5
Using Cameras

In this chapter, we will cover:

- Creating a picture-in-picture effect
- Switching between multiple cameras
- Making textures from screen content
- Zooming a telescopic camera
- Displaying a mini-map
- Creating an in-game surveillance camera – Pro-Only

Introduction

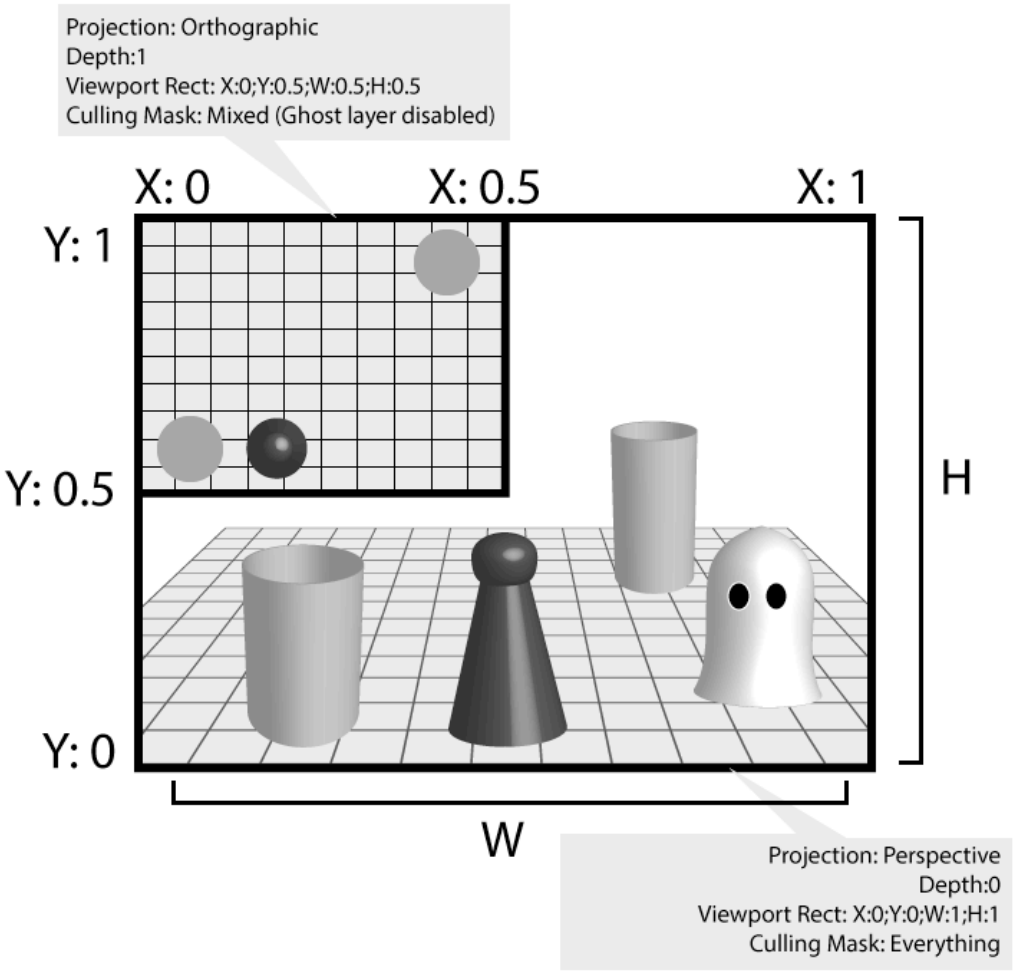
As developers, we should never forget to pay attention to the cameras. After all, they are the windows from which our players see our game. In this chapter, we will take a look at ways of using cameras in interesting ways that enhance the player's experience.

The big picture

Cameras can be customized in many ways:

- They can exclude objects on specific layers from rendering
- They can be set to render in **Orthographic** mode (that is, without perspective)
- They can have their **Field of View** manipulated to simulate wide angle lens
- They can be rendered on top of other cameras, or within specific areas of the screen.
- Plus, if you use Unity Pro, they can be rendered onto Textures.

The list goes on.



Insert image 1362OT_05_00.png

An additional note: throughout this chapter, you will notice that some recipes feature a camera rig that follows the player's third person character. That rig is the **Multipurpose Camera Rig**, originally available from Unity's Sample Assets, which can be imported into your projects via the menu **Assets | Import Package | Camera** , also available at `https://www.assetstore.unity3d.com/en/#!/content/21064` . To make things easier, we have organized the `MultipurposeCamera` Unity Package containing it as a prefab, which can be found inside the `1362_05_codes` folder.

Creating a picture-in-picture effect

Having more than one viewport displayed can be useful in many situations. For example, you might want to show simultaneous events going on in different locations, or maybe you want to have a separate window for hot-seat multiplayer games. Although you could do it manually by adjusting the **Normalized Viewport Rect** parameters on your camera, this recipe includes a series of extra preferences to make it more independent from the user's display configuration.

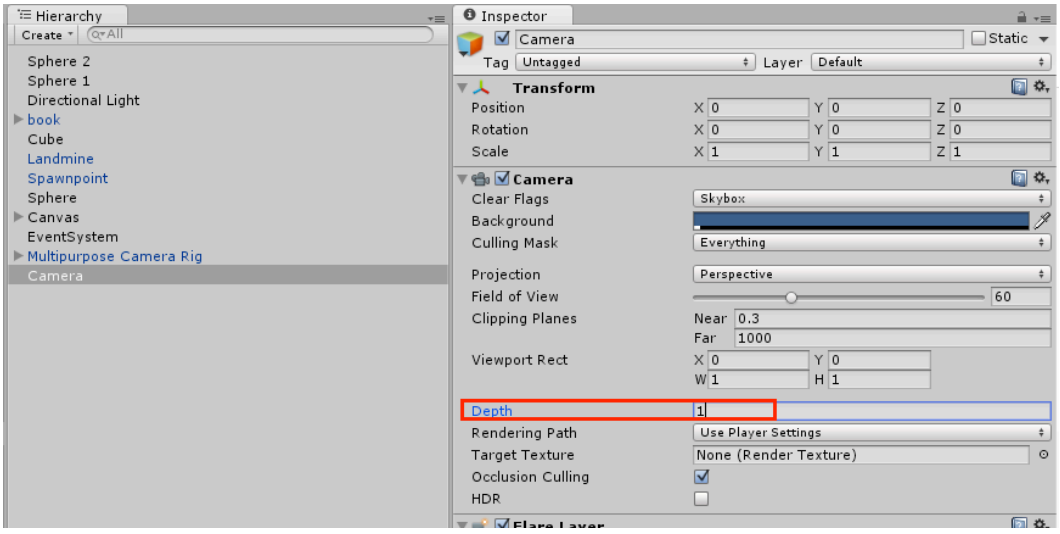
Getting ready

For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`. The package is in the `1362_05_codes` folder.

How to do it...

To create a picture-in-picture display, just follow these steps:

1. Import the `BasicScene` package into your Unity Project.
2. From the **Project** view, open the **mecanim** level. This is a basic scene featuring an animated character and some extra geometry.
3. Add a new **Camera** to the scene through the **Create** dropdown menu on top of the **Hierarchy** view (**Create | Camera**).
4. Select the camera you have created and, from the **Inspector** view, change its **Depth** to **1**.



Insert image 1362OT_05_01.png

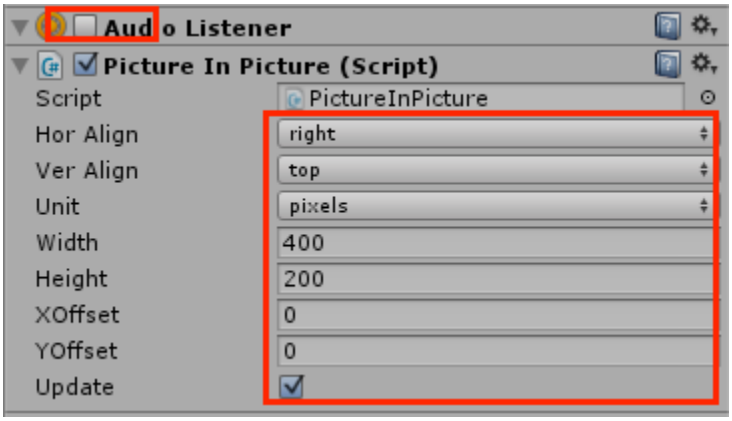
5. From the **Project** view, create a new *C# Script* and rename it `PictureInPicture`.
6. Open your script and replace everything with the following code:

```
using UnityEngine;

public class PictureInPicture: MonoBehaviour {
   public enum hAlignment{left, center, right};
   public enum vAlignment{top, middle, bottom};
   public hAlignment horAlign = hAlignment.left;
   public vAlignment verAlign = vAlignment.top;
   public enum UnitsIn{pixels, screen_percentage};
   public UnitsIn unit = UnitsIn.pixels;
   public int width = 50;
   public int height= 50;
   public int xOffset = 0;
   public int yOffset = 0;
   public bool  update = true;
   private int hsize, vsize, hloc, vloc;

   void Start (){
      AdjustCamera ();
   }
   void Update (){
      if(update)
         AdjustCamera ();
   }

   void AdjustCamera(){
      int sw = Screen.width;
      int sh = Screen.height;
      float swPercent = sw * 0.01f;
      float shPercent = sh * 0.01f;
      float xOffPercent = xOffset * swPercent;
      float yOffPercent = yOffset * shPercent;
      int xOff;
      int yOff;
      if(unit == UnitsIn.screen_percentage){
         hsize = width * (int)swPercent;
         vsize = height * (int)shPercent;
         xOff = (int)xOffPercent;
         yOff = (int)yOffPercent;
      } else {
```

```
            hsize = width;
            vsize = height;
            xOff = xOffset;
            yOff = yOffset;
        }

        switch (horAlign) {
            case hAlignment.left:
                  hloc = xOff;
                  break;
            case hAlignment.right:
                  int justfiedRight = (sw - hsize);
                  hloc = (justfiedRight - xOff);
                  break;
            case hAlignment.center:
                  float justifiedCenter = (sw * 0.5f) - (hsize *
0.5f);
                  hloc = (int)(justifiedCenter - xOff);
                  break;
            }

        switch (verAlign) {
            case vAlignment.top:
               int justifiedTop = sh - vsize;
               vloc = (justifiedTop - (yOff));
               break;
            case vAlignment.bottom:
               vloc = yOff;
               break;
            case vAlignment.middle:
               float justifiedMiddle = (sh * 0.5f) - (vsize * 0.5f);
               vloc = (int)(justifiedMiddle - yOff);
               break;
            }

        GetComponent<Camera>().pixelRect = new
Rect(hloc,vloc,hsize,vsize);
    }
}
```

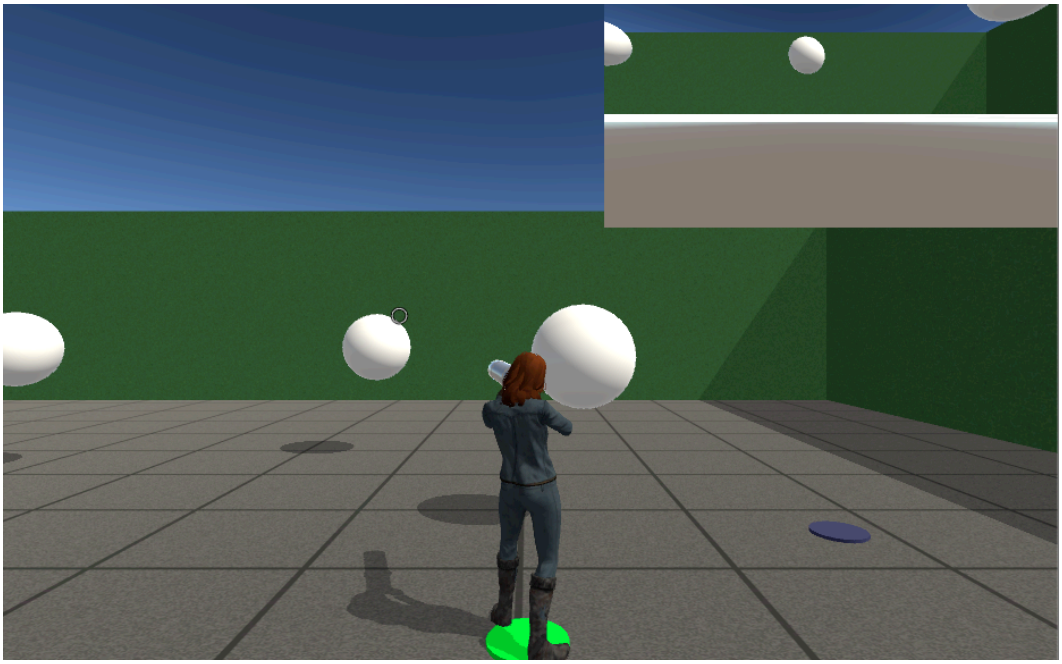
In case you haven't noticed, we are not achieving percentage by dividing numbers by 100, but rather multiplying them by 0.01. The reason behind that is performance: computer processors are faster multiplying than dividing.

7. Save your script and attach it to the camera you have previously created.
8. Uncheck the new camera's **Audio Listener** component and change some of the **PictureInPicture** parameters: change **Hor Align** to **Right**, **Ver Align** to **Top**, **Unit** to **pixels**. Leave **XOffset** and **YOffset** as **0**, change **Width** to **400** and **Height** to **200**, as shown here:



Insert image 1362OT_05_02.png

9. Play your scene. The new camera's viewport should be visible on the top right of the screen.



Insert image 1362OT_05_03.png

How it works...

In this example, we have added a second camera, in order to display the scene from a different point of view. The second camera's relative viewport was originally placed on top of the **main camera's** viewport, hence taking up all screen space.

The `PictureInPicture` script changes the camera's **Normalized Viewport Rect**, thus resizing and positioning the viewport according to the user preferences.

First, it reads user preferences for the component (dimensions, alignment and offset for the PiP viewport) and converts dimensions in screen percentage to pixels.

Later, from the conditional `if(unit == UnitsIn.screen_percentage){,` the script calculates two of the viewport rect parameters (width and height) according to the user's selection.

Later on, two **switch** statements to adjust the other two viewport rect parameters (horizontal and vertical location) according to the total screen dimensions, PiP viewport dimension, vertical/horizontal alignment, and offset.

Finally, a line of code tells the camera to change the location and dimension of the camera´s Viewport Rect:

```
GetComponent<Camera>().pixelRect = new Rect(hloc,vloc,hsize,vsize);
```

.

There's more...

The following are some aspects of your picture-in-picture you could change:

Making the picture-in-picture proportional to the screen's size

If you change the **Unit** option to `screen_percentage`, the viewport size will be based on the actual screen's dimensions instead of pixels.

Changing the position of the picture-in-picture

Ver Align and **Hor Align** can be used to change the viewport's vertical and horizontal alignment. Use them to place it where you wish.

Preventing the picture-in-picture from updating on every frame

Leave the **Update** option unchecked if you don't plan to change the viewport position in running mode. Also, it's a good idea to leave it checked when testing and uncheck it once the position has been decided and set up.

See also

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

- *Displaying a mini-map*

Switching between multiple cameras

Choosing from a variety of cameras is a common feature in many genres: racing, sports, tycoon/strategy, and many others. In this recipe, we will learn how to give players the ability of choosing one from many cameras using their keyboard.

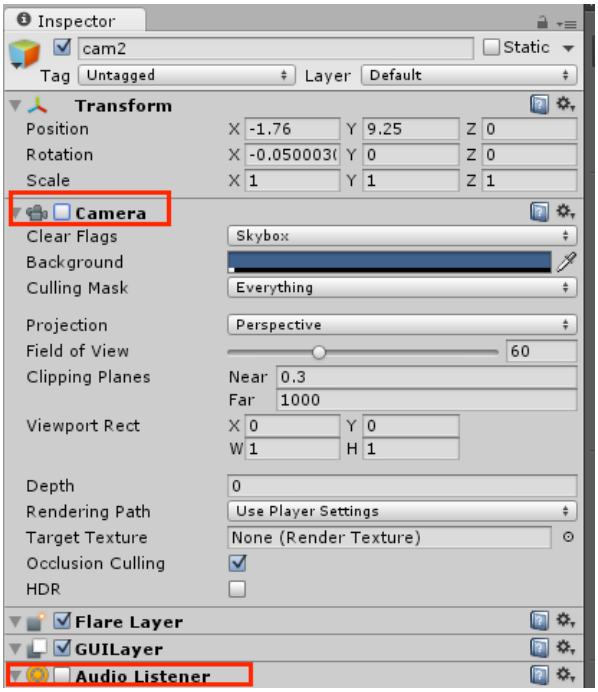
Getting ready

For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`. The package is in the `1362_05_codes` folder.

How to do it...

To implement switchable cameras, follow these steps:

1. Import the `BasicScene` package into a new Project.
2. From the **Project** view, open the **mecanim** level. This is a basic scene featuring an animated character and some extra geometry.
3. Add two more cameras to the scene through the **Create** dropdown menu on top of the **Hierarchy** view (**Create | Camera**). Rename them `cam1` and `cam2`.
4. Change the `cam2` camera's position and rotation so it won't be identical to `cam1`.
5. Create an **Empty** game object the **Create** dropdown menu on top of the **Hierarchy** view (**Create | Create Empty**). Then, rename it `Switchboard`.
6. From the **Inspector** view, disable the **Camera** and **Audio Listener** components of both `cam1` and `cam2`.



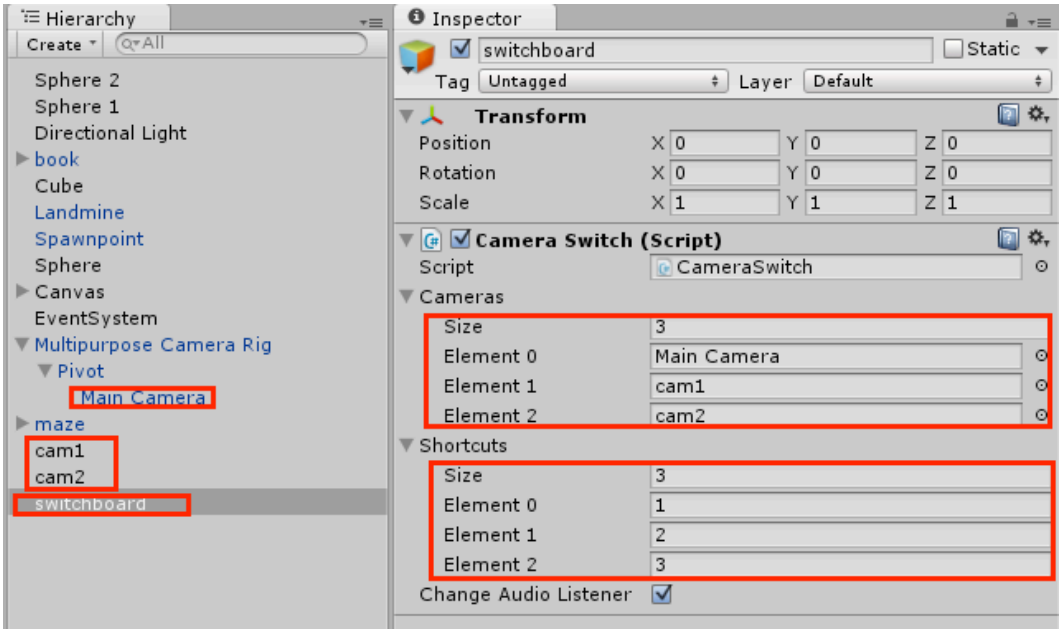
Insert image 1362OT_05_04.png

7. From the **Project** view, create a new C# Script. Rename it `CameraSwitch` and open it in your editor.
8. Open your script and replace everything with the following code:

```
using UnityEngine;

public class CameraSwitch : MonoBehaviour {
   public GameObject[] cameras;
```

```
    public string[] shortcuts;
    public bool  changeAudioListener = true;
    void  Update (){
        if (Input.anyKeyDown) {
            for (int i=0; i<cameras.Length; i++) {
                if (Input.GetKeyDown (shortcuts [i]))
                    SwitchCamera (i);
            }
        }
    }

    void  SwitchCamera ( int index  ){
        for(int i = 0; i<cameras.Length; i++){
            if(i != index){
                cameras[i].GetComponent<Camera>().enabled = false;
                if(changeAudioListener)
                    cameras[i].GetComponent<AudioListener>().enabled =
false;
            } else {
                cameras[i].GetComponent<Camera>().enabled = true;
                if(changeAudioListener)
                    cameras[i].GetComponent<AudioListener>().enabled =
true;
            }
        }
    }
}
```

9. Attach `CameraSwitch` to the `Switchboard` game object.
10. From the **Inspector** view, set both **Cameras** and **Shortcuts** size to `3`. Then, drag populate the **Cameras** slots with the cameras from the scene (including the **Main Camera**, within the **Multipurpose Camera Rig | Pivot** game object) Then, type `1`, `2`, and `3` into the **Shortcuts** text fields, as shown in the next screenshot:



Insert image 1362OT_05_05.png

11. Play your scene and test your cameras by pressing *1*, *2*, and *3* on the keyboard.

How it works...

The script is very straightforward. First, it compares the key being pressed down to the list of shortcuts. If the key is, indeed, included on a list of shortcuts, it is passed onto the `SwitchCamera` function, which, in its turn, goes through a list of cameras, enables the one associated with the shortcut that was received, also enabling its **Audio Listener**, in case the **Change Audio Listener** option is checked.

There's more...

Hera are some ideas on how you could try twisting this recipe a bit.

Using a single enabled camera

A different approach to the problem would be keeping all secondary cameras disabled and assigning their position and rotation to the main camera via a script (you would need to make a copy of the main camera and add it to the list, in case you wanted to save its transform settings).

Triggering the switch from other events

Also, you could change your camera from other Game Object's scripts by using a line of code such as the one given here:

```
GameObject.Find("Switchboard").GetComponent("CameraSwitch").SwitchCam
era(1);
```

See also

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

- *Making an inspect camera*

Making textures from screen content

If you want your game or player to take in-game snapshots and apply it as a texture, this recipe will show you how. This can be very useful if you plan to implement an in-game photo gallery or display a snapshot of a past key moment at the end of a level (Race Games and Stunt Sims use this feature a lot). For this particular example, we will take a snapshot from within a framed region of the screen and print it on the top-right corner of the display.

Getting ready

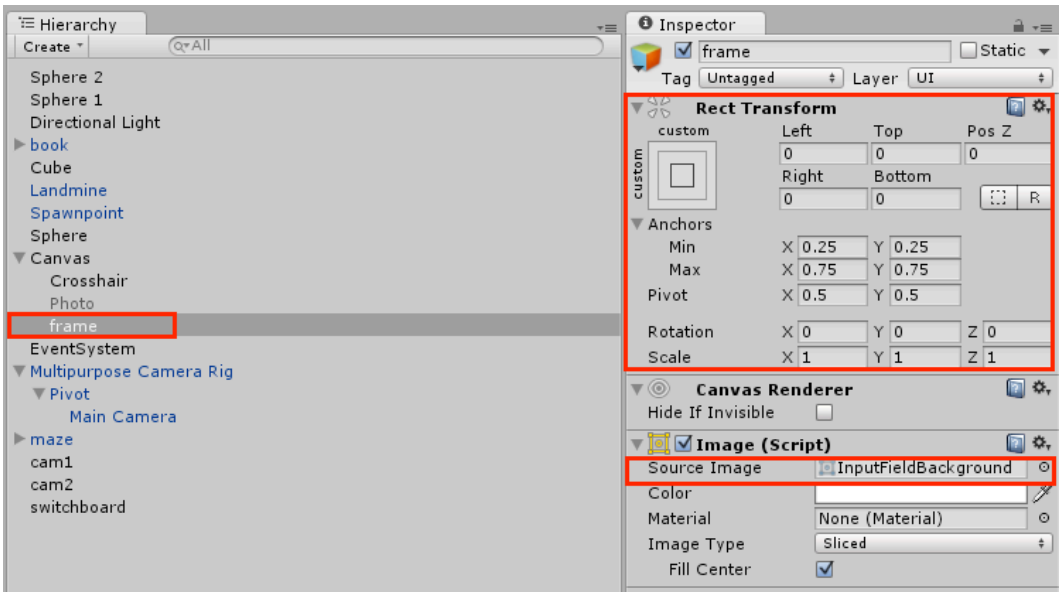
For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`. The package is in the `1362_05_codes` folder.

How to do it...

To create textures from screen content, follow these steps:

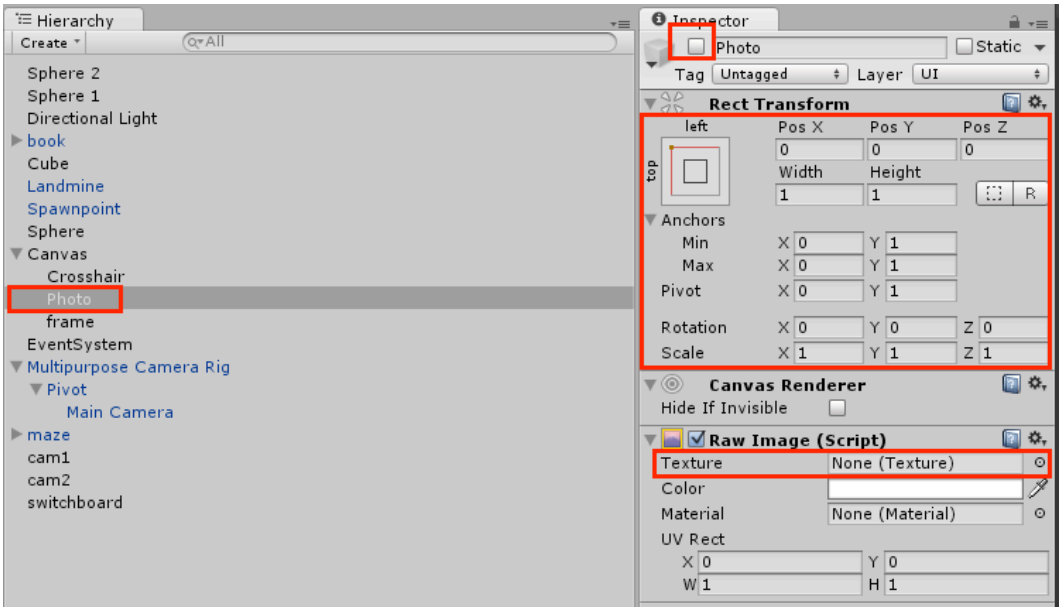
1. Import the `BasicScene` package into a new Project.
2. From the **Project** view, open the **mecanim** level. This is a basic scene featuring an animated character and some extra geometry. It also features a **Canvas** for UI elements.
3. Create a **UI Image** game object from the **Create** dropdown menu on top of the **Hierarchy** view (**Create | UI | Image**). Please note that it will be created as a child of the **Canvas** game object. Then, rename it `frame`.
4. From the **Inspector** view, find the **Image (Script)** component of the **frame** game object and set `InputFieldBackground` as its **Source Image**. This is a sprite that comes bundled with Unity, and it's already sliced for resizing purposes.

5. Now, from the **Inspector** view, change the **Rect Transform** to the following values: **Left**: 0; **Top**: 0; **Pos Z**: 0; **Right**: 0; **Bottom**: 0; **Anchors | Min | X**: 0.25, **Y**: 0.25; **Anchors | Max | X**: 0.75, **Y**: 0.75; **Pivot | X**: 0.5, **Y**: 0.5.



Insert image 1362OT_05_06.png

6. Create a **UI Raw Image** game object from the **Create** dropdown menu on top of the **Hierarchy** view (**Create | UI | RawImage**). Please note that it will be created as a child of the **Canvas** game object. Then, rename it `Photo`.
7. From the **Inspector** view, find the **Image (Script)** component of the **Photo** game object and set `None` as its **Source Image**. Also, from the top of the **Inspector**, disable the **Photo** game object by unchecking the box on the side of its name.
8. Now, from the **Inspector** view, change the **Rect Transform** to the following values: **Left**: 0; **Top**: 0; **Pos Z**: 0; **Width**: 1; **Height**: 1; **Anchors | Min | X**: 0, **Y**: 1; **Anchors | Max | X**: 0, **Y**: 1; **Pivot | X**: 0, **Y**: 1.



Insert image 1362OT_05_07.png

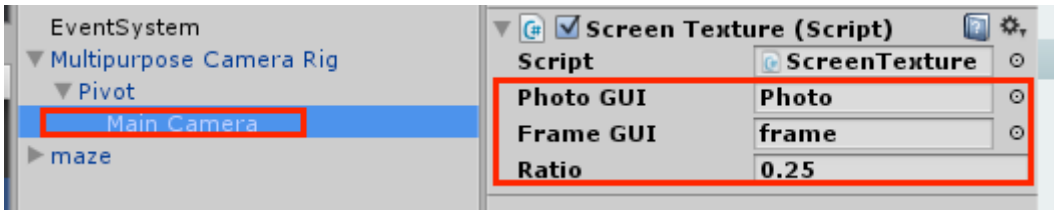
9. We need to create a script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it `ScreenTexture` and open it in your editor.

10. Open your script and replace everything with the following code:

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

public class ScreenTexture : MonoBehaviour {
    public GameObject photoGUI;
    public GameObject frameGUI;
    public float ratio = 0.25f;

    void  Update (){
        if (Input.GetKeyUp (KeyCode.Mouse0))
            StartCoroutine(CaptureScreen());
    }

    IEnumerator  CaptureScreen (){
        photoGUI.SetActive (false);
        int sw = Screen.width;
        int sh = Screen.height;
```

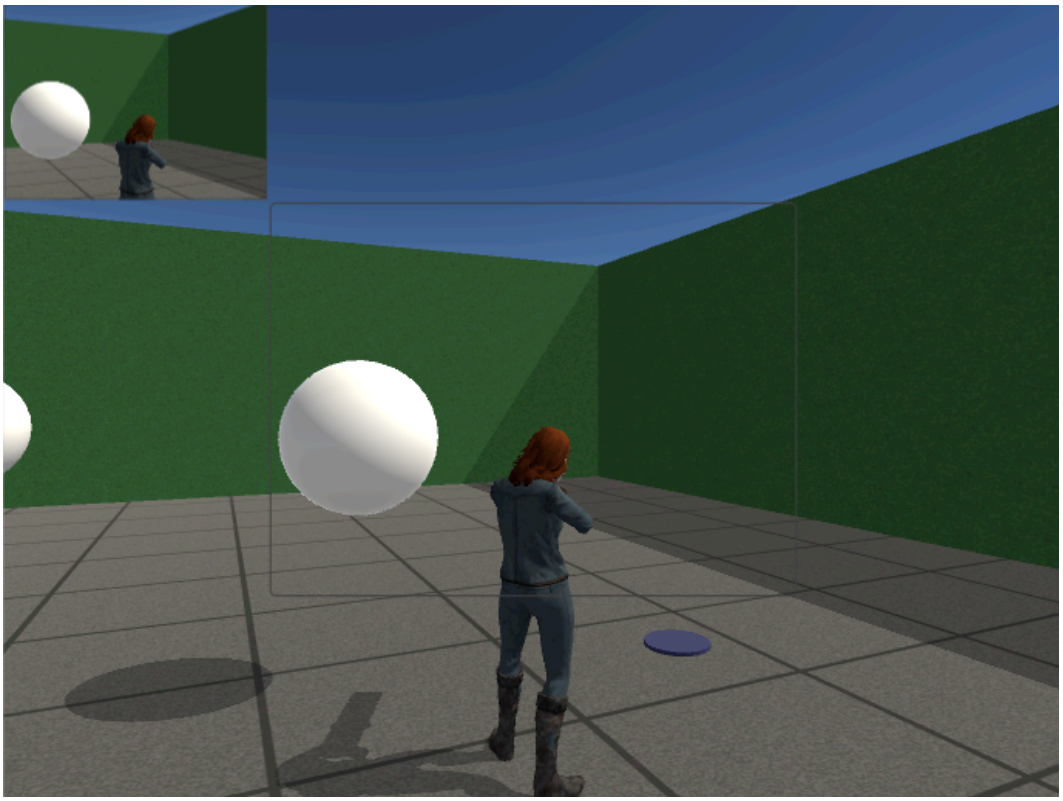
```
        RectTransform frameTransform =
frameGUI.GetComponent<RectTransform> ();
        Rect framing = frameTransform.rect;
        Vector2 pivot = frameTransform.pivot;
        Vector2 origin = frameTransform.anchorMin;
        origin.x *= sw;
        origin.y *= sh;
        float xOffset = pivot.x * framing.width;
        origin.x += xOffset;
        float yOffset = pivot.y * framing.height;
        origin.y += yOffset;
        framing.x += origin.x;
        framing.y += origin.y;
        int textWidth = (int)framing.width;
        int textHeight = (int)framing.height;
        Texture2D texture = new Texture2D(textWidth,textHeight);
        yield return new WaitForEndOfFrame();
        texture.ReadPixels(framing, 0, 0);
        texture.Apply();
        photoGUI.SetActive (true);
        Vector3 photoScale = new Vector3 (framing.width * ratio,
framing.height * ratio, 1);
        photoGUI.GetComponent<RectTransform> ().localScale =
photoScale;
        photoGUI.GetComponent<RawImage>().texture = texture;
    }
}
```

11. Save your script and apply it to the **Main Camera** game object within the **Multipurpose Camera Rig | Pivot** game object.
12. In the **Inspector** view, find the **Screen Texture** component and populate the fields **Photo GUI** and **Frame GUI** with the game objects **Photo** and **frame**, respectively.



Insert image 1362OT_05_08.png

13. Play the scene. You will be able to take a snapshot of the screen (and have it displayed on the top-left corner at a quarter of the original size) by clicking the mouse button.



Insert image 1362OT_05_09.png

How it works...

First, we have created a GUI frame from which to take a snapshot and a GUI element where to apply the texture. Then we have applied a script to the **Main Camera** to capture the screen content and apply it a new texture.

The script creates a new texture and captures the left mouse button being pressed, in which case it starts a co-routine to calculate a Rect area, copy screen pixels from that area, and apply them into a texture to be displayed by the **photo** GUI element, which is also resized to fit the texture.

The size of the Rect is calculated from the screen's dimension and the frame's Rect Transform settings, particularly its Pivot, Anchors, width, and height. The screen pixels

are then captured by the `ReadPixels()` command, and applied to the texture, which is then applied to the **Raw Image** photo, which is resized to meet the desired ratio between photo size / original pixels.

There's more...

Apart from displaying the texture as a GUI element, you could use it in other ways.

Applying your texture to a material

You could apply your texture to an existing object's material by adding a line similar to `GameObject.Find("MyObject").renderer.material.mainTexture = texture;` to the end of the `CaptureScreen` function.

Use your texture as a screenshot

You can encode your texture as a PNG image file and save it. Check out Unity's documentation on this feature at `http://docs.unity3d.com/Documentation/ScriptReference/Texture2D.EncodeToPNG.html`.

Zooming a telescopic camera

In this recipe, we will create a telescopic camera that zooms in whenever the left mouse button is pressed. This can be very useful, for instance, if we have a sniper in our game.

Getting ready...

For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`. The package is in the `1362_05_codes` folder.

How to do it...

To create a telescopic camera, follow these steps:

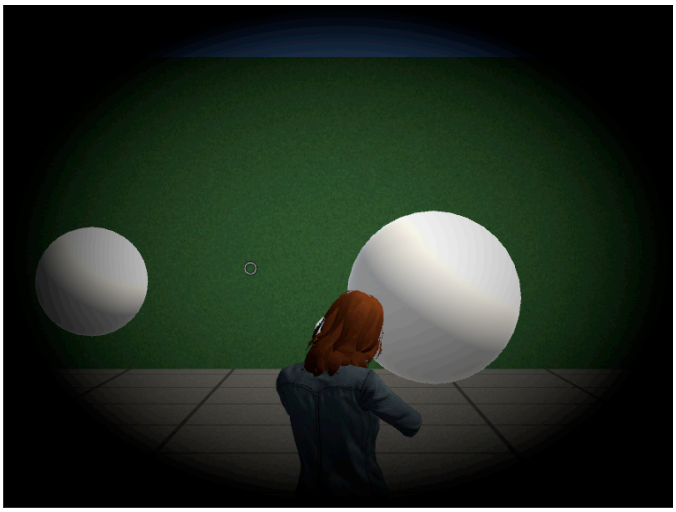
1. Import the `BasicScene` package into a new Project.
2. From the **Project** view, open the **mecanim** level. This is a basic scene featuring an animated character and some extra geometry.
3. We need to create a script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it `TelescopicView` and open it in your editor.
4. Open your script and replace everything with the following code:

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

```
public class TelescopicView : MonoBehaviour{
   public float zoom = 2.0f;
   public float speedIn = 100.0f;
   public float speedOut = 100.0f;
   private float initFov;
   private float currFov;
   private float minFov;
   private float addFov;
   //private VignetteAndChromaticAberration v;
   //public float vMax = 10.0f;

   void Start(){
      initFov = Camera.main.fieldOfView;
      minFov = initFov / zoom;
      //v = this.GetComponent<VignetteAndChromaticAberration>()
as VignetteAndChromaticAberration;
   }
   void Update(){
      if (Input.GetKey(KeyCode.Mouse0))
         ZoomView();
      else
         ZoomOut();
      //float currDistance = currFov - initFov;
      //float totalDistance = minFov - initFov;
      //float vMultiplier = currDistance / totalDistance;
      //float vAmount = vMax * vMultiplier;
      //vAmount = Mathf.Clamp (vAmount,0,vMax);
      //v.intensity = vAmount;
   }

   void ZoomView(){
      currFov = Camera.main.fieldOfView;
      addFov = speedIn * Time.deltaTime;

      if (Mathf.Abs(currFov - minFov) < 0.5f)
         currFov = minFov;
      else if (currFov - addFov >= minFov)
         currFov -= addFov;

      Camera.main.fieldOfView = currFov;
   }

   void ZoomOut(){
      currFov = Camera.main.fieldOfView;
```

```
            addFov = speedOut * Time.deltaTime;

            if (Mathf.Abs(currFov - initFov) < 0.5f)
                currFov = initFov;
            else if (currFov + addFov <= initFov)
                currFov += addFov;

            Camera.main.fieldOfView = currFov;
        }
    }
```

5. Save your script and apply it to the **Main Camera** game object within the **Multipurpose Camera Rig | Pivot** game object.
6. Play your scene. You should be able to zoom in the camera view by clicking the mouse button and zoom out by releasing it.
7. The next steps are for those using Unity Pro. If you are using Unity Pro, please stop the scene.
8. Import Unity's **Image Effects** package by navigating to **Assets | Import Package | Image Effects (Pro Only)**.
9. Select the **Main Camera** game object within the **Multipurpose Camera Rig | Pivot** game object and apply the **Vignette** image effect (by navigating to **Component | Image Effects | Camera | Vignette and Chromatic Aberration**).
10. Open the script `TelescopicView` and uncomment every commented line.
11. Save the script and play the level. You should see an animated vignette effect in addition to the zooming.



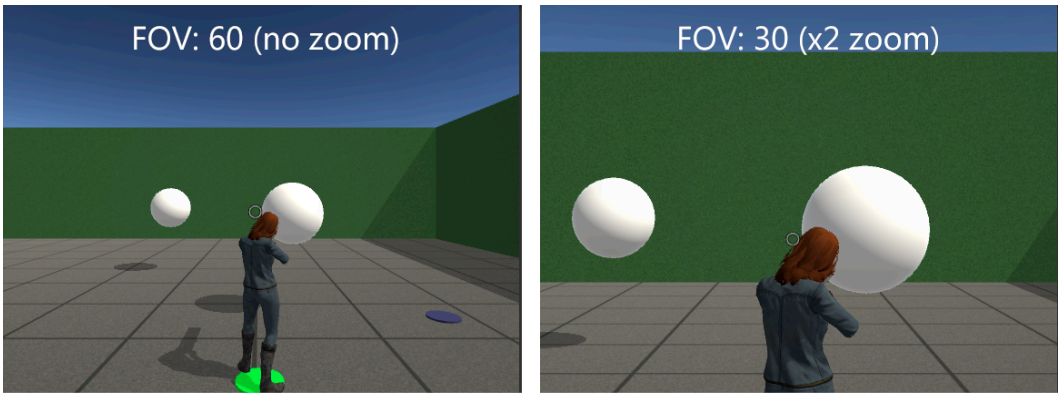
Insert image 1362OT_05_10.png

How it works...

The zooming effect is actually caused by changes on the value of the camera's **Field Of View** property: small values result in a closer view of a smaller area, while high values enlarge the **Field of View** (**FOV**).

The `TelescopicView` script changes the camera's Field of View by subtracting from it whenever the left mouse button is pressed. It also adds up to the FOV value when the mouse button is *not* being held, until it reaches its original value.

The zoom limit, in FOV, can be deduced from the code: `minFov = initFov / zoom;`. That means that the minimum value of FOV is equal to its original value divided by the zoom amount. For instance: if our camera features, originally, an FOV of 60, and we set the **Telescopic View Zoom** amount as `2.0`, the minimum FOV allowed will be 60 / 2 = 30.



Insert image 1362OT_05_11.png

There's more...

If you are using Unity Pro, you could also add a variable to control the **Blur Vignette** level of the **Vignette** image effect.

Displaying a mini-map

In many games, a broader view of the scene can be invaluable for navigation and information. Mini-maps are great for giving players that extra perspective they might need when in first or third-person mode.

Getting ready...

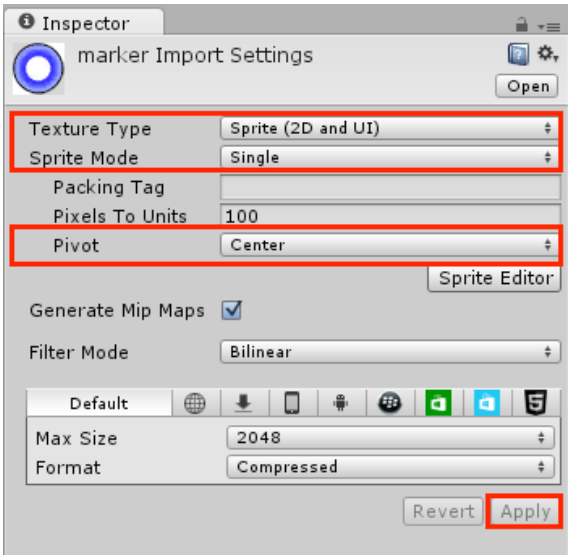
For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`. The package is in the `1362_05_codes` folder. You will also need to import a texture file named `marker.png` and the `PictureInPicture.cs` script. Both are available inside the `1362_05_05` folder.

> The `PictureInPicture` script will be used to calculate the size and position of the mini-map to be displayed on top of the game's main camera. We encourage you to learn about it in detail by going through the first recipe in this chapter, *Creating a picture-in-picture effect*, in case you haven't done it yet.

How to do it...

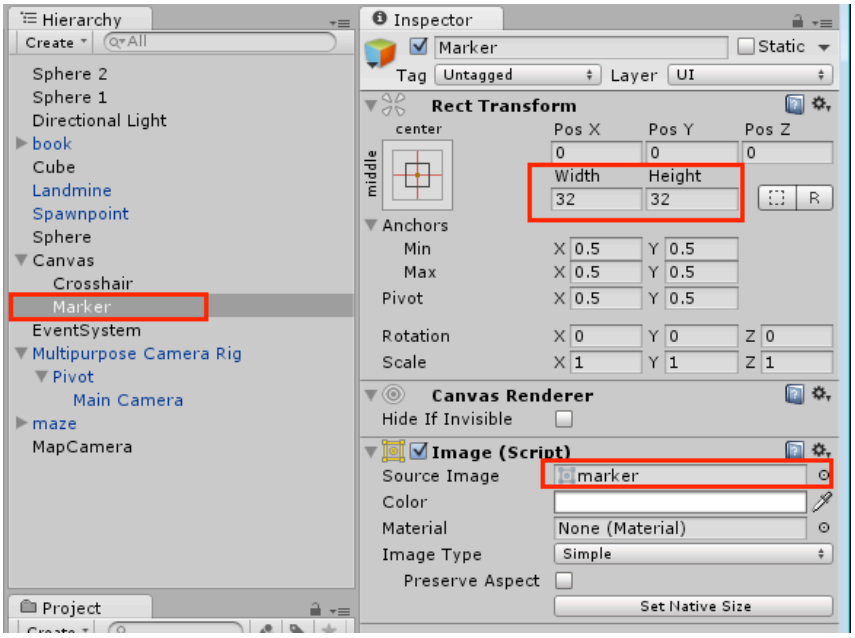
To create a mini-map, follow these steps:

1. Import the `BasicScene` package into a new Project. Also, import the `PictureInPicture.cs` script and the `marker.png` texture.
2. From the **Project** view, select the **marker** texture. Then, from the **Inspector**, change its **Texture Type** to **Sprite (2D and UI)**, leaving **Sprite Mode** as **Single** and **Pivot** at **Center**. Click **Apply** to confirm changes.



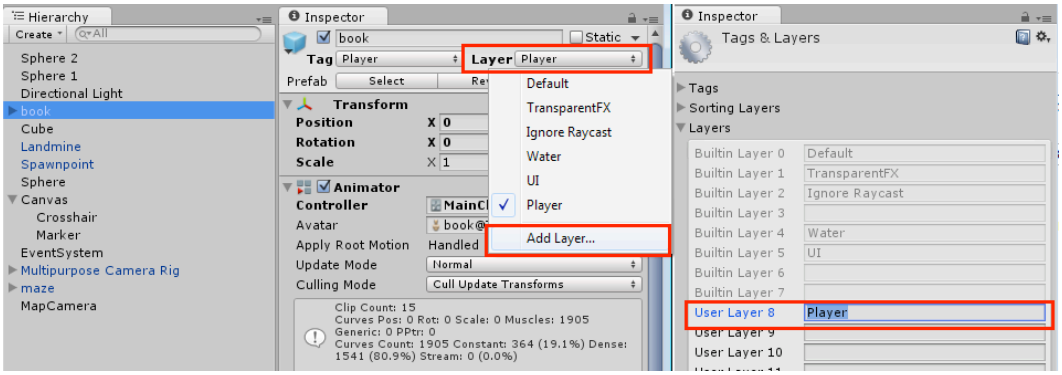
Insert image 1362OT_05_12.png

3. From the **Hierarchy** view, create a new UI Image object (**Create | UI | Image**). It will be created as a child of the UI **Canvas** game object. Rename it **Marker**. Then, from the **Inspector** view, change its **Width** and **Height** to 32. Finally, populate the **Source Image** field, within the **Image** component, with the **marker** sprite.



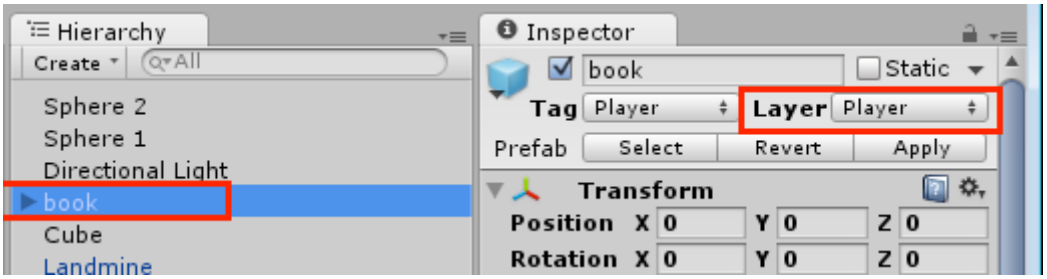
Insert image 1362OT_05_17.png

4. Select the **book** game object (which is the player's character) and, from the top of the **Inspector** view, access the **Layer** dropdown menu. Select **Add Layer….** Then, name a **User Layer** as `Player`.



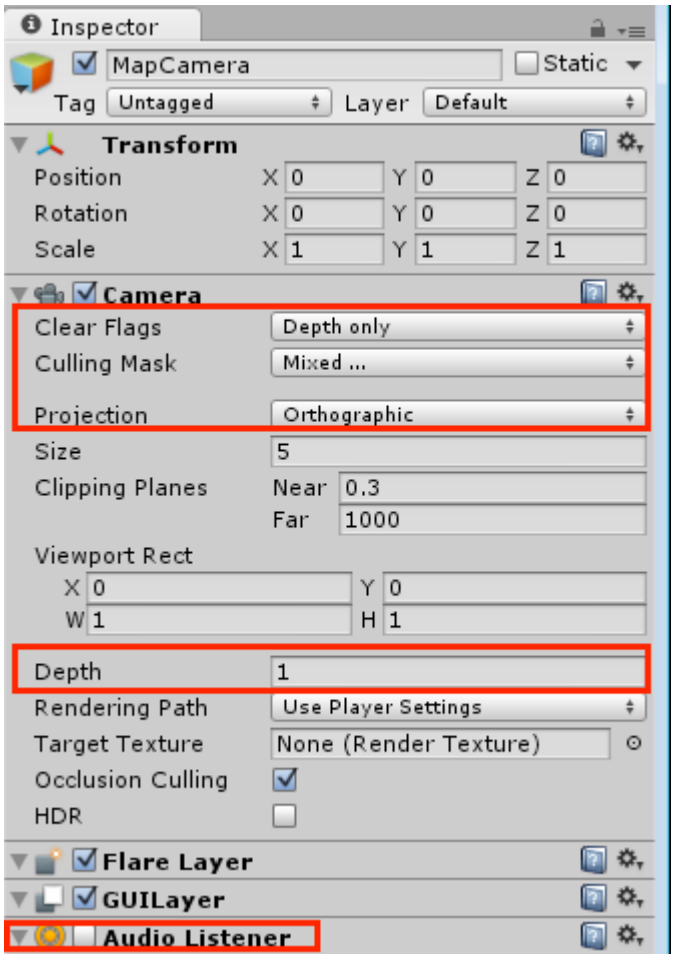
Insert image 1362OT_05_13.png

5. Select the book character again and, from the **Layer** dropdown menu, select **Player**.



Insert image 1362OT_05_14.png

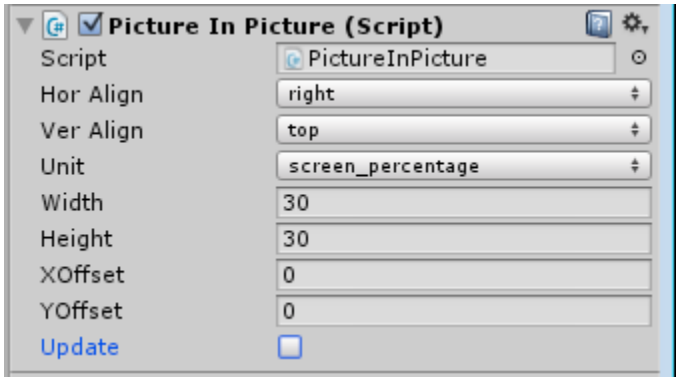
6. From the **Hierarchy** view, create a new camera (**Create | Camera**). Rename it **MapCamera** and, from the **Inspector** view, change its parameters as it follows:
    - **Clear Flags**: **Depth Only**
    - **Culling Mask**: **Mixed…** (unselect **Player**)
    - **Projection**: **Orthographic**
    - **Depth**: **1** (or higher)
    - Also, uncheck the camera's **Audio Listener** Component)



Insert image 1362OT_05_15.png

7. Attach the `PictureInPicture` script, previously imported, to the **MapCamera**. Then, from the **Inspector** view, adjust the parameters of the **Picture In Picture** component as it follows:
    - **Hor Align: Right**
    - **Ver Algin: Bottom**
    - **Unit: screen_percentage**
    - **Width:** 30
    - **Height:** 30
    - **XOffset:** 0

- **YOffset:** 0
- **Update:** Unchecked



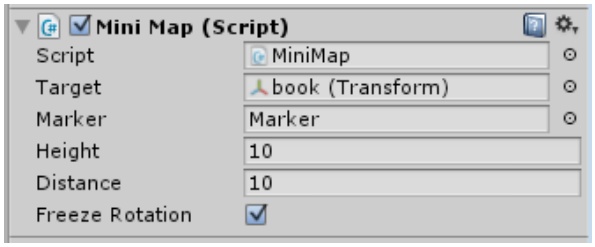
Insert image 1362OT_05_16.png

8. From the **Project** view, create a new **C# Script** and name it `MiniMap`. Open it and replace everything with the following code:

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

public class MiniMap : MonoBehaviour
{
    public Transform target;
    public GameObject marker;
    public float height = 10.0f;
    public float distance = 10.0f;
    public bool freezeRotation = true;
Euler angles
    private Vector3 camAngle;
    private Vector3 camPos;
    private Vector3 targetAngle;
    private Vector3 targetPos;
    private Camera cam;

    void Start(){
        cam = GetComponent<Camera> ();
        camAngle = transform.eulerAngles;
        targetAngle = target.transform.eulerAngles;
        camAngle.x = 90;
```

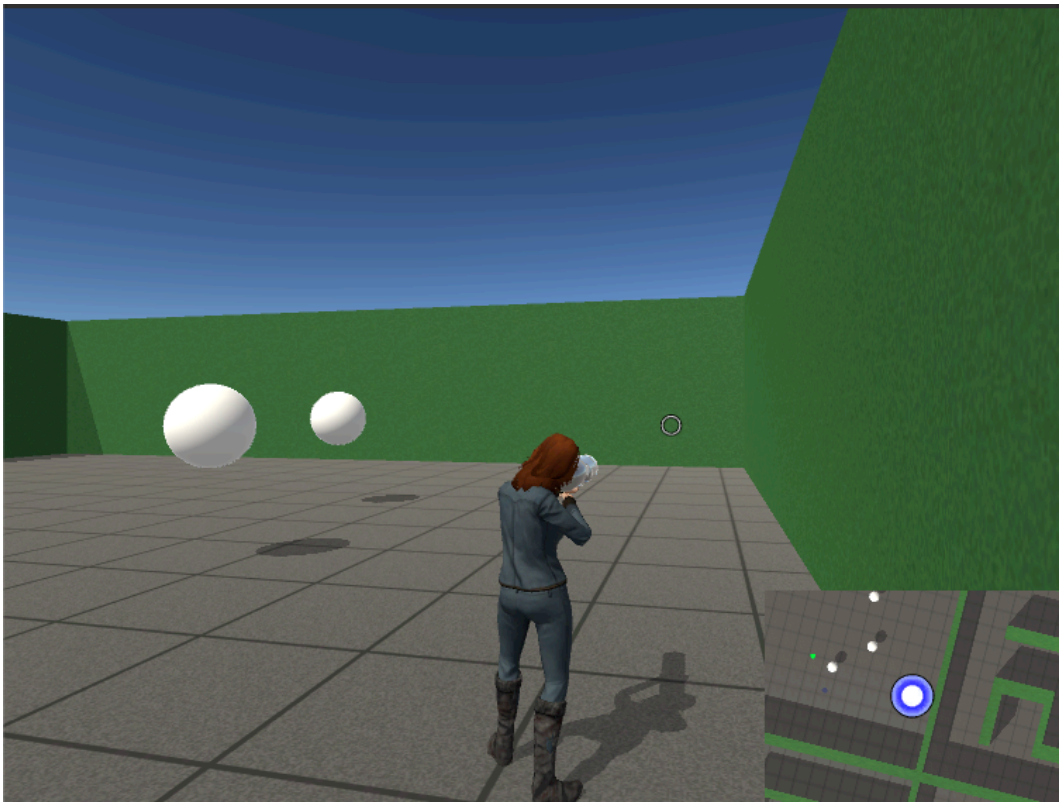
```
        camAngle.y = targetAngle.y;
        transform.eulerAngles = camAngle;
    }

    void Update(){
        targetPos = target.transform.position;
        camPos = targetPos;
        camPos.y += height;
        transform.position = camPos;
        cam.orthographicSize = distance;

        if (freezeRotation){
            camAngle.y = target.transform.eulerAngles.y;
            transform.eulerAngles = camAngle;
        }

        targetPos = cam.WorldToScreenPoint (targetPos);
        marker.GetComponent<RectTransform> ().position = targetPos;
    }
}
```

9. Save the script and attach it to the **MapCamera**. Then, from the **Inspector** view, change parameters of the **Mini Map** component as it follows:
    - **Target**: **book**
    - **Marker**: **Marker** (the UI Image previously created)
    - **Height**: **10**
    - **Distance**: **10**
    - **Freeze Rotation:** Checked



Insert image 1362OT_05_18.png

10. Play the scene. You should be able to see the mini-map functioning in the bottom-right corner of the screen.



Insert image 1362OT_05_19.png

How it works...

First, some necessary adjustments were made to the **MapCamera**:

- Setting its **Depth** to a higher value (so its viewport would be displayed on top of the main camera)
- Changing its **Projection** mode to **Orthographic** (to make it bi-dimensional)
- Excluding the **Player** tag from its **Culling Mask** (making the character's model invisible to the camera)
- Disabling its **Audio Listener** (so it won't conflict with the main camera's)

Second, in order to resize and position the **MapCamera's** viewport, we have attached a **Picture In Picture** component to it. Again, you can learn everything about the **Picture In Picture** component by going through the first recipe in this chapter.

Finally, we created the **MiniMap** class. Basically, what it does is position and rotate the **MapCamera** according to a target's transform settings. Then, it places a selected GUI marker at the screen position where the target object should be. The range of the mini-map is given by the **Distance** parameter: a higher value will result in the coverage of a wider area, as the **MiniMap** class uses this same value as the viewport size of the orthographic camera.

There's more...

If you want to experiment more with your mini-map, read on.

Using Render Texture (Pro Only)

If you use Unity Pro, you could always use **Render Texture** and display your map using a **RawImage** within the UI **Canvas**. For more information on Render Texture, please check Unity documentation at `docs.unity3d.com/Manual/class-RenderTexture.html`.

Adapting your mini-map to other styles

You could easily modify this recipe to make a top or isometric view of a racing game circuit map. Just position the camera manually and prevent it from following the character. Also, don't forget to add markers for all vehicles!

See also

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

- *Creating a picture-in-picture effect*

Creating an in-game surveillance camera

Although using a second viewport can be useful in many situations, there will be times when you need to output the image rendered from a camera to a texture at runtime. To illustrate this point, in this recipe we will make use of a **Render Texture** to create an in-game surveillance camera that transmits its video to a monitor.

Insert image 1362OT_05_24.png

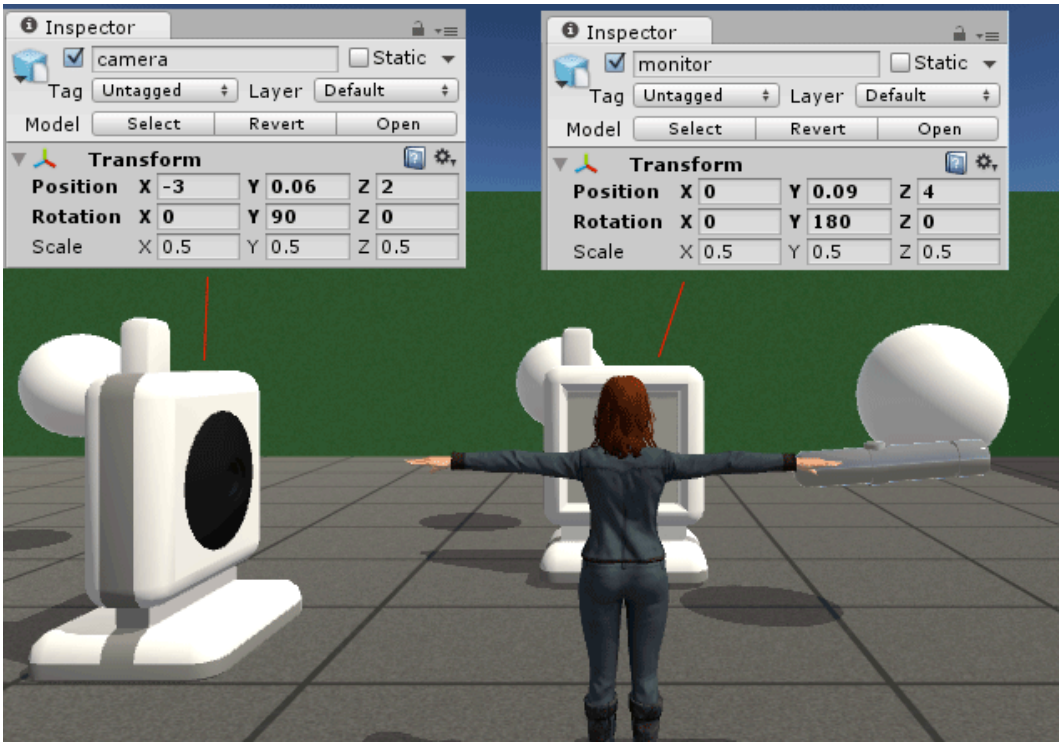
Getting ready

For this recipe, we have prepared the `BasicScene` Unity package, containing a scene named `mecanim`, and also two FBX 3D models for the monitor and camera objects. The package is in the `1362_05_codes` folder, and the 3DD models, inside the `1362_05_06` folder.

How to do it...

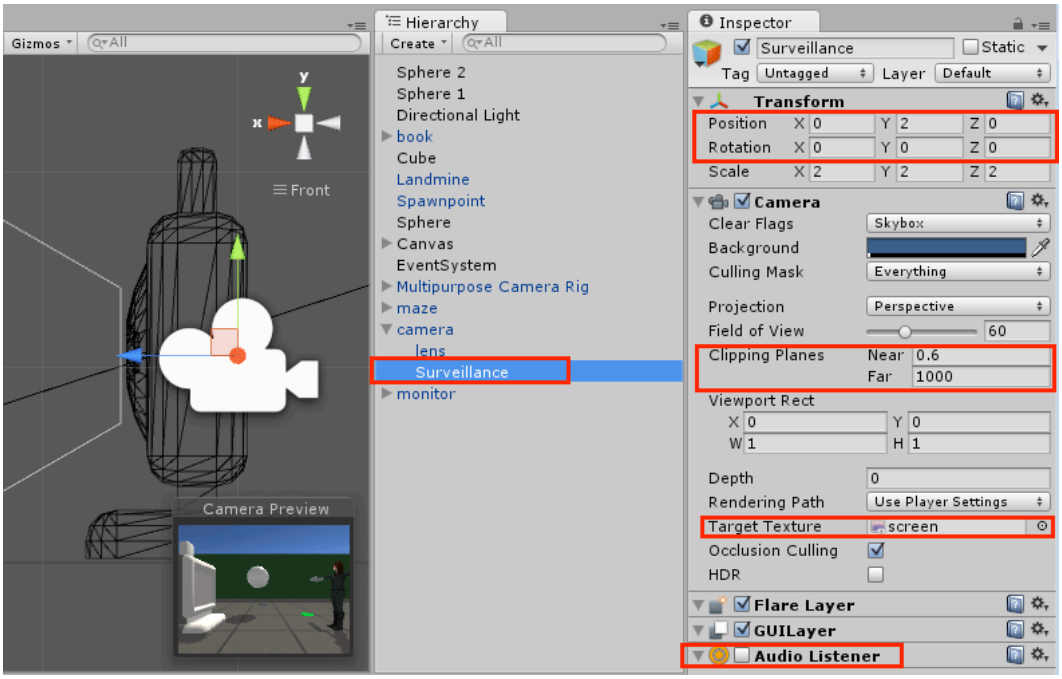
To create a picture-in-picture display, just follow these steps:

1. Import the `BasicScene` package and the `monitor` and `camera` models into your Unity Project.
2. From the **Project** view, open the **mecanim** level. This is a basic scene featuring an animated character and some extra geometry.
3. From the **Project** view, place the **monitor** and **camera** objects into the scene by dragging them into the **Hierarchy**. Their transform settings should be: **monitor: Position**: **X**: `0`; **Y**: `0.09`; **Z**: `4`. **Rotation**: **X**: `0`; **Y**: `180`; **Z**: `0`. **camera: Position**: **X**: `-3`; **Y**: `0.06`; **Z**: `4`. **Rotation**: **X**: `0`; **Y**: `90`; **Z**: `0`.



Insert image 1362OT_05_20.png

4. Create, from the **Project** view, a new **Render Texture** and rename it as `screen`. Then, from the **Inspector** view, change its **Size** to `512` x `512`.
5. Add a new **Camera** to the scene through the **Create** dropdown menu on top of the **Hierarchy** view (**Create | Camera**). Then, from the **Inspector** view, Name it `Surveillance` and make it a child of the **camera** game object. Then, change its transform settings to the following: **Position**: **X**: `0`; **Y**: `2`; **Z**: `0`. **Rotation**: **X**: `0`; **Y**: `0`; **Z**: `0`.
6. Select the `Surveillance` camera you have created and, From the **Inspector** view, change its **Clipping Planes | Near** to `0.6`. Also, populate the **Target Texture** slot with the **screen** Render Texture and disable the camera's **Audio Listener** component.



Insert image 1362OT_05_21.png

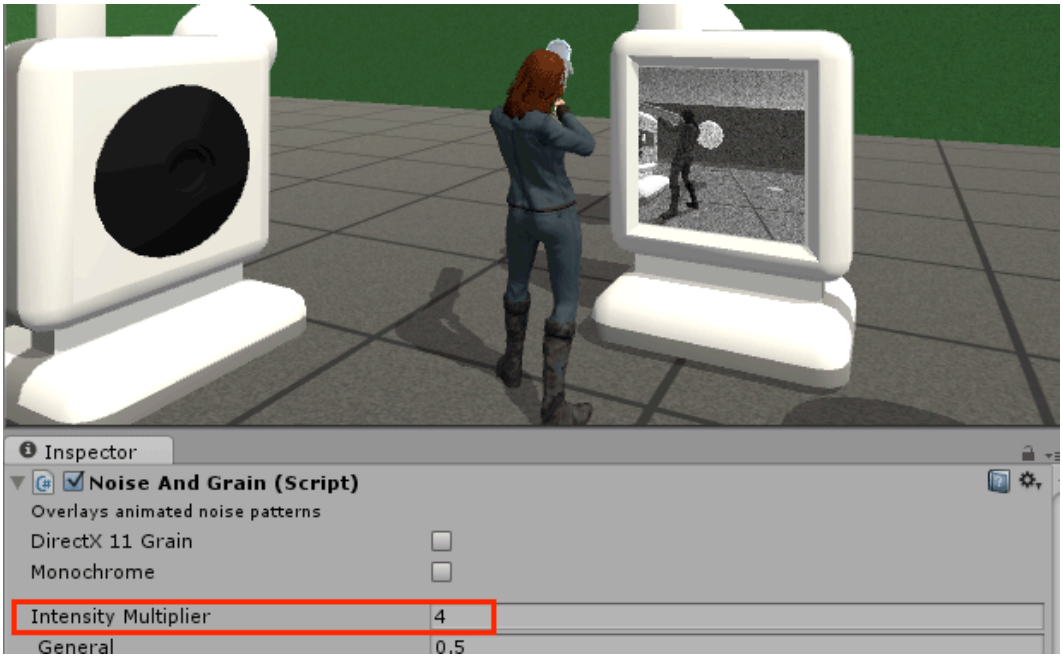
7. From the **Hierarchy** view, expand the **monitor** object and select its **screen** child. Then, from the **Inspector**, find its material (named **Desert**) and, from the **Shader** dropdown menu, change to **Unlit/Texture**. Finally, set the **screen** texture as its base texture.

Insert image 1362OT_05_22.png

8. Now it's time to add some post-processing to the texture. From the main menu, import the **Image Effects** package (**Assets | Import Package | Image Effects (Pro Only)**).
9. From the **Hierarchy**, select the `Surveillance` camera. Then, from the main menu, add the **Grayscale** Image Effect component (**Component | Image Effects**

| Color Adjustments | Grayscale). Also, add the **Noise And Grain** Image Effect (**Component | Image Effects | Noise | Noise and Grain (Filmic)**). Finally, from the **Inspector** view, set the **Intensity Multiplier** of the **Noise And Grain** as 4.

10. Play your scene. You should be able to see your actions in real time on the monitor's screen.



Insert image 1362OT_05_23.png

How it works...

We have achieved the final result by using the **Surveillance** camera as source for the **Render Texture** applied to the **screen**. The camera was made a child of the camera's 3DD model, for easier relocation. Also, its Near Clipping plane was readjusted in order to avoid displaying parts of the camera's 3D model geometry, and its **Audio Source** component, disabled so it wouldn't clash with the Main Camera's component.

In addition to setting up the **Surveillance** camera, two **Image Effects** were added to it: **Noise And Grain** and **Greyscale**. Together, these effects should make the **Render Texture** look closer to a cheap monitor's screen.

Finally, our **screen** render texture is applied to the screen's 3D object's material (which had its shader changed to **Unlit/texture**, so it can be seen on low/no light conditions, as a monitor should).