HOUR 5
Lights and Cameras

What You'll Learn in This Hour:

- How to work with lights in Unity
- The core elements of cameras
- How to work with multiple cameras in a scene
- How to work with layers

In this hour, you'll learn to use lights and cameras in Unity. You'll start by looking at the main features of lights. You'll then explore the different types of lights and their unique uses. When you are finished with lights, you'll begin working with cameras. You'll learn how to add new cameras, place them, and generate interesting effects with them. You'll finish this lesson by learning about layers in Unity.

Lights

In any form of visual media, lights go a long way in defining how a scene is perceived. Bright, slightly yellow light can make a scene look sunny and warm. Take the same scene and give it a low-intensity blue light, and it will look eerie and disconcerting. The color of the lights will also mix with the color of the skybox to give even more realistic-looking results.

Most scenes that strive for realism or dramatic effect employ at least one light (and often many). In previous hours, you briefly worked with lights to highlight other elements. In this section, you work with lights more directly.

NOTE

Repeat Properties

The different lights share many of the same properties. If a light has a property that has already been covered under a different light type, it isn't covered again. Just remember that if two different light types have properties with the same names, those properties do the same thing.

NOTE

What Is a Light?

In Unity, lights are not objects themselves. Instead, a light is a component. This means that when you add a light to a scene, you are really just adding a game object with the Light component, which can be any of the types of light you can use.

Baking Versus Real Time

Before you get started actually working with lights, you need to understand two main ways of using light: baked and real time. The thing to keep in mind is that all light in games is more or less computational. Light has to be figured out by the machine in three steps:

1. The color, direction, and range of simulated light rays are calculated from a light source.
2. When the light rays hit a surface, they illuminate and change the color of the surface.
3. The angle of impact with the surface is calculated, and the light bounces. Steps 1 and 2 are repeated again and again (depending on the light settings). With each bounce, the properties of the light rays are changed by the surfaces they hit (just like in real life).

This process is repeated for every light in every frame and creates global illumination (where objects receive light and color based on the objects around them). This process is helped a little by the feature Precomputed Realtime GI, which is turned on by default and requires no effort from you. With this feature, a part of the light calculation process described above is calculated before the scene starts so only part of the calculations need to be done at runtime. You may have already seen this in operation. If you have ever opened a Unity scene for the first time and noticed that the scene elements are dark for a moment, you have seen the precalculation process.

Baking, on the other hand, refers to the process of completely precalculating light and shadow for textures and objects during creation. You can do this with Unity or with a graphical editor. For instance, if you were to make a wall texture with a dark spot on it that resembled a human shadow and then put a human model next to the wall it was on, it would seem like the model was casting a shadow on the wall. The truth is, though, that the shadow was “baked” into the texture. Baking can make your games run much more quickly because the engine doesn’t have to calculate light and shadow every single frame; however, baking isn’t very necessary for your current needs because the games discussed in this book are not complex enough to require it.

Point Lights

The first light type you will be working with is the point light. Think of a point light as a light bulb. All light is emitted from one central location out in every direction. The point light is the most common type of light for illuminating interior areas.

To add a point light to a scene, select **GameObject > Light > Point Light**. Once in the scene, the Point Light game object can be manipulated just like any other. Table 5.1 describes the point light properties.

TABLE 5.1 Point Light Properties

Property	**Description**
Type	Specifies the type of light that the component gives off. Because this is a point light, the type should be Point. Changing the Type property changes the type of light.
Range	Dictates how far the light shines. Illumination fades evenly from the source of light to the range dictated.
Color	Specifies the color the light shines. Color is additive, which means if you shine a red light on a blue object, it will end up purple.
Mode	Determines if the light is a real-time light, a baked light, or a mixture of the two.
Intensity	Dictates how brightly a light shines. Note that the light still shines only as far as the Range property dictates

	only as far as the Range property dictates.
Indirect Multiplier	Determines how bright the light is after it bounces off objects. (Unity supports Global Illumination, meaning it calculates the results of bouncing light.)
Shadow Type	Specifies how shadows are calculated for this source in a scene. Soft shadows are more realistic but are also more performance intensive.
Cookie	Accepts a cubemap (like a skybox) that dictates a pattern for the light to shine through. Cookies are covered in more detail later in this hour.
Draw Halo	Determines whether a glowing halo appears around the light. Halos are covered in more detail later in this hour.
Flare	Accepts a light flare asset and simulates the effect of a bright light shining into a camera lens.
Render Mode	Determines the importance of this light. The three settings are Auto, Important, and Not Important. An important light is rendered in higher quality, whereas a less-important light is rendered more quickly. Use the Auto setting for now.
Culling Mask	Determines what layers are affected by the light. By default, everything is affected by the light. Layers are covered in detail later in this hour.

▼ TRY IT YOURSELF

Adding a Point Light to a Scene

Follow these steps to build a scene with some dynamic point lighting:

1. Create a new project or scene and delete the directional light that is there by default.
2. Add a plane to the scene (by selecting **GameObject > 3D Object > Plane**). Ensure that the plane is positioned at (0, .5, 0) and rotated to (270, 0, 0). The plane should be visible to the camera, but only from one side in the Scene view.
3. Add two cubes to the scene. Position them at (-1.5, 1, -5) and (1.5, 1, -5).

4. Add a point light to the scene (by selecting **GameObject > Light > Point Light**). Position the point light at (0, 1, -7). Notice how the light illuminates the inner sides of the cubes and the background plane (see Figure 5.1).

5. Set the light's shadow type to **Hard Shadows** and try moving it around. Continue exploring the light properties. Be sure to experiment with the light color, range, and intensity.

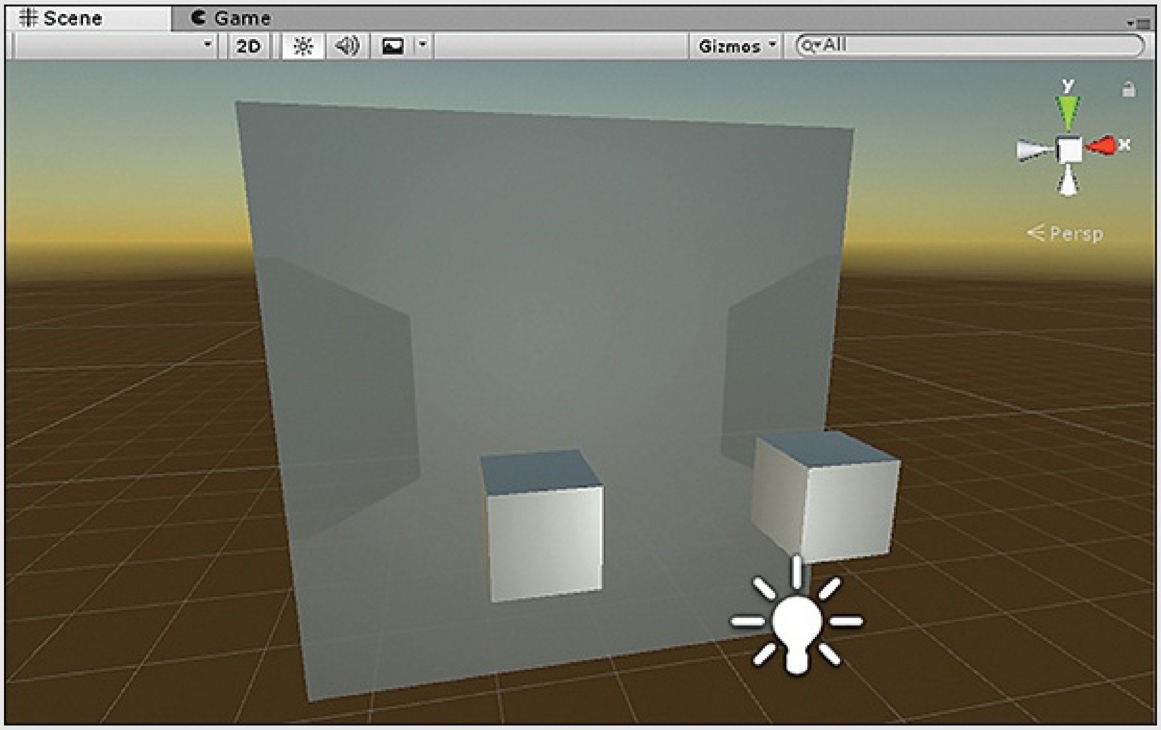


FIGURE 5.1
The results of this Try It Yourself.

Spotlights

Spotlights work a lot like the headlights of a car or flashlights. The light of a spotlight begins at a central spot and radiates out in a cone. In other words, spotlights illuminate whatever is in front of them while leaving everything else in the dark. Whereas a point light sends light in every direction, you can aim a spotlight.

To add a spotlight to a scene, select **GameObject > Create Other > Spotlight**. Alternatively, if you already have a light in your scene, you can change its type to **Spot**, and it becomes a spotlight.

Spotlights have only one property not already covered: Spot Angle. The Spot Angle property determines the radius of the cone of light emitted by the spotlight.

▼ TRY IT YOURSELF

Adding a Spotlight to a Scene

You now have a chance to work with spotlights in Unity. For brevity, this exercise uses the project created in the previous Try It Yourself for point lights. If you have not completed that exercise, do so and then continue with the following steps:

1. Duplicate the Point Light scene from the previous project (by selecting **Edit > Duplicate**) and name the new scene **Spotlight**.
2. Right-click **Point light** in the Hierarchy view and select **Rename**. Rename the object **Spotlight**. In the Inspector, change the Type property to **Spot**. Position the light object at (0, 1, -13).
3. Experiment with the properties of the spotlight. Notice how the range, intensity, and spot angle shape and change the effect of the light.

Directional Lights

The last type of light you'll work with in this hour is the directional light. A directional light is similar to a spotlight in that it can be aimed. Unlike a spotlight, though, a directional light illuminates an entire scene. You can think of a directional light as similar to the sun. In fact, you used a directional light as a sun in Hour 4, "Terrain and Environments." The light from a directional light radiates evenly in parallel lines across a scene.

A new scene comes with a directional light by default. To add a new directional light to a scene, select **GameObject > Light > Directional Light**. Alternatively, if you already have a light in a scene, you can change its type to Directional, and it becomes a directional light.

Directional lights have one additional property that hasn't been covered yet: Cookie Size. Cookies are covered later in this hour, but basically this property controls how big a cookie is and thus how many times it is repeated across a scene.

▼ TRY IT YOURSELF

Adding a Directional Light to a Scene

You will now add a directional light to a Unity scene. Once again, this exercise builds on the project created in the previous Try It Yourself. If you have not completed that exercise, do so, and then follow these steps:

1. Duplicate the Spotlight scene from the previous project (by selecting **Edit > Duplicate**) and name the new scene **Directional Light**.

2. Right-click **Spotlight** in the Hierarchy view and select **Rename**. Rename the object **Directional Light**. In the Inspector, change the Type property to **Directional**.

3. Change the light's rotation to (75, 0, 0). Notice how the sky changes when you rotate the light. This is due to the scene using a procedural skybox. Skyboxes are covered in more detail in Hour 6, "Game 1: *Amazing Racer*."

4. Notice how the light looks on the objects in the scene. Now change the light's position to (50, 50, 50). Notice that the light does not change. Because the directional light is in parallel lines, the position of the light does not matter. Only the rotation of a directional light matters.

5. Experiment with the properties of the directional light. There is no range (because range is infinite), but notice how the color and intensity affect the scene.

NOTE

Area Lights and Emissive Materials

There are two more light types that are not covered in this text: area lights and emissive materials.

An area light is a feature that exists for a process called *lightmap baking*. These topics are more advanced than what you need for basic game projects so are not covered in this book. If you want to learn more about this, see Unity's wealth of online documentation.

An emissive material is a material applied to an object that actually transmits light. This type of light could be very useful for a TV screen, indicator lights, and so on.

Creating Lights Out of Objects

Because lights in Unity are components, any object in a scene can be a light. To add a light to an object, first select the object. Then, in the Inspector view, click the **Add Component** button. A new list should pop up. Select **Rendering** and then **Light**. Now your object has a light component. An alternative way to add a light to an object is to select the object and select **Component > Rendering > Light**.

Note a couple things about adding lights to objects. First, an object will not block the light. This means that putting a light inside a cube will not stop the light from radiating. Second, adding a light to an object does not make it glow. The object itself will not look like it is giving off light, but it is.

Halos

Halos are glowing circles that appear around lights in foggy or cloudy conditions (see Figure 5.2). They occur because light is bouncing off small particles all around the light source. In Unity, you can easily add halos to lights. Each light has a Draw Halo check box. If it is checked, a halo is drawn for the light. If you can’t see the halo, you may be too close to the light, so try backing up a bit.

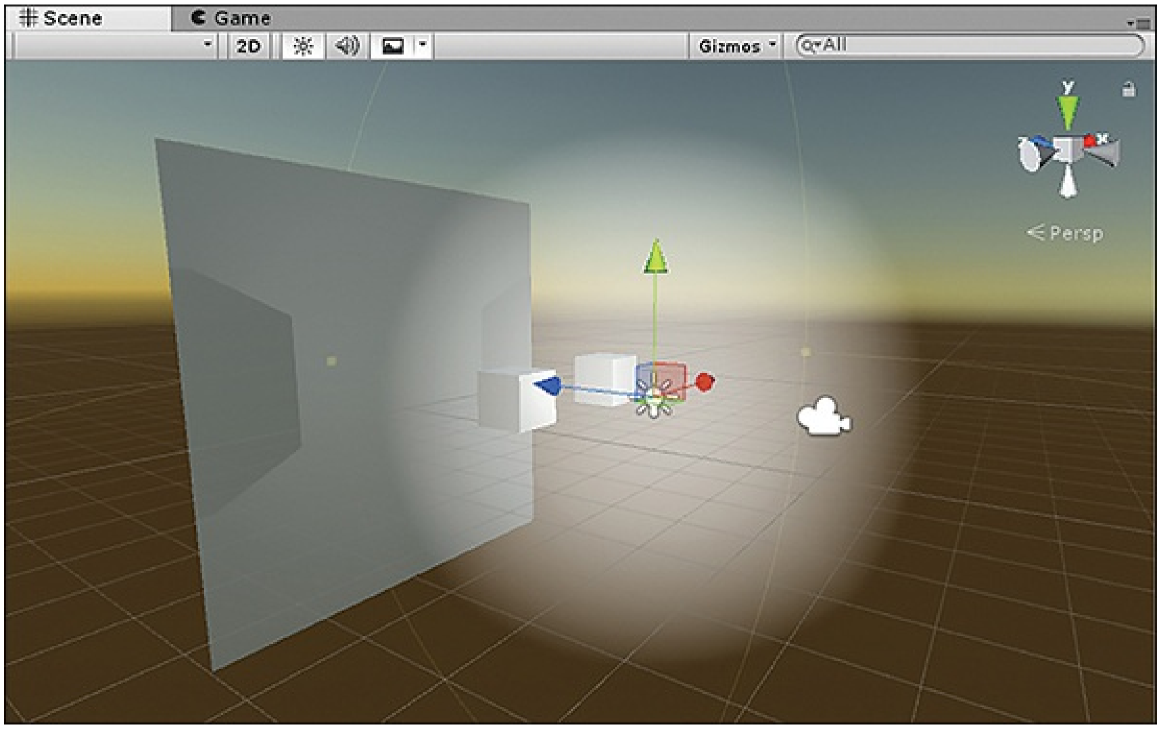
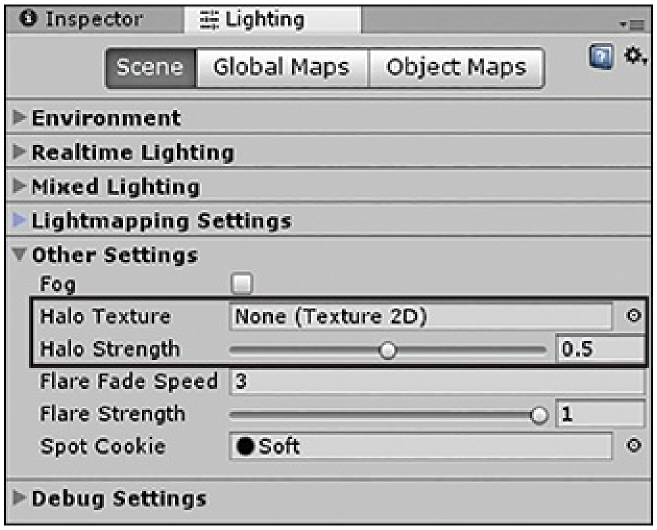


FIGURE 5.2
A halo around a light.

The size of a halo is determined by the light's range. The bigger the range, the bigger the halo. Unity also provides a few properties that apply to all halos in a scene. You can access these properties by selecting **Window > Lighting > Settings**. Expand **Other Settings**, and the settings then appear in the Inspector view (see Figure 5.3).



Fog	☐
Halo Texture	None (Texture 2D)
Halo Strength	0.5
Flare Fade Speed	3
Flare Strength	1
Spot Cookie	● Soft

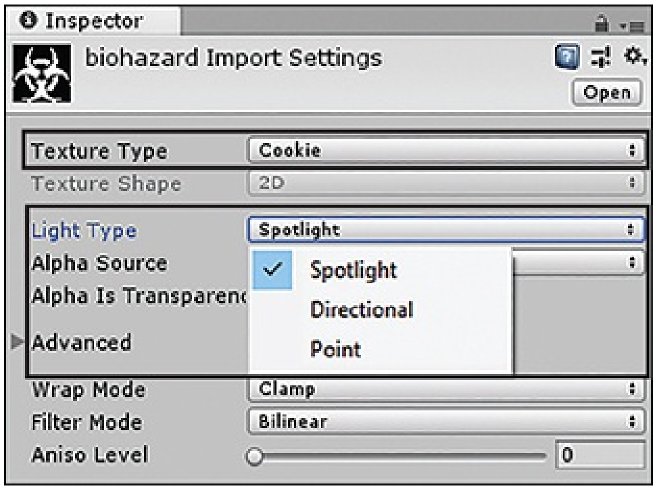
FIGURE 5.3
The scene lighting settings.

The Halo Strength property determines how big the halo will be, based on the light’s range. For instance, if a light has a range of 10 and a strength of 1, the halo will extend out all 10 units. If the strength is .5, the halo extends out only 5 units (10 × .5 = 5). The Halo Texture property allows you to specify a different shape for a halo by providing a new texture. If you do not want to use a custom texture for a halo, you can leave it blank, and the default circular one is used.

Cookies

If you have ever shone a light on a wall and then put your hand in between the light and the wall, you’ve probably noticed that your hand blocks some of the light, leaving a hand-shaped shadow on the wall. You can simulate this effect in Unity by using cookies. Cookies are special textures that you can add to lights to dictate how the light radiates. Cookies differ a little for point, spot, and directional lights. Spotlights and directional lights both use black-and-white flat textures for cookies. Spotlights don’t repeat the cookies, but directional lights do. Point lights also use black-and-white textures, but this type of light must be placed in a cubemap. A cubemap is six textures placed together to form a box (like a skybox).

Adding a cookie to a light is a fairly straightforward process. You simply apply a texture to the Cookie property of the light. The trick to getting a cookie to work is setting up the texture correctly ahead of time. To set up the texture correctly, select it in Unity and then change its properties in the Inspector window. Figure 5.4 shows the property settings for turning a texture into a cookie.


Inspector

biohazard Import Settings

Open

Texture Type: Cookie
Texture Shape: 2D
Light Type: Spotlight
Alpha Source
Alpha Is Transparenc
Advanced
Wrap Mode: Clamp
Filter Mode: Bilinear
Aniso Level: 0

FIGURE 5.4
The texture properties of cookies for point, spot, and directional lights.

The texture properties of cookies for point, spot, and directional lights.

▼ TRY IT YOURSELF

Adding a Cookie to a Spotlight

This exercise requires the biohazard.png image, available in the book assets for Hour 5. Follow these steps to add a cookie to a spotlight so that you can see the process from start to finish:

1. Create a new project or scene. Delete the directional light from the scene.
2. Add a plane to the scene and position it at (0, 1, 0) with a rotation of (270, 0, 0).
3. Add a spotlight to the Main Camera by selecting **Main Camera** and then clicking **Component > Rendering > Light** and changing the type to **Spot**. Set the range to **18**, the spot angle to **40**, and the intensity to **3**.
4. Drag the biohazard.png texture from the book assets into your Project view. Select the texture, and in the Inspector view, change the texture type to **Cookie**, set the light type to **Spotlight**, and set the alpha source to **From Grayscale**. This makes the cookie block light where it's black.
5. With the Main Camera selected, click and drag the biohazard texture into the Cookie property of the light component. You should see the biohazard symbol projected onto the plane (see Figure 5.5).
6. Experiment with different ranges and intensities of the light. Rotate the plane and see how the symbol warps and distorts.

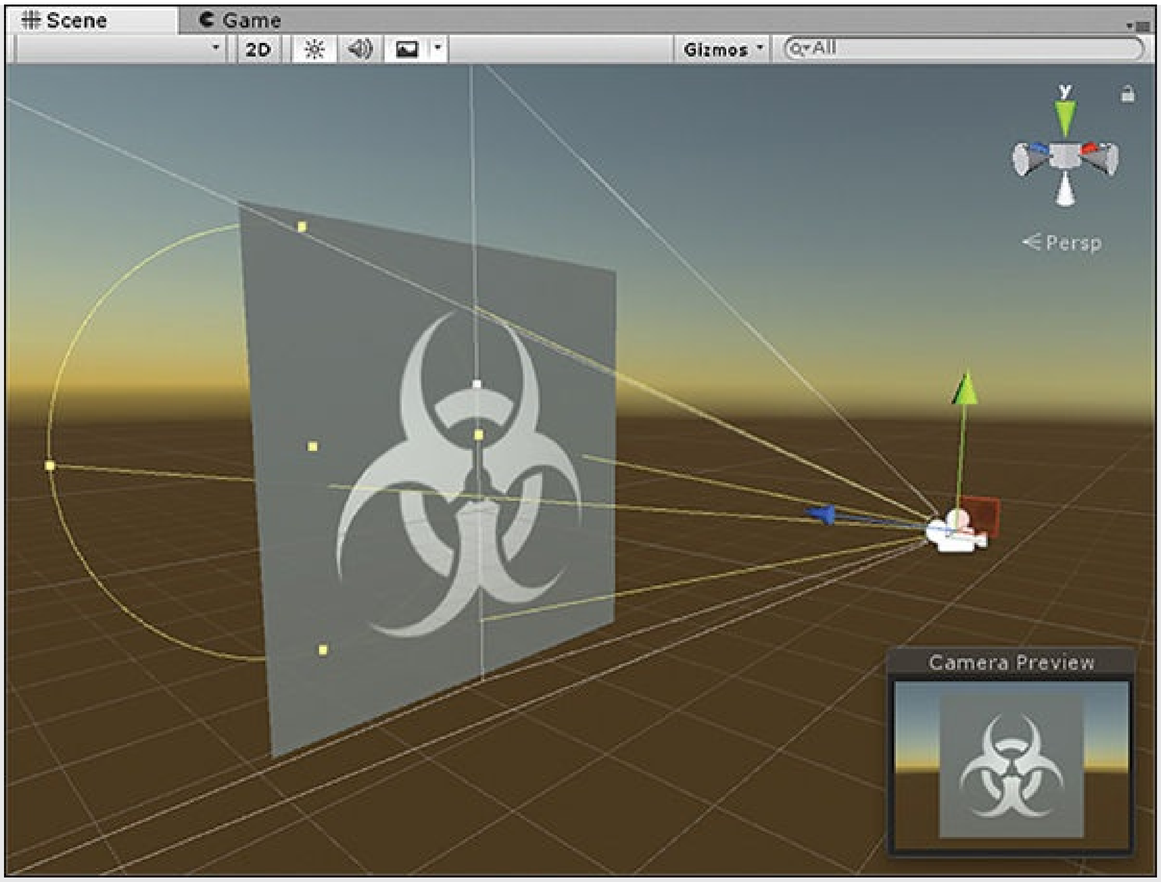


FIGURE 5.5
Spotlight with a cookie.

Cameras

The camera is the player's view into the world. It provides perspective and controls how things appear to the player. Every game in Unity has at least one camera. In fact, a camera is always added for you whenever you create a new scene. The camera always appears in the hierarchy as Main Camera. In this section, you'll learn all about cameras and how to use them for interesting effects.

Anatomy of a Camera

All cameras share the same set of properties that dictate how they behave. Table 5.2 describes all the camera properties.

TABLE 5.2 Camera Properties

Property	Description
Clear Flags	Determines what the camera displays in the areas where there are no game objects. The default is Skybox. If there is no skybox, the camera defaults to a solid color. Depth Only should be used only when there are multiple cameras. Don’t Clear causes streaking and should be used only if you’re writing a custom shader.
Background	Specifies the background color if there is no skybox present.
Culling Mask	Determines what layers are picked up by the camera. By default, the camera sees everything. You can uncheck certain layers (more on layers later in this hour), and they won’t be visible to the camera.
Projection	Determines how the camera sees the world. The two options are Perspective and Orthographic. Perspective cameras perceive the world in 3D, with closer objects being larger and farther objects being smaller. This is the setting to use if you want depth in your game. The Orthographic setting ignores depth and treats everything as flat.
Field of View	Specifies how wide an area the camera can see.
Clipping Planes	Specifies the range where objects are visible to the camera. Objects that are closer than the near plane or farther from the far plane will not be seen.
View Port Rect	Establishes what part of the actual screen the camera is projected on. (View Port Rect is short for View Port Rectangle.) By default, x and y are both set to 0, which causes the camera to start in the lower left of the screen. The width and height are both set to 1, which causes the camera to cover 100% of the screen vertically and horizontally. This is discussed in more detail later in this hour.
Depth	Specifies the priority for multiple cameras. Lower numbers are drawn first, which means higher numbers may be drawn on top and may effectively hide them.
Rendering Path	Determines how the camera renders. It should be left set to Use Player Settings.

Target Texture	Enables you to specify a texture for the camera to draw to instead of the screen.
Occlusion Culling	Disables rendering of objects when they are not currently seen by the camera because they are obscured (occluded) by other objects.
Allow HDR	Determines whether Unity’s internal light calculations are limited to the basic color range. (HDR stands for Hyper-Dynamic Range.) This property allows for advanced visual effects.
Allow MSAA	Enables a basic, but efficient, type of antialiasing called MultiSample antialiasing. Antialiasing is a method of removing pixelated edges when rendering graphics.
Allow Dynamic Resolution	Allows for dynamic resolution adjustment for console games.

Cameras have many properties, but you can set most and then forget about them. Cameras also have a few extra components. The flare layer allows a camera to see the lens flares of lights, and the audio listener allows the camera to pick up sound. If you add more cameras to a scene, you need to remove their audio listeners. There can be only one audio listener per scene.

Multiple Cameras

Many effects in modern games would not be possible without multiple cameras. Thankfully, you can have as many cameras as you want in a Unity scene. To add a new camera to a scene, select **GameObject > Camera**. Alternatively, you can add the camera component to a game object that is already in your scene. To do that, select the object and click **Add Component** in the Inspector. Select **Rendering > Camera** to add the camera component. Remember that adding a camera component to an existing object does not automatically give you the flare layer or audio listener components.

CAUTION

Multiple Audio Listeners

As mentioned earlier, a scene can have only a single audio listener. In older versions of Unity, having two or more listeners would cause an error and prevent a scene from running. Now, if you have multiple listeners, you just prevent a scene from running. Now, if you have multiple listeners, you just see a warning message, although audio might not be heard correctly. This topic is covered in detail in Hour 21, “Audio.”

▼ TRY IT YOURSELF

Working with Multiple Cameras

The best way to understand how multiple cameras interact is to get some practice working with them. This exercise focuses on basic camera manipulation:

1. Create a new project or scene and add two cubes. Place the cubes at (-2, 1, -5) and (2, 1, -5).

2. Move the Main Camera to (-3, 1, -8) and change its rotation to (0, 45, 0).

3. Add a new camera to the scene (by selecting **GameObject > Camera**) and position it at (3, 1, -8). Change its rotation to (0, 315, 0). Be sure to disable the audio listener for the camera by unchecking the box next to the component.

4. Run the scene. Notice that the second camera is the only one displayed. This is because the second camera has a higher depth than the Main Camera. The Main Camera is drawn to the screen first, and then the second camera is drawn on top of it. Change the main camera depth to **1** and then run the scene again. Notice that the Main Camera is now the only one visible.

Split-Screen and Picture-in-Picture

As you saw earlier in this hour, having multiple cameras in a scene doesn’t do much good if one of them simply draws over the other one. In this section, you’ll learn to use the Normalized View Port Rect property to achieve split-screen and picture-in-picture effects.

The normalized view port basically treats the screen as a simple rectangle. The lower-left corner of the rectangle is (0, 0), and the upper-right corner is (1, 1). This does not mean that the screen has to be a perfect square. Instead, think of the coordinates as percentages of the size. So, a coordinate of 1 means 100%, and a coordinate of .5 means 50%. When you know this, placing cameras on the and a coordinate of .5 means 50%. When you know this, placing cameras on the screen becomes easy. By default, cameras project from (0, 0) with a width and height of 1 (or 100%). This causes them to take up the entire screen. If you were to change those numbers, however, you would get a different effect.

▼ TRY IT YOURSELF

Creating a Split-Screen Camera System

This exercise walks through creating a split-screen camera system. This type of system is common in two-player games where the players have to share the same screen. This exercise builds on the previous Try It Yourself for multiple cameras earlier in this hour. Follow these steps:

1. Open the project that you created in the preceding Try It Yourself.

2. Ensure that the Main Camera has a depth of -1. Ensure that the X and Y properties of the camera’s View Port Rect property are both **0**. Set the W and H properties to **1** and **.5**, respectively (that is, 100% of the width and 50% of the height).

3. Ensure that the second camera also has a depth of -1. Set the X and Y properties of the view port to (0, .5). This causes the camera to begin drawing halfway down the screen. Set the W and H properties to **1** and **.5**, respectively.

4. Run the scene and notice that both cameras are now projecting on the screen at the same time (see Figure 5.6). You can split the screen like this as many times as you want.

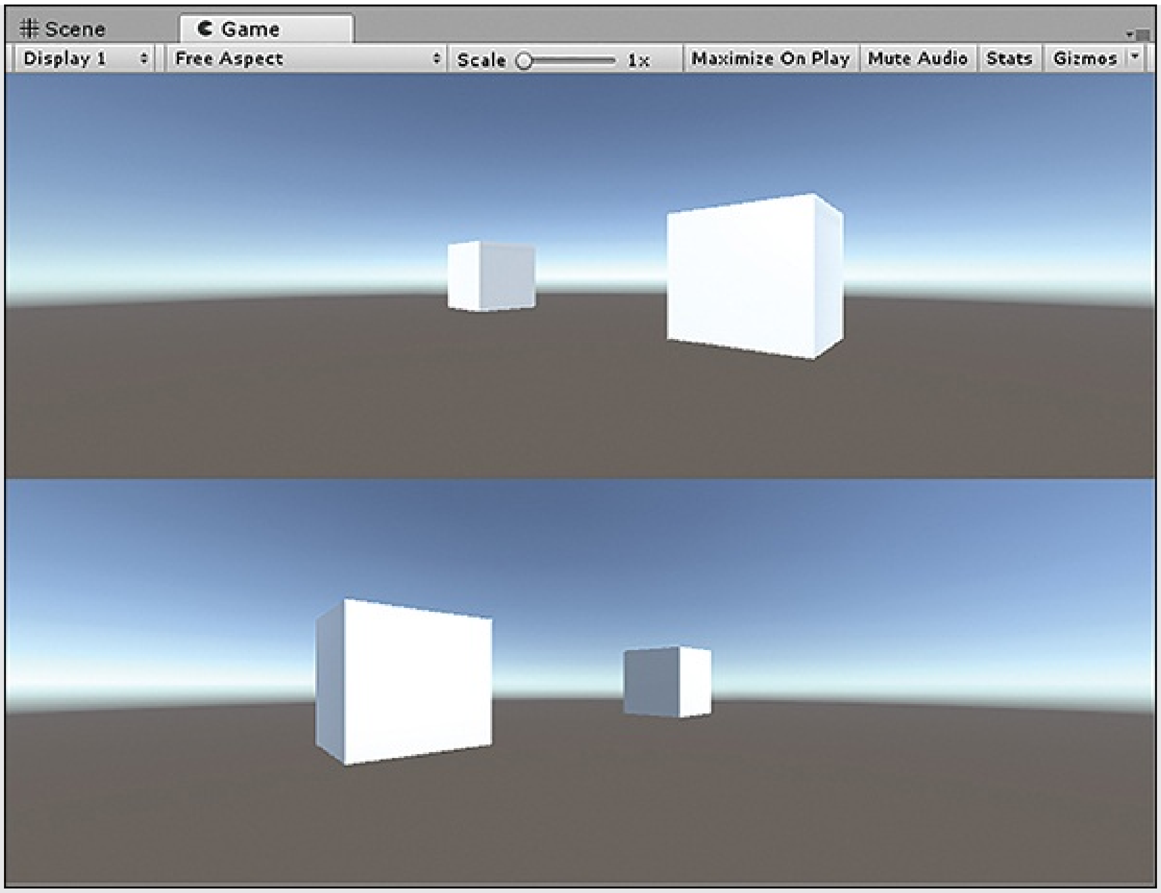


FIGURE 5.6
The split-screen effect.

▼ TRY IT YOURSELF

Creating a Picture-in-Picture Effect

Picture-in-picture is commonly used to create effects like minimaps. With this effect, one camera draws over another one in a specific area. This exercise builds on the previous Try It Yourself for multiple cameras earlier in this hour:

1. Open the project that you created in the Try It Yourself "Working with Multiple Cameras."
2. Ensure that the Main Camera has a depth of -1. Ensure that the X and Y properties of the camera's Normalized View Port Rect property are both **0** and the W and H properties are both **1**.

3. Ensure that the depth of the second camera is 0. Set the X and Y property of the view port to (.75, .75) and set both the W and the H values to **.2**.

4. Run the scene. Notice that the second camera appears in the upper-right corner of the screen (see Figure 5.7). Experiment with the different view port settings to get the camera to appear in the different corners.

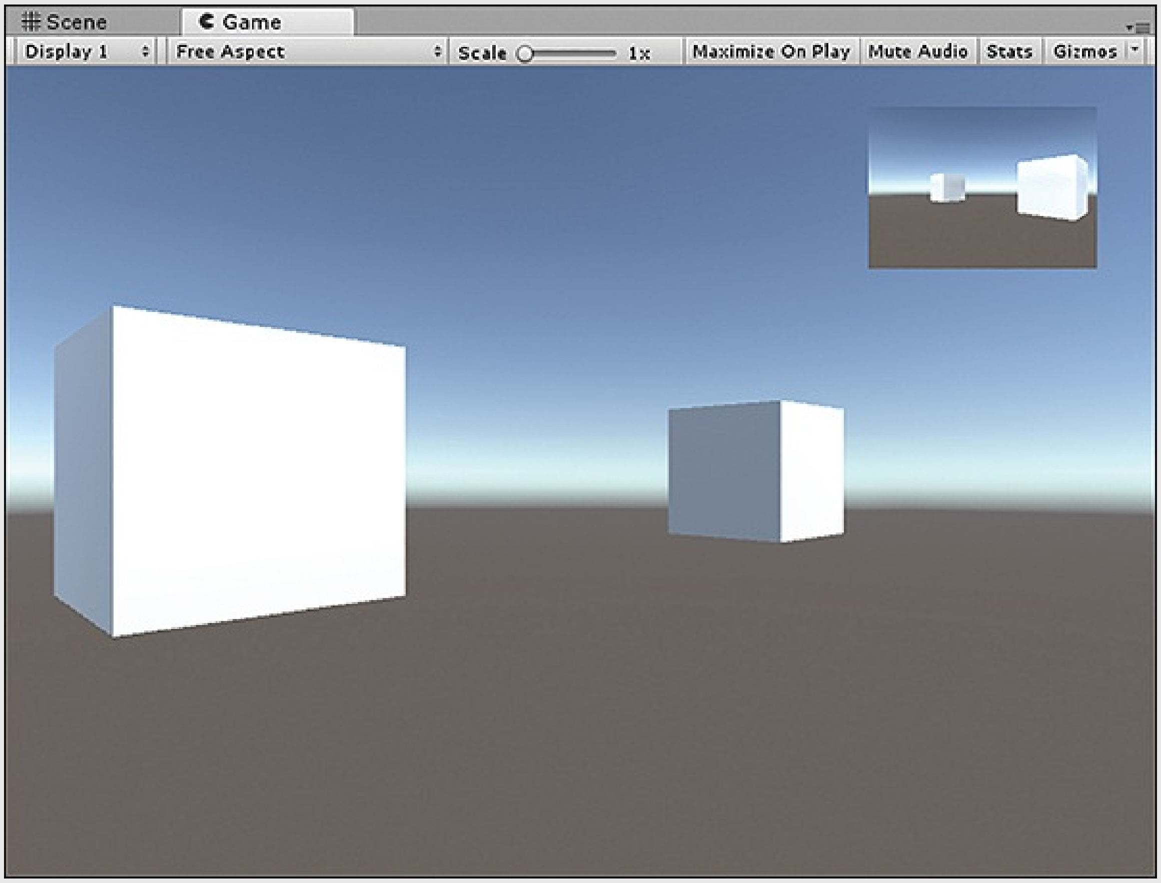


FIGURE 5.7
The picture-in-picture effect.

Layers

It can often be difficult to organize the many objects in a project and in a scene. Sometimes you want items to be viewable by only certain cameras or illuminated by only certain lights. Sometimes you want collisions to occur only between certain types of objects. Unity allows you to organize by using layers. Layers are groupings of similar objects that can be treated a certain way. By