1
Bonus Chapter 10: Working with External Resource Files and Devices

In this chapter, we will cover the following topics:

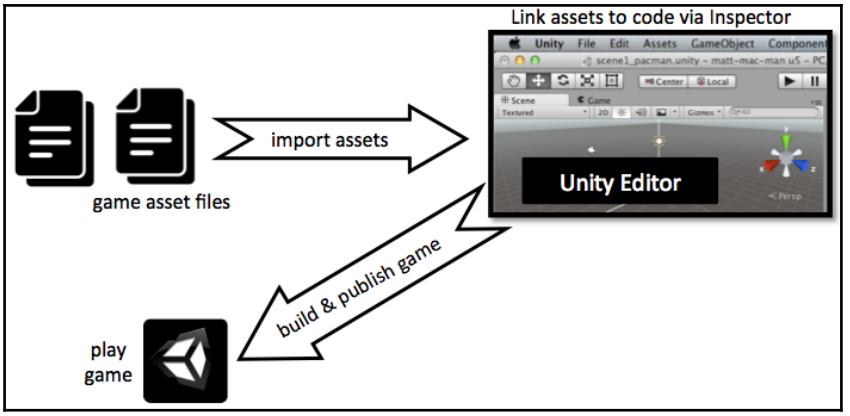
- Loading external resource files—using Unity default resources
- Loading external resource files—downloading files from the internet
- Loading external resource files—manually storing files in the Unity resources or `StreamingAssets` folders
- Saving project files into Unity Asset Bundles
- Loading resources from Unity Asset Bundles

Introduction

For some projects, it works well 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 content at runtime can help file organization and the separation of tasks between game programmers and 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

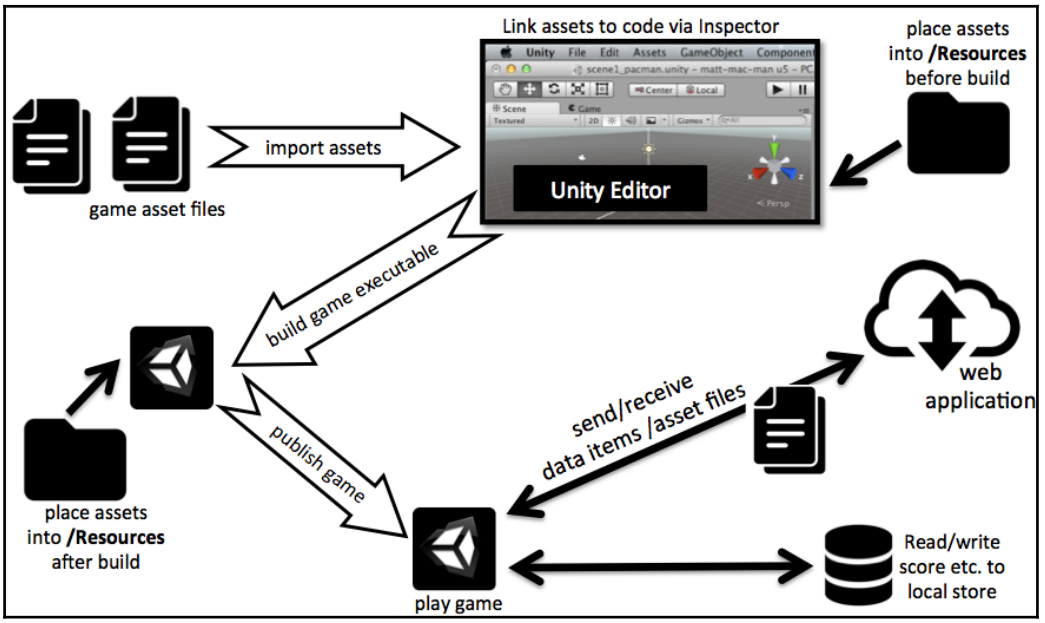
Before getting on with the recipes, let's step back and have a quick review of the role of the asset files and the Unity game building and running process. The most straightforward way to work with assets is to import them into a Unity project, use the **Inspector** window to assign the assets to the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



The WWW is a small class and is straightforward to use. In recent years, Unity has introduced the **Unity Web Request system**, as part of its networking library, for creating and processing HTTP messages (requests and responses). While the **WWW** class is sufficient for the resource saving and loading recipes in this chapter, if you are likely to want to work in a more sophisticated way with HTTP messages, then you would be advised to learn about Unity Web Requests, such as at the following links:

- Unity manual section on Unity Web Requests: `https://docs.unity3d.com/Manual/UnityWebRequest.html`
- Example use of a Unity Web Request on GitHub: `https://gist.github.com/emoacht/89590d4e4571d40f9e1b`
- Examples of HTTP GET/PUT, and so on, requests from StackOverflow: `https://stackoverflow.com/questions/46003824/sending-http-requests-in-c-sharp-with-unity`

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).

This method is perhaps the simplest way to store and read the external resource files. However, it is only appropriate when the contents of the resource files will not change after compilation, since the contents of these files are combined and compiled into the `resources.assets` file.

The `resources.assets` file can be found in the `Data` folder for a compiled game:

Getting ready

In the 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 from Unity Default Resources, do the following:

1. Create a new 3D Unity project.
2. In the `Project` window, create a new folder and rename it `Resources`.
3. Import the `externalTexture.jpg` file and place it in the `Resources` folder.
4. Create a 3D cube, and name it `Cube-1`.
5. Create a C# `ReadDefaultResources` script class and add an instance object as a component to `Cube-1`:

```
using UnityEngine;

 public class ReadDefaultResources : MonoBehaviour {
   public string fileName = "externalTexture";
   private Texture2D externalImage;

   void Start () {
     externalImage = (Texture2D)Resources.Load(fileName);
     Renderer myRenderer = GetComponent<Renderer>();
     myRenderer.material.mainTexture = externalImage;
 }
 }
```

6. Play the scene. The `texture` will be loaded and displayed on the screen.
7. 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.

The string variable `fileName` is public property, so you can select **GameObject Cube-1** in the Hierarchy, and edit the **File Name** string in the **Read Default Resources (Script)** component in the Inspector.

The filename string passed to `Resources.Load()` does not include the file extension (such as `.jpg` or `.txt`).

There's more...

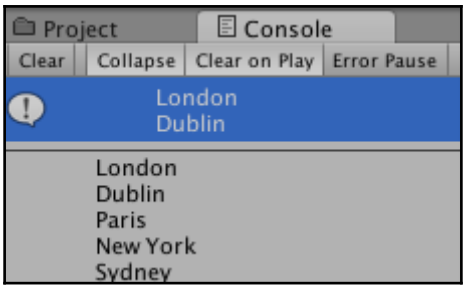
Here are some details that you won't want to miss.

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 content). The `Start()` method uses a temporary `TextAsset` object to receive the text file content, and the text property of this object contains the string content that are to be stored in the private `textFileContents` variable:

```
using UnityEngine;

 public class ReadDefaultResourcesText : MonoBehaviour {
   public string fileName = "textFileName"; // e.g.: cities.txt
   private string textFileContents;

   void Start () {
     TextAsset textAsset = (TextAsset)Resources.Load(fileName);
     textFileContents = textAsset.text;
     Debug.Log(textFileContents);
   }
 }
```

Finally, this string is displayed on the console:



Loading and playing audio files with this method

You can load external audio files using the same approach. The private variable needs to be an `AudioClip`:

```
using UnityEngine;

[RequireComponent (typeof (AudioSource))]
public class ReadDefaultResourcesAudio : MonoBehaviour {
   public string fileName = "soundtrack";

   void Start () {
     AudioSource audioSource = GetComponent<AudioSource>();
     audioSource.clip = (AudioClip)Resources.Load(fileName);
     if(!audioSource.isPlaying && audioSource.clip.loadState ==
AudioDataLoadState.Loaded)
           audioSource.Play();
 }
 }
```

We don't try to play the `AudioClip` until loading has been completed, which we can test with the audio clip's `loadState` property. Learn more about **Audio Load State** in the Unity scripting reference pages: `https://docs.unity3d.com/ScriptReference/AudioDataLoadState.html`

See also

Refer to the following recipes in this chapter for more information:

- Loading external resource files by manually storing files in Unity `Resources` folder
- Loading external resource files by downloading files from the internet

Loading external resource files by downloading files from the internet

One way to store and read a text file data is to store the text files on the web. In this recipe, the content of a text file for a given URL are downloaded, read, and then displayed.

Getting ready

For this recipe, you need to have access to the files on a web server. If you run a local web server such as Apache, or have your own web hosting, then you can use the files in the `10_02` folder and the corresponding URL.

Otherwise, you may find the following URLs useful, since they are the web locations of an image file (a Packt Publishing logo) and a text file (an ASCII-art badger picture):

- `www.packtpub.com/sites/default/files/packt_logo.png`
- `www.ascii-art.de/ascii/ab/badger.txt`

How to do it...

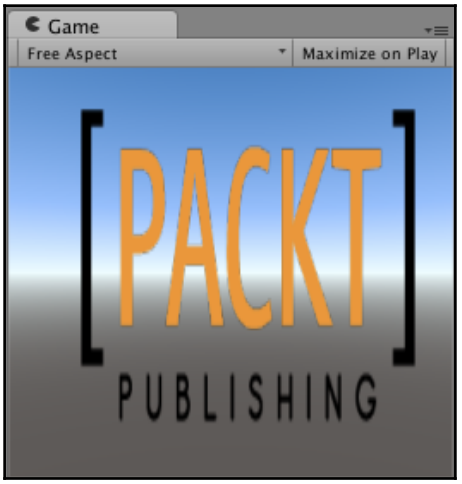
To load external resources by downloading them from the Internet, do the following:

1. In a 3D project, create a new **RawImage UI GameObject**.
2. Create a C# **ReadImageFromWeb** script class and add an instance object as a component to the **RawImage GameObject**:

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

 public class ReadImageFromWeb : MonoBehaviour {
     public string url =
"http://www.packtpub.com/sites/default/files/packt_logo.png";

     IEnumerator Start() {
         WWW www = new WWW(url);
         yield return www;

         Texture2D texture = www.texture;
         GetComponent<RawImage>().texture = texture;
```

```
        }
    }
```

3. Play the **Scene.** Once downloaded, the content of the image file will be displayed:



How it works...

Note the need to use the `UnityEngine.UI` package for this recipe.

When the game starts, our `Start()` method starts the coroutine method called `LoadWWW()`. A **coroutine** is a method that can keep on running in the background without halting or slowing down the other parts of the game and the frame rate. The yield statement indicates that once a value can be returned for `imageFile`, the remainder of the method can be executed—that is, until the file has finished downloading, no attempt should be made to extract the texture property of the `WWW` object variable.

Once the image data has been loaded, execution will progress past the yield statement. Finally, the texture property of the **RawImage GameObject**, to which the script is attached, is changed to the image data that is downloaded from the web (inside the texture variable of the WWW object).

There's more...

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

Converting from Texture to Sprite

While in the recipe, we used a **UI RawImage**, and so we could use the downloaded **texture** directly; however, there may be times when we wish to work with a **Sprite** rather than a **Texture**. To create a **Sprite** object from a **texture create the following script-class:**

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

 public class ImageFromWebTextureToSprite : MonoBehaviour {
     public string url =
"http://www.packtpub.com/sites/default/files/packt_logo.png";

     IEnumerator Start() {
         WWW www = new WWW(url);
         yield return www;

         Texture2D texture = www.texture;
         GetComponent<Image>().sprite = TextureToSprite(texture);
     }

     private Sprite TextureToSprite(Texture2D texture) {
         Rect rect = new Rect(0, 0, texture.width, texture.height);
         Vector2 pivot = new Vector2(0.5f, 0.5f);
         Sprite sprite = Sprite.Create(texture, rect, pivot);
         return sprite;
     }
 }
```

Downloading a text file from the web

Use this technique to download a text file (attach an instance of a script class to a **UI Text** object):

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

 public class ReadTextFromWeb : MonoBehaviour {
```

```
    public string url =
"http://www.ascii-art.de/ascii/ab/badger.txt";

    IEnumerator Start() {
        Text textUI = GetComponent<Text>();
        textUI.text = "(loading file ...)";
        WWW www = new WWW(url);
        yield return www;

        string textFileContents = www.text;
        Debug.Log(textFileContents);
        textUI.text = textFileContents;
    }
}
```

The WWW class and the resource content

The WWW class defines several different properties and methods to allow the downloaded media resource file data to be extracted into appropriate variables for use in the game. The most useful of these include the following:

- `.text`: A read-only property, returning the web data as string
- `.texture`: A read-only property, returning the web data as a `Texture2D` image
- `.GetAudioClip()`: A method that returns the web data as an `AudioClip`

For more information about the Unity WWW class, visit `http://docs.unity3d.com/ScriptReference/WWW.html`

An example using UnityWebRequest

Rather than the `WWW` class, we can also download the texture using the `UnityWebRequest` library. Just replace the content of the `ReadImageFromWeb` script class with the following:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.Networking;
public class ReadImageFromWebUnityWebRequest : MonoBehaviour {
 public string url =
```

```
"http://www.packtpub.com/sites/default/files/packt_logo.png";
IEnumerator Start() {
 using (UnityWebRequest uwr = UnityWebRequestTexture.GetTexture(url))
{
 yield return uwr.SendWebRequest();
if (uwr.isNetworkError || uwr.isHttpError)
 Debug.Log(uwr.error);
 else {
 Texture2D texture = DownloadHandlerTexture.GetContent(uwr);
 UpdateUIRawImage(texture);
 }
 }
 }

 private void UpdateUIRawImage(Texture2D texture) {
 GetComponent<RawImage>().texture = texture;
 }
}
```

See also

Refer to the following recipes in this chapter for more information:

- Loading external resource files by **Unity Default Resources**
- Loading external resource files by manually storing files in the Unity `Resources` folder

Loading external resource files by manually storing files in the Unity Resources or StreamingAssets folders

At times, the content of the external resource files may need to be changed after the game compilation. Hosting the resource files on the web may not be an option. There is a method of manually storing and reading files from the `Resources` folder of the compiled game, which allows for those files to be changed after the game compilation.

The `Resources` folder technique works when you compile to a Windows or Mac standalone executables. The `SteamingAssets` folder technique works with these, and also iOS and Android devices. The next recipe illustrates the Resources folder technique, and then at the end we discuss how to use the `StreamingAssets` approach.

Getting ready

The `10_01` folder provides the texture image that you can use for this recipe:

- `externalTexture.jpg`

How to do it...

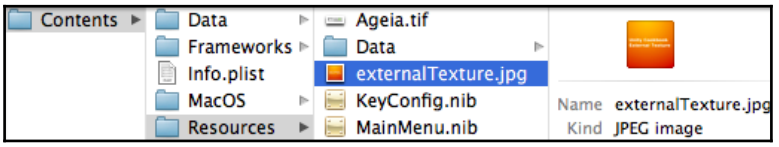
To load external resources by manually storing the files in the Resources folder, do the following:

1. Create a new 3D project.
2. Create a new **UI Image GameObject**. Make this take up most of the screen.
3. Create a new **UI Text GameObject**. Position this to stretch the full width of the screen, along the bottom.
4. Create a C# **ReadManualResourceImageFile** script class and add an instance object as a component to the **UI Image GameObject**:

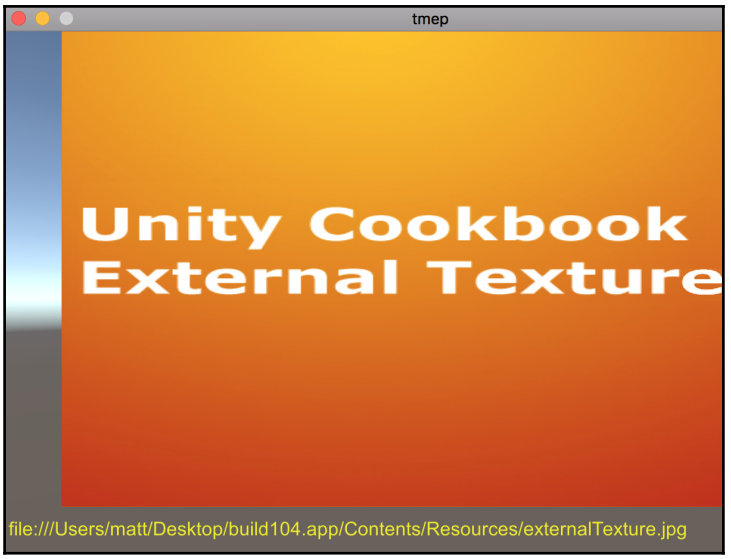
```
using System.Collections;
 using UnityEngine;
 using UnityEngine.UI;
 using System.IO;

 public class ResourceFileLoader : MonoBehaviour
 {
     public Text textUrl;

     private string fileName = "externalTexture.jpg";
     private string urlPrefixMac = "file://";
     private string urlPrefixWindows = "file:///";

     IEnumerator Start()
     {
         //        string url = urlPrefixWindows +
Application.dataPath;
         string url = urlPrefixMac + Application.dataPath;
```

```
            url = Path.Combine(url, "Resources");
            url = Path.Combine(url, fileName);

            textUrl.text = url;

            WWW www = new WWW(url);
            yield return www;

            Texture2D texture = www.texture;
            GetComponent<Image>().sprite =
TextureToSprite(texture);
        }

        private Sprite TextureToSprite(Texture2D texture)
        {
            Rect rect = new Rect(0, 0, texture.width,
texture.height);
            Vector2 pivot = new Vector2(0.5f, 0.5f);
            Sprite sprite = Sprite.Create(texture, rect, pivot);
            return sprite;
        }
    }
```

5. With the **UI Image GameObject** selected in the **Hierarchy**, drag the **UI Text** object to populate the public **Text URL** property, in the Inspector, for the **ResourceFileLoader (Script)** component.
6. Save the current scene, and add it to the build settings.
7. Build your (Windows or Mac) standalone executable.
8. Copy the **externalTexture.jpg** image to your standalone's `Resources` folder:



9. Run your standalone game application, and the image will be read at run-time from the **Resources** folder and displayed (and the path to that resource will be shown in the **UI Text** at the bottom of the application window):



How it works...

A URL-path is defined, stating where Unity can find the desired image in the Resources folder of the standalone application build. This path is shown onscreen by setting the text property of the **UI Text GameObject** to that path's string value.

The **WWW** object spots that the URL starts with the file-protocol, and so Unity attempts to find the external resource file in its Resources folder (waiting until it has finished loading), and then load its content.

The `Start()` method is declared as an `IEnumerator`, allowing it to run as a **co-routine**, and so wait until the WWW class object completes its loading of the image file:

You will need to place the files in the Resources folder manually after every compilation.
When you create a Windows or Linux standalone executable, there is also a `_Data` folder, created with the executable application file. The Resources folder can be found inside this Data folder.
A Mac standalone application executable looks like a single file, but it is actually a macOS package folder. Right-click on the executable file and select **Show Package Contents**. You will then find the standalone's Resources folder inside the Contents folder.

We are using the file protocol for the URL, which must start in the form: `file:///:`
For the OSX standalone, the Unity Application.dataPath will return a path in the form `/user/location-to-Contents`, so we prefix this with `file://` to get a valid file-protocol URL in the form `file:///user/location-to-Contents`.
For Windows the standalone for the Unity Application.dataPath will return a path in the form `C:Projects/MyUnityProject/location-to-Data`, so we prefix this with `file:/// to get a valid file-prototcol` URL in the form `file:///C:Projects/MyUnityProject/location-to-Data`

Learn more from the Unity documentation and Unity answers pages:

- Scripting `Application.dataPath`: `https://docs.unity3d.com/ScriptReference/Application-dataPath.html`
- Unity Answers: `https://answers.unity.com/questions/517414/how-to-use-www-to-load-local-files.html`

There's more...

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

Avoiding cross-platform problems with Path.Combine() rather than / or \

The filepath folder separator character is different for Windows and Mac file systems (backslash \ for Windows, forward slash / for the Mac). However, Unity knows which kind of standalone you are compiling your project into; therefore, the `Path.Combine()` method will insert the appropriate separator slash character form the file URL that is required.

SteamingAssets folder

If deploying to iOS or Android, the `Resources` folder technique won't work, and you should create and store resource files in a folder named `StreamingAssets`. For example, if you had a text file named `MyTextFile.txt`, you could create a folder named `StreamingAssets` in the **Project** panel, store file `MyTextFile.txt` in that folder, and use the following code to load the content of that file at run-time with your built application:

```
string filePath =
System.IO.Path.Combine(Application.streamingAssetsPath,
"MyTextFile.txt");
string contents = System.IO.File.ReadAllText(filePath);
```

Learn more about the StreamingAsset folder in Unity at the following links: `https://docs.unity3d.com/Manual/StreamingAssets.html` and `https://docs.unity3d.com/ScriptReference/Application-streamingAssetsPath.html`

See also

Refer to the following recipes in this chapter for more information:

- Loading external resource files by Unity Default Resources
- Loading external resource files by downloading files from the internet

Saving Project files into Unity Asset Bundles

Unity provides the **Asset Bundle** mechanism as another way to manage the loading of resources at run-time. **Asset Bundles** can be stored locally, or on the intenet.

In this recipe, we will create a prefab (of a 3D cube **GameObject**), and save it into an **Asset Bundle**.

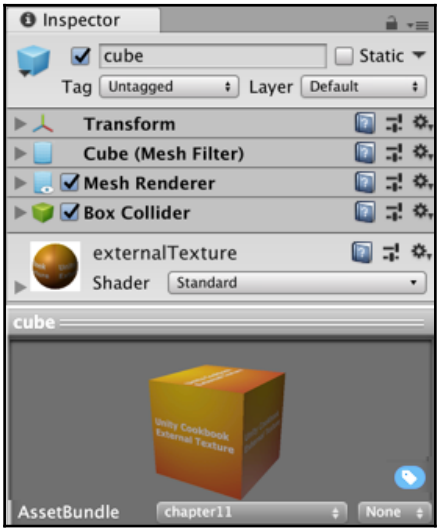
Getting ready

In the `10_06` folder, we have provided the `externalTexture.jpg` image file.

How to do it...

To save Unity **Asset Bundles**, do the following:

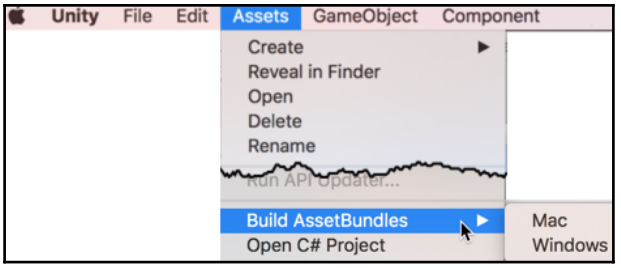
1. Create a new 3D Unity project.
2. Import the provided image file (into a new folder named **Textures**).
3. Create a new **Cube** (named `Cube-1`) in the scene, and apply the imported texure image.
4. In the **Project** window, create a new folder and rename it **Prefabs**.
5. In the **Project** folder **Prefabs**, create a new empty Prefab named **cube**.
6. From the **Scene** panel, drag **GameObject Cube-1** over the Prefab cube in **Project** folder **Prefabs**. The prefab should turn blue and now be a file storing the properties of **GameObject Cube-1**.
7. With the file cube selected in the **Project** panel, go to the bottom of the **Inspector** panel and create a new **AssetBundle**, naming it **chapter11**. See the screenshot illustrating this:



8. In the **Project** panel, create a new folder called `Editor` and `AssetBundles`.
9. In the **Editor** folder, create a new C# **CreateAssetBundles** script class containing the following code:

```
using UnityEditor;

 public class CreateAssetBundles
 {
     [MenuItem("Assets/Build AssetBundles/Mac")]
     static void BuildAllAssetBundlesMac()
     {
         string assetBundleDirectory = "Assets/AssetBundles";
         BuildPipeline.BuildAssetBundles(assetBundleDirectory,
BuildAssetBundleOptions.None, BuildTarget.StandaloneOSX);
     }

     [MenuItem("Assets/Build AssetBundles/Windows")]
     static void BuildAllAssetBundlesWindows()
     {
         string assetBundleDirectory = "Assets/AssetBundles";
         BuildPipeline.BuildAssetBundles(assetBundleDirectory,
BuildAssetBundleOptions.None, BuildTarget.StandaloneWindows);
     }
 }
```

10. You should now see two new menu items added to the bottom of the Assets menu:



11. Choose the create bundles action for the operating system you are using (Mac or Windows).
12. You should now see files created in folder `AssetBundles`: `AssetBundles` and `chapter11`.

How it works...

The `Resources.Load(fileName)` statement makes Unity look inside its compiled project data file called `resources.assets` for the content of a file named `externalTexture`. The content is 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** to which the script has been attached to our `externalImage` variable.

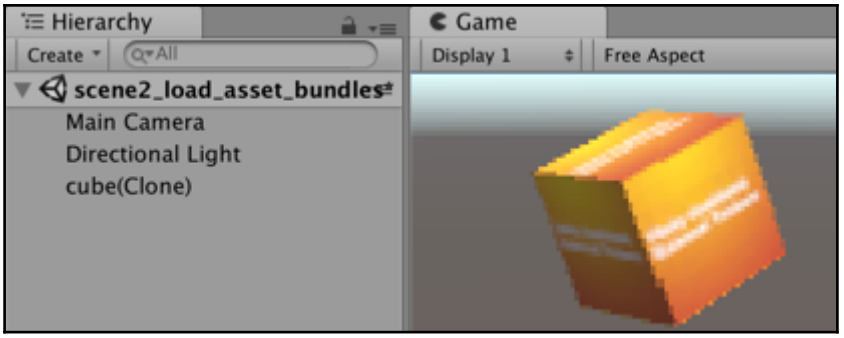
The string varable fileName is a public property, so you can select **GameObject Cube-1** in the Hierarchy, and edit the **File Name** string in the **Read Default Resources (Script)** component in the Inspector.

Learn more about **AssetBundles** and the workflow recommended by Unity at the following addresses:

- Asset Bundles Workflow:
  `https://docs.unity3d.com/2018.1/Documentation/Manual/AssetBundles-Workflow.html`
- Native use of Asset Bundles:
  `https://docs.unity3d.com/2018.1/Documentation/Manual/AssetBundles-Native.html`

Loading resources from Unity Asset Bundles

In this recipe, we will load an **Asset Bundle**, and retrieve a prefab, then create (Instantiate) a `GameObject` in the scene from the retrieved data:



Getting ready

This recipe uses the files created by the previous reciple. We have also provided a copy of folder **AssetBundles** in `10_07` folder.

How to do it...

To load Unity **Asset Bundles**, do the following:

1. Create a new 3D Unity project.
2. Import the provided folder `AssetBundles`.
3. Create the following C# `AssetBundleLoader` script class and add an instance as a component of the **Main Camera**:

```
using UnityEngine;
 using System.IO;
 using UnityEngine.Networking;
 using System.Collections;

 public class AssetBundleLoader : MonoBehaviour
 {
     public string bundleFolderName = "AssetBundles";
     public string bundleName = "chapter11";
     public string resourceName = "cube";

     void Start()
     {
StartCoroutine(LoadAndInstantiateFromUnityWebRequest());
     }

     private IEnumerator
LoadAndInstantiateFromUnityWebRequest()
     {
         // (1) load asset bundle
         string uri = "file:///" + Application.dataPath;
         uri = Path.Combine(uri, bundleFolderName);
         uri = Path.Combine(uri, bundleName);

         UnityWebRequest request =
UnityWebRequestAssetBundle.GetAssetBundle(uri, 0);
         yield return request.SendWebRequest();

         // (2) extract 'cube' from loaded asset bundle
         AssetBundle bundle =
DownloadHandlerAssetBundle.GetContent(request);
```

```
            GameObject cube =
    bundle.LoadAsset<GameObject>(resourceName);

            // (3) create scene GameObject based on 'cube'
            Instantiate(cube);
        }
    }
```

4. Run the **Scene,** and a cube should appear in the scene, created by extracting the **Prefab** cube file from the **Asset Bundle**, and then instantiating a new **GameObject** based on this prefab.

How it works...

Unity method `UnityWebRequestAssetBundle.GetAssetBundle(...)` expects a file-protocol URI to a named **Asset Bundle**. This bundle was loaded into variable bundle, and then the `LoadAsset(...)` method used to extract the file named `cube`. Finally, a GameObject was created at run-time based on this prefab, resulting in the cube that we see when we run the scene.

The variables `bundleFolderName`, `bundleName`, and `resourceName` define the folder, Asset Bundle filename and prefab inside the `Asset` Bundle for this project.

If you select the `chapter11` manifest file (the icon that looks like a piece of paper with lines), then in the Inspector, you can see that it contains our cube prefab:

```
...
 Assets:
 - Assets/Prefabs/cube.prefab
```

Learn more about `AssetBundles` and the workflow recommended by Unity at the following addresses:

- Asset Bundles Workflow:
  `https://docs.unity3d.com/2018.1/Documentation/Manual/AssetBundles-Workflow.html`
- Native use of Asset Bundles:
  `https://docs.unity3d.com/2018.1/Documentation/Manual/AssetBundles-Native.html`

There's more...

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

Loading AssetBundles via AssetBundle.LoadFromFile()

An alternative to `UnityWebRequest` for local loading from tiles is to use `AssetBundle.LoadFromFile()`:

```
       void Start()
        {
            // (1) load asset bundle
            string path = Path.Combine(Application.streamingAssetsPath,
   bundleName);
            AssetBundle myLoadedAssetBundle =
   AssetBundle.LoadFromFile(path);

            if (null == myLoadedAssetBundle)
            {
                Debug.Log("Failed to load AssetBundle: " + path);
                return;
            }

            // (2) extract 'cube' from loaded asset bundle
            string resourceName = "cube";
            GameObject prefabCube =
   myLoadedAssetBundle.LoadAsset<GameObject>(resourceName);

            // (3) create scene GameObject based on 'cube'
            Instantiate(prefabCube);
        }
```

Loading AssetBundles hosted via a web server

Although we have just illustrated how to load an `AssetBundle` from a local file, often such resources are loaded from a web server. In this case, the URI needs to be an internet protocol.

For example, if files are being served locally on port 8000, then the host will be http://localhost:8000 and so on:

```
     public string bundleName = "chapter11";
      public string resourceName = "cube";

      public string host = "http://localhost:8000";

      void Start() {
StartCoroutine(LoadAndInstantiateFromUnityWebRequestServer());
      }

      private IEnumerator LoadAndInstantiateFromUnityWebRequestServer()
{
          string uri = Path.Combine(host, bundleName);
          UnityWebRequest request =
UnityWebRequestAssetBundle.GetAssetBundle(uri, 0);
          yield return request.SendWebRequest();

          AssetBundle bundle =
DownloadHandlerAssetBundle.GetContent(request);
          GameObject cube = bundle.LoadAsset<GameObject>(resourceName);
          Instantiate(cube);
      }
```