore the concepts of saving and restoring a player's preferences and a game's state.

Chapter 6. Preferences and Game State

Our games need to remember things. They need to remember who was playing, what the player was carrying, how many game credits the player had, what trophies the player had, where the player was in the game, the status of the enemies in the game, and the list goes on.

Player preferences and game state are areas where Unity3D does not provide a lot of support. The basic ability to set and get integers, floats, and strings, leaves a lot to be desired for all but the most basic of games.

So, rather than relying on the basic built-in system, we are going to create our own system for managing preferences and game state. In this chapter, we will learn the following:

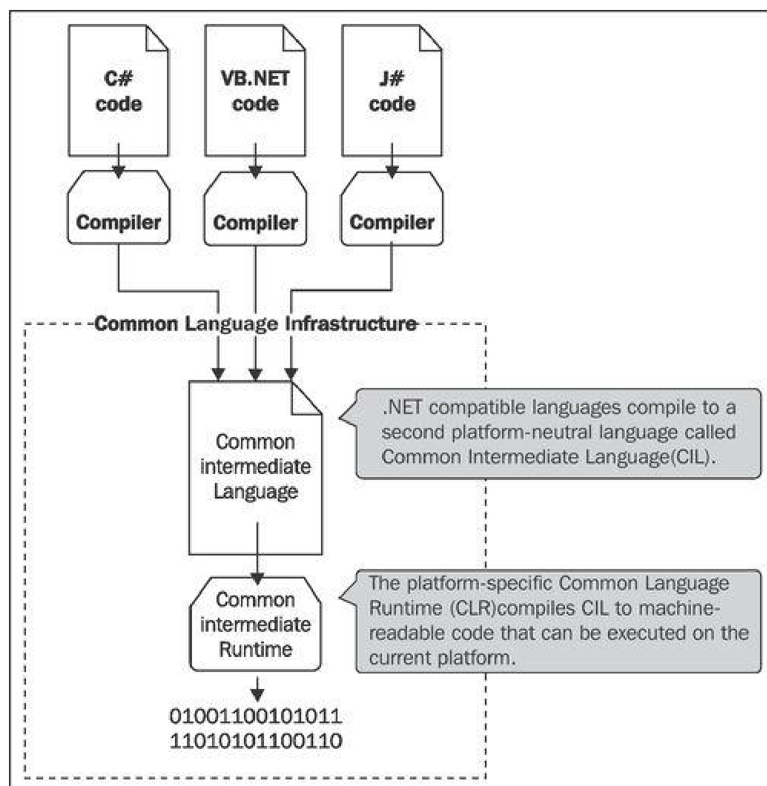
- How the .NET framework can be leveraged to create persistent data
- How property lists can be used to store and retrieve sophisticated information sets
- How to store and retrieve information for multiple players

Introducing .NET

The **.NET framework** is a Microsoft-designed set of programmatic interfaces, functions, and runtime architecture that provides a rich set of language-independent data types and core classes that we can use in Unity3D to interact with low-level system resources.

When Microsoft designed .NET for Windows, they took great care to create architecture in which multiple programming languages are compiled into a common language, and once the common language components were created, the remaining common runtime architecture components were identical, regardless of the original programming language.

The following image shows the common language architecture of this design:



In addition to defining this architecture, Microsoft took it one step further and created an open standard that published the complete architecture specification. This step was essential to opening the .NET architecture up to third parties, and most importantly, to the open source community.

Rather than using the Microsoft implementation of .NET, Unity3D uses an open source project, called **Mono**, which implements .NET, including cross-platform assemblies, so that Unity3D can access .NET functionality on multiple operating systems.

Because Mono/.NET is big, we need to focus on just the components that we are going to use to access our preferences and maintain our game state. The parts that we need to understand are as

follows:

- `System.Environment`
- `System.Collections`
- `System.IO`
- `System.xml`

System.Environment

This Mono/.NET namespace contains information about the current execution environment. We will use `System.Environment` to locate special folders, where we need to store our preferences and game state information.

System.Collections

This Mono/.NET namespace contains classes for storing collections of objects. The objects that we are most interested in are Hashtables and ArrayLists. A `Hashtable` is a dictionary that allows us to store and retrieve values using keys. An `ArrayList` allows us to store a list of items and enumerate over the list.

System.IO

This Mono/.NET namespace contains classes for reading and writing data streams and interacting with the filesystem by getting file and directory (folder) information, as well as creating files and directories. We will use this class to combine paths, to read and write files found on those paths, and create directories where needed.

System.xml

This Mono/.NET namespace contains classes for reading, writing, and manipulating **Extensible Markup Language** (**XML**) documents. We will use this class to read and write XML format `plist` files. A comprehensive discussion of XML is beyond the scope of this book, but we can find complete details on this web page:

http://en.wikipedia.org/wiki/XML

Understanding property lists

In Mac OS X, there is a special kind of file that ends with the extension `plist`. These `plist` files, or property lists, can contain binary (now the default) or XML (which we are going to use with Unity3D) data to represent serialized objects of specific data types. By definition, a property list can contain the following data types:

- `NSNumber`
- `NSString`
- `NSArray`
- `NSDictionary`
- `NSDate`
- `NSData`

Once again a complete discussion of property lists is beyond the scope of this book, but we can find a complete discussion here:

http://en.wikipedia.org/wiki/Property_list

The `Array` and `Dictionary` data types can be nested, so you can have an array of dictionaries, dictionaries that contain arrays, and dictionaries that contain dictionaries. The number contains an internal representation of a number that can be converted to any standard (int, float, double, and so on) numeric type. The string contains a string in a defined encoding, typically UTF-8, but many other encodings are supported.

For now, we are going to ignore the `NSDate` and `NSData` components of a property list, other than just saying one is a date and the other can store bytes that represent any kind of data that one would like.

A sample XML property list file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>key_dictionary</key>
<dict>
<key>Item2</key>
<string></string>
<key>Item1</key>
<string>Item1</string>
</dict>
<key>key_number</key>
<integer>0</integer>
<key>key_string</key>
<string>A String</string>
<key>key_array</key>
<array>
```

```
<string>Item1</string>
<string>Item2</string>
</array>
</dict>
</plist>
```

The reason we are interested in property lists is because there is a very convenient script available from the **Unify Community Wiki** called `PropertyListSerializer` that can take a .NET `Hashtable`, which contains other Hashtables, ArrayLists, Strings, and numeric types, and serialize it to a `plist` file or de-serialize it from a `plist` file. You can find the original page for `PropertyListSerializer` here:

http://www.unifycommunity.com/wiki/index.php?title=PropertyListSerializer

This script, combined with the basic .NET data types, will allow us to create much more sophisticated game preferences and game save states than we could ever have hoped to achieve by using the very basic preferences class available in Unity3D.

The `PropertyListSerialzer` script does require the `System.xml` class from the .NET framework to function, so while we are not going to use that framework directly, it is required for our `plist` system to work.

Sometimes, we find a great script like `PropertyListSerializer` that does exactly what we want, and we simply want to include it in our project.

Unfortunately, with Unity3D for iOS, it is not always the case that code can be easily re-used, because JavaScript in Unity3D for iOS is strongly typed.

The solution was to add a few lines of code, shown in the following code fragment, to make it strongly typed.

Note

There can be significant performance implications to using scripts that are weakly typed, so it is a good idea to use strongly typed code in scripts that are executed in every frame.

The original code is as follows:

```
else if (typeof(value) == DateTime)
{
// Dates need to be stored in ISO 8601 format
valNode = node.OwnerDocument.CreateElement("date");
valNode.InnerText =
value.ToUniversalTime().ToString("o");
node.AppendChild(valNode);
for (element in value.val.Keys)
{
// Recursively attempt to add/replace the elements of the value Hashtable to the
dict's value Hashtable.
// If this fails, post a message stating so and return false.
if (!AddKeyValueToDict(dict[key], element, new ValueObject(value.val[element])))
{
```

```
Debug.LogError("Failed to add key value to dict: " + element + ", " +
value.val[element] + ", " + dict[key]);
return false;
}
```

The following is the strongly typed code:

```
else if (typeof(value) == DateTime)
{
var l_dateTime : DateTime = value;
// Dates need to be stored in ISO 8601 format
valNode = node.OwnerDocument.CreateElement("date");
valNode.InnerText =
l_dateTime.ToUniversalTime().ToString("o");
node.AppendChild(valNode);
for (element in (value.val as Hashtable).Keys)
{ // Recursively attempt to add/replace the elements of the value Hashtable to the
dict's value Hashtable.
// If this fails, post a message stating so and return false.
if (!AddKeyValueToDict(dict[key], element, new ValueObject((value.val as
Hashtable)[element])))
{
Debug.LogError("Failed to add key value to dict: " + element + ", " + (value.val as
Hashtable)[element] + ", " + dict[key]);
return false;
}
```

Handling different game players

Sometimes, when we write a game, we will want to provide the option of having different people play the game while retaining the preferences and game state for each player. The easiest way to manage this is to ask each player for his/her name and store a different set of information in a folder based on each player's name.

Typically, the directory structure that we create will look something like the following lines of code:

```
~/Library/Preferences

com.domain.game

playername1

savefiles1

playername2

savefiles1

savefiles2
```

By organizing save files like this, it is easy (with the exception of two people with the same name) to handle different game states for different players.

A second option would be to create a single `Hashtable`, and use a similar structure, but of nested Hashtables, keyed by the players' names.

The advantage of using separate files, rather than a large `Hashtable`, is that there is a limited amount of time to write game data to disk, when a player presses the home button on an iOS device, so it's important to keep the amount of information that needs to be written to disk, when a game quits, to the absolute minimum.

Deciding where to put files

We want to use the .NET `System.Environment` class to locate specific folders, where our preferences and game state information will be stored. Specifically, we are going to create a globally accessible static function to return a string that contains the desired path. The following script shows the code that we will use to obtain the path we need:

```
// The file name for the options
static var optionsFileName : String =
"com.burningthumb.3dalarm.options.plist";
// The path to the Preferences folder
static var preferencesFolderPath : String =
"Library" + Path.DirectorySeparatorChar +
"Preferences";
// This function returns the path
// to the special folder where
// files should be stored
static function PathToSpecialFolder () : String
{
// The string to return
var path : String;
// On Windows return the
// LocalApplicationData folder
if (Application.platform ==
RuntimePlatform.WindowsPlayer ||
Application.platform ==
RuntimePlatform.WindowsEditor)
{
path = System.Environment.GetFolderPath(
System.Environment.SpecialFolder.LocalApplicationData);
}
// On iOS and Mac OS X return the
// ~/Library/Preferences folder
else
{
path = Path.Combine(System.Environment.GetFolderPath(
System.Environment.SpecialFolder.Personal),
preferencesFolderPath);
}
return path;
}
```

Once we have located the correct special folder, we may also want to create a subfolder at that location for our game, depending on how many files we want to store. In either case of storing files or creating folders, we will always use the reverse domain name notation to avoid naming conflicts. In our case, we use `com.burningthumb` as the prefix for everything we create. Other developers will, similarly, use their own reverse domain names to avoid filename conflicts.

The following script fragment shows how we will invoke the `PathToSpecialFolder` method, and then append a specific filename to the special path using the .NET method `Path.Combine()`:

```
var l_plistfile = Path.Combine(PathToSpecialFolder(),
"com.burningthumb.3dalarm.options.plist");
```

Task: load and save the players' preferences

The first thing that we need to do before loading player preferences is to think about the default values. For example, when the game is launched for the very first time, we could have the game volume off, set to the maximum, or set to something in between. If we decide that we want the game to start with the volume set to half of the maximum, we need a way to specify that for our game.

One way we can do it is to hardcode the default. This is a very common way for defaults to be specified, and simply requires us to check for the presence of the value. If it is not found, then set it to half and write it back out. The next time the game is run, the value will always be found.

One option would be to use default values hardcoded into a script. A second option, and the one that we want to use, is to create a `plist` file that contains our default values and include that `plist` file in one of the games resource folders known as the data path. Then, rather than hardcoding the initial values of the `Hashtable` in a script, we can read the initial values from the `plist` file.

The problem with hardcoding defaults is that it's messy, inflexible, and error prone, and simply not all that great an idea.

A better solution would be to have a generic system that allows us to create one or more `plist` files and have them automatically loaded for us when our game starts. The most challenging, and least documented, part of this system is how to get the `plist` files into the correct folder on our game.

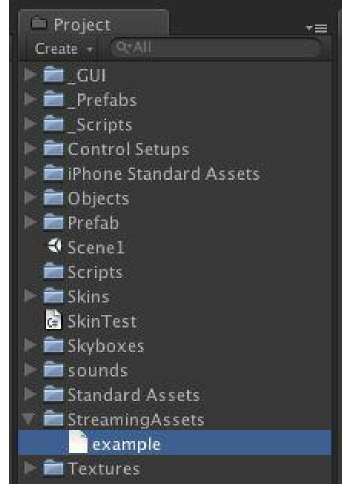
Unity3D provides a variable that we can use to get the path to our application's data. That variable is called `Application.dataPath`. Unfortunately, that variable doesn't get us to the correct folder; it gets us to the folder one level above the correct folder. The next unfortunate issue is that the correct folder has a different name depending upon which platform our game is deployed on.

Since we are only concerned with running our game on the iOS devices or in the Unity3D editor, we can limit the scope of our scripting to those two platforms. However, if we wanted our game to run on all the supported platforms, we would need to add additional code for each platform that we want to support.

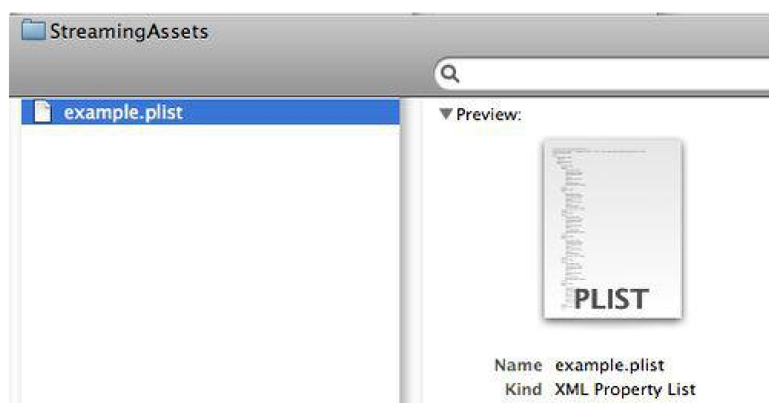
The steps that we follow to add custom `plist` files to our Unity3D project are as follows:

1. Create a **StreamingAssets** folder in the game hierarchy
2. Put our `plist` files in the **StreamingAssets** folder

The following screenshot shows an example of a custom `plist` file included in a Unity3D project in the **StreamingAssets** folder:



And if we look at the **StreamingAssets** folder, in the **Finder**, we can see clearly that it contains a `plist` file. This is shown in the following screenshot:



Now that we know where to put our `plist` files, we need some script code to automatically find and load those `plist` files into Hashtables.

The following script fragments, from `Globals.js`, show the script code required to automatically load all the `plist` files and create the corresponding Hashtables. The Hashtables are accessed using a key that is the same as the filename, so, for example, the key to access the `Hashtable` loaded from `example.plist` is `example`. This is shown as follows:

```
// This is the global Hashtable
// that will contain all of the
// property lists loaded into
// Hashtables from the
// StreamingAssets folder
// of the game hierarchy
static var plists : Hashtable;
// These are the .NET Hashtables
// that will contain the
// players options
```

```
static var optionsHash : Hashtable;
static var alarmHash : Hashtable;
static var dayHash : Hashtable;
// The file name for the options
static var optionsFileName : String =
"com.burningthumb.3dalarm.options.plist";
// The path to the Preferences folder
static var preferencesFolderPath : String =
"Library" + Path.DirectorySeparatorChar +
"Preferences";
// The key for the default options plist
static var keyDefaultOptions : String = "defaultoptions";
// The key for the alarms Hashtable
static var keyAlarms : String = "alarms";
function Awake()
{
// A local Hashtable used for each file as it is
// loaded from disk
var l_plist : Hashtable;
// The global Hashtable that contains a Hashtable
// for each plist file that is loaded
plists = new Hashtable();
// The path to the Raw assets from the
// StreamingAssets folder
//
// In the Editor it will be <project folder>/AssetsAssets/StreamingAssets
// On an iOS device it will be <app bundle>/<AppName.app>/Data/Raw
var l_pathToRawAssets : String;
if (Application.platform == RuntimePlatform.IPhonePlayer)
{
l_pathToRawAssets = Path.Combine((Application.dataPath), "Raw");
}
else
{
l_pathToRawAssets = Path.Combine((Application.dataPath), "StreamingAssets");
}
// Get a list of all files from StreamingAssets
// that end in .plist
var plistFiles =
Directory.GetFileSystemEntries(l_pathToRawAssets, "*.plist");
// Iterate over the list of .plist files
for (var plistFile : String in plistFiles)
{
// Create a new Hashtable for this file
l_plist = new Hashtable();
// Load this .plist file into a Hashtable
PropertyListSerializer.LoadPlistFromFile(plistFile, l_plist);
// Add the new plist to the global list of
// plists with a key that is the filename
plists.Add(Path.GetFileNameWithoutExtension(plistFile), l_plist);
}
// Load the options for this player
LoadPlayerOptions();
// Create a global reference just to
// the alarms Hashtable
alarmHash = optionsHash[keyAlarms];
}
static function LoadPlayerOptions()
{
var l_plistfile = Path.Combine(PathToSpecialFolder(), optionsFileName);
```

```
if (File.Exists(l_plistfile))
{
optionsHash = new Hashtable();
PropertyListSerializer.LoadPlistFromFile(l_plistfile, optionsHash);
}
else
{
DefaultPlayerOptions();
}
}
// This function will load the
// player options and if the
// options plist file is not found
// it will load the options
// from the defaults file
// found in the StreamingAssets
// folder
static function LoadPlayerOptions()
{
// The location of the games
// options file
var l_plistfile =
Path.Combine(PathToSpecialFolder(), optionsFileName);
if (File.Exists(l_plistfile))
{
// If the games options file
// exists, load it into a new
// Hashtable
optionsHash = new Hashtable();
PropertyListSerializer.LoadPlistFromFile(l_plistfile, optionsHash);
}
else
{
// If the game options file
// does not exist, load it from
// the default Hashtable
DefaultPlayerOptions();
}
}
// This function loads the
// player's options with default
// values that were read in
// from a plist file
static function DefaultPlayerOptions ()
{
optionsHash = plists[keyDefaultOptions];
}
```

Finally, we need some scripting to save the options, once the player has changed them. The code fragment for saving the options after the player has changed them is shown as follows:

Note

Please note the following code assumes that the values in the `Hashtable` have been updated by the player using a dialog prior to being invoked.

```
// This function will write the options
// Hashtable to a file in the users
// preferences folder
static function SavePlayerOptions()
{
// The full path to the plist file
var l_plistfile =
Path.Combine(PathToSpecialFolder(),
optionsFileName);
// Write the Hashtable out to the
// file
PropertyListSerializer.SavePlistToFile(l_plistfile,
optionsHash);
}
```

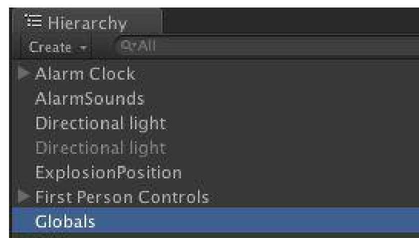
Using the Globals.js script

If you look in the code folder that accompanies this chapter, you will find the completed script named `Globals.js`.

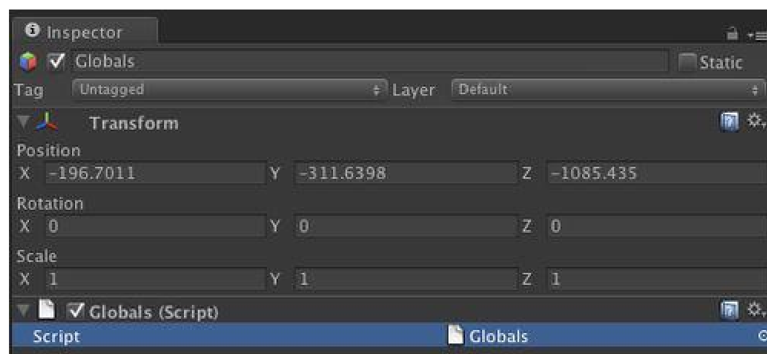
To use the script, we need to do the following:

1. Open the `Chapter 6 Unity Project` folder.
2. Open the `Assets` folder.
3. Drag the `Globals.js` file into the `_Scripts` folder.
4. Double-click **Scene1.unity** to open the scene.
5. Create a new game object in the Unity3D **Hierarchy**.
6. Rename the new game object to **Globals**.
7. Attach the `Globals.js` script to that game object.
8. Open the `Globals.js` script and change the value for the static variable named `optionsFileName` to anything we want for our game.

It will look like the following screenshots. The first screenshot shows the game hierarchy as follows:



The second screenshot shows the **Globals** game object in the Unity **Inspector** as follows:



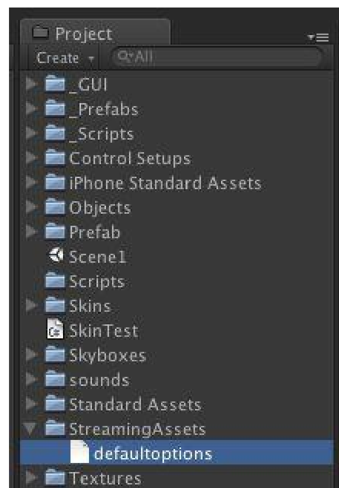
Creating the defaultoptions.plist file

Since we have decided to create a `plist` file that contains our **defaultoptions**, and our `Globals.js` script expects that script to be named **defaultoptions**, we need to do that now. If we look at the original hardcoded solution, we can see the structure of the `plist` file that is expected. There are many ways to create a `plist` file; the easiest is to use a simple text editor and save the file as `.plist` instead of `.txt`.

The steps we need to follow to create the `plist` file are as follows:

1. Look at the hardcoded example as a reference.
2. Use a text editor to create the XML that maps to the coded structure.
3. Save the file as **defaultoptions.plist**.
4. In the Unity3D project, create a folder named **StreamingAssets**.
5. Put the **defaultoptions.plist** file in the **StreamingAssets** folder.

It will look like the following screenshot:



If you look in the code folder that accompanies this chapter, inside the subfolder named `ZZZ_Answers`, you will find the complete **StreamingAssets** folder that contains the **defaultoptions.plist** file.

Provide a GUI dialog for the options

Finally, we need to create a script that will display the options and allow the player to change the options.

To save us some time, the **GUIOptionsButton, GUIOptionsDialog**, and **GUIOptions** (content) game objects, and scripts have already been added to the project scene. The one part of the **GUIOptions** script that we really want to look at is the part that saves the modified options by invoking the **Globals** as follows:

```
// This function is called
// when the OK button
// is pressed
function ok()
{
// Save the player options
Globals.SavePlayerOptions();
// Reset the clock
theAlarm.GetGlobalAlarmTime();
theClock.GetGlobalAlarmTime();
}
```

Tip

`ZZZ_Answers`

If you prefer the quick solution, open `Scene1.unity` and import `ZZZ_Options.unitypackage` from the `ZZZ_Answers` folder. Once we have done that, we then need to instantiate the **Globals** prefab. After doing that, the project will work and the options will function as expected.

Task: create a GUI to get the players' names

This task depends on the load and save functions performed on the players' preferences task. We must complete that task successfully prior to beginning this task.

In our game, we are going to allow different players to sign in, and we are going to maintain a different set of options for each player.

To accomplish this, we are going to ask the players to enter their name when the game is launched, and we are going to create a separate options `plist` file for different players.

Add the player name variable to Globals.js

In order to do this, the very first thing we need is a global location to save the players' names. To do that, the following steps need to be performed:

1. Open the `Chapter 6 Unity Project` folder.
2. Open the `Assets` folder.
3. Double-click **Scene1.unity** to open the scene.
4. In the **Project** hierarchy, locate the `_Scripts` folder.
5. Double-click **Globals** to open it in the Unitron or MonoDevelop editor.
6. Add a new static variable called `playerName`.
7. Save the script.

The script code that needs to be added to `Globals.js` is shown as follows:

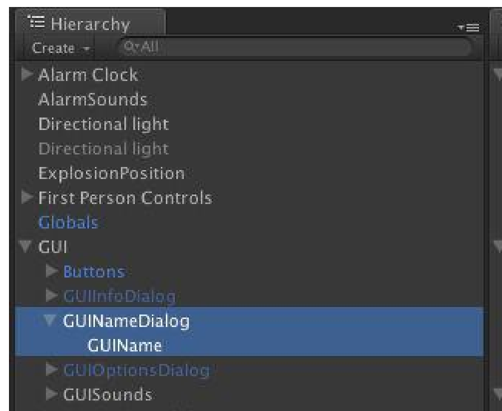
```
// This is the name of the current player
static var playerName : String;
```

Create the new GUI Dialog game objects

If we are going to ask the players to enter their names, we will need a GUI Dialog, which is essentially a box that is displayed with a text entry field where the players can type their names. To create the new GUI Dialog game objects, do the following:

1. Open the `Chapter 6 Unity Project` folder.
2. Open the `Assets` folder.
3. Double-click **Scene1.unity** to open the scene.
4. In the game **Hierarchy**, locate the GUI game object and open it.
5. Create two empty game objects.
6. Rename the first empty game object to **GUINameDialog**.
7. Rename the second empty game object to **GUIName**.
8. Drag **GUIName** in to **GUINameDialog**.
9. Drag **GUINameDialog** in to **GUI**.

At this point, the game **Hierarchy** should look like the following screenshot:



Add the GUI dialog scripts

Game objects don't do anything without scripts. To add the scripts to the **GUINameDialog** game objects, do the following:

1. In the **Editor Project** tab, locate the `_Scripts` folder and open it.
2. Create a new JavaScript file.
3. Rename the new JavaScript file from **NewBehaviorScript** to **GUIName**.
4. Double-click **GUIName** to open it in the Unitron or MonoDevelop editor.
5. Enter the following script content into **GUIName:**

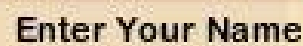
   ```
   function rendercontent(boxSize : Vector2)
   {
   var l_box_x : int = 0;
   var l_box_y : int = 0;
   GUI.Box (Rect (l_box_x, l_box_y, boxSize.x, boxSize.y), GUIContent("Enter Your
   Name"));
   }
   ```

6. Save **GUIName**.
7. Quit Unitron.
8. Drag the **GUIName** script onto the **GUIName** game object.
9. Drag the **GUISlideDialogWrapper** script onto the **GUINameDialog** game object.
10. In the **Editor Inspector** tab, drag the **GUIName** game object to the **Content GO** variable.
11. In the **Editor Inspector** tab, set the **Box Size X** and **Y** values to **1**.
12. In the **Editor Inspector** tab, set the **Start Pos X** and **Y** values to **0.5**.
13. In the **Editor Inspector** tab, set the **End Pos X** and **Y** values to **0.5**.
14. In the **Editor Inspector** tab, uncheck the **Disable When In Start Pos** checkbox.
15. In the **Editor Inspector** tab, uncheck the **Fade** checkbox.
16. In the **Editor Inspector** tab, set the **Skin** value to **CustomSkin**.
17. In the **Editor Inspector** tab, set the **OK Icon Size** to **2**.
18. In the **Editor Inspector** tab, set the **OK Icon Size Element 0** to **checkmark**.
19. In the **Editor Inspector** tab, set the **OK Icon Size Element 1** to **checkmarktouched**.
20. In the **Editor Inspector** tab, uncheck the **Disable When In Start Pos** checkbox.

At this point, when you run the game in the Unity Editor, the screen should be covered in a dialog that looks like the following image:

At this point, we need to add the content to the dialog, so that it asks for the players' names. In this case, we will just use a simple text entry box and some text labels to tell the player, if the name has already been used.

The updated **GUIName.js** script to accept the players' names results in the new dialog, as shown next:



Finally, we need to configure the **GUINameDialog** to send the ok and disable messages (in Unity3D, sending a message to an object causes the functions with the same name as the message to be executed in any script attached to the object) to itself, when the **OK** button is pressed. Basically this results in the script executing the ok function followed by the disable function when the **OK** button is pressed. To do that, we need to perform the following steps:

1. Select the **GUINameDialog** game object in the **Hierarchy** tab.
2. In the **Inspector** tab, set the **Ok Message Size** to **2**.

3. In the **Inspector** tab, set the **Ok Message Element 0** to **ok**.
4. In the **Inspector** tab, set the **Ok Message Element 1** to **disable**.
5. In the **Inspector** tab, set the **Ok Message Type Size** to **2**.
6. In the **Inspector** tab, set the **Ok Message Type Element 0** to **BroadcastMessage**.
7. In the **Inspector** tab, set the **Ok Message Type Element 1** to **SendMessage**.
8. In the **Inspector** tab, set the **Ok Message Delay Size** to **2**.
9. In the **Inspector** tab, set the **Ok Message GO Size** to **2**.

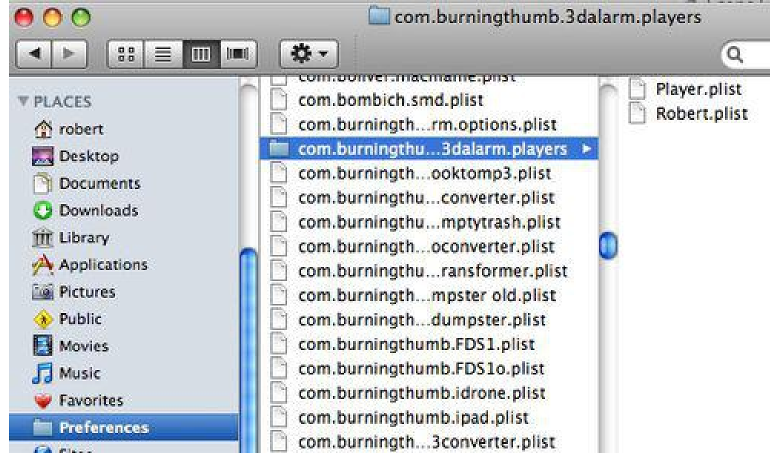
The following screenshot shows a fully configured wrapper for the **GUINameDialog OK** game variables:

Now, each time the game runs, it will ask the players to enter their names.

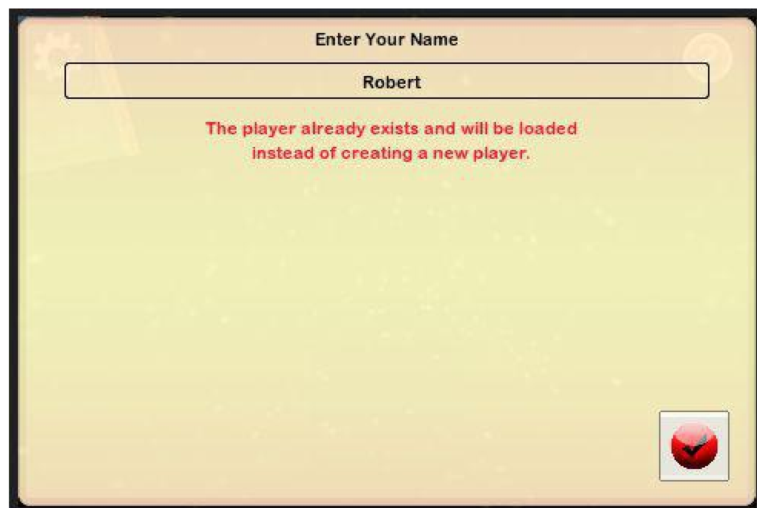
> ## Note
>
> We are keeping this simple for example purposes. In our final game, we would expand this logic to include picking a player's name from a list of already saved players.

If the player decides to not enter a name, they will get the default name of **Player**. If the player enters a name that they entered before, the game will load their existing options.

If we run the game a couple of times in the Unity3D editor, and choose a different player name, we can look in the **Finder**, in our **Preferences** folder, and see that, as expected, multiple `plist` files are being created, as shown in the following screenshot:



And if we change some options and rerun the game with the same player name, we see that not only does the dialog tell us the name is recognized, but also that the options are retained for that player. This is shown in the following two images. The first image shows the player named **Robert** already exists and will be loaded when the checkmark button is pressed:



The second image shows that the player named **Robert** has enabled the **24 hour clock** in the **Options** panel:

If you look in the **Hierarchy** tab, under the **GUI** game object, inside the `ZZZ_Answers` game object, you will find a solution dialog that you can activate and examine to see how it's all done.

Challenge: load and save the player's game state

Loading and saving options typically occur in response to the game player activating a GUI, making some changes, and either accepting or rejecting those changes.

Loading and saving game state can use the same paradigm where the player explicitly saves the game, it can occur at save points, which is essentially the same thing with the additional requirement of some achievement or progress, or it can occur in real time.

Your challenge is to save the player's game position in real time, and save it such that when the game is quit and re-launched, the player is restored to the same position.

Here are some hints:

- When an iOS game receives the quit notification, it has a limited amount of time to perform the saving of a state before the iOS terminates the game. As such, we need to maintain the game state in memory (probably in a `Hashtable)`, so that it can be quickly written out when the quit notification is received. While this may not be important when the only thing we are saving is the player's position, it becomes critically important when the amount of information that we need to preserve of a game state would be time consuming to gather after receiving the quit notification.
- An iOS game project needs to be set up correctly, in the player settings, to exit on suspend, so that the game is quit when the player presses the home button.

Summary

In this chapter, we have covered:

- The fundamentals of the .NET architecture
- The fundamentals of property lists
- How to use property lists and .NET to save and load non-trivial sets of game information using Unity3D
- How and where to save and load information for a game that supports multiple players

In Chapter 7, *The Need for Speed*, we will use a four-wheeled vehicle game to explore the concepts related to faster moving players, physics for different sized objects, fast and realistic foliage, and different culling methods available in Unity3D.

Chapter 7. The Need for Speed

Games don't always have a character as the main player. Sometimes, games have vehicles as the main player and sometimes characters ride in vehicles, during parts of the gameplay. Unity3D provides support for wheeled vehicles through a special wheel collider that allows us to create vehicles with realistic physics.

Once we understand the basics of using a four-wheeled vehicle in Unity3D, it becomes a straightforward task to create any kind of vehicle for any kind of game in which a vehicle can add to the enjoyment of the gameplay. Whether it's a car, a tank, or a motorcycle, Unity3D can make it go. In this chapter, we will learn the following:

- How to create a four-wheeled vehicle
- How to manage physics for both large and small objects
- The essentials of creating both realistic and fast foliage on iOS devices
- How to easily include different culling methods into our game

Adding a four-wheeled vehicle

No matter what kind of game genre we want to create for iOS devices, it is possible that we will want to include a vehicle of some kind in the game.

As with all things, before we create something from scratch, we want to look around and see what we can re-use.

For four-wheeled vehicles in Unity3D, there are really two choices that act as a starting point, both designed to deploy on the desktop version of Unity3D:

- Unity3D Car Tutorial
- JCar

Unity3D Car Tutorial

The Unity3D Car Tutorial can be found on the Unity3D website. At the time of writing, it could be found here:

http://unity3d.com/support/resources/tutorials/car-tutorial.html

While the Car Tutorial is comprehensive and can give you some great ideas, it is really not suitable for use on a mobile computing platform. You should certainly take a look at the tutorial and be familiar with its workings, but using it as a starting point for an iOS four-wheeled vehicle would require a significant amount of effort.

The Car scripts are quite useful, and we should read through them along with the third section of the Car Tutorial that describes how the Car scripts work, as gaining an understanding of that part of the project will provide valuable insight with respect to implementing a four-wheeled vehicle in Unity3D for iOS.

JCar

In late 2008 through early 2009, a Unity3D project called **JCar** and alternatively **NumberRacer** (a version of JCar for iPhone that uses accelerometer controls) was released by a company called **Ctrl-J**. Both of the projects are available for free, and the scripts can be used for both non-commercial and commercial projects with no license fees.

The original source files can be obtained from Ctrl-J on their website here:

http://www.ctrl-j.com.au/pages/jcarsrc.html

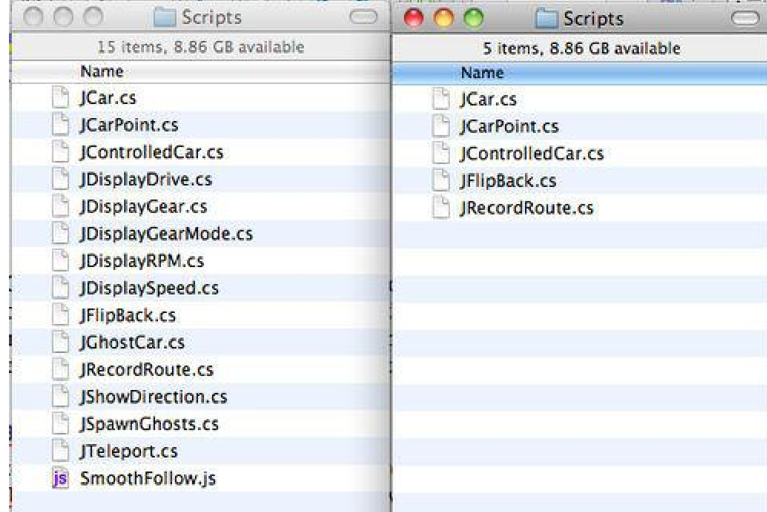
The Car scripts in these projects are ideal starting points for creating a four-wheeled vehicle on the iOS version of Unity3D.

Note

The JCar project contains many scripts, and while we can look at them to get a better understanding of the complete JCar system, we really only need five scripts to create a basic four-wheeled vehicle. The scripts that we need to include in our project are as follows:The JCar scripts are written in C#. As we have said before, while we don't need to be able to develop in C#, it is important to be able to read and understand C#, because many community resources for Unity3D are written in C#.

- `JCar.cs`
- `JCarPoint.cs`
- `JControlledCar.cs`
- `JFlipBack.cs`
- `JRecordRoute.cs`

The following screenshot shows two folders, the first containing the complete set of JCar scripts and the second containing the scripts that we need for our four-wheeled vehicle:



The other thing that we need to do is edit the `JControlledCar.cs` script to allow the use of an iOS virtual joystick to control the car instead of using the keyboard. Of course, we will leave the keyboard control in place as well, so that we can test our game using the editor. The following `diff` listing (diff is a command line tool that shows the differences between files. To use it, open the Terminal and type `diff`, followed by the two filenames) shows the changes that we need to make to the script:

```
5a6,10
// This allows us to attach a
// joystick for steering on
// iOS devices
public Joystick steeringStick ;
39c44,54
// This is used to get the joystick
// steering values
Vector3 steeringInput;
// Scale joystick input with rotation speed
steeringInput.x = steeringStick.position.x;
steeringInput.y = steeringStick.position.y;
steeringInput.z = 0;
steeringInput.Normalize();
47,49c62,74
steer = Input.GetAxis("Horizontal");
accel = Input.GetAxis("Vertical");
brake = Input.GetButton("Jump");
// We look at the keyboard only if the joystick values
// are zero
if ((steeringInput.x == 0) && (steeringInput.y == 0))
{
steer = Input.GetAxis("Horizontal");
accel = Input.GetAxis("Vertical");
brake = Input.GetButton("Jump");
}
else
{
accel = steeringInput.y;
steer = steeringInput.x;
}
```

Once the scripts are included in our project, and edited to include the changes to support the use of a joystick, we can create a basic scene that includes a vehicle, as shown in the following image.

The following image shows the vehicle in the Unity editor, the green box is a box collider, and the green circles are wheel colliders. It also shows the running game window with the two virtual joysticks, one on each side of the vehicle:

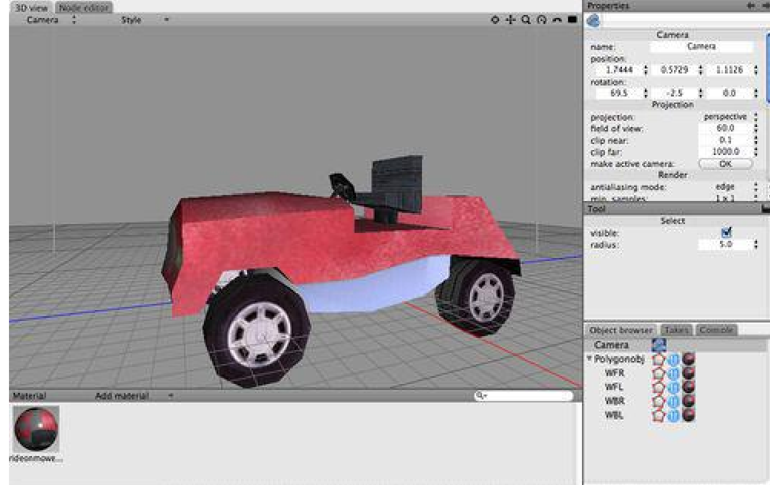
A JCar vehicle model

The model that we have chosen for our four-wheeled vehicle is a low-polygon version of a ride-on lawn mower. We could use any model that we want, and it could be very simple or very complex, but there is one thing that it must contain.

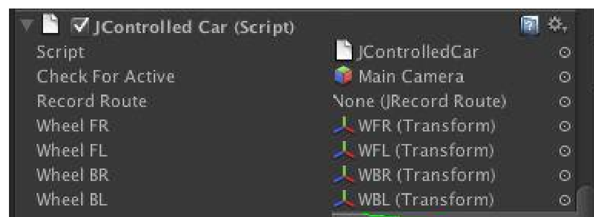
The model must contain four wheels.

Not only does it need to contain four wheels, but each wheel needs to be a separate mesh.

The following image shows the model in our 3D Modelling application with the separate wheel meshes. The wheel meshes have names such as, **WFR** (Wheel Front Right), **WFL** (Wheel Front Left), **WBR** (Wheel Back Right), and **WBL** (Wheel Back Left). We have chosen these names because they are the same names used in the `JControlledCar.cs` script:



After we add the script to the vehicle, the separate wheel meshes need to be dragged into the corresponding variables in the Unity3D **Inspector**, so that the wheels of the model are connected to the wheels in the mesh. The connected wheels are shown in the following screenshot:



Enabling JCar steering

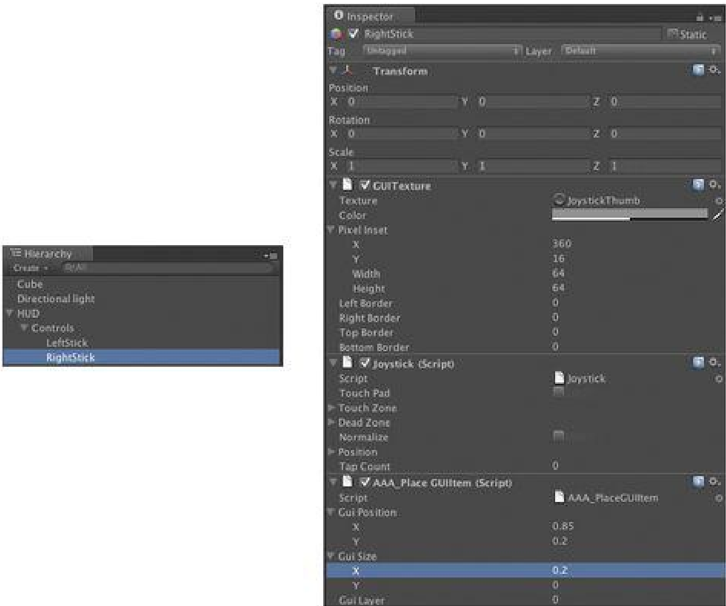
You will also notice in the previous screenshot that the `JControlled Car.cs` script looks for a variable named **Check For Active**. The script uses the game object provided in that variable to ensure that the vehicle is steerable only if that game object is active. In this case, we have used the **Main Camera**, so that the four-wheeled vehicle will be steerable, provided the **Main Camera** is active which it should be always. Of course, we can use any object that we like as the **Check For Active** object, and by activating or deactivating that object, we can control whether or not vehicle steering is enabled.

Note

To fully understand the steering model, we would need to look deeply into the concepts of acceleration, friction, inertia, and velocity implemented in the JCar scripts. Rather than going into detail, we are going to keep things very simple, so that we have a basic vehicle that works. Later we can play with all the variable settings to see how they change the behavior of the vehicle.

Connecting a Joystick to JCar

The final thing that we need to do is connect our **Joystick** object to the **JControlledCar**. The following screenshot shows our **Joystick** object and its configuration:



One thing that we may notice in the **Joystick** configuration is that the **GUI Size** in the **AAA_Place GUIItem** script has an **X** value of **0.2** and a **Y** value of **0**. But when we look at the **Joystick**, as it appears on the screen in the following image, it shows up with an equal height and width. Because we are developing our game for a mobile device, we place images of virtual joysticks on the screen that the player can touch and manipulate in a manner similar to real joysticks. This is shown as follows:

The reason for this is a change we have made to the script that looks at the size, and if one of the values provided is 0, then that value is set to match the other value in proportion to the size of the

screen. The result is that by setting one of the two sizes, we will always end up with a square size for our `GUITexture`. This is an important change, since some items, such as joysticks, should always be square, regardless of the screen resolution. The script code changes required to accomplish this are shown as follows:

```
// if one size or the other is zero then scale it
// to match the other size
// This will, in effect, give us square sizes
// relative to the width or the height of the
// display
if (0 == guiSize.y)
{
guiSize.y = guiSize.x * Screen.width / Screen.height;
}
else
{
guiSize.x = guiSize.y * Screen.height / Screen.width;
}
```

Finally, the following screenshot shows the connection between the **Joystick** and the **JControlledCar**; in this case, the right **Joystick** will be used to steer the four-wheeled vehicle:

The JCar system provides a lot more control over how the four-wheeled vehicle can be configured. Some examples are as follows:

- Front, rear, or four-wheel drive
- Standard or automatic gear shifting
- Engine sound

But for our purposes, we have done all that we need for adding a four-wheeled vehicle to our scene.

Managing physics

Two things are required for an object to use the physics engine built into Unity3D:

- A rigid body
- A collider

The rigid body allows your objects to react to gravity, receive forces, and receive torque. The rigid body is the component that makes your game object interact and move in a realistic world with other rigid bodies.

The collider is used to send and receive collision information between objects. Rigid bodies are typically, but not always, used in conjunction with primitive (sphere, box, capsule) colliders.

Large objects

Managing big immovable objects, such as a grandstand or a tree, is very simple in Unity3D. Simply attach a primitive collider component to the object, and our player can run into them all day long and they will not move.

There is no need to attach a rigid body to a large immovable object.

There may be an exception to the use of primitive colliders, and that is the use of a mesh collider on a large model of background scenery. For convenience, we may attach a mesh collider to such an object or objects. Provided our player is using primitive colliders, we can assign mesh colliders to more complex, but immovable, objects such as roads with curbs.

Note

The one caveat to colliders in Unity3D is that mesh colliders that are not marked as convex do not collide with each other. If two important objects using mesh colliders need to interact, we need to either mark one of the meshes as convex, or we need to replace the mesh collider on one of the two objects with a series of primitive colliders.

Mid-sized objects

Typically, we will want our mid-sized objects to react with each other and with gravity. As a result, mid-sized objects need rigid bodies, colliders, and (often) joints.

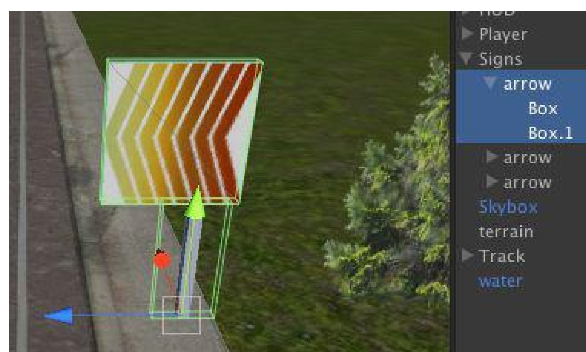
Note

We always keep in mind that using physics is done only as and when needed, due to the processing and memory requirements.

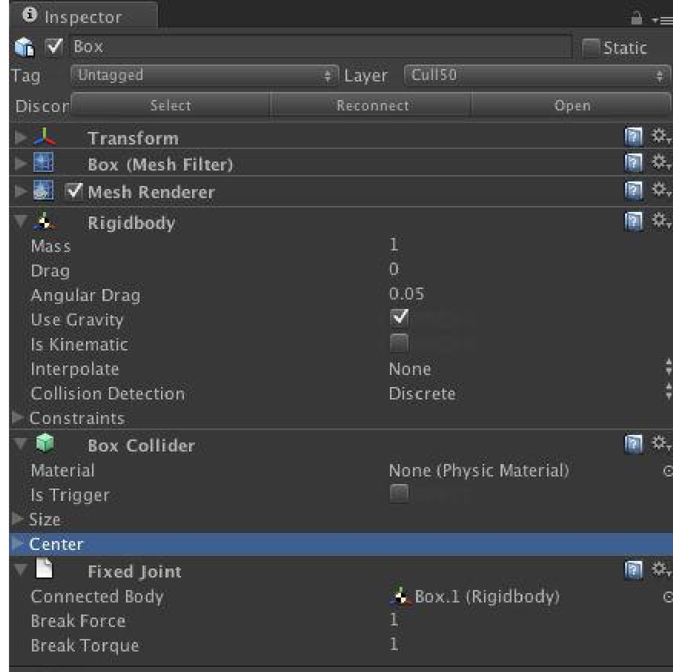
A good example is of a sign and a post that we want to break into two parts, when hit by our player's vehicle. We will construct the sign and post as follows:

- Use a separate mesh for the sign and the post
- Attach a rigid body to each mesh
- Attach a primitive collider to each mesh
- Connect the two meshes with a simple joint

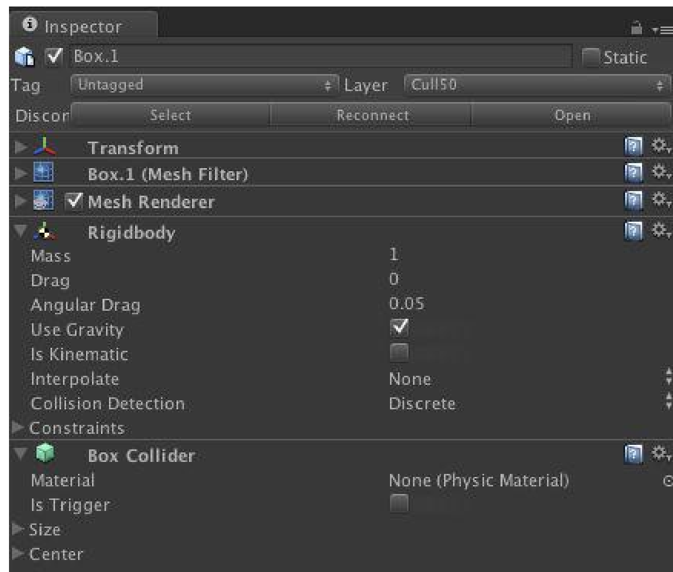
The resulting sign and post is shown in the following images. The first image shows the sign and its colliders in the Unity3D game hierarchy:



The second image shows the **Inspector** settings for the sign arrows. Take note of the **Rigidbody, Box Collider**, and **Fixed Joint** settings, as shown in the following screenshot:



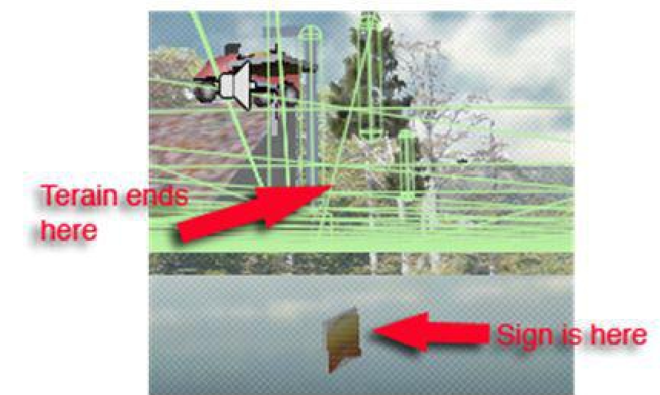
The final image shows the **Inspector** settings for the sign post, which corresponds to the **Connected Body** in the previous image:



And if we create a row of signs and drive our four-wheeled vehicle into the row, everything interacts and behaves just as we would like it to. The following image shows the result of running our vehicle into a row of signs:

One of the interesting things that can happen with medium-sized (and small) objects is that they can fall through other colliders. In the case of the sign, if we knock it over such that it ends up on the grass and then drive over it a couple of times, you will see it *disappear*.

If you examine this behavior more closely, as shown in the following image, you will see that the sign is actually falling through the terrain:

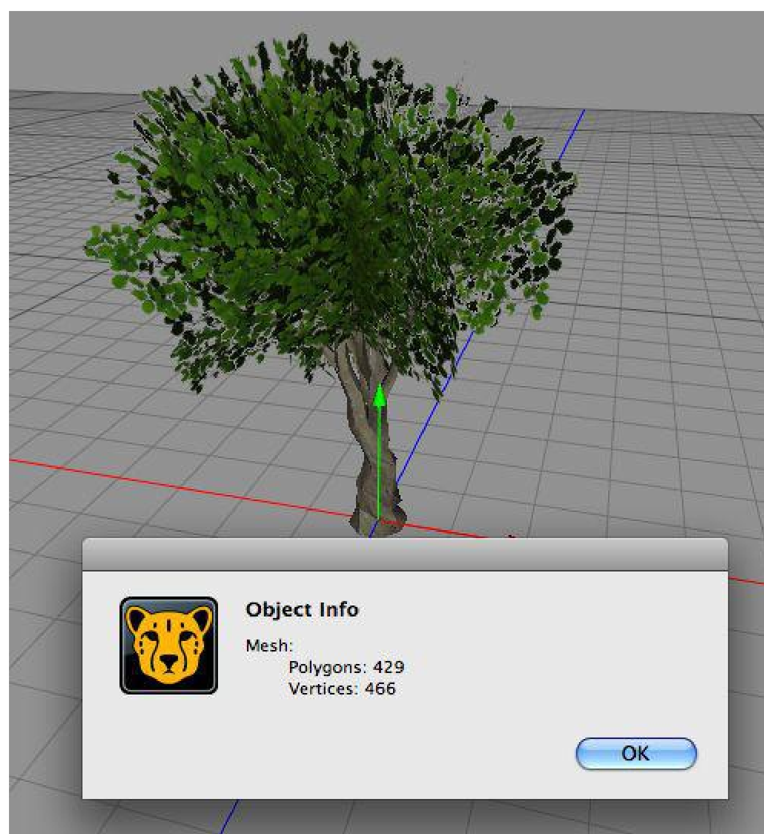


Realistic and fast foliage

Unity3D for iOS does not support the terrain engine. This means that in addition to creating our own terrain using a mesh, we need to create our own foliage. Since the reason that the terrain engine is not supported is performance, we need to be careful about the kind of foliage that we create for our game.

A typical tree

When we model a tree with a desktop computer in mind, we will typically create something like the tree shown in the following image:



While this tree may be acceptable on a desktop computer, it won't take too many of them, with almost 500 vertices, to dramatically reduce the frame rate that can be achieved on a mobile platform.

If we really want to use this kind of tree on an iOS device, it should be reserved for a really important aspect of our scene, and most likely we will want to include some kind of scripting to swap this model in and out of our scene, based on how close the player is to the tree.

Foliage for iOS devices

Because foliage adds a lot of atmosphere to a scene, we do want to make use of as much of it as is practical, without dramatically reducing the frame rate of our scene. In order to do this, we need to look at different ways in which foliage, other than modeling and placing individual trees can be used.

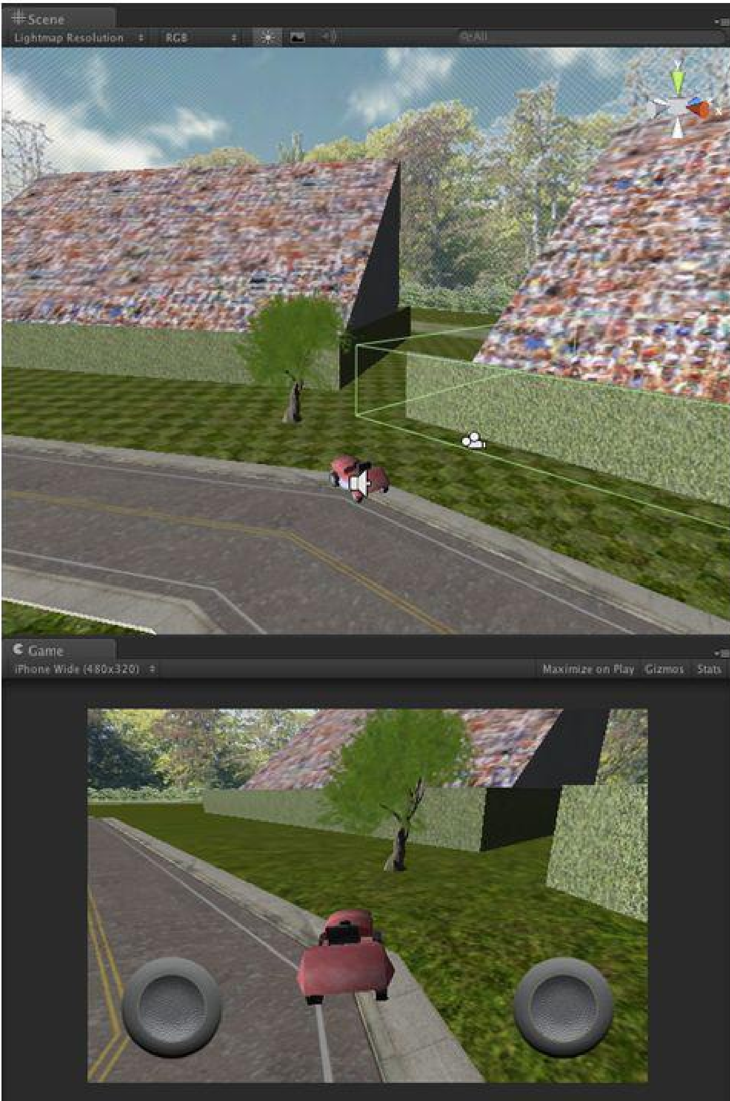
There are several other ways that we will consider, when it comes to foliage in our scene:

- Foliage that does not use transparency
- Foliage that uses a single plane with transparency
- Foliage that uses multiple planes with transparency

Foliage that does not use transparency

If we can create foliage that can use a simple, either lit or unlit diffuse shader, we can use a lot of it with very little impact on performance. Of course, since we are not using transparency, the foliage must fit the shape of our model exactly, so the number of vertices will become the limiting factor. In addition, foliage that does not use transparency will look the least realistic, so we will want to keep the player at a distance from the foliage.

An excellent way to use this kind of foliage is in a hedge with a simple collider that keeps the player a few meters back from the foliage, so that it is not obvious that a simple plane is being used. When done correctly, the foliage can be as simple as a textured plane, as shown in the following image:



Foliage that uses a single plane with transparency

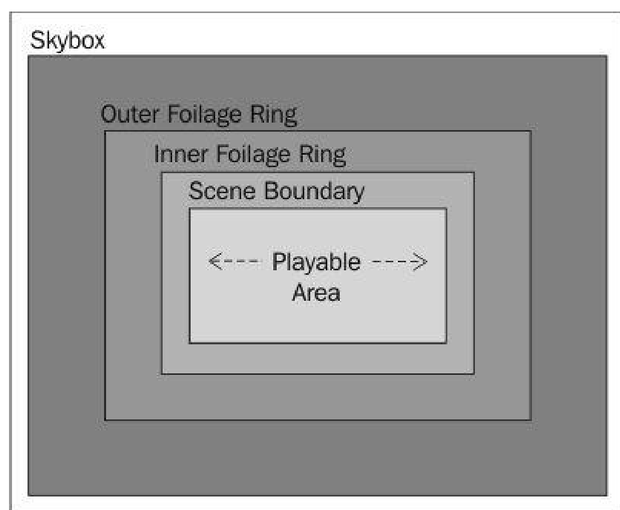
A single plane and a single texture with transparency can be used to create realistic and fast foliage, provided the player still cannot get too close to the foliage.

A good example is of a tree mapped onto a plane with a script that keeps the plane always facing the camera.

Another area where this kind of foliage can be used is the border of our scene. It may be tempting to place numerous individual foliage elements around the **Scene Boundary**, but that would result in a lot of objects that are always being drawn. Instead, we use the technique of a double border to give

the illusion of depth of single planes that are textured with a single foliage texture.

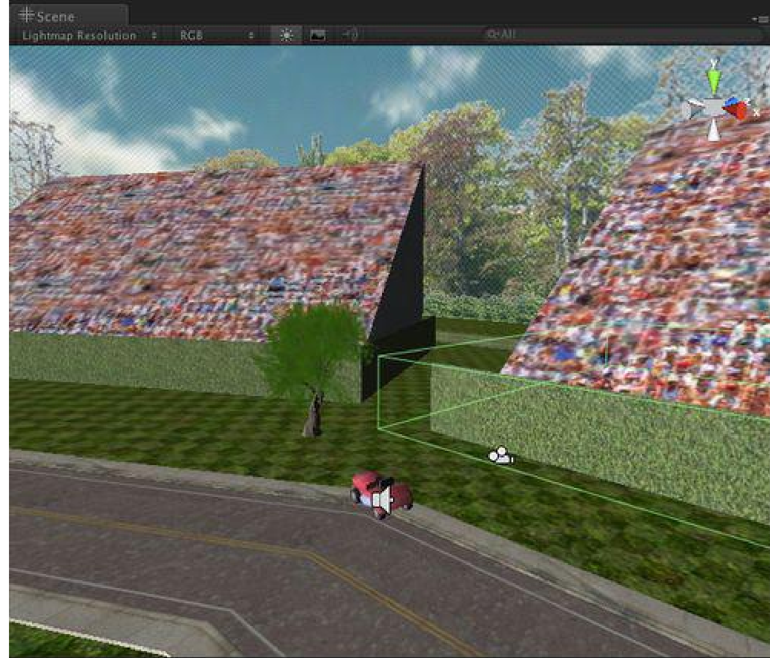
The following image shows how the double border of foliage will be placed:

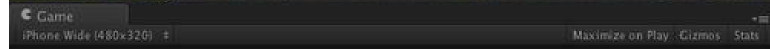


The important things to note are as follows:

- The **Skybox** is the outermost item
- The **Outer Foliage Ring** is between the **Skybox** and the **Playable Area**
- The **Inner Foliage Ring** is on the border of the **Playable Area**
- There is a **Scene Boundary** barrier on the **Playable Area** that prevents the player from moving all the way up to the **Inner Foliage Ring**

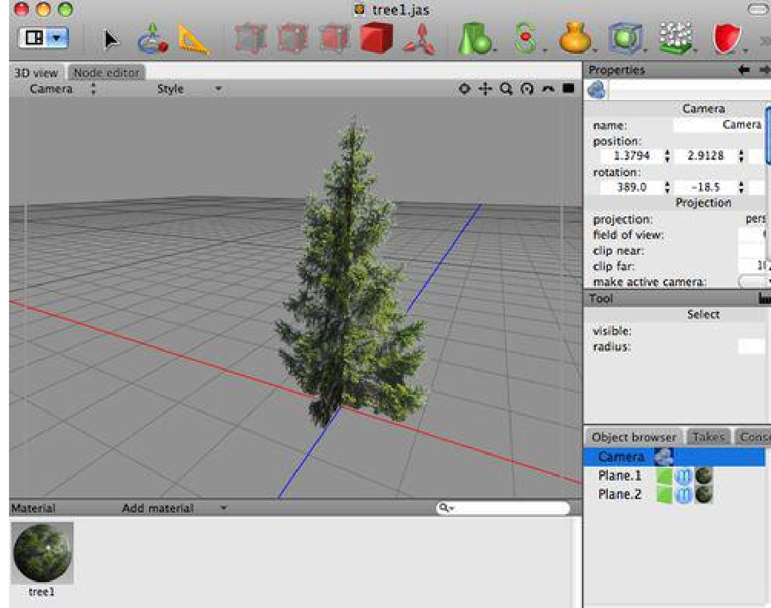
The following image shows how this will look in a Unity3D scene:



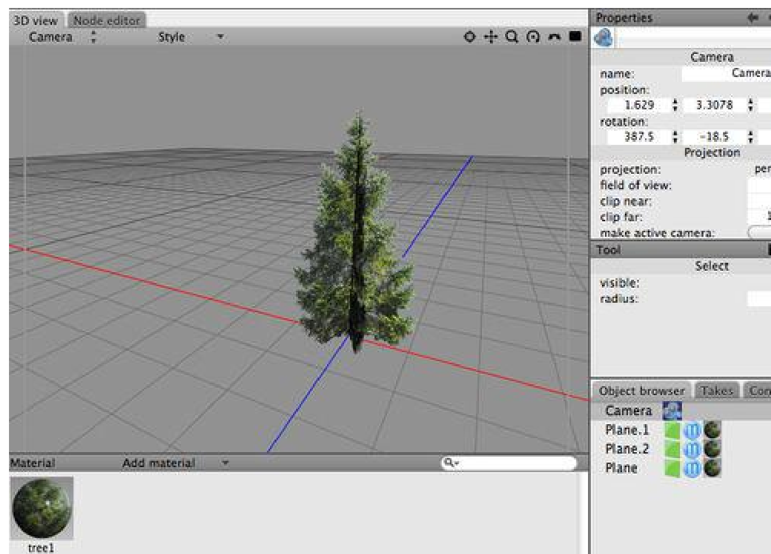


Foliage that uses multiple planes with transparency

This kind of foliage is the typical compromise between flat planes and fully-modeled trees. Essentially, we will use either two or three intersecting planes with the same texture on each plane. The following images show the typical models used for this kind of foliage, which uses considerably fewer polygons than high-resolution models. The first image shows a tree constructed using two planes:

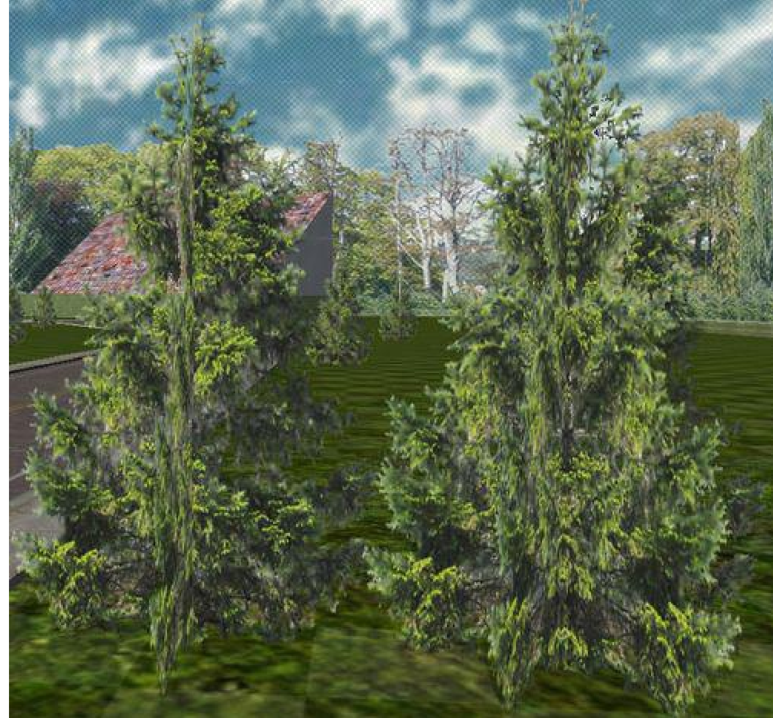


The second image shows a second tree constructed using three planes:



In addition to using transparency, we also need to use a shader that does not cull the back face. And it is also a good idea to use a shader that is not lit.

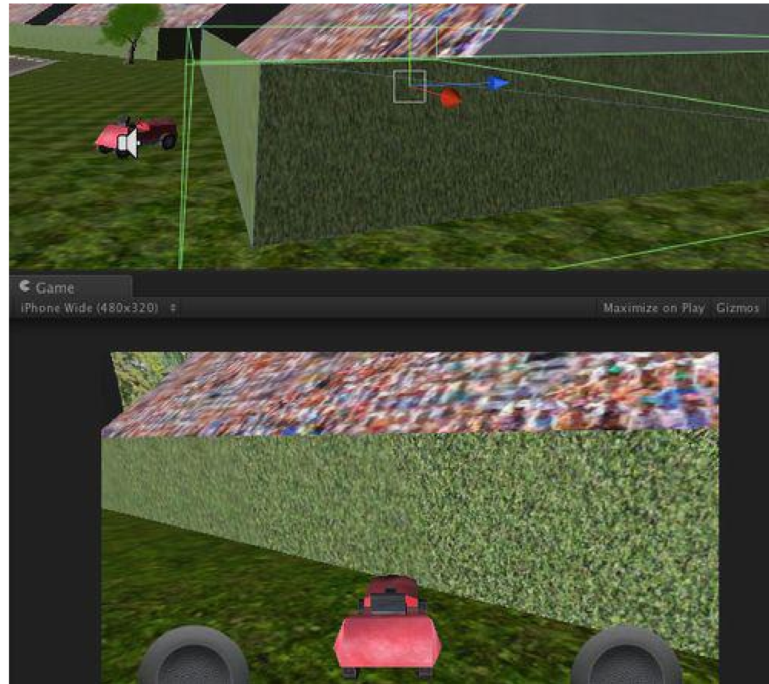
In a Unity3D game scene, these trees will look like the following image. Notice that the tree with three planes does appear fuller and more realistic than the tree with two planes:



Foliage colliders

The kind of colliders that we use depends on the kind of foliage we are using. For example, the foliage around the scene boundary does not require any collider at all. This is because the fence prevents the player from getting close to that foliage.

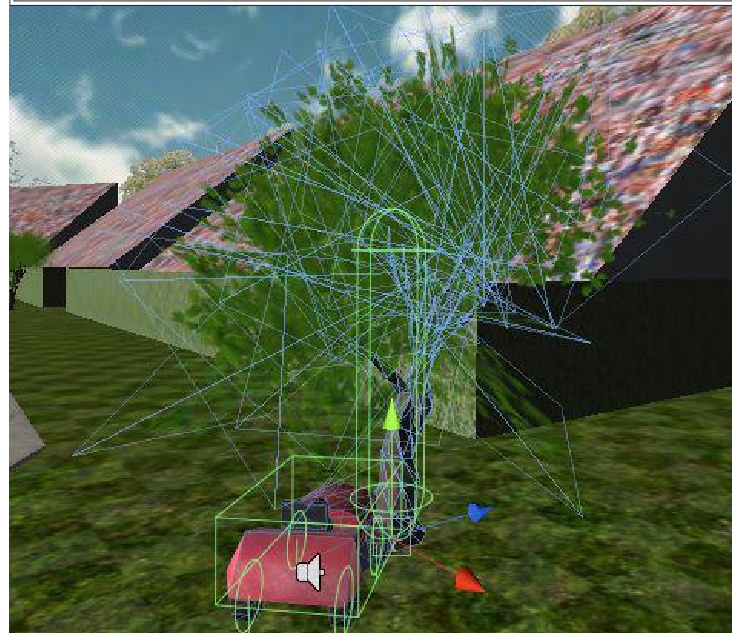
Similarly, foliage that does not use transparency may be behind a secondary image that does use transparency, and has a collider that prevents the player from getting close to it.



Foliage that uses two or three intersecting planes may have a simple, central, capsule collider, or in the case of a smaller shrub or grass, no collider at all. A tree with a simple capsule collider is shown as follows:

And finally, even fully-modeled, complex trees can use a single capsule collider (though sometimes, depending on the angle of the tree trunk, two colliders are needed), as shown in the following image:

> ## Note
>
> The most important thing to remember is that foliage, no matter how simple or how complex, should always use a simple primitive collider, and never use a mesh collider.



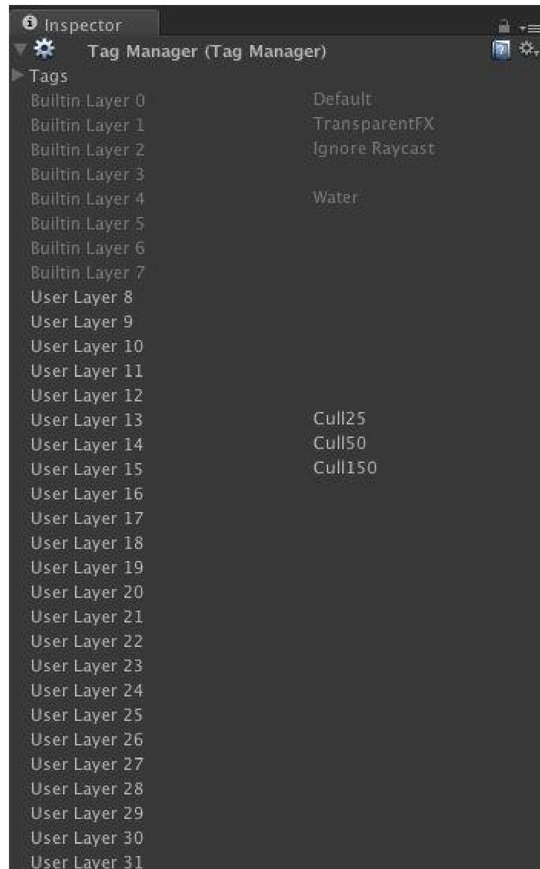
Culling made easy

The two most important forms of culling are distance culling and occlusion culling. There are some very simple steps that we can take to make using each of these culling methods very easy.

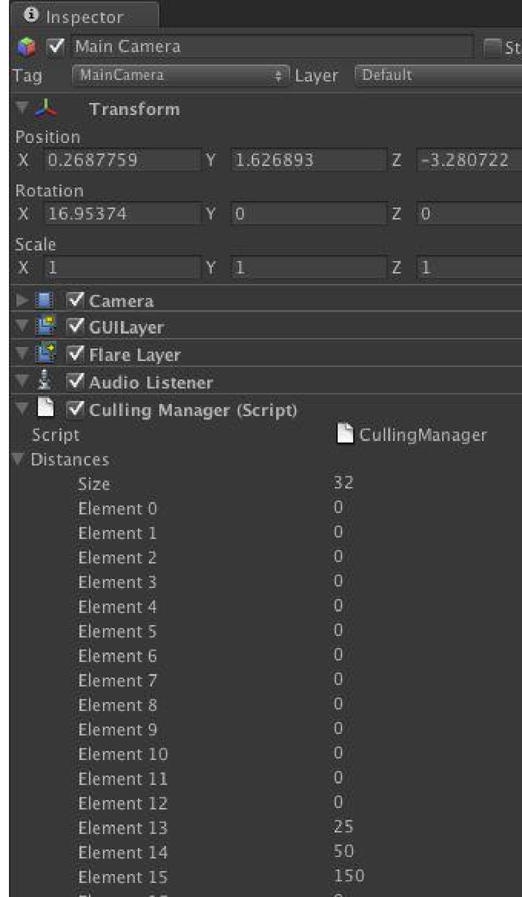
Distance culling

To make distance culling easy, we need to pre-plan the distances at which we want to cull the various objects in our game. We may decide to use 25, 50, and 150 meters as good distances to cull our various game objects. Once we have decided on the distances, we need to pick the game layers that we want to use for objects that will be culled based on those distances and set up the layer names in the **Tag Manager**. For example, we may decide to use layers 13, 14, and 15, and we may name them **Cull25, Cull50**, and **Cull150**.

The following screenshot shows how we would name those layers in the **Tag Manager:**

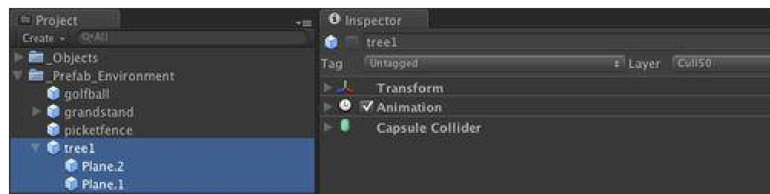


Once we have decided on the layer distances, we need to attach the **CullingManager (Script)** to the **Main Camera**. The **Culling Manager (Script)** will assign the specified distances to the `camera.layerCullDistances` array. After the script is attached to the **Main Camera**, we assign the distance to the array elements that correspond to layers 13, 14, and 15, as shown in the following screenshot:



Finally, having established the linkages between the layers and the culling distances, we can assign the default culling distance to our objects on their **Prefab**. This means that whenever we create a new instance of the object from the **Prefab**, it will automatically be culled at the correct distance.

The following screenshot shows an example of a tree **Prefab** with a default culling distance of 50 meters:



If, for some reason, we decide to change the distance of tree culling to 150 meters, we just need to change the default layer of the **Prefab**, and all of our trees will automatically be updated to reflect the new culling distance.

Note

Please note that the culled models will not be visible, when the camera is further from them than the selected distance.

Occlusion culling

There are two kinds of occlusion areas used by occlusion culling. They are as follows:

- Target
- View

Target areas are used for moving objects, and nothing special needs to be done to use them. View areas are used for stationary objects, and in order for them to work correctly, we need to mark stationary objects as **Static**.

The easiest way to remember to mark stationary objects as static is to make sure we always use Prefabs and to make sure that the Prefabs are marked as **Static**. By following these two steps, we are guaranteed that every stationary object we add to the scene will be correctly set up for occlusion culling. The following screenshot shows one of our culled Prefabs with the **Static** attribute set:

Note

There is a caveat, and it is that object for which camera culling will be applied, and should not be marked as **Static**. If a camera-culled object is marked as **Static**, you can get unexpected behavior that results in some meshes being visible and other meshes being culled.

Once the correct set of objects have been marked **Static**, it becomes a simple matter of creating occlusion areas and a suitable scene geometry to maximize the performance that can be obtained from occlusion culling.

The most important thing we need to remember is to make sure the camera can move to the occlusion areas and to make sure the **View Cell Size** is not too small. As with all things, it becomes a trade-off between the performance improvement and the memory and time needed for the generation of occlusion zones.

The following image shows our scene, which is using a **View Cell Size** of **10** and a single layer of cells because we intend to keep our camera grounded:

Scene
Lightmap Resolution
RGB
Occlusion Culling
Edit
View
Target
Camera Preview
Occl. Culling
Object
Bake
Visualization
View Cell Size
10
Near Clip Plane
0.3
Far Clip Plane
1000
Bake Quality
Production

Task: build your track

If you open the Unity3D project folder named `Chapter 7 Unity Project`, you will find the `Scene1.unity` file in the `Assets` folder.

Now, using the Prefabs and Objects in the **_Prefab_Roads** folder, create your own track in the scene. Be sure to use vertex snapping to align the roads perfectly. Creating a track is simply a matter of dragging in the Prefabs and aligning them. Be as creative as you like.

You can find a sample track in the game object named **ZZZ_Answers**. Simply enable the **Track** game object and its children to see the sample track.

Task: create a death zone

Again, if you open the Unity3D project folder named `Chapter 7 Unity Project`, you will find the `Scene1.unity` file in the `Assets` folder.

When you open the scene and press **Play**, you will discover that we are able to drive our four-wheeled vehicle through the water and up the mountain in our scene. This is probably not a good idea, so we want to create a death zone that causes the vehicle to re-spawn at the start, if it drives into the water.

The first thing we need to do is create the spawn point as a game object. We use what we have learned about gizmos to have the letter **S** appear to indicate it as a spawn point. The following image shows our spawn point, which we have placed two meters above the vehicle in our scene. The spawn point does not move with the vehicle, it is simply the point in the scene where the vehicle will appear when it is created:

The next thing we need to do is create a collider, which is used as a trigger. We use it so that we can detect when the vehicle is under the water and execute a script to spawn it at the spawn point. The following image shows our water trigger. Notice that it is smaller than the water; this allows the vehicle to drive part way into the water before the spawn code executes:

Finally, we need to tie the trigger area and the spawn point together using a script that detects when

the vehicle enters the trigger area. The script is shown as follows:

```
// This is the location where
// the vehicle will appear
// Drag in a spawn point game
// object
var dest : Transform;
// When the vehicle enters the
// death zone it is teleported
// to the dest location
function OnTriggerEnter(colInfo : Collider)
{
if (colInfo.attachedRigidbody)
{
colInfo.attachedRigidbody.transform.position = dest.position;
}
}
```

We also know from our discussion on physics (earlier in this chapter) that it is possible for objects to fall through our terrain. In order to address this issue, we need to create a similar death zone below our terrain to catch and destroy falling objects. The script that will destroy, rather than reposition, falling objects is shown as follows:

```
// When any object enters the
// death zone it is destroyed
// Attach this script to the death zone that the
// vehicle will enter, like water or lava
function OnTriggerEnter(colInfo : Collider)
{
Destroy(colInfo.gameObject);
}
```

To see an example answer, look in the **ZZZ_Answers** game object and enable the `DeathZone` game object and its children.

Challenge: eliminate Z-Fighting

You probably noticed a lot of Z-Fighting on the example track and at the corners of the water around the mountain. Disable your track and re-enable the example track and its children. Eliminate the Z-Fighting on both the tracks and the water using traditional geometry moving as well as using a shader.

Summary

In this chapter, we have covered the following:

- The scripting and physics of four-wheeled vehicles
- How to configure for physical interactions including large, medium, and small objects
- The different ways that realistic foliage can be created and how to make it fast
- How to easily incorporate camera culling and occlusion culling into a game?
- How to build a track and spawn for a four-wheeled vehicle, after it runs into a death zone

In Chapter 8, *Making it Real*, we will look at lighting a scene, creating particle animation, and how to make effective use of shaders on iOS devices.

Chapter 8. Making it real

When it comes to making a realistic scene on iOS devices, Unity3D is the perfect tool for the job. With its advanced object placement, lightmapping, shading, occlusion culling features, and more, it is easy to see why Unity3D is a great choice for making games on iOS.

In this chapter, we will learn:

- How to use the **Beast lightmapping** system on iOS
- How to create efficient particle systems on iOS
- The simple way to use shaders on iOS to accommodate the different capabilities of various iOS devices
- How to create realistic water that performs well on iOS devices

Lighting and lightmapping the track

Lighting completely changes the mood of our scene. It allows us to create an atmosphere with areas that are well-lit and areas that are in shadow. Lighting contributes a lot to the mood of our scene.

Because lighting is so important, we will spend days working out how we want our scene lit and making sure every location that the player can visit is lit correctly.

There is no doubt that lighting will take up a lot of our time during game development because we want to make sure it is done well.

Lightmapping is a technique that allows us to "burn" lighting into a scene so that lighting does not need to be calculated every time the scene is rendered. While it is not as realistic as dynamic lighting, for most games, it is enough to create the atmosphere without the performance overhead. There are several techniques that can be used to achieve lightmapping, but we are going to limit our discussion to the built-in tool (Beast) that comes with Unity3D.

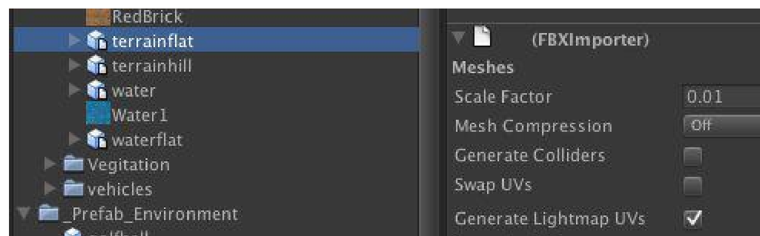
Preparing for lightmapping

Before we use the Unity3D Beast lightmapping, we need to make sure some details are taken care of. The following items need to be done before we begin lighting our scene:

- Make sure the meshes have **Lightmap UVs**
- Make sure that the object to be lightmapped is static
- Make sure meshes are imported correctly and triangulated (converted from non-triangular shapes such as polygons inside the 3D modeling application) prior to being imported into Unity3D, if required
- Adjust the ambient light setting to our desired value prior to lightmapping

Lightmap UVs

In order for Unity3D's lightmap system, Beast, to work, every mesh that will be affected by lighting must have a Lightmap UV. If you light a scene and an object appears really bright with no shadows, then it's because you forgot to include the Lightmap UV. Conveniently, Unity3D's **mesh importer** can generate Lightmap UVs for us and all we need to remember to do, as shown in the following screenshot, is to check the **Generate Lightmap UVs** checkbox.



Static objects only

The other thing required by Unity3D for lightmapping is that the objects being lightmapped are static. Because we may want to lightmap some objects, but not have them included in static batching (because of the limited number of triangles that can be batched), we may find that sometimes we need to temporarily mark an item as static, run the lightmapper, and then mark the object as not static.

A good reason for this is that the **Occlusion Culling** system works on static objects too. So while we may want to lightmap an object, we may not want it to be subject to occlusion culling.

A good example of this is our plane-based trees. These trees are going to be occluded using camera distance culling and so they do not need to be included in occlusion culling. In fact, if we do include them in occlusion culling, then we may get some strange behavior were one of the planes is rendered and the other is not based on the position of the tree and the location of the occlusion cells. This is because occlusion culling is treating each plane in the tree as a separate mesh for

culling calculations.

We could edit our trees and make them a single mesh, but this strategy may not work for all objects; so instead, we can do the following:

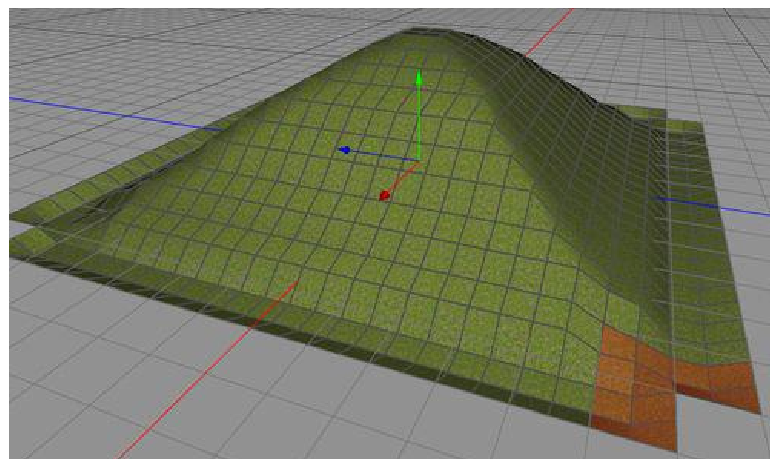
- Make sure our trees are defined using **Prefabs**
- Mark the tree Prefab as static
- Lightmap the scene
- Mark the tree Prefab as not static
- Generate occlusion culling

Triangulate objects before import

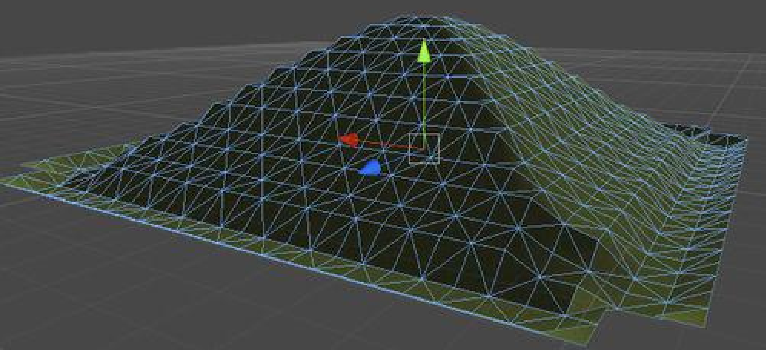
When we are working on objects in our 3D modeling software, we may think of creating rectangles rather than triangles. While this is convenient when modeling, we need to remember that everything is going to be rendered using triangles.

It's possible that the Unity3D importer will create a triangulated geometry that does not work correctly with the Lightmap UVs that were generated by the importer too. This may be a bug, but it still can result in meshes that don't look quite the way we want them to and triangles that are not correctly lit.

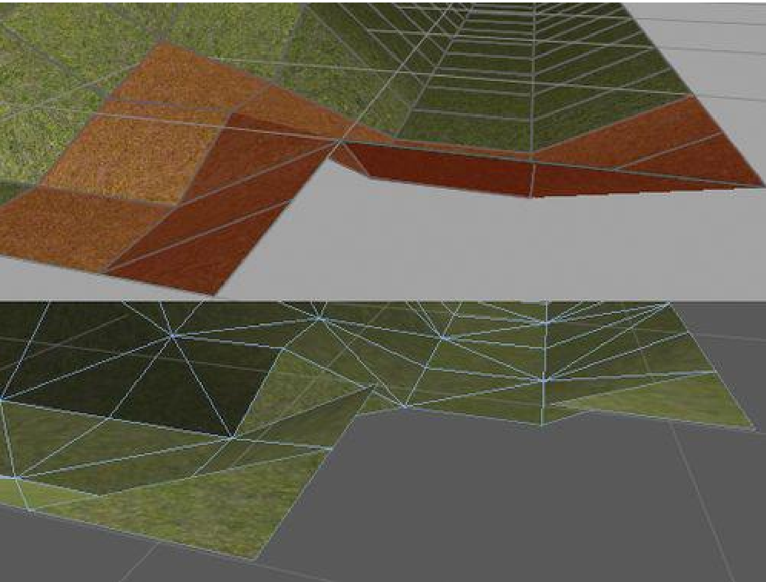
The following screenshot shows a mountain created in our 3D modeling tool, **Cheetah 3D**. You may recognize it as the `terrainhill.jas` file from the `Chapter 7 Unity Project`.



Look closely at the rectangles that make up the corner; they are highlighted at one corner in the image. Now look at the same corner in the following screenshot, which is the mesh, as imported into Unity3D using the default importer settings.



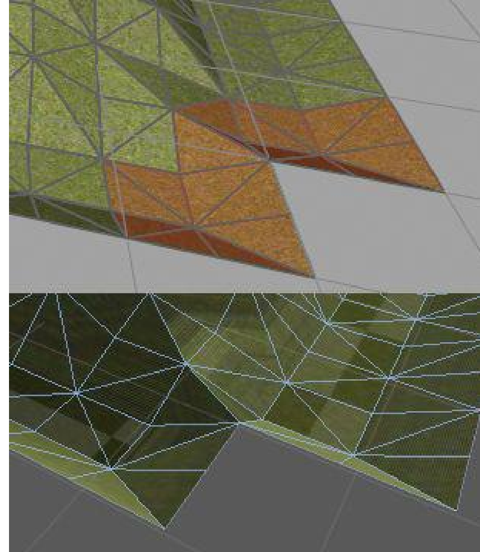
The following screenshot is a closer look at just the corner of the two meshes:



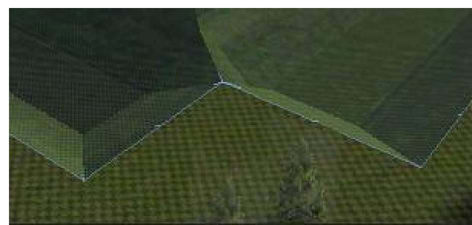
What you should notice is that, not only are the two meshes not the same, but that the flat corners of the imported mesh are the areas that resulted in Z-fighting with the water in the Chapter 7 project. In addition, when we lightmap this mesh using Unity3D, we will discover that those problem corners will not lightmap correctly.

There are a number of ways we could address the problem, for example, we could hand-edit the generated lightmap image to correct the problem, but the simplest way, by far, is to triangulate the mesh in the 3D modeling application prior to importing it into Unity3D.

The following images, from the `Chapter 8 Unity Project`, show the results both in Cheetah 3D and Unity3D. The resulting imported mesh not only looks better, but also lightmaps correctly. The first image is from Cheetah 3D.



The second image shows the model after it has been imported into Unity3D.



Adjust ambient light

The final thing we want to consider is whether or not we want the ambient light to influence our lightmaps. This is really a matter of personal preference, but it's something to consider. If we don't want the ambient light to influence the lightmap, then we need to do the following:

- From the **Edit** menu, choose **Render Settings**
- Click on the Ambient Light color
- Record or remember the current color
- Set the color to black
- Generate the lightmap
- Reset the Ambient Light color (to what it was prior to our change in step 2)

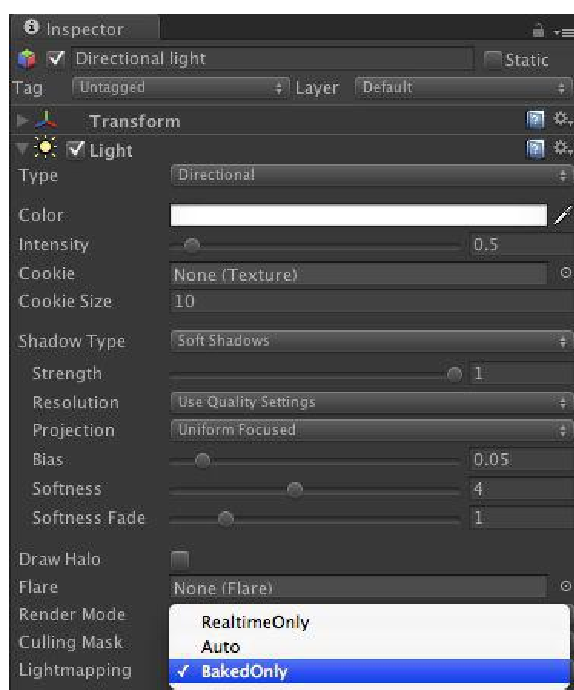
Lighting the track

Lighting is a very creative process and everyone is going to have a different idea on how to light the scene. In addition to deciding how the scene will be lit, we need to decide if lights will be Real Time, Baked Only, or auto (a combination of the Real Time and Baked).

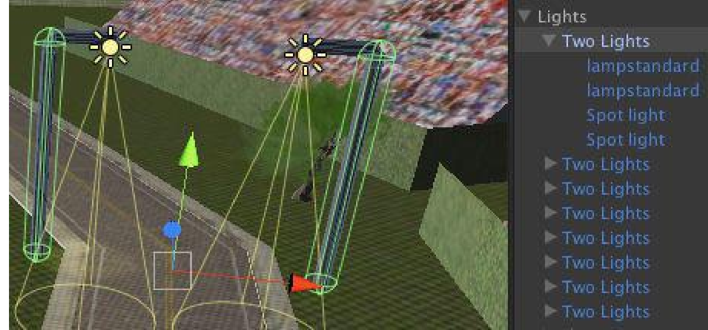
We will go through many iterations of lighting, baking, re-lighting, and re-baking until we are satisfied with the look and mood of our scenes.

In order to light our track, we first need to decide on the time of day. This will influence things such as the color of the sky and the length of the shadows cast by our directional light. Let's assume that, for our scene, we have decided it will be evening with a setting sun.

We tint our skybox slightly orange and then we add the directional light, as shown in the following screenshot, to cast long shadows on our track. We have also decided that the directional light **Render Mode** will be **Baked Only**, as shown in the following screenshot, which means it will only have an influence on the lightmaps and we will be relying on ambient lighting to light our scene:



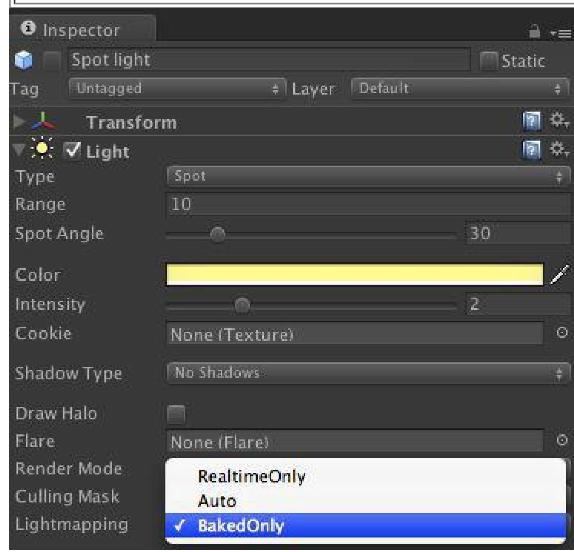
In addition to the skylight, we have decided that we want some tall light standards to provide spots of light at certain points around the track. The following screenshot shows how we will lay out our light standards in the scene at the corners of the track:



We have also decided, as shown in the following screenshot, that the track lights **Render Mode** will be **Baked Only**.

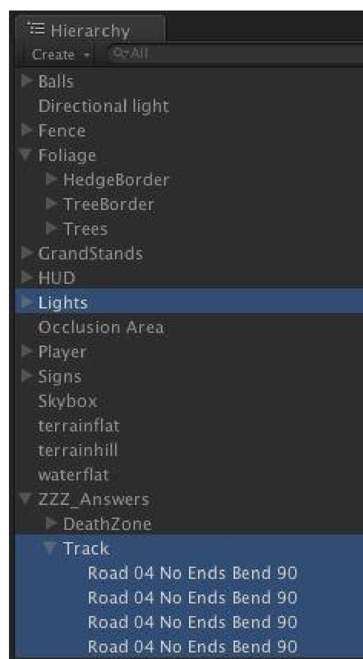
Note

If we want the spotlight to illuminate our four-wheeled vehicle during gameplay, then we would choose **Auto** for the **Render Mode**.



Lightmapping the track

Now that we have finished lighting the track, we can bake the track lighting to see how it is going to look in our final scene. Because lightmapping is a time-consuming process, and at this point, we are only interested in the track lighting, we have only selected the track and the lights in our project hierarchy. The following screenshot shows our selections:



We have limited our selection because we are only interested, at this point, in seeing the lightmapping that will be applied to the track. By limiting what we choose to bake, we can dramatically speed up the bake time. In the following image, ambient lighting has been turned off to show the results of the track bake (notice that only the track and light standards have been lit and that, for example, light and shadows do not (yet) extend to the terrain). The trees are visible because they are using un-lit shaders as a performance optimization.

To further illustrate this, we can lightmap the scene again, but this time, include the portion of the terrain under the track. As you can see from the following image, doing that results in the shadows and light now extending to the included terrain.

For the final scene, we will include every object that we want to be lightmapped, and this will result in a fully lit scene.

Particles that pop

Particles make everything cooler. A game can be entertaining without particles, but adding particle effects will always take it to the next level. Our player can collect a gem or our player can collect a gem that explodes in a shower of delightful particles.

Renderer

Renderers are responsible for drawing, or rendering, the particles. There are three kinds of renderers:

1. Trail
2. Line
3. Particle

While each of the renderers draw particles, it's important to note that the **particle renderer** requires a **particle system** (the renderer, plus an emitter, and an animator) to function correctly, while the **trail renderer** and **line renderers** are not components of a Particle System and so do not need a separate emitter and animator.

All of the renderers draw particles, and it may help us to think of the Particle Renderer as a *Particle "System" Renderer*.

Trail

A trail renderer is used to create a line of particles that follow our game object. In addition to rendering the particles, it also emits and animates the particles. A trail renderer is an expensive object and should be used sparingly.

Line

A line renderer is used to draw a line. We need to provide the line renderer with a list of points and it will draw the particles to create a line between those points. It essentially renders an array of planes between the points and is most useful when we know ahead of time where we want the particles to be drawn.

For line renders, it is recommended that **Textures** be configured to use a **Wrap Mode** of **Clamp**. We also need to be aware that if we use large particles with transparency, it can dramatically impact performance.

Particle

A particle renderer is a component of a particle system. This is the most common kind of renderer that we will use in our game.

Particle system

In addition to the particle renderer, there are three additional components in a particle system:

1. Particle emitter
2. Particle animator
3. World particle collider

Particle emitter

The particle emitter generates or emits the particles and assigns each particle values such as size, energy, velocity, and so on, based on the settings that we specify when we create the emitter. While the settings for a particle emitter are very clear, there are three things that we need to pay attention to on iOS devices.

The first is that an **Ellipsoid Particle Emitter** does not inherit the **Transform Scale** while a **Mesh Particle Emitter** does inherit the Transform Scale. Thus, if we want to increase the size of the ellipsoid, then we need to do so by setting the maximum size and minimum size values of the Ellipsoid Particle Emitter.

Secondly, we need to be aware that a Mesh Particle Emitter is more computationally expensive than an Ellipsoid Particle Emitter, so we should exercise caution and be certain that a mesh emitter is really required before we decide to use it in our game in place of an ellipsoid emitter. For most of the requirements of our games, except in special circumstances, ellipsoid emitters are good enough. If a mesh emitter is required, then we need to be careful that we do not create too many mesh emitters, or it will dramatically decrease our game's frame rate.

Thirdly, we need to understand that **max emissions** are the maximum number of particles that will be created each second, not the absolute maximum number of particles. In the worst case, the absolute maximum number of particles is max emissions times **max energy**, though typically, it will be the average of max and min emissions times the average of max and min energy.

Particle animator

The particle animator controls the color, size, movement, and automatic destruction of particles. Other than making sure transparent particles do not get too large, there are no special iOS considerations for particle animators.

World Particle Collider

The World Particle Collider causes particles to collide with other colliders, and optionally send the `OnCollision` message. The World Particle Collider should be used sparingly, especially in combination with sending collision messages, as its use can have a dramatic impact on performance.

iOS recommendations

We will try to limit any particle system to no more than 50 particles (this is a calculation such that the average number of emissions into/multiplied by the average lifetime is approximately 50) in existence at any one time. This will help reduce the required memory bandwidth needed for the system.

In addition to limiting the number of particles in an individual system, we will try to limit the number of systems in play (visible or not) at any one time to 30. Having 30 visible at the same time is possible only if they are low emission effects. If we try to have 30 systems with 50 particles, all visible at the same time, then our frame rate will be dramatically impacted.

We also want to limit the use of additive shaders; in fact, we will not use them at all unless it is absolutely required. Instead, we can simply use a standard transparent shader, and as particles will always face the player, it is unnecessary to turn off culling.

The particles systems themselves need to be destroyed (or turned off, if we want to reuse them later) after a reasonable amount of time and, if possible, once they are no longer visible to the player.

Finally, we need to remember that a large number of particles that use transparent shaders are an absolute no-no on iOS devices.

Creating particles in your particle system

The creation of particles is done by the emitter component. The emitter also initializes the individual particle values. Because each particle is an individual object, with certain properties, the array of particles within a system can be retrieved from the emitter instance. Once the array has been retrieved, individual particles can be accessed and manipulated.

There are two ways particles are emitted, depending on the emitter type. In mesh particle emitters, particles are emitted from the mesh vertices. In Ellipsoid Particle Emitters, particles are emitted at random points within the ellipsoid above the minimum emission radius. By default, particles do not have a velocity and, if a velocity is not applied to a particle, it will remain stationary.

Individual particle velocity will be assigned based on the velocity parameters specified in the parent particle emitter. Initial velocity is a random floating-point number clamped between the specified local velocity and the local velocity plus the **Rnd Velocity**. Because particles do not contain a `RigidBody` component, an individual particle's velocity is not affected by gravity. As a result, particles will move at a constant speed until a force is applied to change their speed or they are destroyed. The particle animator is responsible for applying the force that decelerates the particles.

Tangent velocity allows particles to be emitted in a spherical manner about the origin of the emitter. For example, setting the tangent velocity of (0,1,0) will cause particles to spiral about the Y-axis in the X-Z plane. To observe this effect more clearly, we can add a positive Y velocity.

Another important consideration for particles is how they will behave in world and/or local space. Enabling the **Simulate In Worldspace** option will cause particles to behave independently of the particle systems' position and rotation. This option is useful for creating storms, exhaust pipe smoke, or rocket trails. Conversely, disabling this option will cause particles to inherit the translation of their parent, which is useful for creating energy fields, clouds, or spells.

The final thing to be considered is if this particle system is **One Shot**. Enabling this behavior will cause a particle system to emit one round of the absolute number of particles, wait till they are destroyed, and then emit another round. Usually, this is used in conjunction with the **Auto-destruct** option in the particle animator, which will automatically destroy the system after one round and is useful for creating explosion effects.

Animating particles

Particles would be rather boring if they could only move about, but never change state. The particle animator takes care of changing the **Red Green Blue Alpha** (**RGBA**) of particles and changing their size and/or velocity. The particle animator also contains a particularly important **autodestruct** functionality that helps automatically manage the destruction of simple effects, so that they don't consume bandwidth.

Emission velocity versus added force

Remember how we set the initial velocity of our particles in the emitter? The particle animator will allow us to change this value, as the lifetime of a particle progresses by applying a force vector. This is useful for creating a *fountain*. However, remember that when particles fall back down, they will not collide with your world unless a particle collider is attached to the system. Because this collision is unnecessary most of the time and is computationally expensive, you should reduce the lifetime of particles so they don't exist for long after passing through the terrain instead of destroying them by detecting impact.

Color

The modification of color in particles will occur as long as animate color option is checked. If you do not intend to animate the color of your particles, then ensure animate color is off to increase your performance. Because animate color is used to fade particles in and out, you may opt to always use animate color. If this is the case, then try to create colorful and vibrant effects, as the bandwidth for these changes would be expended anyway.

Growth

Particles can grow, which is great for creating effects such as the diffusion of gas, but always remember that a particle is little more than a fancy transparent plane bear in the mind of the *large alpha* bug in the iOS rendering pipeline.

Autodestruct

Autodestruct will destroy one-shot particle systems after they have emitted one round of particles and all those particles have died; this prevents the requirement of additional code. Destroying unnecessary objects is one of the most important tasks in Unity3D iOS projects due to memory constraints. Additionally, objects should be destroyed systemically instead of in bulk to avoid tying up the processor. If the particle system is not one-shot, then the autodestruct switch is ignored.

Particle example: Weather

A common mistake made when trying to implement a weather system is trying to create a particle system that encompasses the entire play area. Not only is this impractical, but it also has a negative impact on game performance. The particle system should cover only a small area around the player. Because the camera typically will rotate with the player, the particle system should not be parented on the player game object.

Parenting the particle effect to the player causes a strange skewing effect due to the inherited rotation; instead, use a simple follow script such as the following one:

Note

Notice this script uses an infinite loop and a `yield` statement. This means it is intended to run as a co-routine, so the movement of the storm can be tuned to the movement of the player. For slow moving players, the `updateTime` can be set to larger values.

```
varupdateTime : float = 0.05;
varplayerTransform : Transform;
function Start ()
{
yield Storm();
}
function Storm ()
{
while(true)
{
transform.position.x = playerTransform.position.x;
transform.position.z = playerTransform.position.z;
yieldWaitForSeconds(updateTime);
}
}
```

The particle system must also be instructed to simulate in world space, so that moving the player does not move existing particles.

We can use a higher density particle system to make the weather seem intense and a second lower density system using a larger ellipsoid to ensure the weather appears to be occurring further away as well. It's a good idea to place the system slightly above the origin of the camera.

If we are developing a networked game, then we will instantiate a single weather system per player on the client. We will not use network instantiation for weather systems.

Shaders in the game world

On the original version of Unity3D for the iPhone, the only rendering path available was the **Vertex-Lit**. Now, Unity3D for iOS devices supports two rendering paths:

1. Vertex-Lit
2. Forward rendering

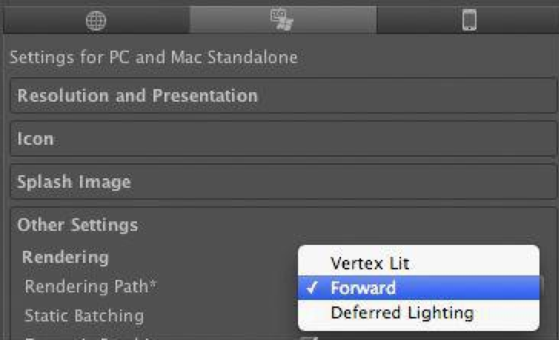
The essential difference is that the **forward rendering path** supports pixel lighting and the Vertex-Lit path calculates lighting in a single pass and only at object vertices.

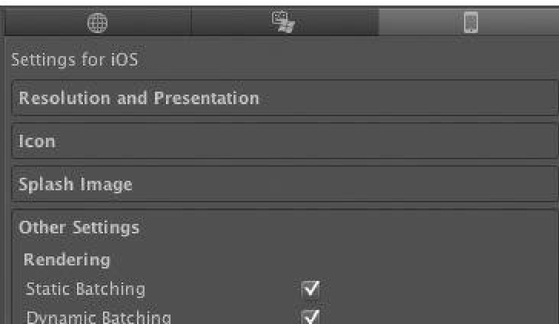
While it would be faster if we could use only the Vertex-Lit path, the reality is that for a typical game, the forward rendering path is essential because it allow us to create effects that are simply not possible using only Vertex-Lit rendering. However, care must be taken when using the capabilities of forward rendering on mobile platforms as there are performance implications to forward rendering, and if we are not careful about what we do, we can dramatically impact the performance of our game.

The moral is that just because something is supported, it doesn't mean we can fully exploit it on the hardware that we will be deploying our game on. We can use it to some degree, but we need to be careful to do the following:

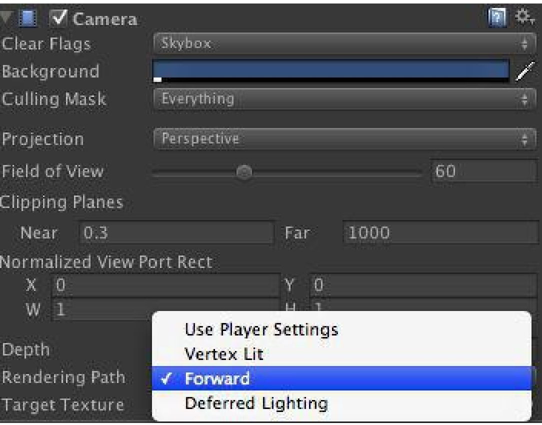
- Test our game in Unity3D using the graphics emulation for the devices we want to deploy our game on
- Test our game on all real devices we wish to support
- Be mindful of the frame rate that we can achieve
- Make game compromises to keep our frame rate up to an acceptable level

In the PC and Mac Standalone version of Unity3D, the player settings allow you to choose the rendering path for your project. This option is notably missing in the iOS project settings, as shown in the following screenshot:





There is an option to select Vertex-Lit shading on a per-Camera basis, so if, for some reason, our game could get away with using only Vertex-Lit scenes, then we could specify Vertex-Lit as the rendering path on our Camera. This option is shown in the following screenshot:



Anatomy of shaders

Shaders are a huge topic. We could write an entire book just on the things that you can do with shaders. This is a very high-level discussion of shaders, which highlights a few things to consider when it comes to iOS support and the performance of shaders.

There are several important things to consider when designing our shaders. We need to think about the following:

- Geometry and its properties
- Other passed values
- Shading approximation
- Depth
- Platform capability and subshaders

Geometry and its properties

Before we look at shaders, we need to understand geometry.

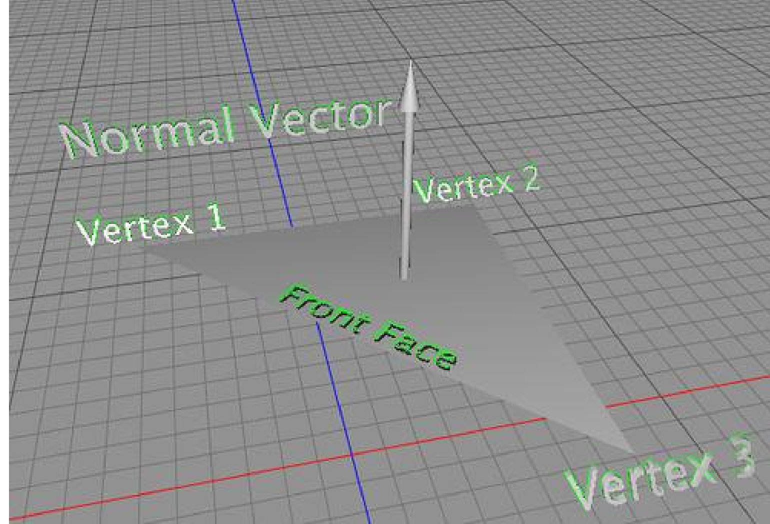
Geometry is essentially a mesh of vertices that are combined to create polygons. For example, the vertices could be combined to form triangles, quadrangles (or Quads), or any other polygon shapes using more than four vertices (referred to as an **nGon)**.

In Unity3D, no matter what kind of polygons you use in your 3D modeling software, all geometry will be triangulated when it is imported. Thus, in Unity3D, geometry is a mesh of vertices combined to create triangles.

Each triangle will have a front face and a back face. As there are two faces, a convention is used, based on the clockwise order in which the vertices of the triangle are stored, to define which face is the front face and which face is the back face.

> ## Note
>
> The front face is called the normal because the math used to calculate the front face is done by calculating a vector perpendicular to the triangle, such that the direction of that vector determines the orientation of the triangle's face.

The following image illustrates these concepts:



Typically, the normal face, or the front face, will be the face that is rendered by a shader and the other face, the back face, will not be rendered. Faces that are not rendered are referred to as having been culled.

Note

While the back face is typically culled, there are special case shaders, such as glass shaders, where the back face will be rendered and not culled.

So the normal face is the important face of the geometry.

This is why, if you walk behind a plane, the object will seem to disappear. It's also why you can sometimes see through things in 3D games when the player moves into a position such that the camera is facing the back face of a triangle.

Note

The normal face is used by the occlusion culling and **mesh collision** system. If you are using a shader that does not cull the back face, then you may get unexpected results from the occlusion culling and mesh collision systems because the object may be culled and other objects will not collide with the back face. This is because those systems only take geometry into account.

We have seen that sometimes Unity3D does not quite get it right when it triangulates our geometry, so we need to pay attention to the meshes that are imported and make sure the triangulation done by Unity3D is not going to cause us a problem.

Sometimes, issues can be resolved by adjusting the settings on the mesh importer for a mesh. But often, we need to make the adjustments in our 3D modeling tools, such as triangulating the mesh, so that Unity3D has a less complex task of importing our geometry.

Finally, we need to know that Unity3D will automatically calculate tangents, but as they are mostly used in bump-mapped shaders, we can turn them off for objects that will not be using bump-mapped shaders and save the memory that would be occupied by the calculated tangent array.

Other passed values

Textures can provide a pattern or image, depth information, lighting information, or some combination of this information. The material information is often provided in the form of textures (2D images). Unity3D shaders can use one or more UV-mapped textures, along with other information including colors and floating point values, to project the information that you want onto the mesh. We can create all of the UV textures or, primarily for lightmap, Unity3D can create the textures for us.

Unity3D also allows us to set values for shaders in the inspector that become arguments for our shader. In addition to our geometry, other values such as textures and lighting are passed to and handled by the shader.

Declaring and referencing properties

When we create a shader, we can declare a number of properties that are exposed in Unity3D in the materials inspector.

Properties are declared after the properties symbol. The syntax for declaring properties is as follows:

```
_VariableName("Inspector Display Name", type) = initial value
```

Types include, but are not limited to:

- Color: A constant RGBA color
- 2D: A texture or GLtexGen
- Range (min,max): A slider
- Cube: A cubemap

To reference a variable in **ShaderLab** code, use `[_myVariableName]`

An example of how to declare properties is as follows:

```
Properties {
_Color ("Main Color", Color) = (1,1,1,0)
_SpecColor ("Spec Color", Color) = (1,1,1,1)
_Emission ("Emmisive Color", Color) = (0,0,0,0)
_Shininess ("Shininess", Range (0.01, 1)) = 0.7
_MainTex ("Base (RGB)", 2D) = "white" {}
}
```

Advanced lighting

The material tag is used to enable advanced lighting by doing one or more of the following:

- setting a diffuse color multiplier
- setting an ambient color multiplier
- enabling specular (including shininess) properties
- enabling emissive properties

Changing the ambient color or diffuse color multipliers is done in special cases where the object being lit has some special properties, for example, reflecting light at double the normal intensity.

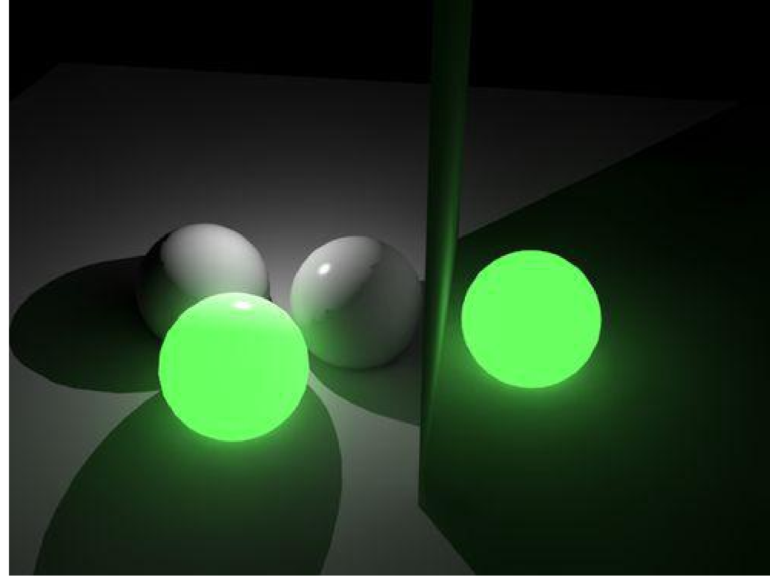
Changing the shininess (size of the highlight) and specular (color and intensity of the highlight) properties is useful for creating simple reflections of light that give a glossy or metallic appearance to an object.

Changing the emissive property is useful for creating self-illuminated objects, though the light emitted from the object onto other objects is not calculated at runtime, but can give fantastic results in lightmap calculations.

In order to understand how the final lighting will appear, based on the values set in the inspector and passed to the shader (they could be hardcoded in the shader, but it's free to expose them in the inspector and provide more flexibility), we can use the following calculation for the final lighting.

```
Ambient * RenderSettings+ (Light Color * Diffuse + Light Color * Specular) + Emission
SubShader{
Pass {
Material {
Diffuse [_Color]
Ambient [_Color]
Shininess [_Shininess]
Specular [_SpecColor]
Emission [_Emission]
}
```

The following image shows the results of both the specular and emission properties when applied to meshes. Notice the self-illumination of objects, the splashes of lighting from objects, and the shininess of some objects compared to others.

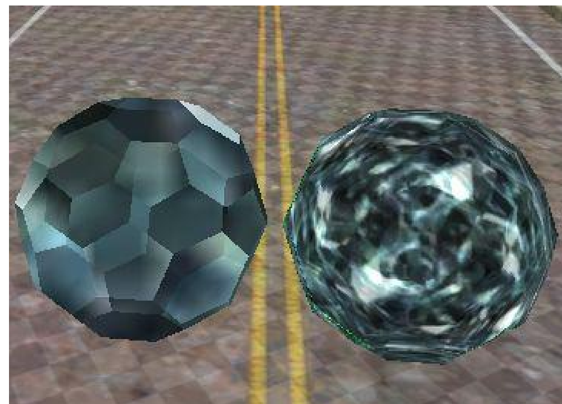


Shading approximation

Shading approximation primarily has to do with how hard you want your object edges to appear when the objects are rendered.

A good way for us to look at the hardness of our edges is to look at gems in Unity3D. Typically, we want the edges on gems to be hard, but if we use a soft shading approximation, then we will get something with much smoother edges.

The following image shows two gems that are using the same shader, the difference between the two is that the one on the left is using a flat-shading approximation, which is actually a literal interpretation of the geometry without any interpolation, and the specular highlight is still being interpolated in a vertex-lit manner. The one on the right is using a smooth (**Phong**) shading approximation. It can be seen quite clearly that shading approximation can have a significant impact on the appearance of an object in our game.



Depth

Every 3D object is rendered in 3D space. In 3D space, an object's depth or distance from the camera is referenced by the Z coordinate. Similarly, when developing shaders, the Z plane is used to determine which objects are rendered in front of each other. The main shader components that deal with depth are as follows:

- ZTest
- ZWrite
- Offset

We can use these components to modify how shaders render objects, and as we have seen with Offset, how objects on the same Z plane can be rendered in front of other objects on the exact same Z plane.

ZTest is a Boolean that instructs a shader whether or not it should consider the position of the geometry, relative to the camera. If ZTest is false, then the object will simply render when it has been instructed to in the queue, and may be subject to overwriting by objects in the same queue.

`ZWrite` is a flag that instructs a shader whether or not it should write pixels to the depth buffer. For opaque effects, turn `ZWrite` off.

Queue order instructs a shader when it should render in the frame; a shader will write to the screen after it has completed calculating which pixels should be overwritten, and will write on top of shaders that already rendered earlier in the queue. In order to instruct a shader to reposition itself in the queue, use the following code:

```
Tags{"Queue" = "QueueLevel"}
Where QueueLevel is one of the following values:
Background
Geometry
Transparency
Overlay
```

The `Overlay` queue level is written last.

Each queue level is a string substitution for integer values 1000, apart from each other (for example, Background = 1000, Geometry = 2000). Any of these values may be specified followed by an increment or decrement using an operator (for example, `"Queue="Background-25")`.

Platform capability and subshaders

When we develop a shader, we may need to take into account the capabilities of different iOS hardware platforms. While it's always a good idea to target our games at the latest hardware offerings, we may, for example, if we are developing a shader for someone else, need to consider the capabilities of older hardware.

Unity3D provides subshaders specifically to deal with different types of hardware. Unity3D will search the subshaders to find the first one that will run on the graphics hardware (starting from the top of the file, subshaders should appear in the order of most complex to simplest) and use it to render the mesh.

If we look at the source code of some of the built-in Unity3D shaders, then we will see that there are many different subshaders.

Another example of where a subshader can be used is in our planar trees. Our original shader code is as follows:

```
Shader "iPhone/Transparent/Vertex Color 2-Sided Trees"
{
// Expose the color, an alpha
// cutoff slider, and the main
// texture to the inspector
Properties
{
_Color ("Main Color", Color) = (1,1,1,1)
_AlphaCutoff ("Alpha Cutoff", Range (0.1, 1)) = 0.5
_MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}
Category
{
// Trees are transparent and
// will ignore projectors
Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
// Standard blending
Blend SrcAlphaOneMinusSrcAlpha
// Test the texture alpha
// against the slider value
Alphatest Greater [_AlphaCutoff]
// No lighting will be applied
Lighting Off
// Backfaces will not be culled
Cull Off
// All iOS devices
SubShader
{
Pass
{
Tags { "LightMode" = "Always" }
// Apply the color to the
// texture
SetTexture [_MainTex]
{
constantColor [_Color]
Combine texture * constant DOUBLE, texture * constant
}
}
}
}
}
```

Note

If we change the graphics emulation from iPhone 4, 3GS, iPad OpenGL ES2.0 to iPhone (MBXLite), something unexpected happens and our trees disappear. In fact, if we leave the graphics emulation alone and simply change the rendering path on our main camera from forward to Vertex-Lit, the trees disappear again.

While this is not important for our game deployment, because we plan to deploy it only on the latest hardware, it does demonstrate that shaders are platform-specific. We can fix the shader by adding a subshader that will execute on the original iPhone or Vertex-Lit cameras, by adding the following code:

```
// Lightmapped, encoded as dLDR
// Seems like a bug but this is
// needed for MBXLite emulation
SubShader
{
Pass
{
Tags { "LightMode" = "VertexLM" }
// Apply the color to the
// texture
SetTexture [_MainTex]
{
constantColor [_Color]
combine texture * constant DOUBLE, texture * constant
}
}
}
```

We can find more information about the `lightmode` tag on the Unity3D website at the following URL:

http://unity3d.com/support/documentation/Components/SL-PassTags.html

The hardware used by the person playing our game may be capable of executing a particular subshader, but the performance of that subshader, on that hardware, may not be adequate to achieve the frame rate that the player would like to achieve. For example, the **Unity Pro water shader** works on iOS devices, but the frame rate is very poor on all of the devices. If the water is beautiful but the frame rate is too slow, then our game players will not be happy with our game.

The way that we can address this issue is called **Level of Detail** or just **LOD**. The LOD subshader command instructs the shader that we want to use a different subshader rather than the most high-ended supported shader. There are two methods of telling shaders to select subshaders based on their LOD, as follows:

We can assign an integer to the `Shader.GlobalMaximumLOD` class member or we can retrieve a reference to a specific shader using `Shader.Find("name")` and then set the `maximumLOD` member of that instance.

The LOD values for built-in shaders may be found on this page:

http://unity3d.com/support/documentation/Components/SL-ShaderLOD.html

Note

It should be noted that the `globalMaximumLOD` takes precedence over individual shader `MaximumLOD` values, if we set the global value to less than the value that we set on a particular shader, then the subshader will be selected based on the global value.

In some very special circumstances, such as when we want to disable functionality (such as transparency) without compromising quality (such as normal maps), we may choose to replace the shader used instead of using LOD to select a subshader. We can replace the shader by retrieving a reference to a shader using the `Shader.Find()` method and then assign the replacement shader to a `material.shader` instance. Typically, this technique is rarely used, and instead, shaders will progressively downscale, using LOD, until they have gone through all possible **Fallback** shaders.

Begin the pass

By this point, we have likely noticed that the subshader may further be subdivided into **passes**. Passes may use many of the same commands like subshaders, and by doing so, they override the values defined by their parent subshader. Intricate surface shaders are not typically practical for less capable graphics hardware, so when writing iOS shaders, we usually use the less expensive alternative and combine passes (fixed function shaders). After having set our options for our pass (if any), we start our combine operations. This is where the magic happens.

The following code fragment shows a combine operation. It has two primary components, namely, the `SetTexture` command and the `combine` parameters.

```
SetTexture [_MainTex]
{
combine previous * texture, previous + texture
}
```

The `SetTexture` command instructs the shader about which texture to treat as the texture referenced by the `texture` variable. The `combine` command instructs the shader about which bitmap operations to perform the syntax for. A `combine` operation is as follows:

```
combine src1 operator src2, src1 operator src2
```

The parameters prior to the comma refer to the RGB channel, and the parameters after the comma refer to the alpha channel. This operation (+, `-`, `*`, `÷`) operates on every pixel in the 2D array (or constant) of `src1` using `src2`. The result may be returned or referenced in the next `combine` operation using the keyword `previous` as the `src`.

Possible values for `src` are:

- **texture:** The texture set by `SetTexture[_texture]`
- **previous:** The result of the previous combine operation in the pass.
- **primary:** The result of the lighting calculation.
- **constant:** The color set by `ConstantColor [_Color]`

The number of `combines` may be done per pass depends on the graphics hardware, and adding passes will slow down rendering time.

Details on hardware limitations and fragment programs may be found on the Unity3D website here:

http://unity3d.com/support/documentation/Components/SL-SetTexture.html

Alpha testing

After the alpha portion of a bitmap is calculated, the shader program must be instructed as to which percentage of alpha is to be rendered. If we are using an alpha map, then we set `AlphaTest Greater 0`, which causes Unity3D to use premultiplied alpha (a more efficient alpha calculation). The `AlphaTest` is represented by a floating point value equal to the percentage of alpha to be used in the comparison. The bit depth of the display may limit the percentages that can be used. For example, if a display has a 1-bit alpha depth setting, then `AlphaTest Greater 0.5` would be functionally equivalent to `AlphaTest Equal 1`.

Premultiplied alpha testing

Because performance is always a consideration on iOS devices, we want to use premultiplied alpha testing whenever possible.

The essential thing to know is that typical alpha compositing formulas require three multiplications at runtime, but with premultiplication, these multiplications can be reduced. Thus reducing the runtime computation required to render transparency. As with any optimization, there is a downside. The downside is that premultiplication creates incorrect colors around the edges of normal alpha channel images (the ugly white bands). We can reduce the appearance of these white bands by creating an alpha map image, whenever we want to use premultiplied alpha testing.

The easiest way to create an alpha map image is to export your image as a `.png` file. If you prefer to manually create the image, you can find instructions for doing so from Photoshop on the Unity3D website here:

http://unity3d.com/support/documentation/Manual/HOWTO-alphamaps.html

If you are interested in the math behind premultiplied alpha, then you can find a good article here:

http://www.cs.princeton.edu/courses/archive/fall00/cs426/papers/smith95a.pdf

To the screen (BLIT and Rasterization)

Data is written to the screen (after depth, alpha and bitmap calculations have been performed) based on the order in which the shader is queued. In addition to queuing, there are two other things to consider:

- **Channel binding:** The space in which data is to be projected

- **Blending:** How data will combine with the previously buffered data

Unity3D will usually determine which properties will be bound to which channels; the one, typical exception is if we wanted to use one UV channel for one pass and the other for mapping to the next channel.

The shader program must be told how it is to combine the result with existing data in the frame buffer to prevent it from completely overwriting data buffered earlier in the queue.

Note

It should be noted that while the iOS device VPU will mostly prevent the overdraw of geometry, the shader program must be told what information will be drawn on top of it.

Because additive and multiplicative blending options are time consuming, the most common blend command on iOS platforms is as follows:

```
//this is no frills alpha testing
Blend SrcAlphaOneMinusSrcAlpha
```

Documentation for the blend command can be found on the Unity3D website at the following URL:

http://unity3d.com/support/documentation/Components/SL-Blend.html

Fallback

While the `fallback` command appears at the end of this section, because syntactically it appears at the end of the shader file, it is part of the shader scope, after the subshader definitions. The `fallback` command, followed by a string literal, instructs the shader program to search for a shader with a matching name, in the event that a compatible shader with the correct LOD was not located at runtime.

Fun examples

Shaders can be used to have a lot of fun in our games. Here we can look at just a couple of examples that can be a lot of fun:

- Gems
- Simple metal

Gems

In their gem pack, Unity3D provides a good gem shader that works well on devices that support cube maps. However, if our device does not support cube maps, or if we just want to use a less CPU-intensive gem shader, then we can use one or two-faceted gem textures to achieve good results. The following script shows how we can change the script provided by Unity3D, so that it supports gems on less capable iOS devices by using sphere maps instead of cube maps.

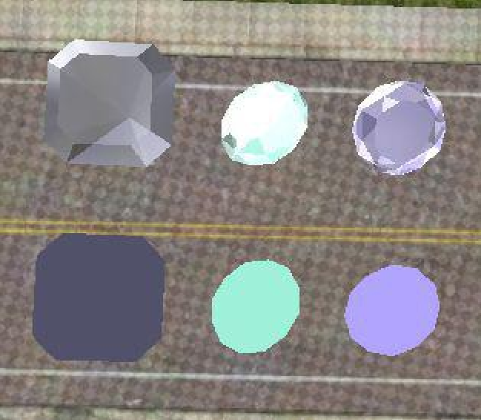
```
Shader "FX/Diamond"
{
Properties {
_Color ("Color", Color) = (1,1,1,1)
_ReflectTex ("Reflection Texture", Cube) = "dummy.jpg" {
TexGenCubeReflect
}
_RefractTex ("Refraction Texture", Cube) = "dummy.jpg" {
TexGenCubeReflect
}
// Add inspector properties for the sphere maps
// and reflective colors
_RefractTexlow ("Refraction LowGPU", 2D) = "dummy.jpg" {
TexGenSphereMap
}
_ReflectTexlow ("Reflect LowGPU", 2D) = "dummy.jpg" {
TexGenSphereMap
}
_Shininess ("Shininess", Range (0.01, 1)) = 0.7
_SpecColor ("Specular", Color) = (1,1,1,1)
_Emission ("Emissive", Color) = (1,1,1,1)
}
SubShader {
Tags {
"Queue" = "Transparent"
}
// First pass - here we render the backfaces of the diamonds. Since those diamonds
are more-or-less
// convex objects, this is effectively rendering the inside of them
Pass {
Color (0,0,0,0)
Offset -1, -1
Cull Front
ZWrite Off
```

```
SetTexture [_ReflectTex] {
combine previous, previous +- texture
}
}
// Second pass - here we render the front faces of the diamonds.
Pass {
Fog { Color (0,0,0,0)}
ZWrite on
Blend One One
SetTexture [_RefractTex] {
constantColor [_Color]
combine texture * constant
}
}
}
// Older cards. Here we remove the bright specular highlight
SubShader {
// Gems should be rendered using transparency
Tags{"Queue" = "Transparent"}
// First pass - here we render the backfaces of the diamonds. Since those diamonds
are more-or-less
// convex objects, this is effectively rendering the inside of them
Pass {
Color (0,0,0,0)
Cull Front
SetTexture [_RefractTex] {
constantColor [_Color]
combine texture * constant, primary
}
}
// Second pass - here we render the front faces of the diamonds.
Pass {
Fog { Color (0,0,0,0)}
ZWrite on
Blend DstColor Zero
SetTexture [_RefractTex] {
constantColor [_Color]
combine texture * constant
}
}
}
// Ancient cards without cubemapping support
// We could use a 2D refletction texture, but the chances of getting one of these
cards are slim, so we won't bother.
SubShader {
// Render the light, reflective back faces
Pass {
Lighting On
SeparateSpecular On
Color (0,0,0,0)
Cull Front
SetTexture [_ReflectTexlow] {
constantColor [_Color]
combine texture * constant, primary
}
}
// Second pass
// Render the front faces
Pass {
Fog { Color (0,0,0,0)}
```

```
ZWrite on
Blend One One
SetTexture [_RefractTexlow] {
constantColor [_Emission]
combine texture * constant
}
}
}
}
```

The difference in setting up the gems is shown in the following screenshot:

The results of using the original and modified scripts on an original iPhone are shown in the following screenshot, where the faceted gems in the upper half of the image are clearly of a higher quality as compared to those in the lower half that look like flat planes:



Simple metal

Whenever we have a real-time light source (it does not work without real-time lighting), we can make objects look metallic and achieve reasonably good performance. We do this by using the built-in **specular** shader as follows:

1. Assign a **Main Color** that is close to the metal you want
2. Use the **eyedropper** tool to choose the same color for the **Specular Color**
3. Adjust the **Specular Color** so that it is slightly brighter
4. Assign a **Base Gloss** with a bit of shading

An example of the resulting metallic object is shown in the following screenshot:

While the previous metal will look good in the Unity3D editor, when you build the project and run it on a device, you will notice that the faces that are directly lit are rendered as totally washed out. They appear white.

This is an example of why we need to test the project on a device, because even though everything may look like it's working perfectly in the editor, in the real world, it looks awful.

The reason is that some shaders are simply not optimized to work around the limitations of the mobile platform.

When we look for a mobile-optimized Specular shader, we discover that the only one provided by Unity3D is a bump-mapped shader, which is doing more than we need for our simple metal.

The solution is to write our own, simplified version of the **Mobile Specular** shader, aptly named `Specular Simple`; it works well on mobile (and desktop) platforms. The shader code that we need for our simple metal is as follows:

```
// Simplified Specular shader.
// Differences from Mobile Bumped Specular one:
// - removed the Normalmap
// - added Main Color
// - added Specular color
Shader "iOS/Specular Simple" {
Properties {
_Color ("Main Color", Color) = (1.0,1.0,1.0,1.0)
_SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
_Shininess ("Shininess", Range (0.03, 1)) = 0.078125
_MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
}
SubShader{
```

```
Tags { "RenderType"="Opaque" }
LOD 300
CGPROGRAM
// Force surface shader to use customized
// MobileBlinnPhong Lighting model
#pragma surface surf MobileBlinnPhongexclude_path:prepassno
lightmapnoforwardaddhalfasview
// Customized BlinnPhone for OpenGLES
inline fixed4 LightingMobileBlinnPhong (SurfaceOutput s, fixed3 lightDir, fixed3
halfDir, fixed atten)
{
// Get lighting data and apply it
fixed diff = max (0, dot (s.Normal, lightDir));
fixednh = max (0, dot (s.Normal, halfDir));
// Calculate alpha mask for specular highlight
fixed spec = pow (nh, s.Specular*128) * s.Gloss;
// Computer color based on SpecColor and Highlight
fixed4 c;
c.rgb = (s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb * (_SpecColor.rgb / 2)
* spec) * (atten*2);
c.a = 0.0;
// Return the final color
return c;
}
sampler2D _MainTex;
fixed4 _Color;
half _Shininess;
struct Input {
float2uv_MainTex;
};
// Return the surface output
// to the drawing library
void surf (Input IN, inoutSurfaceOutput o)
{
// Create UV mapped texture
fixed4tex = tex2D(_MainTex, IN.uv_MainTex);
// Create the tinted texture using color
fixed4 c = tex * _Color;
// Set surface output RGB value
o.Albedo = c.rgb;
// Cutoff the specular highlight if pixel alpha is 0
o.Gloss = tex.a;
// Set the output specular
o.Specular = _Shininess;
// Set the output alpha
o.Alpha = tex.a;
}
ENDCG
}
FallBack "Mobile/VertexLit"
}
```

While our simple metal works well on all iOS devices, even old ones, if we know that our target iOS devices will be iPhone3GS or later, we can use the **Reflective/Specular** shader with a cube map to achieve a better looking simple metal (but it will not look good on devices that do not support cube maps). An example of simple metal using that shader is as follows:

Again, this shader works well in the Unity3D editor, but when we run a scene using this shader on a mobile device, we again, notice the washed out appearance of fully lit faces. We have found another shader that needs to be optimized for mobile (and desktop) devices. It's a good thing that we know how to do it.

Water that works

Tip

Bug Alert iPad2 and Unity3D v3.3

If you run a project that uses lightmaps and fog on the iPad 2, then using Unity3D v3.3 or earlier will cause it to crash. This bug has been reported to the folks at Unity3D and has been fixed in Unity3D v3.4. So, if you try this water on an iPad 2, or with an old copy of Unity3D, then it will crash unless you remove the underwater fog. The best thing to do is to always make sure you are using the latest version of the Unity3D game engine so that known bugs are fixed (and you only need to work around the newly introduced bugs).

There are a few basic things that water needs to do to work nicely on an iOS device:

- Have a reasonably nice translucent texture
- Create a splash when something falls into it
- Create a fog when something is under it
- Move with a current (Optional)

The water script, shown in the following code snippet, accomplishes all of these needs to provide basic water that works on iOS devices for Unity3D:

```
#pragma strict
// This script should be used in conjuction
// wih the /Transparent/SimpleWater shader
// This is a transform at the level of the
// players eyes. It should have a
// kinematicrigidbody and a small sphere
// collider. You can attach TheEyes script
// to any game object to create theEyes
vartheEyes : GameObject;
// This is the particle effect that will be
// used when something enters the water,
// it is "Water Surface Splash" in from
// the standard unity particles package
var splash : Transform;
// The foggynes of the water
vartheFogDensity =.05;
// Set color of the fog. If you set the
// useCameraBackgroundColor your color will
// be replaced by the Main Cameras background
// color
varuseCameraBackgroundColor : boolean = false;
vartheFogColor : Color ;
// A multiplier that affects how foggy the
// water appears on the surface
varmurkeyness : float;
// Interval is how oftwn the routine to
// cause a fade or UV offset will occur
```

```
// Step is how big the offset will be
// FlowStep is the direction the water flows
// FlowSpeed is the speed the water flows
var interval : float;
var step : float;
varflowStep : Vector2;
varflowSpeed : float;
// Fade from 0 to 1 by step
privatevar fade : float = 0;
// True when under the water
privatevar underwater : boolean = false;
// True when water is visible, to make
// it flow only when its visible
privatevar flow = true;
// True when fog is fading in or
// out
privatevar fading = false;
// The density of the original fog
privatevarrevertDensity : float;
// The color of the original fog
privatevarrevertColor : Color;
// State of the original fog
varrevertFog : boolean;
function Start()
{
// Override to use the Camera background color
if (useCameraBackgroundColor)
{
theFogColor=Camera.main.backgroundColor;
}
// Set the material values murkeyness multiplier
// makes the surface appear more or less clear than the
// fog inside
renderer.material.SetFloat("_FogDensity", theFogDensity*murkeyness);
renderer.material.SetColor("_FogColor",theFogColor);
}
// An optimization to cause water flow
// only when visible
functionOnBecameVisible()
{
// Start the water flow
flow = true;
Flow();
}
// An optimization to cause water to not flow
// when invisible
functionOnBecameInvisible()
{
// Stop the water flow
flow=false;
}
function Flow()
{
// An optimization to not do anything
// if the flow speed is too low
if (flowSpeed<0.001)
{
return;
}
//simple periodc UV offset
```

```
while (flow == true)
{
// Move the water material
// to simulate flow
renderer.material.mainTextureOffset += flowStep;
// The reciprocal of flow speed is the
// number of seconds to wait before offsetting again
yieldWaitForSeconds(1/flowSpeed);
}
}
// Something entered the water
functionOnTriggerEnter (Other : Collider)
{
//Create the splash particles at the entry point
if (splash)
{
if (Other.gameObject != theEyes)
{
Instantiate(splash,Other.transform.position, Quaternion.identity);
}
}
// If its not the players eyes the
// do nothing else
if (Other.gameObject != theEyes)
{
return;
}
// The players eyes are underwater
underwater=true;
fade = 0;
// If a fade is in progress then
// there is nothing more to do
if (fading)
{
return;
}
// Set the revertDensity and Color
revertFog = RenderSettings.fog;
revertDensity = RenderSettings.fogDensity;
revertColor = RenderSettings.fogColor;
//Start the fade
FadeWater();
// fading = true;
}
// Something left the water
functionOnTriggerExit (Other : Collider)
{
// Create the splash particles at the entry point
if (splash)
{
if (Other.gameObject != theEyes)
{
Instantiate(splash,Other.transform.position, Quaternion.identity);
}
}
// If its not the players eyes the
// do nothing else
if (Other.gameObject != theEyes)
{
return;
```

```
}
// The players eyes are not under water
underwater=false;
fade = 0;
// If a fade is in progress then
// there is nothing more to do
if (fading)
{
return;
}
//Start the fade
FadeWater();
//fading = true;
}
// This fades the water fog in or out
// based on where the players
// eyes are
functionFadeWater()
{
// A fade is in progress
fading=true;
if (underwater)
{
if (!RenderSettings.fog)
{
RenderSettings.fogDensity = 0;
RenderSettings.fog=true;
}
}
// The fade loop
// Counts up by fadeStep
while (fade <1)
{
// Fade the water fog in
if (underwater)
{
// Change the fog to go between current density and
// the density of the water fog
RenderSettings.fogDensity = Mathf.Lerp(revertDensity, theFogDensity, fade);
// Do the same for color
RenderSettings.fogColor = Color.Lerp(revertColor, theFogColor, fade);
}
// Fade the water fog out
else
{
// Change the fog to go between current density
// and the revert density
// of the water fog
RenderSettings.fogDensity = Mathf.Lerp(theFogDensity, revertDensity, fade);
//do the same for color
if (revertFog)
{
RenderSettings.fogColor = Color.Lerp(theFogColor, revertColor, fade);
}
else
{
//do the same for color
RenderSettings.fogColor = Color.Lerp(theFogColor, Color(theFogColor.r, theFogColor.g,
theFogColor.b, 0), fade);
}
```

```
}
fade+=step;
yieldWaitForSeconds(interval);
}
// Restore the original values
if (!underwater)
{
RenderSettings.fog = revertFog;
RenderSettings.fogColor = revertColor;
RenderSettings.fogDensity = revertDensity;
}
// The fade is done
// Allow us to begin a new fade event
fading = false;
}
```

It important to note these lines in the preceding script:

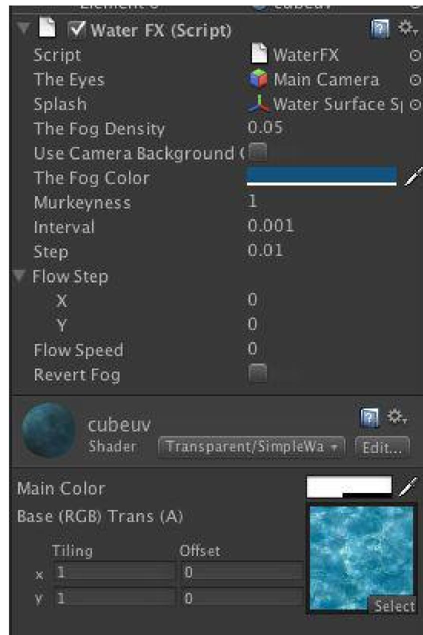
```
// Set the material values murkeyness multiplier
// makes the surface appear more or less clear
// than the fog inside
renderer.material.SetFloat("_FogDensity", theFogDensity*murkeyness);
renderer.material.SetColor("_FogColor",theFogColor);
```

Specifically, the references to `_FogDensity` and `_FogColor` are to be noted. This script is used to set values in the shader. In order for it to work, we need to use a shader that contains those two values. Notice the water shader in the following highlighted lines that also reference `_FogDensity` and `_FogColor`:

```
Shader "Transparent/SimpleWater"
{
Properties
{
_Color ("Main Color", Color) = (1,1,1,1)
_MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}
SubShader
{
Tags
{"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
LOD 200
Cull Off
Fog {Mode Exp2 Color [_FogColor] Density [_FogDensity]
}
CGPROGRAM
#pragma surface surf Lambert alpha
sampler2D _MainTex;
float4 _Color;
// These two variables are set by
// the script and are not exposed
// in the editor
float4 _FogColor;
float _FogDensity;
struct Input
{
float2uv_MainTex;
};
```

```
void surf (Input IN, inoutSurfaceOutput o)
{
half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
o.Albedo = c.rgb;
o.Alpha = c.a;
}
ENDCG
}
Fallback "Transparent/VertexLit"
}
```

The following screenshot shows the configured water in the Unity3D editor:

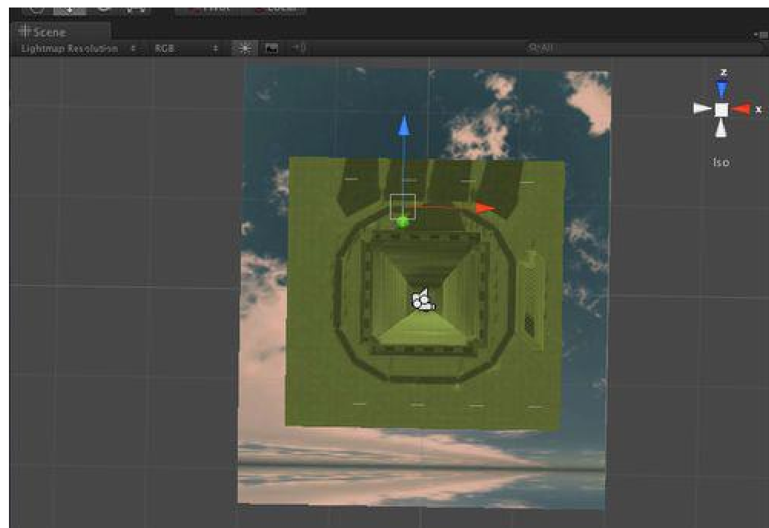


And the following screenshot shows the player under the water:

Task: The final track

If you open the Unity3D project folder named `Chapter 8 Unity Project`, you will find the `Scene1.unity` file in the `Assets` folder.

When you open the scene, you will find a lightmapped terrain that looks like the following screenshot:



If you play the scene and drive the vehicle to the top of the hill, then you will see something that looks like the following screenshot:

Now, using the `Prefabs` and `Objects` in the project, create your own finished track in the scene that includes lights, grandstands, and signs, as you see fit. If you don't want to follow the track marked by the lightmap shadows, then clear the lightmap and be as creative as you like when creating your track.

You can find a sample of a finished track in the game object named `ZZZ_Answers_FinalTrack`; simply enable the `ZZZ_Answers_FinalTrack` game object and its children to see the sample track.

When you are done, be sure to save the scene.

Task: Create a rich and engaging scene

Now, using your saved scene, and the `Prefabs` and `Objects` in the project, create your own finished scene that includes trees, water, lava, and gems as you see fit. If you don't want to follow the scene marked by the lightmap shadows, then clear the lightmap and be as creative as you like when creating your scene.

You can find a sample of a finished track in the game object named `ZZZ_Answers_FinalScene`. Simply enable the `ZZZ_Answers_FinalScene` game object and its children to see the sample scene.

When you are done, be sure to save the scene.

Summary

In this chapter, we have covered:

- The way to prepare and import objects so that they are ready for lightmapping
- How to lightmap a scene using the prepared objects
- The kinds of particle systems that work well on iOS devices
- How to limit and cull particle systems for iOS devices
- How to make realistic looking scene objects that still perform well on iOS devices
- How to create shaders that work across multiple iOS devices and fall back to the capabilities of each device
- How to make reasonable water on the iOS platform that performs well

In the next chapter, we will cover the final parts to finishing a game by integrating it with some backend Internet-based services.

Chapter 9. Putting it all together

There are two major topics left to cover. They are the final pieces we need in order to finish our game.

On an iOS device, people don't have a menu option that allows them to quit from our game. If our game includes the option to quit as a menu item, then Apple will reject it. Instead, we need to develop our game to react to the iOS device's **Home** button being pressed.

There are a number of websites and cloud backend servers that we can use to store game high scores and achievements. Apple has also stepped into this market with the introduction of their **GameKit** API, which allows us to manage a lot of shared game content. The problem with these third-party services is that they capture us and limit what we can do in our game to the services they provide. It turns out that with Unity3D, it is very easy to talk to our own backend server and free ourselves from the limits of any third-party services.

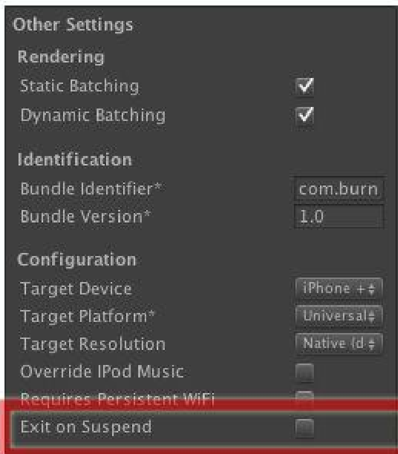
In this chapter, we will learn:

- How to handle the iOS Home button
- How to create WWW forms in Unity3D
- The simple way to store game achievements on our own server
- How to send and receive information between our iOS game and our backend server

Ending or suspending the game under multitasking

When the **Home** button is pressed on an iOS device, the application will either end or suspend. Which event occurs, depends on a `Player Settings` item in the `Other Settings` section, which is aptly titled `Exit on Suspend`, as shown in the following screenshot:

> **Note**
>
> On iOS devices that do not support multitasking, the application will always exit when the **Home** button is pressed.



As game developers, we may need to decide if we want our game to end or suspend.

There is a better alternative, which is to let the game player decide if the game should end or suspend when the **Home** button is pressed.

If we choose to let the player decide, then we need to set the Boolean variable `PlayerSettings.iOS.exitOnSuspend` based on the player's selection from a preference dialog.

However, don't be fooled into thinking you don't have to handle the `OnApplicationQuit` message if you configure your game to suspend. Even games that suspend when the **Home** button is pressed can be killed by the user from the multitasking bar or by the iOS if it is running out of critical resources.

So, while our game may need to handle the `OnApplicationPause` event, it always needs to handle the `OnApplicationQuit` event.

OnApplicationPause

The `OnApplicationPause` method is called on all game objects that implement it when the application is either paused or resumed.

We need to be very careful with what we do in this function; for example, trying to use the `PlayerPrefs` class in this function can lead to a collision with the game engines flushing of the player defaults and it is not recommended. This is just another reason for using a `Hashtable` and flushing it to our own `.plist` file, rather than relying on the built-in `Unity3d` class.

In fact, for the most part, we really should do very little in this function as the iOS should be able to automatically handle everything that needs to be done so that our game resumes exactly where it was paused. A best practice for `OnApplicationPause` is to do nothing. Don't ever implement it unless you have no other way, and I do mean absolutely no other way, to accomplish what you need to do somewhere else.

OnApplicationQuit

The `OnApplicationQuit` method is called on all game objects that implement it when the application is *about to quit*.

Notice the phrase *about to quit*.

When the user presses the **Home** button on an iOS device, and the application is configured to quit rather than suspend, the application is notified that it is about to terminate, but that it still has some time to save its state.

According to Apple, the application has about five seconds to save its state before it is killed by the iOS.

We can think of this as the opposite side of the springboard of death in which an application also has a limited amount of time in which to start before it is killed by the iOS.

Typically, we will write our player preferences to a `.plist` file whenever the player saves the preferences.

Typically, we will write our game state to a `.plist` file whenever the application is quit. Because we only have a short time, it's important to make sure the game state is maintained as the game is played so that it only needs to be written out when the game is quit. It is not a good idea to both determine the game state and write it out in this method.

Top scores board

One of the things that we want to have in our game is a place for players to share their achievements and perhaps gain some bragging rights over their friends. While there are a number of ways to do this, including Apple's own GameKit, we want to do it in a way that draws visitors to our website, not someone else's website.

There are a couple of things that we need in order to pull this off:

- A web host that supports `mysql` databases
- Some PHP scripting knowledge
- Some HTTP posting knowledge

Setting up the database

The first thing that we need to confirm is that our web hosting company provides support for MySQL. Finding a web hosting company that does not support MySQL may actually be harder than finding one that does, but if somehow we managed to find a hosting company that does not support MySQL, then we need to change hosting companies.

Once the database support is sorted out, we need to create the database for games. Once our database is created, we will create a table for players and a second table for the specific achievements of our game.

Create a MySQL database

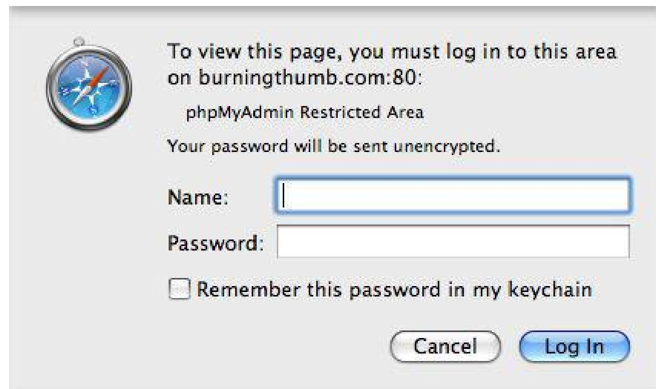
Web hosting companies that support MySQL provide a web frontend to the database called **phpMyAdmin**. If your web hosting company says it supports MySQL but it does not provide phpMyAdmin, then find another company.

Creating a MySQL database with phpMyAdmin is easy:

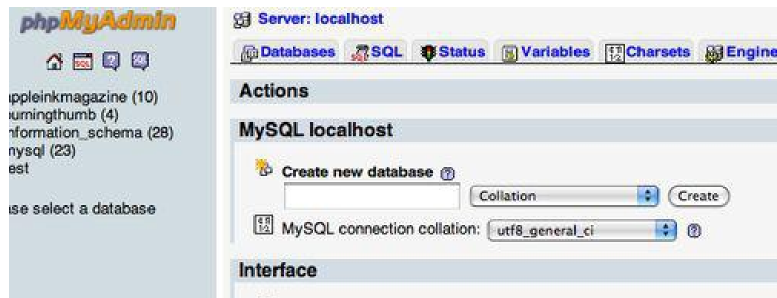
- Log in to phpMyAdmin
- Enter the name of our new database and click on the **Create** button

Log in to phpMyAdmin

Every web hosting company provides their own way of logging into MySQL. Some web hosting companies require you to install MySQL first. We will work with our web hosting company to get MySQL installed and then go to the login page. The login page will look something like the following screenshot:



After we enter our **Name** and **Password**, we will see the main **phpMyAdmin** page, which will look something like the following screenshot:



Enter the name of our new database and click on the **Create** button.

As soon as we see the **Create new database** page, we enter the database name **gamescores** and click on the **Create** button. By doing this, we will have created our database. This is shown in the following screenshot:

Create player and game achievement tables

Once our database is created, we need to create a table that allows our players to log in and a table to store their achievements. You may have noticed that phpMyAdmin is ready to create a new table for us as soon as we have created the new database. It will display the **Create new tables on database gamescores** form, as shown in the following screenshot.

We simply enter **players** in the **Name** box and **3** in the **Number of fields** textbox and click on the **Go** button. The before and after pictures are shown in the following screenshots:

We need to configure the table fields, as shown in the following screenshot, and click on the **Save** button:

And we will get a result that looks like the following screenshot:

Table `gamescores`.`players` has been created.

```
CREATE TABLE `gamescores`.`players` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`name` VARCHAR( 64 ) NOT NULL ,
`password` VARCHAR( 64 ) NOT NULL
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_general_ci
```

Field	Type	Collation	Attributes	Null	Default	Extra
id	int(11)			No	None	auto_increment
name	varchar(64)	utf8_general_ci		No	None	
password	varchar(64)	utf8_general_ci		No	None	

Note

This is a very simple example of a player table. We may want to include fields for optional information such the player's IP address, a banned flag, and so forth. Of course, we can always expand our database later to add more information on players.

In our game, there will be two achievements:

- Lap time
- Cash collected

So, we will create a second table called `racer` that has three fields, one for the player ID, one for the lap time, and one for the cash collected. The results of creating the table are shown in the following screenshots.

The first screenshot shows the settings for the table:

Server: localhost ▸ Database: gamescores ▸ Table: racer

Field	id	laptime	cash
Type	INT	INT	INT
Length/Values[1]			
Default[2]	None	None	None
Collation			
Attributes			
Null	☐	☐	☐
Index	PRIMARY	---	---
A_I	☐	☐	☐
Comments			

The second screenshot shows the results after the table has been created:

Server: localhost ▸ Database: gamescores ▸ Table: racer

Browse | Structure | SQL | Search | Insert | Export

Table `gamescores`.`racer` has been created.

```
CREATE TABLE `gamescores`.`racer` (
 `id` INT NOT NULL ,
 `laptime` INT NOT NULL ,
 `cash` INT NOT NULL ,
 PRIMARY KEY ( `id` )
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_general_ci
```

	Field	Type	Collation	Attributes	Null	Default	Extra
☐	id	int(11)			No	None	
☐	laptime	int(11)			No	None	
☐	cash	int(11)			No	None	

Check All / Uncheck All With selected:

Writing the server PHP scripts

As our web hosting provider supports MySQL and phpMyAdmin, we can be confident the web pages written using the PHP scripting languages will work on our site. We are going to need three scripts to:

- Add a new player
- Update the player's score
- Retrieve the top scores

Add a new player

To add a new player, we need to know the player's name and the player's password. We also need to return a result, which indicates either that the addition was successful or that the addition failed. If it failed, then we also need to return the reason why it failed.

Because we already have the `PropertyListSerialzer.js` script, which converts a `.plist` file to a `Hashtable`, we are going to have our PHP scripts return a property list. We will convert these into a `Hashtable` so that we can easily inspect the results of our database operations.

The PHP Scripts that add a new player to our player's table are shown in the following code snippets:

> ## Tip
>
> Username and password
>
> Notice the username and password are hardcoded into the `enums.php` script. If the `enums.php` script is run on another server, then the username and password need to be updated to match the username and password on that server.

`enums.php`

```
<?php
classerrorCodes
{
constsql = 1;
constnotemail = 2;
const duplicate = 3;
const authentication = 4;
}
class authentication
{
const username = 'ourusername';
const password = 'ourpassword';
}
```

```
?>
```

randompassword.php

```
<?php
functioncreatePassword($length)
{
$chars = "234567890abcdefghijkmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ";
$i = 0;
$password = "";
while ($i<= $length)
{
$password .= $chars{mt_rand(0,strlen($chars))};
$i++;
}
return $password;
}
?>
```

diewitherror.php

```
<?php
functiondieWithError($errorCode, $errorString)
{
echo(' <key>result</key><integer>' . $errorCode . '</integer>');
echo(' <key>reason</key><string>' . $errorString . '</string>');
echo(' </dict>');
die ('</plist>');
}
?>
```

plistheader.php

```
<?php
functionechoPlistHeader()
{
echo ('<?xml version="1.0" encoding="UTF-8"?>');
echo ('<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">');
echo ('<plist version="1.0">');
echo ( '<dict>');
}
```

addplayer.php

```
<?php
// Include the enumerated codes
include ("enums.php");
// Include the function to make a random password
include ("randompassword.php");
// Include the function to report errors
include ("diewitherror.php");
// Include the function to echo a .plist header
include ("plistheader.php");
// Write the .plist header
echoPlistHeader();
```

```
// Connect to the MySQL
$db = mysql_connect("localhost", authentication::username, authentication::password);
// Return an error if the connection fails
if (!$db)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Accept either HTTP POST or GET syntax
if (!empty($_POST))
{
$name = mysql_real_escape_string($_POST["name"]);
}
else
{
$name = mysql_real_escape_string($_GET["name"]);
}
// Player names are actually going to be email addresses
// so we filter that here
if (!filter_var($name, FILTER_VALIDATE_EMAIL))
{
dieWithError(errorCodes::notemail, "Not a valid email address");
}
// Select the gamescores database
$db_selected = mysql_select_db("gamescores",$db);
// Return an error if the select fails
if (!$db_selected)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Check for a duplicate
$query = 'select * from players where name = \'' . $name . '\'';
$result = mysql_query($query);
// Return an error if the qurey fails
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
$numrows = mysql_num_rows($result);
if ($numrows)
{
dieWithError(errorCodes::duplicate, "User already exists");
}
// Generate a 10 character password
$password = createPassword(10);
$query = 'insert into players (name, password) values ("' . $name . '", "' .
md5($password) . '")';
$result = mysql_query($query,$db);
// Return an error if the qurey fails
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
echo(' <key>result</key><integer>0</integer>');
echo(' <key>reason</key><string>Success</string>');
echo(' <key>password</key><string>'. $password . '</string>');
echo(' </dict>');
echo('</plist>');
mysql_close($db);
?>
```

To use the server-side scripts, we upload them to our website. The following example uploads them to the `burningthumb` website for demonstration purposes in the directory `racer` and they can be accessed using the following URL:

http://www.burningthumb.com/racer/addplayer.php

Using either a GET or POST URL.

In our Unity3D project, we create a `Preferences` game object and three scripts: `Preferences.js`, `GUIGetNameContent.js`, and `GetPlayer.js`.

Together, these scripts function to request an e-mail address and receive the returned password from the server if the e-mail address is valid and new.

Tip

Don't store the password

This sample code stores the player's e-mail address and password in the clear in the preferences file. This is only done for illustrative purposes. In the real world, the password would not be stored in the clear on the local machine; instead, the player would be asked to enter the password when needed.

The code for those three scripts is shown in the following code extracts:

`Preferences.js`

```
import System.IO;
// This is a shared object that any other
// game object can use to find the Preferences
// instance
staticvarsharedObject : Preferences;
// This is the .NET Hashtable
// that will contain the
// players preferences
staticvarpreferencesHash : Hashtable;
// The file name for the preferences
staticvarpreferencesFileName : String = "com.burningthumb.racer.preferences.plist";
function Awake ()
{
// Preferences is a singleton, if another one
// is created, destroy it
if (sharedObject)
{
if (this != sharedObject)
{
Debug.Log("Preferences already exists. Destroying myself");
Destroy(this.gameObject);
}
return;
}
// This is The One
DontDestroyOnLoad(this);
sharedObject = this;
// If is a clone, rename it
```

```
transform.name = "Preferences";
LoadPlayerPreferences();
}
// This function will load the
// player options and if the
// options plist file is not found
// it will load the options
// from the defaults file
// found in the StreamingAssets
// folder
static function LoadPlayerPreferences()
{
// The location of the games
// options file
varl_plistfile = Path.Combine (Globals.PathToSpecialFolder(), preferencesFileName);
if (File.Exists(l_plistfile))
{
// If the game options file
// exists, load it into a new
// hashtable
preferencesHash = new Hashtable();
PropertyListSerializer.LoadPlistFromFile(l_plistfile, preferencesHash);
}
else
{
// If the game options file
// does not exist, load it from
// the default hashtable
DefaultPlayerOptions();
}
}
// This function will write the options
// hashtable to a file in the users
// preferences folder
static function SavePlayerPreferences()
{
// The full path to the plist file
varl_plistfile = Path.Combine(Globals.PathToSpecialFolder(), preferencesFileName);
// Write the hashtable out to the
// file
PropertyListSerializer.SavePlistToFile(l_plistfile, preferencesHash);
}
// Create an empty hashtable
static function DefaultPlayerOptions ()
{
preferencesHash = new Hashtable();
SavePlayerPreferences();
}
// Set the email address and password
// in the preferences file
static function RegisterEmail(email: String, password : String)
{
preferencesHash.Add("name", email);
preferencesHash.Add("password", password);
SavePlayerPreferences();
}
```

GUIGetNameContent.js

```
// The dialog name is displayed
// as the dialog box title
vardialogName : String;
// The URL to post the form to add
// a new player to the server MySQL
// database
varnewPlayerURL : String;
// The icon for the verify button
varverifyIcon : Texture2D[];
// The players email address
// and a saved copy of the address
varplayerEmail : String;
varsaveplayerEmail : String;
// Index of which icon to display
privatevargVerifyIconIndex: int = 0;
// The state of the URL post
privatevarverifyState : int = 0;
// The hashtable built from the
// plist returned by the web post
privatevarwebResponse : Hashtable;
function Start()
{
// Start the coroutine that will
// talk to the web server
yieldverifyURLs();
}
// This routine sends the post request
// to the web server and converts
// the response to a hashtable
functionverifyURLs()
{
while (1)
{
// Verify state is set to 1 when the user
// presses the verify button
if (1 == verifyState)
{
// Verify state 2 means posting
verifyState = 2;
// Create a form object for sending
// verify request to the server
var form = new WWWForm();
// The email address of the player
form.AddField( "name", playerEmail );
// Create a download object
var download = new WWW( newPlayerURL, form );
// Wait until the download is done
yield download;
// The hash table that will contain
// the response plist after it is
// converted
webResponse = new Hashtable();
// If the download failed
// return an error
if (download.error)
{
webResponse.Add("result", -1);
webResponse.Add("reason", "Could not contact web server, try again later.");
}
else if (false == PropertyListSerializer .LoadPlistFromString(download.text,
```

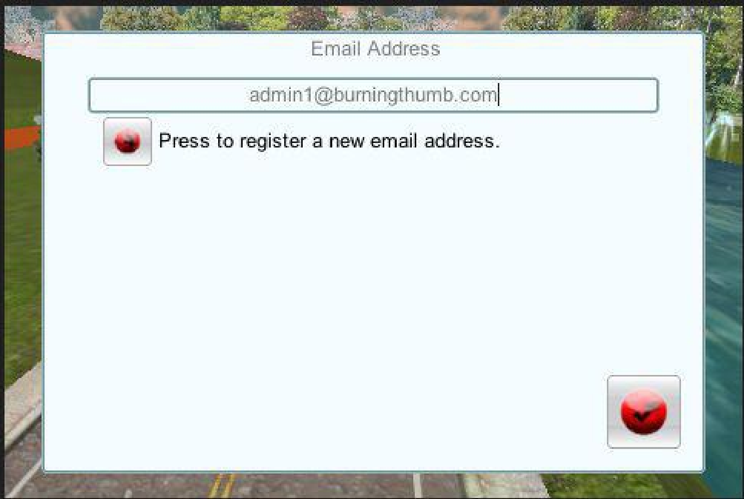
```
webResponse))
{
webResponse.Add("result", -1);
webResponse.Add("reason", "Could not contact web server, try again later.");
}
// If the email was verified a
// password is returned and the
// user information is written to
// the preferences
if (webResponse.ContainsKey("password"))
{
Preferences.RegisterEmail(playerEmail, webResponse["password"]);
}
}
yieldWaitForSeconds(0.1);
}
}
functionrendercontent (boxSize : Vector2)
{
// Some basic GUI variables
varl_box_x : int = 0;
varl_box_y : int = 0;
varl_style : String;
varl_lineHeight : int = 32;
// Draw the dialog box with the correct title
GUI.Box (Rect (l_box_x, l_box_y, boxSize.x, boxSize.y), GUIContent(dialogName));
// Some basic GUI rectangles
varbRect : Rect = Rect(40, 2 * l_lineHeight - 8, 32, 32);
varcRect : Rect = Rect(40 , 4 * l_lineHeight, 32, 32);
var lRect0 : Rect = Rect(l_box_x+30, l_box_y+30, boxSize.x - 60 , 24);
var lRect1 : Rect = Rect(74 , 4 * l_lineHeight + 8, boxSize.x - 60 , 16);
var lRect2 : Rect = Rect(76, 2 * l_lineHeight, boxSize.x - 60 , 16);
// Draw the box where the email address
// will be entered
playerEmail = GUI.TextField(lRect0, saveplayerEmail, "box");
if (playerEmail != saveplayerEmail)
{
gVerifyIconIndex = 0;
webResponse = null;
saveplayerEmail = playerEmail;
}
// Draw the verify button
if (GUI.Button(bRect, verifyIcon[gVerifyIconIndex]))
{
// Change the icon so the player
// knows the request was sent to the
// server
if (1 == gVerifyIconIndex)
{
gVerifyIconIndex = 0;
webResponse = null;
}
else
{
gVerifyIconIndex = 1;
verifyState = 1;
}
}
// If there is no web response display reminder
// text
```

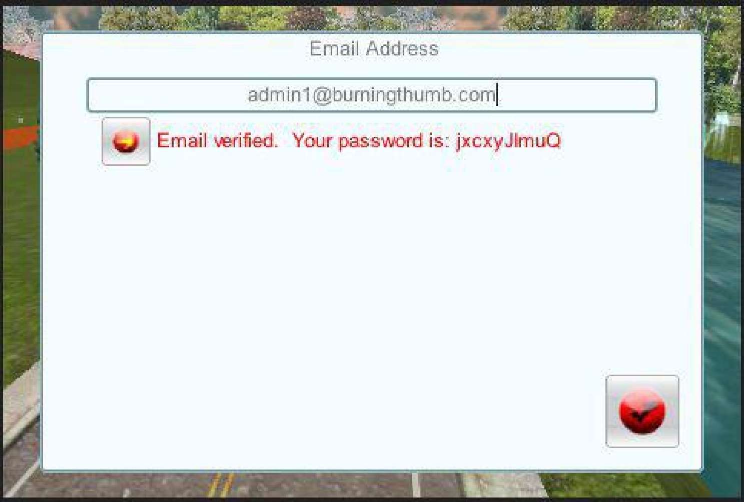
```
if (!webResponse)
{
l_style = "label_left";
GUI.Label(lRect2, "Press to register a new email address.", l_style);
}
// If there is a web response something came back
else
{
l_style = "label_left_red";
// If the response contains a password
// SUCCESS. Display the password
if (webResponse.ContainsKey("password"))
{
GUI.Label(lRect2, "Email verified. Your password is: " + webResponse["password"],
l_style);
}
// Otherwise display the error returned by
// the server
else
{
GUI.Label(lRect2, (webResponse["reason"] as String), l_style);
}
}
}
```

GetPlayer.js

```
varGUIGetNameDialog : GUISlideDialogWrapper;
function Start ()
{
// Wait for the hashtable to exist
// It will, eventually, be created by
// the Preferences class
while (!Preferences.preferencesHash)
{
yieldWaitForSeconds(0.1);
}
// If the preferences does not contain
// and id for this player
// display the dialog and get one if the
// player wants to provide one (it's optional)
if (!Preferences.preferencesHash.ContainsKey("name"))
{
GUIGetNameDialog.displaydialog ();
}
}
```

The following screenshots show an example of the player entering an e-mail address and receiving a password from the server:





The following screenshot shows the values, stored in the clear, in the preferences file on the local computer.

> ## Tip
>
> Don't store the password
>
> This sample code stores the player's e-mail address and password in the clear in the preferences file. This is only done for illustrative purposes. In the real world, the password would not be stored in the clear on the local machine; instead, the player would be asked to enter the password when needed.

com.burningthumb.racer.preferences....

Add Item Delete Item

Key	Type	Value
▼ Root	Dictionary	(2 items)
password	String	jxcxyJlmuQ
name	String	admin1@burningthumb.com

Update the player's score

Before we can update the player's scores on the server, we need to have some basic gameplay to create the score data. For our racing game, we will accumulate money and record the lap time.

When our game player uses the vehicle to push a treasure through the active checkpoint, its cash value will be accumulated in a global variable. Note that if we push a treasure through an inactive checkpoint, nothing will happen. The lap time will be the amount of time it takes from crossing the starting line until the player crosses the finish line.

We assign a value to each treasure by creating a new script named `Treasure.js` with a single variable named `cashValue`. The script is the following single line of code:

```
varcashValue : int = 5;
```

We create checkpoints by adding a trigger collider to our checkpoint and the `Checkpoint.js` script to our game, as shown in the following screenshot:

The `Checkpoint.js` script keeps track of the next checkpoint; it knows if the checkpoint is the start or finish. It provides a visual cue so that the player knows which checkpoint to cross, and uses global variables to keep track of the lap start time and the value of the accumulated treasure. The complete script is as shown in the following code snippet:

```
// Assign the next checkpoint to
// this variable
varnextCheckpoint : Checkpoint;
// If this checkpoint is the start point,
// set this flag to true
varisStart : boolean = false;
// If this checkpoint is the finish point,
// set this flag to true
varisFinish : boolean = false;
// Blends between two materials so that
// the player has a visual indicator
// as to which is the current checkpoint
var material1 : Material;
var material2 : Material;
var duration = 2.0;
privatevar lerp : float = 0;
// Makes sure the player starts the game
// by crossing the first checkpoint within
// a reasonable amount of time. If not the
// lapcounter starts automatically
privatevarstartdelay : float = 0;
privatevardidAutoStart : boolean = false;
function Start()
{
// At start, use the first material
renderer.material = material1;
// Tell the Globals that the start
// point is the next checkpoint
if (isStart)
{
Globals.nextCheckpoint = this;
}
}
function Update ()
{
// If this checkpoint is the next checkpoint
// then provide a visual cue to the player
// by alternating between two materials
if (this == Globals.nextCheckpoint)
{
// ping-pong between the materials over the duration
lerp = Mathf.PingPong (Time.time, duration) / duration;
renderer.material.Lerp (material1, material2, lerp);
}
// If this checkpoint is not the next checkpoint
// fade it back to the original material. This
// makes crossing a checkpoint visually appealing
else if (lerp > 0)
{
lerp = lerp - Time.deltaTime / duration;
if (lerp)
{
renderer.material.Lerp (material1, material2, lerp);
```

```
}
}
else
{
lerp = 0;
renderer.material = material1;
}
// If this checkpoint is the start checkpoint
// and its not crossed within 5 seconds then
// automatically start the lap counter anyway
if (isStart)
{
startdelay = startdelay + Time.deltaTime;
if (startdelay> 5)
{
isStart = false;
didAutoStart = true;
Globals.startTime = Time.time;
}
}
}
// What happens when something crossesa checkpoint
functionOnTriggerEnter (other : Collider)
{
varl_Treasure : Treasure;
// Only do things if this checkpoint is the
// next checkpoint
if (this == Globals.nextCheckpoint)
{
// If the object is a treasure, accumulate
// its cash value and destroy it
l_Treasure = other.gameObject.GetComponent(Treasure);
if (l_Treasure)
{
Globals.totalCash = Globals.totalCash + l_Treasure.cashValue;
Destroy(other.gameObject);
}
// If the object is not a treasure, then its the
// player
else
{
// Activate the next checkpoint
Globals.nextCheckpoint = nextCheckpoint;
// If the lap counter was autostarted
// clear the start flags
if (didAutoStart)
{
didAutoStart = false;
isStart = false;
}
// If this is the start checkpoint,
// start the lap counter
else if (isStart)
{
isStart = false;
didAutoStart = false;
Globals.startTime = Time.time;
}
// If this is the finish checkpoint,
// end the game
```

```
else if (isFinish)
{
isFinish = false;
Globals.endTime = Time.time;
Globals.gameOver = true;
Globals.nextCheckpoint = null;
}
}
}
}
```

The `Globals.js` script includes static variables in order to keep track of our accumulated cash as well as the lap start time, end time, and a flag that indicates if the lap has ended. The following code snippet shows these variables in the `Global.js` script:

```
// This is the next checkpoint in the race
staticvarnextCheckpoint : Checkpoint;
// This is the time the player went through
// the first checkpoint
staticvarstartTime : float;
// This is the time the player went through
// the last checkpoint
staticvarendTime : float;
// This is the cash value of the jewels
// the player pushed through the
// active checkpoint
staticvartotalCash : int;
// A simple flag that says the game is over
staticvargameOver : boolean = false;
```

Finally, we create a GUI to display the lap time and the accumulated cash. The script that renders the GUI content is shown in the following code:

```
// The dialog name is displayed
// as the dialog box title
vardialogName : String;
functionrendercontent (boxSize : Vector2)
{
// Some basic GUI variables
varl_box_x : int = 0;
varl_box_y : int = 0;
varl_style : String;
varl_lineHeight : int = 32;
// Draw the dialog box with the correct title
// The score GUI does not draw a box
// GUI.Box (Rect (l_box_x, l_box_y, boxSize.x, boxSize.y), GUIContent(dialogName));
// Some basic GUI rectangles
varaRect : Rect = Rect(boxSize.x / 2 - 60, boxSize.y / 2 - 12, 120 , 24);
var lRect0 : Rect = Rect(l_box_x+4, l_box_y+4, 120 , 24);
var lRect1 : Rect = Rect(boxSize.x-124, l_box_y+4, 120 , 24);
// If the lap has not yet started
// tell the player to go
if (Globals.startTime< 0.1)
{
l_style = "box_red";
GUI.Label(aRect, "Go! Go! Go!",l_style);
}
```

```
// The lap timer is running
else
{
l_style = "box_black";
// The lap has not ended, so display the current lap time
if (Globals.endTime<Globals.startTime)
{
GUI.Label(lRect0, FormatTime(Time.time - Globals.startTime), l_style);
}
// The lap has ended so display the final lap time
else
{
GUI.Label(lRect0, FormatTime(Globals.endTime - Globals.startTime), l_style);
}
// Display the accumulated cash
GUI.Label(lRect1, String.Format("{0:C}", Globals.totalCash), l_style);
}
}
// This is a utility function to
// return the time as a formatted
// String
functionFormatTime (a_time) : float
{
varintTime : int = a_time;
var minutes : int = intTime / 60;
var seconds : int = intTime % 60;
var fraction : int = (a_time * 100) % 100;
timeText = String.Format ("{0:00}:{1:00}:{2:00}", minutes, seconds, fraction);
returntimeText;
}
```

The following screenshot shows our lap in progress with both the lap timer and the accumulated cash displayed:

Now that we have developed a system to accumulate a score for our player, we need to create the server-side script that updates the player's high score in the database. The script is as follows:

```
<?php
```

```
// Include the enumerated codes
include ("enums.php");
// Include the function to make a random password
include ("randompassword.php");
// Include the function to report errors
include ("diewitherror.php");
// Include the function to echo a .plist header
include ("plistheader.php");
// Write the .plist header
echoPlistHeader();
// Connect to the MySQL
$db = mysql_connect("localhost", authentication::username, authentication::password);
// Return an error if the connection fails
if (!$db)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Accept either HTTP POST or GET syntax
if (!empty($_POST))
{
$name = mysql_real_escape_string($_POST["name"]);
$password = mysql_real_escape_string($_POST["password"]);
$cash = mysql_real_escape_string($_POST["cash"]);
$laptime = mysql_real_escape_string($_POST["laptime"]);
}
else
{
$name = mysql_real_escape_string($_GET["name"]);
$password = mysql_real_escape_string($_GET["password"]);
$cash = mysql_real_escape_string($_GET["cash"]);
$laptime = mysql_real_escape_string($_GET["laptime"]);
}
// Select the gamescores database
$db_selected = mysql_select_db("gamescores",$db);
// Return an error if the select fails
if (!$db_selected)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Authenticate the player
$query = 'select * from players where name = \'' . $name . '\' and password = \''.
md5($password) . '\'';
$result = mysql_query($query);
// Return an error if the query fails
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
$numrows = mysql_num_rows($result);
if (1 != $numrows)
{
dieWithError(errorCodes::authentication, "Invalid email address or password");
}
$row = mysql_fetch_assoc($result);
$id = $row['id'];
// If a worse score exists, update it
// If no score exists, add it
$query = 'select * from racer where id = \'' . $id . '\'';
$result = mysql_query($query);
// Return an error if the query fails
```

```
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
$row = mysql_fetch_assoc($result);
$needtoupdate = 0;
if ($row)
{
if (($cash > $row['cash']) || ($cash == $row['cash'] && $laptime< $row['laptime']))
{
$needtoupdate = 1;
}
}
else
{
$needtoupdate = 1;
}
if ($needtoupdate)
{
$query = 'insert into racer (id, cash, laptime) values (\'' . $id . '\', \'' . $cash
. '\', \'' . $laptime . '\') ON DUPLICATE KEY UPDATE cash=\'' . $cash . '\', laptime
= \'' . $laptime . '\'';
$result = mysql_query($query);
// Return an error if the query fails
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
}
echo(' <key>result</key><integer>0</integer>');
echo(' <key>reason</key><string>Success</string>');
echo(' </dict>');
echo('</plist>');
mysql_close($db);
?>
```

Once the script that updates the high score database is installed on our server, we need to send the new high score information to the server when the lap finishes. The script fragment that does this is as follows:

```
// Post high score
if (Preferences.preferencesHash.ContainsKey("name"))
{
if (Preferences.preferencesHash.ContainsKey("password"))
{
// Create a form object for sending
// verify request to the server
form = new WWWForm();
// The email address of the player and
// the score information
form.AddField( "name", (Preferences.preferencesHash['name'] as String));
form.AddField( "password", (Preferences.preferencesHash['password'] as String));
form.AddField( "cash", Globals.totalCash );
form.AddField( "laptime", (Globals.endTime - Globals.startTime) * 100 );
// Create a download object
download = new WWW( newScoreURL, form );
// Wait until the download is done
yield download;
}
```

```
}
```

Retrieve the top score

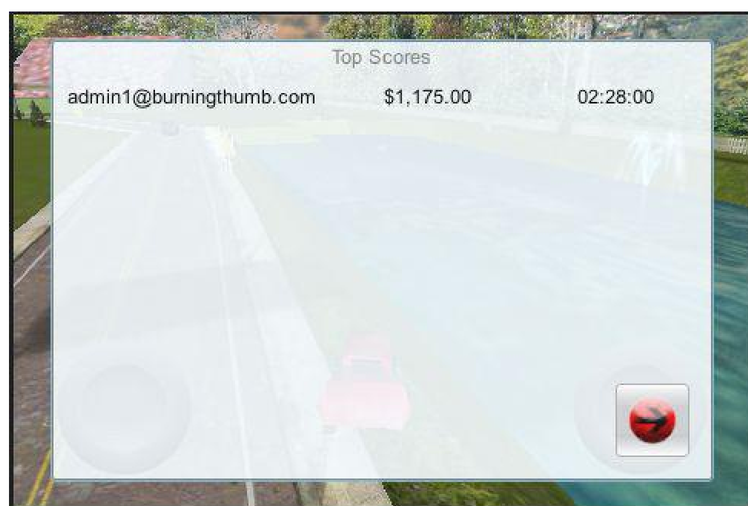
Finally, we need a server-side script to retrieve the top scores from the MySQL database. The script that retrieves the scores is shown in the following code snippet. This script connects to the database, authenticates the player's password, and then retrieves and returns a property list that contains the high scores:

```
<?php
// Include the enumerated codes
include ("enums.php");
// Include the function to make a random password
include ("randompassword.php");
// Include the function to report errors
include ("diewitherror.php");
// Include the function to echo a .plist header
include ("plistheader.php");
// Write the .plist header
echoPlistHeader();
// Connect to the MySQL
$db = mysql_connect("localhost", authentication::username, authentication::password);
// Return an error if the connection fails
if (!$db)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Select the gamescores database
$db_selected = mysql_select_db("gamescores",$db);
// Return an error if the select fails
if (!$db_selected)
{
dieWithError(errorCodes::sql, mysql_error());
}
// Authenticate the player
$query = 'select players.name, racer.cash, racer.laptime from players, racer where
players.id = racer.id order by racer.cash, racer.laptime limit 10';
$result = mysql_query($query);
// Return an error if the query fails
if (!$result)
{
dieWithError(errorCodes::sql, mysql_error());
}
echo(' <key>name</key>');
echo(' <array>');
while($row=mysql_fetch_array($result))
{
echo(' <string>' . $row['name'] . '</string>');
}
echo(' </array>');
mysql_data_seek ($result, 0);
echo(' <key>cash</key>');
echo(' <array>');
while($row=mysql_fetch_array($result))
{
echo(' <integer>' . $row['cash'] . '</integer>');
```

```
}
echo(' </array>');
mysql_data_seek ($result, 0);
echo(' <key>laptime</key>');
echo(' <array>');
while($row=mysql_fetch_array($result))
{
echo(' <integer>' . $row['laptime'] . '</integer>');
}
echo(' </array>');
echo(' <key>result</key><integer>0</integer>');
echo(' <key>reason</key><string>Success</string>');
echo(' </dict>');
echo('</plist>');
mysql_close($db);
?>
```

And once the top scores have been retrieved, we can display them using a Unity3D dialog script (not shown, but included in the accompanying project).

The results of running this script are shown in the following screenshot:



Task: Publish results online

In addition to publishing the top scores in our game dialog, we can create a web page to publish the top scores for the game on our website. The script that does this is essentially a rework of the `topscores.php` script that we use to retrieve the `.plist` file in our game, but instead of formatting a `.plist` file, it formats the results as HTML for display on a web page.

We have created a sample answer file in the `code` folder in the file named `web_topscores.php`. The page that it renders is shown in the following screenshot:

Task: Build and deploy the finished game to multiple iOS devices

If you open the `Unity3D` project folder named `Chapter 9 Unity Project`, you will find the `Scene1.unity` file in the `Assets` folder.

Using everything you have learned in this book, prepare that scene for deployment to multiple iOS devices, build it, and test it out on your own iOS devices.

Summary

In this chapter, we have covered:

- The suspend and terminate events that are generated when the iOS **Home** button is pressed and how to manage them in Unity3D
- How to create a MySQL database on our web host's server
- How to create WWW Forms and WWW objects in Unity3D to communicate with our server storage
- How to write PHP scripts to receive requests and reply with plist-formatted data
- How to create our own web pages that display our game data

In the end, we have examined the most important differences between developing a game for Unity3D on the desktop and Unity3D on the iOS platform. We have examined how to structure our game, the kind of game that is suitable to a mobile platform, the ways to optimize games for iOS, and finally put it all together by interfacing our iOS Unity3D game to backend web services.

We have created a game development strategy that allows us to leverage desktop computing resources to reduce our overall game development time without compromising on the features we implement in our game for the mobile platform.

We have examined the professional quality and features provided by Unity3D. In doing so, we can be confident that Unity3D for iOS is an ideal game engine for developing and deploying a professional-quality game on the iOS mobile platform.