1
Bonus Chapter 16 : Virtual Reality and Extra Features

In this chapter, we will cover the following topics:

- Saving screenshots from the game
- Saving and loading player data – using static properties
- Saving and loading player data – using PlayerPrefs
- Loading game data from a text file map
- UI Slider to change game quality settings
- Pausing the game
- Implementing slow motion
- Gizmo to show the currently selected object in the scene panel
- Editor snap-to grid drawn by Gizmo
- Creating a VR project
- Adding 360-degree videos to a VR project
- Editing VR content inside a VR environment – the XR Editor

Introduction

There are too many features in Unity 2018 to all be covered in a single book. In this chapter, we will present a set of recipes illustrating VR game development in Unity, plus a range of additional Unity features that we wanted to include.

The Big picture

Apart from plain text, there are three sections below will give you an idea of what this chapter is about.

Virtual reality

VR is about presenting to the player an immersive audio-visual experience, engaging enough for them to lose themselves in exploring and interacting with the game world that has been created.

From one point of view, VR simply requires two cameras in order to generate the images for each eye to give that 3D effect. But effective VR needs content, UI controls, and tools to help create them. In this chapter, we will explore recipes that work with 360-degree videos, and Unity's XR Editor toolset.

Gizmos

Gizmos are another kind of Unity editor customization. Gizmos are visual aids for game designers that are provided in the scene panel. They can be useful as setup aids (to help us know what we are doing), or for debugging (understanding why objects aren't behaving as expected).

Gizmos are not drawn through Editor scripts, but as part of Monobehaviours, so they only work for GameObjects in the current scene. Gizmo drawing is usually performed in two methods:

- `OnDrawGizmos()`: This is executed every frame, for every GameObject in the hierarchy
- `OnDrawGizmosSelect()`: This is executed every frame, for just the/those GameObject(s) that are currently selected in the hierarchy

Gizmo graphical drawing makes it simple to draw lines, cubes, and spheres. More complex shapes can also be drawn with meshes, and you can also display 2D image icons (located in the Project folder: `Assets | Gizmos`).

Several recipes in this chapter illustrate how Gizmos can be useful. Often, new GameObjects created from Editor Extensions will have helpful Gizmos associated with them.

Saving/Loading data at runtime

When loading/saving data either locally, 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. When using the **PlayerPrefs** class, we are limited to saving and loading integers, floats, and strings. We provide several recipes illustrating ways to save and load data at Run-Time, including the use of static variables, PlayerPrefs, and a public TextAsset containing text-format data for a 2D game level.

Saving screenshots from the game

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

This technique only works in the Unity **Editor**, or when you build to a **standalone** Windows or Mac executable (it will not work for Web Player builds, for example):

Getting ready

To follow this recipe, please import the screenshots package, which is available in the `16_01` folder, to your project. The package includes a main camera, cube, and sphere—something to be able to recognize when the screenshot is taken!

How to do it...

To save the screenshots from your game, follow these steps:

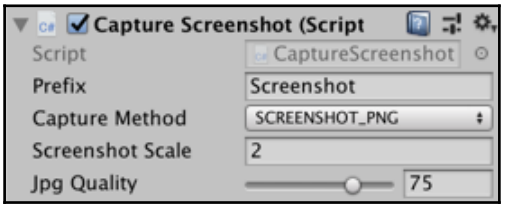
1. Create a new 3D project.
2. Import the screenshots package by choosing the following menu: **`Assets | Import Package | Custom Package...`**
3. Open the scene provided in the screenshots package.
4. Create a C# `CaptureScreenshot` script class and add an instance object as a component to the **Main Camera**:

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

 public class CaptureScreenshot : MonoBehaviour {
     public string prefix = "Screenshot";
     public enum CaptureMethod {
         SCREENSHOT_PNG,
         READ_PIXELS_PNG,
         READ_PIXELS_JPG
     };
     public CaptureMethod captureMethod =
CaptureMethod.SCREENSHOT_PNG;
     public int screenshotScale = 1;

     // A slider from 0 to 100 from which to set JPG quality
     [Range(0, 100)]
     public int jpgQuality = 75;

     private Texture2D texture;
     string date;

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

     private void TakeShot() {
         date = System.DateTime.Now.ToString("_d-MMM-yyyy-HH-
mm-ss-f");

         if (CaptureMethod.SCREENSHOT_PNG == captureMethod){
             string fileExtension = ".png";
             string filename = prefix + date + fileExtension;
```

```
            ScreenCapture.CaptureScreenshot(filename,
screenshotScale);
        } else {
            StartCoroutine(ReadPixels());
        }
    }

    IEnumerator  ReadPixels () {
        byte[] bytes;
        yield return new WaitForEndOfFrame();

        int screenWidth = Screen.width;
        int screenHeight = Screen.height;
        Rect screenRectangle = new Rect(0, 0, screenWidth,
screenHeight);
        texture = new Texture2D (screenWidth, screenHeight,
TextureFormat.RGB24, false);
        texture.ReadPixels(screenRectangle, 0, 0);
        texture.Apply();

        switch(captureMethod){
            case CaptureMethod.READ_PIXELS_JPG:
                bytes = texture.EncodeToJPG(jpgQuality);
                WriteBytesToFile(bytes, ".jpg");
                break;

            case CaptureMethod.READ_PIXELS_PNG:
            default:
                bytes = texture.EncodeToPNG();
                WriteBytesToFile(bytes, ".png");
                break;
        }
    }

    void WriteBytesToFile(byte[] bytes, string fileExtension)
{
        Destroy (texture);
        string filename = prefix + date + fileExtension;
        string path = Application.dataPath;
        path = Path.Combine(path, "..");
        path = Path.Combine(path, filename);
        File.WriteAllBytes(path, bytes);
    }
 }
```

5. With the **Main Camera** selected in the **Hierarchy**, in the **Inspector** access the **CaptureScreenshot (Scripted)** component. Set **Capture Method** as `SCREENSHOT_PNG`. Change **Screenshot Scale** to `2`.

   If you want your image file's name to start with something other than the word **Screenshot**, then change it in the **Prefix** field:



6. 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...

For each frame the `Update()` methods tests whether the *P* key has been pressed. If pressed, the `TakeShot()` method is invoked.

Method `TakeShot()` captures an image of the screen is and stores the image as a file in the main Unity project directory (that is, alongside the **Assets** and **Library** directories, and so on). The public settings of the **CaptureScreenshot (Scripted)** of **MainCamera** determine the properties of the image file created by the project.

The three types of screenshot image are defined as an **enumerated type**:

- `SCREENSHOT_PNG`: built-in Unity function `CaptureScreenshot()`. This is capable of scaling the original screen size, which can be set by our Capture Scale public property.
- `READ_PIXELS_PNG`: `ReadPixels()` function used, encoded to PNG.
- `READ_PIXELS_JPG`: `ReadPixels()` function used, encoded to JPG.

The image data captured by the ReadPixels function is written to file by the built-in Unity function `WriteAllBytes()` in our `WriteBytesToFile(...)` method.

In all cases, the file created will have the appropriate *.png* or *.jpg* file extension, to match its image file format.

There's more...

We have included the options using the `ReadPixel` function as a demonstration of how to save your images to a disk without using Unity's `CaptureScreenshot()` function. One advantage of this method is that it can be adapted to capture and save only a portion of the screen—if a different-sized rectangle is defined.

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 feeling of depth and content. In this recipe, we will learn how to make our game remember the player's score between the different levels (scenes).

Getting ready

We have included a complete project in a Unity package named **game_HigherOrLower** in the `16_02` folder. To follow this recipe, we will import this package as the starting point.

How to do it...

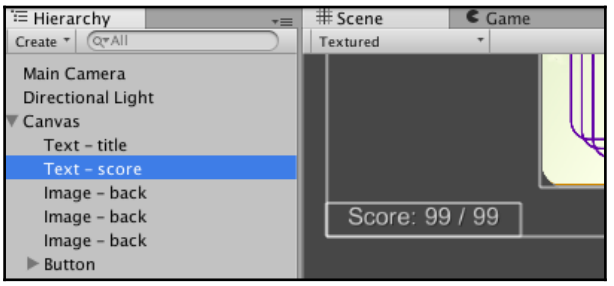
To save and load player data, follow these steps:

1. Create a new 2D project and import the **game_HigherOrLower** package.
2. Add each of the scenes to the build in the 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 content of the scenes. The game starts on the **scene0_mainMenu** scene, inside the `Scenes` folder.

4. Let's create a class to store the number of correct and incorrect guesses made by the user. Create a new C# script called `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 the **scene0_mainMenu** scene, create a **UI Text GameObject** named **Text - score**, containing the placeholder text **Score: 99 / 99**:



6. Next, attach the following C# script to the **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 the `scene2_gameWon` scene, attach the following C# script to the **Main Camera**:

```
using UnityEngine;

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

8. In the `scene3_gameLost` scene, 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 will find that the score and the 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 the static properties are remembered from scene to scene. This class also provides the public static method called `ZeroTotals()` that resets both the values to zero.

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

When the game is running and the user guesses correctly (higher), then the **scene2_gameWon** scene 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 the `Player` class.

When the game is running and the user guesses wrongly (lower), then the 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 the `Player` class.

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

There's more...

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

Hiding the score before the first attempt is 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;
 }
```

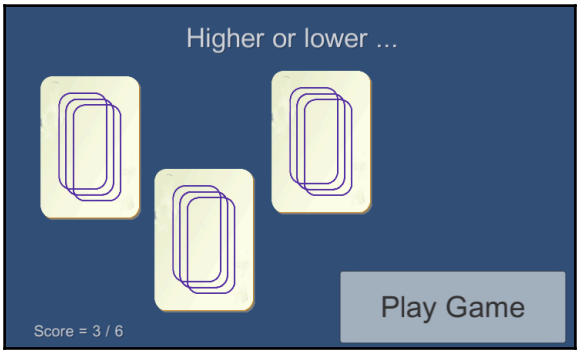
See also

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 the static properties allow a game to remember values between different scenes, these values are forgotten once the game application has quit. Unity provides the `PlayerPrefs` feature to allow a game to store and retrieve data, between the different game-playing sessions:



Getting ready

This recipe builds upon the previous recipe, so make a copy of that project and work on the copy.

How to do it...

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

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

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

4. Now edit the C# script called `IncrementIncorrectScore` by replacing the `Start()` method 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 will find that the player's name, level, and score are now kept between the game sessions.

How it works...

We had no need for the Player class, since this recipe uses the built-in runtime class called `PlayerPrefs`, provided by Unity.

Unity's `PlayerPrefs` runtime class is capable of storing and accessing information (the 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 the `IncrementCorrectScore` and `IncrementIncorrectScore` classes. These methods use the `PlayerPrefs.GetInt("")` method to retrieve the old total, add 1 to it, and then store the incremented total using the `PlayerPrefs.SetInt("")` method.

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

For more information on `PlayerPrefs`, see Unity's online documentation at `http://docs.unity3d.com/ScriptReference/PlayerPrefs.html`.

See also

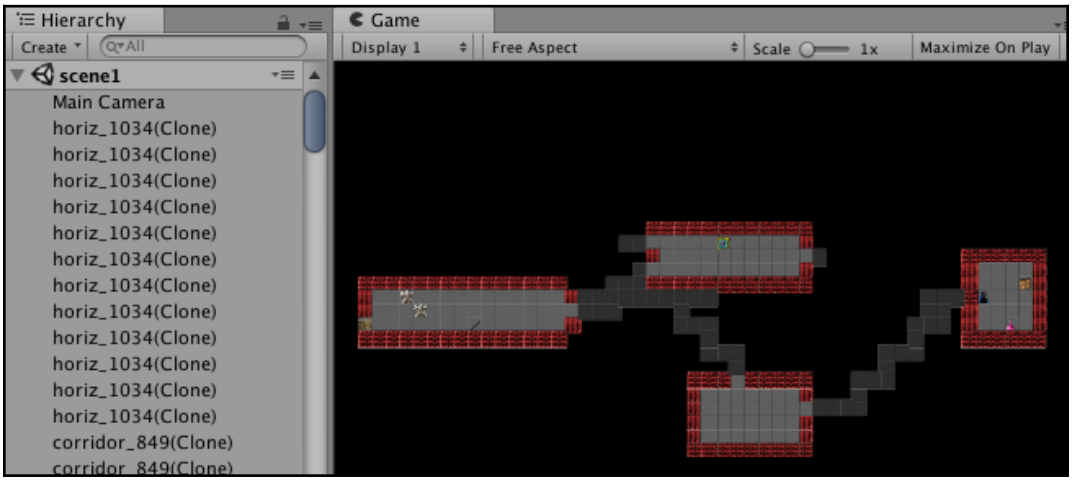
Refer to the following recipe in this chapter for more information:

- Saving and loading player data - using static properties

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 the text files of rows, and columns of characters, where each character corresponds to the type of **GameObject** that is 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*:



Getting ready

In the `16_04` folder, we have provided the following two files for this recipe:

- `level1.txt` (a text file, representing a level)
- `absurd128.png` (a 128 x 128 sprite sheet for NetHack).

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

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

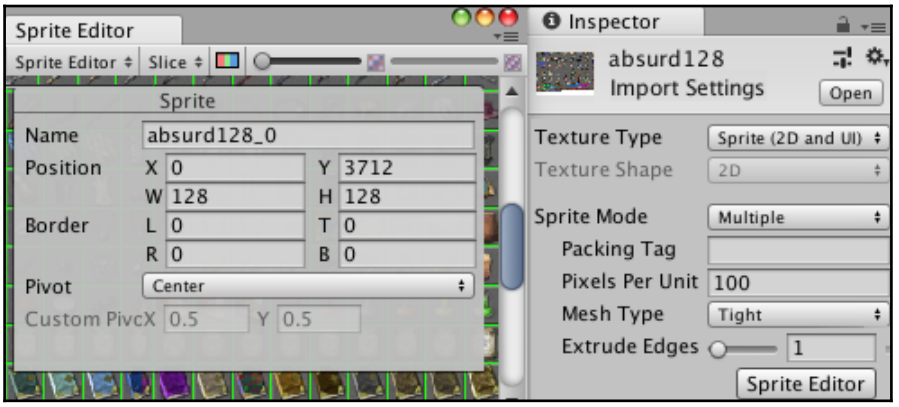
Note that we also included a Unity package with all the prefabs set up, since this can be a laborious task.

How to do it...

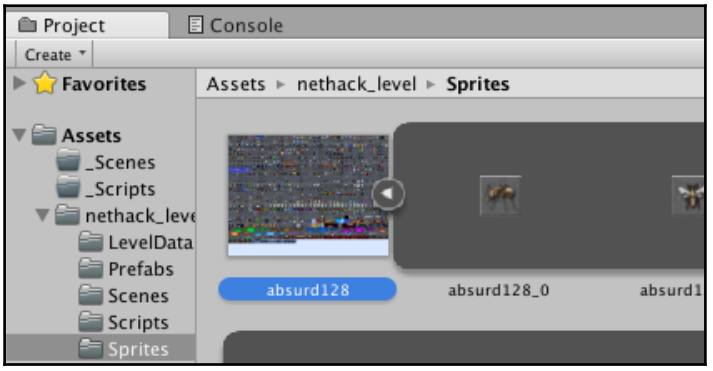
To load game data from a text file map, do the following:

1. Import 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 **128 x 128**, and apply these settings:



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



5. Drag the **Sprite** called **absurd128_175** on to the scene.
6. Create a new **Prefab** named `corpse_175` in the Project panel, and drag onto 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 (that is, 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 that it is set to an Orthographic camera, sized **20**, with **Clear Flags** as **Solid Color** and **Background** as **Black**.
9. Attach the following C# code to the **Main Camera** as the script class called `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 on to the prefabs slots in the **Inspector**, for the `LoadMapFromTextfile` Script component.
11. When you run the scene, you will see that a sprite-based Nethack map will appear, using your prefabs.

How it works...

The Sprite sheet was automatically sliced up into hundreds of 128 x 128 pixel Sprite squares. We created the prefab objects from some of these sprites so that the copies can be created at runtime when needed.

The text file called `level1.txt` contains the 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 the `Start()` method to associate the specific prefab **GameObjects** with some particular characters in the text file.

The `Awake()` method 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 the text data. The `BuildMaze(...)` method is called with the `stringArray`.

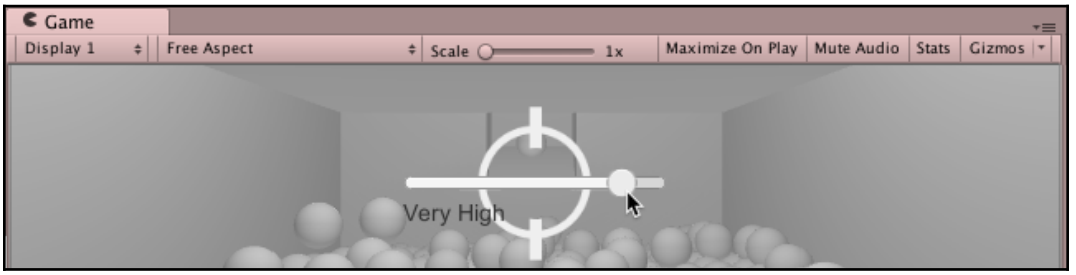
The `BuildMaze(...)` method interrogates the array to find its length (the number of rows of data for this level), and sets `yOffSet` to half this value. This is done to allow the placing of the prefabs half above *y* = 0, and half below, so (0,0,0) is the center of the level map. A for-loop is used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

The `CreateRow(...)` method extracts the length of the string, and sets `xOffSet` to half this value. This is done 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 the `CreatePrefabInstance (...)` method is called, passing the prefab reference in the dictionary for that character, and the x and y value.

The CreatePrefabInstance(...) method instantiates the given prefab at a position of x, y, z, where z is always zero, and there is no rotation (`Quarternion.identity`).

UI Slider to change game quality settings

In this recipe, we will show you how the player can control the quality settings by providing a **UI Slider** for the player. From this, they can choose from an array of possible quality settings for the current project:



Getting ready

For this recipe, we have prepared a package named `BallGame` containing two scenes. The package is in the `16_05` folder.

How to do it...

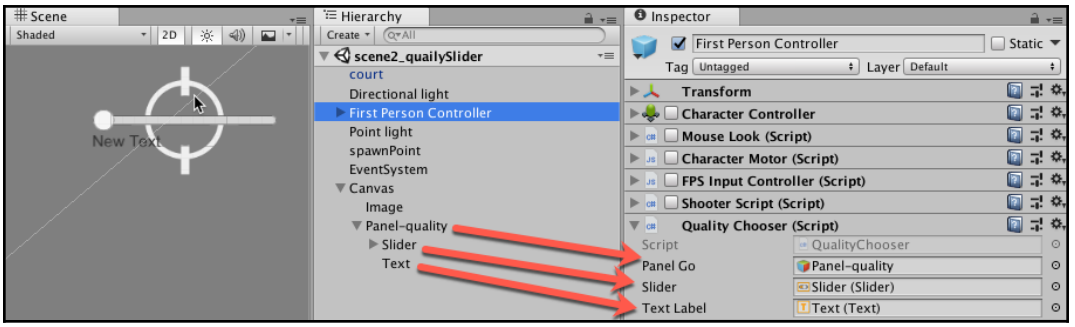
To create a player UI to change the game's quality settings, do the following:

1. Create a new 3D project and import the `BallGame` package.
2. Open the scene named `scene0_ballCourt`.

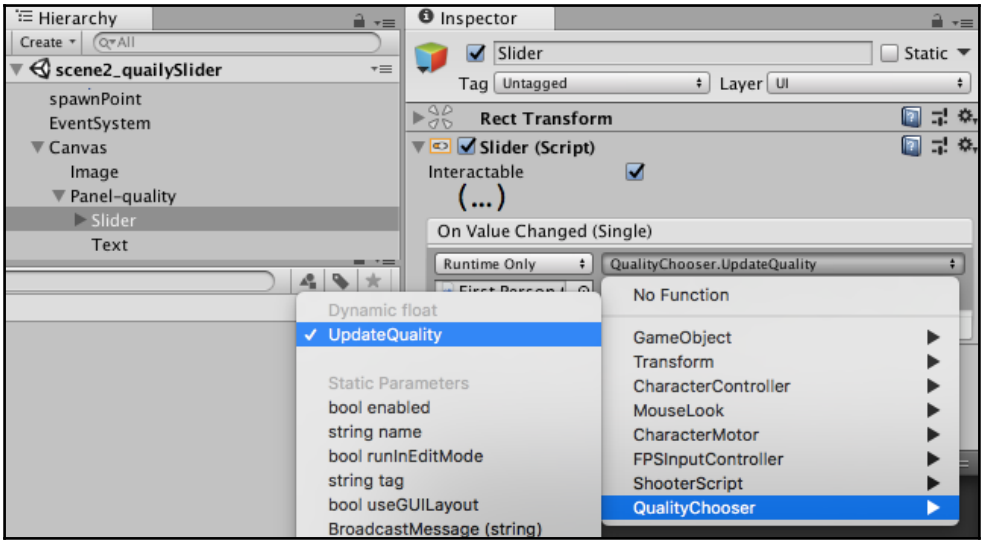
3. In the scene, create a new **UI Panel** named `Panel-quality` by choosing menu: **Create | UI | Panel**.
4. With the GameObject **Panel-quality** selected in the Hierarchy, create a new UI Slider named **Slider-quality** by choosing menu: **Create | UI | Slider**. This GameObject should be childed to **Panel-quality**.
5. With **Panel-quality** selected in the Hierarchy, create a new UI Text GameObject named Text-quality by choosing menu: **Create | UI | Text**. This GameObject should be childed to GameObject **Panel-quality**. In the **Inspector**, set its Transform Position Y value to **-25**.
6. Create a new C# script class named `QualityChooser`, and attach an instance object as a component to the **First Person Controller**:

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

 public class QualityChooser : MonoBehaviour {
     public GameObject panelGo;
     public Slider slider;
     public Text textLabel;

     void Awake () {
         slider.maxValue = QualitySettings.names.Length - 1;
         slider.value = QualitySettings.GetQualityLevel();
         SetQualitySliderActive(true);
     }

     public void SetQualitySliderActive(bool active) {
         Cursor.visible = active;
         panelGo.SetActive(active);
     }

     public void UpdateQuality(float sliderFloat) {
         int qualityInt = Mathf.RoundToInt (sliderFloat);
         QualitySettings.SetQualityLevel (qualityInt);
         textLabel.text = QualitySettings.names [qualityInt];
     }
 }
```

7. In the **Hierarchy,** select the **First Person Controller**. Then, from the **Inspector**, access the Quality Chooser component, and populate the **panelGo**, Slide, and Text Label public fields with the UI GameObjects **Panel-quality**, **Slider**, and **Text**:



8. From the **Hierarchy** panel, select **Slider**. Then, from the **Inspector,** for the **Slider** component, find the list named **On Value Changed (Single)**, and click on the **+** sign to add a command.
9. Drag the **First Person Controller** from the **Hierarchy** into the **Game Object** field of the new command. Then, use the function selector to find the UpdateQuality function in the Dynamic float section (**No Function | QualityChooser | Dynamic float | UpdateQuality**):



10. When you play the scene, you should be able to drag the quality slider to change the quality settings.

How it works...

You created a panel containing a **UI Slider** and a **UI Text** object.

The `SetQualitySliderActive(...)` method receives a true/false value, and uses this to show/hide the mouse cursor and the **UI Panel**.

The `UpdateQuality(...)` method receives a float value from the Slider `OnChange` event. This value is converted to an integer, which is to be the index of the selected quality setting. This index is used both to select the project quality setting, and also to update the **UI Text** label with the name of the currently selected quality setting.

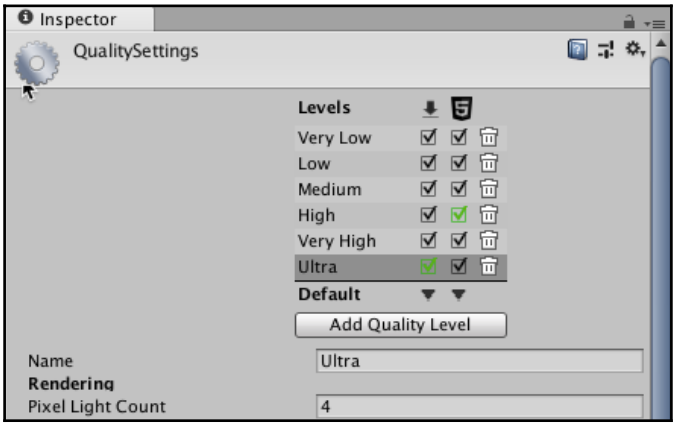
When the **Scene** begins, in the `Awake()` method, the **UI Slider** has its maximum value set to 1 less then the number of project quality items (for example, if there are five items, the slider will be from 0 to 4). Also, the **UI Slider** is moved to the position corresponding to the current quality level, and the `SetQualitySliderActive(...)` method is invoked with the true value in order to display both the mouse pointer and the **UI Panel** showing the slide and text label.

There's more...

Here are some ways to go further with this recipe.

Seeing/editing the list of quality settings

You can view and modify the quality settings available for a project by choosing menu: **Edit** | **Project Settings** | **Quality**:



Pausing the game

As compelling as your next game will be, you should always let players pause it for a short break. Sometimes, a game pause is used to let the player rest, but another reason might be to change some game setting such as volume or graphics quality.

Pausing the game usually involves a combination of freezing a game action, and also hiding or revealing UI items, to display a message to the player and provide UI controls to change settings.

In this recipe, we will implement a simple and effective pause screen that hides the previous recipe's quality settings slide when the game is being played, and reveals it when the game has been paused.

Getting ready

This recipe builds on the previous one, so make a copy of that and use that copy.

How to do it...

To pause your game upon pressing the *Esc* key, follow these steps:

1. Select the **First Person Controller**, and in the **Inspector**, enable the following components:
    - **Character Controller**
    - **Mouse Look (Script)**
    - **Character Motor (Script)**
    - **FPS Input Controller (Script)**
    - **Shooter (Script)**
2. Add the following C# script called `PauseGame` to **First Person Controller**:

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

 public class PauseGame : MonoBehaviour {
     private bool isPaused = false;
     private QualityChooser qualityChooser;

     void Start () {
         qualityChooser = GetComponent<QualityChooser>();
```

```
    }

    void Update () {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            isPaused = !isPaused;
            SetPause ();
        }
    }

    private void SetPause() {
        float timeScale = !isPaused ? 1f : 0f;
        Time.timeScale = timeScale;
        GetComponent<MouseLook> ().enabled = !isPaused;
        qualityChooser.SetQualitySliderActive(isPaused);
    }
}
```

3. Edit the `QualityController` script class, and, in the `Awake()` method, change the last line to pass false (not true) to the `SetQualitySliderActive(...)` method:

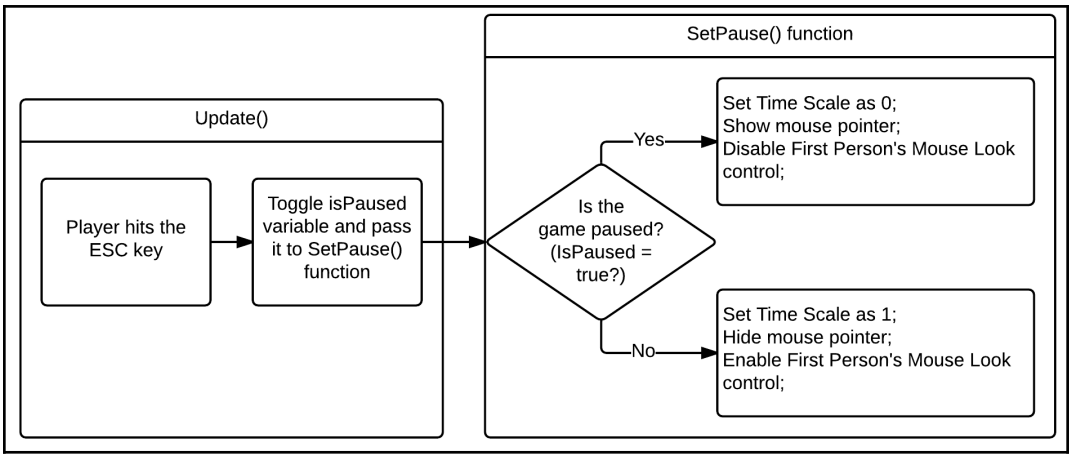
```
void Awake () {
    slider.maxValue = QualitySettings.names.Length - 1;
    slider.value = QualitySettings.GetQualityLevel();
    SetQualitySliderActive(false);
}
```

4. When you play the **Scene**, you should be able to pause/resume the game by pressing the *Esc* key, which will also display/hide the slider that controls the game's quality settings.

How it works...

To pause a Unity game with a script, we need to set the game's Time Scale to 0 (and set it back to 1 to resume). The `SetPause()` method does these actions according to the value of the `isPaused` variable:

- `isPause`= `true`: Time Scale 0 (pause the game), disable `MouseLook` component, and activate the quality slider and mouse cursor
- `isPause` = `false`: Time Scale 1 (resume the game), enable `MouseLook` component, and deactivate the quality slider and mouse cursor:



In the `Update()` method, a test is made for each frame, whether or not the *Esc* key has been pressed. If pressed, the value of `isPaused` is toggled, and the `SetPause()` method is invoked.

There's more...

Here are some ways to go further with this recipe.

Learning more about quality settings

Our code for changing quality settings is a modification of an example given by Unity's documentation. If you want to learn more about this subject, check out the following link: `http://docs.unity3d.com/ScriptReference/QualitySettings.html`.

Offering the user further game settings

You could add more **UI panels** to be activated when the game is paused, for example, for sound volume controls, save/load buttons, and so on.

Implementing slow motion

Since Remedy Entertainment's *Max Payne* video game, slow motion, or bullet time, has become a popular feature in games. For example, Criterion's *Burnout* series has successfully explored the slow motion effect in the racing genre. In this recipe, we will implement a slow motion effect that is triggered by pressing the mouse's right button.

Getting ready

For this recipe, we will use the same package as the previous recipes, `BallGame`, which is in the `16_07` folder.

How to do it...

To implement slow motion, follow these steps:

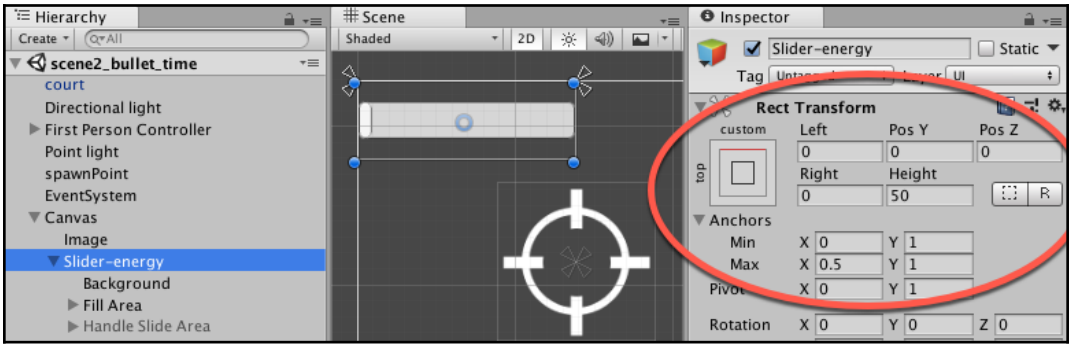
1. Import the `BallGame` package into your project and, from the **Project** panel, open the level named `scene1_ballGame`.
2. Create a C# script class called `BulletTime`, and add an instance object as a component to the **First Person Controller** GameObject:

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

 public class BulletTime : MonoBehaviour {
     public float slowSpeed = 0.1f;
     public float totalTime = 10f;
     public float recoveryRate = 0.5f;
     public Slider EnergyBar;
     private float elapsed = 0f;
     private bool isSlow = false;

     void Update (){
         if (Input.GetButtonDown ("Fire2") && elapsed < totalTime)
             SetSpeed (slowSpeed);
```

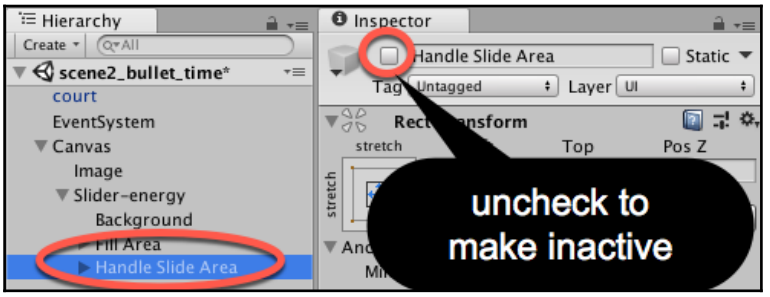
```
            if (Input.GetButtonUp ("Fire2"))
                SetSpeed (1f);

            if (isSlow) {
                elapsed += Time.deltaTime / sloSpeed;

                if (elapsed >= totalTime)
                    SetSpeed (1f);
            } else {
                elapsed -= Time.deltaTime * recoveryRate;
                elapsed = Mathf.Clamp (elapsed, 0, totalTime);
            }

            float remainingTime = (totalTime - elapsed) / totalTime;
            EnergyBar.value = remainingTime;
        }

        private void SetSpeed (float speed) {
            Time.timeScale = speed;
            Time.fixedDeltaTime = 0.02f * speed;
            isSlow = !(speed >= 1.0f);
        }
    }
```

3. Add a **UI Slider** to the scene named **Slider-energy** by choosing menu: **Create | UI | Slider**. Please note that it will be created as a child of the existing Canvas object.
4. Select **Slider-energy** and, from the **Rect Transform** component in the **Inspector**, set its **Anchors** as follows:
    - **Min X: 0, Y: 1**
    - **Max X: 0.5, Y: 1**
    - **Pivot X: 0, Y: 1**

5. Select **Slider-energy** and, from the **Rect Transform** component in the **Inspector**, set its **Position** as follows:
    - **Left: 0**
    - **Pos Y: 0**
    - **Pos Z: 0**
    - **Right: 0**
    - **Height: 50**:



6. In the **Inspector**, set the slider's child **GameObject Handle Slide Area** as inactive:



7. Finally, select the **First Person Controller** from the **Hierarchy**. Then, find the **Bullet Time** component, and drag the GameObject **Slider-energy** from the **Hierarchy** into its **Energy Bar** slot.
8. Play your game. You should be able to activate slow motion by holding down the right mouse button (or whatever alternative you have set for the Input axis Fire2). The slider will act as a progress bar that slowly shrinks, indicating the remaining bullet time you have.

How it works...

Basically, all we need to do to have the slow motion effect is decrease the `Time.timeScale` variable. In our script, we do that by using the `slowSpeed` variable. Please note that we also need to adjust the `Time.fixedDeltaTime` variable, updating the physics simulation of our game.

In order to make the experience more challenging, we have also implemented a sort of energy bar to indicate how much bullet time the player has left (the initial value is given, in seconds, by the `totalTime` variable). Whenever the player is not using bullet time, he/she has his/her quota filled according to the `recoveryRate` variable.

Regarding the **UI Slider**, we have used the Rect Transform settings to place it on the top-left corner and set its dimensions to half of the screen's width and 50 pixels tall. Also, we have hidden the handle slide area to make it similar to a traditional energy bar. Finally, instead of allowing direct interaction from the player with the slider, we have used the `BulletTime` script to change the slider's value.

There's more...

Some suggestions on how you can improve your slow motion effect even further are described in the following subsections.

Customizing the slider

Don't forget that you can personalize the slider's appearance by creating your own sprites, or even by changing the slider's fill color based on the slider's value.

Try adding the following lines of code to the end of the `Update` function:

```
      GameObject fill = GameObject.Find("Fill").gameObject;
       Color sliderColor = Color.Lerp(Color.red, Color.green,
remainingTime);
       fill.GetComponent<Image> ().color = sliderColor;
```

Adding Motion Blur

Motion Blur is an image effect that's frequently identified with slow motion. Once attached to the camera, it can be enabled or disabled depending on the speed float value. For more information on the Motion Blur post-processing image effect, refer to the following links:

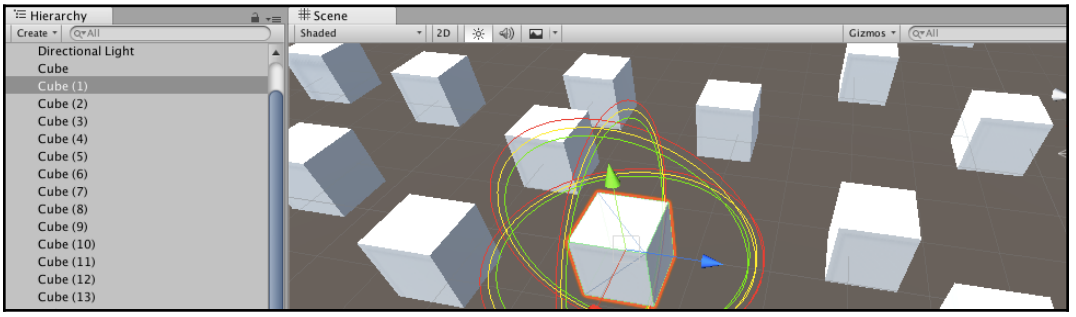
- `https://github.com/Unity-Technologies/PostProcessing/wiki/Motion-Blur`
- `https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/Motion-Blur.html`
- `https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/Manipulating-the-Stack.html`

Creating sonic ambience

Max Payne famously used a strong, heavy heartbeat sound as sonic ambience. You could also try lowering the sound effects volume to convey the character's focus when in slow motion. Plus, using audio filters on the camera could be an interesting option.

Using Gizmo to show the currently selected object in a scene panel

Gizmos are visual aids that are provided to game designers in the scene panel. In this recipe, we'll highlight the **GameObject** that is currently selected in the **Hierarchy** in the **Scene** panel:



How to do it...

To create a Gizmo to show the selected object in the **Scene** panel, follow these steps:

1. Create a new 3D project.
2. Create a 3D cube by choosing menu: **Create | 3D Object | Cube**.
3. Create a C# script class called `GizmoHighlightSelected`, and add an instance object as a component to the **3D Cube**:

```
using UnityEngine;

 public class GizmoHighlightSelected : MonoBehaviour {
     public float radius = 5.0f;

     void OnDrawGizmosSelected() {
         Gizmos.color = Color.red;
         Gizmos.DrawWireSphere(transform.position, radius);

         Gizmos.color = Color.yellow;
         Gizmos.DrawWireSphere(transform.position, radius - 0.1f);

         Gizmos.color = Color.green;
         Gizmos.DrawWireSphere(transform.position, radius - 0.2f);
     }
 }
```

4. Make lots of duplicates of the **3D Cube**, distributing them randomly around the scene.
5. When you select one cube in the Hierarchy, you should see three colored wireframe spheres drawn around the selected **GameObject** in the **Scene** panel.

How it works...

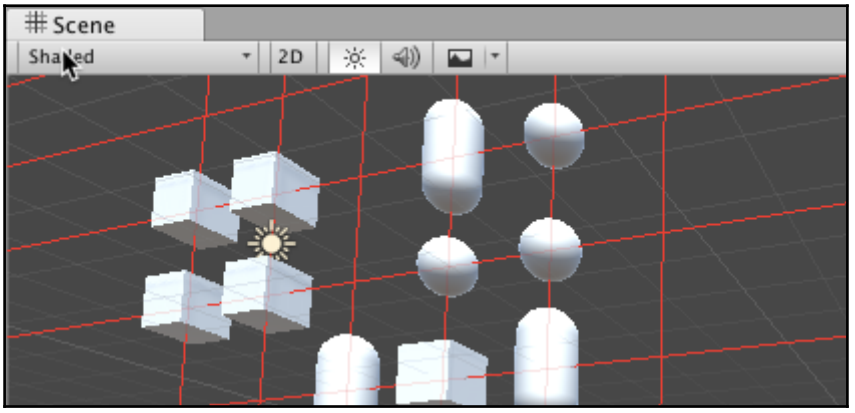
When an object is selected in a scene, if it contains a scripted component that includes the `OnDrawGizmosSelected()` method, then that method is invoked. Our method draws three concentric wireframe spheres in three different colors around the selected object.

You can learn more from the Gizmos Unity manual entry at `https://docs.unity3d.com/Manual/GizmosMenu.html`.

Editor snap-to grid drawn by Gizmo

If the positioning of objects needs to be restricted to specific increments, it is useful to have a grid drawn in the scene panel to help ensure that new objects are positioned based on those values, and also code to snap objects to that grid.

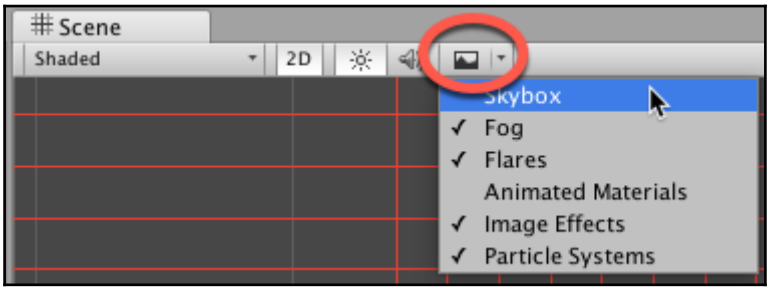
In this recipe, we'll use Gizmos to draw a grid with a customizable grid size, color, number of lines, and line length. The result of following this recipe will be as follows:



How to do it...

To create a Gizmo to show the selected object in the **Scene** panel, follow these steps:

1. Create a new 3D project.
2. For the **Scene** panel, turn off the **Skybox** view (or simply toggle off all the visual settings) so that you have a plain background for your grid work:



3. The display, and updating the child objects, will be performed by a script class called `GridGizmo`. Create a new C# script class called `GridGizmo` which contains the following:

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

 public class GridGizmo : MonoBehaviour {
     [SerializeField]
     public int grid = 2;

     public void SetGrid(int grid) {
         this.grid = grid;
         SnapAllChildren();
     }

     [SerializeField]
     public Color gridColor = Color.red;

     [SerializeField]
     public int numLines = 6;

     [SerializeField]
     public int lineLength = 50;

     private void SnapAllChildren() {
         foreach (Transform child in transform)
             SnapPositionToGrid(child);
     }

     void OnDrawGizmos() {
         Gizmos.color = gridColor;

         int min = -lineLength;
         int max = lineLength;

         int n = -1 * RoundForGrid(numLines / 2);
         for (int i = 0; i < numLines; i++) {
             Vector3 start = new Vector3(min, n, 0);
             Vector3 end = new Vector3(max, n, 0);
             Gizmos.DrawLine(start, end);

             start = new Vector3(n, min, 0);
             end = new Vector3(n, max, 0);
             Gizmos.DrawLine(start, end);

             n += grid;
```

```
            }
        }

        public int RoundForGrid(int n) {
            return (n/ grid) * grid;
        }

        public int RoundForGrid(float n) {
            int posInt = (int) (n / grid);
            return posInt * grid;
        }

        public void SnapPositionToGrid(Transform transform) {
            transform.position = new Vector3 (
                RoundForGrid(transform.position.x),
                RoundForGrid(transform.position.y),
                RoundForGrid(transform.position.z)
            );
        }
    }
```

4. Let's use an Editor script to add a new menu item to the **GameObject** menu. Create a folder named `Editor`, and in that folder, create a new C# script class called `EditorGridGizmoMenuItem`, which contains the following:

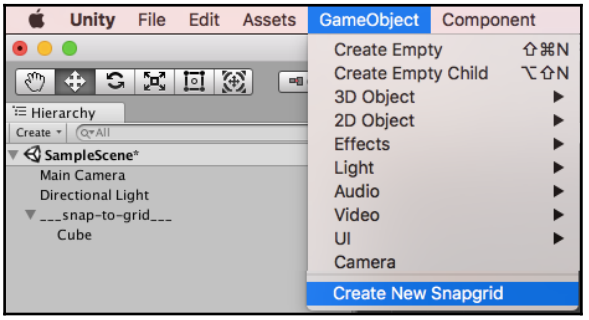
```
    using UnityEngine;
     using UnityEditor;
     using System.Collections;

     public class EditorGridGizmoMenuItem : Editor {
         [MenuItem("GameObject/Create New Snapgrid", false, 10000)]
         static void CreateCustomEmptyGameObject(MenuCommand
menuCommand) {
             GameObject gameObject = new GameObject("___snap-to-
grid___");

             gameObject.transform.parent = null;
             gameObject.transform.position = Vector3.zero;
             gameObject.AddComponent<GridGizmo>();
         }
     }
```

5. Now, let's add another **Editor** script for a custom Inspector display (and updater) for the `GridGizmo` components. Also, in your **Editor** folder, create a new C# script class called `EditorGridGizmo`, which contains the following:

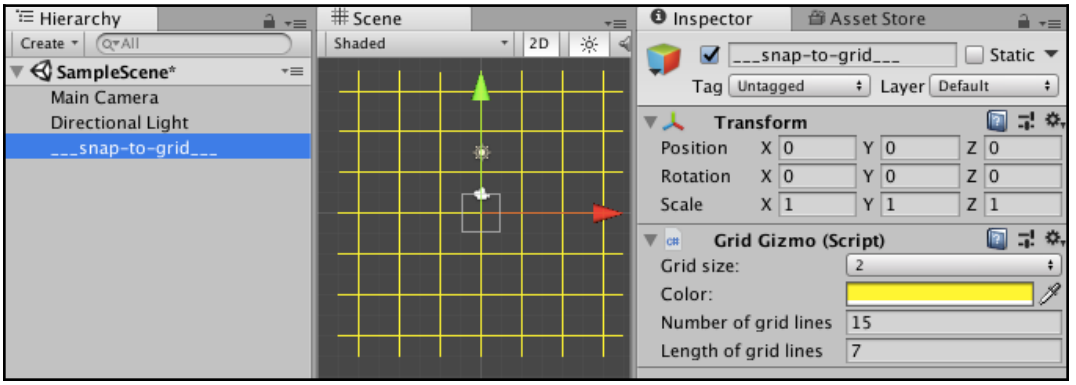
```
using UnityEngine;
 using UnityEditor;
 using System.Collections;

 [CustomEditor(typeof(GridGizmo))]
 public class EditorGridGizmo : Editor {
     private GridGizmo gridGizmoObject;
     private int grid;
     private Color gridColor;
     private int numLines;
     private int lineLength;

     private string[] gridSizes = {
         "1", "2", "3", "4", "5"
     };

     void OnEnable() {
         gridGizmoObject = (GridGizmo)target;
         grid = serializedObject.FindProperty("grid").intValue;
         gridColor =
serializedObject.FindProperty("gridColor").colorValue;
         numLines =
serializedObject.FindProperty("numLines").intValue;
         lineLength =
serializedObject.FindProperty("lineLength").intValue;
     }

     public override void OnInspectorGUI() {
         serializedObject.Update ();

         int gridIndex = grid - 1;
         gridIndex =  EditorGUILayout.Popup("Grid size:",
gridIndex, gridSizes);
         gridColor = EditorGUILayout.ColorField("Color:",
gridColor);
         numLines =  EditorGUILayout.IntField("Number of grid
lines",  numLines);
         lineLength =  EditorGUILayout.IntField("Length of grid
lines",  lineLength);

         grid = gridIndex + 1;
         gridGizmoObject.SetGrid(grid);
```

```
            gridGizmoObject.gridColor = gridColor;
            gridGizmoObject.numLines = numLines;
            gridGizmoObject.lineLength = lineLength;
            serializedObject.ApplyModifiedProperties ();
            sceneView.RepaintAll();
        }
    }
```

6. Add a new `GizmoGrid` GameObject to the scene by choosing menu: **GameObject | Create New Snapgrid**. A new **GameObject** named **___snap-to-grid___** should be added to the **Hierarchy**:



7. Select **GameObject ___snap-to-grid___**, and modify some of its properties in the **Inspector**. You can change the grid size, the color of the grid lines, the number of lines, and their length:



8. Create a **3D Cube** by choosing menu: **Create | 3D Object | Cube**. Now, drag the **3D Cube** into the **Hierarchy** and child it to **GameObject ___snap-to-grid___**.

9. We now need a small script class so that, each time the **GameObject** is moved (in Editor mode), it asks for its position to be snapped by the parent scripted component `SnapToGizmoGrid`. Create a C# script class called `SnapToGizmoGrid` and add an instance object as a component to the **3D Cube**:

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

 [ExecuteInEditMode]
 public class SnapToGridGizmo : MonoBehaviour {
     public void Update()
     {
 #if UNITY_EDITOR
transform.parent.GetComponent<GridGizmo>().SnapPositionToGrid(transfor
m);
 #endif
     }
 }
```

10. Make lots of duplicates of the 3D cube, distributing them randomly around the scene—you'll find that they snap to the grid.
11. Select **GameObject ___snap-to-grid___ again**, and modify some of its properties in the Inspector. You'll see that the changes are instantly visible in the scene, and that all child objects that have a scripted component of `SnapToGizmoGrid` are snapped to any new grid size changes.

How it works...

The `EditorGridGizmoMenuItem` script class adds a new item to the **GameObject** menu. When selected, a new **GameObject** is added to the Hierarchy named **___snap-to-grid___**, positioned at (**0, 0, 0**), and containing an instance object component of the `GridGizmo` script class.

`GridGizmo` draws a 2D grid based on public properties for grid size, color, number of lines, and line length. Regarding the `SetGrid(...)` method, as well as updating the integer grid size variable grid, it also invokes the `SnapAllChildren()` method, so that each time the grid size is changed, all child **GameObjects** are snapped into the new grid positions.

The `SnapToGridGizmo` script class includes an Editor attribute `[ExecuteInEditMode]` so that it will receive `Update()` messages when its properties are changed at design time in the Editor. Each time `Update()` is invoked, it calls the `SnapPositionToGrid(...)` method in its parent `GridGizmo` instance object so that its position is snapped based on the current settings of the grid. To ensure this logic and code is not compiled into any final build of the game, the contents of `Update()` are wrapped in an #if UNITY_EDITOR compiler test. Such content is removed before a build is compiled for the final game.

The `EditorGridGizmo` script class is a custom Editor Inspector component. This allows for both control of which properties are displayed in the **Inspector**, how they are displayed, and it allows actions to be performed when any values are changed. So, for example, after changes have been saved, the `sceneView.RepaintAll()` statement ensures that the grid is re-displayed, since it results in an `OnDrawGizmos()` message being sent.

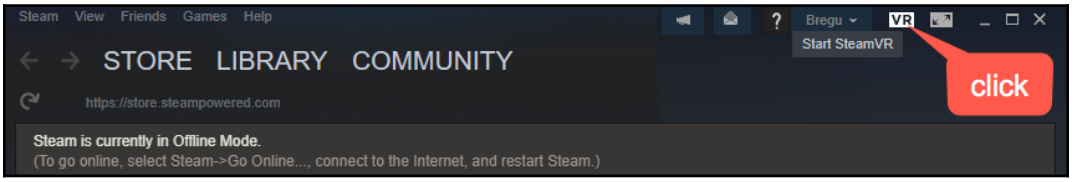
Creating a VR project

In this recipe, we will go through the steps for setting up a basic VR scene in Unity, using the Vive VR headset on a Windows computer.

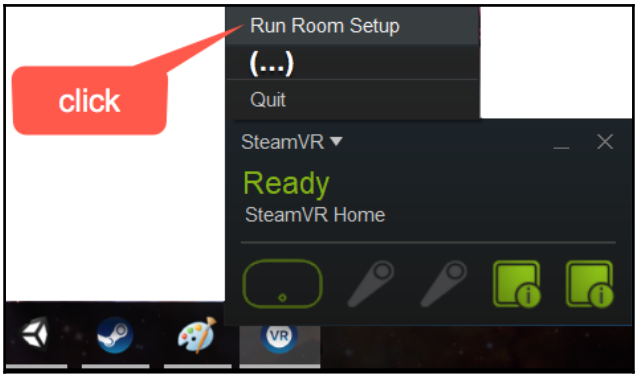
Getting ready

You need Steam VR with the Standing Only or Room-scale set up. If you have not done so yet, then follow these steps to set up your Vive headset so that it's ready for Unity game development:

1. Install Steam
2. Install Steam VR
3. Plug in your Vive headset
4. From the Steam application window, run Steam VR (click VR at top right of the Steam app window):



5. Steam VR should run. Then, choose the **Run Room Setup** menu item:



6. Position your **Light Houses** to cover the space in the room that you want to use.
7. From the Room Setup screen, choose your room setup: **Standing Only** or **Room-Scale**:
    - **Standing Only**:
        - Use the headset to set the center
        - Use the headset to locate the floor
    - **Room-Scale**:
        - Position your Light Houses to cover the space you want to use
        - Calibrate the floor by putting hand controllers on the floor
        - Walk around room, using hand controllers with the trigger to trace the space you can safely move around
8. You can now explore **Steam VR Home**.

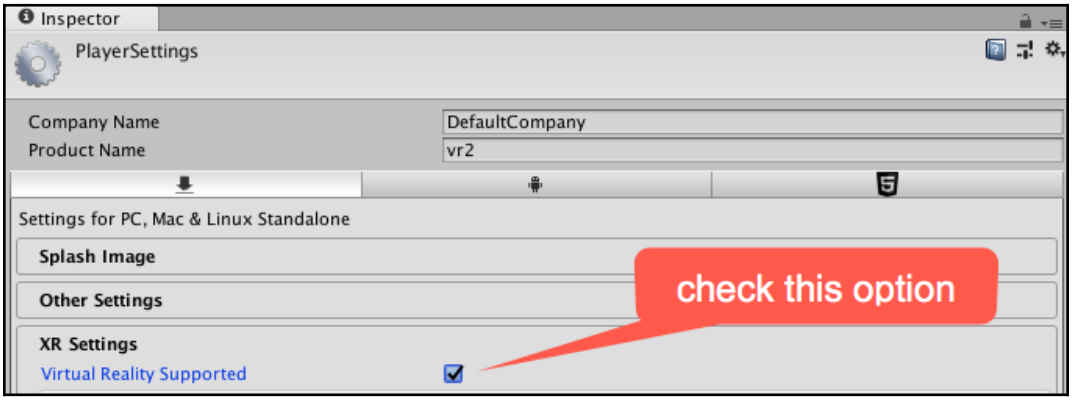
How to do it...

To create a basic VR Unity project, follow these steps:

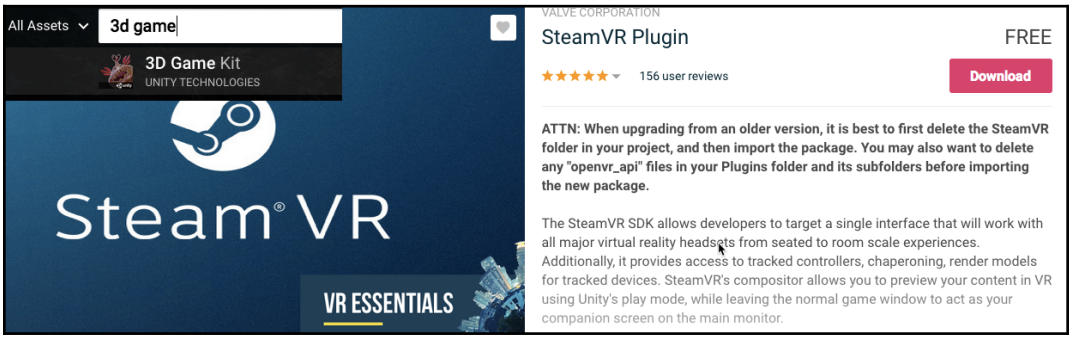
1. Start a new 3D project.

Either work with a new 3D Unity project, or make a backup of any project to which you are about to add VR features to, since you will be adding a package that makes changes to many settings, and it could mess up settings in an existing project.

2. Delete the **Main Camera** from the **Scene**.
3. Display the Unity **Player** settings in the **Inspector** by choosing menu: **Edit | Project Settings | Player**.
4. Check the **Virtual Reality Supported** option near the bottom of the **Inspector**:



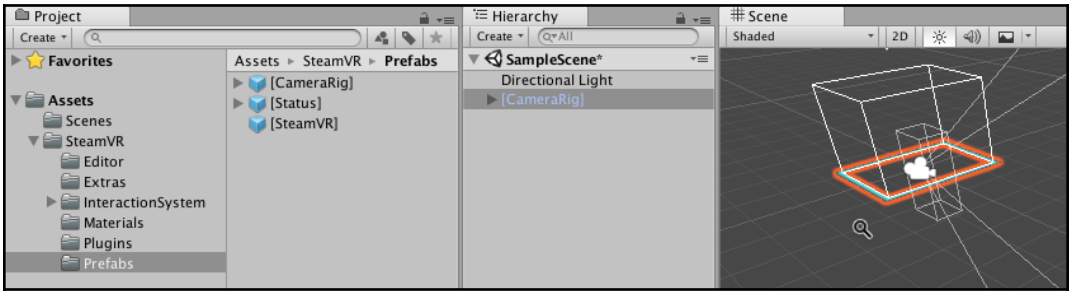
5. Ensure that you are logged into your Unity Account (before accessing the **Asset Store**).
6. Visit the **Asset Store** and search for **Steam VR** from the Valve Corporation:



7. Download and import the package (you'll be warned about having made a backup before importing...).

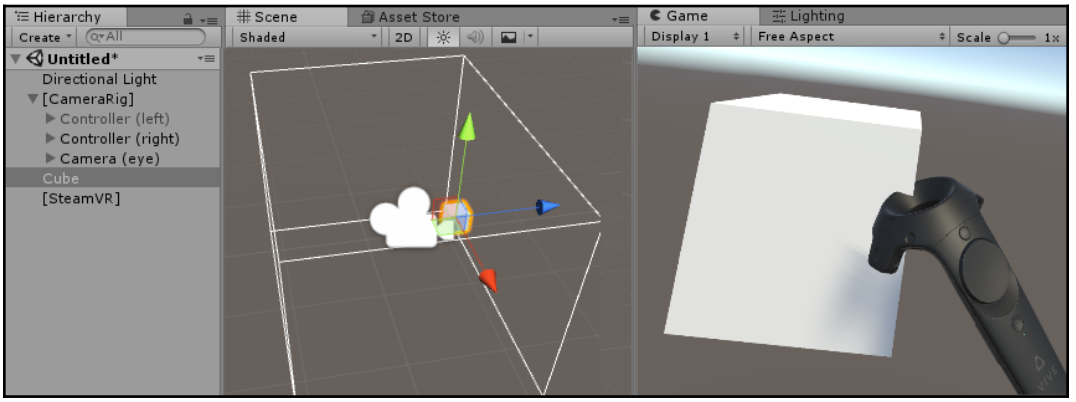
8. Choose your preferred options from any pop-up about builds:

9. Drag into the **Scene** a clone of the **[CameraRig]** prefab from folder: **Project | SteamVR | Prefabs**. You'll see a 3D space representing your room setup (our examples show the rectangular-based space from the **Standing only** room setup):



10. Run the **Scene**, put on your VR headset, and pick up your hand controllers. You should see 3D representations of the position, trigger settings, and so on of your hand controllers in the virtual space.
11. Import/create 3D Objects in your scene - for example, add a **3D Cube** to the scene inside the room space of the **CameraRig**.

12. Run the **Scene**, and try moving a virtual hand controller to collide with your 3D cube:



13. Test the physics by adding a **Rigid Body** component to the **3D Cube** (and turn off gravity), adding/making the **Box Collider** to the GameObject **Controller (right)** smaller.
14. Run the scene - you should be able to push the **3D Cube** with the virtual hand controller.

How it works...

You've learned to set up Vive and locate and install the **Steam VR** package.

This package contains prefabs so that the headset and handset work in the standing/room space you set up.

You created a Unity project with the Vive prefabs. You then added a **3D Cube** to the scene and interacted with it by adding colliders to the cube and a hand controller GameObject.

This was tested with **Unity 2017.4.9 LTS**, since it wasn't fully working with a 2018 version at the time of writing.

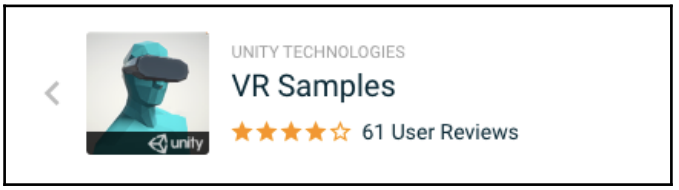
There's more...

Here are some suggestions for taking this recipe further.

Exploring free VR/XR samples/tutorials

Some good sources of free sample VR projects to explore include the following:

- Unity Technologies VR Samples: `https://assetstore.unity.com/packages/essentials/tutorial-projects/vr-samples-51519`:



- A good tutorial on Vive and Unity from **Ray Wenderlick**: `https://www.raywenderlich.com/792-htc-vive-tutorial-for-unity`
- Valve Lab Renderer on the Unity **Asset Store**: `https://assetstore.unity.com/packages/tools/the-lab-renderer-63141`
- Vive input utility on the Unity **Asset Store**: `https://assetstore.unity.com/packages/tools/integration/vive-input-utility-64219`

Setup with Oculus Rift

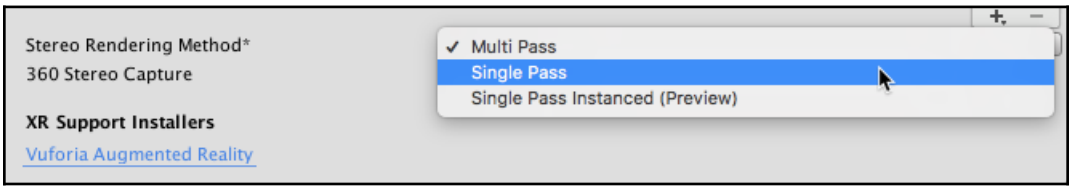
The Oculus Rift setup is similar to that with the Vive, although it actually integrates a little better with Unity. You need to do the following:

1. Install the Oculus runtime.
2. Setup for room/standing for the Infrared Cameras.
3. There is no need download any package, and no need to remove the **Main Camera**.

You can learn more about this at the Oculus Rift Unity documentation site: `https://developer.oculus.com/documentation/unity/latest/concepts/book-unity-gsg/`.

Using a Single Pass if working with the Lightweight Rendering Pipeline

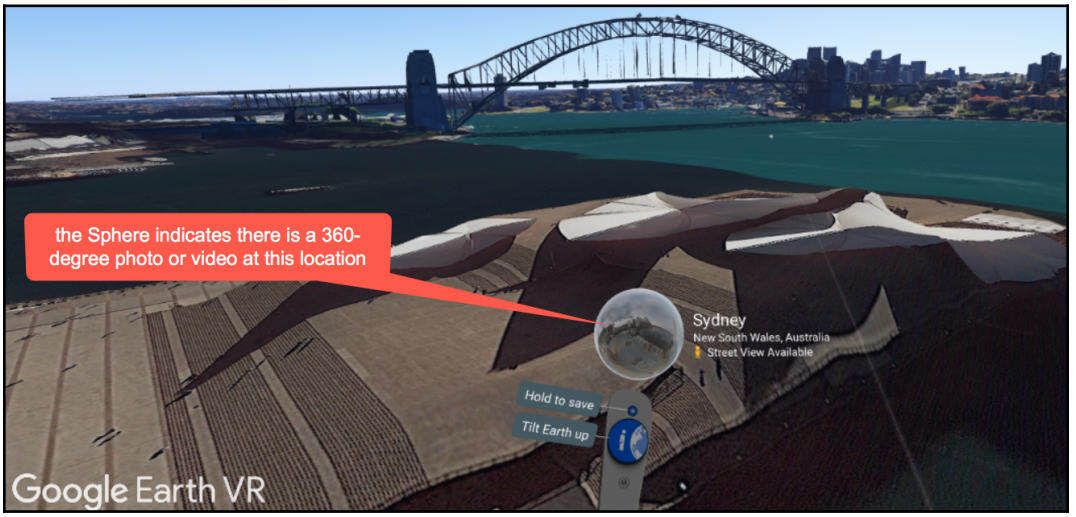
If you're using the **Lightweight Rendering Scripted Pipeline**, then you need to choose **Single Pass** for the Stereo `RenderingMethod*` property when setting up XR in the settings, having chosen menu: **Edit | Project Settings | Player**:



Adding 360-degree videos to a VR project

Google Earth VR is great fun! This screenshot from a live VR session shows the virtual hand controller, and a virtual screen menu showing photos and text about 6 suggested locations to visit:

Affordable, 360-degree cameras means that it's easy to create your own, or find free online 360-degree images and video clips. In this recipe, we'll learn how to add a 360-degree video clip as a Skybox in a VR project. You will also learn how the 360-degree video clips can be played on the surface of 3D objects, including the inside of a sphere—a bit like **Google Earth VR** mode when you raise the sphere to your head to view its 360-degree image contents:



Getting ready

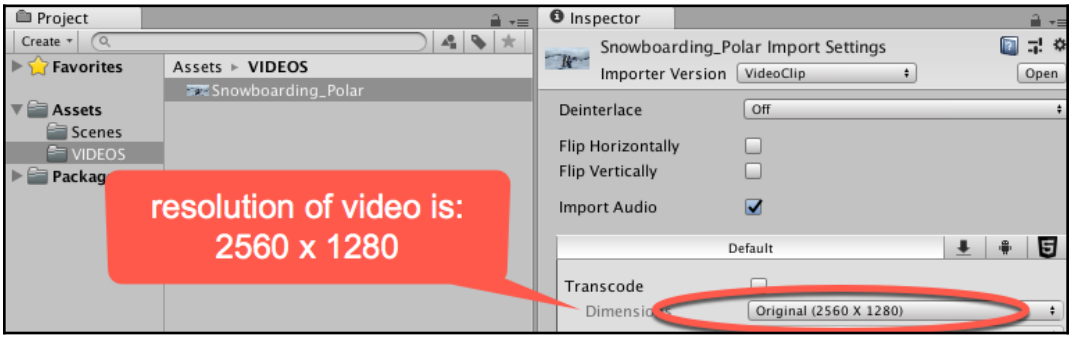
For this recipe, we have provided a short `Snowboarding_Polar.mp4` video in the `16_07` folder. This project builds on the previous one (a basic VR project), so make a copy of that and work on the copy.

Special thanks to Kris Szczurowski for permission to use his snowboarding 360-degree video clip, and for his help with these VR project recipes. Good luck with the PhD!

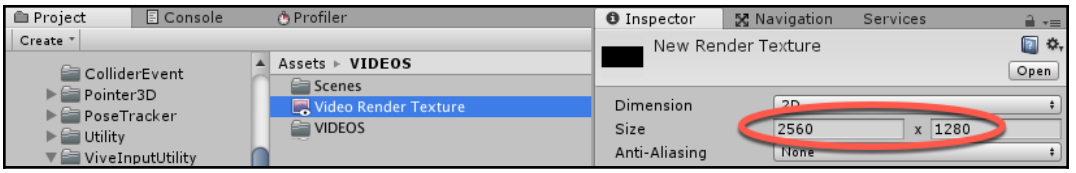
How to do it...

To add 360-degree videos to a VR project, follow these steps:

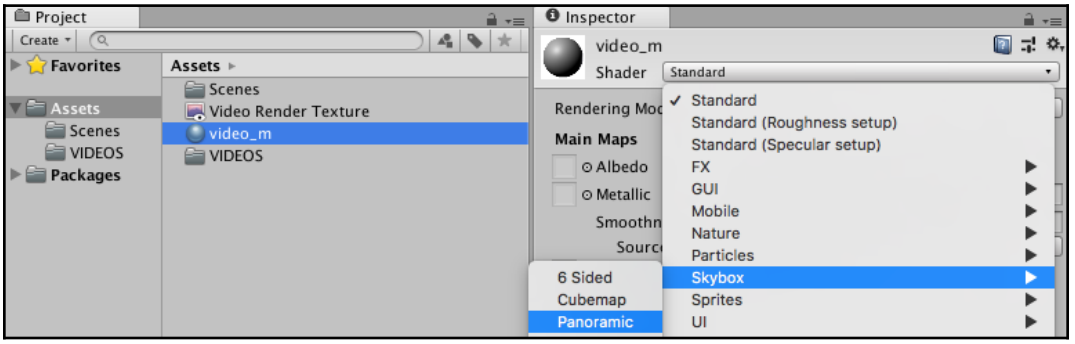
1. Import your 360-degree polar format video clip into your Unity project (in our example, this is `Snowboarding_Polar.mp4`).
2. Select the video asset in the **Project** panel, and in the **Inspector**, make a note of its resolution (we'll need this later), for example, **2,560 x 1,280**:



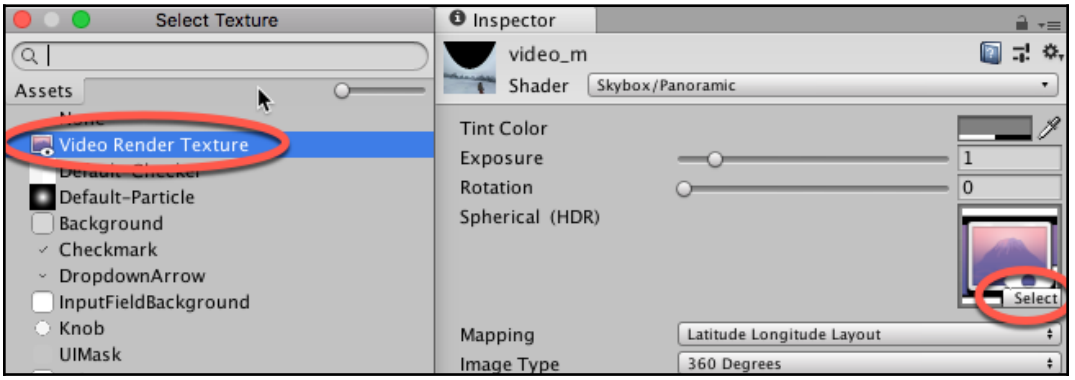
3. Create a new empty GameObject named **video-player** by choosing menu: **Create | Empty**.
4. Select **video-player** in the **Hierarchy**, and in the **Inspector**, add a component **Video Player** component by choosing: **Add Component | Video | Video Player**.
5. From the **Project** panel, drag your video asset file, for example, `Snowboarding_Polar`, into the **Video Clip** property of the **Video Player** component.
6. Create a new **Render Texture** asset file named `VideoRenderTexture` by choosing menu: **Create | Render Texture**.
7. Set the Resolution of the `VideoRenderTexture` to match the video asset resolution, for example, **2,560 x 1,280**:



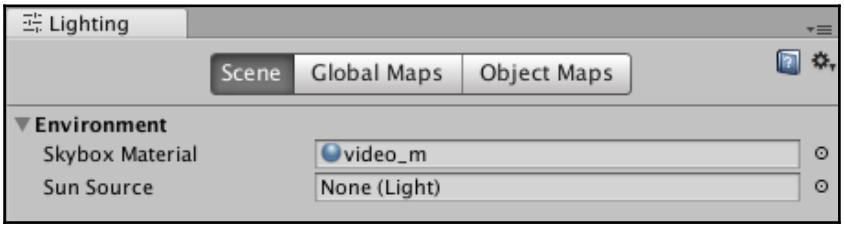
8. In the **Hierarchy**, select **GameObject** video-player, and for the **Video Player** component, set the target texture to be `VideoRenderTexture`.
9. Create a new **Material** named `video_m` by choosing menu: **Create** | **Material**.
10. With `video_m` selected in the **Project** panel, change its **Shader** to **Skybox** | **Panoramic**:



11. In the **Inspector**, for the Spherical HDR property, click the **Select** button and choose Texture `VideoRenderTexture`:



12. Open the **Lighting Settings** panel, choose menu: **Window** | **Rendering** | **Lighting Settings**. In the **Inspector**, set the **Skybox Material** to `video_m`:



13. Play the **Scene**, put on your VR headset, and you should see the 360-degree video playing all around you.

How it works...

You have created a GameObject with a **Video Player** component, and made it play your 360-degree video. You made this component render to a **Render Texture** of the same dimensions as your video.

You created a **Skybox-panomaric Material,** and selected your **Render Texture** as the **Texture** for this **Material**. You then set this **Material** as the **Skybox** for the **Lighting Settings**.

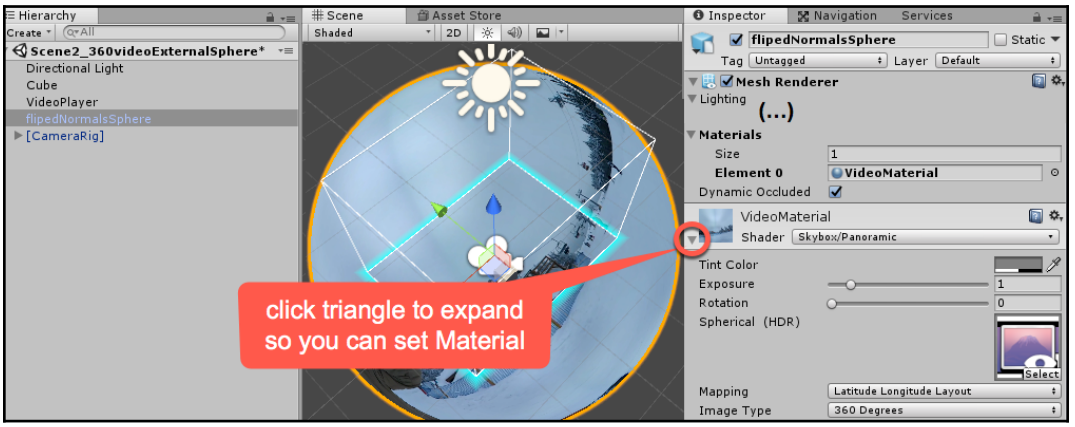
This was tested with **Unity 2017.4.9 LTS**, since it wasn't fully working with a 2018 version at the time of writing.

There's more...

Here are some suggestions for taking this recipe further.

Playing 360-degree videos on the surface of a 3D object

To play 360-degree videos on the surface of a 3D object, perform the aforementioned steps, but do not set the Skybox to `video_m`. Instead, set the **Material** of the **Mesh Renderer** component of your 3D object to `video_m`:

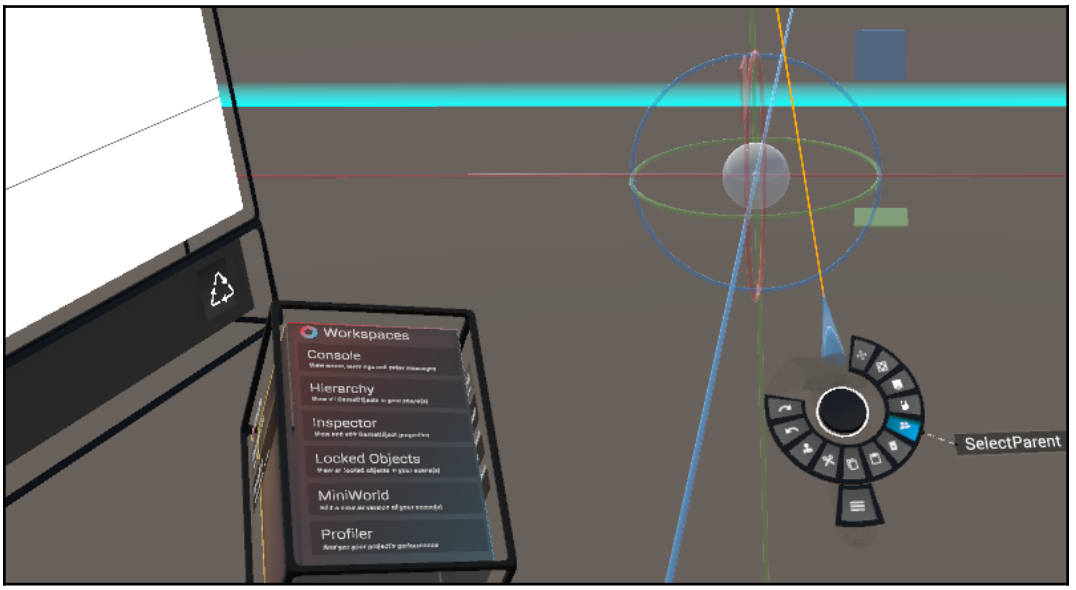


This works for 3D objects with inverted normals, for example, a hollow 3D Sphere that you can look at from the inside.

Working with VR content inside a VR environment – the XR Editor

An exciting project from Unity is the **XR-Editor**, which is a VR environment that allows you to edit a scene in VR. The project provides great examples of VR UI elements, including 3D menus and laser pointer selectors. It allows you to see Console reports in the environment and interact with the **Hierarchy** of GameObjects: In this screenshot we can see the main Workspace menu displayed on the virtual left hand controller, and the virtual right hand controller is being presented as a rotary menu for actions that can be performed on the currently selected GameObject.

The 'Select Parent' option is currently selected in this rotary menu:



In this recipe, we will set up the XR-Editor for use in a Unity project.

Getting ready

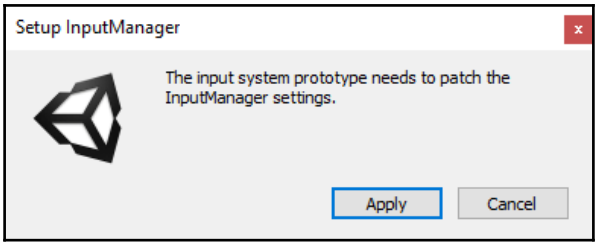
Start this recipe with a copy of the Basic VR project created in the recipe before the previous one.

How to do it ...

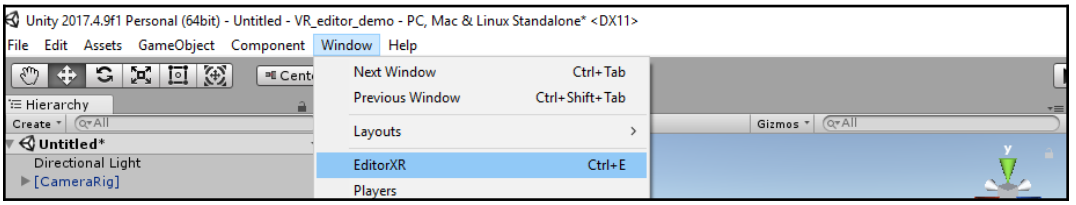
To work with VR content inside a VR environment in the XR Editor, do the following:

1. Install the **TextMeshPro** package, choosing menu: **Window | Package Manager**, and selecting **Text Mesh Pro**.
2. Download the **Editor XR** package from the links on the Unity blog page:
    - `http://rebrand.ly/EditorVR-package`
    - `https://blogs.unity3d.com/2016/12/15/editorvr-experimental-build-available-today/`
3. Import the **EditorXR-package** package into your project.

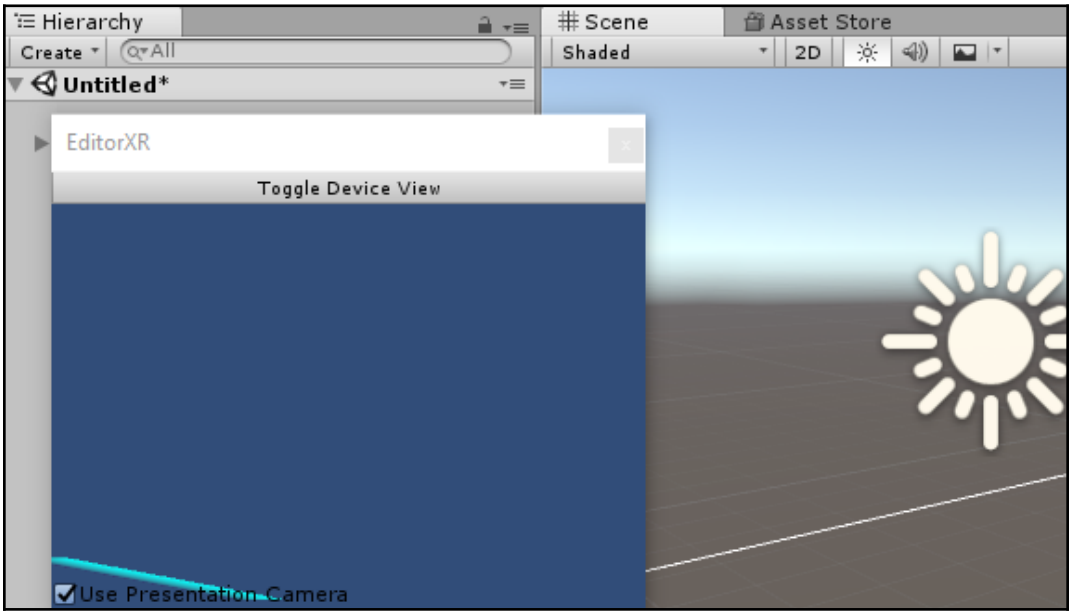
4. Agree to the prompt on-screen about patching the **Input Manager** settings:



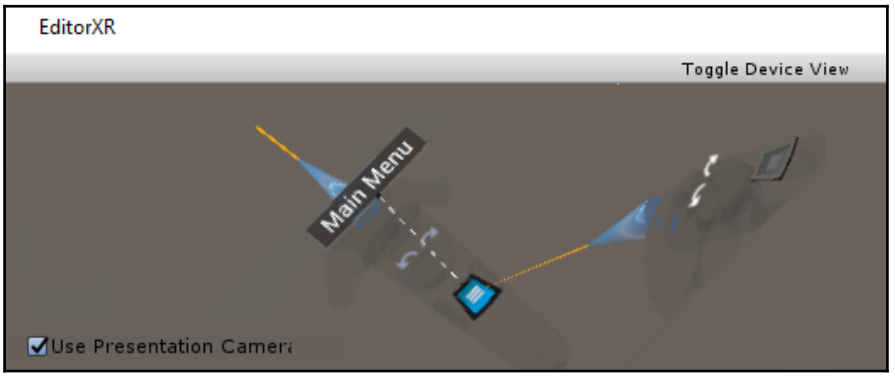
5. There should be a new item on the Window Menu, **Window | EditorXR**. Choose this menu item:



6. A new, floating EditorXR application window should now be created:

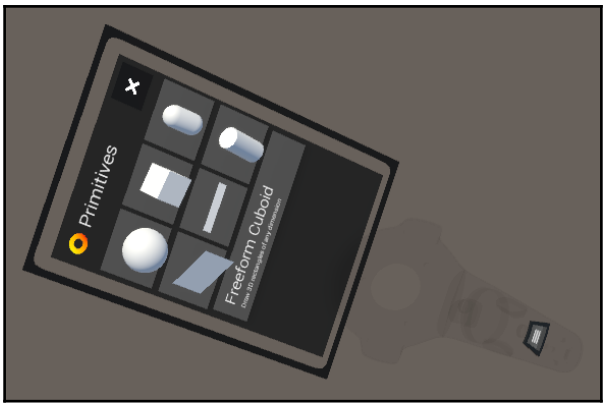


7. Put on your VR headset and start creating. Do NOT run the scene—**Editor XR** works at **Design Time**, not **Run-Time**, so do NOT play the **Scene**.
8. You display the main menu by directing the laser pointer of one virtual hand controller onto the menu bar on the other hand controller:

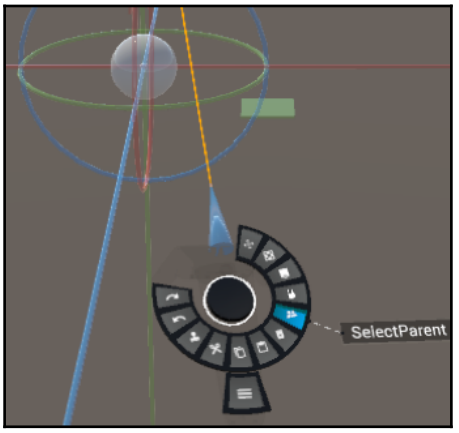


9. You'll then get a cube-like, multi-sided, rotatable main menu, from which you can choose items with the laser pointer. For example on the **Workspaces** side of the menu object there are options to display the **Console**, **Hierarchy**, **Inspector**, **Locked Objects**, **MiniWorld** and **Profiler** panels:

10. You can choose the **Primatives** menu where you can create new 3D objects:



11. With a GameObject selected, a wheel-menu offers actions, such as deleting or selecting the parent, and so on:



How it works...

In this recipe, you learned how to install and start interacting with Unity's **Editor-XR**.

This was tested with **Unity 2017.4.9 LTS**, since it wasn't fully working with a 2018 version at the time of writing.

You can learn more about **Editor XR/VR** at the Unity blog at `https://blogs.unity3d.com/2016/12/15/editorvr-experimental-build-available-today/`.