HOUR 23

# Polish and Deploy

**What You'll Learn in This Hour:**

- How to manage scenes in a game
- How to save data and objects between scenes
- The different player settings
- How to deploy a game

In this hour, you'll learn all about polishing a game and deploying it. You'll start by learning how to move about different scenes. Then you'll explore ways to persist data and game objects between scenes. From there, you'll take a look at the Unity player and its settings. You'll then learn how to build and deploy a game.

## Managing Scenes

So far, everything you have done in Unity has been in the same scene. Although it is certainly possible to build large and complex games in this way, it is generally much easier to use multiple scenes. The idea behind a scene is that it is a self-contained collection of game objects. Therefore, when transitioning between scenes, all existing game objects are destroyed, and all new game objects are created. However, you can prevent this, as discussed in the next section.

NOTE

### What Is a Scene? Revisited

You learned about the basics of scenes early on in this book. It is time, however, to revisit that concept with the knowledge you now possess. Ideally, a scene is like a level in a game. With games that get consistently harder or games that have dynamically generated levels, though, this is not necessarily true. Therefore, it can be good to think of a scene as a common list of assets. A game consisting of many levels that use the same objects can actually consist of one scene. It is only when you need to get rid of a bunch of objects and load a bunch of new objects that the idea of a new scene really becomes necessary. Basically, you should not split levels into different scenes just because you can. Create new scenes only if required by the gameplay and for asset management.

---

NOTE

### Building?

This hour tosses around two key terms: *building* and *deploying*. Although they can often mean the same thing, they do have a slight difference. *Building* a project means telling Unity to turn your Unity project into a final, executable set of files. Therefore, building for the Windows OS will produce an .exe file with a data folder, building for Mac OS will build a .dmg file with all game data in it, and so on. *Deploying* means sending a built executable to a platform to be run. For example, when building for Android, an .apk file (the game) is built, and then it is deployed to an Android device for playing. These options are managed in Unity's build settings, which are covered shortly.

---

## Establishing Scene Order

Transitioning between scenes is relatively easy. It just requires a little setup. The first thing you do is add the scenes of your project to the project's build settings, as follows:

1. Open the build settings by selecting **File > Build Settings**.
2. With the Build Settings dialog open, click and drag any scenes you want in your final project into the Scenes in Build window (see Figure 23.1).

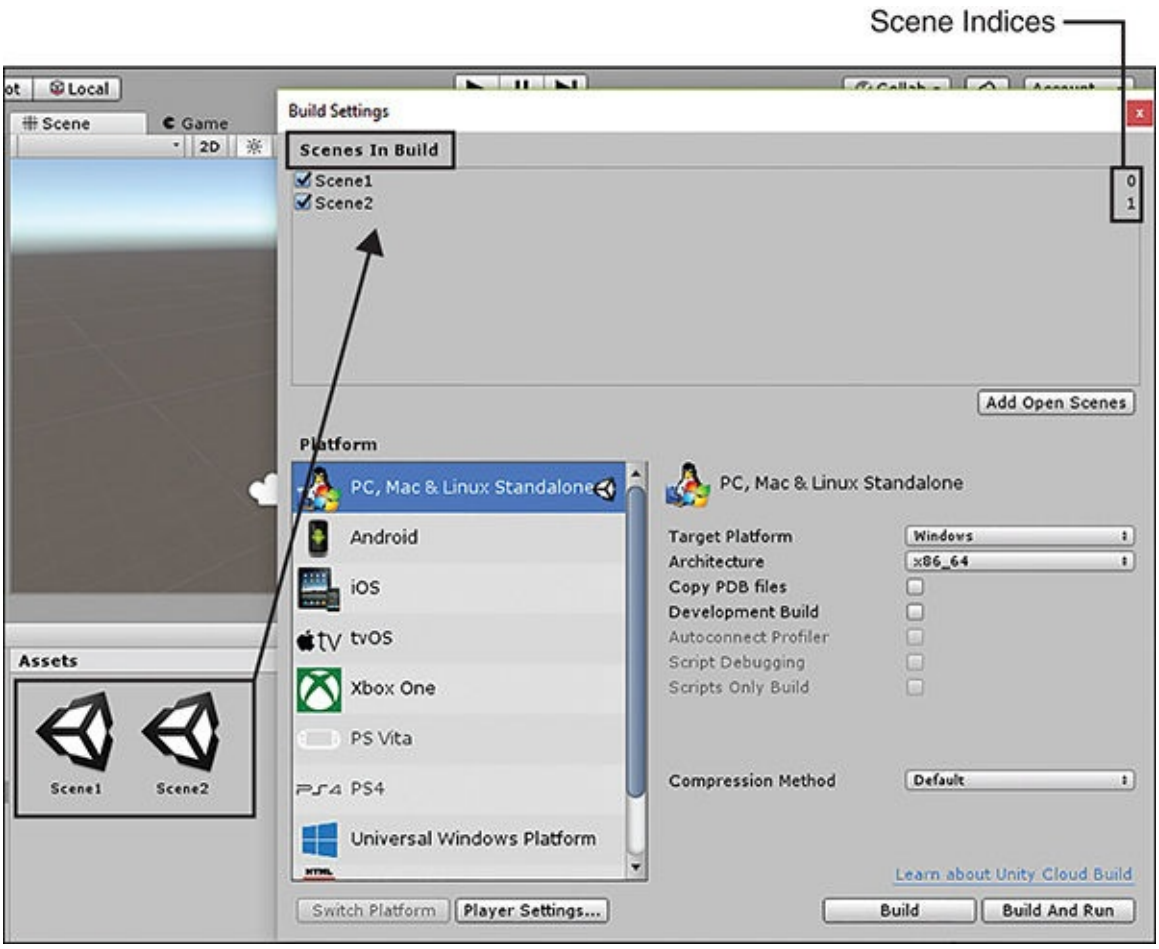


**FIGURE 23.1**
Adding scenes to the build settings.

**3.** Make note of the number that appears next to each scene in the Scenes in Build window. You will need these numbers later.

## ▼ TRY IT YOURSELF

### Adding Scenes to Build Settings

In this exercise, you'll add scenes to the build settings of a project. Be sure to save the project you make here because you will be using it again in the next section. Follow these steps:

**1.** Create a new project and add to it a new folder named Scenes.

**2.** Click **File > New Scene** to create a new scene and then select **File > Save Scene As** to save it. Save the scene in the Scenes folder as **Scene1**. Repeat this step to save another scene called **Scene2**.

3. Open the build settings (by selecting **File > Build Settings**). Drag **Scene1** into the Scenes in Build window first, and then drag **Scene2** into the same window. Ensure that Scene1 has an index of 0 and Scene2 has an index of 1. If not, reorder them by dragging them around.

# Switching Scenes

Now that the scene order is established, switching between them is easy. To change scenes, use the method `LoadScene()`, which is a part of the `SceneManager` class. In order to use this class, you need to tell Unity that you want to access it. You can do that by adding the following line to the top of any script that needs it:

**Click here to view code image**

```
using UnityEngine.SceneManagement;
```

The `LoadScene()` method takes a single parameter that is either an integer representing the scene's index or a string representing the scene's name. Therefore, to load a scene that has an index of 1 and the name Scene2, you could write either of these two lines:

**Click here to view code image**

```
SceneManager.LoadScene(1);        // Load by index
SceneManager.LoadScene("Scene2"); // Load by name
```

This method call immediately destroys all existing game objects and loads the next scene. Note that this command is immediate and irreversible, so make sure that it is what you want to do before calling it. (The LevelManager prefab from the Exercise at the end of Hour 14, "User Interfaces," uses this method.)

TIP

## Async Scene Loading

So far this book has focused on loading scenes immediately. That is, when you tell the `SceneManager` class to change a scene, the current scene is unloaded and then the next scene is loaded. This works well for small scenes, but if you are trying to load a very large scene, there can be a pause between levels during which the screen is black. To avoid this, one thing you could do is load a scene *asynchronously*. This "async" style of loading tries to load a

scene “behind the scenes” while the main game continues on. When the new scene is loaded, the scene is switched. This method isn’t instant, but it can help prevent gameplay hiccups. Generally speaking, loading scenes asynchronously can be a bit complex and is beyond the scope of this book.

---

## Persisting Data and Objects

Now that you have learned how to switch between scenes, you have undoubtedly noticed that data doesn’t transfer during a switch. In fact, so far all your scenes have been completely self-contained, with no need to save anything. In more complex games, however, saving data (often called *persisting* or *serializing*) becomes a real necessity. In this section, you’ll learn how to keep objects from scene to scene and how to save data to a file to access later.

## Keeping Objects

An easy way to save data in between scenes is just to keep the objects with the data alive. For example, if you have a player object that has scripts on it containing lives, inventory, score, and so on, the easiest way to ensure that this large amount of data makes it into the next scene is just to make sure that it doesn’t get destroyed. There is an easy way to accomplish this, and it involves using a method called `DontDestroyOnLoad()`. The method `DontDestroyOnLoad()` takes a single parameter: the game object that you want to save. Therefore, if you want to save a game object that is stored in a variable named `Brick`, you could write the following:

```
DontDestroyOnLoad (Brick);
```

Because the method takes a game object as a parameter, another great way for objects to use it is to call it on themselves using the `this` keyword. For an object to save itself, you put the following code in the `Start()` method of a script attached to it:

```
DontDestroyOnLoad (this);
```

Now when you switch scenes, your saved objects will be there waiting.

▼ TRY IT YOURSELF

### Persisting Objects

In this exercise, you'll save a cube from one scene to the next. For this exercise you need the project created in the Try It Yourself "Adding Scenes to Build Settings." If you have not completed the project yet, do so before continuing. Be sure to save this project because you will be using it again in the next section. Follow these steps:

1. Load the project created in the Try It Yourself "Adding Scenes to Build Settings." Load **Scene2** and add a sphere to the scene. Position the sphere at (0, 2, 0).
2. Load **Scene1** and a cube to the scene and position the cube at (0, 0, 0).
3. Create a Scripts folder and in it create a new script named **DontDestroy**. Attach the script to the cube.
4. Modify the code in the DontDestroy script so that it contains the following:

**Click here to view code image**

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class DontDestroy : MonoBehaviour
{
    void Start()
    {
        DontDestroyOnLoad(this);
    }

    // A neat trick for easily detecting mouse clicks on an object
    void OnMouseDown()
    {
        SceneManager.LoadScene(1);
    }
}
```

This code tells the cube to persist between scenes. In addition, it uses a little trick for detecting when the user clicks on the cube (`OnMouseDown()`; store this one away for a rainy day). When the user clicks on the cube, Scene2 is loaded.

5. Run the scene and notice that when you click the cube in the Game view, the scene transitions, and you see the sphere appear next to the cube (see Figure 23.2).

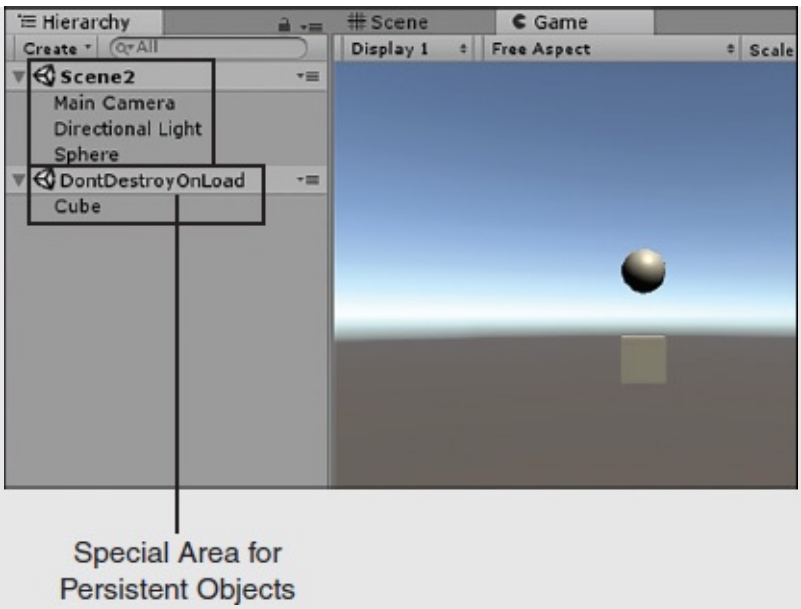


**FIGURE 23.2**
Cube persisting into Scene2.

CAUTION

### Dark Scenes

You may have noticed that when you switch scenes, the new scene can be darker even though there is a light in there. This is due to not having complete lighting data for dynamically loaded scenes. Luckily, the solution is simple: Just load the scene in Unity and select **Window > Lighting > Settings**. Uncheck **Auto Generate** at the bottom and then click **Generate Lighting**. This tells Unity to stop using temporary light calculations for the scene and to commit these calculations to an asset. A folder appears next to your scene, with the same name as the scene. Thereafter, the lighting will be fine when loading into the scene.

## Saving Data

Sometimes you need to save data to a file so you can access it and later. Some things you might need to save are the player's score, configuration preferences, or inventory. There are certainly many complex and feature-rich ways to save data, but a simple solution is something called *PlayerPrefs*. PlayerPrefs is an object that exists to save basic data to a file locally on your system. You can use

PlayerPrefs to pull the data back out.

Saving data to PlayerPrefs is as simple as supplying some name for the data and the data itself. The methods you use to save the data depend on the type of data. For instance, to save an integer, you call the `SetInt()` method. To get the integer, you call the `GetInt()` method. Therefore, the code to save a value of 10 to PlayerPrefs as the score and get the value back out would look like this:

**Click here to view code image**

```
 PlayerPrefs.SetInt ("score", 10);
PlayerPrefs.GetInt ("score");
```

Likewise, there are methods to save strings (`SetString()`) and floats (`SetFloat()`). Using these methods, you can easily persist any data you want to a file.

## TRY IT YOURSELF

### Using PlayerPrefs

In this exercise, you'll save data to the PlayerPrefs file. For this exercise you need the project created in the Try It Yourself "Persisting Objects." If you have not completed the project yet, do so before continuing. You will use the legacy GUI for this exercise. The focus is on PlayerPrefs, not the UI. Follow these steps:

1. Open the project you created in the Try It Yourself "Persisting Objects" and ensure that **Scene1** is loaded. Add a new script named **SaveData** to the Scripts folder and add the following code to the script:

**Click here to view code image**

```
public string playerName = "";

void OnGUI()
{
    playerName = GUI.TextField(new Rect(5, 120, 100, 30), playerNa

    if (GUI.Button(new Rect(5, 180, 50, 50), "Save"))
    {
        PlayerPrefs.SetString("name", playerName);
    }
}
```

2. Attach the script to the Main Camera object. Save Scene1 and load **Scene2**.

3. Create a new script called **LoadData** and attach it to the Main Camera object. Add the following code to the script:

**Click here to view code image**

```
string playerName = "";

void Start()
{
    playerName = PlayerPrefs.GetString("name");
}

void OnGUI()
{
    GUI.Label(new Rect(5, 220, 50, 30), playerName);
}
```

4. Save Scene2 and reload Scene1. Run the scene. Type your name into the text field and click the **Save** button. Now click the cube to load Scene2. (The scene switching cube was implemented in the previous Try It Yourself, "Persisting Objects.") Notice that the name you entered is written on the screen. In this Try It Yourself, the data was saved to PlayerPrefs and then reloaded from PlayerPrefs in a different scene.

CAUTION

## Data Safety

Although using PlayerPrefs to save game data is very easy, it is not very secure. The data is stored in an unencrypted file on the player's hard drive. Therefore, players could easily open the file and manipulate the data inside. This could enable them to gain an unfair advantage or break the game. Be aware that PlayerPrefs, just as the name indicates, is intended for saving player preferences. It just so happens that it is useful for other things. True data security is a difficult thing to achieve and is definitely beyond the scope of this book. Just be aware that PlayerPrefs will work for what you need it for at this early stage in your development of games, but in the future you will want to look into more complex and secure means of saving player data.

# Unity Player Settings

Unity provides several settings that affect how the game works after it is built. These settings are called the player settings, and they manage things like the game's icon and supported aspect ratios. There are many settings, and many of them are self-explanatory. If you select **Edit > Project Settings > Player**, the Player Settings window opens in the Inspector view. Take your time to look through these settings and read about them in the following sections so you can learn what they do.

## Cross-Platform Settings

The first player settings you see are the cross-platform settings (see Figure 23.3). These are the settings applied to the built game regardless of the platform (Windows, iOS, Android, Mac, and so on) you built it for. Most of the settings in this section are self-explanatory. The product name is the name that will appear as the title of your game. The icon should be any valid texture image file. Note that the dimensions of the icon have to be square powers of 2, such as 8×8, 16×16, 32×32, 64×64, and so on. If the icon doesn't match these dimensions, the scaling may not work properly, and the icon quality might be very low. You can also specify a custom cursor and define where the cursor hotspot (the spot on the cursor that is the actual "click" location) is.

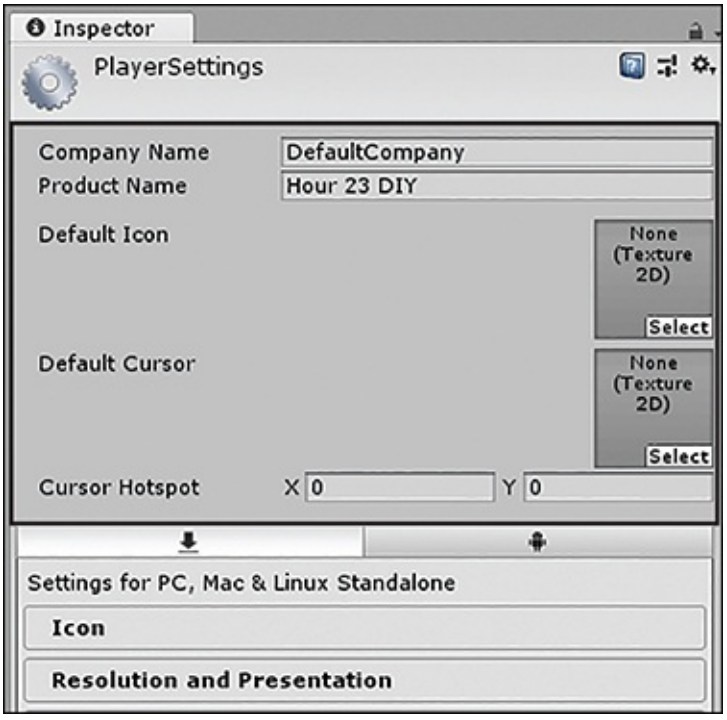


FIGURE 23.3

**FIGURE 23.3**
The cross-platform settings.

## Per-Platform Settings

The per-platform settings are specific to each platform. Even though there are several repeat settings in this section, you still have to set up each one of them for every platform you want to build your game for. You can select a specific platform by choosing its icon from the selection bar (see Figure 23.4). Note that you see icons only for the platforms you currently have installed for Unity. As you can see in Figure 23.4, only the Standalone (PC, Mac, and Linux) and Android platforms are currently installed on this machine.

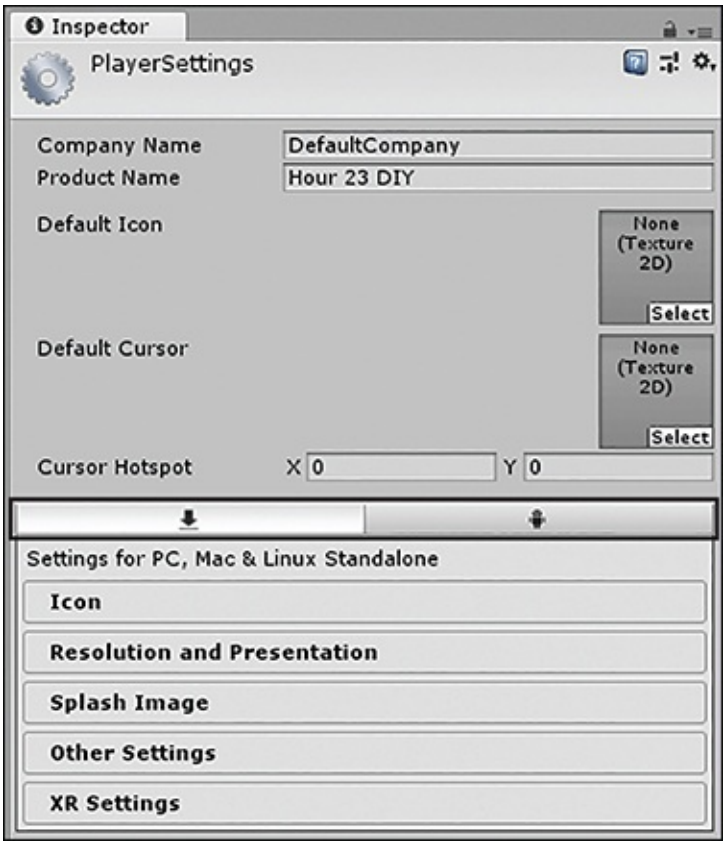


**FIGURE 23.4**
The platform selection bar.

Many of these settings require a more specific understanding of the platform for which you are building. You should not modify these settings until you better understand how a particular platform works. Other settings are rather straightforward and need to be modified only if you are trying to achieve a specific goal. For instance, the Resolution and Presentation settings deal with the

specific goal. For instance, the Resolution and Presentation settings deal with the dimensions of the game window. For desktop builds, these can be windowed or full screen, with a large array of different supported aspect ratios. By enabling or disabling the different aspect ratios, you allow or disallow different resolutions that the player can choose when playing the game.

The icon settings are auto-populated for you if you specify an icon image for the Default Icon property in the Cross-Platform Settings section. You can see that various sizes of the icon image will be generated, based on a single provided image. This is why it is important for the provided image to have the correct dimensions. You can also provide a splash image for your game in the Splash Image section. A *splash image* is an image that is added to the Player Settings dialog when a player first starts up the game.

NOTE

### Too Many Settings

You probably noticed that a large number of settings in the Player Settings dialog aren't covered in this section. Most of the properties are already set to default values so that you can just quickly build a game. The other settings exist to achieve advanced functionality or polish. You shouldn't toy with most of the settings if you don't understand what they do because changes can lead to strange behaviors or prevent your game from working at all. In short, use only the more basic settings until you get more comfortable with game-building concepts and the different features you have already used.

NOTE

### Too Many Players

The term *player* is used a lot in this hour because there are two ways in which the term can be applied. The first, obviously, is the player who actually plays your game. This is a person. The second way the term can be used is to describe the *Unity player*, the window that the game is played in (like a movie player or a TV). The player exists on the computer (or device). Therefore, when you hear *player*, it probably means a person, but when you hear *player settings*, it probably refers to the software that actually displays the game.

# Building Your Game

Let's say that you've finished building your first game. You've completed all the work and tested everything in the editor. You have even gone through the player settings and set everything up the way you want it. Good job! It is now time to build your game. You need to be aware of two settings windows used in this process. The first is the Build Settings window, which is where you determine the final results of the build process. The second is the Game Settings window, which involves settings that human players see and configure.

## Build Settings

The Build Settings window contains the terms under which the game is built. It is here that you specify the platform the game will be built under as well as the various scenes in the game. You have seen this dialog once before, but now you should take a closer look at it.

To open the Build Settings dialog, select **File > Build Settings**. In the Build Settings dialog, you can change and configure your game as you want. Figure 23.5 shows the Build Settings dialog and the various items on it.

As you can see, in the Platform section, you can specify a new platform to build for. If you choose a new platform, you need to click **Switch Platform** to make the switch. Clicking the **Player Settings** button opens the Player Settings dialog in the Inspector view. You have seen the Scenes in Build section before; this is where you determine which scenes will make it into the game and their order. You also have the various build settings for the specific platform that you chose. The PC, Mac & Linux Standalone settings are self-explanatory. The only thing to note here is the Development Build option, which allows the game to run with a debugger and performance profiler.

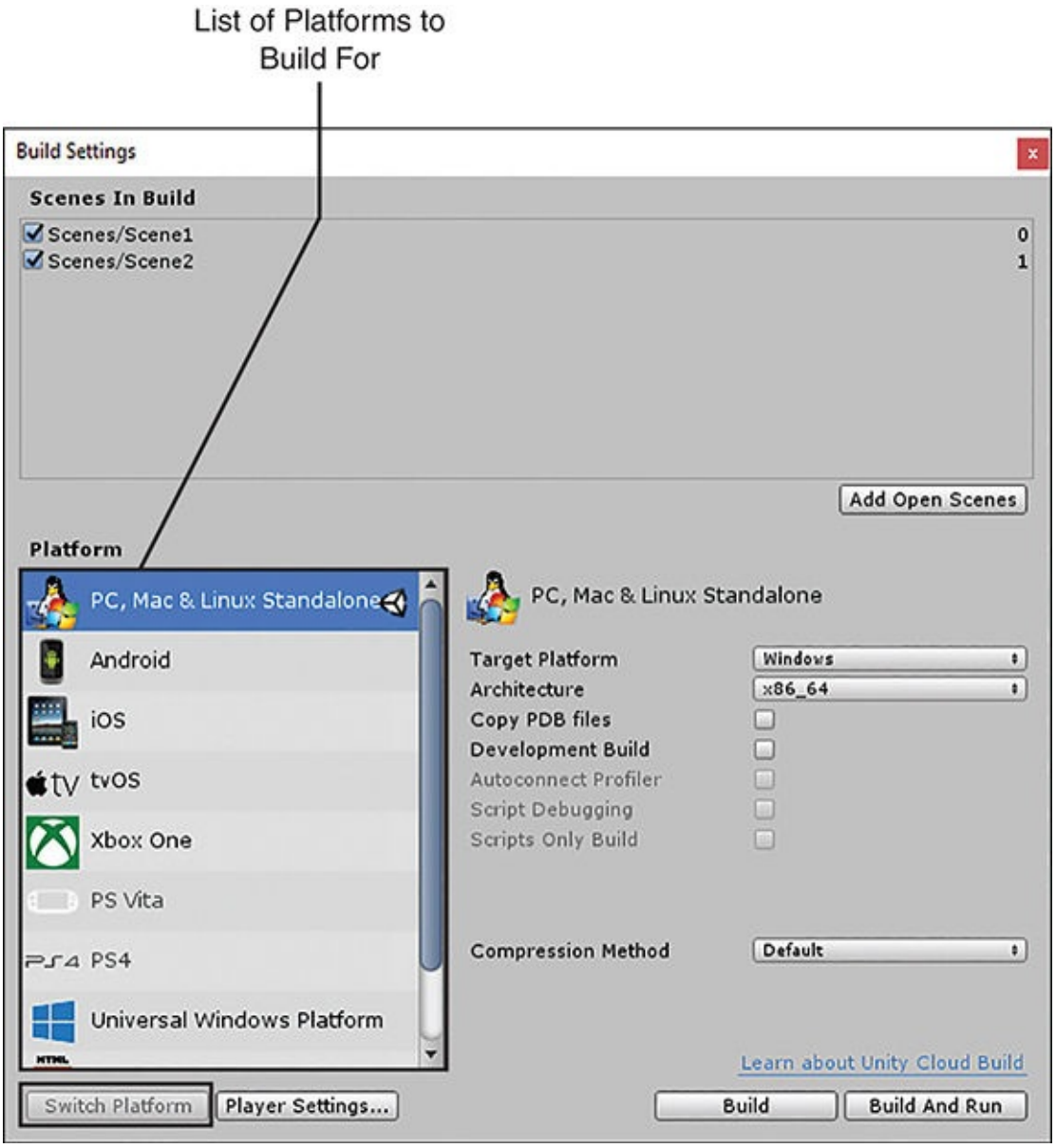


**FIGURE 23.5**
The Build Settings dialog.

When you are ready to build your game, you can either click **Build** to just build the game or **Build and Run** to build the game and then run it immediately. The file that Unity creates will depend on the platform chosen.

## Game Settings

When a built game is run from its actual file (not from within Unity), the player is presented with a Game Settings dialog (see Figure 23.6). From this dialog,

players choose options for their game experience.

The first thing you may notice is that the name of the game appears in the title bar of the window. Also, any splash image you provided in the Player Settings dialog appears at the top of this window. This first tab, Graphics, is where players specify the resolution at which they want to play the game. The list of available resolutions is determined by the aspect ratios you allowed or disallowed in the Player Settings dialog and the operating system. Players can also choose to run the game in a window or full screen and can pick their quality settings.

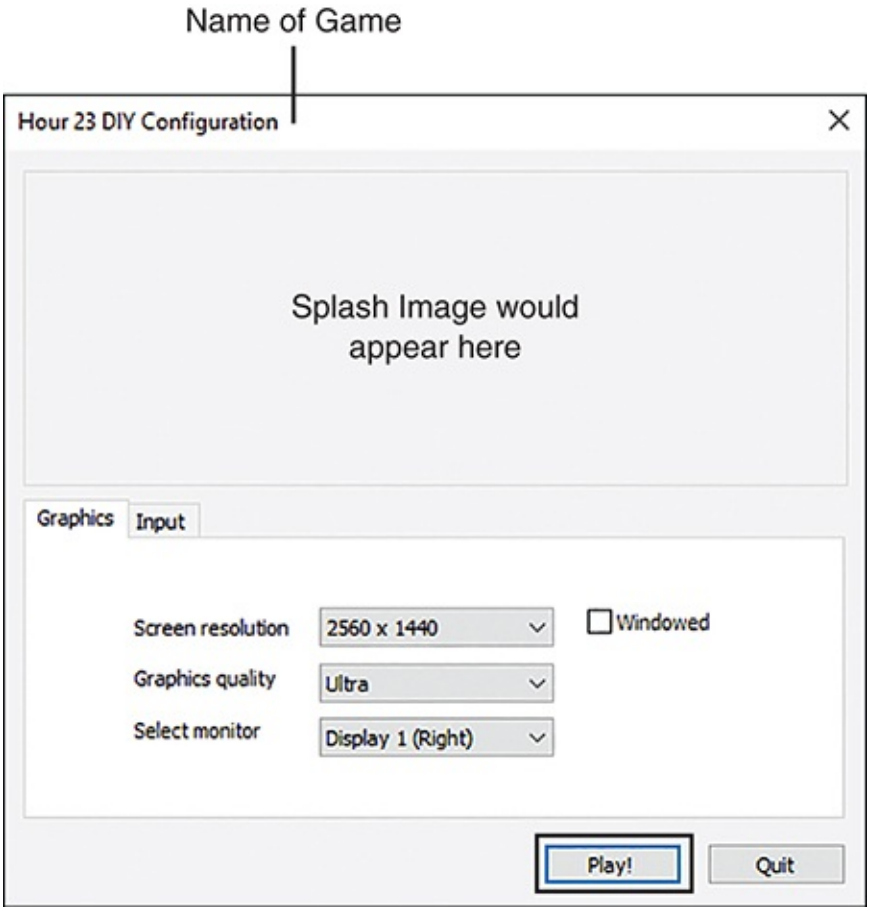


**FIGURE 23.6**
The Game Settings dialog.

Players can then switch over to the Input tab (see Figure 23.7), where they can remap any input axes to their desired keys and buttons.

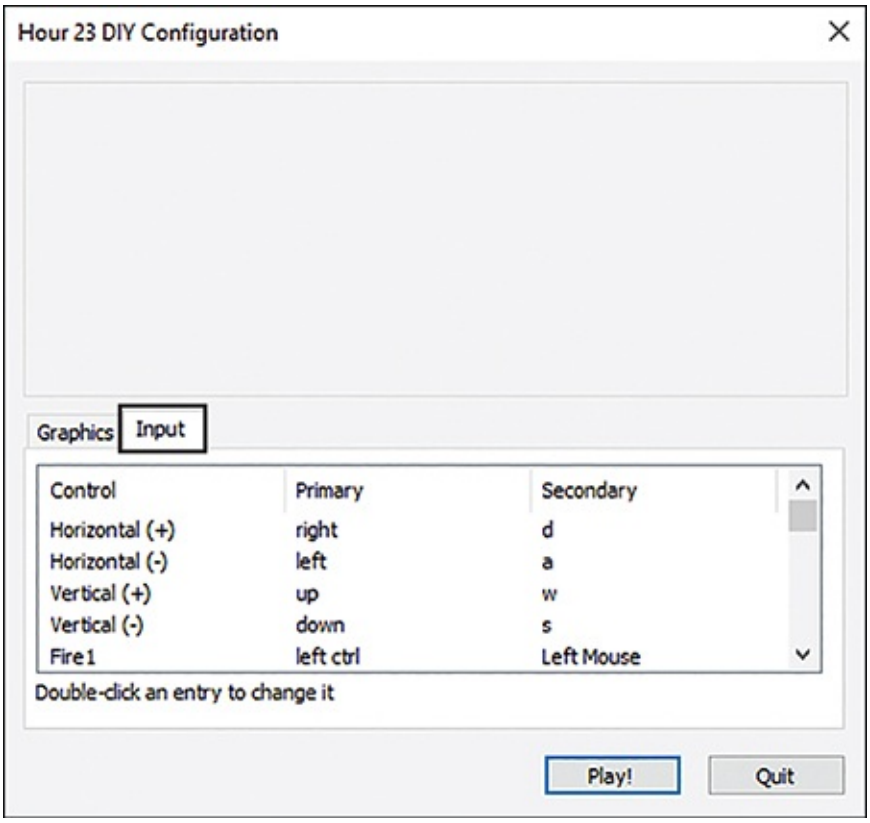


**FIGURE 23.7**
The input settings.

NOTE

## Told You So!

You might recall that earlier in this book I said you should always try to ensure that the input you are reading from a player is based on one of the input axes and not the specific keys. This is why. If you looked for specific keys instead of axes, the player would have no choice but to use the control scheme you intended. If you think this isn't a big deal, just remember that a lot of people out there (people with disabilities, for instance) use nonstandard input devices. If you deny them the ability to remap controls, they might not be able to play your games. Using axes instead of specific keys is a negligible amount of work on your part and can make the difference between players loving or hating your game.

After players choose the settings they want, they just click **Play!** and begin enjoying your game.

## Summary

In this hour, you've learned all about polishing and building games in Unity. You started by learning how to change scenes in Unity using the `SceneManager.LoadScene()` method. From there, you learned how to persist game objects and data. After that, you learned about the various player settings. Finally, you wrapped up the hour by learning to build your games.

## Q&A

**Q. A lot of these settings looked important. Why doesn't this hour cover them?**

**A.** Truth be told, most of those settings are unnecessary for you. The fact is that they aren't important…until they are important. Most of the settings are platform specific and are beyond the scope of this book. Instead of spending many pages going over settings you might never use, it is left up to you to learn about them if you ever need them.

## Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

### Quiz

1. How do you determine the indexes of the scenes in your game?
2. True or False: Data can be saved using the PlayerPrefs object.
3. What dimensions should an icon for your game have?
4. True or False: The input settings in the game settings allow the player to remap all inputs in your game.

### Answers

1. After you add the scenes to the Scenes in Build list, each one has an index assigned to it.
2. True
3. A game icon should be a square with sides that are powers of 2: 8×8, 16×16, 32×32, and so on.

4. False. The player can only remap inputs that were established based on input axes, not specific keypresses.

## Exercise

In this exercise, you'll build a game for your desktop operating system and experiment with the various features. There isn't much to this exercise, and you should spend most of your time trying out various settings and watching their impact. Because this is just an example to get you building your games, there isn't a completed project to look at in the book assets.

1. Open any project you have created previously or create a new project. (Don't worry, I'll wait.)
2. Go into the Player Settings dialog and configure your player however you want.
3. Go into the Build Settings dialog and ensure that you have added your scenes to the Scenes in Build list.
4. Ensure that the Platform is set to PC, Mac & Linux Standalone.
5. Build the game by clicking **Build**.
6. Locate the game file that you just built and run it. Experiment with the different game settings and see how they affect the gameplay.