Bonus Recipes

Creating an AssetBundle

There are times when we can't or don't want to store all game assets in our application build. Some data may only be available at runtime, such as new movie clips or sound files. We may wish to build into our game the ability to download new levels as they are made available on a web server, and so on. We may be creating games to run on hardware with limited memory (such as wireless VR devices). Another motivation for using AssetBundles is to separate the location of game assets from the code and Unity project, allowing game development teams to work independently on logic and media assets. So, AssetBundles stored on a network can allow the Unity game programmers to always have access to the latest version of the game assets, without any complicated archiving or exchange of files between the asset creators and the programmers. Unity provides several ways to allow a game to download new content at runtime, with the most popular and straightforward being Unity's AssetBundles. AssetBundles allow the packaging of non code game assets (textures, materials, models, audio, video, and so on) in a way that allows them to be downloaded and integrated into a game at runtime. One strong feature of AssetBundles is that they can refer to each other, so you could package some materials in one AssetBundle, and they could reference textures in another AssetBundle.

This recipe takes you through the steps for creating an AssetBundle. The following recipe shows how to write a game that can download an AssetBundle when the game is running.

Getting ready

This recipe assumes that you either have your own web hosting, or are running a local web server. You could use the built-in PHP web server, or a web server such as Apache or Nginx. If you have your own website, you could upload an AssetBundle to that website and see how a running game can download the assets from your web server.

How to do it...

To create an AssetBundle, follow these steps:

1. Create a new 3D Unity project.
2. Display the **Package Manager** panel (choose **Window** | **Package Manager**), and
   select **All** from the **Packages** drop-down menu:

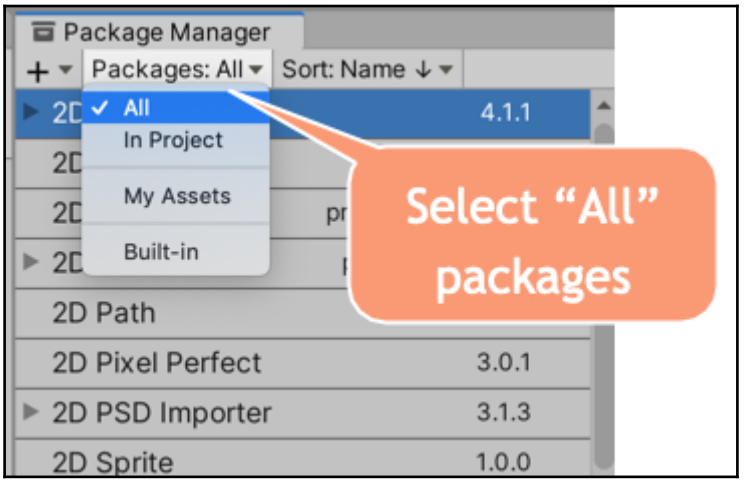

Figure 1.1 – Selecting All packages in the Package Manager panel

3. Enter **Asset** in the search box, select **AssetBundle Browser**, and then click the
   Install button:

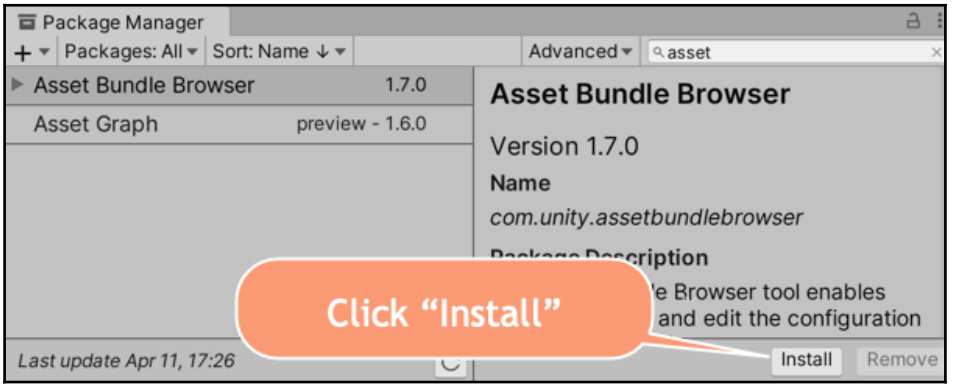

Figure 1.2 – Installing the AssetBundle Browser package

4. Your **Window** menu should now have a new option for **AssetBundle Browser**.
   Choose this menu option to open the **AssetBundles** panel:

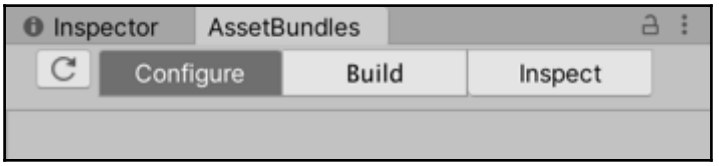

Figure 1.3 – The AssetBundles panel

5. In the default **Scene** window, create a new cube.
6. In the **Project** panel, create a new red Material, `m_red`, making it red by choosing that color for its **Albedo** property in the **Inspector** window. Apply `m_red` to the cube in the **Scene** window.
7. Now, drag the cube from the **Scene** window into the **Project** panel to create a new Prefab file. Name this Prefab `CubeRed`:

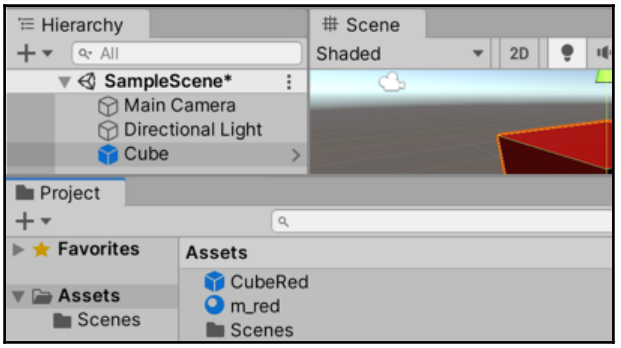

Figure 1.4 – Prefab CubeRed created in the Project panel

8. Create a new AssetBundle of our `CubeReb` Prefab by dragging the Prefab file from the **Project** panel into the **AssetBundles** panel:

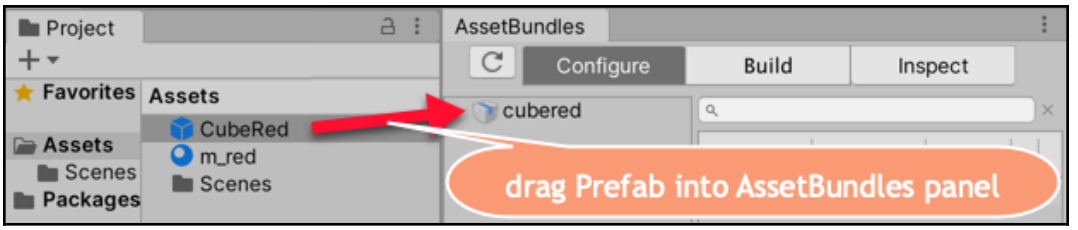

Figure 1.5 – Creating a new AssetBundle by dragging the Prefab into the AssetBundles panel

9. You should now have created a new AssetBundle named `cubered` - note that AssetBundles are spelled in lowercase letters.
10. In the **AssetBundles** panel, select the **Build** tab (the center one).

NOTE:
In order to build an AssetBundle, you must choose the appropriate build target (for example, Standalone Windows). You must have the build target installed as part of your Unity setup. You can always add more build targets by installing them through the Unity Hub application.

11. To build our AssetBundle, we need to simplify the output path to `AssetBundles`, check the box to C**opy to StreamingAssets**, and then click the **Build** button:

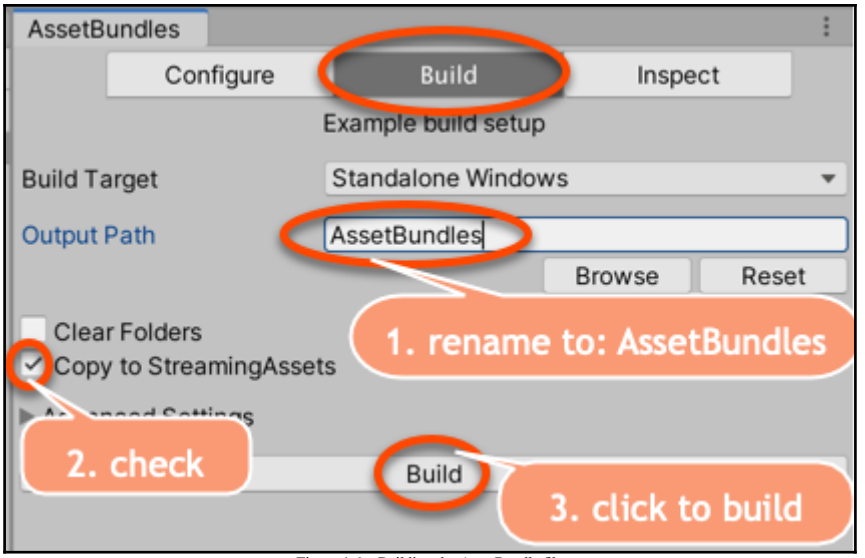

Figure 1.6 – Building the AssetBundle file

12. After Unity refreshes its view (which you can achieve by selecting **Assets** | **Refresh**), you should now see a new folder in your **Project** panel named `StreamingAssets`, containing files named `AssetBundles` and `cubered`:

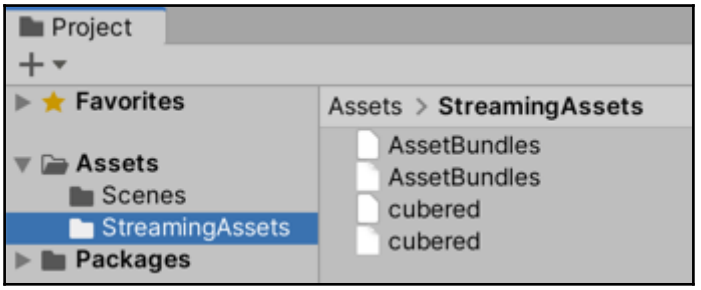

Figure 1.7 – The created StreamingAssets folder

13. You have now created an AssetBundle named `cubered`, containing the **Cube** Prefab, and its `m_red` Material.

How it works...

You can create an AssetBundle by creating project asset files, such as models, materials, and prefabs. In this recipe, you created a cube and a red Material, and from that a Prefab project file. You then used **AssetBundle Browser** to create an AssetBundle containing the Prefab and its associated `m_red` Material.

This AssetBundle can be loaded at runtime in this, or another, project. The following two recipes illustrate how to load AssetBundle resources at runtime from local files in a folder named StreamingAssets, and also from a web server. The folder name `StreamingAssets` is a special folder in Unity projects. Learn more from the Unity documentation at the following link: `https://docs.unity3d.com/Manual/StreamingAssets.html`

NOTE: It may look as if there are duplicate files in the `SteamingAssets` folder (see *Figure 8.35*). However, there are two files of each name, but they differ in their file extensions. For each file (**AssetBundles** and **cubered**), there is a second `.manifest` file - however, the Unity Editor hides the extension, which is why we see two files each with the same name.

There's more...

Here are some ways to go further with this recipe.

Creating AssetBundles using menu at the bottom of the Inspector window

As we saw above, one way to create AssetBundles is to drag an asset file into the AssetBundle Browser panel. Another way to create an AssetBundle is by selecting the Prefab, and then choosing New in the AssetBundle menu at the bottom of the Inspector window properties for the selected file, and entering a name in lowercase (for example, cubered):

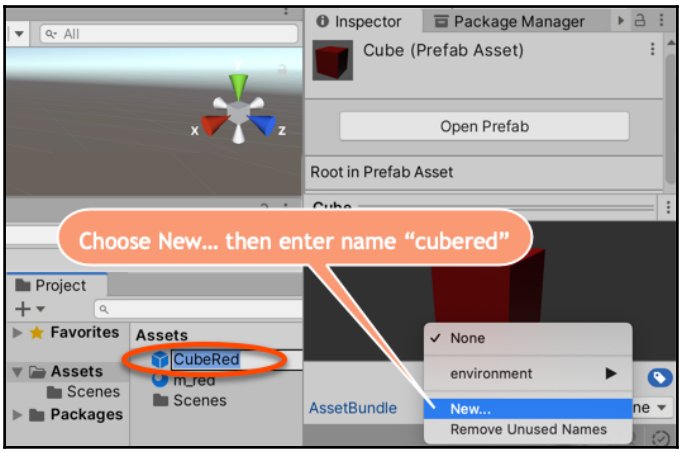

Figure 1.8 – Creating a new AssetBundle at the bottom of the Inspector window for selected asset files

Loading an AssetBundle from a local StreamingAssets folder

This recipe follows on from the previous one. Having created an AssetBundle in the previous recipe, we'll now create a new game to load and instantiate the red cube from the AssetBundle at runtime.

Getting ready

Make a copy of the previous recipe, and work on that copy for this recipe.

How to do it...

To load an AssetBundle from the `StreamingAssets` folder at runtime, follow these steps:

1. Create a new 3D Unity project.
2. Copy the StreamingAssets folder you created in the previous recipe into the Assets folder for this project.
3. Create a new empty GameObject named Loader.
4. Create a new C# script class named LoadFromAssetBundle, and add an instance object as a component to the Loader GameObject:

```
using System;
using UnityEngine;
using System.IO;
public class LoadFromAssetBundle : MonoBehaviour
{
    public String assetBundleName = "cubered";
    public String prefabName = "CubeRed";
    void Start()
    {
        var myLoadedAssetBundle = AssetBundle.LoadFromFile(
            Path.Combine(Application.streamingAssetsPath,
                         assetBundleName));
        if (myLoadedAssetBundle == null) {
            Debug.Log("Failed to load AssetBundle!");
            return;
        }
       // create and instantiate prefab - red Cube
       var prefab =
           myLoadedAssetBundle.LoadAsset<GameObject>(
               prefabName);
       Instantiate(prefab);
    }
}
```

5. An object instance of this script class needs two names to work, but we've set these as defaults in the code. The first is `assetBundleName`, the name of the AssetBundle to load from, inside the `StreamingAssets` folder. The second is `prefabName`, the name of the Prefab object inside the AssetBundle to be instantiated.
6. Run the Scene, and you should see a red cube appear – a GameObject created by
instantiating the cubered Prefab, loaded from the AssetBundle in the

`StreamingAssets` folder:

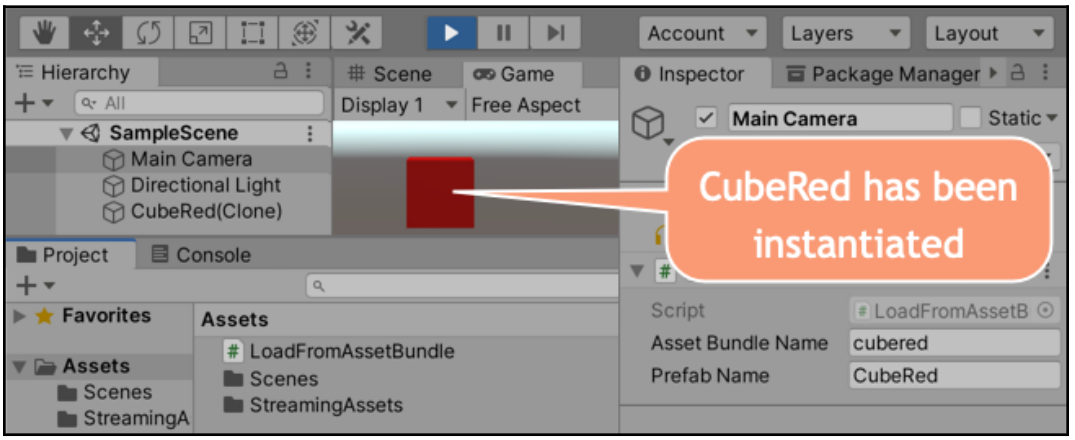

Figure 1.9 – The instantiated red cube, loaded from the AssetBundle in the SteamingAssets folder

How it works...

When the game runs, all GameObjects in the Scene (including `Loader`) are sent a `Start()` message. Since we've added an instance object of the `LoadFromAssetBundle` class to the `Loader` GameObject, and this has a `Start()` method, then this method is invoked. The AssetBundle is loaded into the `myLoadedAssetBundle` variable through the execution of the `AssetBundle.LoadFromFile(....)` method, with the path to the `StreamingAssets` folder and the name of the AssetBundle to load in the `assetBundleName` variable. The Prefab is created in the prefab variable through execution of the `LoadAsset(...)` method of `myLoadedAssetBundle`, being passed the name of the Prefab to create in the `prefabName` variable. Finally, a GameObject is created in the Scene by instantiating the Prefab in the final statement of the `Start()` method.

If the bundle fails to be read from the file, then the `myLoadedAssetBundle` variable will be `null`, in which case a message is logged and the method ends with no further action.

As we can see from *Figure 8.37*, the name of the GameObject added at runtime to the Scene has (Clone) added as a suffix, even when loaded from an AssetBundle. This automatic naming of `GameObject` instances created at runtime is a good example of

why we should **not** write code that assumes/relies on the name of a GameObject at runtime. Instead, our code should either directly store references to created GameObjects, or find them by unique tags and so on.

Note that the script in this recipe was based on the one Unity provides in their documentation page regarding AssetBundle workflows: `https://docs.unity3d.com/2020.1/Documentation/Manual/AssetBundles-Workflow.html`

Downloading an AssetBundle from a webserver

This recipe follows on from the previous one, but rather than loading an AssetBundle at runtime from a local `StreamingAssets` folder, we're going to place the AssetBundle on a web server and download it from there.

Getting ready

Make a copy of the previous recipe, and work on that copy for this recipe. This recipe involves downloading an AssetBundle from a web server, and then instantiating a GameObject in the Scene from a Prefab in the AssetBundle. So we need a web server online that is publishing our AssetBundle. At the time of writing, there is a raw option to allow a URI to directly refer to an AssetBundle in a GitHub repository. So, we can actually use the URI for the cubered AssetBundle in the GitHub project for the previous recipe: `https://github.com/dr-matt-smith/unity-cookbook-2020-ch08-07-load-from-streaming-assets/blob/masterAssets/StreamingAssets/cubered?raw=true`

Again, at the time of writing, I'm running a website where I have hosted the AssetBundle files, and if that site is still running when you try this recipe, then this URI may also work: `https://chess.frb.io/AssetBundles/cubered`

If the above is not working, then you will need to host the AssetBundle yourself for this recipe. If you have your own webspace (for example, `https://mydomain.com`) with a valid HTTPS certificate, then I suggest you create an `AssetBundles` folder containing the files for a `cubered` AssetBundle. Hence, the URI to use will be `https://mydomain.com/AssetBundles/cubered`.

If you don't have your own hosting, and the preceding links aren't working, refer to the *There's more...* section at the end of this recipe for the steps to use a local web server on your own computer to try out this recipe.

How to do it...

To load an AssetBundle from a web server at runtime, follow these steps:

1. Update the code in the C# script class named `LoadFromAssetBundle` to contain the following:

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

public class LoadFromAssetBundle : MonoBehaviour
{
    public String prefabName = "CubeRed";
    public String uri = "https://chess.frb.io/AssetBundles/cubered";

    void Start()
    {
         StartCoroutine(nameof(DownloadAndCreatePrefab));
    }

    IEnumerator DownloadAndCreatePrefab()
    {
        UnityEngine.Networking.UnityWebRequest request =
            UnityEngine.Networking.UnityWebRequestAssetBundle.
                GetAssetBundle(uri,0);
        yield return request.SendWebRequest();
        AssetBundle myLoadedAssetBundle =
            DownloadHandlerAssetBundle.GetContent(request);
        // create and instantiate prefab - red Cube
        GameObject cubePrefab =
            myLoadedAssetBundle.LoadAsset<GameObject>(prefabName);
        Instantiate(cubePrefab);
    }
}
```

2. Run the Scene and you should see a red cube appear – a GameObject created by instantiating the `cubered` Prefab, loaded from the AssetBundle downloaded from the web server:

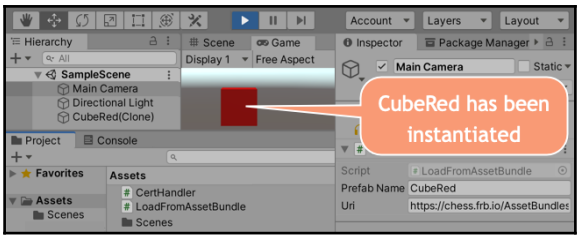

Figure 1.10 – The instantiated red cube, loaded from the AssetBundle from a web server

How it works...

When the Scene starts, the `Start()` method of the C# script class, `LoadFromAssetBundle`, is invoked. This method starts the `DownloadAndCreatePrefab()` co-routine. The coroutine method creates and sends a `UnityWebRequest` object using the uri public string variable. Once a response is received, the AssetBundle is extracted from the `Response` body, and a Prefab created using the `prefabName` public string variable. Finally, a GameObject in the Scene is instantiated from the Prefab.

There's more...

Here are some ways to go further with this recipe.

Working with a local web server running running HTTPS

Refer to the *Technical requirements* section of *Chapter 8, Web Server Communication and Online Version Control* at the beginning for links of how to install PHP or a web server application on your computer. You could use the built-in PHP web server, or a web server such as Apache or Nginx.

In the public folder for your web server, create an `AssetBundles` folder containing the files for the **cubered** AssetBundle. You can use either `127.0.0.1` or `localhost` to refer to a web server running on your local computer, and web servers usually run on port `8000` or `80`. So, the URI for the **cubered** AssetBundle will be `https://localhost:8000/AssetBundles/cubered`.

Solving local serve HTTP security certificate issues

If you cannot run an HTTPS web server, or there is no valid certificate for your computer, then, in order for UnityWebRequest to work, you'll have to fool the game into being happy with the certificate for your computer. You can do this by performing the following steps:

1. Create a C# script class named `CertHandler` containing the following code:

```
/// source: Unity Threads: post by Anagr Oct 2016
///https://forum.unity.com/threads/uniy-2018-2-https-webreques
t-failes-onlatest-hololens-os-udpate.550429/

using UnityEngine.Networking;
public class CertHandler : CertificateHandler
{
    protected override bool ValidateCertificate(byte[]
certificateData)
    {
       return true;
    }
}
```

NOTE: The `CertHandler` class always returns true, which bypasses any checking of a security certificate. So this should only be used when testing a game on a local machine, and never for a game downloading content from an internet server.

2. Update the `DownloadAndCreatePrefab()` method of the `LoadFromAssetBundle` script class as shown below. We need to add an extra statement after creating the `UnityWebRequest` object, to set an object instance of `CertHandler` as the certificate handler for the Unity Request object:

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

public class LoadFromAssetBundle : MonoBehaviour
{
    public String prefabName = "CubeRed";
    public String uri =
        "https://chess.frb.io/AssetBundles/cubered";
    void Start()
```

```
    {
        StartCoroutine(nameof(DownloadAndCreatePrefab));
    }
    IEnumerator DownloadAndCreatePrefab()
    {
        UnityEngine.Networking.UnityWebRequest request =
            UnityEngine.Networking.UnityWebRequestAssetBundle.
                GetAssetBundle(uri, 0);
        // cope with security issue for local host
        // <<<<< NEW CODE HERE <<<<<<<<<<<<<<
        request.certificateHandler = new CertHandler();
        yield return request.SendWebRequest();
        AssetBundle myLoadedAssetBundle =
            DownloadHandlerAssetBundle.GetContent(request);
        // create and instantiate prefab - red Cube
        GameObject cubePrefab =
myLoadedAssetBundle.LoadAsset<GameObject>(prefabName);
        Instantiate(cubePrefab);
    }
}
```

An easy way of checking that AssetBundle files are successfully being published by a web server is to request the `AssetBundles.manifest` text file from the URI. This lists the contents of the AssetBundle. If you can see this text in the browser, it's a good indication that your game will be able to download the AssetBundle from the URI. This screenshot shows the manifest (bundle contents summary text) being successfully retrieved from a local web server at localhost:8000:

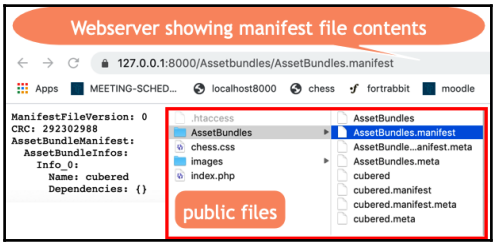

Figure 1.11 – Successful web browser download of a manifest file content summary

Addressables – Changing assests in memory at runtime

Assets compiled into a game build take up as little space as possible. However, another way to use up memory when a game plays on a mobile device is to load several different assets at once and use code to switch which is played in the Scene.

Poorly written code means even though only one asset is being used (for example, one song playing or one background texture being displayed), all have been loaded into memory, thereby reducing the speed and frame rate performance of your game.

Unity Addressables allow assets to be swapped in memory at runtime, ensuring memory for only one asset is required at any time. Two good introductions to Unity Addressables are those published by Unity and The Game Guru at the following links:

- `https://learn.unity.com/project/getting-started-with-addressables`
- `https://thegamedev.guru/unity-addressables/tutorial-learn-the-basics/`

A good video tutorial about AssetBundles and addressables by Jason Weimann (Oct 2019)
can be found at Unity3D.college:

- `https://unity3d.college/2019/10/07/unity3d-addressables-for-beginnersnext-level-of-assetbundles/`

Adding a language interpreter to allow runtime logic downloads

In almost all cases, runtime assets cannot include C# code scripts, since these need to have been compiled into the game when building. However, one way to download new game logic at runtime is to build your game to include a language interpreter in your game. Interpreted languages are evaluated at runtime, and so can run logic downloaded after a game has been built. One of the most popular interpreted languages for game scripting is Lua, and a great Lua interpreter for Unity is by Moonsharp:
`https://assetstore.unity.com/packages/tools/moonsharp-33776`

Using a Line Renderer to create a spinning laser trap

Interesting visual effects can be achieved through a scripted approach by creating line renderers. Unity line renderers can draw lines in the scene using assigned materials and their colors and Alpha Transparency. Lines can be drawn for many different reasons in games, such as to simulate laser beams, floating arrows to give the player hints on the direction to follow, and so on.

In this recipe, we'll create the effect of a rotating 3D Cube emitting a laser beam for a short distance. We'll also learn how to create a scaled invisible Cube at the same place as the laser, whose Box Collider can be used for collisions, as if with the laser.

Finally, in the *There's more...* section, we'll learn how to add these effects to the scene from the previous recipe, to then create explosions when the player's character collides with the laser (invisible cube's) collider:

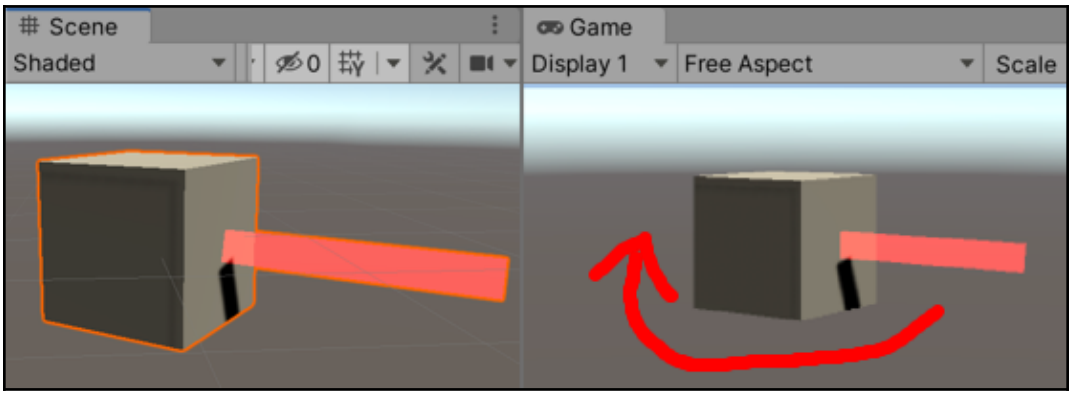

Figure 1.12 – The rotating Cube with the laser being emitted

How to do it...

To enhance the laser's aim with a Line Renderer, follow these steps:

1. Create a new **Unity 3D project**.
2. Our **Line Renderer** will need a **Material** to work with. Create a new **Material** named `m_laser` (menu: **Create** | **Material**).
3. With `m_laser` selected in the **Project** window, in the **Inspector** window, select the **Particles/Standard Unit** shader and set **Rendering Mode** to **Additive**.
4. Add a 3D **Cube** to the scene (menu: **Create** | **3D Object** | **Cube**) called `Cubelaser`.
5. Create a new C# script class called `LaserDisplay` containing the following and add an instance object as a component to the `Cube-laser` GameObject:

```
using UnityEngine;
```

```
public class LaserDisplay : MonoBehaviour
{
    public float lineWidth = 0.2f;
    public float lineLength = 2;
    public Color color = Color.white;
    public Material material;
    public float rotationSpeed = 0.1f;

    private LineRenderer _lineRenderer;

    void Awake()
    {
        _lineRenderer =
gameObject.AddComponent<LineRenderer>();
        _lineRenderer.material = material;
        _lineRenderer.positionCount = 2;
        _lineRenderer.startWidth = lineWidth;
    }
    void Update ()
    {
        _lineRenderer.material.SetColor("_Color", color);
        Vector3 forward =
            transform.TransformDirection(Vector3.forward);
        Vector3 lineStart = transform.position;
        Vector3 lineEnd = transform.position + forward *
lineLength;
        _lineRenderer.SetPosition(0, lineStart);
        _lineRenderer.SetPosition(1, lineEnd);
        transform.Rotate(0, rotationSpeed, 0);
    }
}
```

6. With the `Cube-laser` GameObject selected in the **Hierarchy** window, from the **Project** window, drag the `m_laser` material into the **Material** variable slot in the **Inspector** window for the **Laser Display (Script)** component:

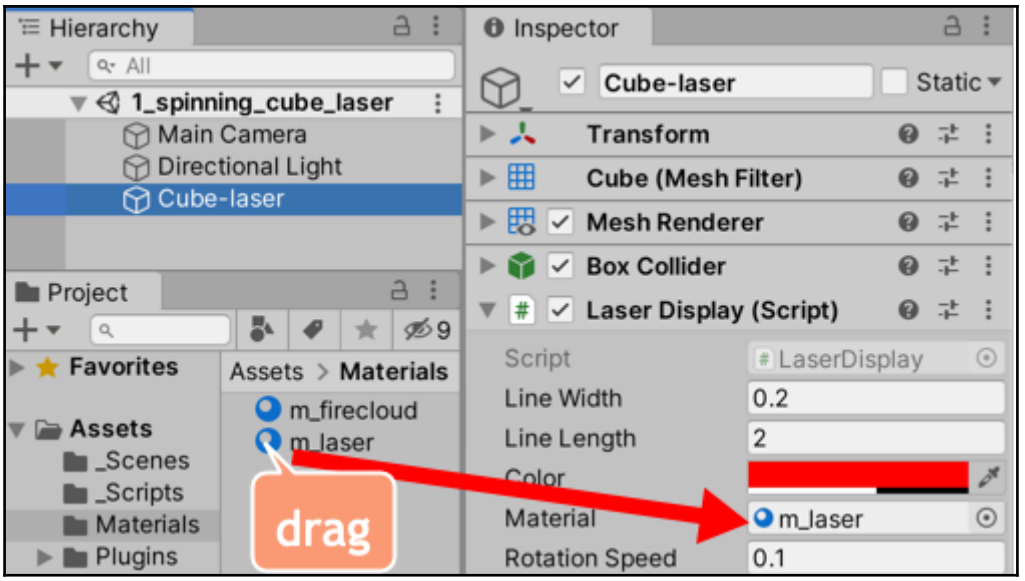

Figure 1.13 – Assigning the m_laser material for the LaserDisplay scripted component

7. In the **Inspector** window, use the **Color** selector popup to set a red color for the **Laser Display (Script)** component. Of course, you can play around with the color and its **Alpha Transparency** to get the effect you want for your game.
8. Run the scene; you should see a slowly spinning cube with a laser beam coming out of one of its sides.
   Let's also prepare this GameObject for collision usage by creating a collider that matches the space the beam is drawn in.

9. Create a 3D **Cube** named `Cube-collider`. Make this a child of the `Cubelaser` GameObject and set its **Position** to `(0, 0, 1)` and its **Scale** to `(0.1, 0.1, 2)`. Disable its **Mesh Renderer** (so that the player won't see it) and check the **Box Collider Is Trigger** option so that collision events will be generated:

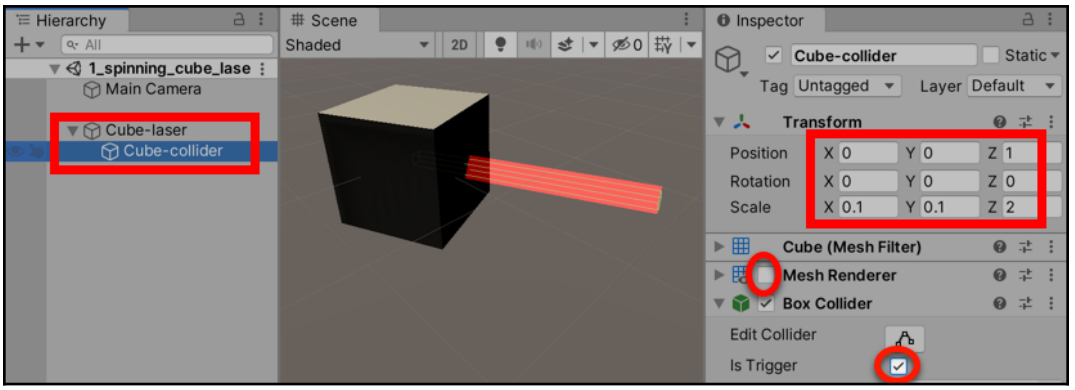

Figure 1.14 – Creating Cube-collider so that it matches the laser space

10. Run the scene again – you won't see any difference, but we've now created an object with a collider that matches the laser beam so that `OnTriggerEnter(...)` collision event messages will be generated when the GameObjects, such as a player character, collide with the beam.

How it works...

The script we wrote draws with a **Line Renderer** at runtime. This draws lines in the scene that the player sees, but the lines don't interact with other GameObjects. Our **Line Renderer** needed a **Material**, which we created and assigned, and can be customized through the public variables of the scripted component in terms of **Line width**, **Length**, and Color. Different colors can be set at design time or through code, with different transparency settings being used to achieve different looks for the lines that are drawn.

The script contains a statement to rotate the GameObject. The speed of rotation can also be adjusted or set to zero for no rotation.

If the amount of degree of rotation is quite high (above 10 degrees, for example) there is a chance that a collision might not occur because the player's character might be in the "gap" as the laser jumps from rotated position to rotated position. This could be solved with a slightly large collider, beyond the visible laser area.

Finally, we created an invisible **Box Collider** that took up the same space as the drawn line so that `OnTriggerEnter(...)` collision event messages will be

generated when the GameObjects, such as a player character, collide with the beam. There are two values of this `Cube-collider` GameObject that are important, to ensure the collider matches the laser being drawn. The position of the `Cube-collider` child GameObject should be `(0, 0, laserLength/2)`. This ensures that the collider starts in the center of `Cube-laser`, just like the line renderer. So, with a laser length of 2, the position of `Cube-collider` was `(0, 0, 1)`. The second important value is the scaler of `Cube-collider` – the Z-value should match the laser beam's length; that is, `(0.1, 0, 1, laserLength)`. So, if we wish to change the laser beam's length, then these values need to change too.

There's more...

Here are some ways to enhance this recipe.

Triggering an explosion when the player's character collides with the beam

With some scripting and an explosion particle system, we can trigger an explosion when the player's character collides with our laster-style line renderer:

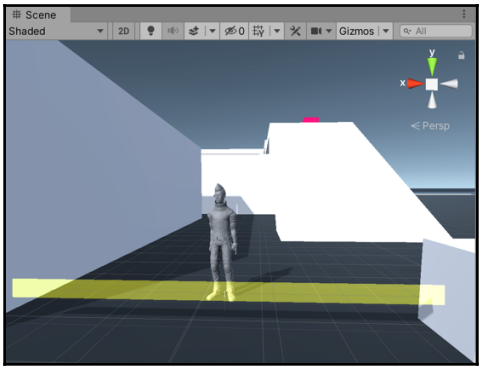

Figure 1.15 – Triggering an explosion when a character collides with the laser

Do the following:

1. Create a copy of the project from the *Simulating an explosion* recipe.
2. Delete the `crystal` GameObject from the scene.
3. Create a rotating 3D **Cube** with a laser-style **Line Renderer**, as described in this recipe.

4. With the `Cube-laser` GameObject selected in the **Hierarchy** window, set the **Rotation Speed** value to `0` for the **Laser Display (Script)** component in the **Inspector** window.
5. Create a new C# script class called `LaserCollision` and add an instance object as a component to the `Cube-collider` GameObject (the child object inside `Cube-laser`):

```
using UnityEngine;

public class LaserCollision : MonoBehaviour
{
    public GameObject explosionPrefab;

    private void OnTriggerEnter(Collider other)
    {
        // create explosion at same location as this Crystal
        GameObject explosion = Instantiate(explosionPrefab,
            other.transform.position,
other.transform.rotation);

        // destroy particle system after 1 second
        Destroy(explosion, 1);

        // do other logic here - e.g. reduce player health
etc. ...
    }
}
```

6. With the `Cube-collider` GameObject selected in the **Hierarchy** window, from the **Project** window's `Prefabs` folder, drag the **Particle System-explosion** prefab into the **Explosion Prefab** variable slot into the **Inspector** window for the **Laser Collision (Script)** component:

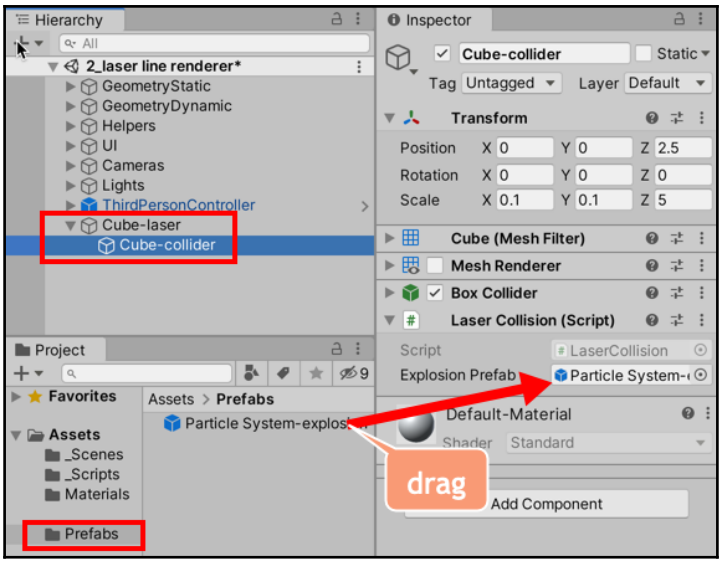

Figure 1.16 – Assigning the explosion prefab to the LaserCollision scripted component

7. Run the scene. Each time the player's character hits the laser, an explosion will be created.

In the preceding script, an explosion is instantiated at the location of the object that collided with the laser GameObject (`other.transform.position`).

You can add logic either to the `LaserCollision` script or the script for the third-person controller to reduce the score, reduce the player's health, or whatever else is appropriate in your game for when the player hits a laser.

Creating and applying a cookie texture to a spotlight

Cookie textures can work well with Unity spotlights to simulate shadows coming from projectors, windows, and so on. An example of this would be for the bars of a prison window. In this recipe, we'll create and apply a cookie Texture that can be used with Unity lights:

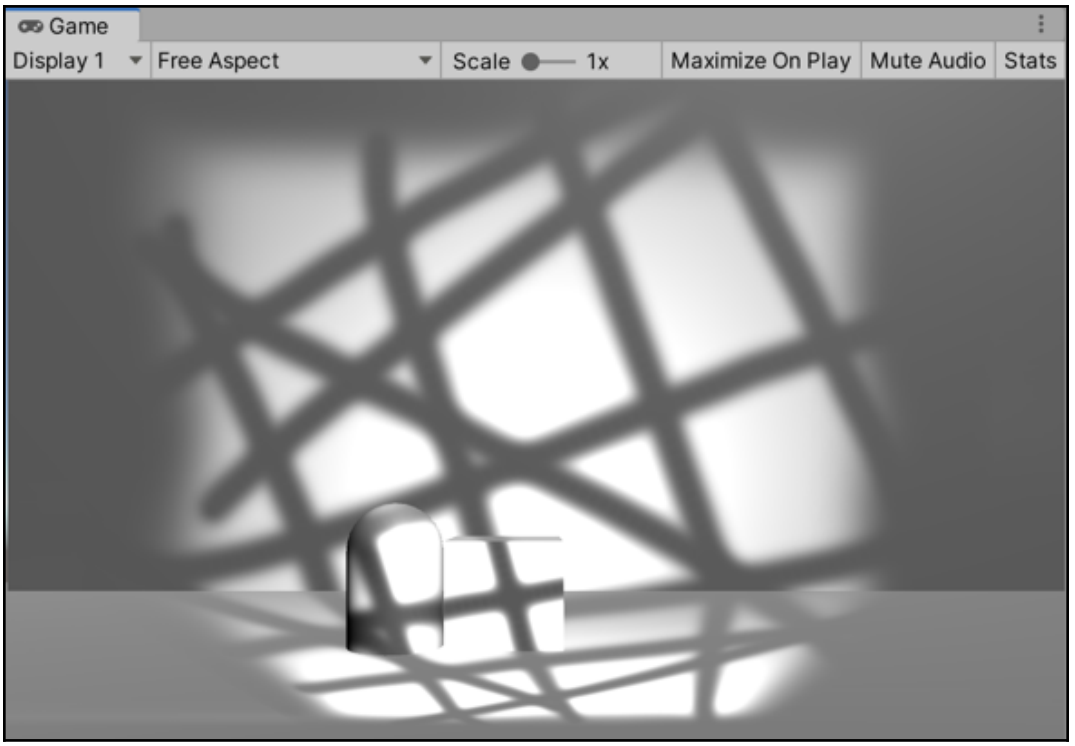

Figure 1.17 – The shadows in the scene from the cookie texture's spotlight

Getting ready

If you don't have access to an image editor or you would prefer to skip the details regarding the Texture map to focus on the implementation, we have provided the prepared cookie image file, called `spotCookie.tif`, inside the `14_07` folder.

How to do it...

To create and apply a cookie texture to a spotlight, follow these steps:

1. In your image editor, create a new `512 x 512` grayscale pixel image.
2. Ensure that the image's **border** is completely black by setting the brush tool's color to black and drawing around the four edges of the image. Then, draw some crisscrossed lines. Save your image, naming it `spotCookie`:

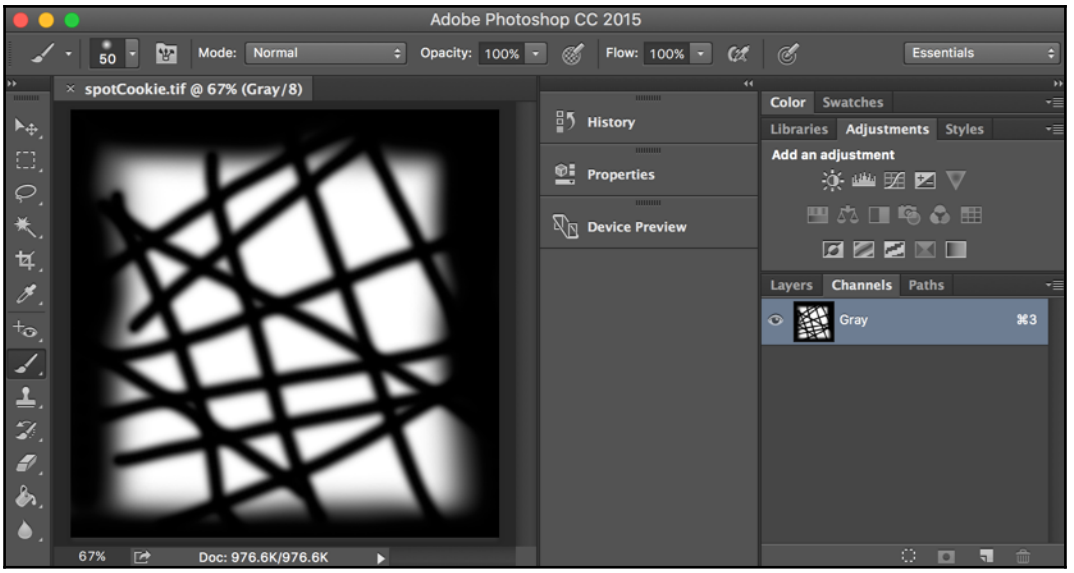

Figure 1.18 – Creating our cookie texture in an image editor

3. Create a new 3D project and import your cookie image.
4. With your cookie image asset files selected in the **Project** window, in the Inspector window, change its **Texture Type** to **Cookie**, its **Light Type** to **Spotlight**, and set its **Alpha Source** to **From Gray Scale**. Then, click on **Apply**:

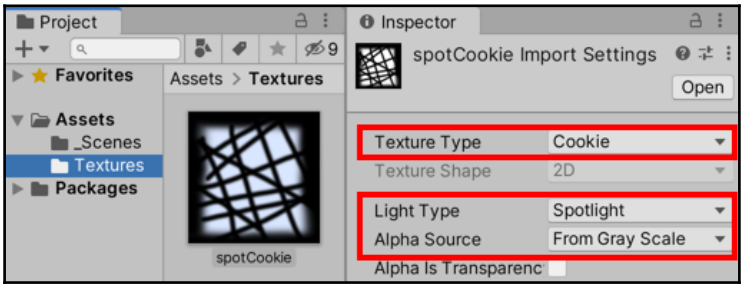

Figure 1.19 – Texture asset settings for use as a cookie

Now, let's create a floor and wall from scaled 3D cubes as a location where we can view our light and shadows.

5. Create a 3D **Cube** GameObject named `Cube-wall`. In the **Inspector** window, set its **Position** to `(1, 0, 3)` and its **Scale** to `(20, 20, 1)`.

6. Create a second 3D **Cube** GameObject named `Cube-floor`. In the **Inspector** window, set its **Position** to `(0, -1, 0)` and its **Scale** to `(20, 1, 20)`.
7. Now, add a **Spotlight** to the scene by going to **Create** | **Light** | **Spotlight**.
8. Orient the spotlight so that it's pointing in the direction of `Cube-wall` – you'll probably have to reset its **Rotation** to `(0, 0, 0)`.

Spotlights are usually created with an X-rotation of 90 degrees so that theyare facing downward.

9. Increase the **Range**, **Spot Angle**, and **Intensity** properties in the **Inspector** window until you see a bright circle of light from the spotlight shining onto the capsule and cube and the floor and wall.
10. In the **Inspector** window, ensure the spotlight's **Shadow Type** property is **No Shadows**. Then, drag your `spotCookie` Texture asset file from the **Project** window into the **Cookie** slot in the **Inspector** window:

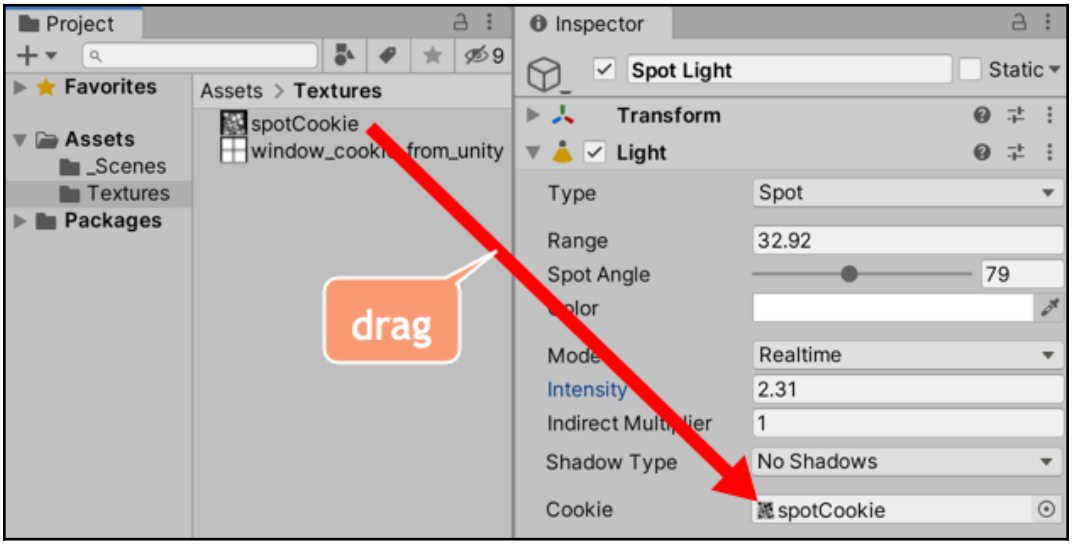

Figure 1.20 – Assigning the spotCookie texture image to Spot Light

11. Make the scene more interesting by adding one or two 3D objects between the spotlight and the wall.
12. Play the scene. You should see shadows cast on the objects, floor, and wall

as if there are sticks or branches in front of the spotlight.

13. You may wish to reduce the **Intensity** property of **Directional Light** in the scene (or disable it altogether for a nighttime effect).

How it works...

In this recipe, we created a grayscale **Texture** to be used with Unity spotlights that's completely black around the edges so that light does not "bleed" around the edge of our **Spotlight** emission. The black lines in the texture are used by Unity to create shadows in the light emitted from the spotlight, creating the effect that there are some straight beams of wood or metal that our **Spotlight** is being shone through.

The important the prepared cookie image file `spotCookie.tif` needed to be configured so that it could be used as a cookie. Once a **Texture** asset file has been configured, it can be applied to a spotlight to create dynamic shadow effects, adding depth and visual interest to the scene.

You can learn more about creating spotlight cookies in Unity by going to the Unity tutorial page: `http://docs.unity3d.com/Manual/HOWTO-LightCookie.html`.

There's more...

Cookies can be applied to other kinds of light. In the following screenshot, the window frame image from the Unity cookie manual page (`https://docs.unity3d.com/Manual/Cookies.html`) was applied to the **Directional Light** GameObject in our scene (and **Spot Light** was removed):

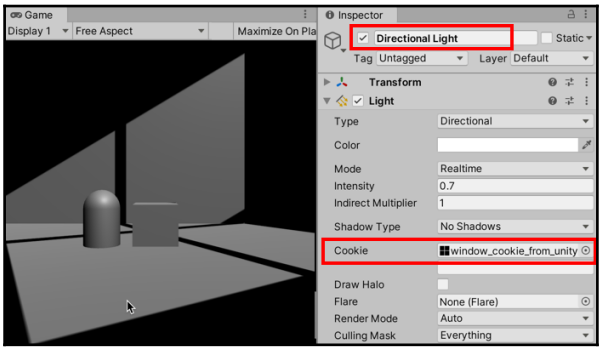

Figure 1.21 – Cookie texture applied to Directional Light

Baking light from GameObjects with an emissive material

Emissive materials can help add visual effects to a scene. In this recipe, as shown in the following screenshot, we will use an emissive material to bake a light from a glowing lamp onto local areas in a scene:

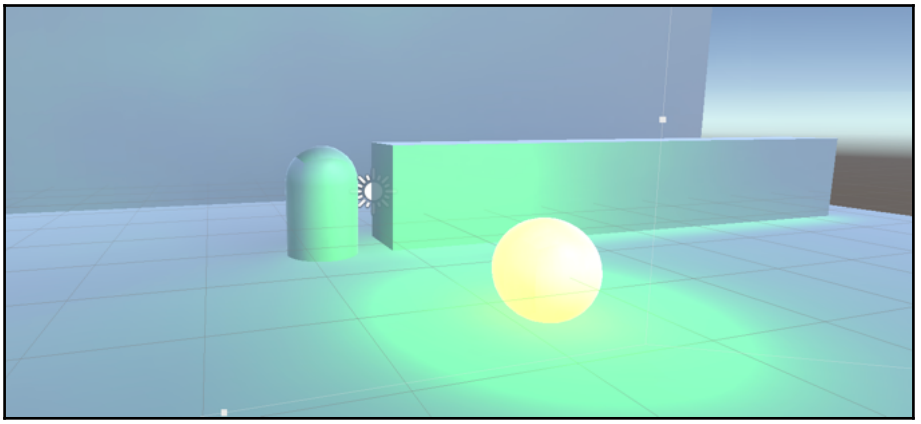

Figure 1.22 –Baking light from a GameObject's emissive material into a scene

Getting ready

This recipe follows on from the previous one, so make a copy of that and work on the copy.

How to do it...

To create a glowing GameObject using Material emission and then bake its light into the scene, follow these steps:

1. Begin working on the copy of the project for the previous recipe.
2. Since we'll be creating an emissive **Sphere** as the light source in this scene, let's delete the **Spot Light** GameObject from the scene.
3. Also for the **Directional Light** set the **Cookie Texture** to **None**.
4. Add a 3D **Sphere** to the scene (menu: **Create | 3D Object | Sphere**), in front of **Cube** and **Cylinder**.
5. Also, scale and move the cube in the X-axis so that it is much wider – this

will give us a good surface to see the diminishing light being emitted from the sphere. Set the **X-Position** to `3` and the **X-scale** to `7`.

6. Baked lighting only works for Static objects, so, with the exception of **Main Camera**, select everything in the **Hierarchy** window and check the **Static** option at the top right of the **Inspector** window:

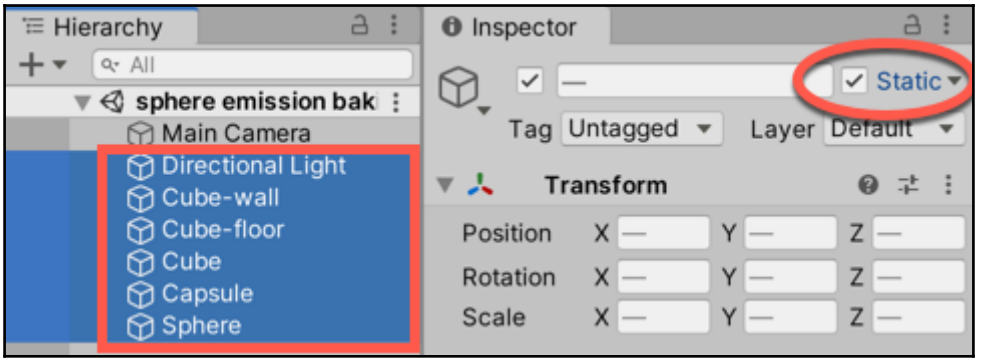

Figure 1.23 – Click the Static box for all items in the Hierarchy window except Main Camera

7. Select **Directional Light** in the **Hierarchy** window, reduce its **Intensity** to `0.5` (to emphasize our glowing **Sphere**), and change its light **Mode** drop-down menu property to **Baked**:

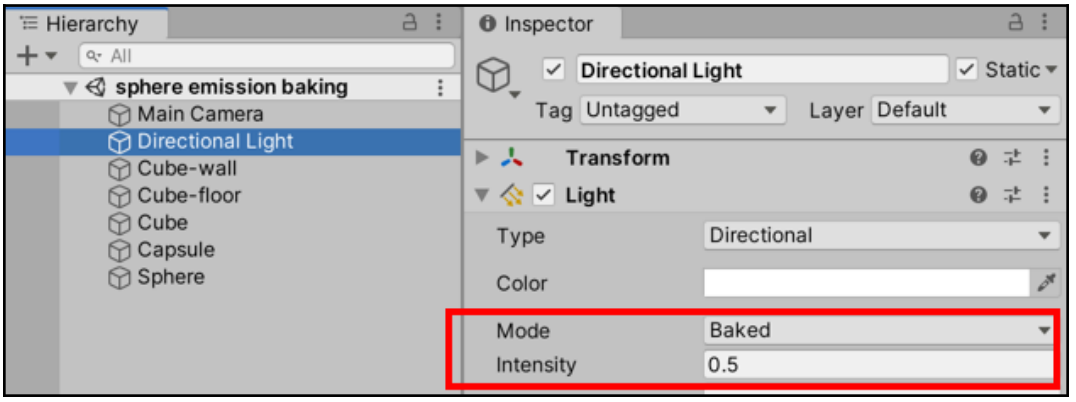

Figure 1.24– Adjusting the light Mode of Directional Light to Baked

8. Let's create a **Material** the emits green light for this **Sphere**. In the **Project** window, create a new **Material** named `m_greenEmission` (menu: **Create | Material**).

9. Drag `m_greenEmission` from the **Project** window onto the **Sphere** GameObject in the **Hierarchy** window.
10. With `m_greenEmission` selected in the **Project** window, in the **Inspector** window, check the **Emission** option. New **Color** and **Global Illuminator** properties should now appear immediately below the checked Emission box.
11. Set the **Global Illuminator** property to **Baked** so that Unity will pre-calculate light emissions from GameObjects using this **Material**.
12. Click the **HDR** color box, choose a bright green color, and increase the intensity of this light-emitting **Material** to **1** or **2** (this is a value you may wish to play with and "tweak" to get your desired settings for a scene). You can change **Intensity** in three ways: type in a number, use the slider, or click the **-2/-1/0/+1/+2** buttons:

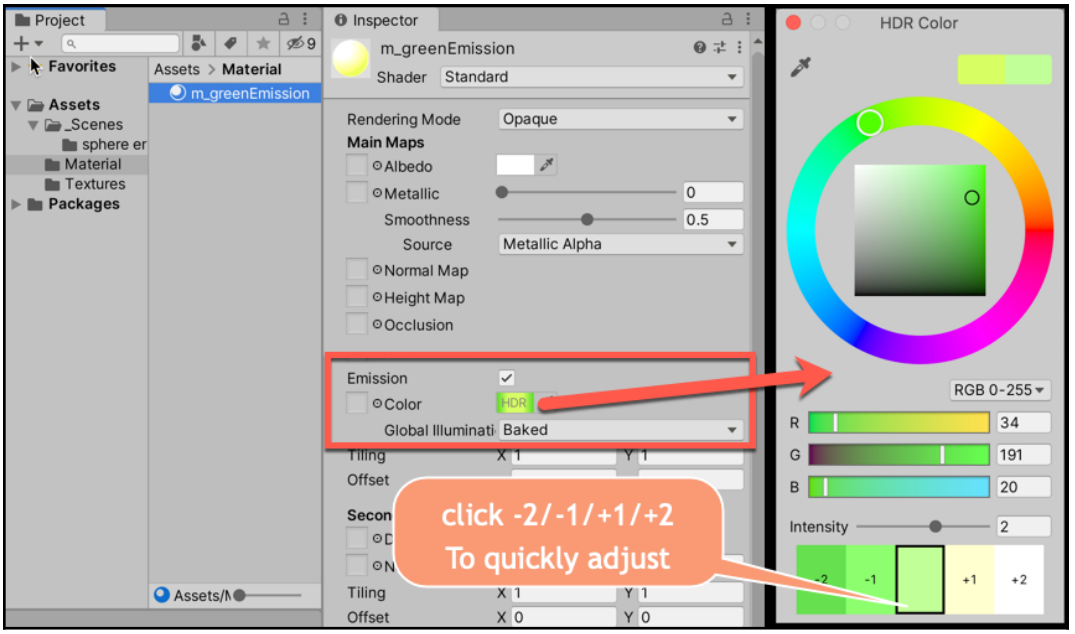

Figure 1.25 – Light intensity settings for the material

13. Display the **Lighting** window by going to **Window | Rendering | Lighting** or by clicking the light bulb icon at the bottom right of the Unity Editor. Click the **Scene** button and for the **Mixed Lighting** section, check the **Baked Global Illumination** option. Below **Workflow Settings**, uncheck **Auto Generate** and click the **Generate Lighting** button to "bake" the

Ambient light and green lamp emission light into the scene. You should click the **Generate Lighting** button once you have made changes and each time you wish Unity to recalculate the scene's lighting:

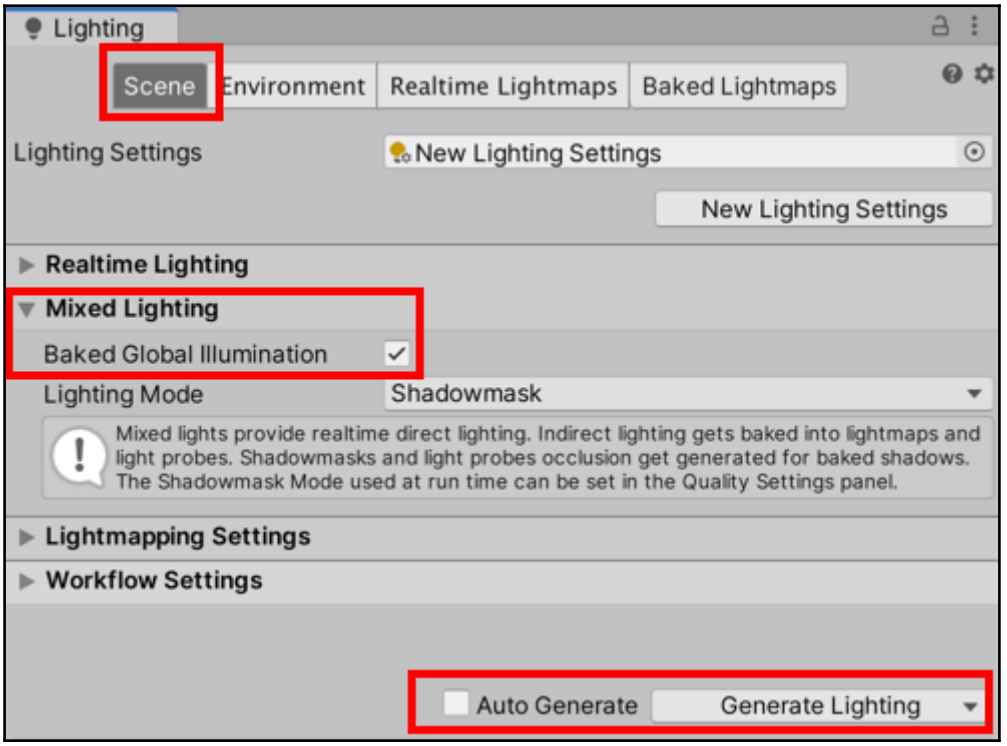

Figure 1.26 – Using the Lighting window to bake a LightMap for the scene

14. For a few seconds (depending on the speed of your computer and the complexity of the scene), you'll see a progress bar of the **Lightmap** baking process at the bottom right of the Unity Editor application window. When baking is completed, you'll see the new environment lighting for the **Scene** and **Game** windows:

Figure 1.27 – LightMap baking progress bar

15. Run your scene. You should see how the scene objects are lit both by

the directional light and by the green texture being emitted from the sphere.

16. Play with the settings of the scene light sources; for example, change the rotation of **Directional Light** and try setting its **Light Intensity** and **Indirect Multiplier** to `0.5`. You can also adjust the **HDR** intensity of the `m_greenEmission` and rebake **LightMap** to make the **Sphere** emission more emphasized (and **Directional Light** play a lesser role).

How it works...

By making all the scene GameObjects (apart from **Main Camera**) **Static**, you have enabled them to have their lighting pre-calculated by the Unity Global Illumination system.

The material you created was made to be **Emissive** so that GameObjects using that material (such as the sphere) will become additional sources of light for their local areas of the scene.

From the **Lighting** window, you enabled baked **Global Illumination**, and by disabling **AutoGeneration**, you decided when to make Unity recalculate lightmaps for **Static** objects - rather than have Unity continually recalculate in the background before you've finished making changes.