10
Working with External Resource Files and Devices

In this chapter, we will cover:

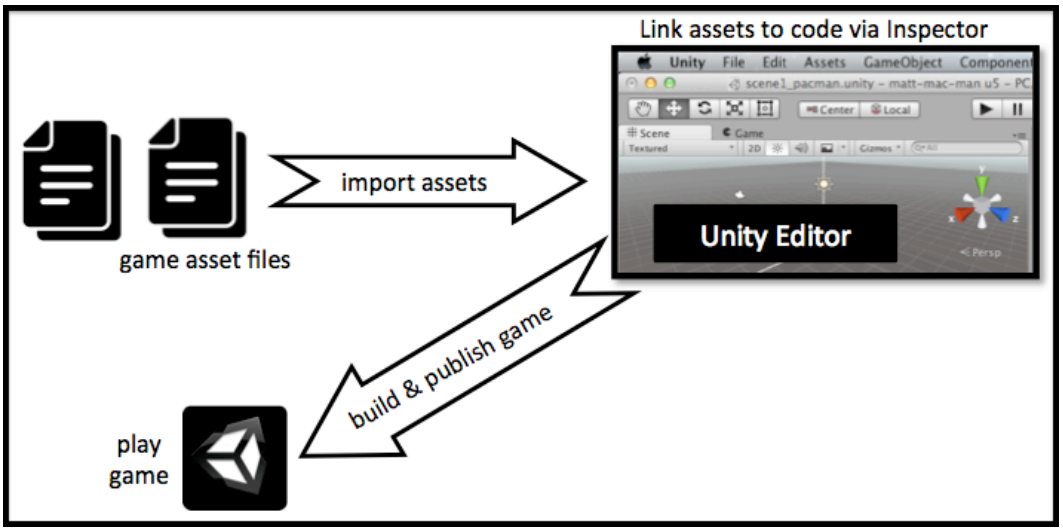
- Loading external resource files - by Unity Default Resources
- Loading external resource files - by downloading file from internet
- Loading external resource files - by manually storing files in Unity Resources folder
- Saving and loading player data - using static properties
- Saving and loading 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
- Publishing for multiple devices via Unity Cloud

Introduction

For some projects, it works fine to use the **Inspector** to manually assign imported assets to 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 run time 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

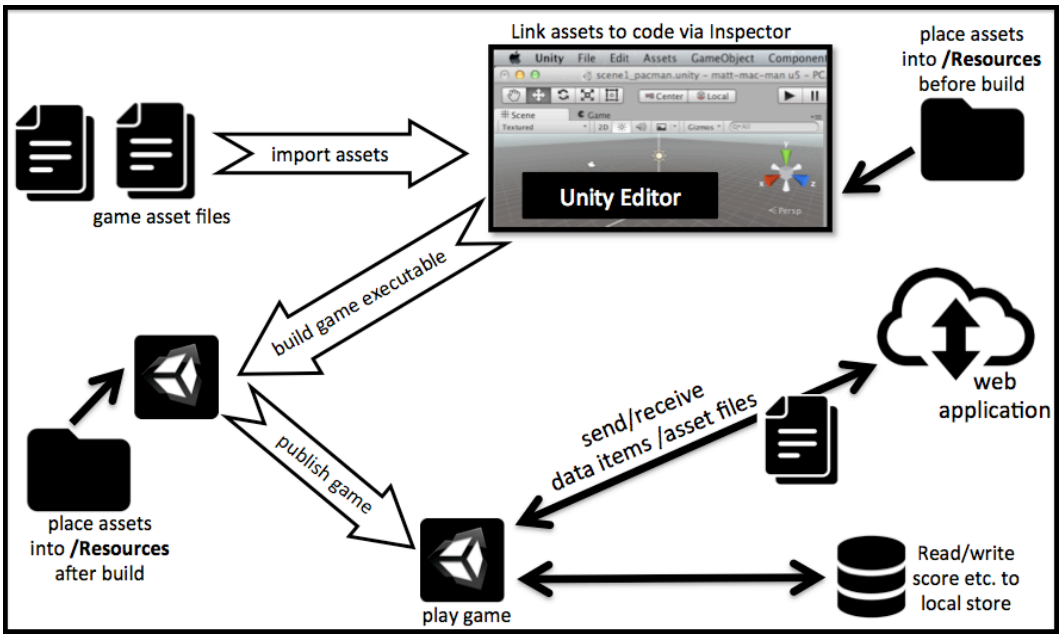
Before getting on with the recipes, let's step back and have a quick review of the role of asset files and the Unity game building and running process. The most straightforward way to work with assets is to import them into a Unity project, use the Inspector to assign the assets to components in the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



Insert image 1362OT_10_01.png

Stand-alone executables offer another possible workflow, which is the adding of files into the "Resources" folder of the game **after** it has been built. This would 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 multi-player server, sending and receiving information and files.



Insert image 1362OT_10_02.png

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 local storage using Unity's **PlayerPrefs** class, we are restricted in the types of data we can work with. Unity's WWW class permits 3 file types (text files, binary audio clips, and binary image textures) - but, for example, for 2D UIs we sometimes need Sprite images, not Textures, so we have provided in this chapter a recipe 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 URL encoded data, we are restricted to whatever we can place into strings (we include a PHP web-based high score recipe where 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 continuous integration of code updates to working builds. **Unity Cloud** will "pull" updated source code projects from your online repository, and then build the game for designated versions of Unity and deployment devices. Developers will get emails to confirm 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 for multiple devices.

Acknowledgement: Thanks to the following for publishing Creative Commons (BY 3.0) licensed icons: Elegant Themes, Picol, Freepik, Yannick, Google, www.flaticon.com.

Loading external resource files - by 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 is compiled).

> NOTE: This method is perhaps the simplest way to store and read 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 text 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.

Insert image 1362OT_10_03.png

Getting ready

In folder `1362_10_01`, we have provided the following for this recipe: an image file, a text file, and an audio file in `Ogg` format:

- `externalTexture.jpg`
- `cities.txt`

- `soundtrack.ogg`

How to do it...

To load external resources by 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 into the `Resources` folder.
4. Create a 3D cube.
5. 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);
   }
}
```

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**, change the public file name to the name of your image file and play the scene again. The new image should now be displayed.

How it works...

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

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

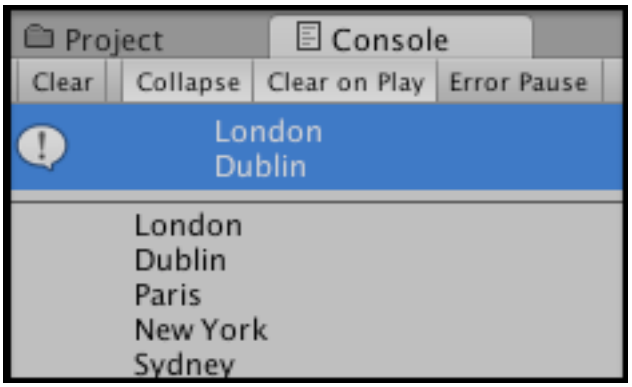
There's more...

Some details you don't want to miss:

Loading text files with this method

You can load 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 to be stored in the `private` variable `textFileContents`. Finally, this string is displayed on the console:

```
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);
   }
}
```



Insert image 1362OT_10_04.png

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;
using System.Collections;

[RequireComponent (typeof (AudioSource))]
```

```
public class ReadDefaultResourcesAudio : MonoBehaviour {
   public string fileName = "soundtrack";
   private AudioClip audioFile;

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

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 file from internet*

Loading external resource files - by downloading file from internet

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

Getting ready

For this recipe, you need to have access to files on a web server. If you run a local web server such as **Apache**, or have your own web hosting, then you could use the files in folder `1362_10_01` and the corresponding URL.

Otherwise, you may find the following URLs useful; since they are the web locations of an image file (a Packt 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 2D project, create a new RawImage UI **GameObject**

2. Add C# script class `ReadImageFromWeb` as a component of your Image object:

```
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 contents of the image file will be displayed:

Insert image 1362OT_10_05.png

How it works...

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

When the game starts, our `Start()` method starts the co-routine method `LoadWWW()`. A co-routine 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 downloaded from the web (inside the `texture` variable of the `www` object).

There's more...

Some details you don't want to miss:

Converting from Texture to Sprite

While in the recipe we used a UI **RawImage**, and so could use the downloaded **Texture** directly, there may be times when we wish to work with a **Sprite** rather than a **Texture**. Use this method to create a **Sprite** object given a **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:

```
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;
```

```
    }
}
```

WWW and resource contents

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

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

> NOTE: For more information about the Unity WWW class see:
>
> `docs.unity3d.com/ScriptReference/WWW.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 manually storing files in Unity Resources folder*

Loading external resource files - by manually storing files in Unity Resources folder

At times, the contents of external resource files may need to be changed after game compilation. Hosting 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 game compilation.

> NOTE: This technique only works when you compile to a Windows or Mac *stand alone executable* – it will not work for Web Player builds for example.

Getting ready

Folder `1362_10_01` provides the texture image you can use for this recipe:

- `externalTexture.jpg`

How to do it...

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

1. In a 2D project, create a new Image UI **GameObject**
2. Add C# script class `ReadManualResourceImageFile` as a component of your Image object:

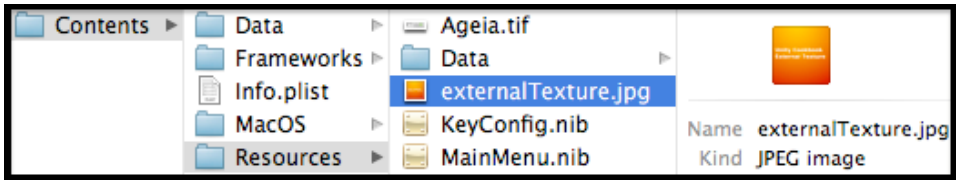
```
using UnityEngine;
using System.Collections;

using UnityEngine.UI;
using System.IO;

public class ReadManualResourceImageFile : MonoBehaviour {
   private string fileName = "externalTexture.jpg";
   private string url;
   private Texture2D externalImage;

    IEnumerator Start () {
      url = "file:" + Application.dataPath;
      url = Path.Combine(url, "Resources");
      url = Path.Combine(url, fileName);

      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;
   }
}
```

3. Build your (Windows or Mac) standalone executable.
4. Copy image `externalTexture.jpg` into your standalone's `Resources` folder.

> NOTE: You will need to place the files in the `Resources` folder manually after **every** compilation.

> WINDOWS and LINUX: When you create a Windows or Linux standalone executable, there is also a `_Data` folder created with executable application file. The `Resources` folder can be found inside this data folder.
>
> MAC: A Mac standalone application executable looks like single file, but it is actually a MacOS "package" folder. Right-click the executable file and select **Show Package Contents**. You will then find the standalone's `Resources` folder inside the `Contents` folder.

5. Run your standalone game application, and the image should be displayed.



Insert image 1362OT_10_06.png

How it works...

Note the need to use the `System.IO` and `UnityEngine.UI` packages for this recipe.

When the executable runs, the `WWW` object spots that the URL starts with the word `file` and so Unity attempts to find the external resource file in its `Resources` folder and then load its contents.

There's more...

Some details 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 to form the file URL that is required.

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 file from internet*

Saving and loading player data - using static properties

Keeping track of the player's progress and user settings during a game is vital to give your game a greater feel of depth and content. In this recipe, we will learn how to make our game "remember" the player's score between different levels (scenes).

Getting ready

We have included a complete project in a Unity package named `game_HigherOrLower` in the folder `1362_10_04`. In order to follow this recipe, make a copy of that folder as your starting point.

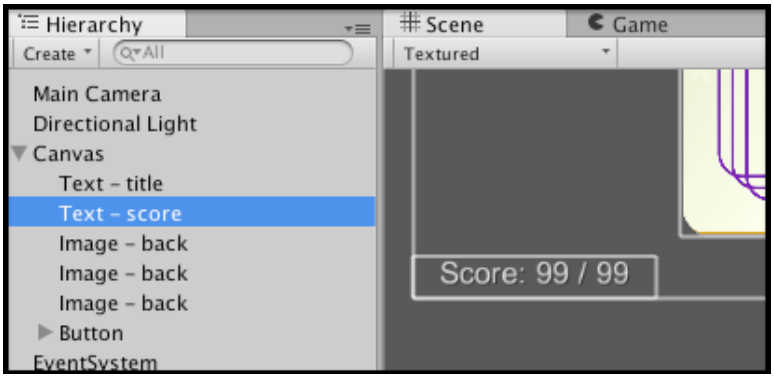
How to do it...

To save and load player data, follow these steps:

1. Create a new 2D project and import the **game_HigherOrLower** package.
2. Add each of the scenes to the build in sequence (**scene0_mainMenu** then **scene1_gamePlaying** and so on).
3. Make yourself familiar with the game by playing it a few times and examining the contents of the scenes. The game starts on scene **scene0_mainMenu**, inside the folder `Scenes`.
4. Let's create a class to store the number of correct and incorrect guesses made by the user. Create a new C# script `Player` with the following code:

```
using UnityEngine;

public class Player : MonoBehaviour {
   public static int scoreCorrect = 0;
   public static int scoreIncorrect = 0;
}
```

5. In the lower left corner of scene **scene0_mainMenu**, create a UI Text **GameObject** named **Text – score** containing the placeholder text `Score: 99 / 99`.



Insert image 1362OT_10_07.png

6. Next, attach the following C# script to UI **GameObject** **Text – score**:

```
using UnityEngine;
using System.Collections;

using UnityEngine.UI;

public class UpdateScoreText : MonoBehaviour {
   void Start(){
      Text scoreText = GetComponent<Text>();
      int totalAttempts = Player.scoreCorrect +
Player.scoreIncorrect;
      string scoreMessage = "Score = ";
      scoreMessage += Player.scoreCorrect + " / " +
totalAttempts;

      scoreText.text = scoreMessage;
   }
}
```

7. In scene **scene2_gameWon**, attach the following C# script to the **Main Camera**:

```
using UnityEngine;

public class IncrementCorrectScore : MonoBehaviour {
   void Start () {
      Player.scoreCorrect++;
   }
}
```

8. In scene **scene3_gameLost**, attach the following C# script to the **Main Camera**:

```
using UnityEngine;
```

```
public class IncrementIncorrectScore : MonoBehaviour {
   void Start () {
      Player.scoreIncorrect++;
   }
}
```

9. Save your scripts and play the game. As you progress from level (scene) to level, you should find that the score and player's name are "remembered", until you quit the application.

How it works...

The `Player` class uses `static` (class) properties `scoreCorrect` and `scoreIncorrect` to store the current total number of correct and incorrect guesses. Since these are public static properties, any object from any scene can access (set or get) these values, since static properties are "remembered" from scene to scene. This class also provides the public static method `ZeroTotals()` that resets both values to zero.

When scene **scene0_mainMenu** is loaded, all GameObjects with scripts will have their `Start()` methods executed. UI Text GameObject **Text – score** has an instance of class `UpdateScoreText` as s script component, so that scripts `Start()` method will be executed, which retrieves the correct and incorrect totals from the `Player` class, creates string `scoreMessage` about the current score, and updates the text property so that the user sees the current score.

When the game is running, if the user guesses correctly (higher) then scene **scene2_gameWon** is loaded. So the `Start()` method of the `IncrementCorrectScore` script component of the Main Camera in this scene is executed, which adds 1 to the `scoreCorrect` variable of class `Player`.

When the game is running, if the user guesses wrongly (lower) then scene **scene3_gameLost** is loaded. So the `Start()` method of the `IncrementIncorrectScore` script component of the Main Camera in this scene is executed, which adds 1 to the `scoreIncorrect` variable of class `Player`.

Next time the user visits the main menu scene, the new values of the correct and incorrect totals will be read from class `Player`, and the UI Text onscreen will inform the user of their updated total score for the game.

There's more...

Some details you don't want to miss:

Hide score before first attempt completed

Showing a score of zero out of zero isn't very professional. Let's add some logic so that the score is only displayed (a non-empty string) if the total number of attempts is greater than zero:

```
void Start(){
   Text scoreText = GetComponent<Text>();
   int totalAttempts = Player.scoreCorrect +
Player.scoreIncorrect;

   // default is empty string
   string scoreMessage = "";
   if( totalAttempts > 0){
      scoreMessage = "Score = ";
      scoreMessage += Player.scoreCorrect + " / " +
totalAttempts;
   }

   scoreText.text = scoreMessage;
}
```

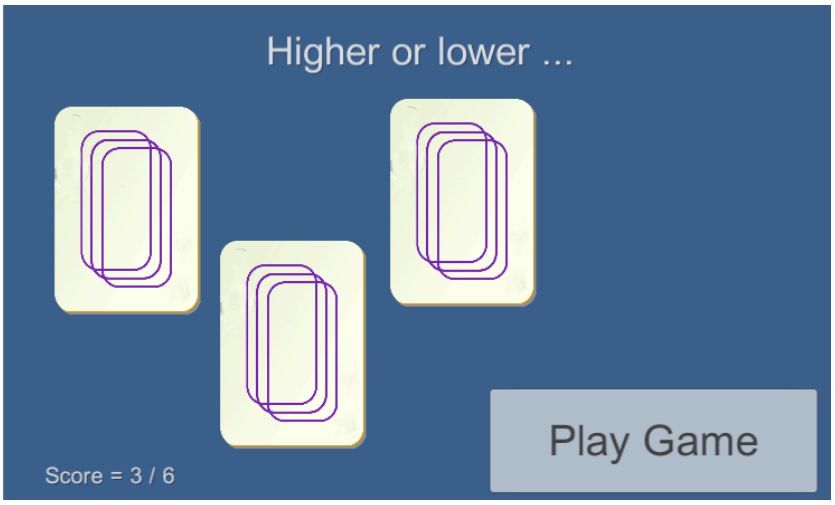
See also

Refer to the following recipe in this chapter for more Information:

- *Saving and loading player data - using PlayerPrefs*

Saving and loading player data - using PlayerPrefs

While the previous recipe illustrates how static properties allow a game to "remember" values between different scenes, those values are "forgotten" once the game application has quit. Unity provides the `PlayerPrefs` feature to allow a game to store and retrieve data between different game playing sessions.



Insert image 1362OT_10_08.png

Getting ready

This recipe builds upon the previous recipe. In case you haven't completed the previous recipe, we have included a Unity package named `game_scoreStaticVariables` in the folder `1362_10_05`. In order to follow this recipe, you must do the following:

1. Create a new 2D project and import the **game_HigherOrLower** package.
2. Add each of the scenes to the build in sequence (**scene0_mainMenu** then **scene1_gamePlaying** and so on).

How to do it...

To save and load player data using PlayerPrefs, follow these steps:

1. Delete the C# script `Player`.
2. Edit C# script `UpdateScoreText` by replacing the method `Start()` with the following:

```
void Start(){
Text scoreText = GetComponent<Text>();

int scoreCorrect = PlayerPrefs.GetInt("scoreCorrect");
int scoreIncorrect = PlayerPrefs.GetInt("scoreIncorrect");

int totalAttempts = scoreCorrect + scoreIncorrect;
string scoreMessage = "Score = ";
scoreMessage += scoreCorrect + " / " + totalAttempts;
```

```
    scoreText.text = scoreMessage;
}
```

3. Now edit C# script `IncrementCorrectScore` by replacing method `Start()` with the following code:

```
void Start () {
    int newScoreCorrect = 1 + PlayerPrefs.GetInt("scoreCorrect");
    PlayerPrefs.SetInt("scoreCorrect", newScoreCorrect);
}
```

4. Now edit C# script `IncrementIncorrectScore` by replacing method `Start()` with the following code:

```
void Start () {
    int newScoreIncorrect = 1 +
PlayerPrefs.GetInt("scoreIncorrect");
    PlayerPrefs.SetInt("scoreIncorrect", newScoreIncorrect);
}
```

5. Save your scripts and play the game. Quit from Unity and then restart the application. You should find that the player's name, level, and score are now kept between game sessions.

How it works...

We had no need for class `Player`, since this recipe uses the built-in Runtime Class that Unity provides called `PlayerPrefs`.

Unity's `PlayerPrefs` Runtime Class is capable of storing and accessing information (`String`, `Int`, and `Float` variables) in the user's machine. Values are stored in a `plist` file (Mac) or the registry (Windows), in a similar way to web browser "cookies", and therefore, "remembered" between game application sessions.

Values for the total correct and incorrect scores are stored by the `Start()` methods in classes `IncrementCorrectScore` and `IncrementIncorrectScore`. These methods use the `PlayerPrefs.GetInt("<variableName>")` method to retrieve the old total, add 1 to it, and then store the incremented total using the `PlayerPrefs.SetInt("<variableName>")` method.

These correct and incorrect totals are then read each time scene **scene0_mainMenu** is loaded, and the score totals displayed via the UI **Text** object on screen.

> NOTE: For more information on `PlayerPrefs`, see Unity's online documentation:

docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html

See also

Refer to the following recipe in this chapter for more Information:

- *Saving and loading player data - using static properties*

Saving screenshots from the game

In this recipe, we will learn how to take in-game snapshots and save it in an external file. Better yet, we will make it possible to choose between three different methods.

NOTE: This technique only works when you compile to a Windows or Mac *stand alone executable* - it will not work for Web Player builds for example.

Getting ready

In order to follow this recipe, please import the package `screenshots`, available in the folder `1362_10_06`, into your project. The package includes a basic terrain and a camera that can be rotated via mouse.

How to do it...

To save screenshots from your game, follow these steps:

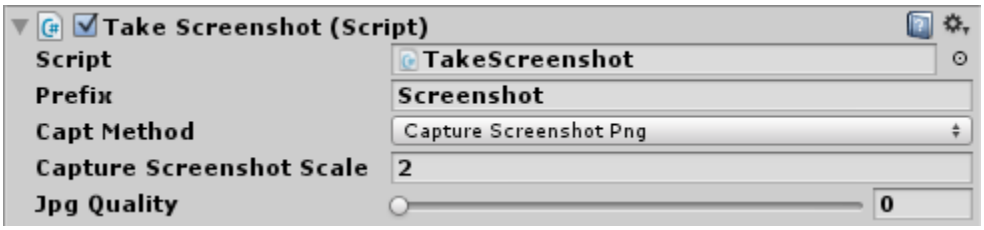
1. Import the **screenshots** package and open the **screenshotLevel** scene.
2. Add the following *C# Script* to the **Main Camera**:

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

public class TakeScreenshot : MonoBehaviour {
public string prefix = "Screenshot";
   public enum method{captureScreenshotPng, ReadPixelsPng,
ReadPixelsJpg};
   public method captMethod = method.captureScreenshotPng;
   public int captureScreenshotScale = 1;
   [Range(0, 100)]
   public int jpgQuality = 75;
```

```
    private Texture2D texture;

    void  Update (){
        if (Input.GetKeyDown (KeyCode.P))
            StartCoroutine(TakeShot());
    }

    IEnumerator  TakeShot (){
        string date = System.DateTime.Now.ToString("_d-MMM-yyyy-HH-
mm-ss-f");
        int sw = Screen.width;
        int sh = Screen.height;
        Rect sRect = new Rect(0,0,sw,sh);
        byte[] bytes;
        string format;
        if (captMethod == method.captureScreenshotPng){
            Application.CaptureScreenshot(prefix + date + ".png",
captureScreenshotScale);
        } else {
            yield return new WaitForEndOfFrame();
            texture = new Texture2D
(sw,sh,TextureFormat.RGB24,false);
            texture.ReadPixels(sRect,0,0);
            texture.Apply();
            if(captMethod == method.ReadPixelsJpg){
                bytes= texture.EncodeToJPG(jpgQuality);
                format = ".jpg";
                Destroy (texture);
                File.WriteAllBytes(Application.dataPath +
"/../"+prefix + date + format, bytes);
            }else if(captMethod == method.ReadPixelsPng){
                bytes= texture.EncodeToPNG();
                format = ".png";
                Destroy (texture);
                File.WriteAllBytes(Application.dataPath +
"/../"+prefix + date + format, bytes);
            }
        }
    }
}
```

3. Save your script and attach it to the **Main Camera** Game Object by dragging it from the **Project** view to the **Main Camera** game object in the **Hierarchy** view.
4. Access the **Take Screenshot** component. Set **Capt Method** as **Capture Screenshot Png**, Change **Capture Screenshot Scale** to 2.

NOTE: If you want your image files name to start with something different than `Screenshot`, change it in the field **Prefix**.



Insert image 1362OT_10_09.png

5. Play the scene. A new screenshot, with twice the original size, will be saved in your project folder every time you press *P*.

How it works...

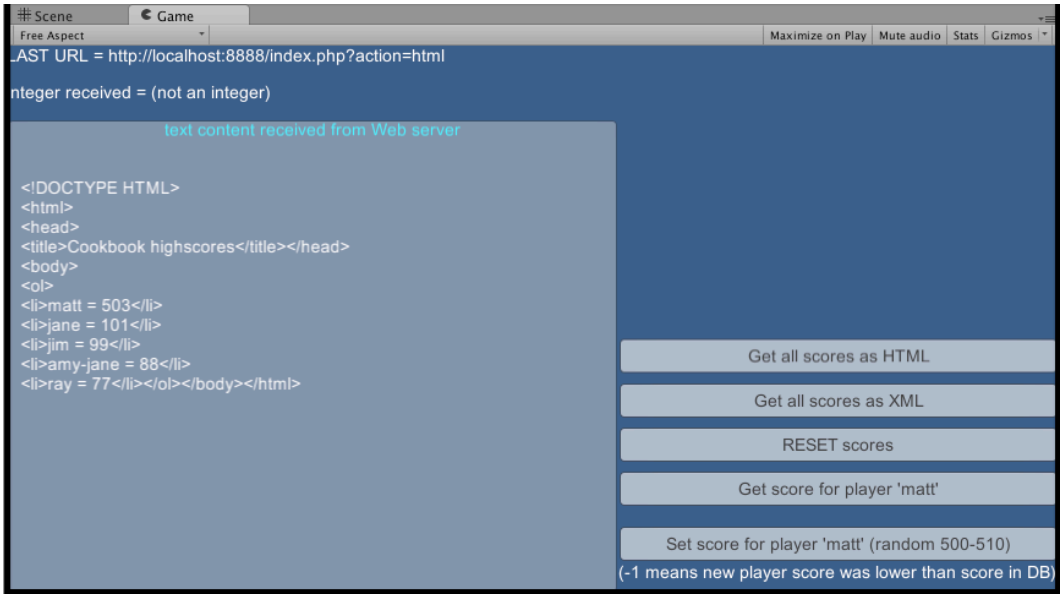
Once the script has detected that the *P* key was pressed, the screen is captured and stored as an image file into the same folder where the executable is. In case the **Capture Screenshot Png** option is selected, the script will call a built-in Unity function called `CaptureScreenshot()`, capable of scaling up the original screen size (in our case, based on the `Scale` variable of our script). If not, the image will be captured by the `ReadPixels` function, encoded to PNG or JPG and, finally, written via the `WriteAllBytes` function.

There's more...

We have included the options using the `ReadPixel` function as a demonstration of how to save your images to disk without using Unity's `CaptureScreenshot()` function. One advantage of this method is that it could be adapted to capture and save only a portion of the screen. The `Scale` variable from our script will not affect it, though.

Setting up a leaderboard using PHP/MySQL

Games are more fun when there is a leaderboard of high scores that players have achieved. Even single player games can communicate to a shared web-based leaderboard. This recipe includes both the client side (Unity) code, as well as the web-server side (PHP) scripts to set and get player scores from a MySQL database.



Insert image 1362OT_10_10.png

Getting ready

This recipe assumes that you either have your own web hosting, or are running a local webserver and database server such as XAMPP or MAMP. Your web server needs to support PHP, and you also need to be able to create MySQL databases.

All the SQL, PHP, and C# scripts for this recipe can be found in folder `1362_10_07`.

Since the scene contains several UI elements, and the code of the recipe is the communication with the PHP scripts and SQL database, in folder `1362_10_07`, we have provided a Unity package called `PHPMySQLeaderboard` containing a scene with everything setup for the Unity project.

> NOTE: If you are hosting your leaderboard on a public website, you should change the names of the database, database user and password for reasons of security. You should also implement some form of secret game code, as described in the *There's more…* section.

How to do it...

To set up a leaderboard using PHP and MySQL, do the following:

1. (On your server) Create a new MySQL database named **cookbook_highscores**.

2. (On your server) Create a new database user (username=cookbook. password=cookbook), with full rights to the database you just created.
3. (On your server) Execute the following SQL to create database table **score_list**:

```
CREATE TABLE `score_list` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `player` varchar(25) NOT NULL,
  `score` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1;
```

4. (On your server) Copy the provided PHP script files to your web server:
   - `index.php`
   - `scoreFunctions.php`
   - `htmlMenu.php`
5. Create a new 2D Unity project, and extract Unity package `PHPMySQLeaderboard`.
6. Run the provided scene, and click the buttons to make Unity communicate with the PHP scripts that have access to the high score database.

How it works...

The player's scores are stored in a MySQL database. Access to the database is facilitated through the PHP scripts provided. In our example, all the PHP scripts were placed into the web server root folder. So, the scripts are accessed via the URL `http://localhost:8888/.` However, since **URL** is a public string variable, this can be set before running to the location of your server and site code.

All access is through the PHP file `index.php`. There are 5 actions implemented, and each is indicated by adding the action name at the end of the URL (this is the GET HTTP method, sometimes used for web forms - take a look at the address bar of your browser next time you search Google for example …). The actions and their parameters (if any) are as follows:

- **action = html, no parameters:**
  - This action asks for HTML text listing all player scores to be returned
  - Returns: HTML text
- **action = xml, no parameters:**
  - This action asks for XML text listing all player scores to be returned
  - Returns: XML text

- **action = reset, no parameters:**
    - this action asks for a set of default player name and score values to replace the current contents of the database table
    - Returns: the string “reset”
- **action = get, parameters:** `player = matt`**:**
    - This action asks for the integer score of the named player to be found
    - Returns: score integer
- **action = set, parameters:** `player = matt, score = 101`**:**
    - This action asks for the provide score of the named player to be stored in the database (but only if this new score is greater than the currently stored score)
    - Returns: score integer (if database update was successful), otherwise a negative value (to indicate no update took place)

There are 5 buttons in the Unity scene (corresponding to the 5 actions). Also, 5 buttons are offered to the user, which set up the corresponding action and parameters to be added to the URL for the next call to the web server via method `LoadWWW()`. `OnClick` actions have been setup for each button, to call the corresponding methods of the `WebLeaderBoard` C# script of the `Main Camera`.

There are also 3 UI Text objects. The first displays the most recent URL string sent to the server. The second displays the integer value that was extracted from the response message received from the server (or message “not an integer” if some other data was received). The third UI Text object is inside a panel, and has been made large enough to display full, multi-line, text string received from the server (which is stored inside variable `textFileContents`).

The 3 UI Text objects have been assigned to public variables of the `WebLeaderBoard` C# script for the `Main Camera`. When the scene first starts, the `Start()` method calls method `UpdateUI()` to update the 3 text UI elements. When any of the buttons are clicked, the corresponding method of `WebLeaderBoard` method is called, which builds the URL string with parameters, and then calls method `LoadWWW()`. This method sends the request to the URL, and waits (by virtue of being a coroutine) until a response is received. It then stores the content received in variable `textFileContents`, and calls method `UpdateUI()`.

There's more...

Here is some information on how to fine tune and customize this recipe:

Extract the full leaderboard data as XML for display within Unity

The XML text that can be retrieved from the PHP web server provides a useful method of allowing a Unity game to retrieve the full set of leaderboard data from the database, and then the leaderboard can be displayed to the user in the Unity game (perhaps in some nice 3D fashion, or through a game-consistent GUI). See the bonus PDFs for several recipes that illustrate ways to work with XML data in Unity.

Using secret game codes to secure your leaderboard scripts

The Unity and PHP code presented illustrates a simple, unsecured web-based leaderboard. To prevent players "hacking" into the board with false scores; it is usual to encode some form of secret game code (or "key") into the communications. Only update requests that include the correct code will actually cause a change to the database.

The Unity code would combine the secret key (in this example, the string "`harrypotter`"), with something related to the communication - for example, the same MySQL/PHP leader board may have different database records for different games, identified with a game Id:

```
// Unity Csharp code
string key = "harrypotter"
string gameId = 21;
string gameCode = Utility.Md5Sum(key + gameId);
```

The server-side PHP code would receive both the encrypted game code, and also the piece of game data used to create that encrypted code (in this example, the game Id, and the MD5 hashing function, which is available both in Unity and in PHP). The secret key ("`harrypotter`") is used with the game Id to create an encrypted code that can be compared with the code received from the Unity game (or whatever user agent or browser is attempting to communicate with the leaderboard server scripts). Database actions will only be executed if the game code created on the server matches that send along with the request for a database action.

```
// PHP - security code
$key = "harrypotter"
$game_id =  $_GET['game_id'];
$provided_game_code =  $_GET['game_code'];
$server_game_code = md5($key.$game_id);

if( $server_game_code == $provided_game_code ) {
   // codes match - do processing here
}
```

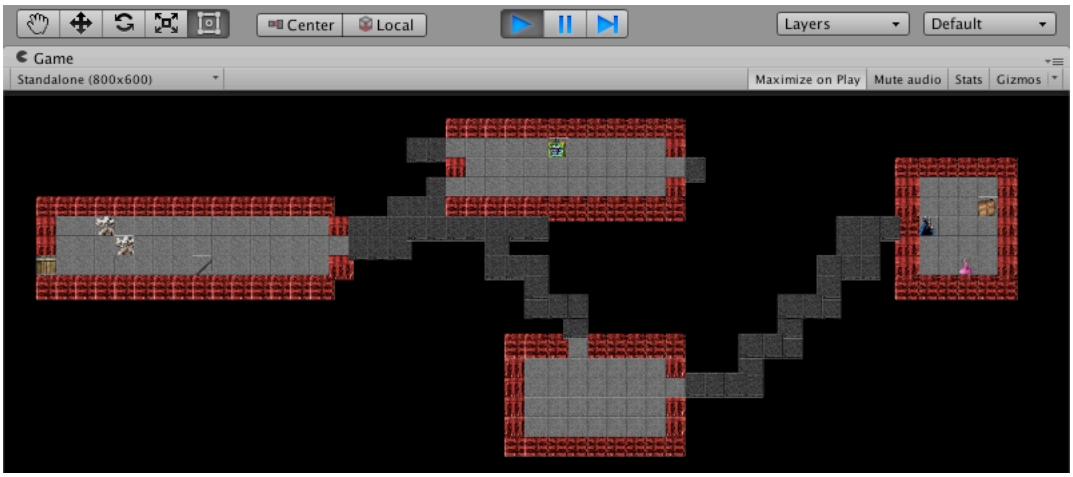
See also

Refer to the following recipe for more Information:

- *Preventing your game from running on unknown servers* in *Chapter 11*

Loading game data from a text file map

Rather than, for every level of a game, having to create and place every **GameObject** on the screen by hand, a better approach can be to create text files of rows and columns of characters, where each character corresponds to the type of **GameObject** to be created in the corresponding location. In this recipe, we'll use a text file and set of prefab sprites to display a graphical version of a text-data file for a screen from the classic game "NetHack".



Insert image 1362OT_10_11.png

Getting ready

In the folder `1362_10_08`, we have provided 2 files for this recipe: `level1.txt` (a text file representing a level), and `absurd128.png` (a 128x128 sprite sheet for Nethack).

The level data came from the Nethack Wikipedia page, and the sprite sheet from SourceForge:

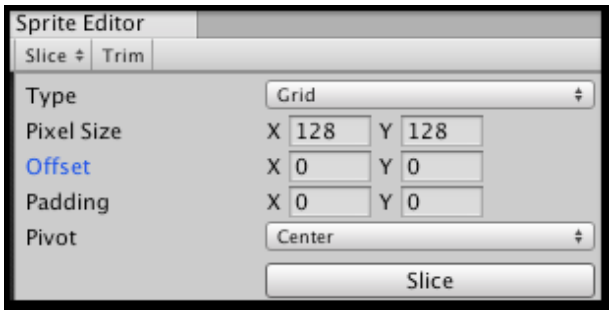
- `http://en.wikipedia.org/wiki/NetHack`
- `http://sourceforge.net/projects/noegnud/files/tilesets_nethack-3.4.1/absurd%20128x128/`

Note that we also included a Unity package with all the prefabs setup, since this can be a laborious task.

How to do it...

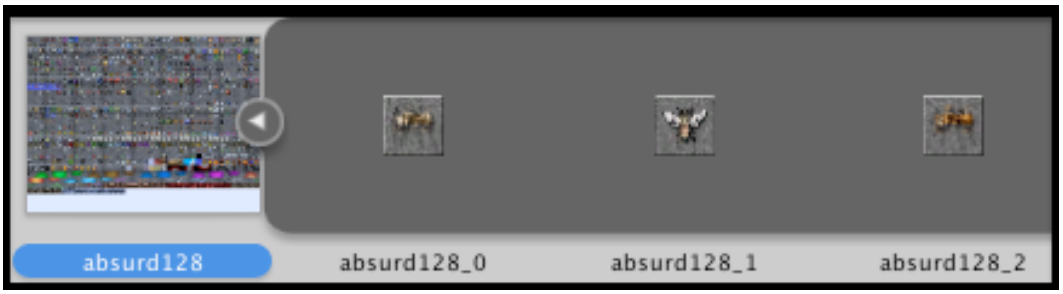
To load game data from a text file map, do the following:

1. Import the text file `level1.txt` and image file `absurd128.png`.
2. Select `absurd128.png` in the **Inspector**, and set **Texture Type** to **Sprite (2D/uGUI)**, and **Sprite Mode** to **Multiple**.
3. Edit this sprite in the **Sprite Editor**, choosing **Type** as **Grid** and **Pixel Size** as 128x128, and Apply these settings.



Insert image 1362OT_10_12.png

4. In the **Project** panel, click the right facing white triangle to "explode" the icon to show all the sprites in this sprite sheet individually.



Insert image 1362OT_10_13.png

5. Drag Sprite absurd128_175 into the scene.
6. Create a new **Prefab** named copse_175 in the **Project** panel, and drag into this blank prefab Sprite absurd128_175 from the scene. Now delete the sprite instance from the scene. You have now created a prefab containing the sprite 175.
7. Repeat this process for the following sprites (i.e. create prefabs for each one):

- floor_848
- corridor_849
- horiz_1034
- vert_1025
- door_844
- potion_675
- chest_586
- alter_583
- stairs_up_994
- stairs_down_993
- wizard_287

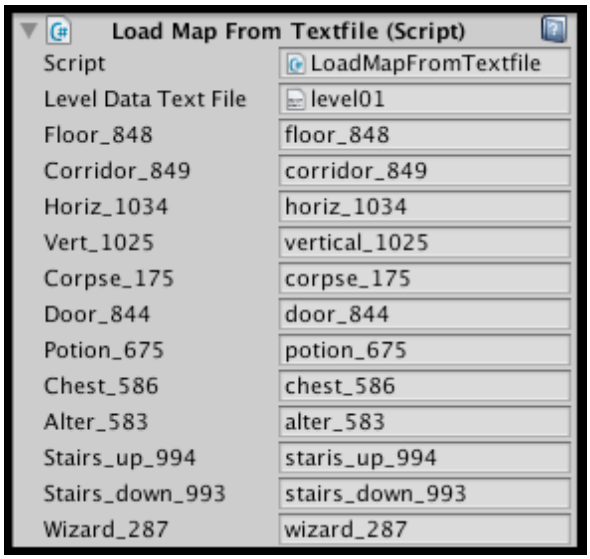
8. Select the **Main Camera** in the **Inspector**, and ensure it is set to an Orthographic camera, sized 20, with **Clear Flags** as **Solid Color** and Background Black.
9. Attach the following C# code to the **Main Camera** as script class `LoadMapFromTextfile`:

```
using UnityEngine;
using System.Collections;

using System.Collections.Generic;

public class LoadMapFromTextfile : MonoBehaviour
{
   public TextAsset levelDataTextFile;

   public GameObject floor_848;
   public GameObject corridor_849;
   public GameObject horiz_1034;
   public GameObject vert_1025;
   public GameObject corpse_175;
   public GameObject door_844;
   public GameObject potion_675;
   public GameObject chest_586;
   public GameObject alter_583;
   public GameObject stairs_up_994;
   public GameObject stairs_down_993;
   public GameObject wizard_287;

   public Dictionary<char, GameObject> dictionary = new
Dictionary<char, GameObject>();
```

```
   void Awake(){
      char newlineChar = '\n';

      dictionary['.'] = floor_848;
      dictionary['#'] = corridor_849;
      dictionary['('] = chest_586;
      dictionary['!'] = potion_675;
      dictionary['_'] = alter_583;
      dictionary['>'] = stairs_down_993;
      dictionary['<'] = stairs_up_994;
      dictionary['-'] = horiz_1034;
      dictionary['|'] = vert_1025;
      dictionary['+'] = door_844;
      dictionary['%'] = corpse_175;
      dictionary['@'] = wizard_287;

      string[] stringArray =
levelDataTextFile.text.Split(newlineChar);
      BuildMaze( stringArray );
   }

   private void BuildMaze(string[] stringArray){
      int numRows = stringArray.Length;

      float yOffset = (numRows / 2);

      for(int row=0; row < numRows; row++){
         string currentRowString = stringArray[row];
         float y = -1 * (row - yOffset);
         CreateRow(currentRowString, y);
      }
   }

   private void CreateRow(string currentRowString, float y) {
      int numChars = currentRowString.Length;
      float xOffset = (numChars/2);

      for(int charPos = 0; charPos < numChars; charPos++){
         float x = (charPos - xOffset);
         char prefabCharacter = currentRowString[charPos];

         if (dictionary.ContainsKey(prefabCharacter)){
            CreatePrefabInstance( dictionary[prefabCharacter], x,
y);
```

```
            }
        }
    }

    private void CreatePrefabInstance(GameObject objectPrefab,
float x, float y){
        float z = 0;
        Vector3 position = new Vector3(x, y, z);
        Quaternion noRotation = Quaternion.identity;
        Instantiate (objectPrefab, position, noRotation);
    }
}
```

10. With the **Main Camera** selected, drag the appropriate prefabs into the prefabs slots in the **Inspector** for the `LoadMapFromTextfile` Script component.



Insert image 1362OT_10_14.png

11. When you run the scene, you should see a sprite-based Nethack map appears, using your prefabs.

How it works...

The sprite sheet was automatically sliced up into hundreds of 128x128 pixels sprites square. We created prefab objects from some of these sprites, so that copies can be created at run time when needed.

Text file `level1.txt` contains lines of text characters. Each non-space character represents where a sprite prefab should be instantiated (column = X, row = Y). A C# `Dictionary` variable named `dictionary` is declared, and initialized in method `Start()`, to associate specific prefab **GameObjects** with particular characters in the text file.

Method `Awake()` splits the string into an array using the newline character as a separator. So now we have `stringArray`, with an entry for each row of text data. Method `BuildMase(…)` is called with the `stringArray`.

Method `BuildMaze(…)` interrogates the array to find its length (the number of rows of data for this level), and sets `yOffSet` to half this value. This is to allow the placing of the prefabs half above Y=0 and half below, so (0,0,0) is the center of the level map. A FOR-loop is used to read each row's string from the array, and passes it to method `CreateRow(…)`, along with the Y-value corresponding to the current row.

Method `CreateRow(…)` extracts the length of the string, and sets `xOffSet` to half this value. This is to allow the placing of the prefabs half to the left of X=0 and half to the right, so (0,0,0) is the center of the level map. A FOR-loop is used to read each character from the current row's string, and (if there is an entry in our dictionary for that character) then method `CreatePrefabIInstance (…)` is called, passing the prefab reference in the dictionary for that character, and the X and Y value.

Method `CreatePrefabInstance(…)` instantiates the given prefab, at a position of (X, Y, Z), where Z is always zero, and no rotation (`Quarternion.identity`).

Managing Unity project code using Git version control and GitHub hosting

Distributed Version Control Systems (**DVCS**) are becoming a bread-and-butter everyday tool for software developers. An issue with Unity projects can be the many binary files in each project. There are also many files in a local system's Unity project directory that are not needed for archiving/sharing, such as OS specific thumbnail files, trash files, and so on. Finally, some Unity project folders themselves do not need to be archived, such as Temp and Library.

While Unity provided its own "Asset Server", many small game developers chose not to pay for this extra feature. Also, GIT and Mercurial (the most common DVCSs) are free and work with any set of documents to be maintained (programs in any programming language, text-files, and so on). So it makes sense to learn how to work with a third-party, industry standard DVCS for Unity projects. In fact, the code and diagrams and documents for this very book were all archived and version controlled using a private GitHub repository!

In this recipe, we will setup a Unity project for GIT DVCS, through a combination of Unity Application settings and use of the GitHub GUI-client application.

> We created a real project this way - a pacman-style game, which you can explore and download/pull at the following public GitHub URL:
>
> https://github.com/dr-matt-smith/matt-mac-man

Getting ready

This recipe can be used with any Unity project. In the folder `1362_10_09`, we have provided a Unity package of our matt-mac-man game, if you wish to use that one - in which case create a new 2D project in Unity, and import this package.

Since this recipe illustrates hosting code on GitHub, you'll need to create a (free) GitHub account at `github.com` if you do not already have one.

Before starting this recipe you need to have installed Git and the GitHub client application.

Learn how, and download the client from the following links:

- http://git-scm.com/book/en/Getting-Started-Installing-Git
- http://git-scm.com/downloads/guis

How to do it...

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

1. In the root directory of your Unity project, add the following code into a file named .**gitignore** (ensure the filename starts with the DOT):

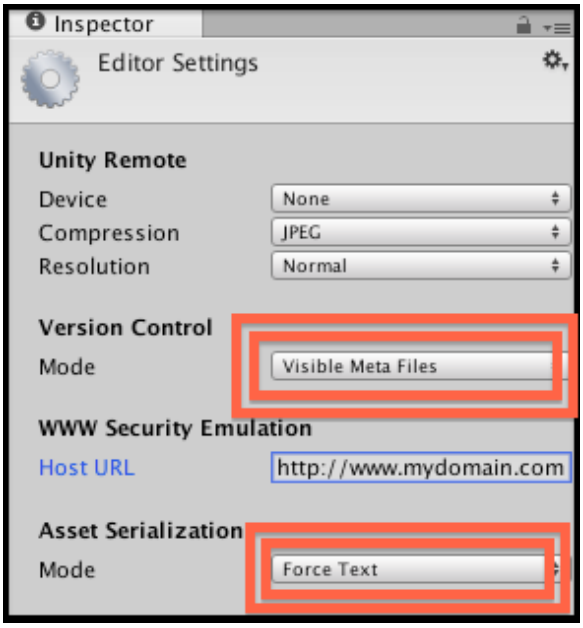
```
# =============== #
# Unity generated #
# =============== #
Temp/
Library/

# ================================== #
# Visual Studio / MonoDevelop generated #
# ================================== #
ExportedObj/
obj/
*.svd
*.userprefs
/*.csproj
```

```
*.pidb
*.suo
/*.sln
*.user
*.unityproj
*.booproj

# ============ #
# OS generated #
# ============ #
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

NOTE: This special file (.gitignore) tells the version control system which files do NOT need to be archived. For example, we don't' need to record the Windows or Mac image thumbnail files (.DS_STORE or Thumbs.db).

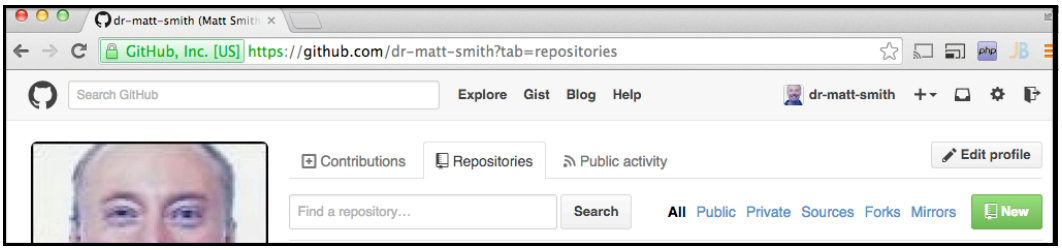
---

2. Open the **Editor Settings** in the **Inspector** by choosing menu: **Edit | Project Settings | Editor**
3. In the **Editor Settings**, set the **Version Control Mode** to **Visible Meta Files**.
4. In the **Editor Settings**, set the **Asset Serialization Mode** to **Force Text**.



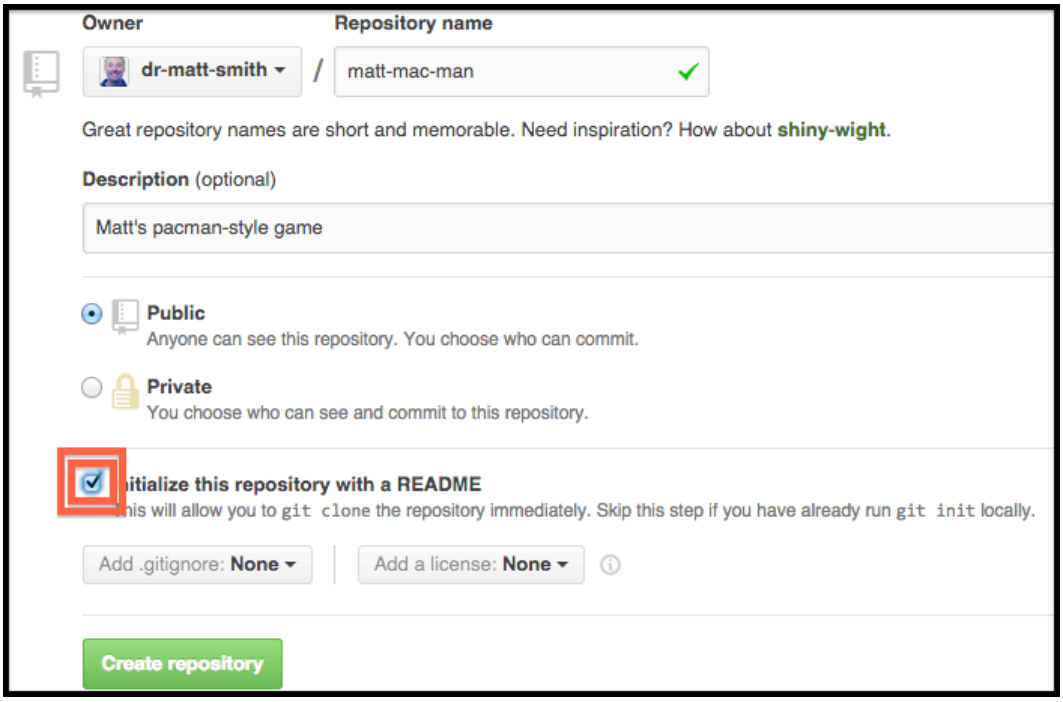
Insert image 1362OT_10_15.png

5. Save your project, so these new settings are stored. Then close the Unity application.
6. Log on to your GitHub account.
7. On your GitHub home page click the green **New** button to start creating a new repository.



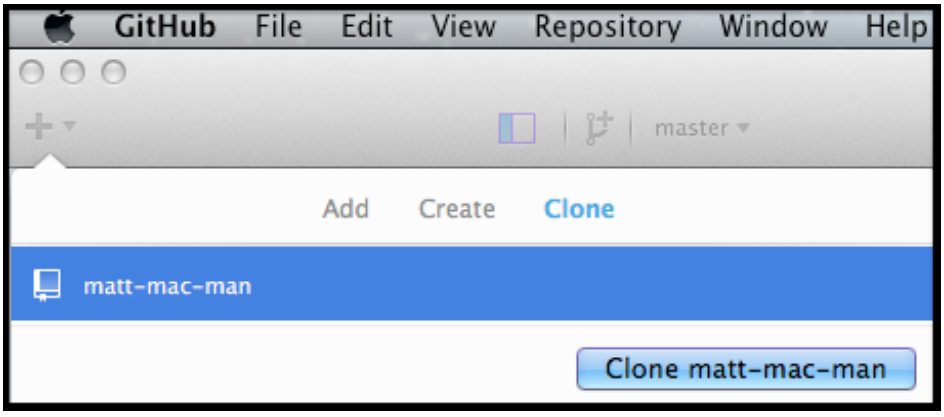
Insert image 1362OT_10_16.png

8. Give your new repository a name (we chose “matt-mac-man”) and check the option **Initialize this repository with a README**.



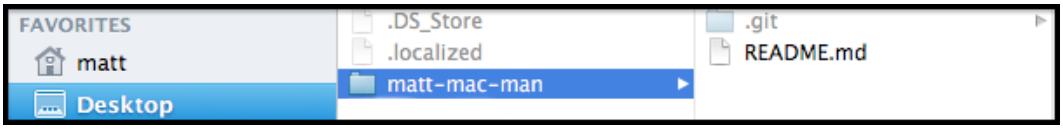
Insert image 1362OT_10_17.png

9. Startup your GitHub client application on your computer, and get a list of repositories to clone to the local computer by choosing menu: **File | Clone Repository ...**. From the list provided, select your new repository (for us, it was matt-mac-man) and click the button to **Clone** this repository.



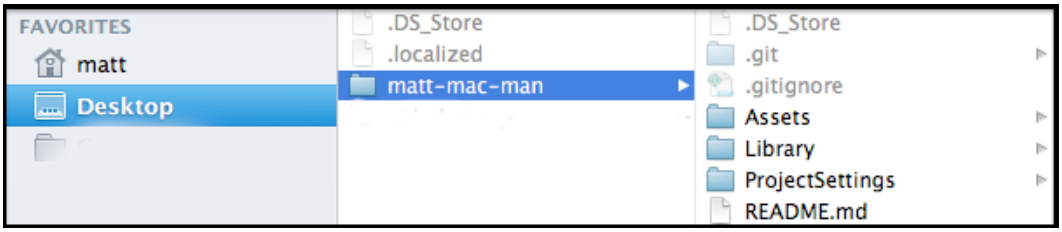
Insert image 1362OT_10_18.png

10. You'll be asked where to store this repository on your local computer (we simply chose our **Desktop**). You should now see a folder with the repository name on your computer's disk, containing a hidden .git folder, and a single file named README.md.



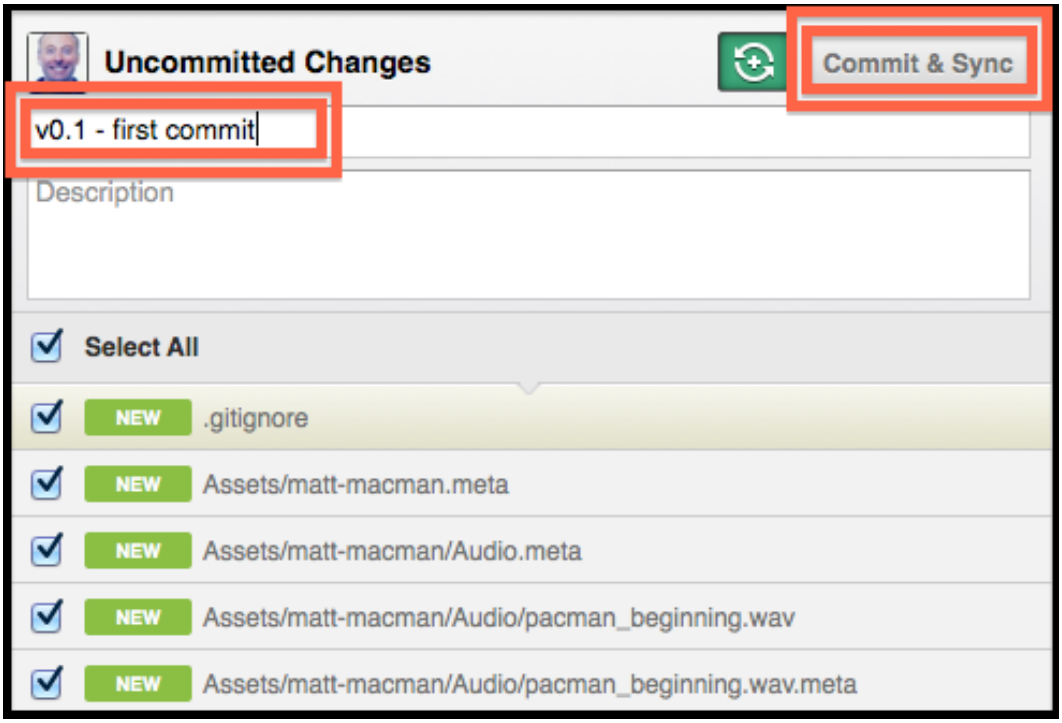
Insert image 1362OT_10_19.png

11. Now, copy into this local repository folder the following files and folders from your Unity project:
    - `.gitignore`
    - `/Assets`
    - `/Library`
    - `/ProjectSettings`



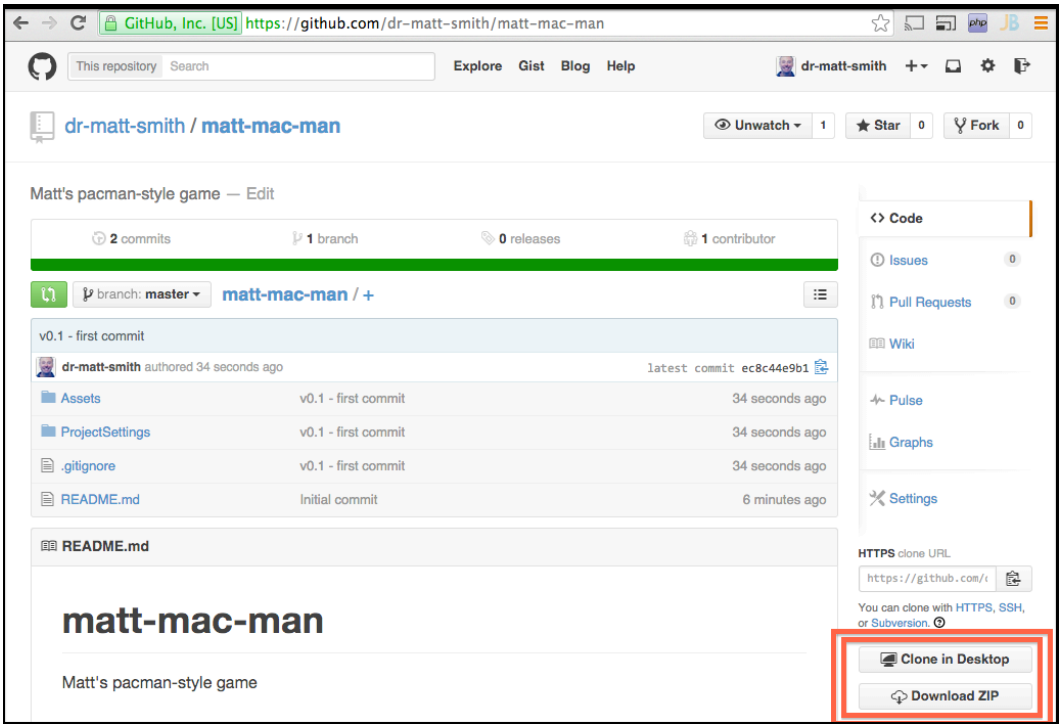
Insert image 1362OT_10_20.png

12. In your GitHub client application, you should now see lots of **Uncommitted Changes**. Type in a short comment for your first commit (we typed our standard: "v0.1 – first commit"), and click the **Commit & Sync**, to PUSH the contents of this Unity project folder up to your GitHub account repository.



Insert image 1362OT_10_21.png

13. Now, if you visit your GitHub project page, you should see all these Unity project files available to download for people's computers, either as a ZIP archive or be "cloned" using a Git client.



Insert image 1362OT_10_22.png

How it works...

The special file `.gitngnore` lists all the files and directories that are NOT to be archived.

Changing Unity **Editor Settings** for **Version Control Mode** to **Meta Files** ensures Unity stores required "housekeeping" data for each asset in its associated meta file. Selecting **Visible** rather than **Hidden** simply avoids any confusion as to whether GIT will record the meta files or not - GIT will record them whether visible or not, so by making them visible, it is obvious to developers working with the files that they will be included.

Changing Unity **Editor Settings** for **Asset Serialization Mode** to **Force Text** attempts to solve some of the difficulties of managing changes with large binary files. Unity projects tend to have quite a few binary files, such as `.unity` scene files, and prefabs, and so on. There seems to be some debate about the best setting to use; we have found that **Force Text** works fine and so use that at present.

You'll see 2 commits on GitHub, since the very first was when we created the new repository, and the second was our first commit of the repository using the GitHub client,

when we added all of our code into the local repository and pushed (committed) it to the remote server.

There's more...

Some details you don't want to miss:

Learn more about Distributed Version Control Systems (DVCS)

The following video link is a short introduction to Distributed Version Control Systems:

```
http://youtu.be/1BbK9o5fQD4
```

Note that the Fogcreek Kiln "harmony" feature now allows seamless work between GIT and Mercurial with the same Kiln repository:

```
http://blog.fogcreek.com/kiln-harmony-internals-the-basics/
```

Using Bitbucket and SourceTree

If you prefer to use Bitbucket and SourceTree with your Unity projects, you can find a good tutorial at the following URL:

```
http://yeticrabgames.blogspot.ie/2014/02/using-git-with-unity-without-using.html
```

Using the command line rather than Git-client application

While for many, using a GUI client such as the GitHub application is a gentler introduction to using DVCS, at some point, you'll want to learn more and get to grips with working in the command line.

Since both Git and Mercurial are open source, there is lots of great, free online resources available. The following are good sources to get started on:

- Learn all about Git, download free GUI clients, and even get free online access to The Pro Git book (by Scott Chacon) available through Creative Commons license at the following URL:
  ```
  http://git-scm.com/book
  ```
- An online interactive Git command line to practice in:
  ```
  https://try.github.io/levels/1/challenges/1
  ```
- The main Mercurial website, Including free online access to the book *Mercurial: The Definitive Guide* (by *Bryan O'Sullivan*) available through the Open Publication License:
  ```
  http://mercurial.selenic.com/
  ```
- SourceTree - a free Mercurial and Git GUI client:
  ```
  http://www.sourcetreeapp.com/
  ```

See also

Refer to the following recipe for more Information:

- *Publishing for multiple devices via Unity Cloud*

Publishing for multiple devices via Unity Cloud

One reason for the GIT recipe in this chapter is to allow you to prepare your projects for one of the most exciting new services offered to Unity developers in recent years: Unity Cloud! Unity Cloud takes all the work out of building different versions of your project for different devices - you PUSH your updated Unity project to your online DVCS (such as GitHub), then Unity Cloud will see the update and PULL your new code and build your game for the range of devices/deployment platforms you have set up.

Getting ready

First, log on to the Unity Cloud Build website and create an account:

```
http://unity3d.com/unity/cloud-build
```

For this recipe, you need access to a project's source code. If you haven't your own (for example, you haven't completed the Git recipe in this chapter), then feel free to use the matt-mac-man project available at the following public GitHub URL:

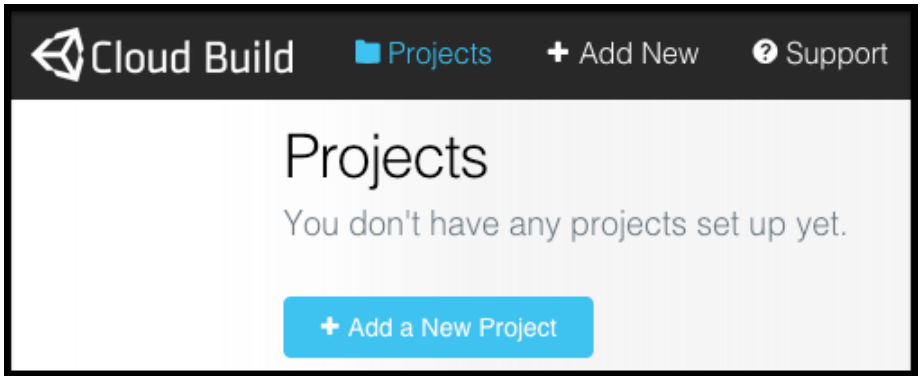
```
https://github.com/dr-matt-smith/matt-mac-man
```

> NOTE: A common reason for a test project first build to fail is forgetting to add at least one scene to the build settings for the project.

How to do it...

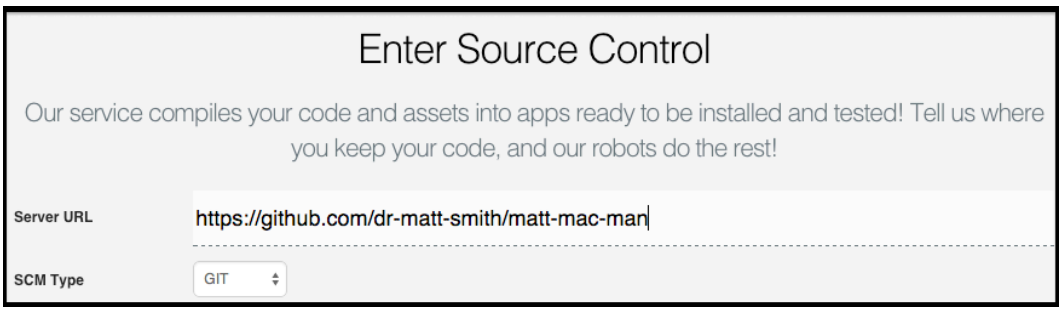
To load external resources by Unity Default Resources, do the following:

1. Log on to your Unity Cloud Build account
2. On the Project's page, click the button **Add a New Project**.



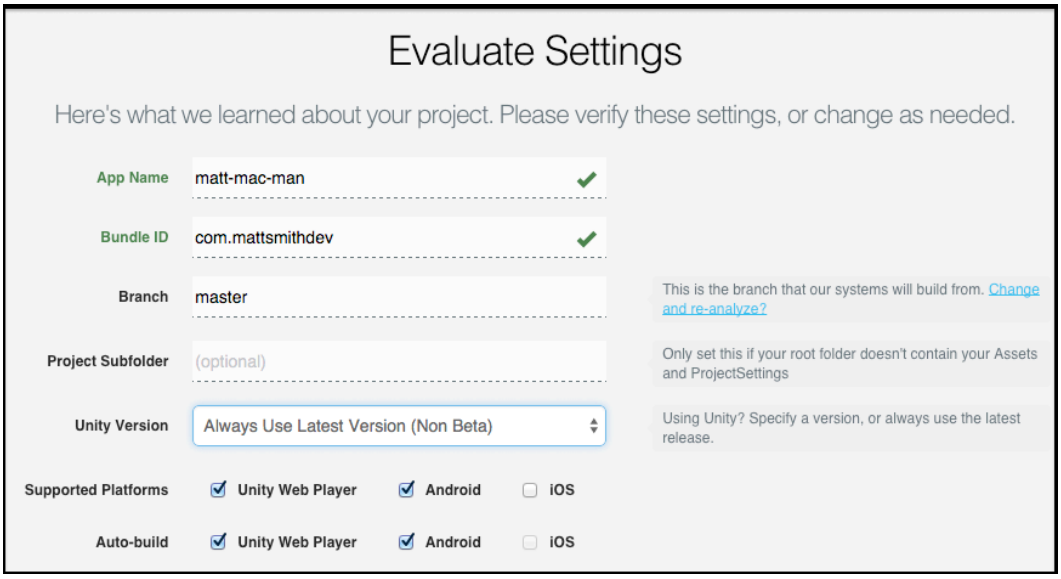
Insert image 1362OT_10_23.png

3. Next, you'll need to add the URL for your source code, and the **Source Control Method** (**SCM**). For our project, we entered our matt-mac-man URL and **GIT** for the SCM.



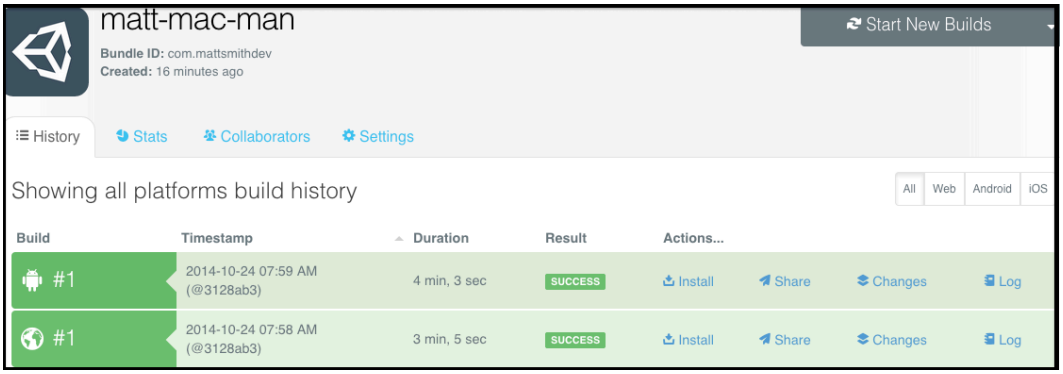
Insert image 1362OT_10_24.png

4. Next, you need to enter some settings. Unity Cloud Build will choose your source code project name as the default application name (most times this is fine). You need to enter a **Bundle ID** - commonly, the reverse of your website URL is used here to ensure that the **App Name** plus **Bundle ID** is unique. So, we entered `com.mattsmithdev`. Unless testing "branches" of the code, the default master branch is fine, and likewise, unless testing sub-folders, the default (no subfolder) is fine. Unless you are using the latest "beta" versions, the **Unity Version** option should be left to the default **Always Use Latest Version**. Finally, check the build options you wish to have created. Note, you'll need to have setup the Apple codes if building for IOS; but you will be able to build for Unity Web Player and Android immediately.



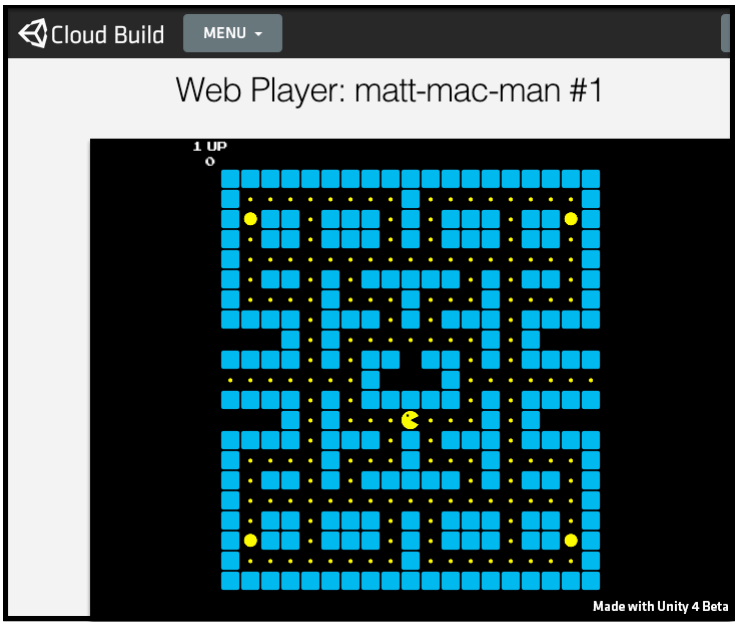
Insert image 1362OT_10_25.png

5. Next are the app “credentials”. Unless you have Android credentials, you can choose the default ”development” credentials. But this means users will be warned when installing the application.
6. Unity Cloud will then start to build your application - this will take a few minutes (depending on the load on their server).
7. When built, you’ll get an email (for each deployment target - so we got one for Web Player and one for Android). If the build fails, you’ll still get an email, and you can look up in the “logs” for the reasons the build fail.



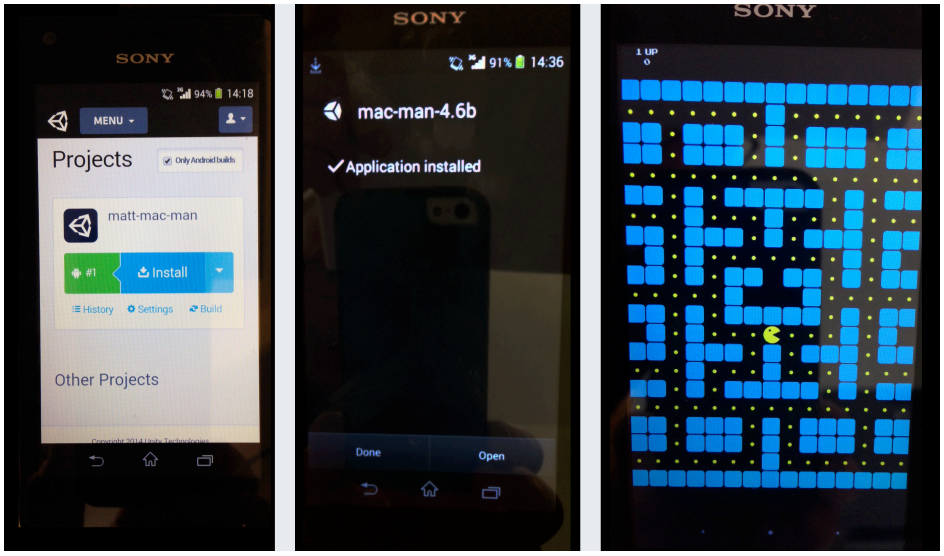
Insert image 1362OT_10_26.png

8. You can then play web player version immediately:



Insert image 1362OT_10_27.png

9. To test with Android or iOS, you download it onto the device (from the Unity Cloud web server) and play the game.



Insert image 1362OT_10_28.png

How it works...

Unity Cloud *pulls* your project source code from the DVCS system (such as GitHub). It then compiles your code, using the settings chosen for Unity version and deployment platforms (we chose Web Player and Android in this recipe). If the build is successful, Unity Cloud makes the build applications available to download and run.

There's more...

Some details you don't want to miss:

Learn more about Unity Cloud

Learn more in the Support section of the Unity Cloud website, and the Unity main website Cloud Build information page:

- `https://build.cloud.unity3d.com/support/`
- `http://unity3d.com/unity/cloud-build`

See also

Refer to the following recipe for more Information:

Managing Unity project code using Git version control and GitHub hosting