6
Lights and Effects

In this chapter, we will cover:

- Using lights and cookie textures to simulate a cloudy outdoor
- Adding a custom Reflection map to a scene
- Creating a laser aim with Projector and Line Renderer
- Reflecting surrounding objects with Reflection Probes
- Setting up an environment with Procedural Skybox and Directional Light
- Lighting a simple scene with Lightmaps and Light Probes

Introduction

Whether you're willing to make a better looking game or add interesting features, lights and effects can boost your project and help you deliver a higher quality product. In this chapter, we will look at creative ways of using lights and effects, and also take a look at some of Unity's new features such as **Procedural Skyboxes**, **Reflection Probes**, **Light probes**, and Custom **Reflection Sources**.

Lighting is certainly an area that has received a lot of attention from Unity, which now features real-time **Global Illumination** technology provided by **Enlighten**. This new technology provides better, more realistic results for both Realtime and Baked lighting. For more information on Unity's Global Illumination system, check its documentation at `docs.unity3d.com/Manual/GIIntro.html`.

The Big Picture

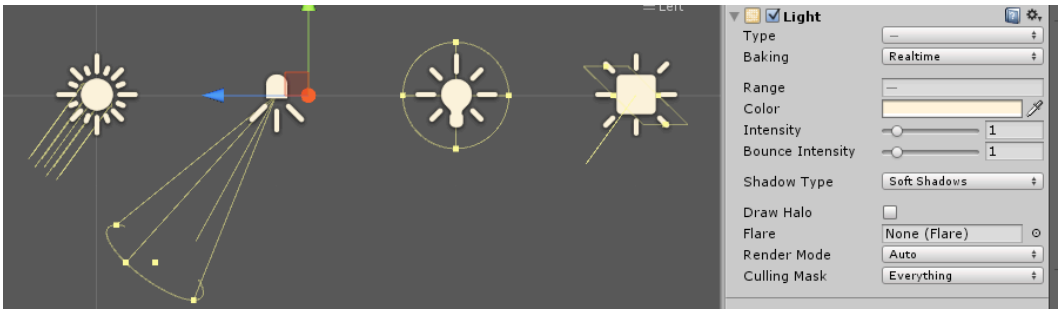
There are many ways of creating light sources in Unity. Here's a quick overview of the most common methods:

Lights

Placed into the scene as game objects featuring a **Light** component, Lights can function in **Realtime**, **Baked**, or **Mixed** modes. Among other properties, they can have their **Range**, **Color**, **Intensity**, and **Shadow Type** set by the user. There are four types of lights:

- **Directional Light**: Normally used to simulate the sunlight
- **Spot Light**: Works like a cone-shaped spot light
- **Point Light**: Bulb lamp-like, omnidirectional light
- **Area Light**: This baked-only light type is emitted in all directions from a rectangle-shaped entity, allowing for smooth, realistic shading

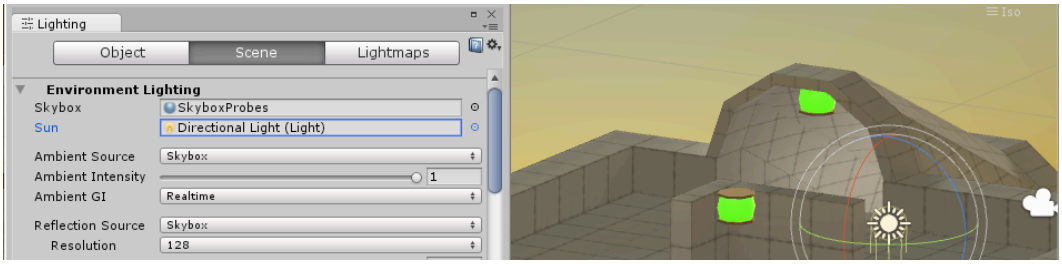
For an overview of light types, check Unity's documentation at docs.unity3d.com/Manual/Lighting.html.



Insert image 1362OT_06_50.png

Environment Lighting

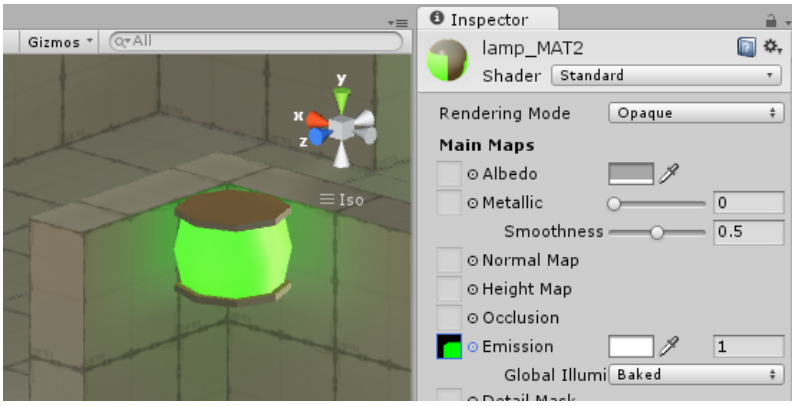
Unity's **Environment Lighting** is often achieved through the combination of a **Skybox** material and a sunlight defined by the scene's **Directional Light**. Such combination creates an ambient light that is integrated into the scene's environment, and which can be set as **Realtime** or **Baked into Lightmaps**.



Insert image 1362OT_06_51.png

Emissive Materials

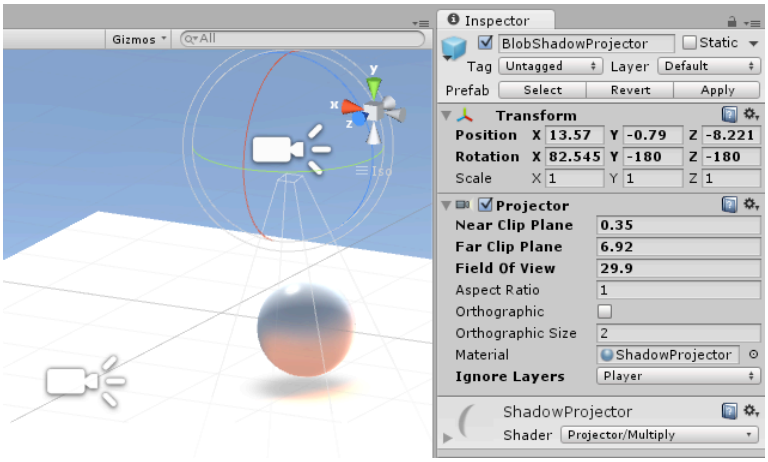
When applied to **static** objects, materials featuring **Emission** colors or maps will cast light over surfaces nearby, in both Realtime and Baked modes.



Insert image 1362OT_06_52.png

Projector

As its name suggests, a **Projector** can be used to simulate projected lights and shadows, basically by projecting a material, and its texture map, onto other objects.



Insert image 1362OT_06_53.png

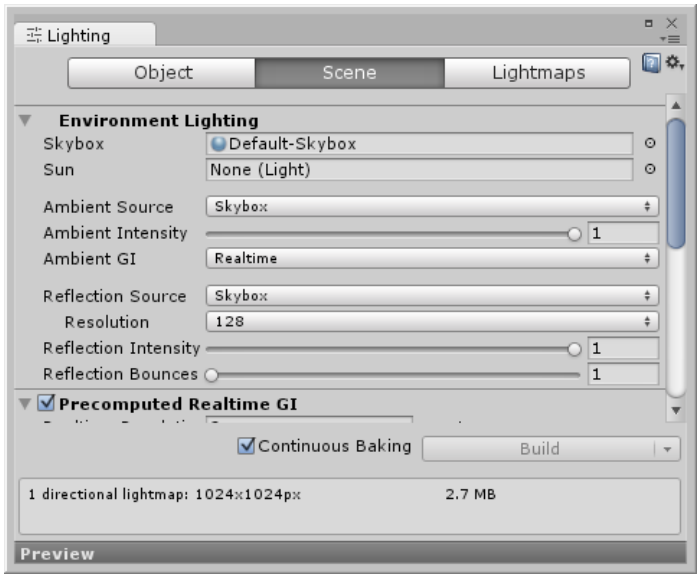
Lightmaps and Light Probes

Lightmaps are basically texture maps generated from the scene's lighting information and applied to the scene's static objects in order to avoid the use of processing-intensive realtime lighting.

Light Probes are a way of sampling the scene's illumination at specific points in order to have it applied onto dynamic objects without the use of realtime lighting.

The Lighting window

The **Lighting** window, which can be found through the **Window | Lighting** menu, is the hub for setting and adjusting the scene's illumination features such as Lightmaps, Global Illumination, Fog, and much more. It's strongly recommended that you take a look at Unity's documentation on the subject, which can be found at: `docs.unity3d.com/Manual/GlobalIllumination.html`.



Insert image 1362OT_06_54.png

Using lights and cookie textures to simulate a cloudy outdoor

As it can be seen in many first person shooters and survival horror games, lights and shadows can add a great deal of realism to a scene, helping immensely to create the right atmosphere to the game. In this recipe, we will create a cloudy outdoor environment

lighting using cookie textures. Cookie textures work as masks for lights. It functions by adjusting the intensity of the light projection to the cookie texture's alpha channel. This allows for a silhouette effect (just think of the bat-signal) or, as in this particular case, subtle variations that give a filtered quality to the lighting.

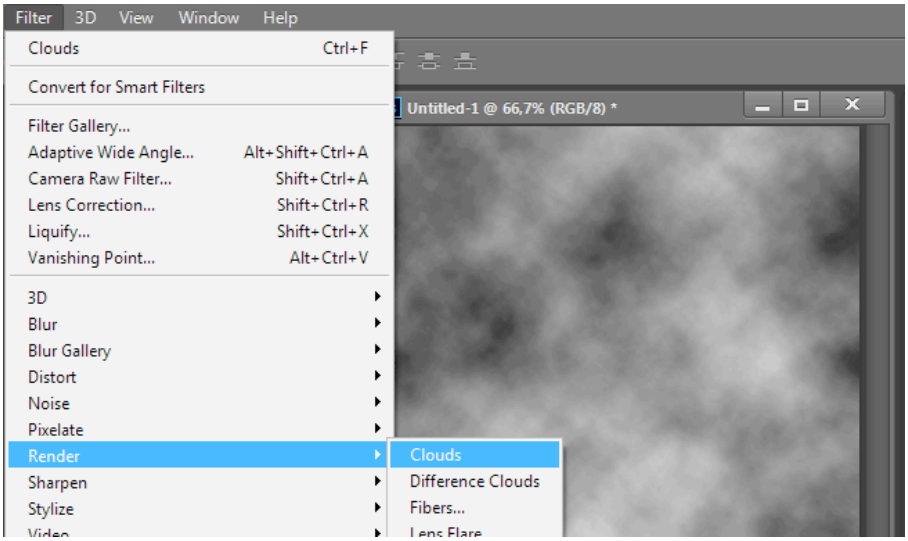
Getting ready

If you don't have access to an image editor, or prefer to skip the texture map elaboration in order to focus on the implementation, please use the image file `cloudCookie.tga` provided inside the `1362_06_01` folder.

How to do it...

To simulate a cloudy outdoor environment, follow these steps:

1. In your image editor, create a new 512 x 512 pixel image.
2. Using black as foreground color and white as background color, apply the **Render Clouds** filter (in Photoshop, this is done through the menu **Filter | Render | Clouds**):

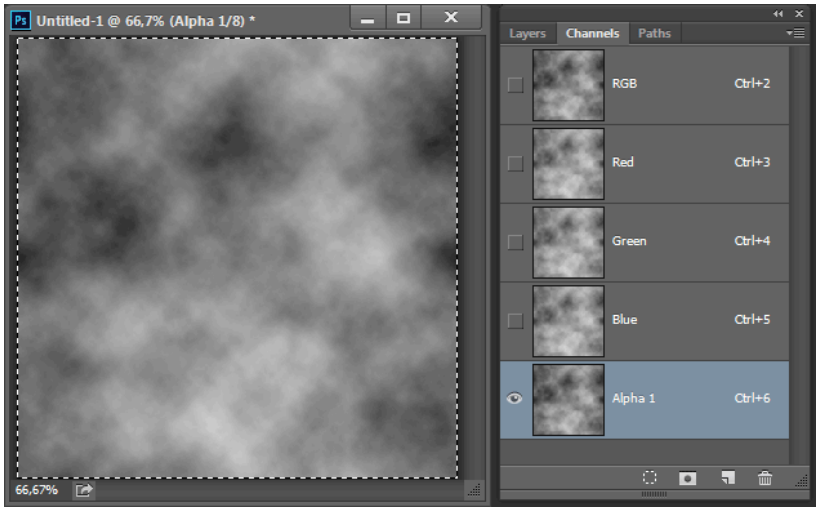


Insert image 1362OT_06_01.png

> Image editors usually have a filter or command that renders clouds. If your image editor doesn't have such capability you can either paint it

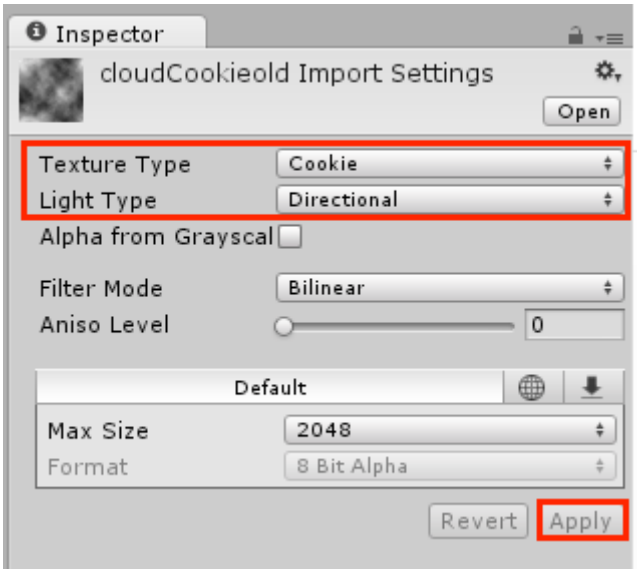
manually or use the image file `cloudCookie.tga` provided in the `1362_06_01` folder).

3. Select your entire image and copy it.
4. Open the **Channels** window (in Photoshop this can be done through the menu **Window | Channels**).
5. There should be three channels: **Red**, **Green**, and **Blue**. Create a new channel. That will be the **Alpha** channel.
6. In the **Channels** window, select the Alpha channel and paste your image into it.



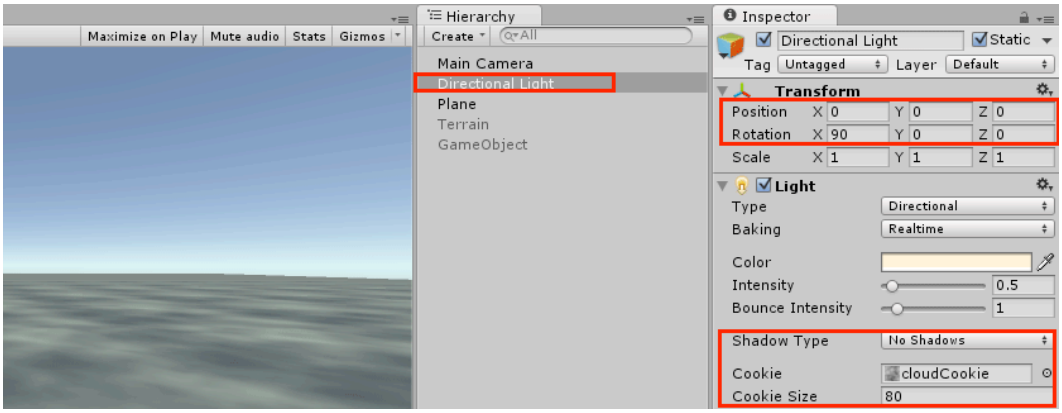
Insert image 1362OT_06_02.png

7. Save your image file as `cloudCookie.PSD` or `TGA`.
8. Import your image file into Unity and select it in the **Project** view.
9. From the **Inspector** view, change its **Texture Type** to **Cookie** and its **Light Type** to **Directional**. Then, click **Apply**.



Insert image 1362OT_06_03.png

10. We will need a surface to actually see the lighting effect. You can either add a plane to your scene (via the menu **GameObject | 3D Object | Plane**) or create a **Terrain** (menu option menu **GameObject | 3D Object | Terrain**) and edit it, if so you wish.
11. Let's add a light to our scene. Since we want to simulate sunlight, the best option is to create a **Directional Light**. You can do that through the dropdown menu named **Create | Light | Directional Light** in the **Hierarchy** view.
12. Using the **Transform** component of the **Inspector** view, reset the light's **Position** to **X:** 0; **Y:**0; **Z:**0 and its **Rotation** to **X:** 90; **Y:**0; **Z:**0.
13. In the **Cookie** field, select the **cloudCookie** texture you have imported earlier. Change the field **Cookie Size** to 80, or a value that you feel as more appropriate to the scene's dimension. Please leave the **Shadow Type** as **No Shadows**.



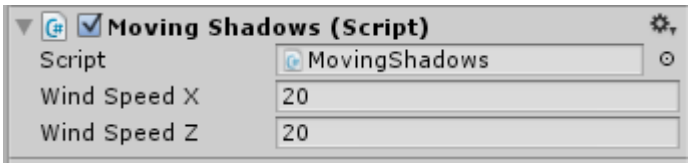
Insert image 1362OT_06_04.png

14. Now, we need a script to translate our light and, consequently, the **Cookie** projection. Using the **Create** dropdown menu in the **Project** view, create a new C# Script named `MovingShadows.cs`.
15. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class MovingShadows : MonoBehaviour{
   public float windSpeedX;
   public float windSpeedZ;
   private float lightCookieSize;
   private Vector3 initPos;

   void Start(){
      initPos = transform.position;
      lightCookieSize = GetComponent<Light>().cookieSize;
   }

   void Update(){
      Vector3 pos = transform.position;
      float xPos= Mathf.Abs (pos.x);
      float zPos= Mathf.Abs (pos.z);
      float xLimit = Mathf.Abs(initPos.x) + lightCookieSize;
      float zLimit = Mathf.Abs(initPos.z) + lightCookieSize;

      if (xPos >= xLimit)
         pos.x = initPos.x;
```

```
            if (zPos >= zLimit)
               pos.z = initPos.z;

            transform.position = pos;
            float windX = Time.deltaTime * windSpeedX;
            float windZ = Time.deltaTime * windSpeedZ;
            transform.Translate(windX, 0, windZ, Space.World);
        }
    }
```

16. Save your script and apply it to the **Directional Light**.
17. Select the **Directional Light**. In the **Inspector** view, change the parameters **Wind Speed X** and **Wind Speed Z** to `20` (you can change these values as you wish).



Insert image 1362OT_06_05.png

18. Play your scene. The shadows should be moving.

How it works...

With our script, we are telling the **Directional Light** to move across the X and Z axis, causing the **Light Cookie** texture to be displaced as well. Also, we reset the light object to its original position whenever it travels a distance equal to or greater than the **Light Cookie Size**. The light position must be reset to prevent it from travelling too far, causing problems in real time render and lighting. The **Light Cookie Size** parameter is used to ensure a smooth transition.

The reason we are not enabling shadows is because the light angle for the X axis must be 90 degrees (or there will be a noticeable gap when the light resets to the original position). If you want dynamic shadows in your scene, please add a second Directional Light.

There's more...

In this recipe we have applied a cookie texture to a **Directional Light**. But what if we were using **Spot** or **Point Lights**?

Creating Spot Light Cookies

Unity documentation has an excellent tutorial on how to make **Spot Light** cookies. This is great to simulate shadows coming from projectors, windows, and so on. You can check it out at `docs.unity3d.com/Manual/HOWTO-LightCookie.html`.

Creating Point Light Cookies

If you want to use a cookie texture with a **Point Light**, you'll need to change the **Light Type** in the **Texture Importer** section of the **Inspector**.

Adding a custom Reflection map to a scene

Whereas Unity **Legacy Shaders** use individual **Reflection Cubemaps** per material, the new **Standard Shader** gets its reflection from the scene's **Reflection Source**, as configured in the **Scene** section of the **Lighting** window. The level of reflectiveness for each material is now given by its **Metallic** value or **Specular** value (for materials using Specular setup). This new method can be a real time saver, allowing you to quickly assign the same reflection map to every object in the scene. Also, as you can imagine, it helps keeping the overall look of the scene coherent and cohesive. In this recipe, we will learn how to take advantage of **the Reflection Source** feature.

Getting ready

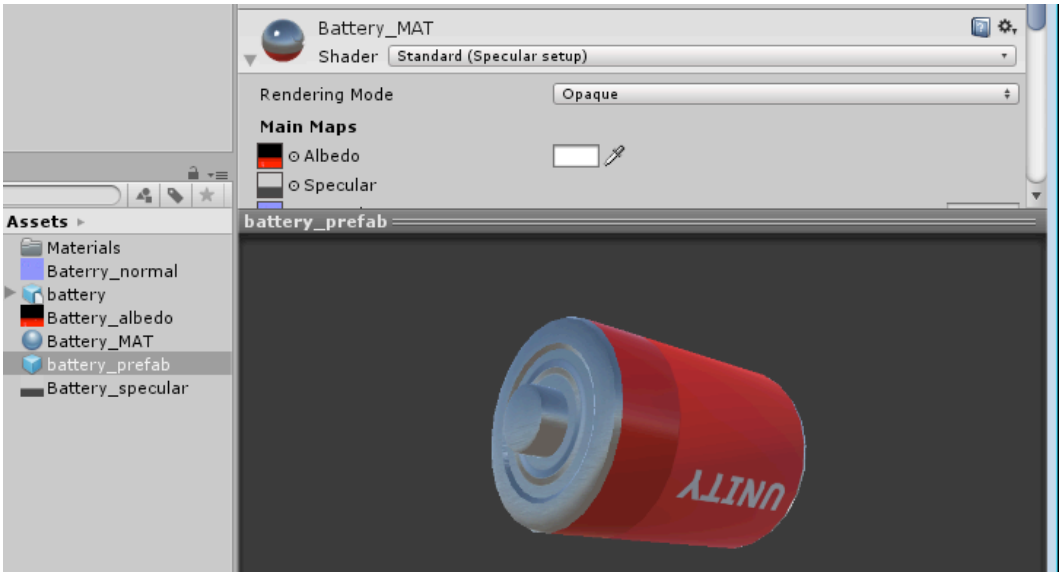
For this recipe, we will prepare a **Reflection Cubemap**, which is basically the environment to be projected as a reflection onto the material, and which can be made from either six or, as shown in this recipe, a single image file.

To help us with this recipe, it's been provided a Unity package containing a prefab made of a 3D object and a basic Material (using a PNG as Diffuse Map), and also a JPG file to be used as reflection map. All files are inside the `1362_06_02` folder.

How to do it...

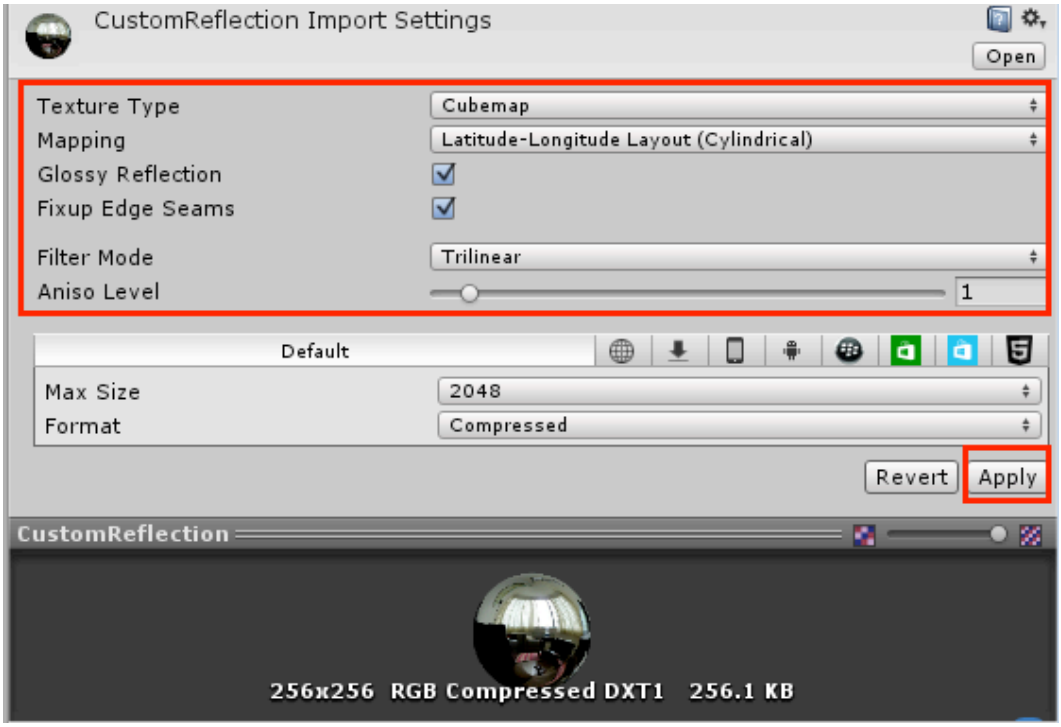
To add Reflectiveness and Specularity to a material, follow these steps:

1. Import `batteryPrefab.unitypackage` into a new project. Then, select `battery_prefab` from the **Assets** folder, in the **Project** view.
2. From the **Inspector** view, expand the **Material** component and observe the asset preview window. Thanks to the **Specular** map, the material already features a reflective look. However, it looks as if reflecting the scene's default Skybox.



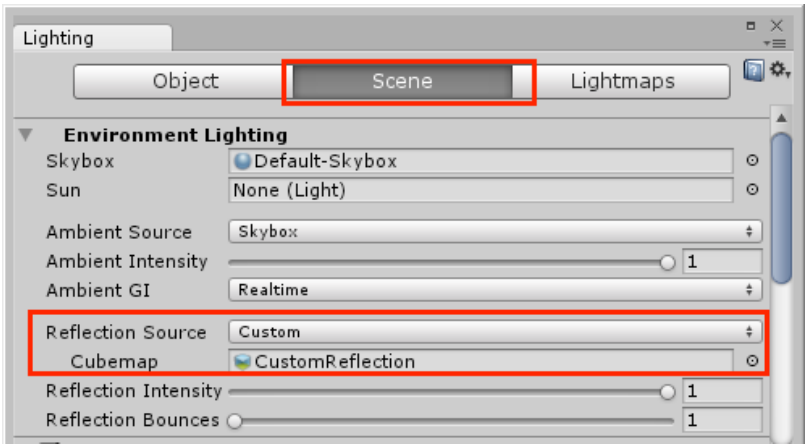
Insert image 1362OT_06_06.png

3. Import the `CustomReflection.jpg` image file. From the **Inspector** view, change its **Texture Type** to **Cubemap**, its **Mapping** to **Latitude + Longitude Layout (Cylindrical)**, and tick the boxes for **Glossy Reflection** and **Fixup Edge Seams**. Finally, change its **Filter Mode** to **Trilinear** and click the **Apply** button.



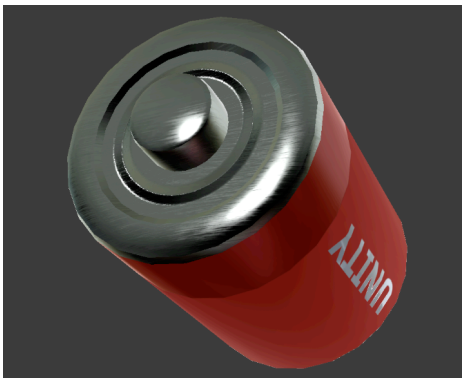
Insert image 1362OT_06_07.png

4. Let's replace the Scene's Skybox with our newly created **Cubemap**, as the **Reflection map** for our scene. In order to do that, open the **Lighting** window via the **Window | Lighting** menu option. Select the **Scene** section and use the dropdown menu to change the **Reflection Source** to **Custom.** Finally, assign the newly created `CustomReflection` texture as the **Cubemap**.



Insert image 1362OT_06_08.png

5. Check out for the new reflections on the `battery_prefab`.



Insert image 1362OT_06_09.png

How it works...

While it is the material's specular map that allows for a reflective look, including the intensity and smoothness of the reflection, the refection itself (that is: the image you see on the reflection) is given by the **Cubemap** we have created from the image file.

There's more...

Reflection Cubemaps can be done in many ways and have different mapping properties.

Mapping coordinates

The **Cylindrical** mapping we applied was well suited for the photograph we used. However, depending on how the reflection image is generated, a **Cubic** or **Spheremap**-based mapping could be more appropriate. Also, note that the option **Fixup Edge Seams** will try to make the image seamless.

Sharp reflections

You might have noticed the reflection is somewhat blurry compared to the original image. The reason is because we have ticked the **Glossy Reflections** box. To get a sharper-looking reflection, deselect that option, in which case you could also leave the **Filter Mode** option as default (Bilinear).

Maximum Size

At 512 x 512 pixels, our reflection map would probably run fine on lower end machines. However, if the quality of the reflection map is not so important in your game's context, and the original image dimensions are big (say, 4096 x 4096), you might want to change the texture's **Max Size**, at the **Import Settings**, to a lower number.

Creating a laser aim with Projector and Line Renderer

Although using GUI elements, such as a cross-hair, is a valid way to allow players to aim, replacing (or combining) it with a projected laser dot might be a more interesting approach. In this recipe, we will use the **Projector** and **Line** components to implement this concept.

Getting ready

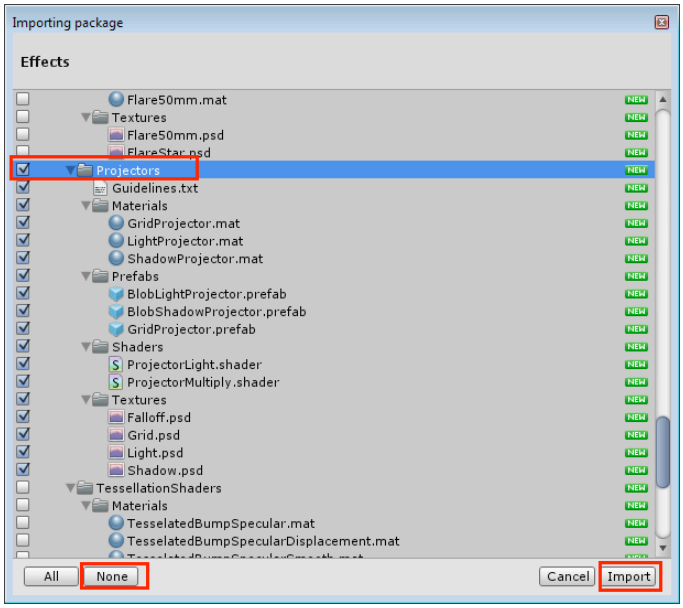
To help us with this recipe, it's been provided a Unity package containing a sample scene featuring a character holding a laser pointer, and also a texture map named `LineTexture`. All files are inside the `1362_06_03` folder. Also, we'll make use of the **Effects** assets package provided by Unity (which you should have installed when installing Unity).

How to do it...

To create a laser dot aim with a Projector, follow these steps:

1. Import `BasicScene.unitypackage` into a new project. Then, open the scene named **mecanim**. This is a basic scene featuring a player character whose aim is controlled via mouse.

2. Import the **Effects** package via the **Assets | Import Package | Effects** menu. If you want to import only the necessary files within the package, deselect everything in the **Importing package** window by clicking the **None** button, and then check the **Projectors** folder only. Then, click **Import**.

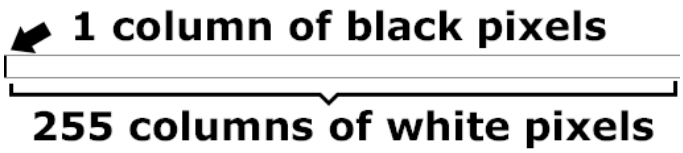


Insert image 1362OT_06_10.png

3. From the **Inspector** view, locate the **ProjectorLight** shader (inside the **Assets | Standard Assets | Effects | Projectors | Shaders** folder). Duplicate the file and name the new copy `ProjectorLaser`.
4. Open `ProjectorLaser`. From the first line of code, change `Shader "Projector/Light"` to `Shader "Projector/Laser"`. Then, locate the line of code: `Blend DstColor One` and change it to Change it to `Blend One One`. Save and close the file.

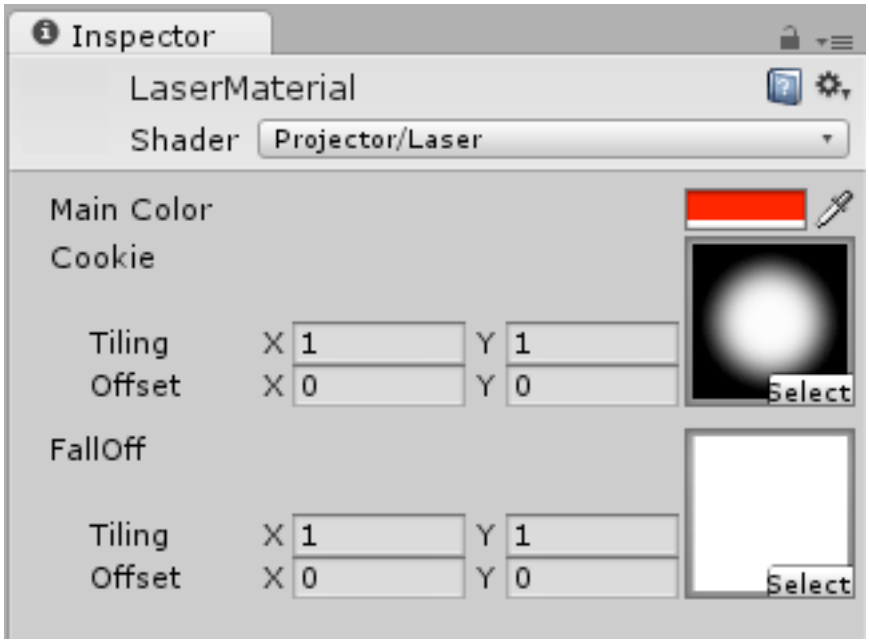
> The reason for editing the shader for the laser was making it stronger by changing its blend type to **Additive**. Shader programming is a complex subject, beyond the scope of this book. However, if you want to learn more about it, check out Unity's documentation on the subject, available at `docs.unity3d.com/Manual/SL-Reference.html`, and also the book *Unity Shaders and Effects Cookbook*, published by Packt.

5. Now that we have fixed the shader, we need a material. From the **Project** view, use the **Create** dropdown menu to create a new **Material**. Name it `LaserMaterial`. Then, select it from the **Project** view and, from the **Inspector** view, change its shader to **Projector/Laser**.
6. From the **Project** view, locate the **Falloff** texture. Open it in your image editor and, except for the first column of pixels, which should be black, paint everything white. Save the file and go back to Unity.



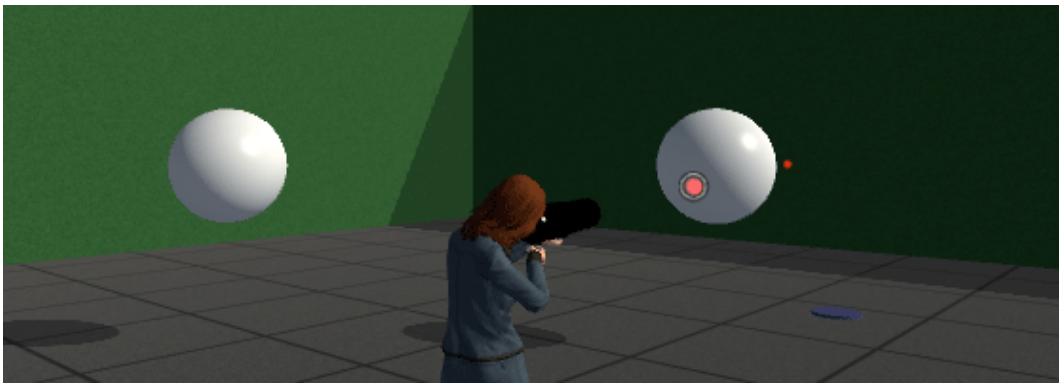
Insert image 1362OT_06_55.png

7. Change the LaserMaterial's **Main Color** to red (RGB: `255`,`0`,`0`). Then, from the texture slots, select the **Light** texture as **Cookie** and the **Falloff** texture as **Falloff**.



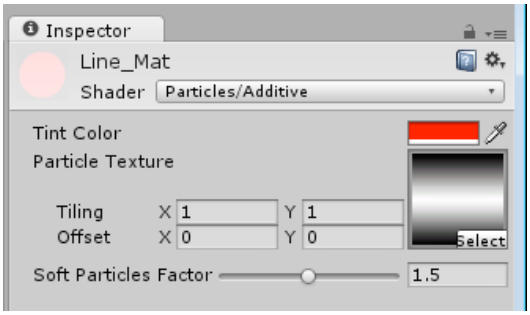
Insert image 1362OT_06_11.png

8. From the **Hierarchy** view, find and select the **pointerPrefab** object (**book | mixamorig:Hips | mixamorig:Spine | mixamorig:Spine1 | mixamorig:Spine2 | mixamorig:RightShoulder | mixamorig:RightArm | mixamorig:RightForeArm | mixamorig:RightHand | pointerPrefab**). Then, from the **Create** dropdown menu, select **Create Empty Child**. Rename the new child of **pointerPrefab** as **LaserProjector**.
9. Select the **LaserProjector** object. Then, from the **Inspector** view, click the **Add Component** button and navigate to **Effects | Projector**. Then, from the **Projector** component, set the option **Orthographic** as true and set **Orthographic Size** as `0.1`. Finally, select **LaserMaterial** from the **Material** slot.
10. Test the scene. You should be able to see the laser aim dot.



Insert image 1362OT_06_12.png

11. Now, let's create a material for the **Line Renderer** component we are about to add. From the **Project** view, use the **Create** dropdown menu to add a new **Material**. Name it **Line_Mat**.
12. From the **Inspector** view, change the shader of the **Line_Mat** to **Particles/Additive**. Then, set its **Tint Color** to red (RGB: `255`;`0`;`0`).
13. Import the `LineTexture` image file. Then, set it as the **Particle Texture** for the **Line_Mat**.



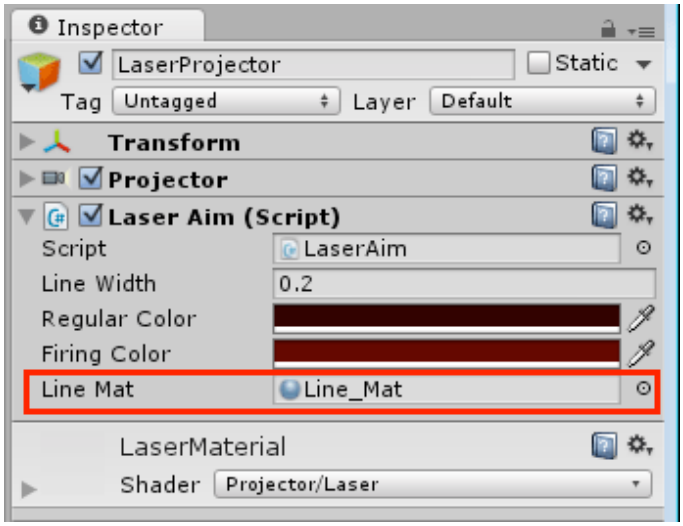
Insert image 1362OT_06_13.png

14. Use the **Create** dropdown menu from **Project** view to add a C# script named `LaserAim`. Then, open it in your editor.
15. Replace everything with the following code:

```
using UnityEngine;
using System.Collections;
public class LaserAim : MonoBehaviour {

    public float lineWidth = 0.2f;
    public Color regularColor = new Color (0.15f, 0, 0, 1);
    public Color firingColor = new Color (0.31f, 0, 0, 1);
    public Material lineMat;
    private Vector3 lineEnd;
    private Projector proj;
    private LineRenderer line;

    void Start () {
        line = gameObject.AddComponent<LineRenderer>();
        line.material = lineMat;
        line.material.SetColor("_TintColor", regularColor);
        line.SetVertexCount(2);
        line.SetWidth(lineWidth, lineWidth);
        proj = GetComponent<Projector> ();
    }

    void Update () {
        RaycastHit hit;
        Vector3 fwd =
transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast (transform.position, fwd, out hit)) {
            lineEnd =  hit.point;
            float margin = 0.5f;
```

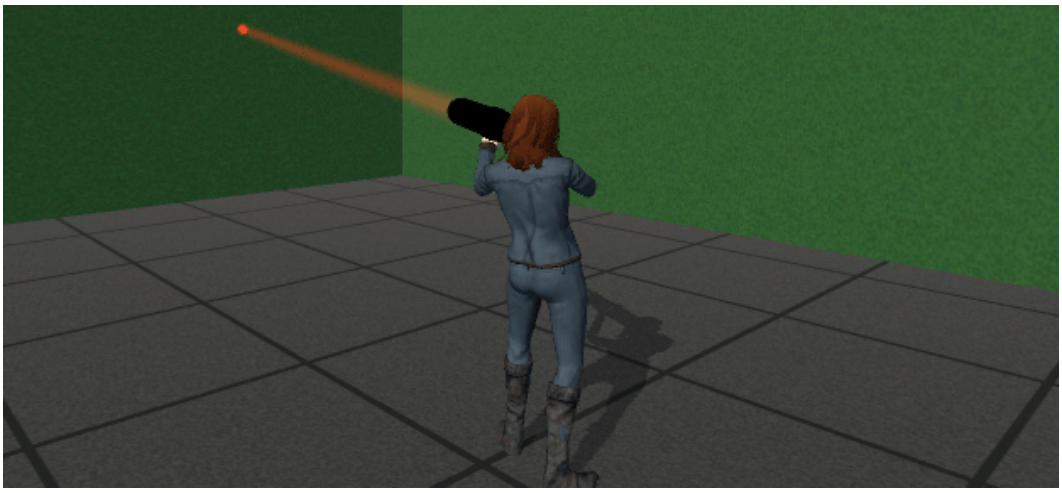
```
            proj.farClipPlane = hit.distance + margin;

        } else {
            lineEnd = transform.position + fwd * 10f;
        }
        line.SetPosition(0, transform.position);
        line.SetPosition(1, lineEnd);

        if(Input.GetButton("Fire1")){
            float lerpSpeed = Mathf.Sin (Time.time * 10f);
            lerpSpeed = Mathf.Abs(lerpSpeed);
            Color lerpColor = Color.Lerp(regularColor, firingColor,
lerpSpeed);
            line.material.SetColor("_TintColor", lerpColor);

        }
        if(Input.GetButtonUp("Fire1")){
            line.material.SetColor("_TintColor", regularColor);
        }
    }
}
```

16. Save your script and attach it to the **LaserProjector** game object.
17. Select the **LaserProjector** game object. From the **Inspector** view, find the **Laser Aim** component and fill the **Line Material** slot with the `Line_Mat` material.



Insert image 1362OT_06_14.png

18. Play the scene. The laser aim is ready.



Insert image 1362OT_06_15.png

> In this recipe, the width of the laser beam and its aim dot have been exaggerated. Should you need a more realistic thickness for your beam, change the **Line Width** field of the **Laser Aim** component to `0.05`, and the **Orthographic Size** of the **Projector** component to `0.025`. Also, remember to make the beam more opaque by setting the **Regular Color** of the **Laser Aim** component brighter.

How it works...

The laser aim effect was achieved by combining two different effects: a **Projector** and a **Line Renderer**.

The **Projector**, which can be used to simulate light, shadows and more, is a component that projects a material (and its texture) onto other game objects. By attaching a Projector to the **Laser Pointer** object, we have ensured it would face the right direction at all times. To get the right, vibrant look, we have edited the projector material's shader, making it brighter. Also, we have scripted a way to prevent projections from going through objects, by setting its Far Clip Plane on approximately the same level of the first object receiving the projection. The line of code responsible for that action is: `proj.farClipPlane = hit.distance + margin;`

Regarding the **Line Renderer**, we have opted to create it dynamically, via code, instead of manually adding the component to the game object. The code is also responsible for

setting up its appearance, updating the line vertices position, and changing its color whenever the fire button is pressed, giving it a glowing / pulsing look.

For more details on how the script works, don't forget to check out the commented code, available within the `1362_06_03 | End` folder.

Reflecting surrounding objects with Reflection Probes

If you want your scene's environment to be reflected by game objects featuring reflective materials (such as the ones with high Metallic or Specular levels), you can achieve such effect using **Reflection Probes**. They allow for real-time, baked or even custom reflections through the use of Cubemaps.

Real-time reflections can be expensive in terms of processing, in which case you should favor Baked reflections unless it's really necessary to display dynamic objects being reflected (mirror-like objects, for instance). Still, there are some ways real-time reflections can be optimized. In this recipe, we will test three different configurations for reflection probes:

- Real-time reflections (constantly updated)
- Real-time reflections (updated on-demand) via script
- Baked reflections (from the Editor)

Getting ready

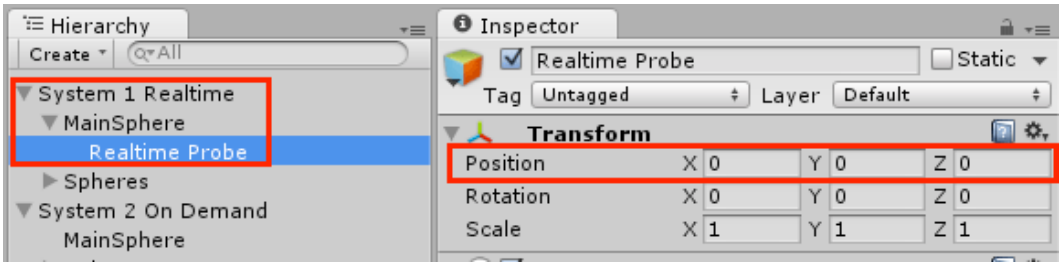
For this recipe, we have prepared a basic scene featuring three sets of reflective objects: one is constantly moving, one is static, and one moves whenever it is interacted with. The `Probes.unitypackage` package containing the scene can be found inside the `1362_06_04` folder.

How to do it...

To reflect surrounding objects using Reflection probes, follow these steps:

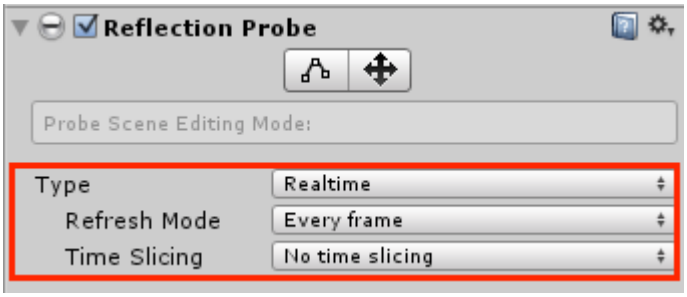
1. Import `Probes.unitypackage` into a new project. Then, open the scene named **Probes**. This is a basic scene featuring three sets of reflective objects.
2. Play the scene. Observe that one of the systems is dynamic, one is static and one rotates randomly whenever a key is pressed.
3. Stop the scene.
4. First, let's create a constantly updated real-time reflection probe. From the **Create** dropdown button of the **Hierarchy** view, add a **Reflection Probe** to the

Scene (**Create | Light | Reflection Probe**). Name it `RealtimeProbe` and make it a child of the **System 1 Realtime | MainSphere** game object. Then, from the **Inspector** view, **Transform** component, change its **Position** to **X:**0;**Y:**0;**Z:**0.



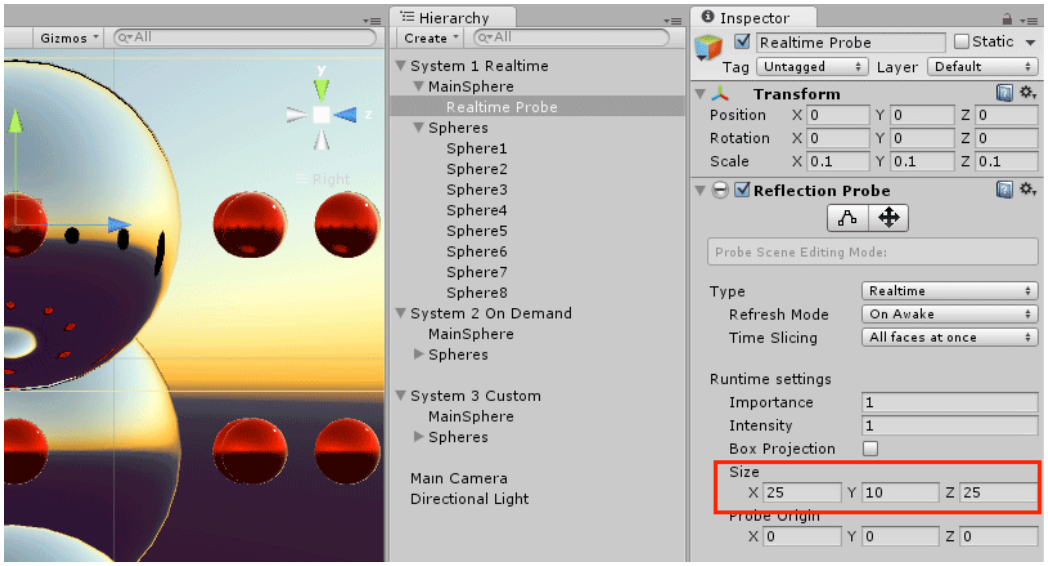
Insert image 1362OT_06_16.png

5. Now go to the **Reflection Probe** component. Set **Type** as **Realtime**; **Refresh Mode** as **Every Frame** and **Time Slicing** as **No time slicing**.



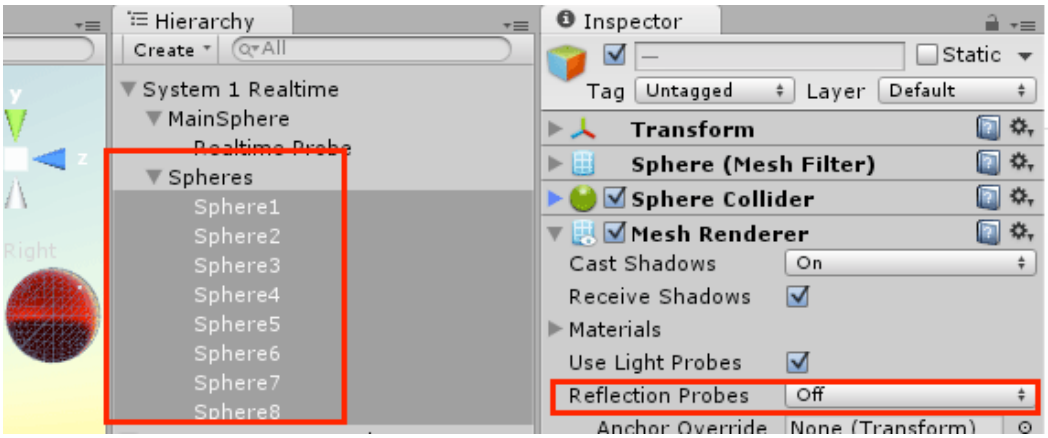
Insert image 1362OT_06_17.png

6. Play the scene. Reflections should be updated in real time. Stop the scene.
7. Observe that the only object displaying real-time reflections is the **System 1 Realtime | MainSphere**. The reason for that is the **Size** of the Reflection Probe. From the **Reflection Probe** component, change its **Size** to **X**:25; **Y**:10; **Z**:25. Note that the small red spheres are now affected as well. However, it is important to notice that all objects display the same reflection. Since our reflection probe's origin is placed at the same location as the **MainSphere**, all reflective objects will display reflections from that point of view.



Insert image 1362OT_06_18.png

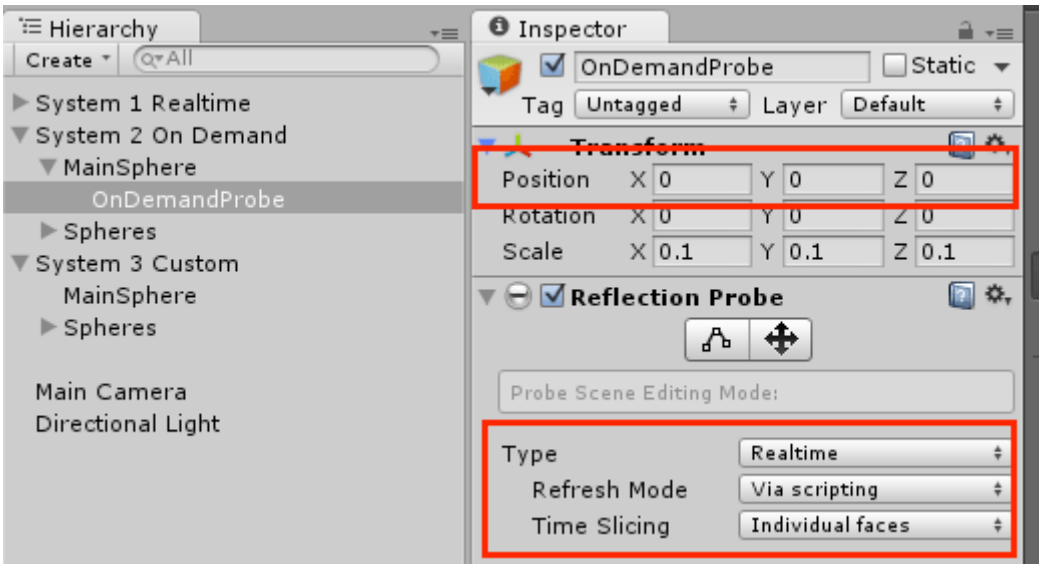
8. If you want to eliminate the reflection from reflective objects within the reflection probe, such as the small red spheres, select the objects and, from the **Mesh Renderer** component, set **Reflection Probes** as **Off**.



Insert image 1362OT_06_19.png

9. Add a new Reflection Probe to the scene. This time, name it `OnDemandProbe` and make it a child of the **System 2 On Demand | MainSphere** game object. Then, from the **Inspector** view, **Transform** component, change its **Position** to **X:**0;**Y:**0;**Z:**0.

10. Now go to the **Reflection Probe** component. Set **Type** as **Realtime**; **Refresh Mode** as **Via scripting** and **Time Slicing** as **Individual faces**.



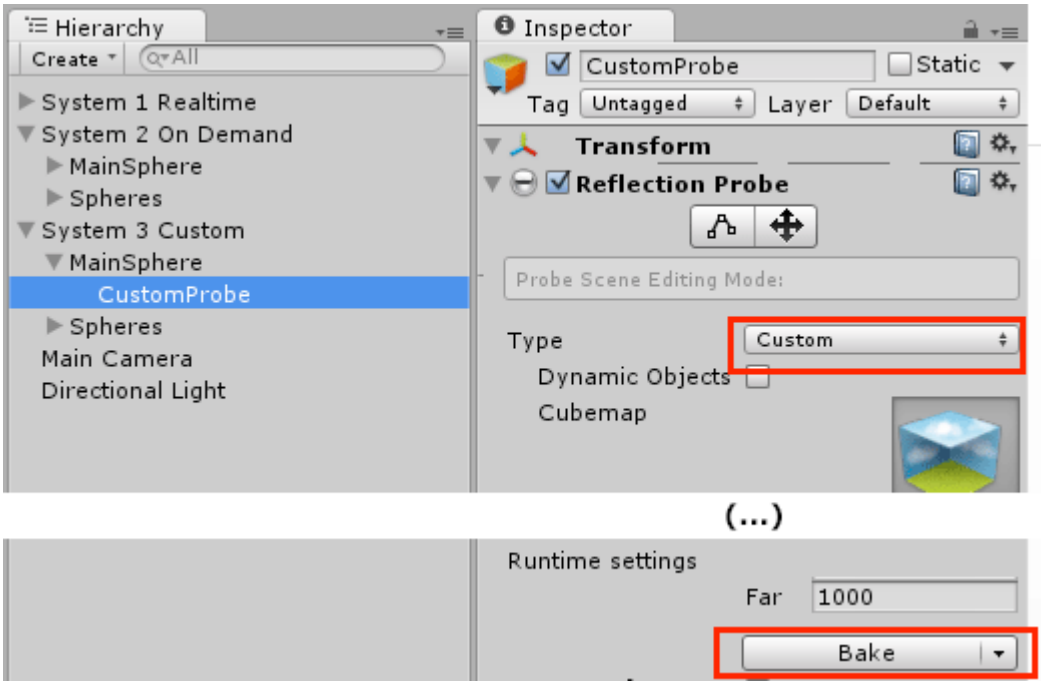
Insert image 1362OT_06_20.png

11. Using the **Create** dropdown menu in the **Project** view, create a new C# Script named `UpdateProbe.cs`.
12. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class UpdateProbe : MonoBehaviour {
    private ReflectionProbe probe;

    void Awake () {
        probe = GetComponent<ReflectionProbe> ();
        probe.RenderProbe();
    }

    public void RefreshProbe(){
        probe.RenderProbe();
    }
}
```

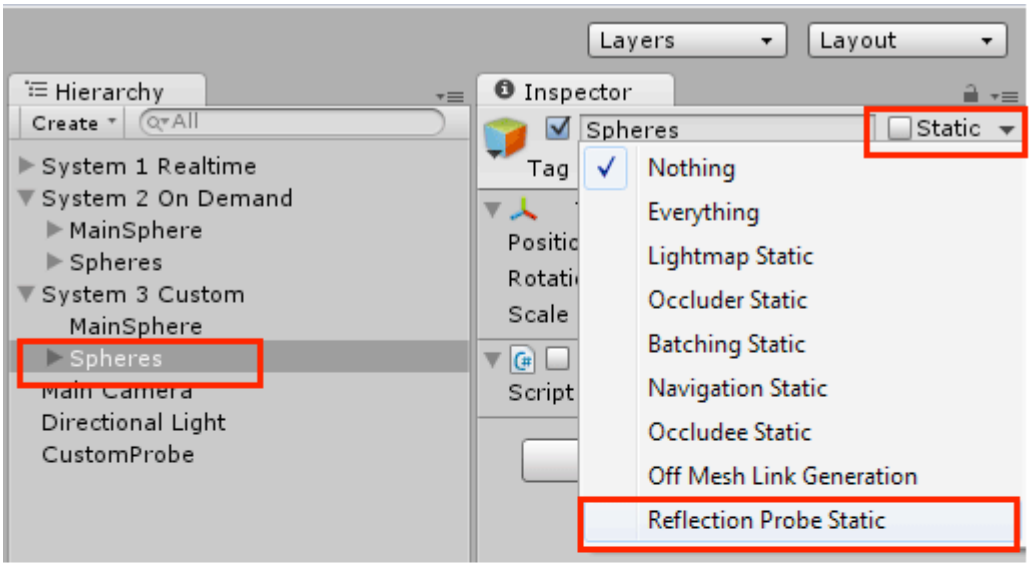
13. Save your script and attach it to the **OnDemandProbe**.

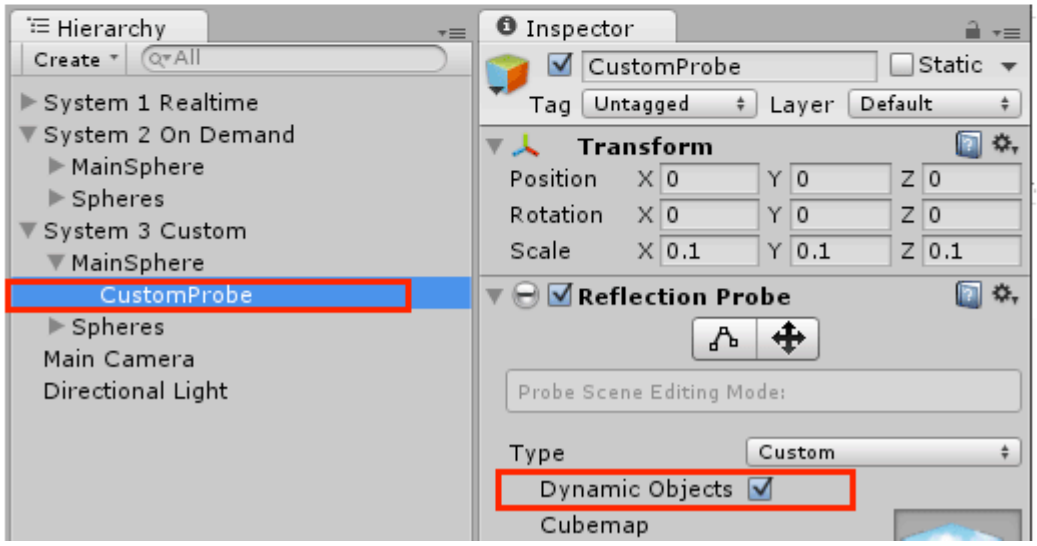
14. Now find the script named `RandomRotation`, which is attached to the **System 2 On Demand | Spheres** object, and open it in the code editor.
15. Right before the `Update()` function, add the following lines:
```
private GameObject probe;
private UpdateProbe up;
void Awake(){
    probe = GameObject.Find("OnDemandProbe");
    up = probe.GetComponent<UpdateProbe>();
}
```
16. Now locate the line of code `transform.eulerAngles = newRotation;` and, immediately after it, add the following line:
```
up.RefreshProbe();
```
17. Save the script and test your scene. Observe how the reflection probe is updated whenever a key is pressed.
18. Stop the scene. Add a third Reflection Probe to the scene. Name it `CustomProbe` and make it a child of the **System 3 On Custom | MainSphere** game object. Then, from the **Inspector** view, **Transform** component, change its **Position** to **X:**0;**Y:**0;**Z:**0.
19. Go to the **Reflection Probe** component. Set **Type** as **Custom** and click the **Bake** button.



Insert image 1362OT_06_21.png

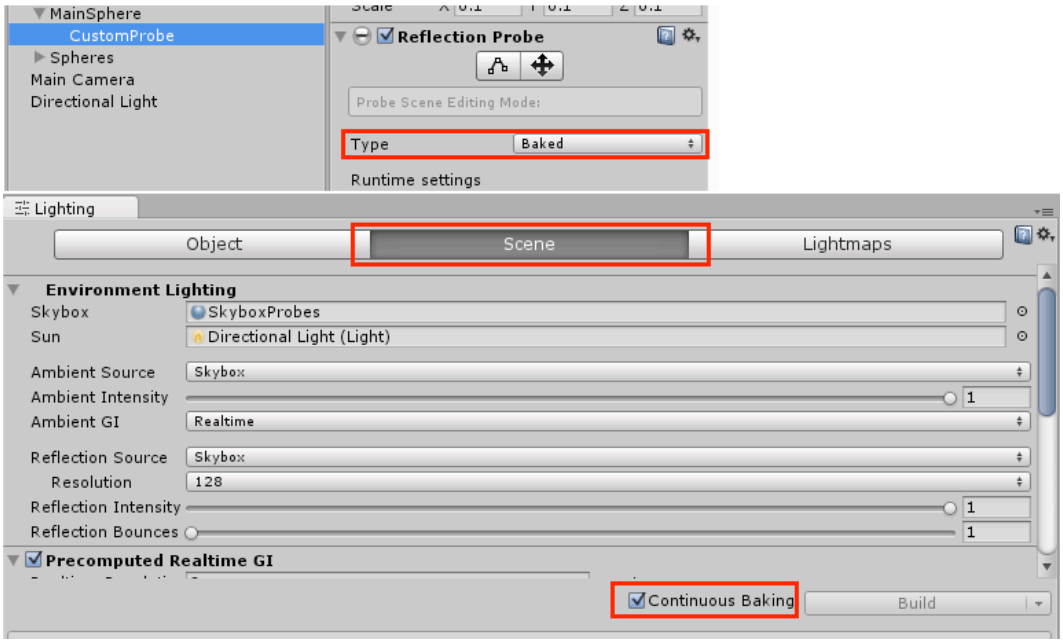
20. A Save File dialog window will show up. Save the file as `CustomProbe-reflectionHDR.exr`.
21. Observe the reflection map does not include the reflection of red spheres on it. To change that, you have two options: set the **System 3 On Custom | Spheres** game object (and all its children) as **Reflection Probe Static** or, from the **Reflection Probe** component of the **CustomProbe** game object, check the **Dynamic Objects** option and bake the map again (by clicking the **Bake** button).





Insert image 1362OT_06_22.png

22. If you want your reflection Cubemap to be dynamically baked while you edit your scene, you can set the Reflection Probe **Type** to **Baked**, open the **Lighting** window (menu **Assets | Lighting**), access the **Scene** section and check the **Continuous Baking** option. Please note this mode won't include dynamic objects in the reflection, so be sure to set **System 3 On Custom | Spheres** and **System 3 On Custom | MainSphere** as **Reflection Probe Static**.



Insert image 1362OT_06_23.png

How it works...

Reflection Probes act like omnidirectional cameras that render **Cubemaps** and apply them onto objects within their constraints. When creating Reflection Probes, it's important to be aware how different types work.

Realtime: Cubemaps are updated at runtime.

Realtime Reflection Probes have three different **Refresh Modes**: **On Awake** (Cubemap is baked once right before the scene starts); **Every frame** (Cubemap is constantly updated); **Via scripting** (Cubemap is updated whenever the RenderProbe function is used).

Since Cubemaps feature six sides, Reflection Probes features **Time Slicing**, so each side can be updated independently. There are three different types of Time Slicing: **All Faces at Once** (renders all faces at once, calculates mipmaps over 6 frames. Updates the probe in 9 frames); **Individual Faces** (each face is rendered over a number of frames. Updates the probe in 14 frames. Results can be a bit inaccurate, but is the least expensive solution in terms of framerate impact); **No Time Slicing** (Probe is rendered and mipmaps are calculated in one frame. High accuracy, but also the most expensive in terms of framerate).

Baked: Cubemaps are baked during scene editing. Cubemaps can be either manually or automatically updated, depending whether the **Continuous Baking** option is checked (it can be found at the **Scene** section of the **Lighting** window).

Custom: Custom Reflection Probes can be either manually baked from the scene (and even include Dynamic Objects), or created from a pre-made Cubemap.

There's more...

There are a number of additional settings that can be tweaked, such as **Importance**, **Intensity**, **Box Projection**, **Resolution**, **HDR**, and so on. For a complete view on each of these settings, we strongly recommend you to read Unity's documentation on the subject, which is available at `docs.unity3d.com/Manual/class-ReflectionProbe.html`.

Setting up an environment with Procedural Skybox and Directional Light

Besides the traditional 6 Sided and Cubemap, Unity features now a third type of skybox: the **Procedural Skybox**. Easy to create and setup, the Procedural Skybox can be used in conjunction with a Directional Light to provide an Environment Lighting to your scene. In this recipe, we will learn about different parameters of the Procedural Skybox.

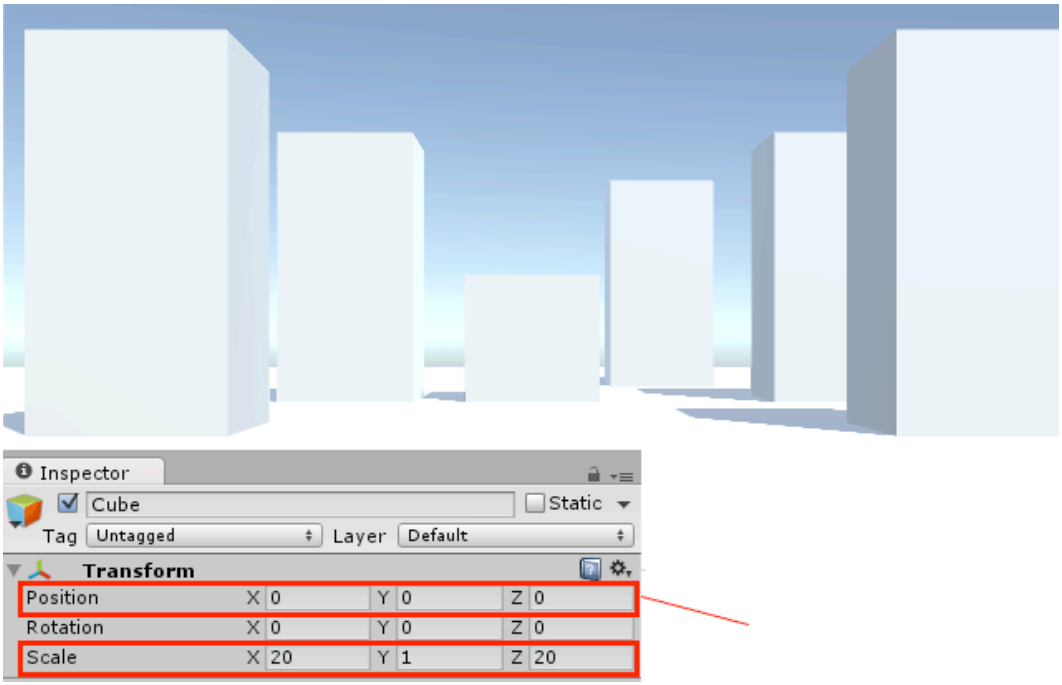
Getting ready

For this recipe, you will need to import Unity's Standard Assets Effects package, which you should have installed when installing Unity.

How to do it...

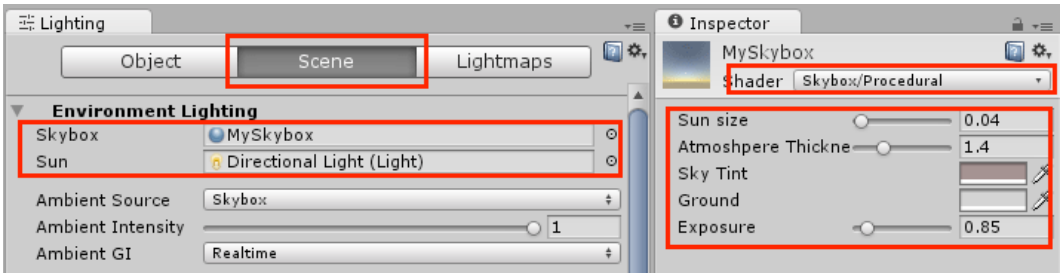
To set up an Environment Lighting using Procedural Skybox and Directional Light, follow these steps:

1. Create a new scene inside a Unity project. Observe that a new scene already includes two objects: the **Main Camera** and a **Directional Light**.
2. Add some cubes to your scene, including one at **Position X:0;Y:0;Z:0** scaled to **X:20; Y:1;Z:20**, to be used as the ground.



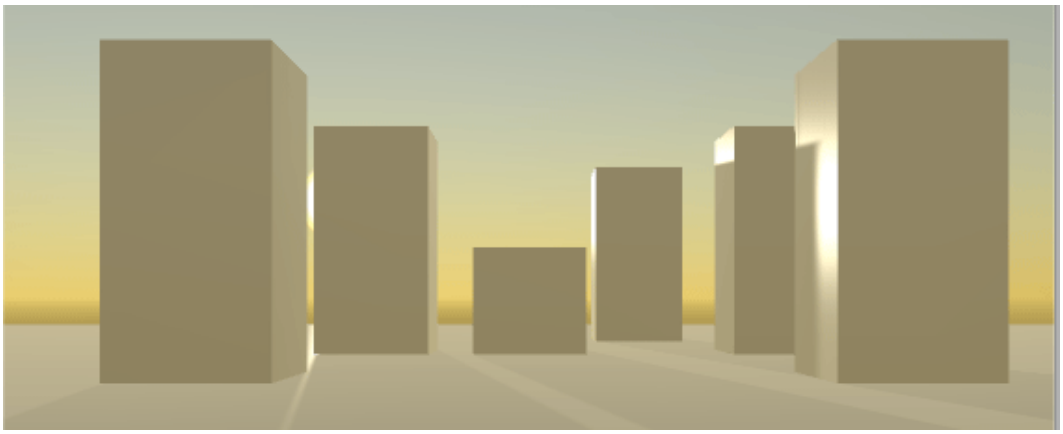
Insert image 1362OT_06_24.png

3. Using the **Create** dropdown menu from the **Project** view, create a new Material and name it `MySkybox`. From the **Inspector** view, use the appropriate dropdown menu to change the **Shader** of **MySkybox** from **Standard** to **Skybox/Procedural**.
4. Open the **Lighting** window (menu **Window | Lighting**), access the **Scene** section and, at the **Environment Lighting** subsection, populate the **Skybox** slot with the **MySkybox** material and the **Sun** slot with the **Directional Light** from the Scene.
5. From the **Project** view, select **MySkybox**. Then, from the **Inspector** view, set **Sun size** as `0.05` and **Atmosphere Thickness** as `1.4`. Experiment by changing the **Sky Tint** color to RGB: `148`;`128`;`128` and the **Ground** color to a value that resembles the scene cube floor's color (such as RGB: `202`;`202`;`202`). If you feel the scene is too bright, try bringing the **Exposure** level down to `0.85`.



Insert image 1362OT_06_25.png

6. Select the Directional Light and change its Rotation to **X:**5;**Y:**170;**Z:**0. Note that the scene should resemble a dawning environment.



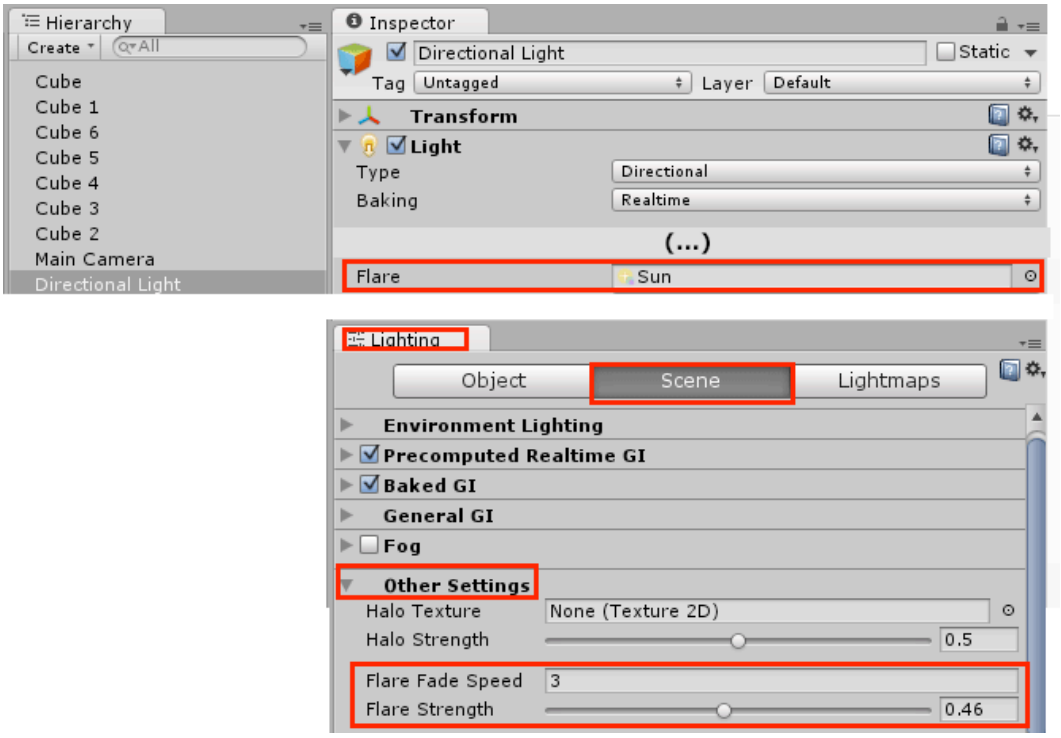
Insert image 1362OT_06_26.png

7. Let's make things even more interesting: using the **Create** dropdown menu in the **Project** view, create a new C# Script named `RotateLight`. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;
public class RotateLight : MonoBehaviour {
   public float speed = -1.0f;
   void Update () {
      transform.Rotate(Vector3.right * speed * Time.deltaTime);
   }
}
```

8. Save it and add it as a component to the **Directional Light**.

9. Import the **Effects** Assets package into your project (via the **Assets | Import Package | Effects** menu).
10. Select the **Directional Light**. Then, from **Inspector** view, **Light** component, populate the **Flare** slot with the `Sun` flare.
11. From the **Scene** section of the **Lighting** window, find the **Other Settings** subsection. Then, set **Flare Fade Speed** as `3` and **Flare Strength** as `0.5`.



Insert image 1362OT_06_27.png

12. Play the scene. You should see the sun rising, and the Skybox colors changing accordingly.

How it works...

Ultimately, the appearance of Unity's native Procedural Skyboxes lies on the five parameters that make them up:

- **Sun size:** The size of the bright yellow sun drawn onto the skybox, located according to the Directional Light's Rotation on X and Y axes.

- **Atmosphere Thickness:** Simulates how dense is the atmosphere for that skybox. Lower values (less than 1.0) are good for simulating outer space settings. Moderate values (around 1.0) are suitable for earth-based environments. Values a bit above 1.0 can be useful when simulating air pollution and other dramatic settings. Exaggerated values (like more than 2.0) can help illustrating extreme conditions or even alien settings.
- **Sky Tint:** Color to tint the skybox. Useful for fine-tuning or creating stylized environments.
- **Ground:** The color of the ground. It can really affect the Global Illumination of the scene, so choose a value that is close to the level's terrain and/or geometry (or a neutral one).
- **Exposure:** How much light gets in the skybox. Higher levels simulate overexposure, while lower values simulate underexposure.

It is important to notice that the Skybox appearance will respond to the scene's Directional Light playing the role of the Sun. In this case, rotating the light around its X axis can create dawn and sunset scenarios, whereas rotating it around its Y axis will change the position of the sun, changing the cardinal points of the scene.

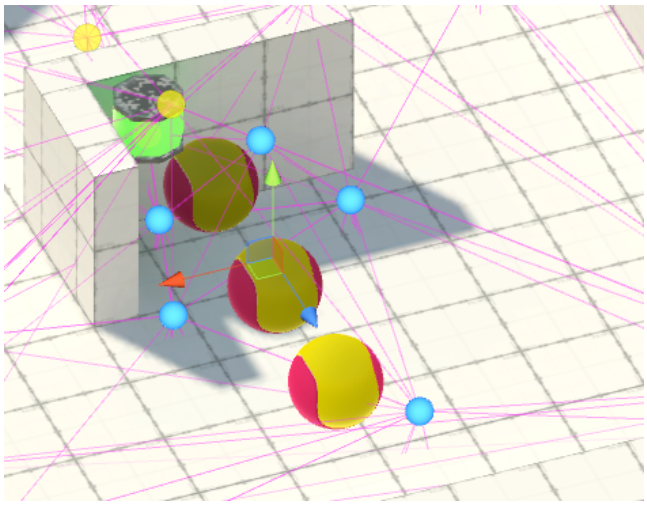
Also, regarding the **Environment Lighting**, note that although we have used the Skybox as the **Ambient Source**, we could have chosen a **Gradient** or a single **Color** instead – in which case the scene's illumination wouldn't be attached to the Skybox appearance.

Finally, also regarding the **Environment Lighting**, please note that we have set the **Ambient GI** to **Realtime**. The reason for that was allowing the realtime changes in the GI promoted by the rotating Directional Light. In case we didn't need those changes at runtime, we could have chosen the **Baked** alternative.

Lighting a simple scene with Lightmaps and Light Probes

Lightmaps are a great alternative to Realtime lighting, as they can provide the desired look to an environment without being processor-intensive. There is one downside, though: since there is no way of baking Lightmaps onto dynamic objects, the lighting of important elements of the game (such as player characters themselves) could look artificial, failing to match the intensity of the surrounding area. The solution? **Light Probes**.

Light probes work by sampling the light intensity over the location they are placed at. Dynamic objects, once Light Probe-enabled, will be lit according to the interpolation of nearest probes around them.



Insert image 1362OT_06_48.png

Getting ready

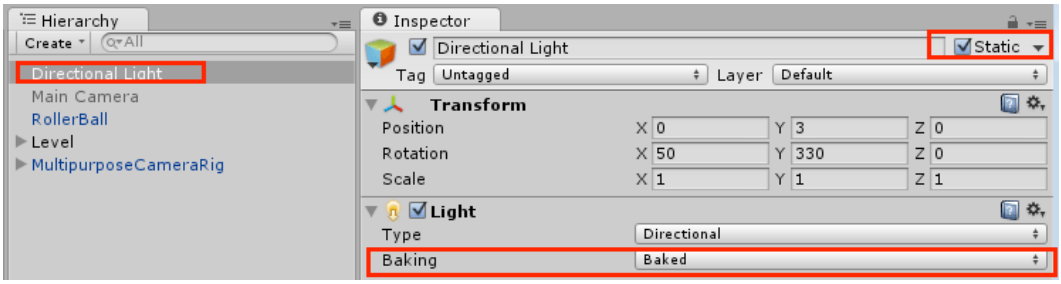
For this recipe, we have prepared a basic scene including a simple game environment and an instance of Unity's Rollerball sample asset, which will be used as the player character. The geometry for the scene was created using **ProBuilder 2.0**, an extension developed by **ProCore** and sold at Unity's **Asset Store** and at ProCore's website (`http://www.protoolsforunity3d.com`).ProBuilder is a fantastic level design tool that speeds up the design process considerably for both simple and complex level design.

The `LightProbes.unitypackage` package containing the scene and all necessary files can be found inside the `1362_06_06` folder.

How to do it...

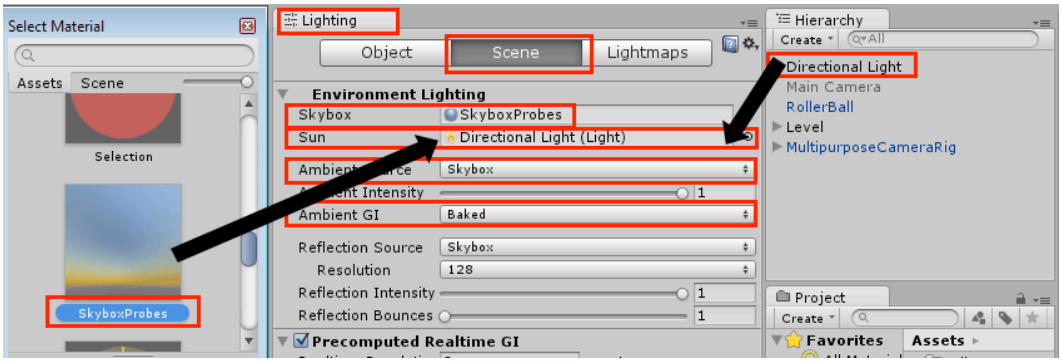
To reflect surrounding objects using Reflection probes, follow these steps:

1. Import `LightProbes.unitypackage` into a new project. Then, open the scene named **LightProbes**. The scene features a basic environment and a playable Rollerball.
2. First, let's set up the light from our scene. From the **Hierarchy** view, select the **Directional Light**. Then, from the **Inspector** view, set **Baking** as **Baked**. Also, from the top of the **Inspector**, on the right to the object's name, check the **Static** box.



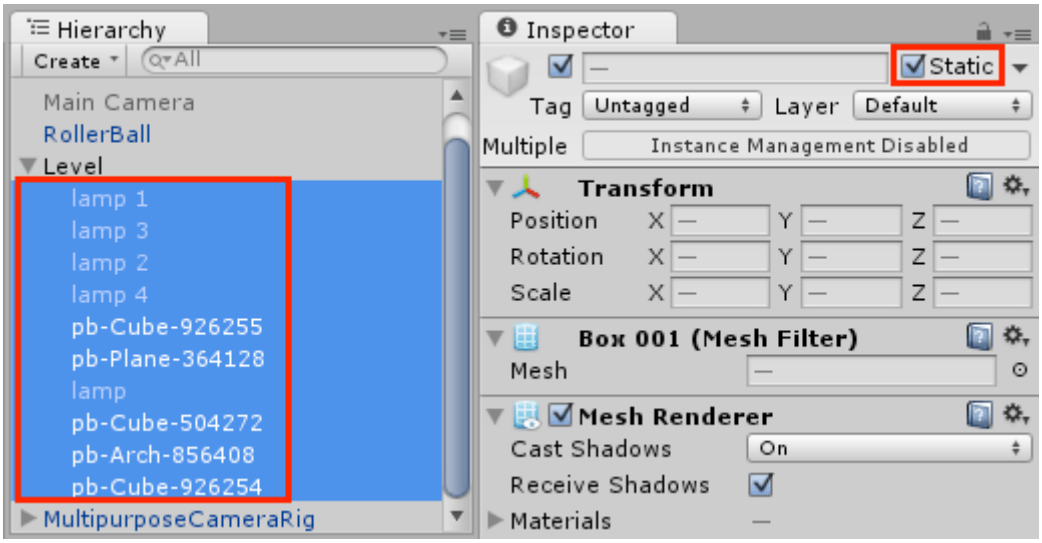
Insert image 1362OT_06_28.png

3. Now, let's set up the **Global Illumination** for the scene. Open the **Lighting** window (via the menu **Window | Lighting**) and select the **Scene** section. Then, from the **Environment Lighting** subsection, set `SkyboxProbes` (available from the **Assets**) as the **Skybox** and the scene's **Directional Light** as the **Sun**. Finally, change the **Ambient GI** option from **Realtime** to **Baked**.



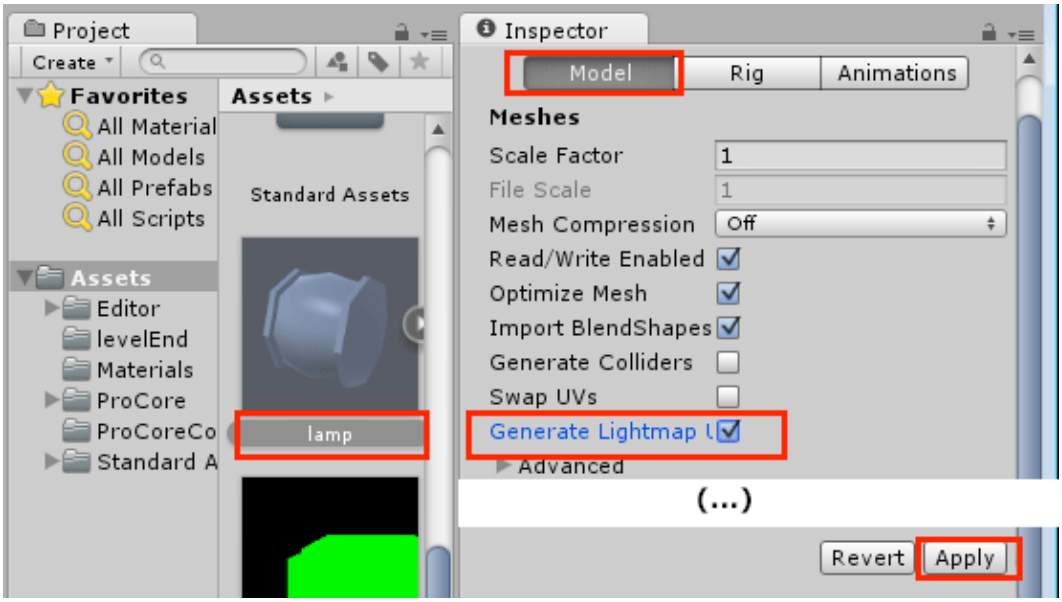
Insert image 1362OT_06_29.png

4. Lightmaps can be applied onto Static objects only. From the **Hierarchy** view, expand the **Level** game object to reveal the list of children objects. Then, select every child and set them as **Static**.



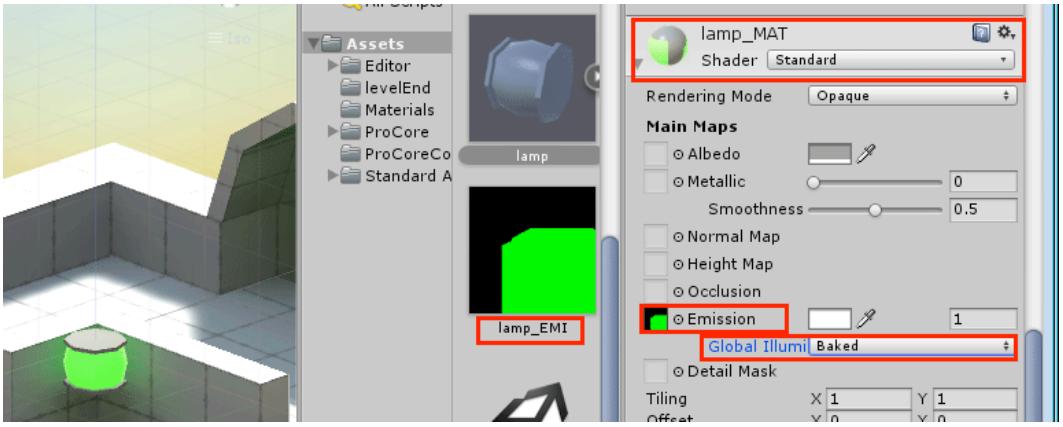
Insert image 1362OT_06_30.png

5. Imported 3D meshes must feature **Lightmap UV Coordinates**. From the **Project** view, find and select the `lamp` mesh. Then, from the **Inspector** view, within the **Model** section of the **Import Settings**, check the **Generate Lightmap UVs** option and click the **Apply** button to confirm changes.



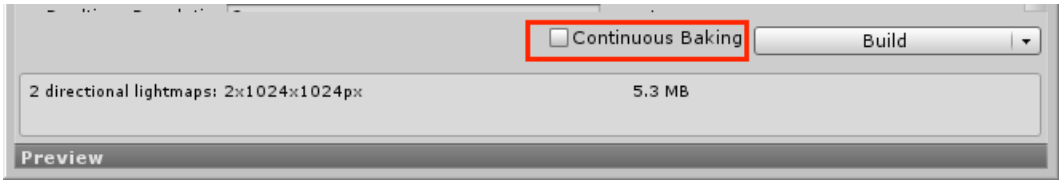
Insert image 1362OT_06_31.png

6. Scroll down the **Import Settings** view and expand the lamp's **Material** component. Then, populate the **Emission** field with the texture named `lamp_EMI`, available from the **Assets** folder. Finally, change **the Global Illumination** option to **Baked**. That will make the lamp object emit a greenlight that will be baked into the Lightmap.



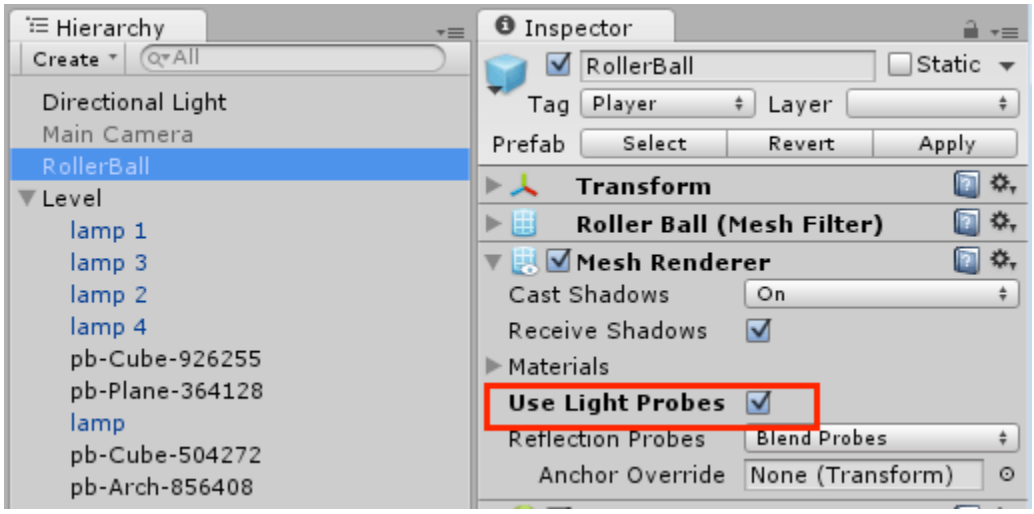
Insert image 1362OT_06_32.png

7. Open the **Lighting** window. By default, the **Continuous Baking** option should be checked. Uncheck it, so we can bake Lightmaps on demand.



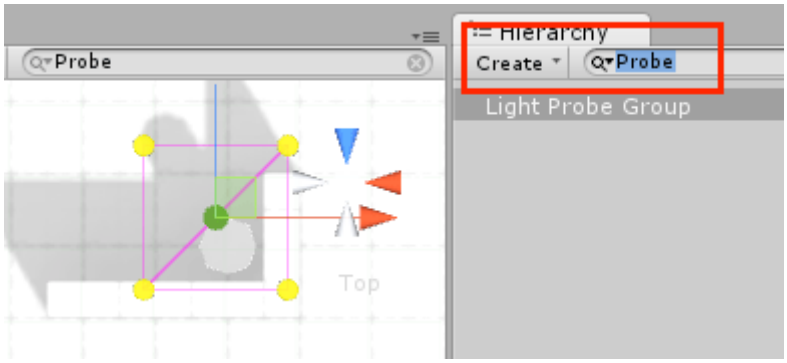
Insert image 1362OT_06_33.png

8. Click the **Build** button and wait for the Lightmaps to be generated.
9. From the **Hierarchy** view, select the **RollerBall**. Then, from the **Inspector** view, find the **Mesh Renderer** component and check the **Use Light Probes** option.



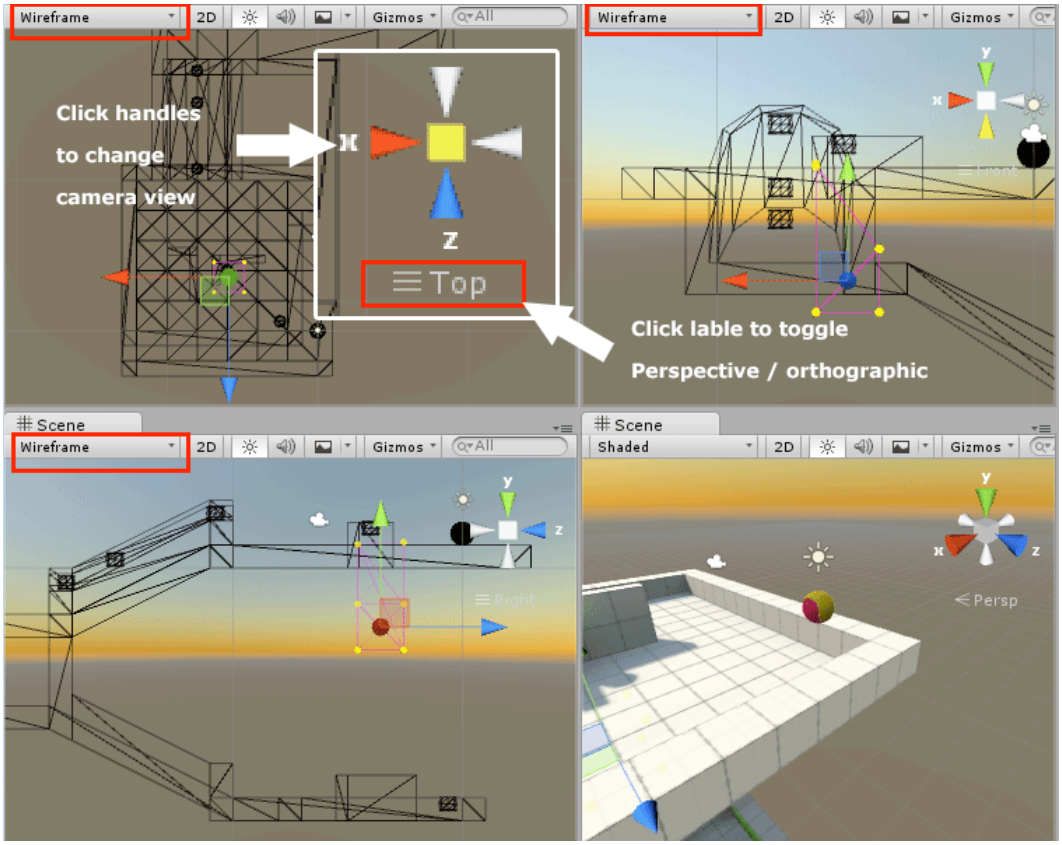
Insert image 1362OT_06_34.png

10. Now we need to create the **Light Probes** for the scene. From the **Hierarchy** view, click the **Create** dropdown menu and add a **Light Probe Group** to the scene (**Create | Light | Light Probe Group**).
11. To facilitate the manipulation of the probes, type `Probe` into the search field of the **Hierarchy** view. That should isolate the newly created Light Probe Group, making it the only editable object on the scene.



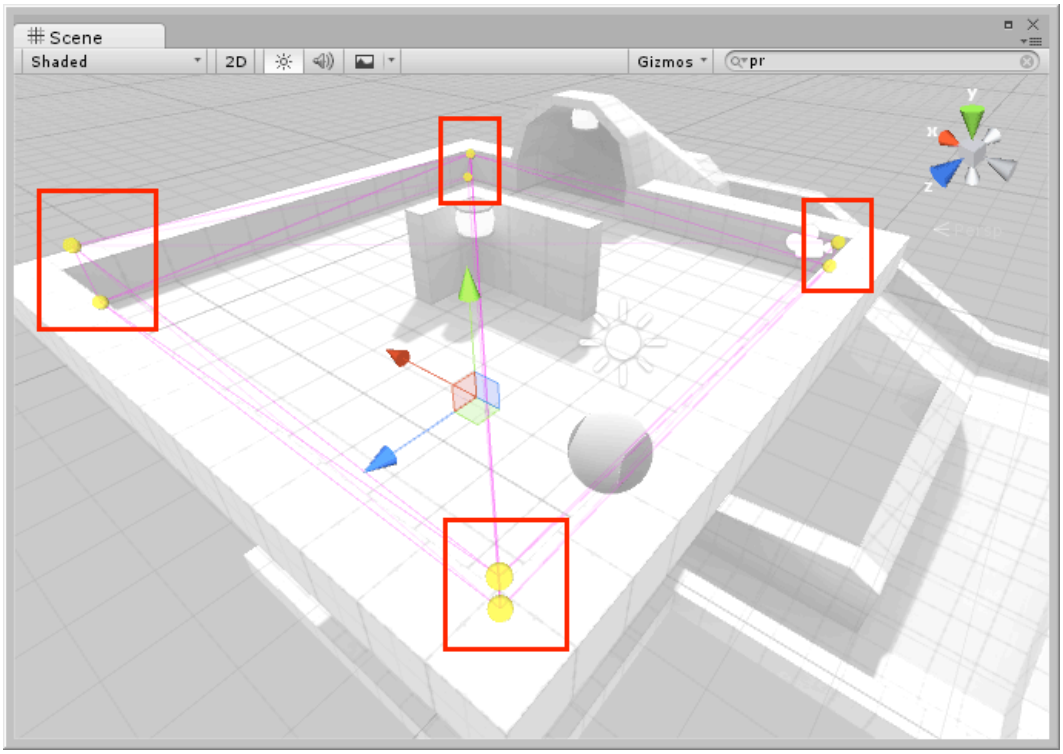
Insert image 1362OT_06_35.png

12. Change your viewport layout to **4 Split** via menu **Window | Layouts | 4 Split**. Then, set viewports as **Top**, **Front**, **Right**, and **Persp**. Optionally, change Top, Front and Right views to **Wireframe** mode. Finally, make sure they are set to orthographic view. That should make easier to position the Light Probes.



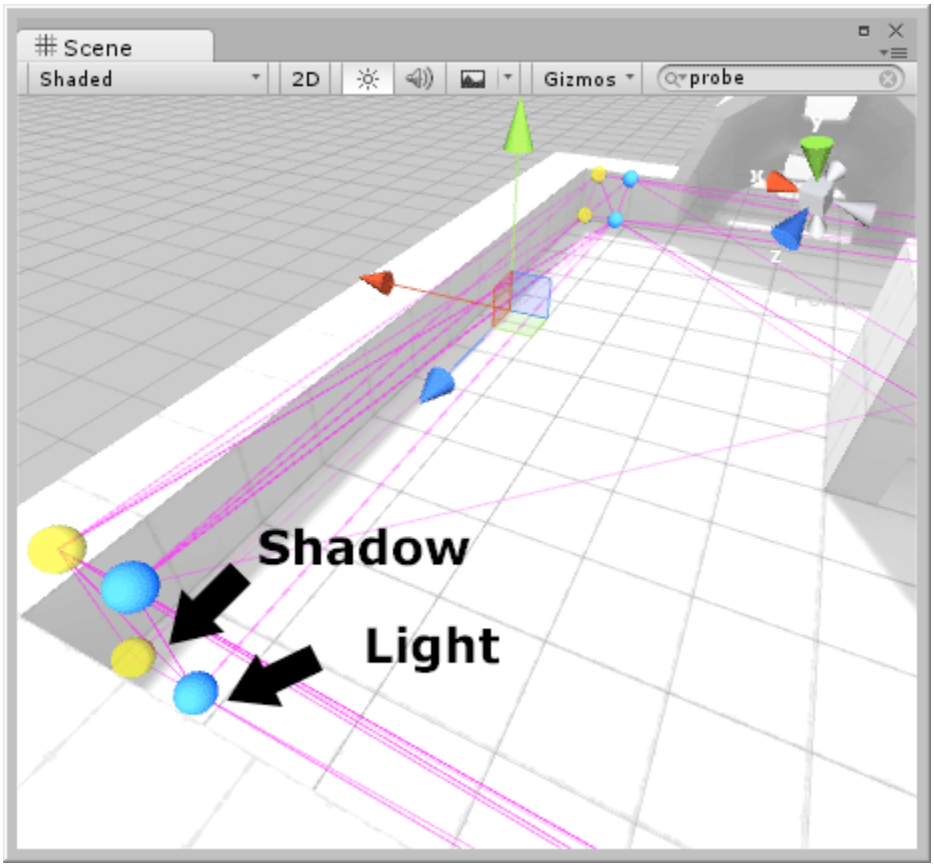
Insert image 1362OT_06_36.png

13. Position the initial Light Probes at the corners of the top room of the level. To move the Probes around, simply click and drag them.



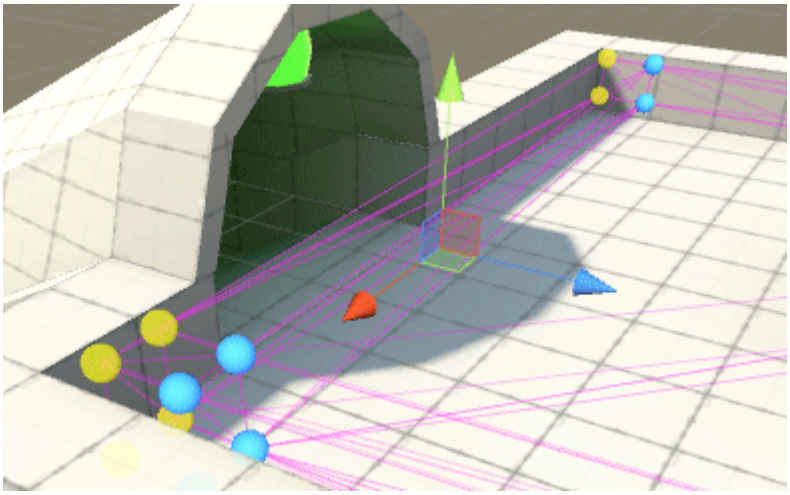
Insert image 1362OT_06_37.png

14. Select the four probes to the left side of the tunnel's entrance. Then, duplicate them by clicking the appropriate button on the **Inspector** view or, alternatively, use the *Ctrl/Cmd +D* keys. Finally, drag the new probes slightly to the right, to the point they are no longer over the shadow projected by the wall.



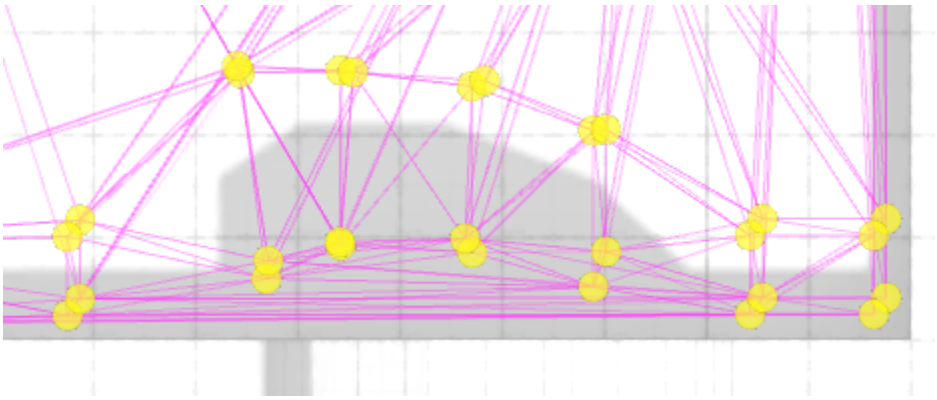
Insert image 1362OT_06_38.png

15. Repeat the last step, this time duplicating the probes next to the tunnel's entrance and bringing them inwards the group. To delete selected probes, either use the respective button on the **Light Probe Group** component or use the *Ctrl/Cmd + Backspace* keys.



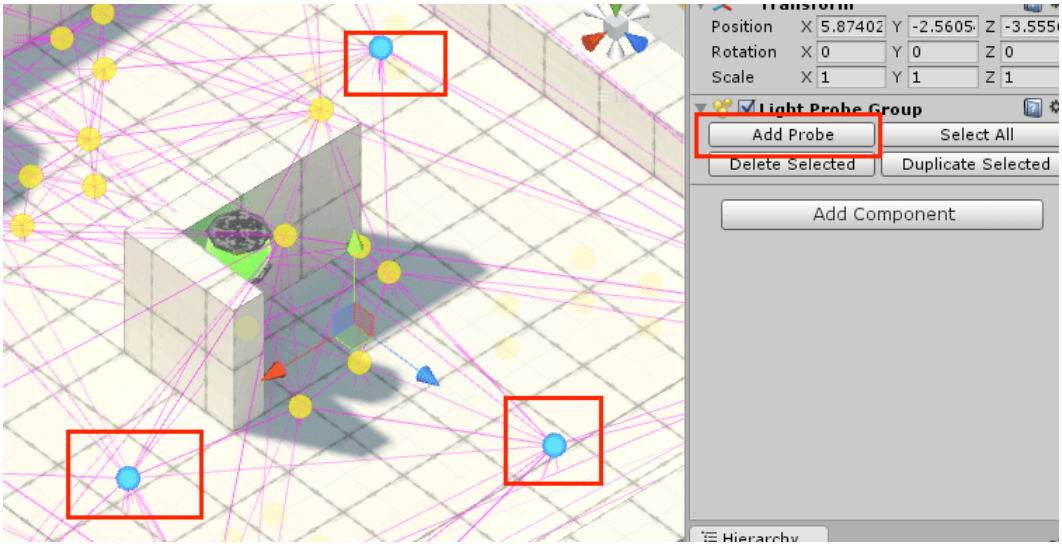
Insert image 1362OT_06_39.png

16. Duplicate and reposition the four probes that are nearest to the tunnel, repeating the operation five times and conforming each duplicate set to the shadow projected by the tunnel.



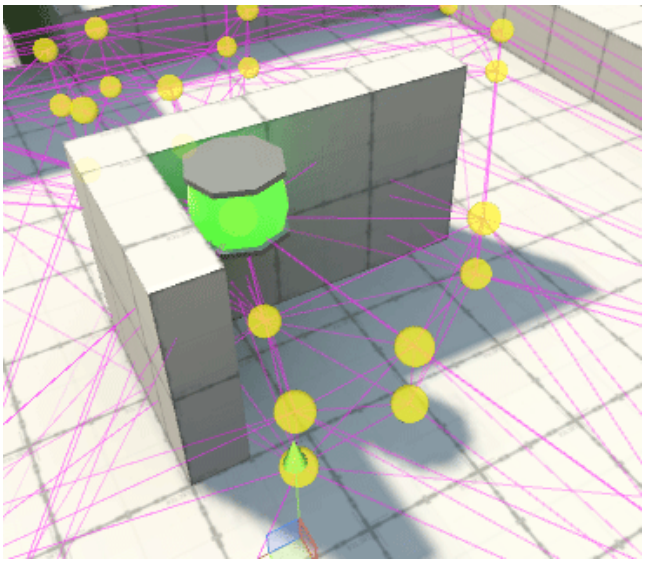
Insert image 1362OT_06_40.png

17. Use the **Add Probe** button to place three probes over well-lit areas of the scene.



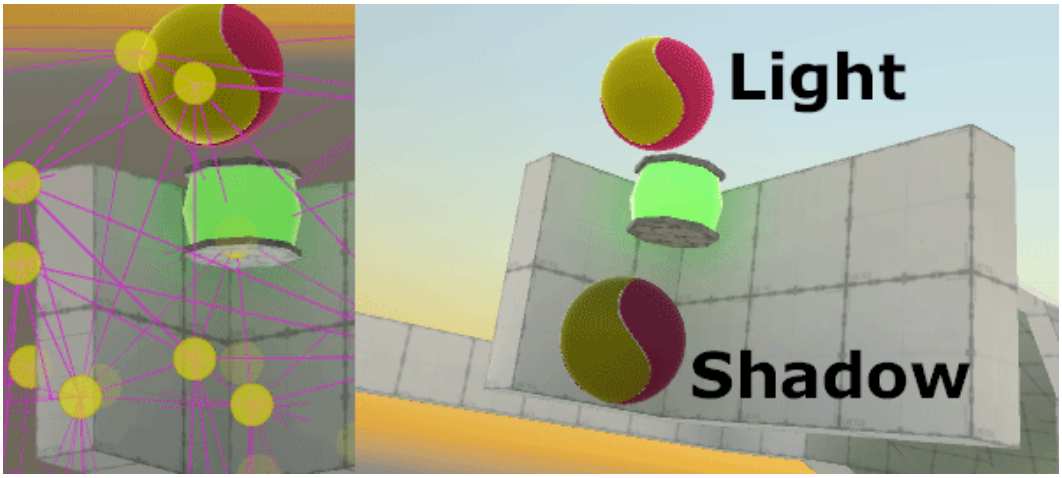
Insert image 1362OT_06_41.png

18. Now, add light probes within the shadow projected by the L-shaped wall.



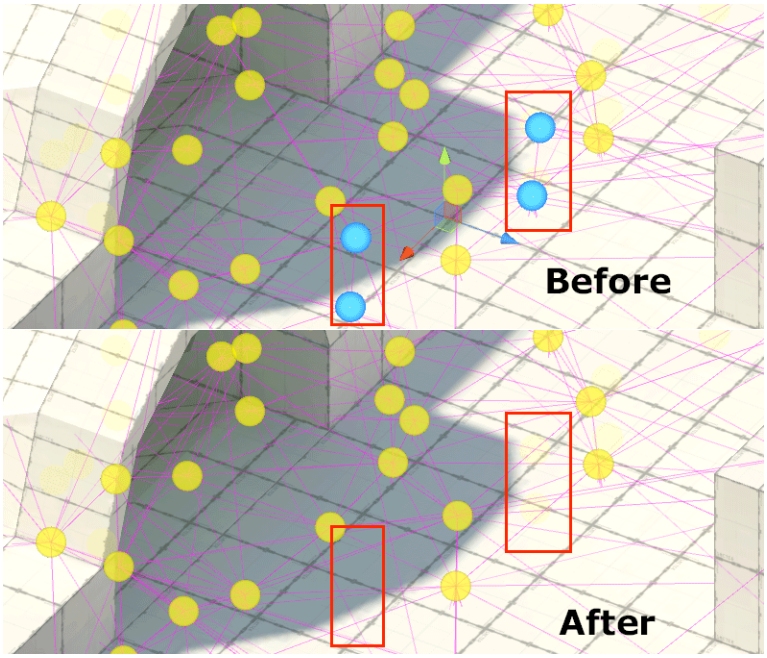
Insert image 1362OT_06_42.png

19. Since the Rollerball is able to jump, place the higher probes even higher, so they will sample the lighting above the shadowed areas of the scene.



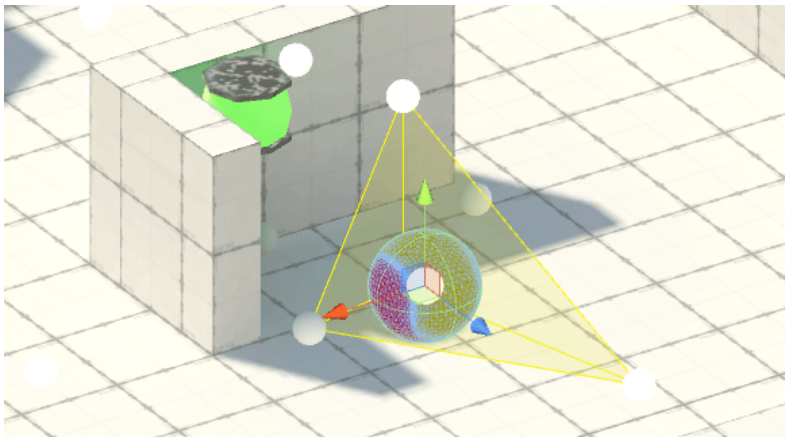
Insert image 1362OT_06_45.png

20. Placing too many Light Probes on a scene might be memory-intensive. Try optimizing the Light Probes group by removing probes from regions the player won´t have access to. Also, avoid overcrowding regions of continuous lighting conditions, removing probes that are too close to others in the same lighting condition.



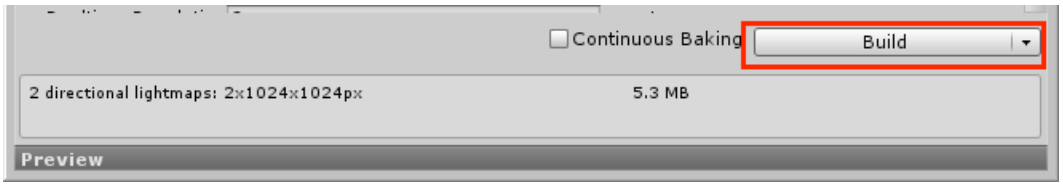
Insert image 1362OT_06_46.png

21. To check out which Light Probes are influencing the **Rollerball** at any place, move the **Rollerball** game object around the scene. A polyhedron will indicate which probes are being interpolated at that position.



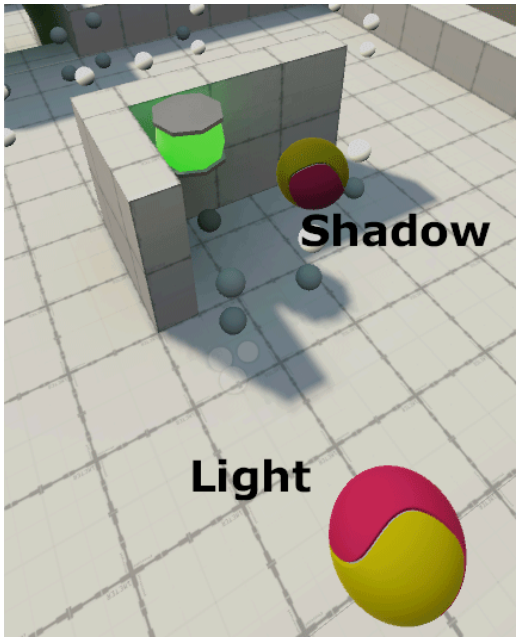
Insert image 1362OT_06_47.png

22. From the bottom of the **Lighting** window, click the **Build** button and wait the Lightmaps to be baked.



Insert image 1362OT_06_43.png

23. Test the scene. The Rollerball should be lit according to the light probes.



Insert image 1362OT_06_44.png

24. Keep adding probes until the level is completely covered.

How it works...

Lighmaps are basically texture maps including scene lights/shadows, Global Illumination, indirect illumination and objects featuring Emissive materials. They can be generated automatically or on-demand by Unity's lighting engine. However there are some points you should pay attention to, such as:

- Set all non-moving objects and lights to be baked as **Static**
- Set the game lights as **Baked**
- Set the scene's **Ambient GI** as **Baked**
- Set the **Global Illumination** option of emissive materials as **Baked**
- **Generate Light UVs** for all 3D meshes (specially the imported ones)
- Either **Build** the Lightmaps manually from the **Lighting** window or set the **Continuous Baking** option checked.

Light Probes work by sampling the scene's illumination at the point they're placed at. A dynamic object that has **Use Light Probes** enabled has its lighting determined by the interpolation between lighting values of the four Light Probes defining a volume around

it (or, in case there are no probes suited to define a volume around the dynamic object, a triangulation between the nearest probes is used).

It is important to notice that even if you are working on a level that is flat, you shouldn't place all your probes on the same level, as Light Probe Groups should form a volume in order to the interpolation be calculated correctly. This and much other information on the subject can be found in Unity's documentation at `docs.unity3d.com/Manual/LightProbes.html`.

There's more...

In case you can spare some processing power, you can exchange the use of Light probes for a **Mixed** Light. Just delete the **Light Probe Group** from your scene, select the **Directional Light** and, from the **Light** component, change **Baking** to **Mixed**. Then, set **Shadow Type** as **Soft Shadows** and **Strength** as `0.5`. Finally, click the **Build** button and wait for the Lightmaps to be baked. Realtime light/shadows will be cast into/from dynamic objects such as the **Rollerball**.

Insert image 1362OT_06_49.png

Conclusion

This chapter aimed to present you to some of Unity's new features in lighting, and occasionally teach you a few tricks with lights and effects. By now, you should be familiar with some of the concepts introduced by Unity 5, comfortable with a variety of techniques and, hopefully, willing to explore deeper some of the functionalities discussed throughout the recipes.

As always, Unity's documentation on the subject is excellent, so we encourage you to go back to the recipes and follow the provided URLs.