Working with External Resource Files   
and Devices

In this chapter, we will cover:

Loading external resource files – using Unity Default Resources

Loading external resource files – by downloading files from the Internet

Loading external resource files – by manually storing files in the Unity   
Resources folder

Saving and loading player data – using static properties

Saving and loading player data – using PlayerPrefs

Saving screenshots from the game

Setting up a leaderboard using PHP/MySQL

Loading game data from a text file map

Managing Unity project code using Git version control and GitHub hosting

# Introduction

For some projects, it works fine to use the Inspector window to manually assign imported assets to the component slots, and then build and play the game with no further changes. However, there are also many times when external data of some kind can add flexibility and features to a game. For example, it might add updateable or user-editable content; it can allow memory of user preferences and achievements between scenes, and even game-playing sessions. Using code to read local or Internet file contents at runtime can help file organization and separation of tasks between game programmers and the content designers. Having an arsenal of different assets and long-term game memory techniques means providing a wide range of opportunities to deliver a rich experience to players and developers alike.

## The big picture

Before getting on with the recipes, let's step back and have a quick review of the role of the asset files and the Unity game building and running process. The most straightforward way   
to work with assets is to import them into a Unity project, use the Inspector window to assign the assets to the components in the Inspector, and then build and play the game.

Standalone executables offer another possible workflow, which is the adding of files into the Resources folder of the game after it has been built. This will support game media asset developers being able to provide the final version of assets after development and building has been completed.

However, another option is to use the WWW class to dynamically read assets from the web at runtime; or perhaps, for communication with a high score or multiplayer server, and sending and receiving information and files.

When loading/saving data either locally or via the web interface, it is important to keep in mind the data types that can be used. When writing C# code, our variables can be of any type permitted by the language, but when communicated by the web interface, or to a local storage using Unity's PlayerPrefs class, we are restricted in the types of data that we can work with. Unity's WWW class permits three file types (text files, binary audio clips, and binary image textures), but, for example, for 2D UIs we sometimes need Sprite images and not Textures, so that we have provided in this chapter a C# method to create a Sprite from a Texture. When using the PlayerPrefs class, we are limited to saving and loading integers, floats, and strings. Similarly, when communicating with a web server using the URL encoded data, we are restricted to whatever we can place into strings (we include a PHP web-based high score recipe, where the user scores can be loaded and saved via such a method).

Finally, managing Unity project source code with an online Distributed Version Control System (DVCS) like Git and GitHub opens up new workflows for the continuous integration of code updates to the working builds. Unity Cloud will pull the updated source code projects from your online repository, and then build the game for designated versions of Unity and the deployment devices. Developers will get e-mails to confirm the build success, or to list the reasons for any build failure. The final two recipes in this chapter show you how to manage your code with Git and GitHub, and use Unity Cloud to build projects for multiple devices.

# Loading external resource files – using Unity Default Resources

In this recipe, we will load an external image file, and display it on the screen, using the Unity Default Resources file (a library created at the time the game was compiled).

## Getting ready

In the 1362\_10\_01 folder, we have provided an image file, a text file, and an audio file in the .ogg format for this recipe:

externalTexture.jpg

cities.txt

soundtrack.ogg

## How to do it...

To load the external resources by Unity Default Resources, do the following:

Create a new 3D Unity project.

In the Project window, create a new folder and rename it Resources.

Import the externalTexture.jpg file and place it in the Resources folder.

Create a 3D cube.

Add the following C# Script to your cube:

using UnityEngine;

using System.Collections;

public class ReadDefaultResources : MonoBehaviour {

public string fileName = "externalTexture";

private Texture2D externalImage;

void Start () {

externalImage = (Texture2D)Resources.Load(fileName);

Renderer myRenderer = GetComponent<Renderer>();

myRenderer.material.SetTexture("\_MainTex", externalImage);

}

}

Play the scene. The texture will be loaded and displayed on the screen.

If you have another image file, put a copy into the Resources folder. Then, in the Inspector window, change the public file name to the name of your image file and play the scene again. The new image will now be displayed.

## How it works...

The Resources.Load(fileName) statement makes Unity look inside its compiled project data file called resources.assets for the contents of a file named externalTexture. The contents are returned as a texture image, which is stored into the externalImage variable. The last statement in the Start() method sets the texture of the GameObject   
the script has been attached to our externalImage variable.

## There's more...

There are some details that you don't want to miss.

### Loading text files with this method

You can load the external text files using the same approach. The private variable needs to be a string (to store the text file contents). The Start() method uses a temporary TextAsset object to receive the text file contents, and the text property of this object contains the string contents that are to be stored in the private variable textFileContents:

public class ReadDefaultResourcesText : MonoBehaviour {

public string fileName = "textFileName";

private string textFileContents;

void Start () {

TextAsset textAsset = (TextAsset)Resources.Load(fileName);

textFileContents = textAsset.text;

Debug.Log(textFileContents);

}