

Chapter 28. Prototype 1: Apple Picker

Here it is. Today, you make your first digital game prototype.

Because this is your first prototype, it is rather simple. As you continue through the prototyping chapters, the projects get more complex and use more of the features of Unity.

By the end of this chapter, you will have a working prototype of a simple arcade game.

The Purpose of a Digital Prototype

Before we start making the prototype of Apple Picker, now's probably a good time to think again about the purpose of a digital prototype. In the first part of the book, there was considerable discussion of paper prototypes and why they are useful. Paper game prototypes help you do the following:

- Test, reject, and/or refine game mechanics and rules quickly
- Explore the dynamic behavior of your game and understand the emergent possibilities created by the rules
- Ascertain whether rules and gameplay elements are easily understood by players
- Understand the emotional response that players have to your game

Digital prototypes also add the fantastic ability to see how the game *feels*; in fact, that is their primary purpose. Although it is possible for you to spend hours describing game mechanics to someone in detail, it is much more efficient (and interesting) for them to just play the game and see how it feels. This is discussed at length in the book *Game Feel* by Steve Swink.¹

¹ Steve Swink, *Game Feel: A Game Designer's Guide to Virtual Sensation* (Boston: Elsevier, 2009).

In this chapter, you create a working game, and the end result will be something that you can show to friends and colleagues. After letting them play it for a while, you can ask whether the difficulty feels too easy, too difficult, or just right. Use that information to tweak the variables in the game and custom craft a specific difficulty for each of them.

Let's get started making Apple Picker.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup Procedure.”](#)

- **Project name:** Apple Picker Prototype
- **Scene name:** _Scene_0 (The underscore at the beginning of the scene name will keep it sorted at the top of the Project pane.)
- **C# script names:** ApplePicker, Apple, AppleTree, and Basket

Attach the ApplePicker C# script to the Main Camera in _Scene_0. Do not attach the C# scripts Apple, AppleTree, or Basket to anything.

Preparing

Happily, you've already done a lot of the preparation for this prototype in [Chapter 15](#), “[Thinking in Digital Systems](#),” when we analyzed Apple Picker and the classic game *Kaboom!* As mentioned in that chapter, Apple Picker will have the same game mechanics as *Kaboom!* It's very useful for you as a designer to understand classic games and how they worked, and this is a nice easy one to start with. Take a moment to look back at [Chapter 15](#) and make sure that you understand the flow charts for each element: the AppleTree, the Apples, and the Baskets.

Getting Started: Art Assets

As a prototype, this game doesn't need fantastic art; it needs to work. The kind of art that we'll create throughout this book is commonly known as *programmer art*, which is the placeholder art made by programmers that will eventually be replaced by the actual game art created by artists. As with nearly everything in a prototype, the purpose of this art is to get you from a concept to a playable prototype as quickly as possible. If your programmer art doesn't look terrible, that's nice, but it's certainly not necessary.

AppleTree

From the menu bar, choose *GameObject > Create Other > Cylinder*. This will be the trunk of our tree. Set the name of the Cylinder to *Trunk* by selecting it in the Hierarchy and clicking its name at the top of the Inspector. Set the Transform component of Trunk to match the settings of the Transform component shown in [Figure 28.1](#).

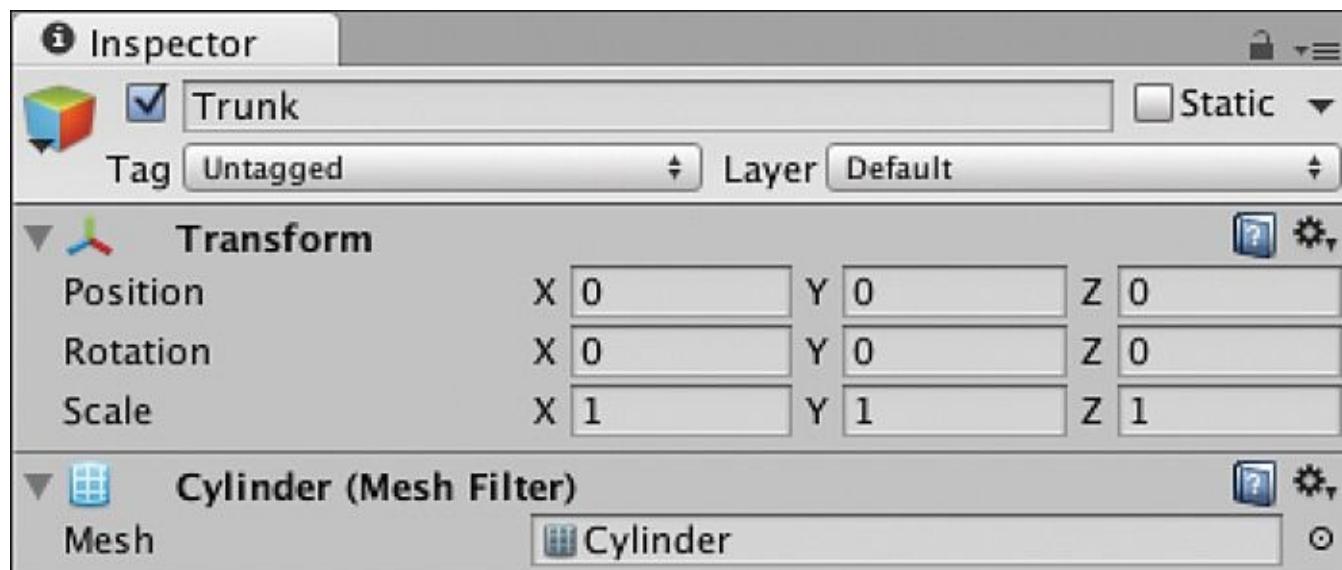


Figure 28.1 The Transform component for the cylinder named Trunk

Throughout the tutorials in this book, I use the following format to give you settings for *GameObject* transform components:

Trunk (Cylinder) P:[0,0,0] R:[0,0,0] S:[1,1,1]

The preceding line instructs you to set the transform of the *GameObject* named Trunk to a position of x=0, y=0, and z=0; a rotation of x=0, y=0, and z=0; and a scale of x=1, y=1, and z=1. The word *Cylinder* in parentheses tells you the type of *GameObject* that it is. You will also sometimes see this format listed in the middle of a paragraph as P:[0,0,0] R:[0,0,0] S:[1,1,1].

Now choose *GameObject > Create Other > Sphere*. Set its transform as follows:

Sphere (Sphere) P:[0,0.5,0] R:[0,0,0] S:[3,2,3]

This sphere and the cylinder together should look (a bit) like a tree, but they are currently two separate objects. You need to create an empty GameObject to contain the two of them in a single object.

From the menu bar, choose *GameObject > Create Empty*. This should create an empty GameObject. Make sure that its transform is set to the following:

GameObject (Empty) P:[0,0,0] R:[0,0,0] S:[1,1,1]

An empty GameObject only includes a Transform component, and it is therefore a simple, useful container for other GameObjects.

In the Hierarchy pane, first change the name of GameObject to *AppleTree*. This can be done by clicking the name GameObject to highlight it, waiting for a second, and either pressing Return on the keyboard (F2 on PC) or clicking it a second time. Next, individually drag the Trunk and Sphere GameObjects onto AppleTree (similar to how you attach a C# script to a GameObject), and they will be placed under AppleTree in the Hierarchy. You can click the new disclosure triangle next to the word *AppleTree* to see them. When you're done, your AppleTree should look like the one shown in [Figure 28.2](#).

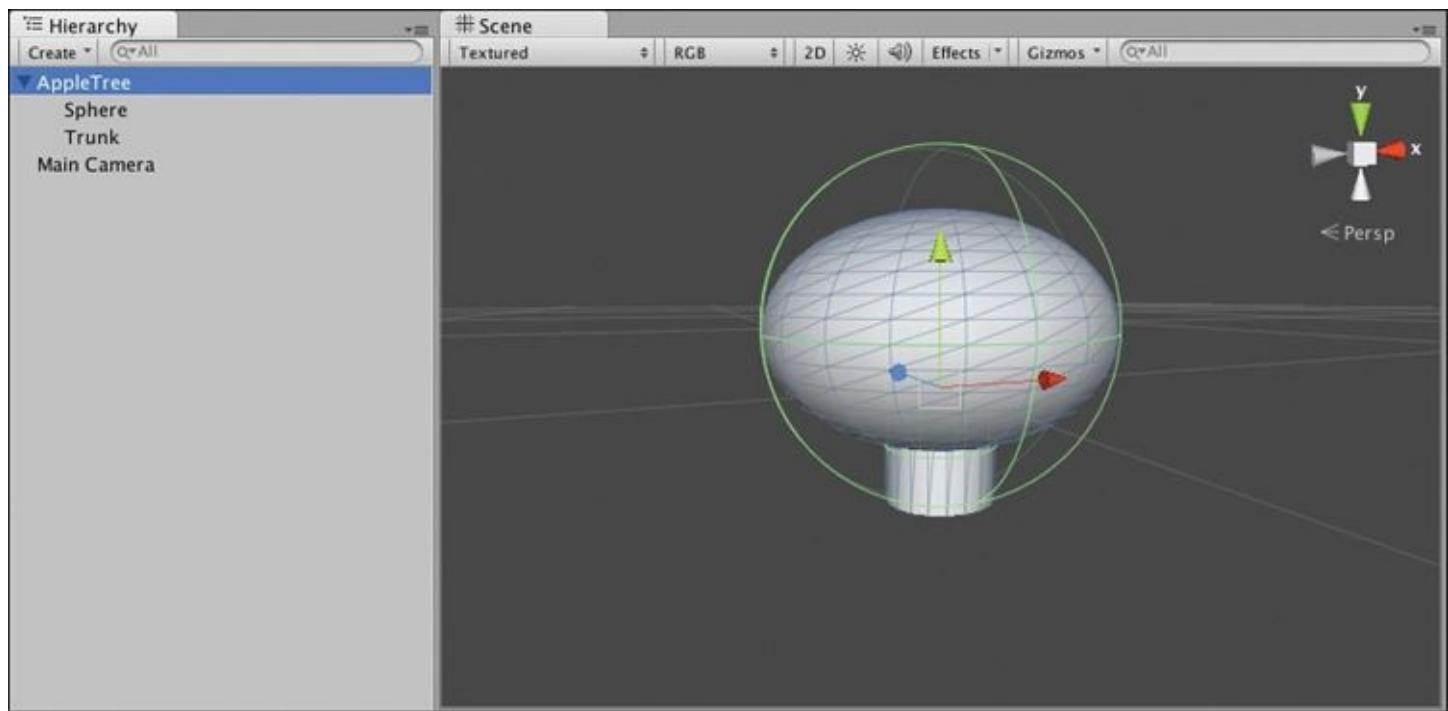


Figure 28.2 The AppleTree shown in both the Hierarchy and Scene panes

Now that Trunk and Sphere GameObjects are parented to AppleTree, if you move, scale, or rotate AppleTree, both Trunk and Sphere will move, rotate, and scale alongside it. Give it a try by manipulating the Transform component of AppleTree. After you're done playing with this, set the transform of AppleTree to the following:

AppleTree P:[0,0,0] R:[0,0,0] S:[2,2,2]

These settings move the AppleTree to the origin and make it twice as large as it was initially.

Simple Materials for AppleTree

Though this is all programmer art, that doesn't mean that it has to be just flat shaded white objects. Let's add a little color to the scene.

From the menu bar, choose *Assets > Create > Material*. This will make a new material in the Project pane. Rename this material to *Mat_Wood*. Set the main color in the Inspector for *Mat_Wood* to a

brown of your liking. Then, drag the *Mat_Wood* material onto Trunk in your Scene or Hierarchy pane.

Do the same to create a material named *Mat_Leaves*, color it green, and drag it onto Sphere in your Hierarchy or Scene pane.

Once this is done, drag AppleTree from the Hierarchy pane over to the Project pane to make a prefab from it. As you saw in previous chapters, this creates an AppleTree prefab in the Project pane and turns the name of AppleTree in the Hierarchy blue.

Add some light to the scene by choosing *GameObject > Create Other > Directional Light* from the menu bar. Set the position, rotation, and scale of the directional light to the following:

Directional Light P:[0,10,0] R:[50,-30,0] S:[1,1,1]

This should put a nice diagonal light across the scene. It's worth noting here that the position of a directional light is unimportant—directional lights shine in the same direction regardless of position—but I've given you the position of [0,10,0] to move it out of the middle of the scene view since its *gizmo* (that is, icon) would be in the middle of the Scene pane otherwise.

Now, move the AppleTree up and out of the way a bit. Select the AppleTree in the Hierarchy and change its position to:

AppleTree P:[0,4,0] R:[0,0,0] S:[2,2,2]

This may move it out of the view of the Scene pane, but you can zoom out to see it by scrolling your mouse wheel while the mouse cursor is over the Scene pane.

Apple

Now that you have the AppleTree, you need to make the Apple GameObject prefab that it will drop. From the menu bar, choose *GameObject > Create Other > Sphere*. Rename this sphere to *Apple*, and set its transform as follows:

Apple (Sphere) P:[0,0,0] R:[0,0,0] S:[1,1,1]

Create a new material named *Mat_Apple*, color it red (or light green, if you prefer green apples), and attach it to Apple.

Adding Physics to the Apple

Select Apple in the Hierarchy pane. From the menu bar, choose *Component > Physics > Rigidbody*. As you may remember from [Chapter 16, “Introducing Our Development Environment: Unity”](#), the Rigidbody component enables an object to react to physics (e.g., fall, collide with other objects). If you click the Play button now, you will see Apple fall off screen due to gravity.

Giving Apples the Tag “Apple”

Eventually you will want to quickly get an array of all the Apple GameObjects on screen, and giving the Apples a specific tag can help with this. With Apple selected in the Hierarchy, choose *Add Tag* from the pop-up menu next to *Tag* in the Inspector, as shown in the left half of [Figure 28.3](#). This will bring up Unity's *Tag and Layers Manager* in the Inspector. Click the disclosure triangle next to Tags and type *Apple* into the Element 0 field, as shown in the right half of [Figure 28.3](#).

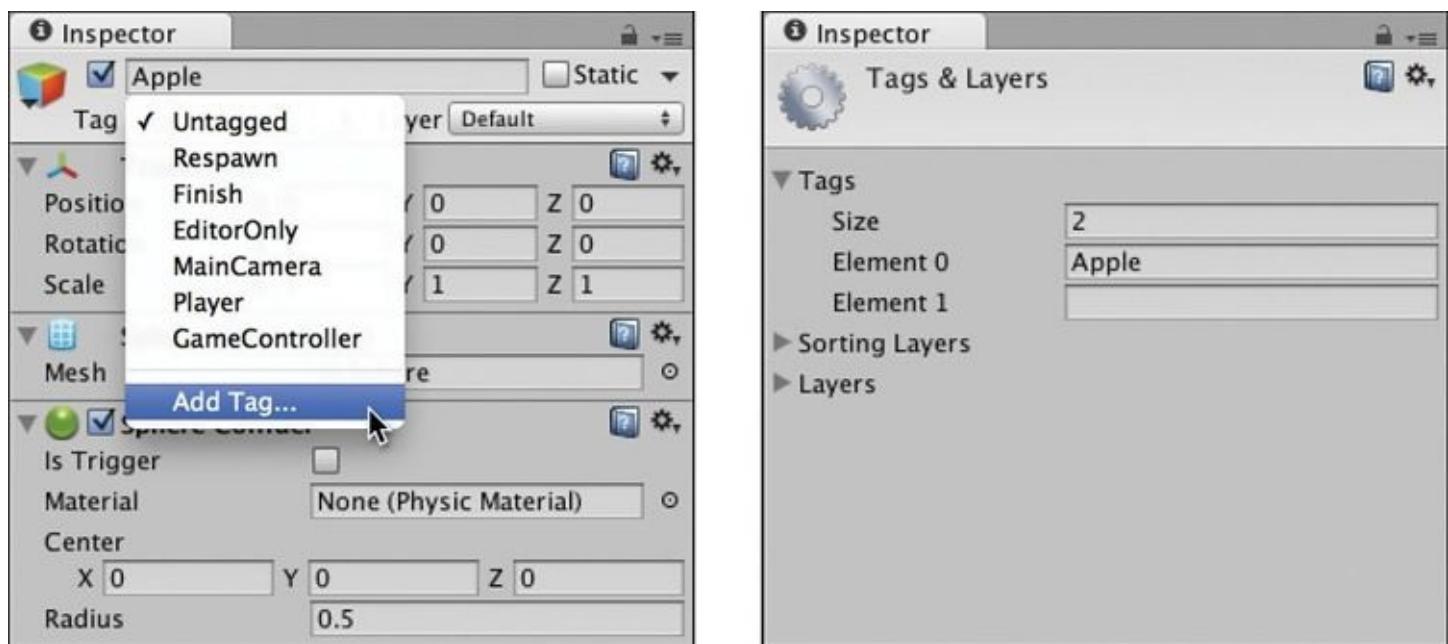


Figure 28.3 Adding the Apple tag to the list of tags

Now, click Apple in the Hierarchy again to return to the Inspector for Apple. Clicking the Tags pop-up menu once more will now give you Apple as a tag option. Choose *Apple* from the list of tags. All apples will now have the tag *Apple*, which will make them easier to identify and select.

Making the Apple Prefab

Finally, drag Apple from the Hierarchy pane to the Project pane to make it a prefab. Once you're sure that there is an Apple prefab in the Project pane, click the Apple instance in the Hierarchy pane and delete it (by choosing Delete from the right-click menu or by pressing Command-Delete [just Delete for Windows] on your keyboard). Because the apples in our game will be instantiated from the Apple prefab in the Project pane, we don't need to start with one in the scene.

Basket

Like the other art assets, the basket will be very simple. Choose *GameObject > Create Other > Cube* from the menu bar. Rename Cube to *Basket* and set its transform to the following:

Basket (Cube) **P:[0,0,0] R:[0,0,0] S:[4,1,4]**

This should give you a flat, wide rectangular solid. Now create a new material named *Mat_Basket*, color it a light, desaturated yellow (like straw), and apply it to Basket. Drag Basket from the Hierarchy pane to the Project pane to make it into a prefab and delete the remaining instance of Basket from the Hierarchy. Be sure to save your scene.

Your Project and Hierarchy panes should now look like [Figure 28.4](#).

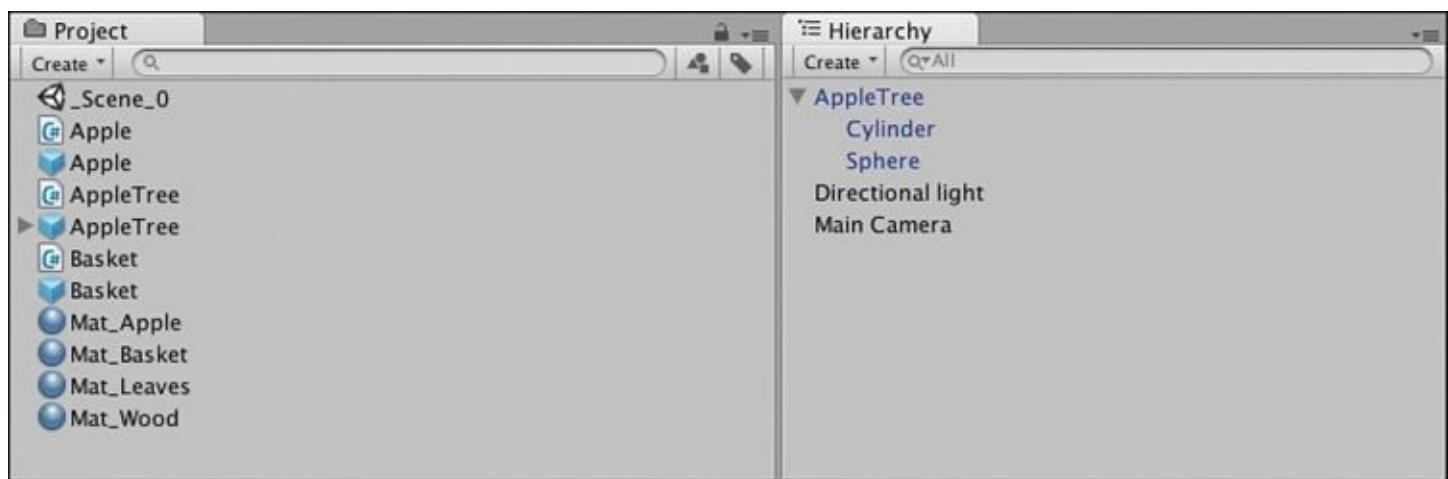


Figure 28.4 The Project and Hierarchy panes at this point in the prototype

Camera Setup

One of the most important things to get right in your games is the position of the camera. For Apple Picker, we want a camera that will show a decent-sized play area. Because the gameplay in this game is entirely two dimensional, we also want an orthographic camera instead of a perspective one. See the sidebar for more information on the two types of camera projections.

Orthographic Versus Perspective Cameras

Orthographic and *perspective* are two types of virtual 3D cameras in games and can be seen in [Figure 28.5](#).

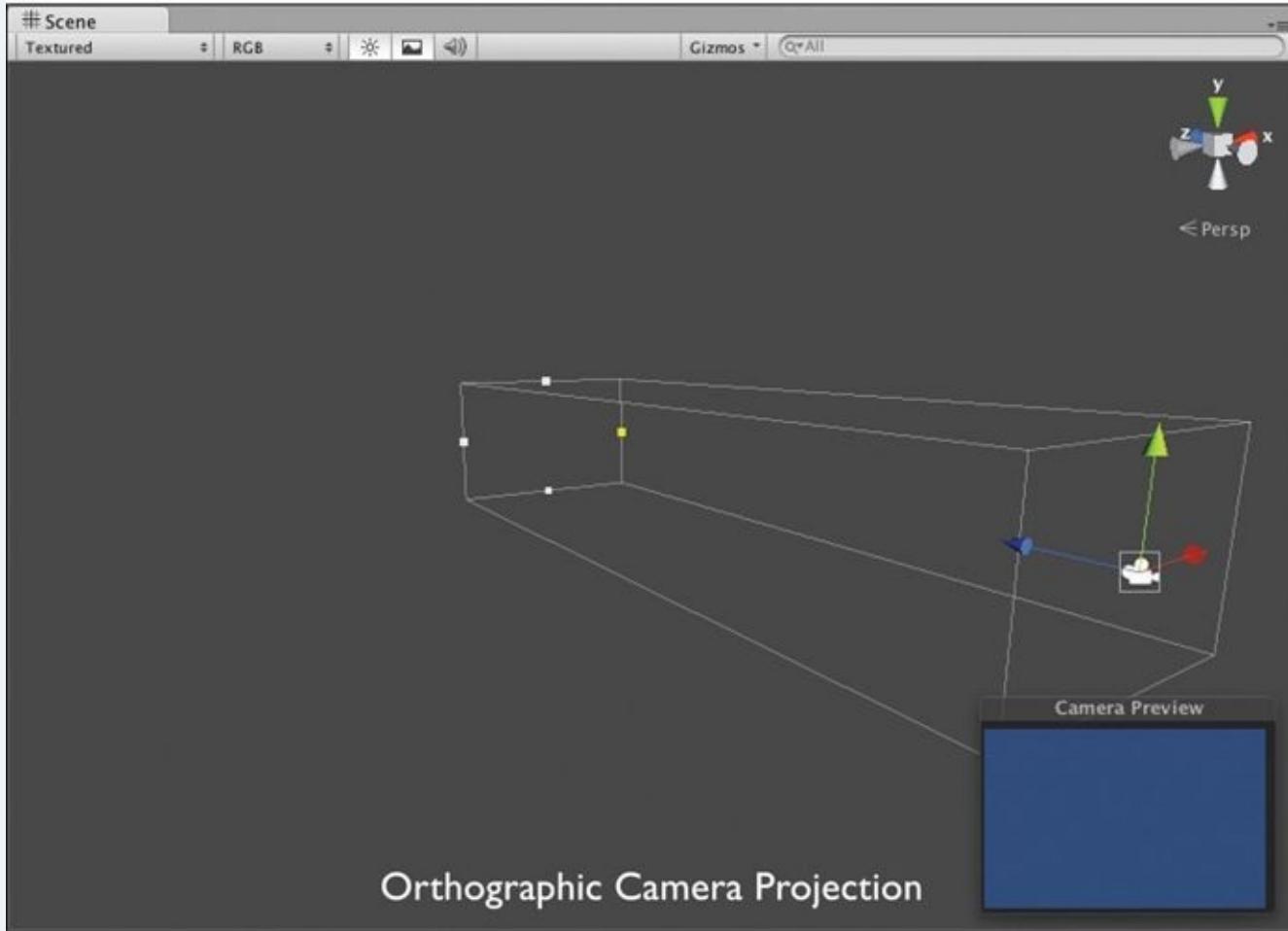
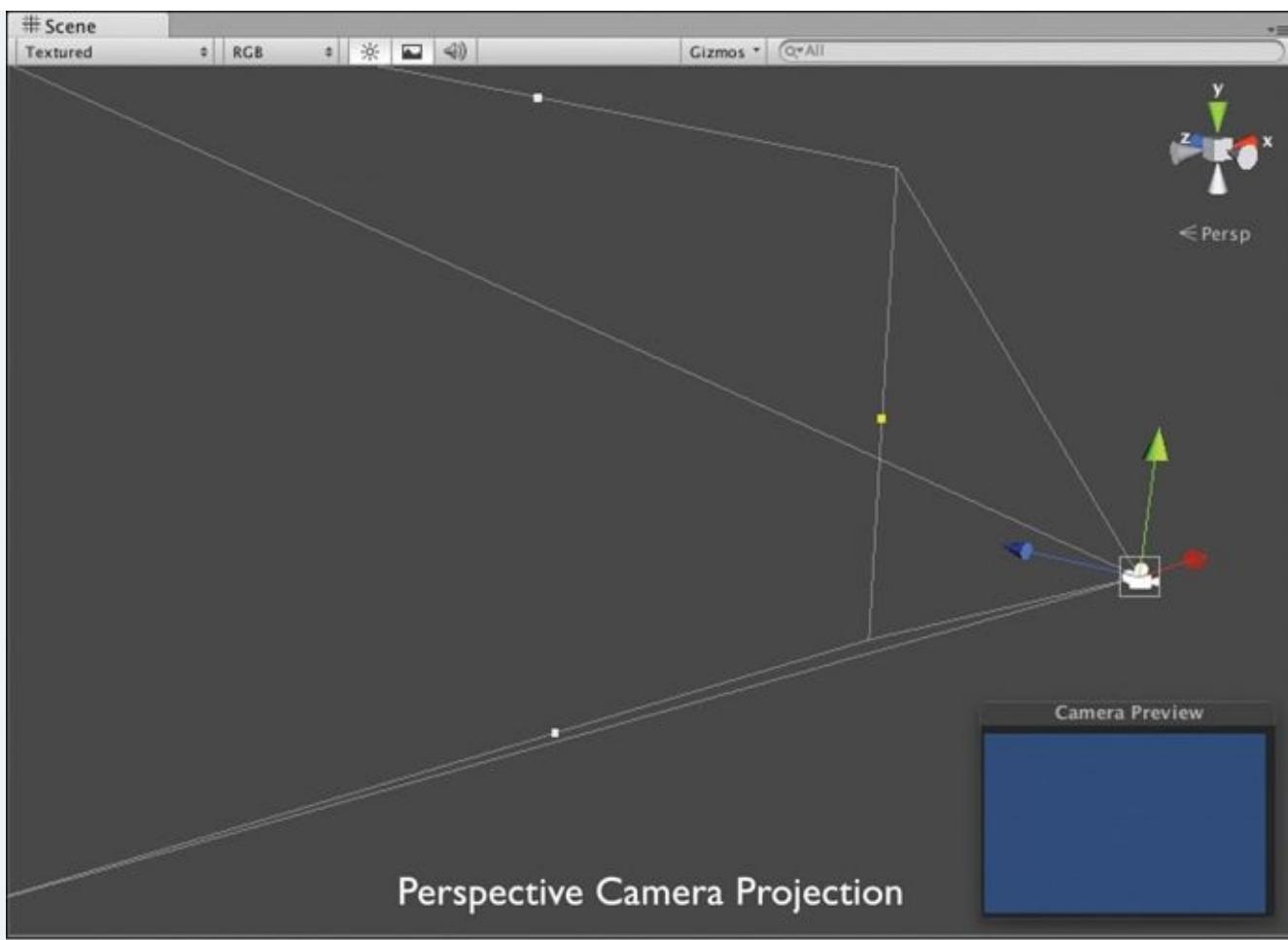


Figure 28.5 Comparison of perspective and orthographic camera projections

A *perspective* camera works like the human eye; because light comes in through a lens,

objects that are close to the camera appear larger, and objects that are farther away appear smaller. This gives a perspective camera a field of view (a.k.a. projection) shaped like a square frustum (or more simply, like a square pyramid with the top cut off). To see this, click Main Camera in your hierarchy, and then zoom out in the Scene pane. The pyramidal wireframe shape extending out from the camera is the *view frustum* and shows everything that the camera will see.

Through an *orthogonal* camera, an object will appear to be the same size regardless of how far it is from the camera. The projection for an orthogonal camera is rectangular rather than frustum shaped. To see this, select Main Camera in the Hierarchy pane. Find the Camera component in the Inspector and change the projection from Perspective to Orthogonal. Now, the gray view frustum represents a 3D rectangle rather than a pyramid.

It is also sometimes useful to set the Scene pane to be orthogonal rather than perspective. To do this, click the word *<Persp* under the axes gizmo in the upper-right corner of the Scene pane (see each of the images in [Figure 28.5](#)). Clicking the *<Persp* under the axes gizmo will switch between perspective and *isometric* (abbreviated =*Iso*) scene views (isometric being another word for orthographic).

Camera Settings for Apple Picker

Select Main Camera in the Hierarchy pane. In the Inspector, set the Camera component's projection to *Orthographic*. Now, set the *size* to 16. This makes the AppleTree appear to be a good size and leaves room for the apples to fall and be caught by the player. Often, you will make a good first guess at things like camera settings and then will refine them once you've had a chance to play the game. Just like everything else in game development, there is an iterative process to finding the right settings for the camera. For now, I recommend the following:

Main Camera (Camera) P:[0,0,-10] R:[0,0,0] S:[1,1,1]

This position will move the camera viewpoint down 1 meter (a unit in Unity is the equivalent of 1m in length) to be at a height of exactly 0. Because Unity units are equivalent to meters, I sometimes abbreviate "1 unit" as 1m in this book. Your final Main Camera Inspector should now look like what is shown in [Figure 28.6](#).

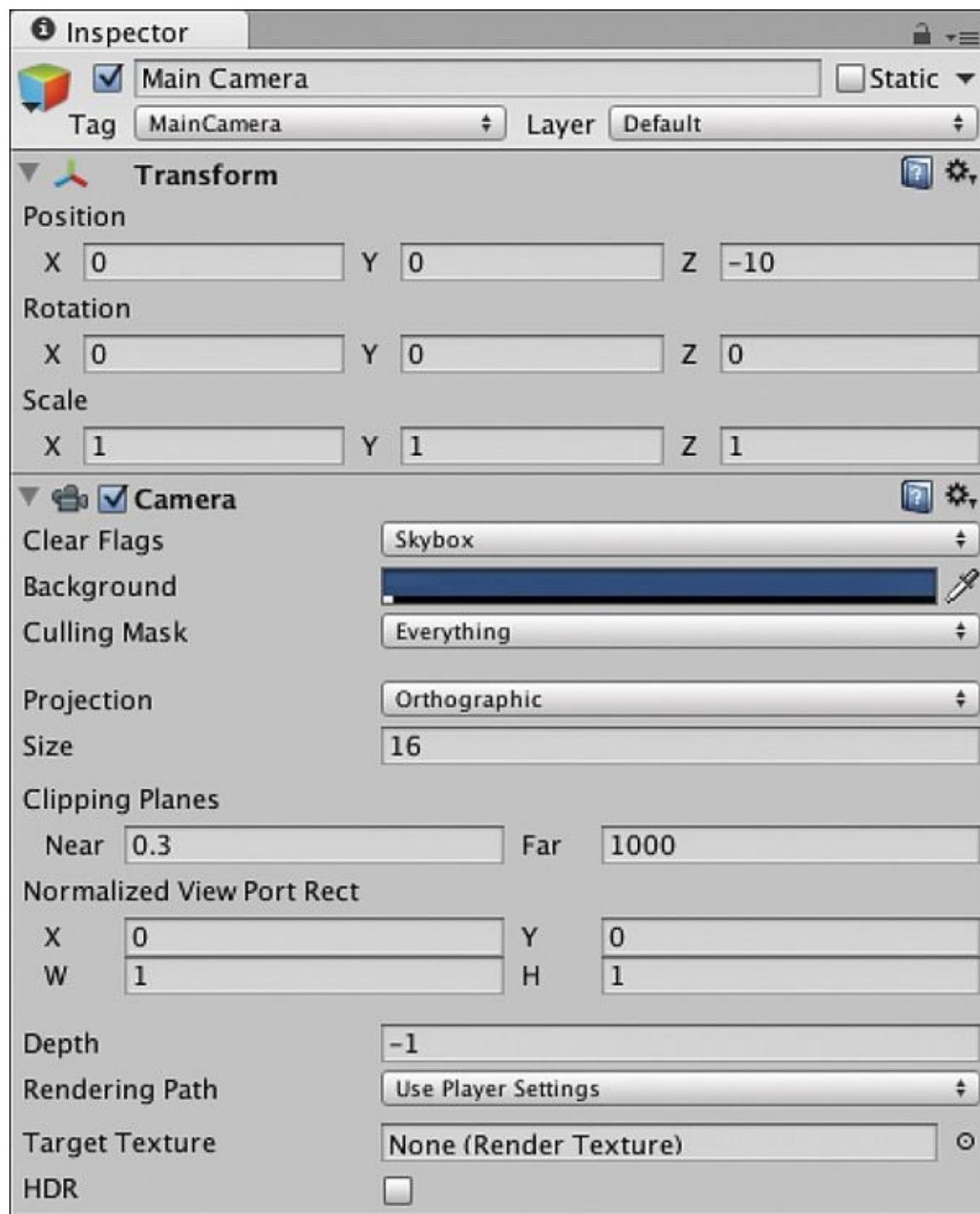


Figure 28.6 Main Camera Inspector settings

Coding the Apple Picker Prototype

Now it's time to make the code of this game prototype actually work. [Figure 28.7](#) presents the flow chart of the AppleTree's actions from [Chapter 15](#).

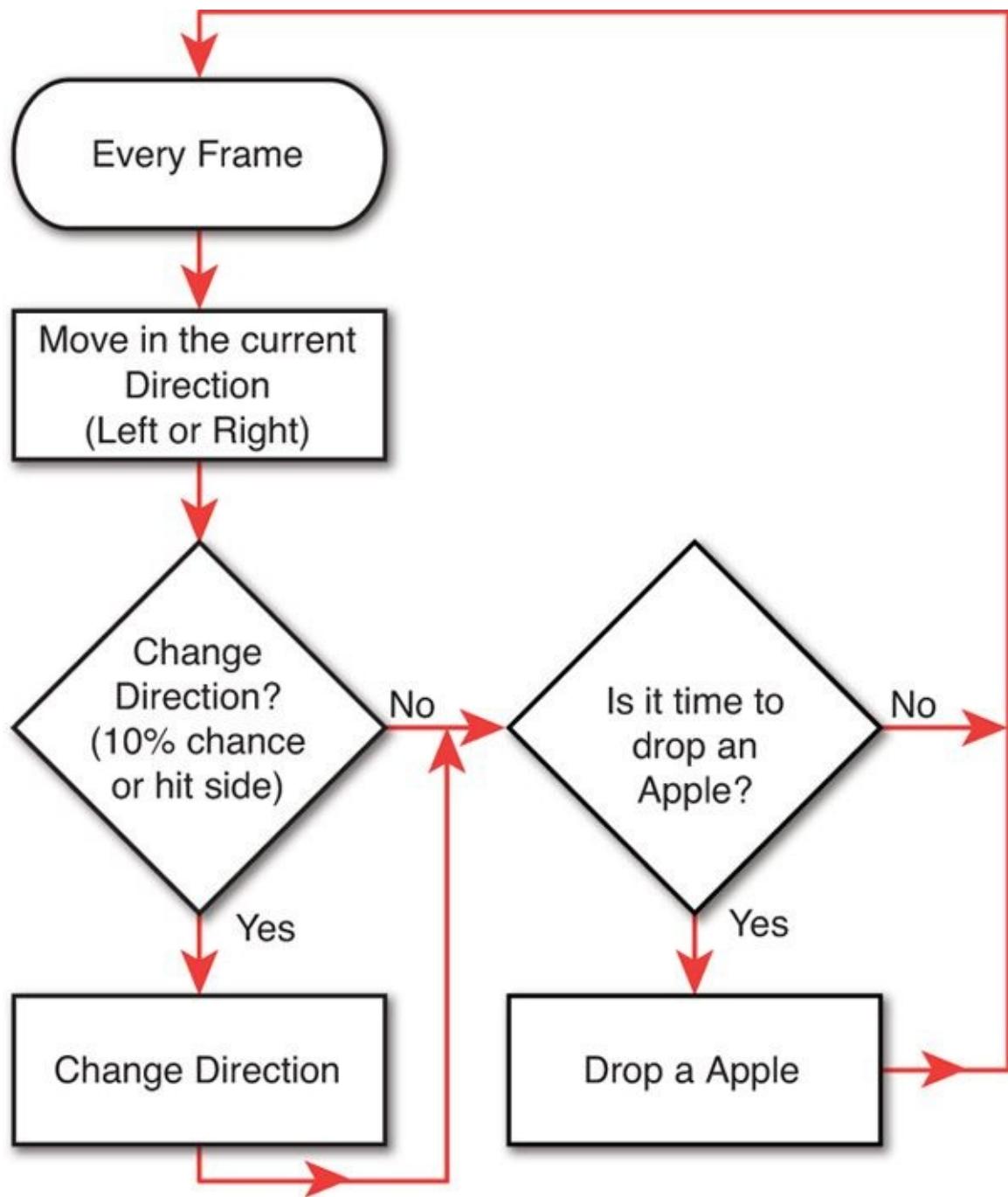


Figure 28.7 AppleTree flow chart

The actions we will need to code for the AppleTree are as follows:

1. Move at a certain speed every frame.
2. Change directions upon hitting the edge of the play area.
3. Change directions based on random chance.
4. Drop an Apple every second.

That's it! Let's start coding. Double-click the AppleTree C# script in the Project pane to open it. We will need some configuration variables, so alter the AppleTree class to look like this:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;

public class AppleTree : MonoBehaviour {
  
```

// Prefab for instantiating apples

```

public GameObject applePrefab;

// Speed at which the AppleTree moves in meters/second
public float speed = 1f;

// Distance where AppleTree turns around
public float leftAndRightEdge = 10f;

// Chance that the AppleTree will change directions
public float chanceToChangeDirections = 0.1f;

// Rate at which Apples will be instantiated
public float secondsBetweenAppleDrops = 1f;

void Start () {
    // Dropping apples every second
}

void Update () {
    // Basic Movement
    // Changing Direction
}
}

```

You may have noticed that the preceding code does not include the line numbers that were present in prior chapters. The code listings in this part of the book will generally not have line numbers because I needed every character possible to fit the code on the page.

To see this code actually do something, you need to attach it to the AppleTree GameObject. Drag the AppleTree C# script from the Project pane onto the AppleTree prefab that is also in the Project pane. Then, click the AppleTree instance in the Hierarchy pane, and you'll see that the script has been added not only to the AppleTree prefab but also to its instance in the Hierarchy.

With the AppleTree selected in the Hierarchy, you should see all of the variables you just typed appear in the Inspector under the *AppleTree (Script)* component.

Try moving the AppleTree around in the scene by adjusting the X and Y coordinates in the Transform Inspector to find a good height (position.y) for the AppleTree and a good limit for left and right movement. On my machine, 12 looks like a good position.y, and it looks like the tree can move from -20 to 20 in position.x and still be on screen. Set the position of AppleTree to [0,12,0] and set the leftAndRightEdge float in the AppleTree (Script) component Inspector to 20.

The Unity Engine Scripting Reference

Before you get too far into this project, it's extremely important that you remember to look at the Unity Scripting Reference if you have any questions at all about the code you see here. There are two ways to get into the Script Reference:

1. Choose *Help > Scripting Reference* from the menu bar in Unity. This brings up the Scripting Reference that is saved locally on your machine, meaning that it will work even without a connection to the Internet. You can type any function or class name into the search field on the left to find out more about it.

Enter *MonoBehaviour* into the search field and press Return. Then click the first result to see all the methods built in to every MonoBehaviour script (and by extension, built in to every class script you will attach to a GameObject in Unity). For readers from the United States, note the European spelling of *Behaviour*.

2. When working in MonoDevelop, select any text you would like to learn more about and then choose *Help > Unity API Reference* from the menu bar. This will launch an online version of the Unity Scripting Reference, so it won't work properly without Internet access, but it has the exact same information as the local reference that you can reach through the first method.

Unfortunately, all the code examples in the Scripting Reference default to JavaScript, but there is either a pop-up menu or a C# button (depending on the version of the documentation) that allows you to switch nearly all code examples to C#. Trust me; this is a small price to pay for using a vastly superior language.

Basic Movement

Right now, rather than include code that actually moves the AppleTree, this script only includes code comments (preceded by `/*`) that describe the actions that will be added to the code. I often find it useful to list these actions in code comments first and then add functionality progressively. This can help you organize your thoughts and is similar to writing an outline for a paper.

Now, make these bolded changes to the `Update` method in the AppleTree script:

[Click here to view code image](#)

```
void Update () {
    // Basic Movement
Vector3 pos = transform.position;
pos.x += speed * Time.deltaTime;
transform.position = pos;
    // Changing Direction
}
```

The first bold line in this code defines the `Vector3 pos` to be the current position of the AppleTree. Then, the `x` component of `pos` is increased by the speed times `Time.deltaTime` (which is a measure of the number of seconds since the last frame). This makes the movement of the AppleTree *time-based*, which is a very important concept in game programming (see the sidebar “[Making Your Games Time-Based](#)”). The third line assigns this modified `pos` back to `transform.position` (which moves AppleTree to a new position).

Making your Games Time-Based

When movement in a game is *time-based*, it happens at the same speed regardless of the framerate at which the game is running. `Time.deltaTime` enables this because it tells us the number of seconds that have passed since the last frame. `Time.deltaTime` is usually very small. For a game running at 25 fps (frames per second), `Time.deltaTime` is `0.04f`, meaning that each frame takes $4/100^{\text{th}}$ s of a second to display. If this line of code were run at 25 fps, the result would resolve like this:

[Click here to view code image](#)

```
pos.x += speed * Time.deltaTime;
pos.x += 1.0f * 0.04f;
pos.x += 0.04f;
```

So, in $1/25^{\text{th}}$ of a second, `pos.x` would increase by 0.04m per frame. Over the course of a full second, `pos.x` would increase by 0.04m per frame * 25 frames, for a total of 1 meter

in 1 second. This equals the 1m/s that speed is set to.

If instead the game were running at 100 fps, it would resolve as follows:

[Click here to view code image](#)

```
pos.x += speed * Time.deltaTime;  
pos.x += 1.0f * 0.01f;  
pos.x += 0.01f;
```

So, in 1/100th of a second, `pos.x` would increase by 0.01m per frame. Over the course of a full second, `pos.x` would increase by `0.01m per frame * 100 frames`, for a total of 1 meter in 1 second.

Time-based movement ensures that regardless of framerate, the elements in your game will move at a consistent speed, and it is this consistency that will enable you to make games that are enjoyable for both players using the latest hardware and those using older machines. Time-based coding is also very important to consider when programming for mobile devices because the speed and power of mobile devices is changing very quickly.

You might be wondering why this was three lines instead of just one. Why couldn't the code just be this:

[Click here to view code image](#)

```
transform.position.x += speed * Time.deltaTime;
```

The answer is that `transform.position` is a *property*, a method that is masquerading as a field through the use of `get {}` and `set {}` accessors (see [Chapter 25, “Classes”](#)). Although it is possible to read the value of a property's subcomponent, it is not possible to set a subcomponent of a property. In other words, `transform.position.x` can be read, but it cannot be set directly. This necessitates the creation of the intermediate `Vector3 pos` that can be modified and then assigned back to `transform.position`.

When you press the Play button, you'll notice that the AppleTree is moving very slowly. Try some different values for `speed` in the Inspector and see what feels good to you. I personally set it to 10, which makes it move at 10m/s (10 meters per second or 10 Unity units per second).

Changing Direction

Now that the AppleTree is moving at a decent rate, it will run off of the screen pretty quickly. Let's make it change directions when it hits the `leftAndRightEdge` value. Modify the AppleTree script as follows:

[Click here to view code image](#)

```
void Update () {  
    // Basic Movement  
    Vector3 pos = transform.position;  
    pos.x += speed * Time.deltaTime;  
    transform.position = pos;  
    // Changing Direction  
    if ( pos.x < -leftAndRightEdge ) {  
        speed = Mathf.Abs(speed); // Move right  
    } else if ( pos.x > leftAndRightEdge ) {  
        speed = -Mathf.Abs(speed); // Move left  
    }  
}
```

Press Play and see what happens. The first line under `//Changing Direction` checks to see whether the new `pos.x` that was just set in the previous lines is less than the side-to-side limit that is set by `leftAndRightEdge`. If `pos.x` is too small, `speed` is set to `Mathf.Abs(speed)`, which takes the absolute value of `speed`, guaranteeing that the resulting value will be positive, which translates into movement to the right. If `pos.x` is greater than `leftAndRightEdge`, then `speed` is set to the negative of `Mathf.Abs(speed)`, ensuring that the `AppleTree` will move to the left.

Changing Direction Randomly

Add the bolded lines shown here to introduce random changes in direction as well:

[Click here to view code image](#)

```
// Changing Direction
if ( pos.x < -leftAndRightEdge ) {
    speed = Mathf.Abs(speed); // Move right
} else if ( pos.x > leftAndRightEdge ) {
    speed = -Mathf.Abs(speed); // Move left
} else if ( Random.value < chanceToChangeDirections ) {
    speed *= -1; // Change direction
}
```

`Random.value` is a static property of the class `Random` that returns a random float value between 0 and 1 (non-inclusive, which means that the results can get very close to 1, but `Random.Value` will never actually return 1 itself). If this random number is less than `chanceToChangeDirections`, the `AppleTree` will change directions by setting `speed` to the negative of itself. If you press Play, you'll see that this happens far too often at a `chanceToChangeDirections` of `0.1f`. In the Inspector, change the value of `chanceToChangeDirections` to `0.02`, and it should feel a lot better. Note that you do not add the `f` at the end when typing a float value into the Inspector.

To continue the discussion of time-based games, this chance to change directions is actually not time based. Every frame, there is a 2% chance that the `AppleTree` will change directions. On a very fast computer, that chance could happen 200 times per second (yielding an average of 4 direction changes per second), whereas on a slow computer, it could happen as few as 30 times per second (for an average of 0.6 direction changes per second). To fix this, move the direction change code out of `Update()` (which is called as fast as the computer can render frames) into `FixedUpdate()` (which is called exactly 50 times per second, regardless of the computer on which it's running).

[Click here to view code image](#)

```
void Update () {
    // Basic Movement
    Vector3 pos = transform.position;
    pos.x += speed * Time.deltaTime;
    transform.position = pos;
    // Changing Direction
    if ( pos.x < -leftAndRightEdge ) {
        speed = Mathf.Abs(speed); // Move right
    } else if ( pos.x > leftAndRightEdge ) {
        speed = -Mathf.Abs(speed); // Move left
    }
}

void FixedUpdate() {
    // Changing Direction Randomly
    if ( Random.value < chanceToChangeDirections ) {
        speed *= -1; // Change direction
    }
}
```

```
}
```

This will cause the AppleTree to randomly change directions an average of 1 time every second (50 FixedUpdatees per second * a random chance of 0.02 = 1 time per second). You should also note that the code for the AppleTree class still lists the value for chanceToChangeDirections as 0.1f. However, because chanceToChangeDirections is a public field, it is /serialized/ by Unity, which allows it to be seen in the Inspector and allows the value of 0.02 in the Inspector to override the value in the script. If you were to change the value of this field in the script, you would not see any change in the behavior of the game because the Inspector value will always override the value in the script for any serialized field.

Dropping Apples

Select AppleTree in the Hierarchy and look at the *Apple Tree (Script)* component in its Inspector. Currently, the value of the field applePrefab is *None (Game Object)*, meaning that it has not yet been set (the *GameObject* in parentheses is there to let you know that the type of the applePrefab field is GameObject). This value needs to be set to the Apple GameObject prefab in the Project pane. You can do this either by clicking the tiny target to the right of *Apple Prefab None (Game Object)* in the Inspector and selecting Apple from the Assets tab or by dragging the Apple GameObject prefab from the Project pane onto the applePrefab value in the Inspector pane.

Return to MonoDevelop and add the following bolded code to the AppleTree class:

[Click here to view code image](#)

```
void Start () {
    // Dropping apples every second
    InvokeRepeating( "DropApple", 2f, secondsBetweenAppleDrops );
}

void DropApple() {
    GameObject apple = Instantiate( applePrefab ) as GameObject;
    apple.transform.position = transform.position;
}
```

The InvokeRepeating function will call another named function on a repeating basis. In this case, the first argument tells it to call the new function DropApple(). The second argument, 2f, tells InvokeRepeating to wait 2 seconds before the first time that it calls DropApple(). The third argument tells it to then call DropApple() again every secondsBetweenAppleDrops seconds thereafter (in this case, every 1 second based on the settings in the Inspector). Press Play and see what happens.

Did you expect the Apples to fly off to the sides? Remember the Hello World example that we did with all the cubes flying all over the place? The same thing is happening here. The Apples are colliding with the AppleTree, and that causes them to fly off to the left and right rather than falling straight down. To fix this, you need to put them in a *layer* that doesn't collide with the AppleTree. Layers are groups of objects that can either collide with or ignore each other. If the AppleTree and Apple GameObjects are placed in two different physics layers, and those layers are set to ignore each other in the Physics Manager, then the AppleTree and Apples will cease colliding with each other.

Setting GameObject Layers

First, you will need to make some new layers. Click the AppleTree in the Hierarchy and then choose *Add Layer* from the pop-up menu next to Layer. This will open up the *Tags and Layers Manager* in the Inspector, which allows you to set the names of layers under the *Layers* label (make sure you're not editing *Tags* or *Sorting Layers*). You can see that Builtin Layers 0 through 7 are grayed out, but you are able to edit User Layers 8 through 31. Name User Layer 8 *AppleTree*, User Layer 9 *Apple*, and User Layer 10 *Basket*. It should look like [Figure 28.8](#).

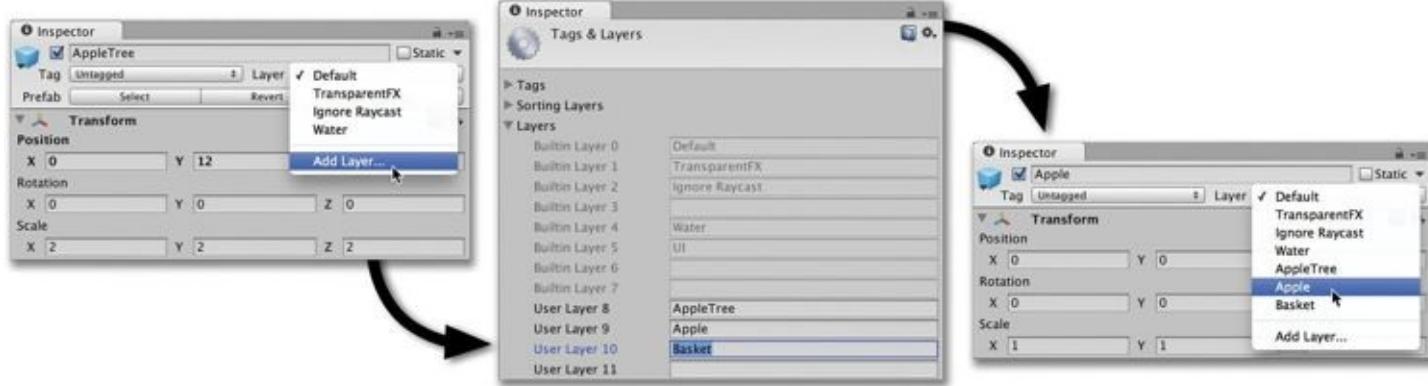


Figure 28.8 The steps required to make new physics layers and assign them

From the menu bar, now choose *Edit > Project Settings > Physics*. This will set the Inspector to the *Physics Manager*. The *Layer Collision Matrix* grid of check boxes at the bottom of the Physics Manager sets which layers will collide with each other (and whether GameObjects in the same layer will collide with each other as well). You want the Apple to collide with neither the AppleTree nor other Apples, but to still collide with the Basket, so your Layer Collision Matrix grid should look like what is shown in [Figure 28.9](#).

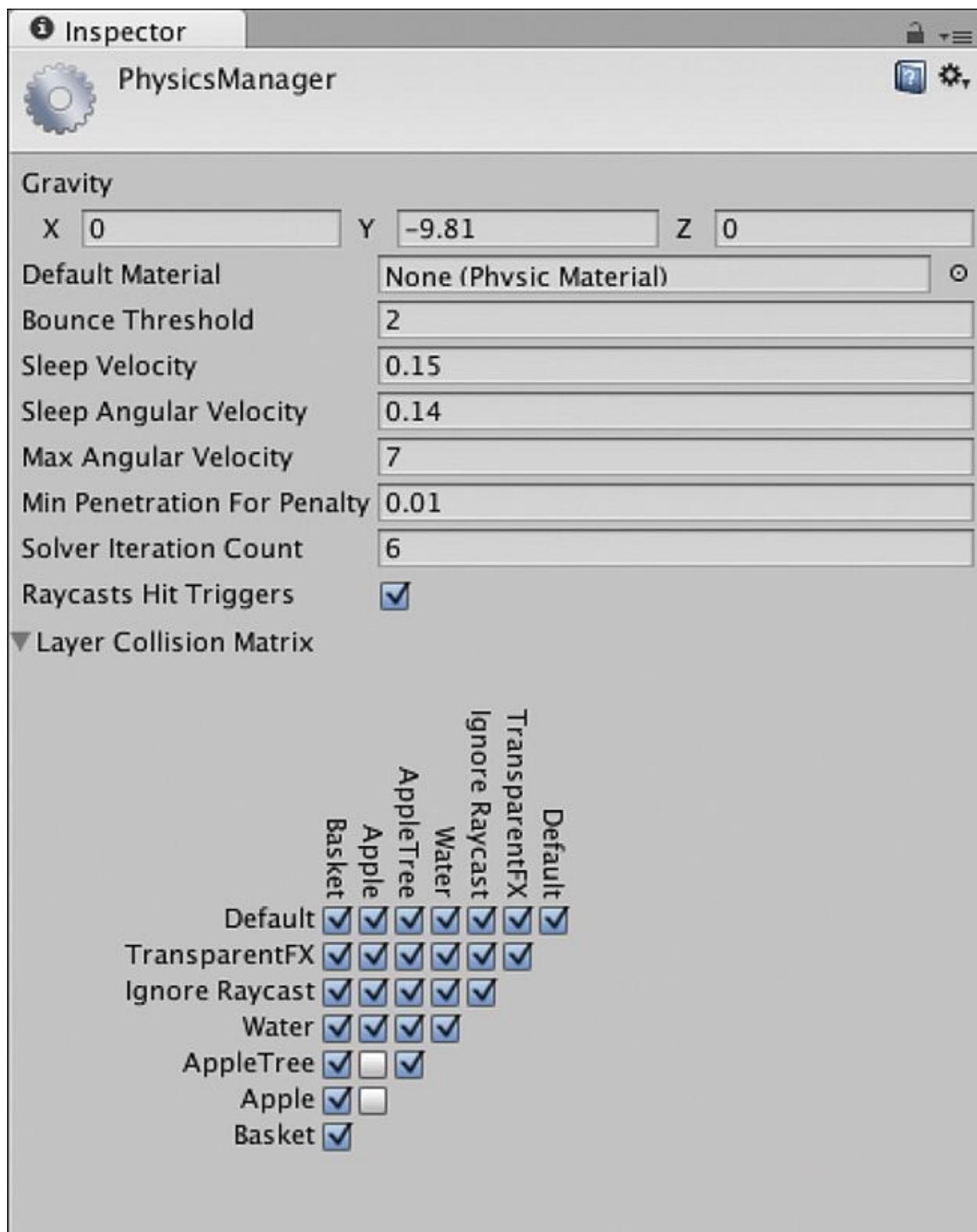


Figure 28.9 The required Layer Collision Matrix settings in the Physics Manager

Now that the Layer Collision Matrix is set properly, it's time to assign layers to the important GameObjects in the game. Click Apple in the Project pane. Then, at the top of the Inspector, select the Apple layer from the pop-up menu next to Layer at the top of the Inspector pane. Select the Basket in the Project pane and set its Layer to Basket. Then select the AppleTree in the Project pane and set its Layer to AppleTree. When you choose the layer for AppleTree, Unity will ask you if you want to change the layer for just AppleTree or for both AppleTree and its children. You definitely want to choose Yes because you need the cylinder and sphere that make up the trunk and the leaves of the tree to also be in the AppleTree physics layer. This change will also trickle forward to the AppleTree instance in the scene. You can click AppleTree in the Hierarchy pane to confirm this.

Now if you press Play, you should see the apples dropping properly from the tree.

Stopping Apples If They Fall Too Far

If you leave the current version of the game running for a while, you'll notice that there are *a lot* of Apples in the Hierarchy. That's because the code is creating a new Apple every second but never deleting any Apples. Open the Apple C# script and add the following code to kill each Apple once it reaches a depth of `transform.position.y == -20` (which is comfortably off-screen). Here's the code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Apple : MonoBehaviour {
    public static float bottomY = -20f;

    void Update () {
        if ( transform.position.y < bottomY ) {
            Destroy( this.gameObject );
        }
    }
}
```

You will need to attach the Apple C# script to the Apple GameObject prefab in the Project pane for this code to function in the game. To do so, drag the script onto the prefab and release. Now, if you press Play in Unity and zoom out in the scene, you can see that Apples drop for a ways, but once they reach a y position of -20, they disappear.

The bolded `public static float` line declares and defines a static variable named `bottomY`. As was mentioned in [Chapter 25](#), static variables are shared by all instances of a class, so every instance of Apple will have the same value for `bottomY`. If `bottomY` is ever changed for one instance, it will simultaneously change for all instances. However, it's also important to point out that static fields like `bottomY` do *not* appear in the Inspector.

The `Destroy()` function removes things that are passed into it from the game, and it can be used to destroy both components and GameObjects. `Destroy(this.gameObject)` must be used in this case because `Destroy(this)` would just remove the Apple (Script) component from the Apple GameObject instance. In any script, `this` refers to the current instance of the C# class in which it is called (the Apple class in this instance), not to the entire GameObject. Any time you want to destroy an entire GameObject from within an attached component class, you must call `Destroy(this.gameObject)`.

This is all we need to do for the Apple GameObject.

Instantiating the Baskets

To make the Basket GameObjects work, we're going to introduce a concept that will recur throughout these prototype tutorials. While object-oriented thinking encourages us to create an independent class for each GameObject (as we have just done for AppleTree and Apple), it is often very useful to also have a script that runs the game as a whole. From the menu bar, choose *Assets > Create > C# Script* and name the script *ApplePicker*. Attach the ApplePicker script to the Main Camera in the Hierarchy. I often attach these game management scripts to the Main Camera because I am guaranteed that there is a Main Camera in every scene. Open the ApplePicker script in MonoDevelop and type the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class ApplePicker : MonoBehaviour {

    public GameObject basketPrefab;
    public int numBaskets = 3;
    public float basketBottomY = -14f;
    public float basketSpacingY = 2f;

    void Start () {
        for (int i=0; i<numBaskets; i++) {
            GameObject tBasketGO = Instantiate( basketPrefab ) as GameObject;
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
        }
    }
}
```

Click Main Camera in the Hierarchy pane and set the `basketPrefab` in the Inspector to be the Basket GameObject prefab that was made earlier, and then click Play. You'll see that this code creates three baskets at the bottom of the screen.

Moving the Baskets with the Mouse

Open the Basket C# script in MonoDevelop and enter this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Basket : MonoBehaviour {

    void Update () {
        // Get the current screen position of the mouse from Input
        Vector3 mousePos2D = Input.mousePosition; // 1

        // The Camera's z position sets the how far to push the mouse into 3D
        mousePos2D.z = -Camera.main.transform.position.z; // 2

        // Convert the point from 2D screen space into 3D game world space
        Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mousePos2D ); // 3

        // Move the x position of this Basket to the x position of the Mouse
        Vector3 pos = this.transform.position;
        pos.x = mousePos3D.x;
    }
}
```

```
        this.transform.position = pos;
    }
}
```

1. `Input.mousePosition` gets assigned to `mousePos2D`. This value is in screen coordinates, meaning that it measures how many pixels the mouse is from the top-left corner of the screen. The `z` position of `Input.mousePosition` will always start at 0 because it is essentially a two-dimensional measurement.
2. This line sets the `z` coordinate of `mousePos2D` to the negative of the Main Camera's `z` position. In our game, the Main Camera is at a `z` of -10, so `mousePos2D.z` is set to 10. This tells the upcoming `ScreenToWorldPoint` function how far to push the `mousePos3D` into the 3D space.
3. `ScreenToWorldPoint()` converts `mousePoint2D` into a point in 3D space inside the scene. If `mousePos2D.z` were 0, the resulting `mousePos3D` point would be at a `z` of -10 (the same as the Main Camera). By setting `mousePos2D.z` to 10, `mousePos3D` is projected into the 3D space 10 meters away from the Main Camera position, resulting in a `mousePos3D.z` of 0. This doesn't change the resultant `x` or `y` position in games with an orthographic camera projection, but it matters significantly in games with a perspective camera projection. If this is at all confusing, I recommend looking at `Camera.ScreenToWorldPoint` in the Unity Scripting Reference.

Now that the Baskets are moving, you can use them to collide with Apples, though the Apples aren't really being caught yet; instead, they're just landing on the Baskets.

Catching Apples

Add the following bold lines to the Basket C# script:

[Click here to view code image](#)

```
public class Basket : MonoBehaviour {

    void Update () {
        ...
    } // 1

    void OnCollisionEnter( Collision coll ) { // 2
        // Find out what hit this basket
        GameObject collidedWith = coll.gameObject; // 3
        if ( collidedWith.tag == "Apple" ) { // 4
            Destroy( collidedWith );
        }
    }
}
```

1. Throughout the tutorial chapters of this book, I use ellipses (. . .) to indicate parts of the code that I am skipping in the code listing. Without these, the code listings would be ridiculously long in some of the later chapters. When you see ellipses like these, you shouldn't change anything about the code where they are; just leave it alone and focus on the new code (which is bolded for clarity). This code listing requires no changes to the `Update()` function, so I have used ellipses to skip it.
2. The `OnCollisionEnter` method is called whenever another `GameObject` collides with this Basket, and a `Collision` argument is passed in with information about the collision,

including a reference to the `GameObject` that hit this Basket's Collider.

3. This line assigns this colliding `GameObject` to the temporary variable `collidedWith`.
4. Check to see whether `collidedWith` is an Apple by looking for the "Apple" tag that was assigned to all Apple `GameObjects`. If `collidedWith` is an Apple, it is destroyed. Now, if an Apple hits this Basket, it will be destroyed.

At this point, the game functions very similarly to our inspiration *Kaboom!*, though it doesn't yet have any *graphical user interface* (GUI) elements like a score or a representation of how many lives the player has remaining. However, even without these elements, Apple Picker would be a successful prototype in its current state. As is, this prototype will allow you to tweak several aspects of the game to give it the right level of difficulty.

Save your scene. Then click the `_Scene_0` in the Project pane to select it. Press Command-D on the keyboard (Control+D on Windows) to duplicate the scene. This will create a new scene named `_Scene_1`. Double-click `_Scene_1` to open it. As an exact duplicate of `_Scene_0`, the game in this new scene will work as well. This gives you a chance to tweak variables in the scene without changing any of them in `_Scene_0` because each scene will store different Inspector values for serialized public fields in C# script components. Try making the game more difficult by increasing the speed of the `AppleTree`, increasing the random chance of the `AppleTree` changing direction, dropping apples more frequently, and so on. After you have the game balanced for a harder difficulty level in `_Scene_1`, save it and reopen `_Scene_0`. If you're ever concerned about which scene you have open, just look at the title at the top of the Unity window. It will always include the scene name.

GUI and Game Management

The final things to add to our game are the GUI and *game management* that will make it feel like more of a real game. The GUI element we'll add is a score counter, and the game management elements we'll add are rounds and the ability for the game to restart when the player has run out of Baskets.

Score Counter

The score counter will help players get a sense of their level of achievement in the game.

Open `_Scene_0` by double-clicking it in the Project pane. Then go to the menu bar and choose `GameObject > Create Other > GUI Text`. This will place a `GUIText` in the middle of the screen with the words `Gui Text` in it. Rename `GUIText` to `ScoreCounter`. Try changing the x and y position of `ScoreCounter`. You'll notice that the coordinates for `GUITexts` differ completely from those for other `GameObjects`. This is because `GUITexts` are positioned relative to the screen rather than being positioned in world space. An x value of 0 is the far-left edge of the screen, and an x value of 1 is the right edge. A y value of 0 is the bottom of the screen, and a y value of 1 is the top. (Note that this also differs from the screen coordinates of `Input.mousePosition`, for which a y value of 0 is the top of the screen.)

Set the Transform and `GUIText` components of `ScoreCounter` as shown in the left half of [Figure 28.10](#).

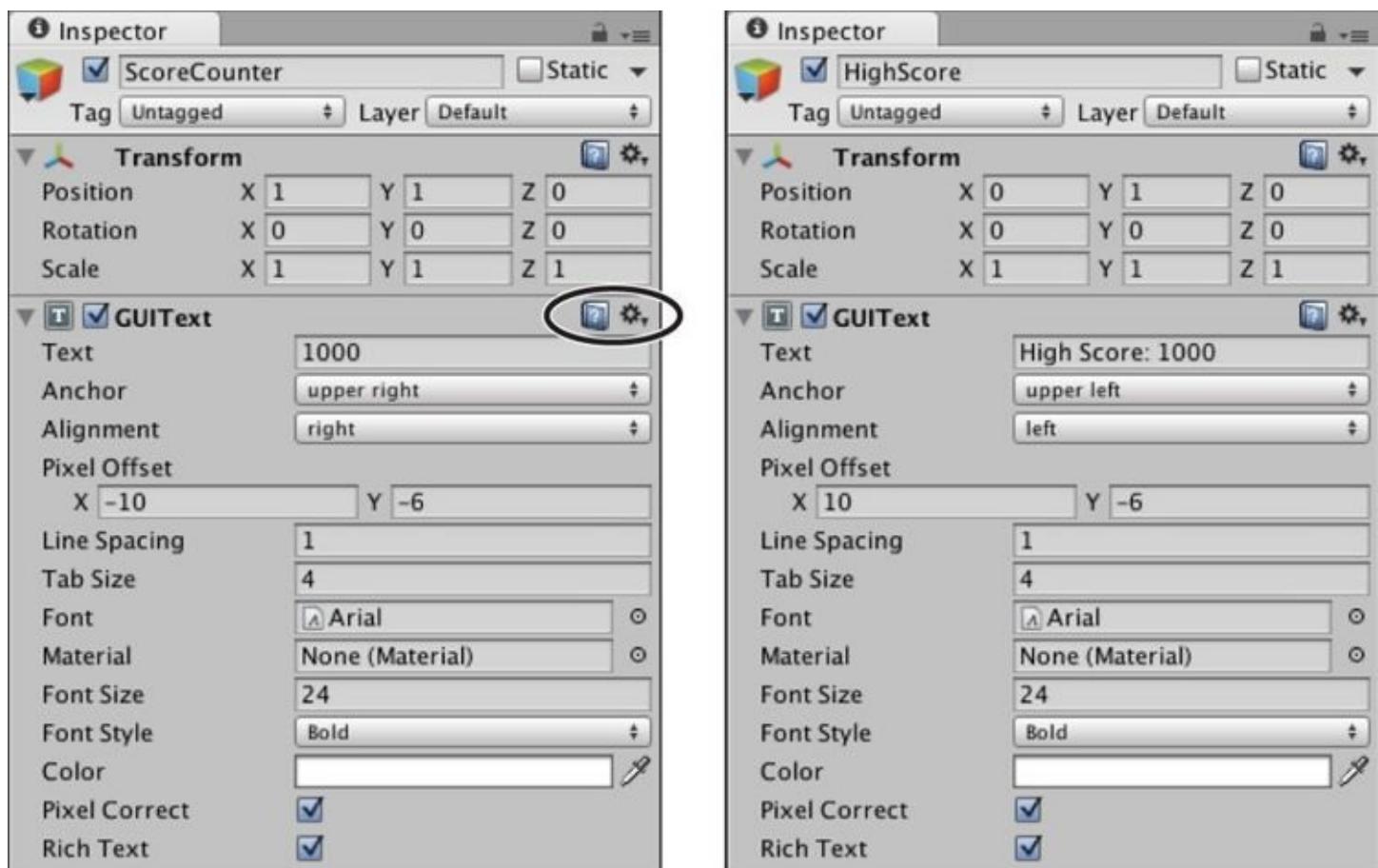


Figure 28.10 Transform and GUIText component settings for ScoreCounter and HighScore

For more information on the GUIText component, click the help icon in the top-right corner of the GUIText component (circled in black in [Figure 28.10](#)). You can use these help icons to learn more about any component.

Add Points for Each Caught Apple

There are two scripts that are notified when a collision occurs between an Apple and a Basket: the Apple and Basket scripts. In this game, there is already an `OnCollisionEnter()` method on the Basket C# script, so we'll modify this to give the player points for each Apple that is caught. 100 points per Apple seems like a reasonable number (though I've personally always thought it was a little ridiculous to have those extra zeroes at the end of scores). Open the Basket script in MonoDevelop and add the bolded lines shown here:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Basket : MonoBehaviour {

    public GUIText scoreGT; // 1

    void Update () {
        ...
    }

    void Start() {
        // Find a reference to the ScoreCounter GameObject
        GameObject scoreGO = GameObject.Find("ScoreCounter"); // 2
        // Get the GUIText Component of that GameObject
        scoreGT = scoreGO.GetComponent<GUIText>(); // 3
    }
}
```

```

    // Set the starting number of points to 0
    scoreGT.text = "0";
}

void OnCollisionEnter( Collision coll ) {
    // Find out what hit this Basket
    GameObject collidedWith = coll.gameObject;
    if ( collidedWith.tag == "Apple" ) {
        Destroy( collidedWith );
    }

    // Parse the text of the scoreGT into an int
    int score = int.Parse( scoreGT.text ); // 4
    // Add points for catching the apple
    score += 100;
    // Convert the score back to a string and display it
    scoreGT.text = score.ToString();
}
}

```

1. Be sure you don't neglect to enter this line. It's in an earlier part of the code than the others.
2. `GameObject.Find("ScoreCounter")` searches through all the `GameObjects` in the scene for one named "ScoreCounter" and assigns it to the local variable `scoreGO`.
3. `scoreGO.GetComponent<GUIText>()` searches for a `GUIText` component on the `scoreGO` `GameObject`, and this is assigned to the public field `scoreGT`. The starting score is then set to zero on the next line.
4. `int.Parse(scoreGT.text)` takes the text shown in `ScoreCounter` and converts it to an integer. 100 points are added to the `int score`, and it is then assigned back to the text of `scoreGT` after being parsed from an `int` to a `string` by `score.ToString()`.

Notifying Apple Picker That an Apple Was Dropped

Another aspect of making Apple Picker feel more like a game is ending the round and deleting a Basket if an Apple is dropped. At this point, Apples manage their own destruction, which is fine, but the Apple needs to somehow notify the ApplePicker script of this event so that Apple Picker can end the round and destroy the rest of the Apples. This will involve one script calling a function on another. Start by making these modifications to the Apple C# script:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;

public class Apple : MonoBehaviour {
    public static float      bottomY = -20f;

    void Update () {
        if ( transform.position.y < bottomY ) {
            Destroy( this.gameObject );
        }

        // Get a reference to the ApplePicker component of Main Camera
        ApplePicker apScript = Camera.main.GetComponent<ApplePicker>(); // 1
        // Call the public AppleDestroyed() method of apScript
        apScript.AppleDestroyed(); // 2
    }
}

```

1. Grabs a reference to the ApplePicker script component on the Main Camera. Because the Camera class has a built-in static variable Camera.main that references the Main Camera, it is not necessary to use GameObject.Find("Main Camera") to obtain a reference to Main Camera. GetComponent<ApplePicker>() is then used to grab a reference to the *ApplePicker (Script)* component on Main Camera and assign it to apScript. After this is done, it is possible to access public variables and methods of the ApplePicker instance that is attached to Main Camera.

2. This calls a non-existent AppleDestroyed() method of the ApplePicker instance.

There is currently no public AppleDestroyed() method in the ApplePicker script, so you will need to open the ApplePicker C# script in MonoDevelop and make the following bolded changes:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class ApplePicker : MonoBehaviour {

    public GameObject basketPrefab; // 1
    ...
    public float basketSpacingY = 2f;

    void Start () {
        ...
    }

    public void AppleDestroyed() { // 2
        /// Destroy all of the falling Apples
        GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple"); // 3
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }
    }
}
```

1. This is another way that ellipses (...) are used to shorten code listings. Here, lines have been omitted between the lines above and below the ellipses. Again, this is an indication that you don't need to modify any code between the lines.
2. The AppleDestroyed() method must be declared `public` for other classes (like Apple) to be able to call it. By default, methods are all *private* and unable to be called (or even seen) by other classes.
3. `GameObject.FindGameObjectsWithTag("Apple")` will return an array of all existing Apple GameObjects. The subsequent `foreach` loop iterates through each of these and destroys them.

Destroying a Basket When an Apple Is Dropped

The final bit of code for this scene will manage the deletion of one of the Baskets each time an Apple is dropped and stop the game when all the Baskets have been destroyed. Make the following changes to the ApplePicker C# script:

[Click here to view code image](#)

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

```

using System.Collections.Generic; // 1

public class ApplePicker : MonoBehaviour {
    ...
    public float basketSpacingY = 2f; // 2
    public List<GameObject> basketList;

    void Start () {
        basketList = new List<GameObject>();
        for (int i=0; i<numBaskets; i++) {
            GameObject tBasketGO = Instantiate( basketPrefab ) as GameObject;
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
            basketList.Add( tBasketGO ); // 3
        }
    }

    public void AppleDestroyed() {
        /// Destroy all of the falling Apples
        GameObject[] tAppleArray = GameObject.FindGameObjectsWithTag( "Apple" );
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }

        //// Destroy one of the Baskets
        // Get the index of the last Basket in basketList
        int basketIndex = basketList.Count-1;
        // Get a reference to that Basket GameObject
        GameObject tBasketGO = basketList[basketIndex];
        // Remove the Basket from the List and destroy the GameObject
        basketList.RemoveAt( basketIndex );
        Destroy( tBasketGO );
    }
}

```

1. We will be storing the Basket GameObjects in a List, so it is necessary to use the `System.Collections.Generic` code library. (For more information about Lists, see [Chapter 22, “Lists and Arrays.”](#)) The `public List<GameObject> basketList` is declared at the beginning of the class, and it is defined and initialized in the first line of `Start()`.
2. Here, the ellipses omit all the lines before `public float basketSpacingY = 2f;.`
3. A new line is added to the end of the for loop that Adds the baskets to `basketList`. The baskets are added in the order they are created, which means that they are added bottom to top.

In the method `AppleDestroyed()` a new section has been added to destroy one of the Baskets. Because the Baskets are added from bottom to top, it's important that the last Basket in the List is destroyed first (to destroy the Baskets top to bottom).

Adding a High Score

Create a new `GUIText` in the scene just as you did for the `ScoreCounter` and name it *HighScore*. Give its `Transform` and `GUIText` components the settings shown in the right side of [Figure 28.10](#).

Next, create a new C# script named *HighScore*, attach it to the `HighScore` GameObject in the Hierarchy pane, and give it the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class HighScore : MonoBehaviour {
    static public int score = 1000;

    void Update () {
        GUIText gt = this.GetComponent<GUIText>();
        gt.text = "High Score: "+score;
    }
}
```

The lines in `Update()` simply display the value of `score` in the `GUIText` component. It is not necessary to call `ToString()` on the `score` in this instance because when the `+` operator is used to concatenate a string with another data type (the `int score` in this case), `ToString()` is called implicitly (that is, automatically).

Making the `int score` not only public but also static gives us the ability to access it from any other script by simply typing `HighScore.score`. This is one of the powers of static variables that we will use throughout the prototypes in this book. Open the `Basket` C# script and add the following lines to see how this is used:

[Click here to view code image](#)

```
void OnCollisionEnter( Collision coll ) {
    ...
    // Convert the score back to a string and display it
    scoreGT.text = score.ToString();

    // Track the high score
    if (score > HighScore.score) {
        HighScore.score = score;
    }
}
```

Now `HighScore.score` is set any time the current score exceeds it.

Finally, open the `ApplePicker` C# script and add the following lines to reset the game whenever a player runs out of Baskets:

[Click here to view code image](#)

```
public void AppleDestroyed() {
    ...

    //// Destroy one of the Baskets
    ...
    basketList.RemoveAt( basketIndex );
    Destroy( tBasketGO );

    // Restart the game, which doesn't affect HighScore.Score
    if ( basketList.Count == 0 ) {
        Application.LoadLevel( "_Scene_0" );
    }
}
```

`Application.LoadLevel("_Scene_0")` will reload `_Scene_0`. This effectively resets the game to its beginning state. However, because `HighScore.score` is a static variable, it is not reset along with the rest of the game. This means that high scores will remain from one round to the next.

However, whenever you press the Play button again to stop the game, `HighScore.score` will reset. It is possible to fix this through the use of Unity's `PlayerPrefs`. `PlayerPrefs` store information from Unity scripts on the computer so that the information can be recalled later and isn't destroyed when playback stops. Add the following bolded changes to the `HighScore` C# script:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class HighScore : MonoBehaviour {
    static public int score = 1000;

    void Awake() { // 1
        // If the ApplePickerHighScore already exists, read it
        if (PlayerPrefs.HasKey("ApplePickerHighScore")) { // 2
            score = PlayerPrefs.GetInt("ApplePickerHighScore");
        }
        // Assign the high score to ApplePickerHighScore
        PlayerPrefs.SetInt("ApplePickerHighScore", score); // 3
    }

    void Update () {
        GUIText gt = this.GetComponent<GUIText>();
        gt.text = "High Score: "+score;
        // Update ApplePickerHighScore in PlayerPrefs if necessary
        if (score > PlayerPrefs.GetInt("ApplePickerHighScore")) { // 4
            PlayerPrefs.SetInt("ApplePickerHighScore", score);
        }
    }
}
```

1. `Awake()` is a built-in Unity method (like `Start()` or `Update()`) that happens when the instance of `HighScore` is first created (so `Awake()` always occurs before `Start()`).
2. `PlayerPrefs` is a dictionary of values that are referenced through keys (that is, unique strings). In this case, we're referencing the key `ApplePickerHighScore`. Here, the first line checks to see whether an `ApplePickerHighScore` already exists in `PlayerPrefs` and reads it in if it does exist.
3. The last line of `Awake()` assigns the current value of `score` to the `ApplePickerHighScore` `PlayerPrefs` key. If an `ApplePickerHighScore` already exists, this will rewrite the value back to `PlayerPrefs`; if the key does not already exist, however, this ensures that an `ApplePickerHighScore` key is created.
4. With the added lines, `Update()` now checks every frame to see whether the current `HighScore.score` is higher than the one stored in `PlayerPrefs` and updates `PlayerPrefs` if that is the case.

This usage of `PlayerPrefs` enables the Apple Picker high score to be remembered on this machine, and the high score will survive stopping playback, quitting Unity, and even restarting your computer.

Summary

Now you have a game prototype that plays very similarly to the original Activision game *Kaboom!* Although this game still lacks elements like steadily increasing difficulty and an opening and closing screen, these are things that you can add yourself once you gain more experience.

Next Steps

Here is some further explanation of the elements that you could add to the prototype in the future:

- **Start screen:** A start screen could be added. You could build it in its own scene and give it a splash image and a *Start* button. The *Start* button could then call `Application.LoadLevel("_Scene_0")` to start the game.
- **Game Over screen:** Another scene could be created as a Game Over screen. The Game Over screen could display the final score that the player achieved and could let the player know if she exceeded the previous high score. It should have a button labeled *Play Again* that calls `Application.LoadLevel("_Scene_0")`.
- **Increasing difficulty:** Varying difficulty levels are discussed in later prototypes, but if you wanted to add them here, it would make sense to store an array or List for each of the values on AppleTree, such as speed, chanceToChangeDirections, and secondsBetweenAppleDrops. Each element in the list could be a different level of difficulty, with the 0th element being the easiest and the last element being the most difficult. As the player played the game, a level counter could increase over time and be used as the index for these lists; so at level=0, the 0th element of each variable would be used.

Chapter 29. Prototype 2: Mission Demolition

Physics games are some of the most popular around, making games like *Angry Birds* household names. In this chapter, you make your own physics game that is inspired by *Angry Birds* and all the physics games, such as the board game *Crossbows and Catapults*, that came before it.

This chapter covers the following: physics, collision, mouse interaction, levels, and game state management.

Getting Started: Prototype 2

Because this is the second prototype and you now have some experience under your belt, this chapter is going to move a bit faster than the last on things that you already know. However, new topics will still be covered in detail.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup Procedure.”](#)

- **Project name:** Mission Demolition Prototype
- **Scene name:** _Scene_0
- **C# script names:** None yet

Game Prototype Concept

In this game, the player will be using a slingshot to fire projectiles at a castle, hoping to demolish it. Each castle will have a goal area that must be touched by the projectile to continue to the next level.

This is the sequence of events we want:

1. When the player’s mouse pointer is in range of the slingshot, the slingshot should glow.
2. If the player presses the left mouse button (numbered button 0 in Unity) down while the slingshot is glowing, a projectile will instantiate at the location of the mouse pointer.
3. As the player moves and drags the mouse around with the button held down, the projectile follows it, yet will remain within the limits of the Sphere Collider on the slingshot.
4. A white line will stretch from each arm of the slingshot around the projectile to make it look more like an actual slingshot.
5. When the player releases mouse button 0, the projectile will be fired from the slingshot.
6. A castle will be set up several meters away; the player’s goal will be to knock down the castle to hit a target area inside.
7. The player can fire as many shots as she likes to hit the goal. Each shot will leave a trail so that the player may better judge her next shot.

Many of these events relate to mechanics, but one is exclusively aesthetics: Step 4. All of the other elements that mention art use that art for the purpose of game mechanics, but Step 4 is just to make the game look nicer, so it’s less critical to the prototype. When you’re writing down your concepts for

games, this is an important thing to keep in mind. That isn't to say that you shouldn't implement things that are entirely aesthetics in a prototype; you just need to be aware of and prioritize the elements that will have the most direct impact on the game mechanics. For the sake of time and space, this prototype will focus on the other elements, and we'll leave the implementation of Step 4 for you to tackle later.

Art Assets

You should create several art assets now to prepare the project for coding.

Ground

To create the ground, follow these steps:

1. Open `_Scene_0` and create a cube (`GameObject > Create Other > Cube` from the menu bar). Rename the cube to `Ground`. To make a rectangular solid that is very wide in the x direction, set the transform of `Ground` to the following:
Ground (Cube) **P:[0,-10,0]** **R[0,0,0]** **S[100,1,1]**
2. Create a new material (`Assets > Create > Material`) and name it `Mat_Ground`. Give it a brown color and attach it to the `Ground` `GameObject` in the Hierarchy. (These actions are described in detail in the preceding chapter.)
3. Save your scene.

Directional Light

Next you want to drop a simple directional light into the scene, as follows:

1. From the menu bar, select `GameObject > Create Other > Directional Light`. One of the features of directional lights is that their position doesn't matter to the scene; only the rotation of a directional light is taken into consideration. That being the case, move it out of the way by setting its transform to the following:
Directional Light **P:[-10,0,0]** **R:[50,-30,0]** **S:[1,1,1]**
2. Save your scene.

Camera Settings

Camera settings are next:

1. Select the Main Camera in the Hierarchy and rename it to `_Main Camera` to keep it sorted at the top of the Hierarchy pane. Then set its transform to the following:
_Main Camera **P:[0,0,-10]** **R:[0,0,0]** **S:[1,1,1]**
2. Now set `_Main Camera`'s `Projection` to Orthographic and its `Size` to 10. Then choose a brighter background color to look more like a blue sky. The final settings should look like [Figure 29.1](#).

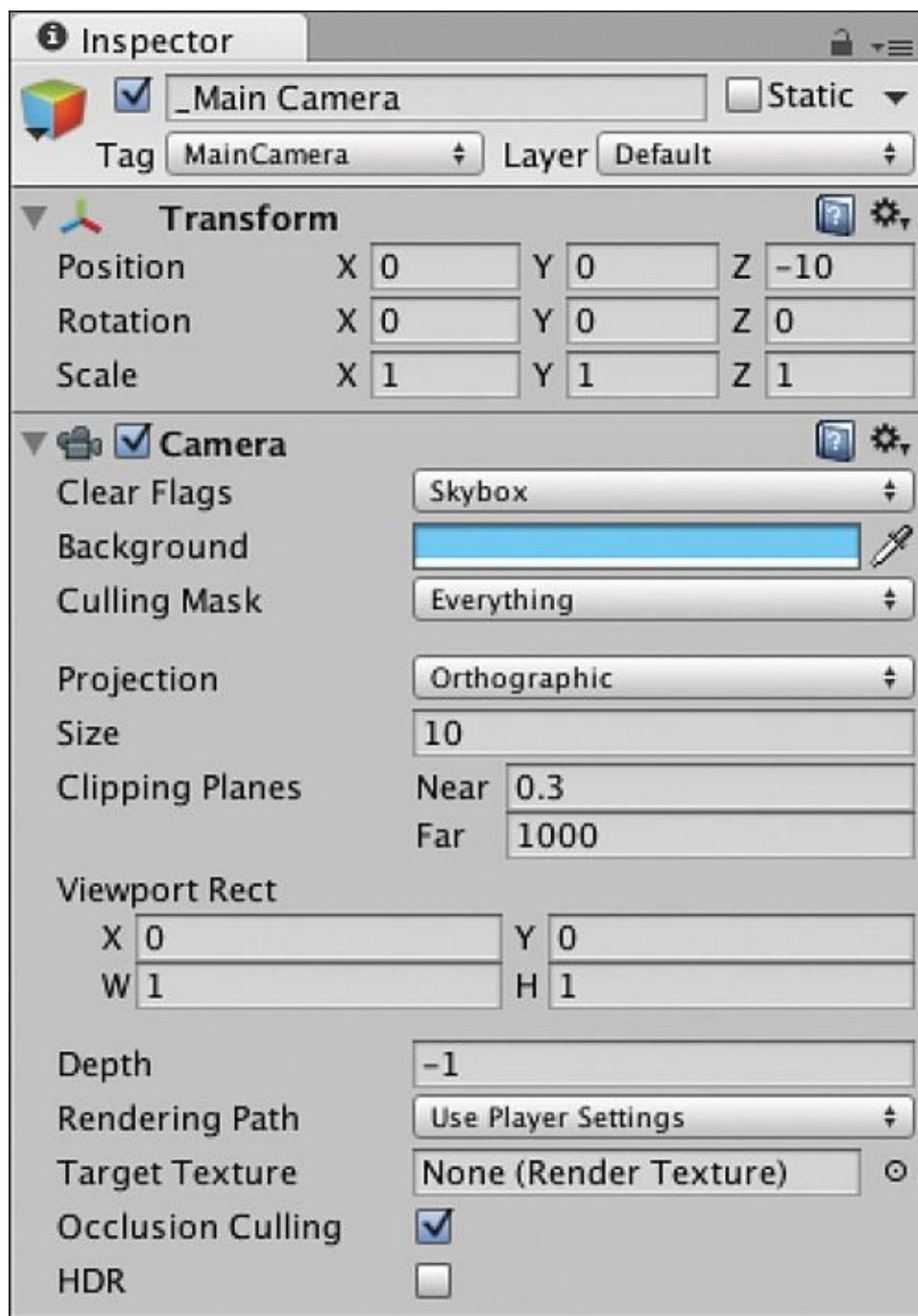


Figure 29.1 _Main Camera settings for Transform and Camera components

3. Though you have used orthographic cameras before, the meaning of the Size setting was never really discussed. In an orthographic projection, Size sets the distance from the center to the bottom of the camera view, so Size is half the height of what the camera is able to see. You can see that illustrated now in the Game pane. Ground is at a y position of -10, and it is perfectly bisected by the bottom of the Game window. Try changing the aspect ratio of the Game pane via the pop-up menu highlighted in [Figure 29.2](#). You will see that no matter what aspect ratio you select, the center of the Ground cube is still positioned perfectly at the bottom of the Game pane. After you've explored this for a while, choose an aspect ratio of 16:9, as shown the figure.

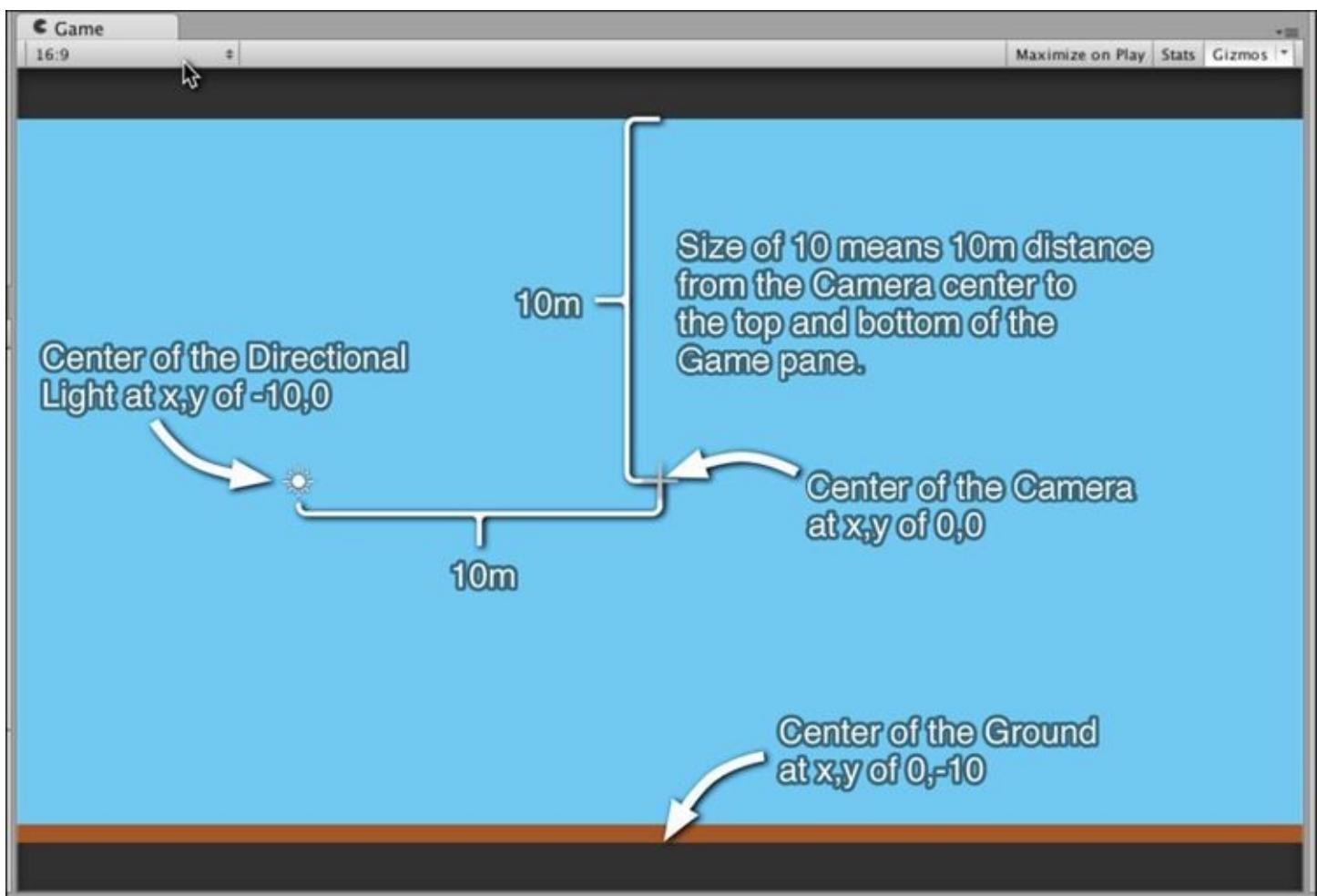


Figure 29.2 Demonstration of the meaning of an orthographic camera size of 10

4. Save your scene. Always save your scene.

The Slingshot

We'll make a simple slingshot out of three cylinders:

1. Start by creating an empty GameObject (*GameObject > Create Empty*). Change the GameObject's name to *Slingshot* and set its transform to the following:
Slingshot (Empty) **P:[0,0,0]** **R:[0,0,0]** **S:[1,1,1]**
2. Create a new cylinder (*GameObject > Create Other > Cylinder*) and change its name to *Base*. Drag it under Slingshot in the Hierarchy, making Slingshot its parent. Click the disclosure triangle next to Slingshot and select Base again. Set Base's transform to the following:
Base (Cylinder) **P:[0,1,0]** **R:[0,0,0]** **S:[0.5,1,0.5]**

When modifying the transform of a GameObject like Base that is the child of another GameObject, you're working in *local coordinates*, meaning that the position you're setting is the position of Base *relative to* its parent Slingshot, not the position of Base in global, world coordinates.

3. With Base selected, click the gear icon next to the Capsule Collider component in the Inspector and select *Remove Component* (see [Figure 29.3](#)). This will remove the Collider component from Base.

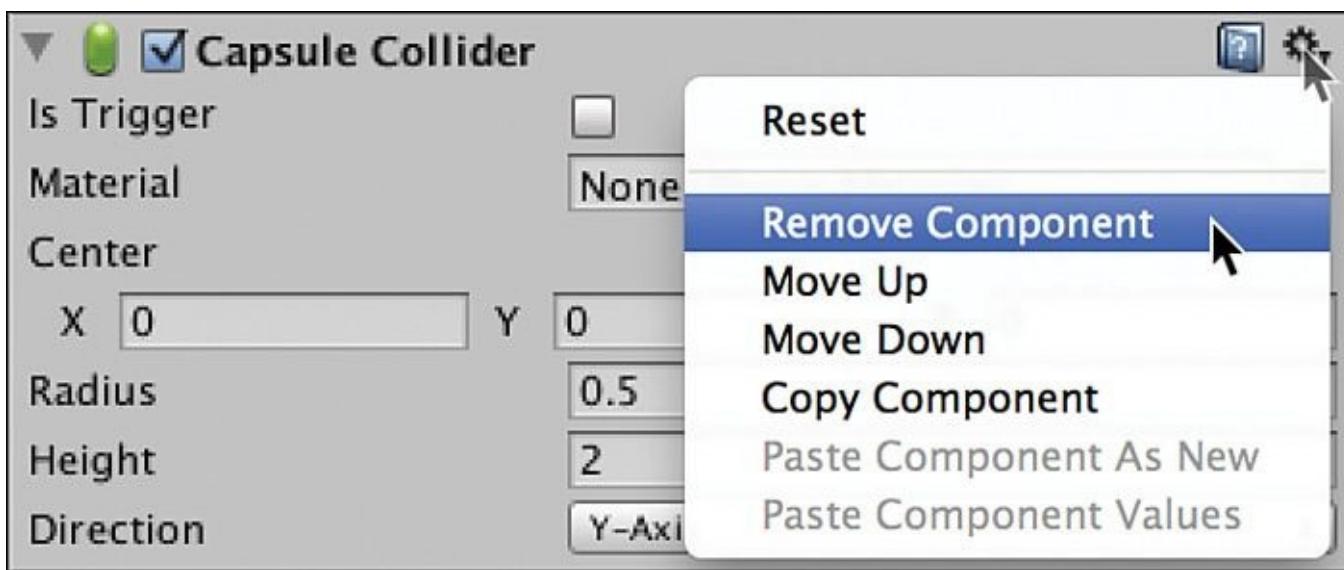


Figure 29.3 Removing the Collider Component

4. Create a new Material named *Mat_Slingshot* and color it a light yellow (or whatever color you want). Drag *Mat_Slingshot* onto *Base* to apply the material to the *GameObject*.
5. Select *Base* in the Hierarchy pane and duplicate it by pressing Command-D on your keyboard (Control+D on Windows machines). By duplicating, you ensure that the new duplicate is also a child of *Slingshot* and retains the *Mat_Slingshot* material. Change the name of the new duplicate from *Base* to *LeftArm*. Set the transform of *LeftArm* to the following:

LeftArm (Cylinder) P:[0,3,1] R:[45,0,0] S:[0.5,1.414,0.5]

This makes one of the arms of the slingshot.

6. Select *LeftArm* in the Hierarchy and duplicate it (Command-D). Rename this instance *RightArm*. Set the transform of *RightArm* to the following:

RightArm (Cylinder P:[0,3,-1] R:[-45,0,0] S:[0.5,1.414,0.5]

7. Select *Slingshot* in the Hierarchy. Add a Sphere Collider component to *Slingshot* (*Component > Physics > Sphere Collider*). Set the Sphere Collider component to the settings shown in [Figure 29.4](#).

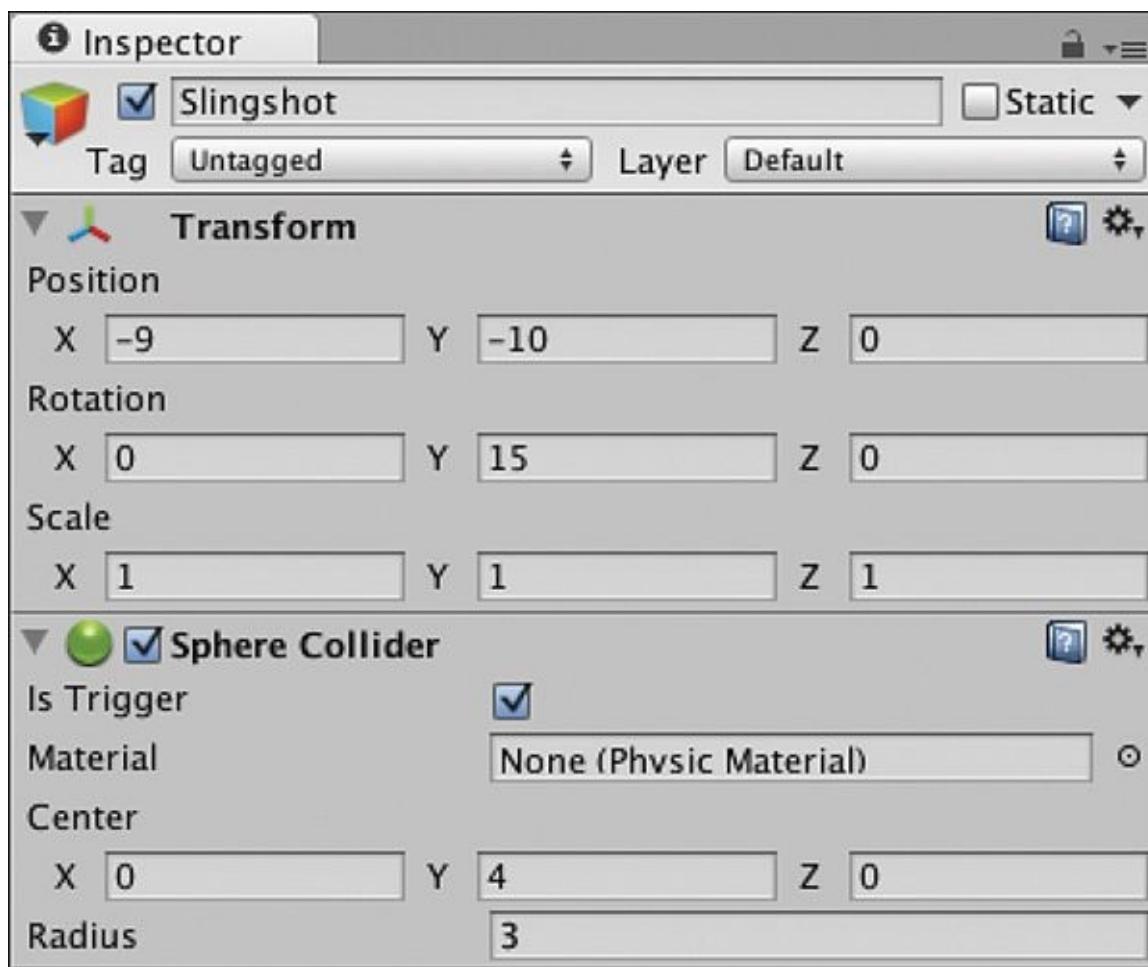


Figure 29.4 Settings for the Sphere Collider component of Slingshot

8. Make sure to set *Is Trigger* to true (checked). As you might expect, a collider with *Is Trigger* = *true* is known as a *trigger*. Triggers are part of the physics simulation in Unity and send notifications when other colliders or triggers pass through them. However, other objects don't bounce off of triggers as they do normal colliders. We'll use this large spherical trigger to handle the mouse interaction with Slingshot.
9. Set the transform of Slingshot to the following:

Slingshot (Empty) P:[-9,-10,0] R:[0,15,0] S:[1,1,1]

This will ground it on the left side of the screen, and the 15° y rotation will give it a bit of dimensionality, even through an orthographic camera.

10. Finally, you need to add a launch point to the slingshot that will give it a specific location from which to shoot the projectiles. Create an empty GameObject (*GameObject* > *Create Empty*) and name it *LaunchPoint*. Drag *LaunchPoint* onto *Slingshot* in the Hierarchy to make *Slingshot* its parent. Set the transform of *LaunchPoint* to the following:

LaunchPoint (Empty) P:[0,4,0] R:[0,-15,0] S:[1,1,1]

The -15° y rotation of *LaunchPoint* in local coordinates causes *LaunchPoint* to be aligned with the xyz axes in world coordinates. (That is, it removes the 15° rotation that was added to *Slingshot*.)

11. Save your scene.

Projectile

Next comes the projectile.

1. Create a sphere and name it *Projectile*. Select Projectile in the Hierarchy and attach a Rigidbody component (*Component > Physics > Rigidbody*). This Rigidbody component will allow the projectile to be physically simulated, similar to the Apples in Apple Picker.
2. Create a new material and name it *Mat_Projectile*. Make Mat_Projectile a dark gray color and apply it to Projectile.
3. Drag Projectile from the Hierarchy pane to the Project pane to make it a prefab. Then delete the Projectile instance that remains in the Hierarchy pane.

Your final Project and Hierarchy panes should look like those shown in [Figure 29.5](#).

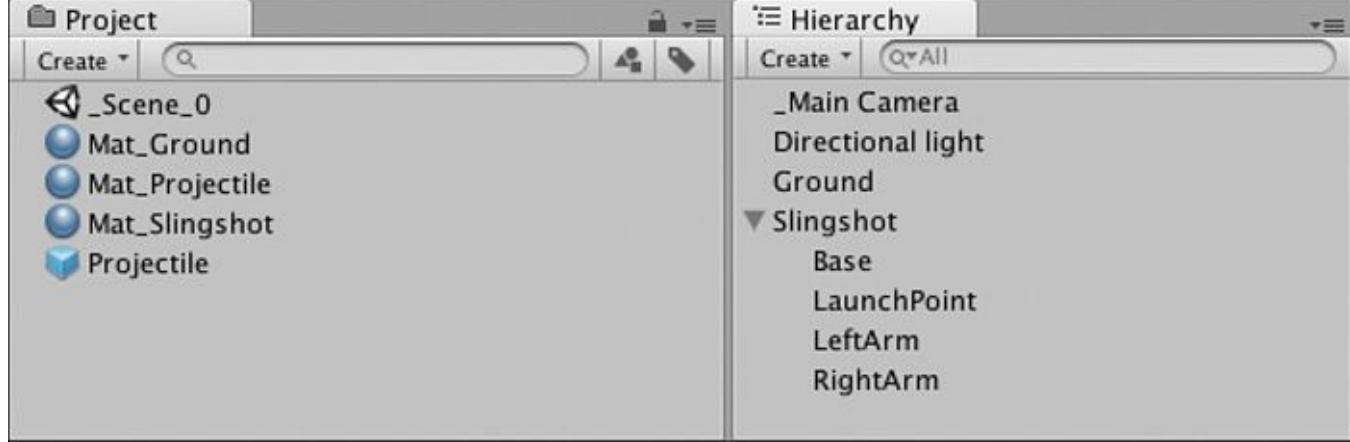


Figure 29.5 Project and Hierarchy panes prior to coding

4. Save your scene.

Coding the Prototype

With the art assets in place, it's time to start adding code to this project. The first script we'll add is one for Slingshot that will cause it to react to mouse input, instantiate a Projectile, and fire that Projectile. This script will be approached in an iterative manner with you adding only small sections of code at a time, testing the code, and then adding a little more. When you start creating your own scripts, this is a fantastic way to approach them: Implement something small and easy to code, test it, implement another small thing, repeat.

Slingshot

Follow these steps to create the Slingshot class:

1. Create a new C# script and name it *Slingshot* (*Assets > Create > C# Script*). Attach it to the Slingshot in the Hierarchy and open the Slingshot C# script in MonoDevelop. Enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Slingshot : MonoBehaviour {

    void OnMouseEnter() {
        print("Slingshot:OnMouseEnter()");
    }

    void OnMouseExit() {
        print("Slingshot:OnMouseExit()");
    }
}
```

}

2. Press Play, and pass the mouse pointer over Slingshot in the Game pane. You'll see that when the mouse enters the Sphere Collider of Slingshot, "Slingshot:OnMouseEnter()" is output to the Console pane. When the mouse exits the Sphere Collider, "Slingshot:OnMouseExit()" is output. `OnMouseEnter()` and `OnMouseExit()` are functions that work on any collider or trigger.

This is just the first step of the script we'll write to launch Projectiles, but it's important to start with small steps and build progressively.

3. Save your scene.

Showing When the Slingshot Is Active

Next, let's add a highlight to show the player that the slingshot is active:

1. Select LaunchPoint in the Hierarchy. Then add a Halo component to LaunchPoint (*Component* > *Effects* > *Halo*). Set the *Size* of the halo to 1 and make the *Color* a light gray to make sure that it's visible (my settings are [r:191,g:191,b:191,a:255]).
2. Now add the following code to the Slingshot C# script. As you can see, this is also a good time to comment out the `print()` statements from the last test:

[Click here to view code image](#)

```
public class Slingshot : MonoBehaviour {
    public GameObject launchPoint;

    void Awake() {
        Transform launchPointTrans = transform.Find("LaunchPoint");
        launchPoint = launchPointTrans.gameObject;
        launchPoint.SetActive( false );
    }

    void OnMouseEnter() {
        //print("Slingshot:OnMouseEnter()");
        launchPoint.SetActive( true );
    }

    void OnMouseExit() {
        //print("Slingshot:OnMouseExit()");
        launchPoint.SetActive( false );
    }
}
```

}

Now when you press Play, you'll see that the halo turns on and off, indicating that the player can interact with the slingshot.

The `SetActive()` method on `GameObjects` like `launchPoint` tells the game whether or not to ignore them. If a `GameObject` has `active` set to `false`, it will not render on screen, and it will not receive any calls to functions like `Update()` or `OnCollisionEnter()`. This does not destroy the `GameObject`; it just removes it from being an active part of the game. In the Inspector for a `GameObject`, the check box at the top of the Inspector just to the left of the `GameObject`'s name indicates whether the `GameObject` is active (see [Figure 29.6](#)).

Components have a similar check box. This sets whether a component is enabled. For most components (for example, `Renderer` and `Collider`) this can also be set in code (e.g., `Renderer.enabled = false`), but for some reason, `Halo` is not an accessible component

in Unity, meaning that we can't affect a Halo component from C#. Every once in a while, you will encounter an inconsistency like this, and you need to find a workaround. In this case, we can't disable the halo, so instead we deactivate the GameObject that contains it.

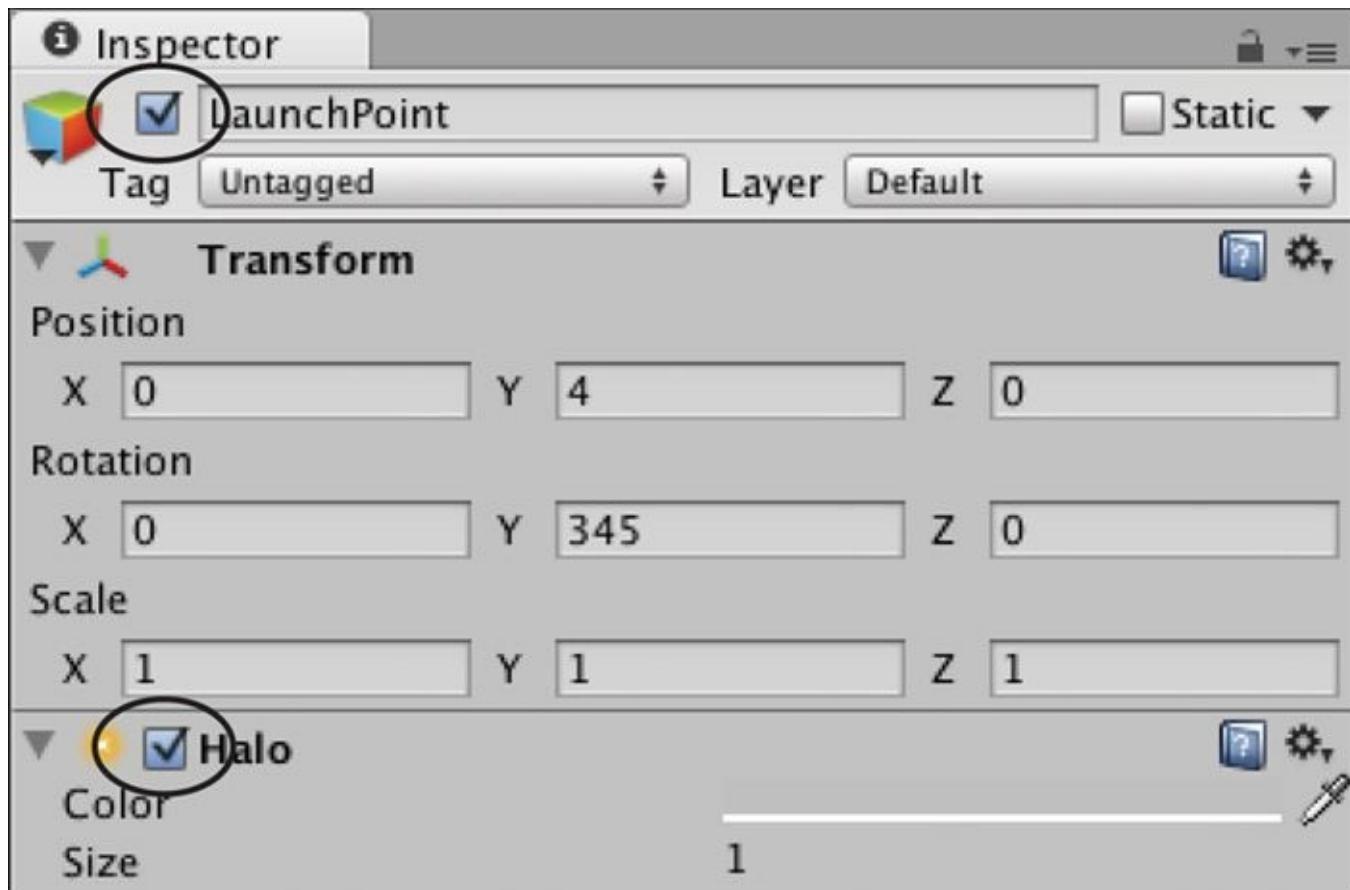


Figure 29.6 The GameObject active check box and the component enabled check box.

3. Save your scene.

Instantiating a Projectile

The next step is to instantiate the Projectile when mouse button 0 is pressed.

Warning

Don't Change OnMouseEnter() or OnMouseExit()!!! This was mentioned in the previous chapter, but is repeated here just in case.

In the following code listing for Slingshot, OnMouseEnter() and OnMouseExit() contain an ellipsis between braces: { ... }. As we write more and more complicated games, the scripts are going to get longer and longer. Whenever you see the name of a preexisting function followed by { ... }, this indicates that all the code from the previous listing is to remain unchanged between those braces. In this example, OnMouseEnter() and OnMouseExit() should still remain:

[Click here to view code image](#)

```
void OnMouseEnter() {  
    //print("Slingshot:OnMouseEnter()");  
    launchPoint.SetActive( true );  
}  
  
void OnMouseExit() {  
    //print("Slingshot:OnMouseExit()");  
    launchPoint.SetActive( false );  
}
```

Be sure to watch for these. Anywhere that you see ellipses in code, it means that I've used them to help shorten the code listings in this book and eliminate things you've already typed. { . . . } is **not** actual C# code.

1. Add the following code to Slingshot:

[Click here to view code image](#)

```
public class Slingshot : MonoBehaviour {  
    // fields set in the Unity Inspector pane  
    public GameObject      prefabProjectile;  
    public bool             _____;  
    // fields set dynamically  
    public GameObject      launchPoint;  
    public Vector3          launchPos;  
    public GameObject      projectile;  
    public bool             aimingMode;  
  
    void Awake() {  
        Transform launchPointTrans = transform.FindChild("LaunchPoint");  
        launchPoint = launchPointTrans.gameObject;  
        launchPoint.SetActive( false );  
        launchPos = launchPointTrans.position;  
    }  
  
    void OnMouseEnter() { . . . }    // Do not change OnMouseEnter()  
  
    void OnMouseExit() { . . . }     // Do not change OnMouseExit()  
  
void OnMouseDown() {  
    // The player has pressed the mouse button while over Slingshot  
    aimingMode = true;  
    // Instantiate a Projectile  
    projectile = Instantiate( prefabProjectile ) as GameObject;  
    // Start it at the launchPoint
```

```

        projectile.transform.position = launchPos;
        // Set it to isKinematic for now
        projectile.rigidbody.isKinematic = true;
    }
}

```

The first thing to note here are all the additional fields (that is, variables) at the top of the Slingshot class. One public bool has a name that looks particularly strange:

. This is a variable that is to be used for a very specific purpose: the division of the Slingshot script component in the Inspector into fields that are meant to be set in the Inspector and fields that will be set dynamically by your code once the game is running. In this example, `prefabProjectile` (a reference to the prefab for all the Projectiles) must be set in the Unity Inspector before running the game, whereas all the other variables are meant to be set dynamically. Because the Unity Inspector sorts serialized public fields by the order in which they're declared, the underline bool variable will appear in the Inspector as a divider between these preset and dynamic public variables.

The other new fields are much more self-explanatory: `launchPos` stores the 3D world position of `launchPoint`, and `projectile` is a reference to the new Projectile instance that is created. `mouseActive` is normally false, but is set to true when mouse button 0 has been pressed down over Slingshot. This is a state variable that lets the rest of the code know how to behave. In the next section, we'll be writing code for Slingshot's `Update()` that only runs when `mouseActive == true`.

In `Awake()`, we've added a single line to set `launchPos`.

The `OnMouseDown()` method contains the bulk of changes for this listing. `OnMouseDown()` will only be called when the player presses the mouse button down over the Collider component of the Slingshot GameObject, so this method can only be called if the mouse is in a valid start position. An instance of `prefabProjectile` is created and assigned to `projectile`. Then `projectile` is placed at the `launchPos` location. Finally, `isKinematic` on Projectile's Rigidbody is set to true. When a Rigidbody is kinematic, it is not moved automatically by physics but is still part of the simulation (meaning that a kinematic Rigidbody will not move as a result of a collision or gravity but can still cause other nonkinematic Rigidbodies to move).

2. Before you press Play, select Slingshot in the Hierarchy pane and set `prefabProjectile` to be the Projectile prefab in the Project pane (either by clicking the target to the right of `prefabProjectile` in the Inspector or by dragging the Projectile prefab from the Project pane onto the `prefabProjectile` in the Inspector).
3. Press Play, move your mouse pointer inside the active area for the Slingshot, and click. You will see the Projectile field instance appear.
4. Now let's make it do more. Add the following field and `Update()` method to the class Slingshot:

[Click here to view code image](#)

```

public GameObject      prefabProjectile;
public float          velocityMult = 4f;
public bool           mouseActive;
...
// As you see from the ellipses, some lines are skipped here
void Update() {

```

```

// If Slingshot is not in aimingMode, don't run this code
if (!aimingMode) return;
// Get the current mouse position in 2D screen coordinates
Vector3 mousePos2D = Input.mousePosition;
// Convert the mouse position to 3D world coordinates
mousePos2D.z = -Camera.main.transform.position.z;
Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mousePos2D );
// Find the delta from the launchPos to the mousePos3D
Vector3 mouseDelta = mousePos3D-launchPos;
// Limit mouseDelta to the radius of the Slingshot SphereCollider
float maxMagnitude = this.GetComponent<SphereCollider>().radius;
if (mouseDelta.magnitude > maxMagnitude) {
    mouseDelta.Normalize();
    mouseDelta *= maxMagnitude;
}
// Move the projectile to this new position
Vector3 projPos = launchPos + mouseDelta;
projectile.transform.position = projPos;

if ( Input.GetMouseButtonUp(0) ) {
    // The mouse has been released
    aimingMode = false;
    projectile.rigidbody.isKinematic = false;
    projectile.rigidbody.velocity = -mouseDelta * velocityMult;
    projectile = null;
}
}

```

Most of this is explained in the in-line comments; however, a little vector math bears closer examination (see [Figure 29.7](#)).

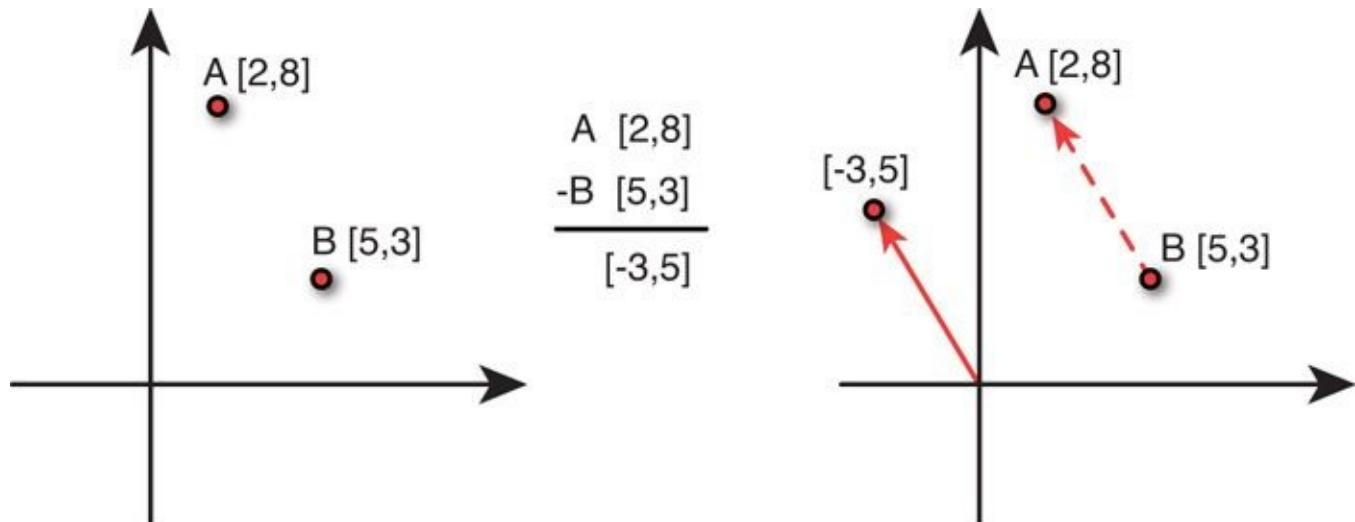


Figure 29.7 Two-dimensional vector subtraction

As you can see in [Figure 29.7](#), vectors are added and subtracted one component at a time. The figure is two dimensional, but the same methods work for 3D. The x components of the vectors A and B are subtracted as are the y components, making a new Vector2 defined as Vector2(2-5, 8-3), which becomes Vector2(-3, 5). The figure illustrates that A-B gives us the vector distance between A and B, which is also the distance and direction that one must move to get from point B to point A. A mnemonic to remember which one the vector will point at is AMBLAA, “A Minus B Looks At A.”

This is important in the `Update()` method because Projectile needs to be positioned along the vector from `launchPos` to the current `mousePos3D`, and this vector is named `mouseDelta`. However, the distance that the Projectile can move along `mouseDelta` is limited to

maxMagnitude, which is the radius of the SphereCollider on Slingshot (currently set to 3m in the Inspector for the Collider component).

If mouseDelta is longer than maxMagnitude, its magnitude is clamped to maxMagnitude. This is accomplished by first calling mouseDelta.Normalize() (which sets the length of mouseDelta to 1 but keeps it pointing in the same direction) and then multiplying mouseDelta by maxMagnitude.

projectile is moved to this new position, and if you play the game, you will see that the Projectile moves with the mouse but is limited to a specific radius.

Input.GetMouseButtonUp(0) will return true only on the *first frame* that the left mouse button has been released. That means that the if statement at the end of Update() is executed on the frame that the mouse button is released. On this frame, aimingMode is set to false. projectile's Rigidbody is set to nonkinematic, allowing it to once again respond to gravity. projectile is given a velocity that is proportional to the distance that it is from launchPos. Finally, projectile is set back to null. This doesn't delete the Projectile instance that was created, it just opens the field projectile to be filled by another instance when the slingshot is fired again.

5. Press Play and see how the Slingshot feels. Is the Projectile instance launching at a good velocity? Try adjusting velocityMult in the Inspector to see what value feels right to you. I ended up with a value of 10.

As it is now, the Projectile instance flies off screen very quickly. Let's make a follow camera to chase after the Projectile as it flies.

6. Save your scene.

Making a Follow Camera

We need _Main Camera to follow the Projectile when launched, but the behavior is a little more complicated than that. The full behavior should be as follows:

1. The camera sits at an initial position and doesn't move during Slingshot's aimingMode.
2. Once a Projectile is launched, the camera follows it (with a little easing to make it feel smoother).
3. As the camera moves up into the air, increase the Camera.orthographicSize to keep Ground in view.
4. When the Projectile comes to rest, the camera stops following it and returns to the initial position.

Follow these steps:

1. Start by creating a new C# script and naming it *FollowCam* (Assets > Create > C# Script). Drag the FollowCam Script onto _Main Camera in the Inspector to make it a component of _Main Camera.
2. Now double-click the FollowCam script to open it and input the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class FollowCam : MonoBehaviour {
```

```

static public FollowCam S; // a FollowCam Singleton

// fields set in the Unity Inspector pane
public bool _____;

// fields set dynamically
public GameObject poi; // The point of interest
public float camZ; // The desired Z pos of the camera

void Awake() {
    S = this;
    camZ = this.transform.position.z;
}

void Update () {
    // if there's only one line following an if, it doesn't need braces
    if (poi == null) return; // return if there is no poi

    // Get the position of the poi
    Vector3 destination = poi.transform.position;
    // Retain a destination.z of camZ
    destination.z = camZ;
    // Set the camera to the destination
    transform.position = destination;
}
}

```

The first thing you'll notice at the top of the code listing is the `FollowCam` singleton `S`. As covered in the “[Software Design Patterns](#)” section of [Appendix B](#), “[Useful Concepts](#),” a *singleton* is a design pattern that is used when there will be only one instance of a specific class in the game. Because there is only a single camera in *Mission Demolition*, it is a good candidate for a singleton. As a public static variable, the singleton `S` can be accessed anywhere in the code of the game as `FollowCam.S`, allowing us to set the public `poi` field from anywhere by setting `FollowCam.S.poi`.

The next thing that you'll notice is that there are no fields in this class that are meant to be set in the Inspector. That will change shortly.

The remaining two fields are `poi`, which holds the `GameObject` that the camera will follow, and `camZ`, which holds the initial z position of the camera.

In `Update()`, the camera is moved to the position of the `poi` except for the z coordinate, which is set to `camZ` every frame. (This prevents the camera from being so close to `poi` that `poi` becomes invisible.)

3. Open the `Slingshot` C# script and add the single bold line from the following code listing between the other two lines near the end of `Update()`:

[Click here to view code image](#)

```

projectile.rigidbody.velocity = -mouseDelta * velocityMult;
FollowCam.S.poi = projectile;
projectile = null;

```

This line uses the `FollowCam.S` singleton to set the value of `poi` for the camera. The new `poi` will be the newly fired Projectile. Try pressing Play and see how it looks.

You should notice a few issues:

A. If you zoom out the Scene pane view enough, you'll see that the Projectile actually flies past the

end of Ground.

- B. If you fire at Ground, you'll see that the Projectile neither bounces nor stops once it has hit Ground. If you pause right after firing, select the Projectile in the Hierarchy, and then unpause, you'll see that it rolls upon hitting Ground and never stops rolling.
- C. When the Projectile is first launched, the camera jumps to the position of the Projectile, which is visually jarring.
- D. Once the Projectile is at a certain height, all you see is sky, so it's difficult to tell how high up the Projectile is.

Each of these issues will be fixed in order by taking the following steps (which are generally ordered from the easiest fix to the most difficult).

1. First, fix issue A by setting the transform of Ground to P:[100,-10,0] R:[0,0,0] S:[400,1,1]. This will make Ground extend much farther to the right.
2. To fix issue B, both Rigidbody constraints and a *Physic Material* need to be added to Projectile. Select the Projectile prefab in the Project pane. In the Rigidbody component, click the disclosure triangle next to Constraints and check *Freeze Position z* and *Freeze Rotation x, y, & z*. *Freeze Position z* will keep the projectile from moving toward or away from the camera (basically keeping it in the same z depth as both the ground and the castle that will be added later). *Freeze Rotation x, y, & z* will keep it from rolling around.

While here, you should also set the pop-up menu for *Collision Detection* to Continuous. For information about the types of collision detection, click the help icon in the top-right corner of the Rigidbody component. In short, *continuous* collision detection takes more processor power than *discrete*, but it is more accurate for fast-moving objects like the Projectiles. Press Play and try launching a Projectile again.

3. These Rigidbody settings keep the Projectile from rolling endlessly, but it still doesn't feel right. You've spent your whole life experiencing physics, and that gives you an intuitive feel for the kinds of behaviors that feel like natural, real-world physics. This is true for your players as well, which means that even though physics is a complex system that requires a lot of math to model, if you make your game physics feel like the physics that players are used to, you won't have to explain that math to them.

Adding a Physic Material can make your physically simulated objects feel a lot more realistic. From the menu bar, choose *Assets > Create > Physic Material*. Name this Physic material *PMat_Projectile*. Click *PMat_Projectile* and set the *bounciness* to 1 in the Inspector. Then drag *PMat_Projectile* in the Project pane onto the Projectile prefab (also in the Project pane) to apply it to *Projectile.SphereCollider*. Selecting Projectile should reveal that *PMat_Projectile* has been assigned as the *material* of the Sphere Collider in the Inspector. Now when you press Play, you'll see that the projectile bounces to a stop instead of just gliding along the ground.

4. Issue C will be fixed via two means: easing through interpolation and limits on the camera's location. To start with easing, add the following bolded lines to *FollowCam*:

[Click here to view code image](#)

```
// fields set in the Unity Inspector pane
public float easing = 0.05f;
public bool _____;
...
Vector3 destination = poi.transform.position;
// Interpolate from the current Camera position toward destination
destination = Vector3.Lerp(transform.position, destination, easing);
```

```
// Retain a destination.z of camZ
```

The `Vector3.Lerp()` method interpolates between two points, returning a weighted average of the two. If `easing` is 0, `Lerp()` will return the first point (`transform.position`); if `easing` is 1, `Lerp()` will return the second point (`destination`). If `easing` is any value in between 0 and 1, it will return a point between the two (with an easing of 0.5 returning the midpoint between the two). By setting `easing = 0.05`, we are telling Unity to move the camera about 5% of the way from its current location to the location of the `poi` every frame. Because the `poi` is constantly moving, this gives us a nice smooth camera follow movement. Try playing with the value of `easing` to see how it affects the camera movement. This kind of use of `Lerp` is a very simplistic, non-time-based form of linear interpolation. For more information on linear interpolation, you can read about it in [Appendix B](#).

5. You might now notice a little glitching and jerkiness in the movement of the camera even with this easing. That is happening because the physics simulation is updating at a rate of 50fps while `Update()` is being called at the highest framerate possible. On a fast machine, this means that the camera will update many more times per second than the physics, causing the camera to catch up to the Projectile over several `Updates()` before the Projectile has a chance to move. To fix this, change the name of the `Update()` method to `FixedUpdate()`. Unlike an `Update()`, which happens every frame, a `FixedUpdate()` happens every frame of the physics simulation (or exactly 50fps) regardless of computer speed. Making this change should smooth out the jerkiness in `FollowCam`.

6. Now, add some limits to the `FollowCam` position:

[Click here to view code image](#)

```
// fields set in the Unity Inspector pane
public float easing = 0.05f;
public Vector2 minXY;
public bool _____;
...
Vector3 destination = poi.transform.position;
// Limit the X & Y to minimum values
destination.x = Mathf.Max( minXY.x, destination.x );
destination.y = Mathf.Max( minXY.y, destination.y );
// Interpolate from the current Camera position toward destination
```

The default value of `Vector2 minXY` is `[0,0]`, which works perfectly for our needs. The `Mathf.Max()` chooses the maximum value of the two passed in. When the projectile is initially launched, its x and y coordinates are both negative, so the `Mathf.Max()` ensures that the camera never moves into negative territory along either the x or y axes. This also keeps the camera from dipping below the ground when the projectile lands.

7. Issue D is fixed by dynamically adjusting the `orthographicSize` of the camera. Add the following bolded lines to the `FollowCam` script:

[Click here to view code image](#)

```
transform.position = destination;
// Set the orthographicSize of the Camera to keep Ground in view
this.camera.orthographicSize = destination.y + 10;
}
```

This works because we know that the `destination.y` will never be allowed to be less than 0. So, the minimum `orthographicSize` is 10, and the camera's `orthographicSize` will

expand as needed to always keep the ground in view. Double-click Ground in the Hierarchy to zoom out and show the whole thing in the Scene pane. Then select _Main Camera, press Play, and launch a Projectile. In the Scene pane, you will see the field of view of the camera expand smoothly as the Projectile flies.

8. Save your scene.

Vection and a Sense of Speed

The FollowCam moves pretty well now, but it's still difficult to tell how fast the Projectile is moving, especially when it's high in the air. To fix this issue, we're going to take advantage of the concept of *vection*. Vection is the sensation of movement that you get from seeing other things passing by quickly, and it is the concept that led to *parallax scrolling* in 2D video games. Parallax scrolling causes foreground objects to pass by quickly while background objects move more slowly relative to the movement of the main camera in a 2D game. While a full parallax system is beyond the scope of this tutorial, it is possible to at least get a simple feeling of vection by creating a lot of clouds and distributing them randomly through the sky. As the Projectile passes by them, the player will have more of a feeling of movement.

Making Cloud Art

To make this work, you're going to need to make some simple clouds:

1. Start by creating a new sphere (*GameObject > Create Other > Sphere*). Hover your mouse over the name of the Sphere Collider component in the Inspector for the sphere. Right-click and choose *Remove Component* from the pop-up menu. Set the Transform.Position of the Sphere to [0,0,0] so that it is visible in the Game pane as well as the Scene pane. Rename Sphere to *CloudSphere*.
2. Create a new material and name it *Mat_Cloud* (*Assets > Create > Material*). Drag *Mat_Cloud* onto *CloudSphere* and then select *Mat_Cloud* in the Project pane. From the pop-up menu next to *Shader* in the Inspector, choose *Self-Illumin > Diffuse*. This shader is self-illuminating (it generates its own light), and it also responds to the directional light in the scene. Click the color swatch next to *Main Color* in the Inspector for *Mat_Cloud* and set it to a gray of 65% (or RGBA of [166,166,166,255] in the Unity color picker). This should give *CloudSphere* just a little gray on the bottom-left side in the Game pane, which looks a bit like a cloud in the sun.
3. Drag *CloudSphere* to the Project pane to make it a prefab, and then drag several instances of *CloudSphere* into the Hierarchy pane and position and scale them to make an object that looks a bit like a cloud. Create an empty GameObject (*GameObject > Create Empty*), rename it *Cloud_0*, and set its transform to P:[0,0,0], R:[0,0,0], S:[1,1,1]. Make *Cloud_0* the parent of the *CloudSphere* group (by dragging them all underneath it in the Hierarchy). Try to make sure that the *CloudSpheres* are grouped around the *Cloud_0* parent. Each *CloudSphere*'s position in the Inspector should be between -2 and 2 in each dimension (x, y, & z). Drag *Cloud_0* to the Project pane to make a prefab. Then delete the remaining *Cloud_0* instance from the Hierarchy.
4. Click on *Cloud_0* in the Project pane and duplicate it by pressing Command-D (Control+D on Windows). The duplicate will automatically be named *Cloud_1*. Drag *Cloud_1* into the Hierarchy to place an instance of it into the Scene pane. Choose each of the *CloudSpheres* in *Cloud_1* and scale and reposition them as you wish to make a different-looking cloud. When you're satisfied, select *Cloud_1* in the Hierarchy, click the *Apply* button (to the right of the word *Prefab* at the top of the Inspector for *Cloud_1*). This will apply the changes you made back to

the Cloud_1 prefab.

5. Repeat this duplication, modification, and application process to create a total of five different clouds named Cloud_0 through Cloud_4. You can make as many clouds as you would like, but five will be sufficient for our purposes. When finished, if you add an instance of each cloud to the Hierarchy, it should look something like [Figure 29.8](#).

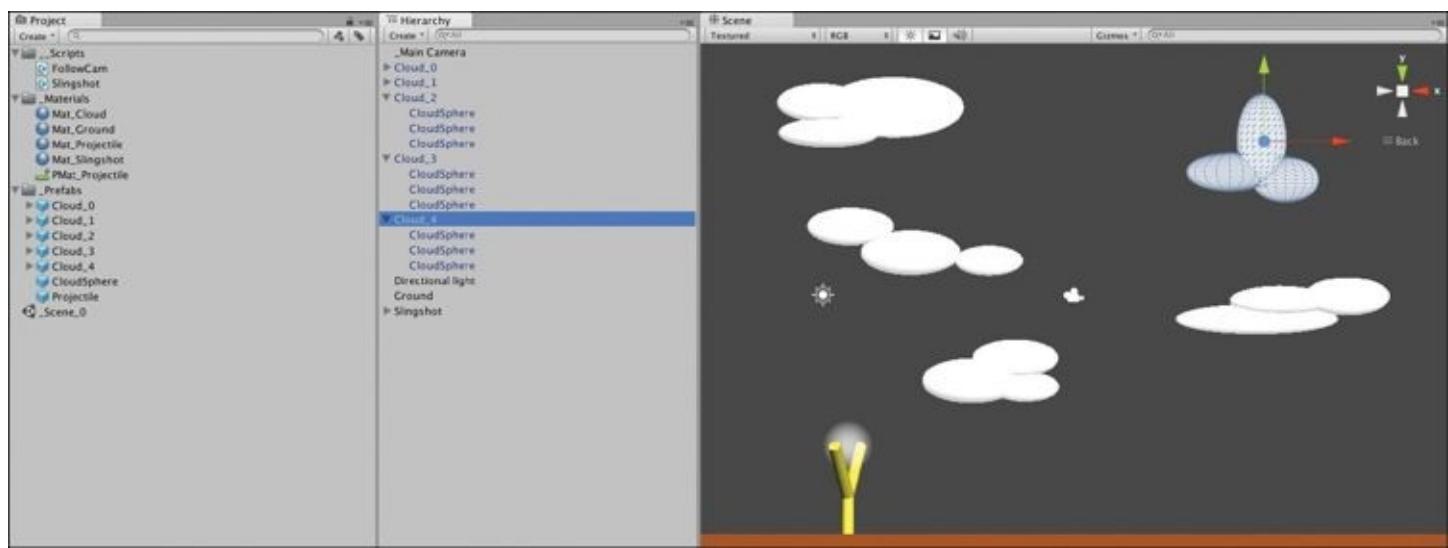


Figure 29.8 An example image of the hierarchical Cloud_#s composed of CloudSpheres

6. You can also see in [Figure 29.8](#) that I've added folders to organize my Project pane. I usually do this at the very beginning of a project but have waited this time so that you could experience how much better the project feels once it's organized. Create a folder by choosing *Assets > Create > Folder* from the menu bar. The folders I've created are named *_Scripts*, *_Materials*, and *_Prefabs*. The underscores in their names help sort them above any nonfoldered assets, and the double-underscore of the *_Scripts* folder ensures that it is the top folder in the Project pane. After you have created these folders, drag the proper assets into them in the Project pane. This will simultaneously create folders on your hard drive inside the *Assets* folder for your project, so not only your Project pane but also your *Assets* folder will be organized.

7. Now, delete the instances of Cloud_#s in the Hierarchy pane. Create a new empty GameObject named *CloudAnchor* (*GameObject > Create Empty*). This will give us a GameObject to act as the parent for all Cloud_#s, which will keep the Hierarchy tidy while the game is running.

8. Create a new C# script titled *CloudCrafter*. Drag *CloudCrafter* into the *_Scripts* folder and also drag it onto *_Main Camera*. This will add a second Script component to *_Main Camera*, which is perfectly fine in Unity so long as the two scripts don't conflict with each other (e.g., so long as they don't both try to set the position of the GameObject each frame). Because *FollowCam* is moving the camera, and *CloudCrafter* will just be placing Cloud_#s in the air, they shouldn't conflict at all. Enter the following code into *CloudCrafter*:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class CloudCrafter : MonoBehaviour {
    // fields set in the Unity Inspector pane
    public int numClouds = 40;          // The # of clouds to make
    public GameObject[] cloudPrefabs;    // The prefabs for the clouds
    public Vector3    cloudPosMin;      // Min position of each cloud
    public Vector3    cloudPosMax;      // Max position of each cloud
    public float      cloudScaleMin = 1; // Min scale of each cloud
```

```

public float          cloudScaleMax = 5;      // Max scale of each cloud
public float          cloudSpeedMult = 0.5f; // Adjusts speed of clouds

public bool           _____;

// fields set dynamically
public GameObject[]  cloudInstances;

void Awake() {
    // Make an array large enough to hold all the Cloud_ instances
    cloudInstances = new GameObject[numClouds];
    // Find the CloudAnchor parent GameObject
    GameObject anchor = GameObject.Find("CloudAnchor");
    // Iterate through and make Cloud_s
    GameObject cloud;
    for (int i=0; i<numClouds; i++) {
        // Pick an int between 0 and cloudPrefabs.Length-1
        // Random.Range will not ever pick as high as the top number
        int prefabNum = Random.Range(0,cloudPrefabs.Length);
        // Make an instance
        cloud = Instantiate( cloudPrefabs[prefabNum] ) as GameObject;
        // Position cloud
        Vector3 cPos = Vector3.zero;
        cPos.x = Random.Range( cloudPosMin.x, cloudPosMax.x );
        cPos.y = Random.Range( cloudPosMin.y, cloudPosMax.y );
        // Scale cloud
        float scaleU = Random.value;
        float scaleVal = Mathf.Lerp( cloudScaleMin, cloudScaleMax, scaleU );
        // Smaller clouds (with smaller scaleU) should be nearer the ground
        cPos.y = Mathf.Lerp( cloudPosMin.y, cPos.y, scaleU );
        // Smaller clouds should be further away
        cPos.z = 100 - 90*scaleU;
        // Apply these transforms to the cloud
        cloud.transform.position = cPos;
        cloud.transform.localScale = Vector3.one * scaleVal;
        // Make cloud a child of the anchor
        cloud.transform.parent = anchor.transform;
        // Add the cloud to cloudInstances
        cloudInstances[i] = cloud;
    }
}

void Update() {
    // Iterate over each cloud that was created
    foreach (GameObject cloud in cloudInstances) {
        // Get the cloud scale and position
        float scaleVal = cloud.transform.localScale.x;
        Vector3 cPos = cloud.transform.position;
        // Move larger clouds faster
        cPos.x -= scaleVal * Time.deltaTime * cloudSpeedMult;
        // If a cloud has moved too far to the left...
        if (cPos.x <= cloudPosMin.x) {
            // Move it to the far right
            cPos.x = cloudPosMax.x;
        }
        // Apply the new position to cloud
        cloud.transform.position = cPos;
    }
}
}

```

There are several fields here that must be set in the Inspector. They should be set to the values shown in [Figure 29.9](#). To set `cloudPrefabs`, click the disclosure triangle next to the variable

name *Cloud Prefabs* and enter 5 for the *Size*. Then drag each numbered Cloud_ prefab from the Project pane into the five *Element* slots of *cloudPrefabs*.

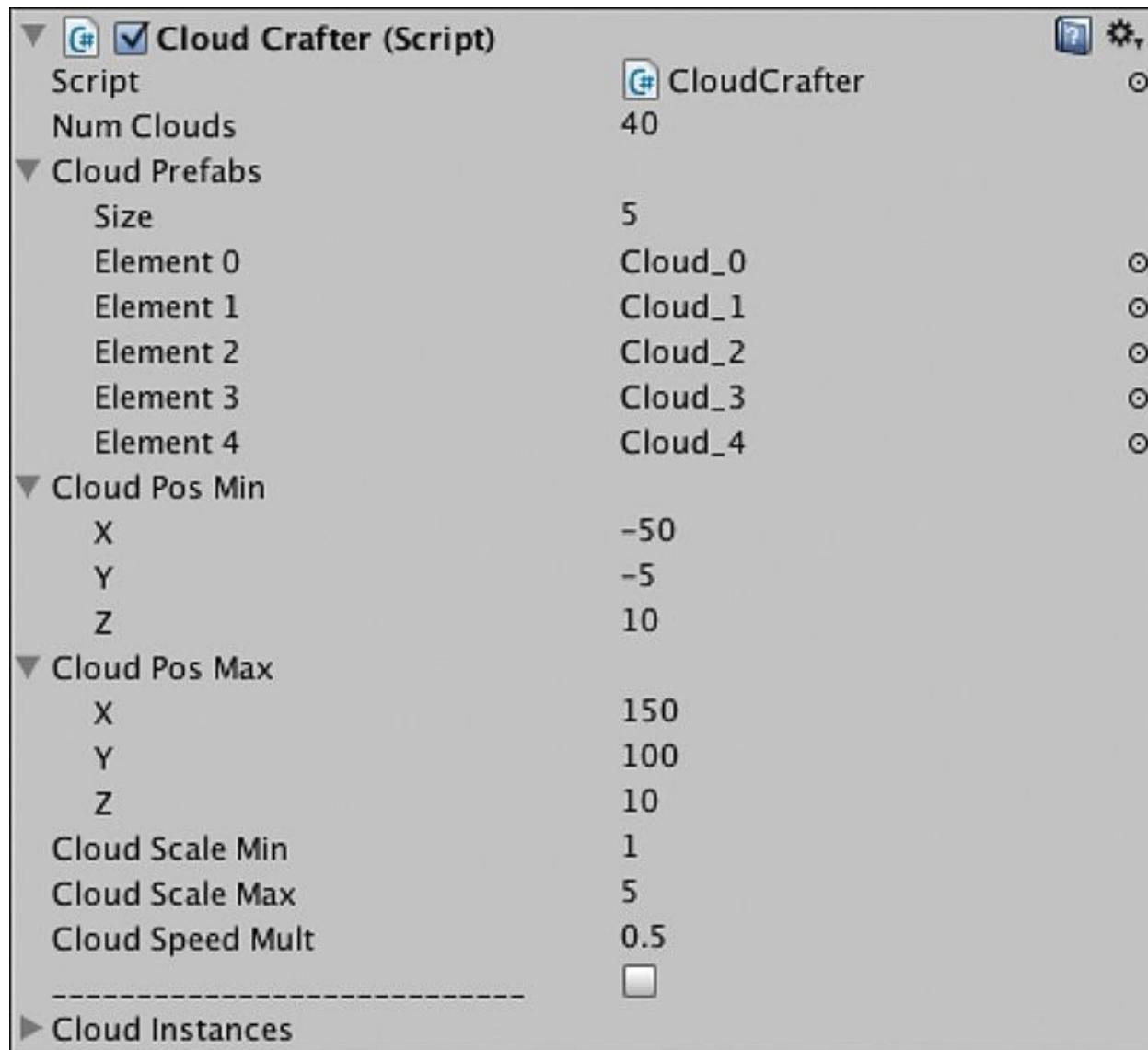


Figure 29.9 The settings for the CloudCrafter Script component

9. Save your scene.

In the CloudCrafter class, the `Awake()` method creates all the clouds and positions them. The `Update()` method moves each cloud a little to the left every frame. When a cloud moves to the left past `cloudPosMin.x`, it is moved to `cloudPosMax.x` on the far right. Zoom out in the Scene pane and watch the clouds blow by. Now when you launch a Projectile, thevection of the clouds passing by should make it feel much more like the projectile is actually moving.

Building the Castle

Mission Demolition needs something to demolish, so let's build a castle to serve that purpose. [Figure 29.11](#) shows what the final castle should look like.

1. Adjust the Scene pane so that you are viewing the scene from the back in isometric view by clicking the arrow on the axes gizmo opposite the z-axis (see [Figure 29.10](#)).

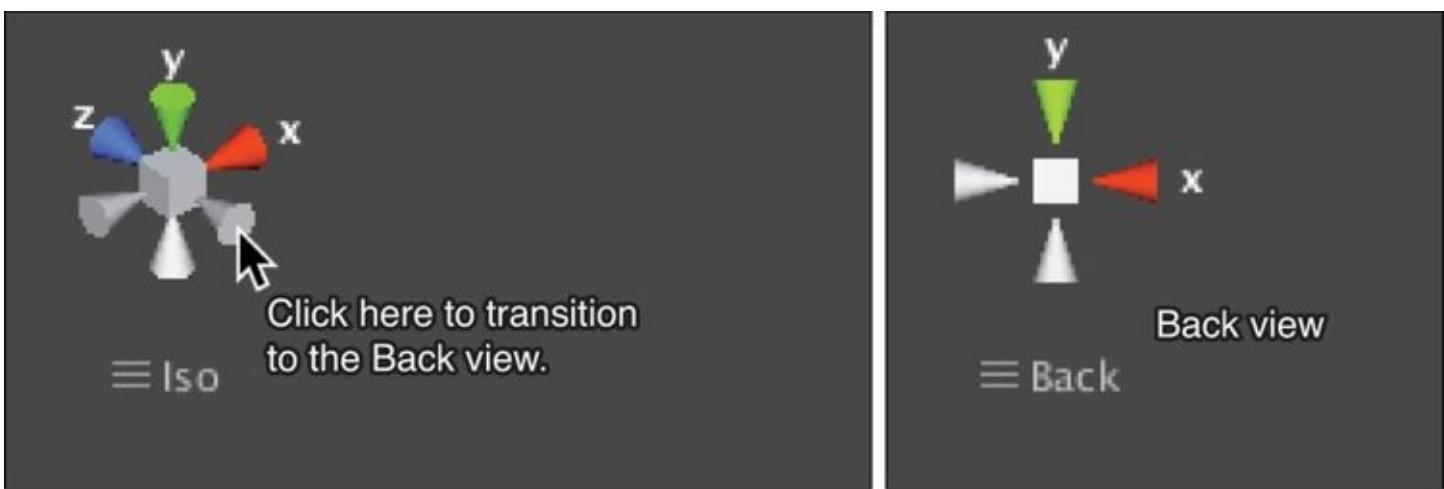


Figure 29.10 Selecting the Back view

2. Now, double click `_Main Camera` in the Hierarchy to zoom the Scene pane to a good view from which to build the castle.
3. Create an empty `GameObject` to be the root node of the castle (`GameObject > Create Empty`). Name it *Castle* and set its transform to `P:[0,-9.5,0] R:[0,0,0] S:[1,1,1]`. This will position it well for construction and put its base resting exactly on top of *Ground*.
4. Make the vertical walls of the castle:
 - 4.1. Create a new cube (`GameObject > Create Other > Cube`) and rename it *Wall_Stone*.
 - 4.2. Drag it onto *Castle* in the Hierarchy to make it a child of *Castle*.
 - 4.3. Add a `Rigidbody` component to *Wall_Stone* (`Component > Physics > Rigidbody`). Use the Inspector to constrain the `z` position of the *Wall_Stone* by setting `Rigidbody.FreezePosition.z` to true. Set the `Rigidbody.mass` to 4.
 - 4.4. Set the *Wall_Stone* Transform to `P:[-2,2,0] R:[0,0,0] S:[1,4,1]`.
 - 4.5. Drag *Wall_Stone* to the Project pane to make it a prefab (be sure to put it in the `_Prefabs` folder).
 - 4.6. Make three duplicates of *Wall_Stone* in the Hierarchy and set their `x` positions to -6, 2, and 6. This will form the 4 vertical walls of the first floor of the castle.
5. Make the horizontal walls that form the ceiling of the first floor.
 - 5.1. Create another cube and name it *Wall_Stone_H* (for “Horizontal”).
 - 5.2. Make *Wall_Stone_H* a child of *Castle* and set its transform to `P:[0,4.25,0] R:[0,0,0] S:[4,0.5,1]`.
 - 5.3. Add a `Rigidbody` component to *Wall_Stone_H* (`Component > Physics > Rigidbody`). Constrain the `z` position of the *Wall_Stone_H* by setting `Rigidbody.FreezePosition.z` to true. Set the `Rigidbody.mass` to 4.
 - 5.4. Make *Wall_Stone_H* a prefab and place it in the `_Prefabs` folder.
 - 5.5. Make two duplicates of *Wall_Stone_H* in the Hierarchy with `x` positions of -4 and 4.
6. To make the second floor of the castle, use your mouse to select three adjacent *Wall_Stones* of the first floor and the two *Wall_Stone_Hs* above them. Duplicate them (`Command-D` or `Control+D`) and move them to be resting above the others. You will need to tweak their positions, and the final positions for the new Walls should be as follows:

Wall_Stone	P:[-4,6.5,0]
Wall_Stone	P:[0,6.5,0]
Wall_Stone	P:[4,6.5,0]
Wall_Stone_H	P:[-2,8.75,0]
Wall_Stone_H	P:[2,8.75,0]

7. Continue the duplication trick to make the third floor by adding three more vertical walls and one more horizontal wall:

Wall_Stone	P:[-2,11,0]
Wall_Stone	P:[2,11,0]
Wall_Stone	P:[0,15.5,0]
Wall_Stone_H	P:[0,13.25,0]

8. The final GameObject to add to Castle is a goal for the player to hit with the Projectile.

8.1. Create a cube, name it *Goal*, make it a child of Castle, and set its transform to P:[0,2,0] R: [0,0,0] S:[3,4,4].

8.2. Create a new material named *Mat_Goal*. Drag *Mat_Goal* onto *Goal* to apply it. Select *Mat_Goal* in the Project pane and choose the *Transparent > Diffuse* shader. Then set the color to a bright green with an opacity of 25% (an RGBA in the Unity color picker of [0,255,0,64]).

8.3. Select *Goal* in the Hierarchy and set *BoxCollider.isTrigger* to true.

8.4. Drag *Goal* into the *_Prefabs* folder in the Project pane to make it a prefab.

9. One of the major advantages of building a castle out of prefabs like this is that we can easily make changes to every *Wall_Stone_H* if we want. Select the *Wall_Stone_H* prefab in the Project pane and set its *scale.x* to 3.5 in the Transform Inspector. Every *Wall_Stone_H* in your castle should reflect this change. Your finished castle should look like [Figure 29.11](#).

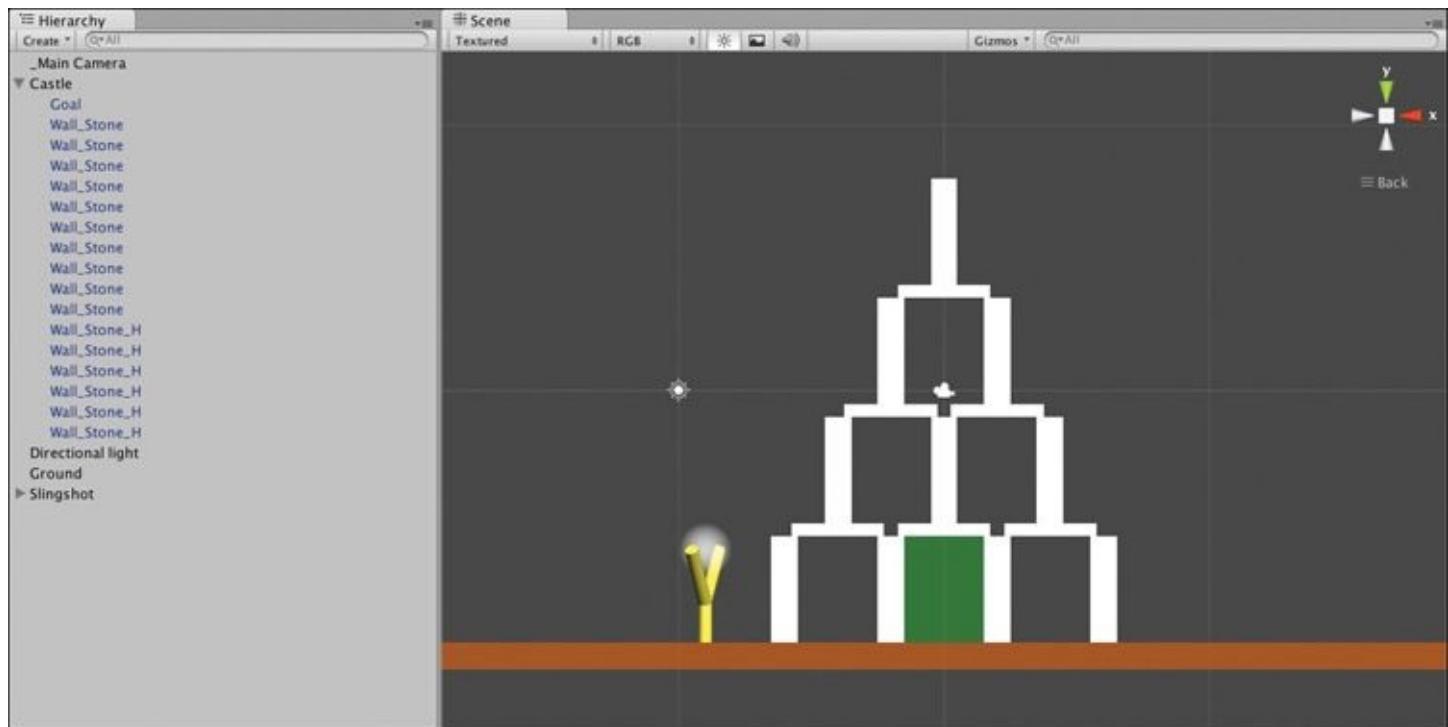


Figure 29.11 The finished castle

10. Set the position of Castle to P:[50,-9.5,0]. And press Play. You may have to try and restart a

couple times, but you should be able to hit Castle with a Projectile.

If you want, you can also add a material to the walls to make them something other than just solid white.

11. Save your scene.

Returning for Another Shot

Now that there's a castle to knock down, it's time to add a little more game logic. Once the Projectile has settled, the camera should move back to focus on Slingshot again:

1. Before doing anything else, you should add a tag of *Projectile* to the Projectile prefab. Select the Projectile prefab in the Project pane. In the Inspector, click the pop-up menu next to Tag and choose *Add Tag*. Click the disclosure triangle next to Tags and enter *Projectile* into *Element 0*. Click Projectile in the Project pane again. Give it a tag by selecting *Projectile* from the updated list of tags in the Inspector.

2. Open the FollowCam C# script in MonoDevelop and modify the following lines:

[Click here to view code image](#)

```
void FixedUpdate () {
    Vector3 destination;
    // If there is no poi, return to P:[0,0,0]
    if (poi == null) {
        destination = Vector3.zero;
    } else {
        // Get the position of the poi
        destination = poi.transform.position;
        // If poi is a Projectile, check to see if it's at rest
        if (poi.tag == "Projectile") {
            // if it is sleeping (that is, not moving)
            if (poi.rigidbody.IsSleeping() ) {
                // return to default view
                poi = null;
                // in the next update
                return;
            }
        }
    }
    // Limit the X & Y to minimum values
    ...
    this.camera.orthographicSize = destination.y + 10;
}
```

Now, once a Projectile has stopped moving (which makes `Rigidbody.IsSleeping()` true), the FollowCam will nullify its `poi`, resetting the camera back to its default position.

3. Save your scene.

Adding a Projectile Trail

While Unity does have a built-in Trail Renderer effect, it won't really serve our purpose because we need more control over the trail than it allows. Instead, we'll make use of the Line Renderer Component upon which the Trail Renderer is built:

1. Start by creating an empty GameObject (`GameObject > Create Empty`) and naming it `ProjectileLine`. Add a Line Renderer component (`Components > Effects > Line Renderer`). In the Line Renderer Inspector for `ProjectileLine`, expand the disclosure triangles for *Materials* and *Parameters*. Set them to the settings shown in [Figure 29.12](#).

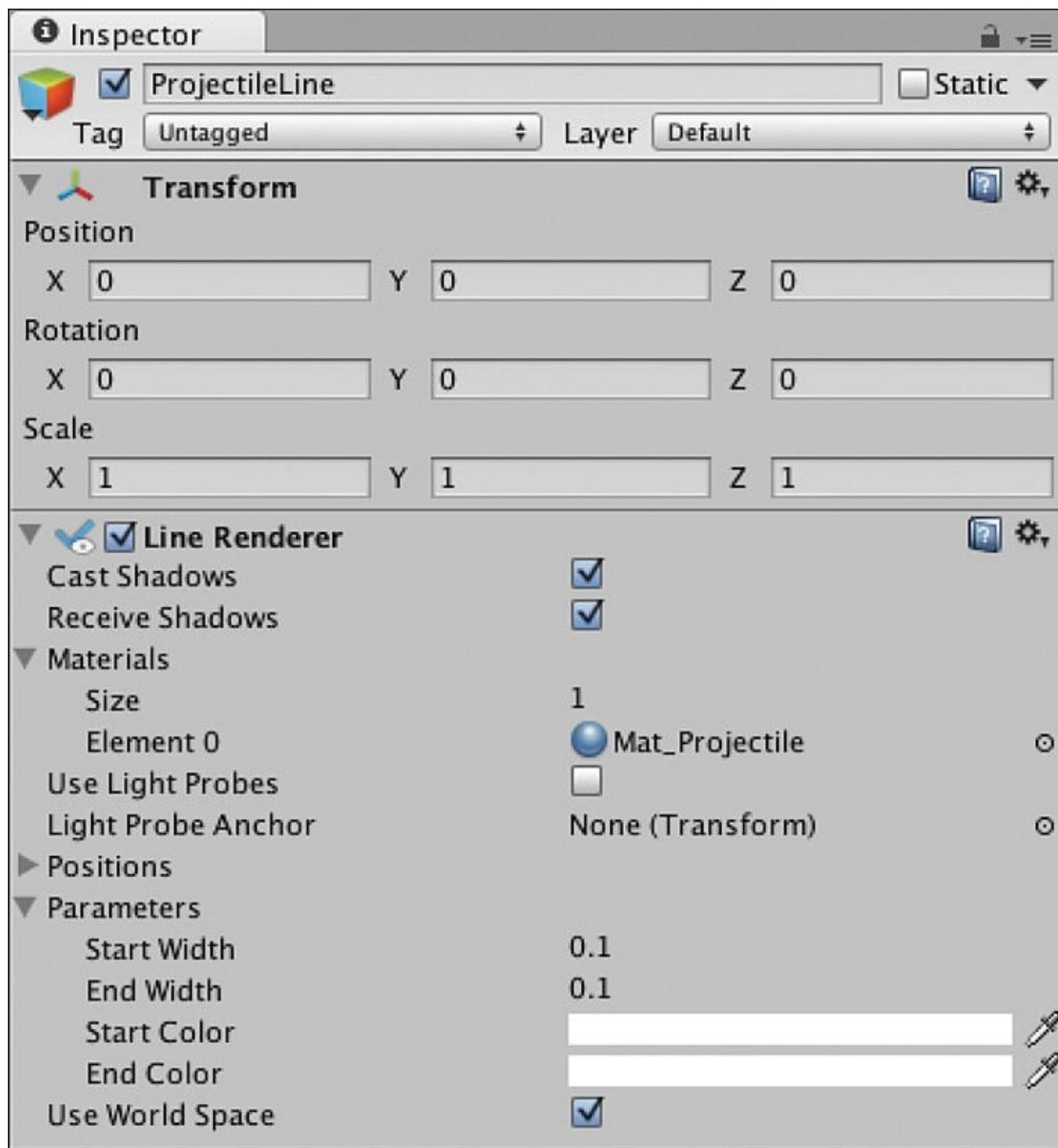


Figure 29.12 Settings for ProjectLine

2. Create a C# script (*Asset > Create > C# Script*), name it *ProjectileLine*, and attach it to the ProjectileLine GameObject. Open the ProjectileLine script in MonoDevelop and write the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;
// Remember, the following line is needed to use Lists
using System.Collections.Generic;

public class ProjectileLine : MonoBehaviour {
    static public ProjectileLine S; // Singleton

    // fields set in the Unity Inspector pane
    public float minDist = 0.1f;
```

```
public bool _____;

// fields set dynamically
public LineRenderer      line;
private GameObject        _poi;
public List<Vector3>     points;

void Awake() {
    S = this; // Set the singleton
    // Get a reference to the LineRenderer
    line = GetComponent<LineRenderer>();
    // Disable the LineRenderer until it's needed
    line.enabled = false;
    // Initialize the points List
    points = new List<Vector3>();
}

// This is a property (that is, a method masquerading as a field)
public GameObject poi {
    get {
        return( _poi );
    }
    set {
        _poi = value;
        if ( _poi != null ) {
            // When _poi is set to something new, it resets everything
            line.enabled = false;
            points = new List<Vector3>();
            AddPoint();
        }
    }
}

// This can be used to clear the line directly
public void Clear() {
    _poi = null;
    line.enabled = false;
    points = new List<Vector3>();
}

public void AddPoint() {
    // This is called to add a point to the line
    Vector3 pt = _poi.transform.position;
    if ( points.Count > 0 && (pt - lastPoint).magnitude < minDist ) {
        // If the point isn't far enough from the last point, it returns
        return;
    }
    if ( points.Count == 0 ) {
        // If this is the launch point...
        Vector3 launchPos = Slingshot.S.launchPoint.transform.position;
        Vector3 launchPosDiff = pt - launchPos;
        // ...it adds an extra bit of line to aid aiming later
        points.Add( pt + launchPosDiff );
        points.Add(pt);
        line.SetVertexCount(2);
        // Sets the first two points
        line.SetPosition(0, points[0] );
        line.SetPosition(1, points[1] );
        // Enables the LineRenderer
        line.enabled = true;
    } else {
        // Normal behavior of adding a point
        points.Add( pt );
        line.SetVertexCount( points.Count );
```

```

        line.SetPosition( points.Count-1, lastPoint );
        line.enabled = true;
    }

    // Returns the location of the most recently added point
    public Vector3 lastPoint {
        get {
            if (points == null) {
                // If there are no points, returns Vector3.zero
                return( Vector3.zero );
            }
            return( points[points.Count-1] );
        }
    }

    void FixedUpdate () {
        if ( poi == null ) {
            // If there is no poi, search for one
            if (FollowCam.S.poi != null) {
                if (FollowCam.S.poi.tag == "Projectile") {
                    poi = FollowCam.S.poi;
                } else {
                    return; // Return if we didn't find a poi
                }
            } else {
                return; // Return if we didn't find a poi
            }
        }
        // If there is a poi, it's loc is added every FixedUpdate
        AddPoint();
        if ( poi.rigidbody.IsSleeping() ) {
            // Once the poi is sleeping, it is cleared
            poi = null;
        }
    }
}

```

3. You will also need to add a singleton to the Slingshot C# script. This will allow AddPoint() to reference the location of Slingshot's launchPoint:

[Click here to view code image](#)

```

public class Slingshot : MonoBehaviour {
    static public Slingshot S;

    // fields set in the Unity Inspector pane
    ...
    void Awake() {
        // Set the Slingshot singleton S
        S = this;

        Transform launchPointTrans = transform.FindChild("LaunchPoint");
    }
}

```

Now when you play the game, you should get a nice gray line that traces the path of the Projectile as it moves. The line is replaced with each subsequent shot.

4. Save your scene.

Hitting the Goal

The goal of the castle needs to react when hit by the projectile:

1. Create a new C# script named *Goal* and attach it to the Goal prefab. Enter the following code into the Goal script.

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Goal : MonoBehaviour {
    // A static field accessible by code anywhere
    static public bool goalMet = false;

    void OnTriggerEnter( Collider other ) {
        // When the trigger is hit by something
        // Check to see if it's a Projectile
        if ( other.gameObject.tag == "Projectile" ) {
            // If so, set goalMet to true
            Goal.goalMet = true;
            // Also set the alpha of the color to higher opacity
            Color c = renderer.material.color;
            c.a = 1;
            renderer.material.color = c;
        }
    }
}
```

Now when you fire, if you can hit the goal, it will turn bright green.

2. Save your scene.

Adding More Levels and Game Logic

The single castle as served us well so far, but let's add a few more.

1. Rename Castle to *Castle_0* and make it a prefab by dragging it into the Project pane.
2. Make a duplicate of *Castle_0* (which will name itself *Castle_1*).
3. Drop *Castle_1* into the Scene pane, and change its layout. It's very likely that you will "lose the prefab" if you delete one of the walls. That is completely fine. Just structure *Castle_1* however you like, and then delete the *Castle_1* Prefab in the Project pane and drag the new *Castle_1* in from the Hierarchy.
4. Repeat this process to make a few different castles. [Figure 29.13](#) shows a few that I made.

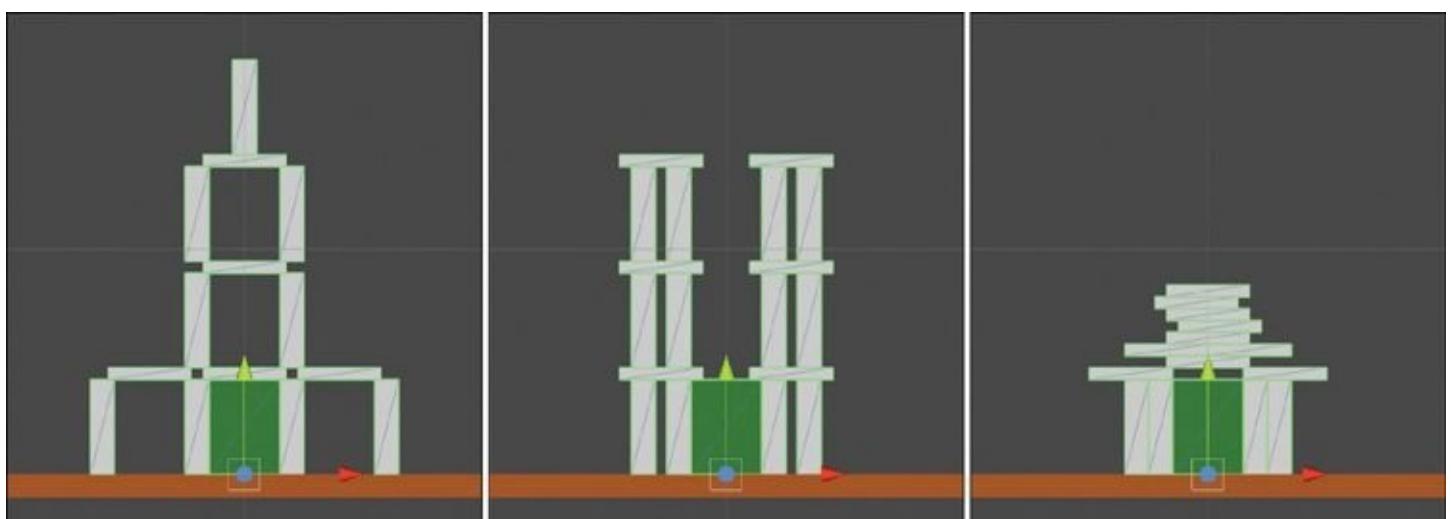


Figure 29.13 More castles

5. Save your scene.

6. Add a *GUIText* to your scene (*GameObject > Create Other > GUIText*) and name it *GT_Level*. Create a second *GUIText* and name it *GT_Shots*. Give each the settings shown in [Figure 29.14](#).

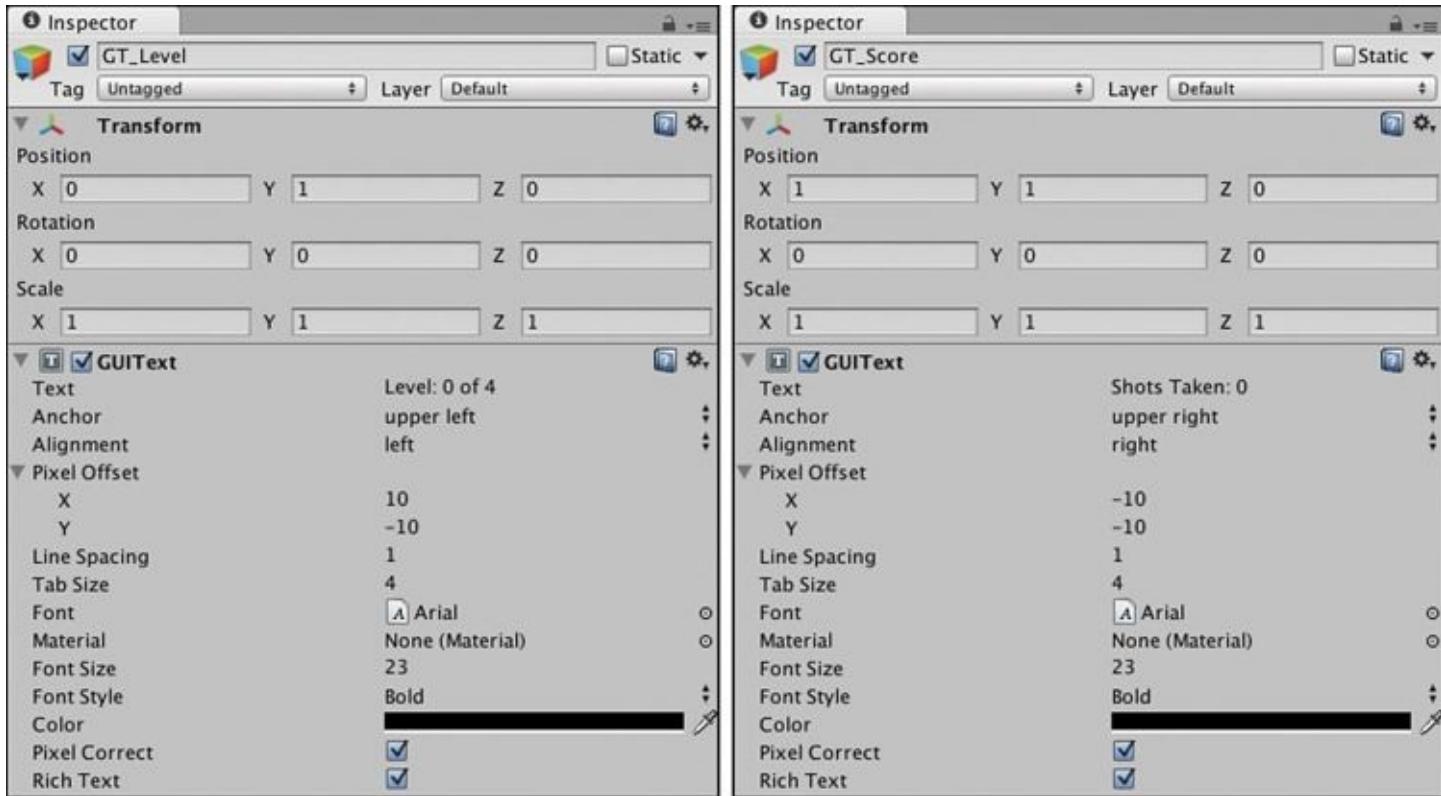


Figure 29.14 Settings for GT_Level and GT_Score

7. Create a new empty GameObject (*GameObject > Create Empty*) and name it *ViewBoth*. Set the transform of ViewBoth to P:[25,25,0] R:[0,0,0] S:[1,1,1]. This will serve as the *poi* for the camera when we want to view both the Castle and the Slingshot.

8. Create a new C# script named *MissionDemolition* and attach it to *_Main Camera*. This will serve as the game state manager for the game. Open it and write the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public enum GameMode {
    idle,
    playing,
    levelEnd
}

public class MissionDemolition : MonoBehaviour {
    static public MissionDemolition S; // a Singleton

    // fields set in the Unity Inspector pane
    public GameObject[] castles; // An array of the castles
    public GUIText gtLevel; // The GT_Level GUIText
    public GUIText gtScore; // The GT_Score GUIText
    public Vector3 castlePos; // The place to put castles

    public bool
    level;
    levelMax; // The number of levels
}
```

```

public int           shotsTaken;
public GameObject   castle;      // The current castle
public GameMode     mode = GameMode.idle;
public string        showing = "Slingshot"; // FollowCam mode

void Start() {
    S = this; // Define the Singleton

    level = 0;
    levelMax = castles.Length;
    StartLevel();
}

void StartLevel() {
    // Get rid of the old castle if one exists
    if (castle != null) {
        Destroy( castle );
    }

    // Destroy old projectiles if they exist
    GameObject[] gos = GameObject.FindGameObjectsWithTag("Projectile");
    foreach (GameObject pTemp in gos) {
        Destroy( pTemp );
    }

    // Instantiate the new castle
    castle = Instantiate( castles[level] ) as GameObject;
    castle.transform.position = castlePos;
    shotsTaken = 0;

    // Reset the camera
    SwitchView("Both");
    ProjectileLine.S.Clear();

    // Reset the goal
    Goal.goalMet = false;

    ShowGT();

    mode = GameMode.playing;
}

void ShowGT() {
    // Show the data in the GUITexts
    gtLevel.text = "Level: "+(level+1)+" of "+levelMax;
    gtScore.text = "Shots Taken: "+shotsTaken;
}

void Update() {
    ShowGT();

    // Check for level end
    if (mode == GameMode.playing && Goal.goalMet) {
        // Change mode to stop checking for level end
        mode = GameMode.levelEnd;
        // Zoom out
        SwitchView("Both");
        // Start the next level in 2 seconds
        Invoke("NextLevel", 2f);
    }
}

void NextLevel() {
    level++;
}

```

```

        if (level == levelMax) {
            level = 0;
        }
        StartLevel();
    }

void OnGUI() {
    // Draw the GUI button for view switching at the top of the screen
    Rect buttonRect = new Rect( (Screen.width/2)-50, 10, 100, 24 );

    switch(showing) {
        case "Slingshot":
            if ( GUI.Button( buttonRect, "Show Castle" ) ) {
                SwitchView("Castle");
            }
            break;

        case "Castle":
            if ( GUI.Button( buttonRect, "Show Both" ) ) {
                SwitchView("Both");
            }
            break;

        case "Both":
            if ( GUI.Button( buttonRect, "Show Slingshot" ) ) {
                SwitchView( "Slingshot" );
            }
            break;
    }
}

// Static method that allows code anywhere to request a view change
static public void SwitchView( string eView ) {
    S.showing = eView;
    switch (S.showing) {
        case "Slingshot":
            FollowCam.S.poi = null;
            break;

        case "Castle":
            FollowCam.S.poi = S.castle;
            break;

        case "Both":
            FollowCam.S.poi = GameObject.Find("ViewBoth");
            break;
    }
}

// Static method that allows code anywhere to increment shotsTaken
public static void ShotFired() {
    S.shotsTaken++;
}
}

```

9. Now that there is a static `ShotFired()` method on the `MissionDemolition` class, it's possible to call it from the `Slingshot` class. Add the following bold line to the `Slingshot` C# script:

[Click here to view code image](#)

```

public class Slingshot : MonoBehaviour {
    ...
    void Update() {
        ...
        if ( Input.GetMouseButtonUp(0) ) {
            ...
            projectile = null;
            MissionDemolition.ShotFired();
        }
    }
}

```

Because the `ShotFired()` method on `MissionDemolition` is static, it can be accessed through the `MissionDemolition` class itself rather than requiring you to access it via a specific instance of `MissionDemolition`. When `Slingshot` calls `MissionDemolition.ShotFired()`, it causes `MissionDemolition.S.shotsTaken` to increment.

10. Switch back to Unity and select `_Main Camera` in the Hierarchy. In the *MissionDemolition* (Script) component Inspector, you will need to set a few variables.
 - 10.1. First, set `castlePos` to `[50,-9.5,0]`, which will place the castles a nice distance from your `Slingshot`.
 - 10.2. To set `gtLevel`, click the target in the Inspector to the right of `gtLevel` and select `GT_Level` from the Scene tab in the pop-up dialog box.
 - 10.3. Click the target next to `gtScore` in the Inspector and choose `GT_Score` from the Scene tab.
 - 10.4. Next, click the disclosure triangle next to `castles` and set its length to the number of castles you made previously. (In the example in [Figure 29.15](#), I made four castles.)

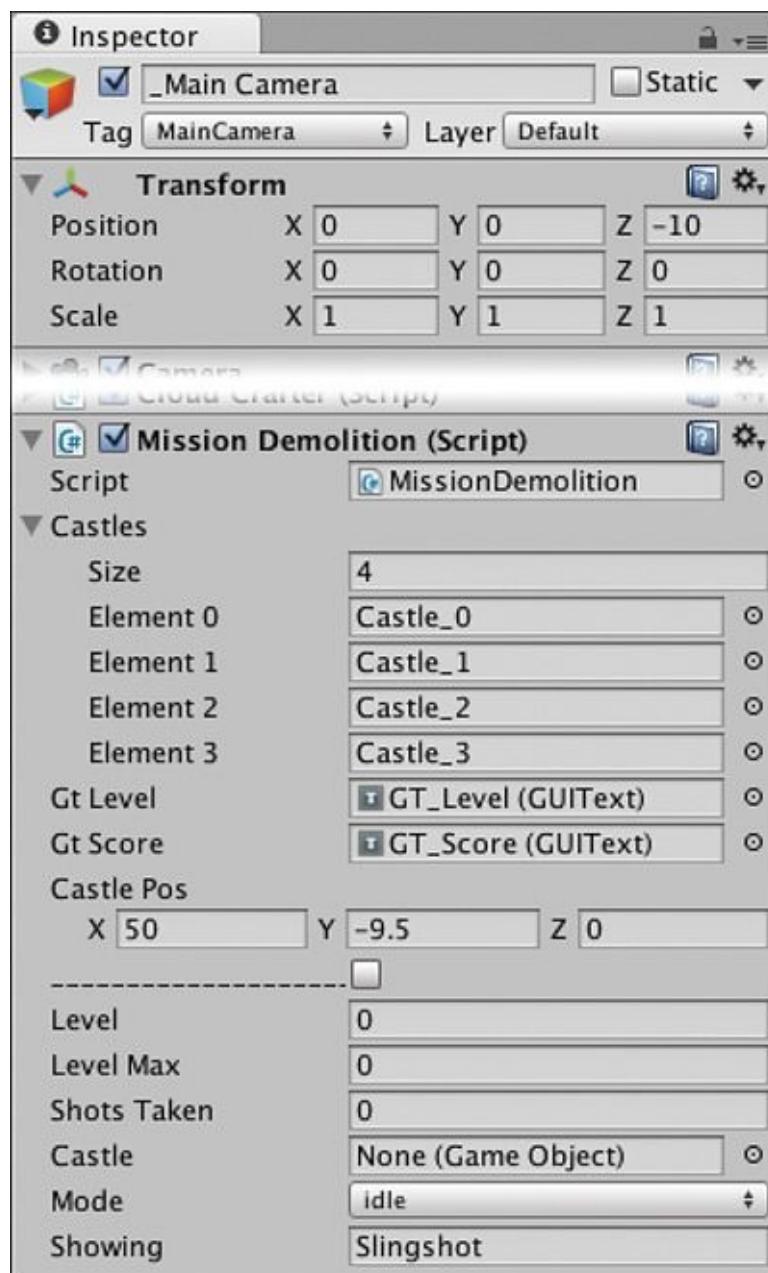


Figure 29.15 Final settings (with Castles array) for `_Main Camera:MissionDemolition`

- 10.5.** Drag each of the numbered Castle prefabs you made into an element of the `castles` array to set the levels for your game. Try to order them from easiest to most difficult.
- 10.6.** Save your scene and press Play. Now the game will play through various levels and keep track of how many shots you've fired.

Summary

That's it for the Mission Demolition prototype. In just one chapter, you've made a physics-based game like *Angry Birds* that you can continue to improve and expand on your own. This and all of the following tutorials are really meant to be frameworks on top of which you can build whatever game you want, and there are a ton of additional features you could add, some of which include the following:

1. Use PlayerPrefs to store the best score on each level as was done in Apple Picker.
2. Make the castle parts out of various materials, some of which would have more or less mass. Some materials could even break if struck hard enough.
3. Show lines for multiple previous paths rather than just the most recent one.

4. Use a Line Renderer to draw the rubber band of the slingshot.
5. Implement actual parallax scrolling on the background clouds, and add more background elements like mountains or buildings.
6. Limit the number of shots so that the player loses the level if she doesn't hit the goal in only 3 shots. Adding risk like this increases the tension and excitement in the game.
7. Anything else that you want!

Next Steps

After you've worked your way through the other prototypes, come back to this one and think about what you could add to it. Create your own designs, show them to people, and iterate to make the game better. Remember that design is always an iterative process. If you make a change you don't like, don't let it discourage you; just chalk it up to experience and try something else.

Chapter 30. Prototype 3: Space SHMUP

The SHMUP (or shoot 'em up) game genre includes such classic games as *Galaga* and *Galaxian* from the 1980s and the modern masterpiece *Ikaruga*.

In this chapter, you create a SHMUP using several programming techniques that will serve you well throughout your programming and prototyping careers, including class inheritance, enums, static fields and methods, and the singleton pattern. Though you've seen many of these techniques before, they will be used more extensively in this prototype.

Getting Started: Prototype 3

In this project, you make a prototype for a classic space-based SHMUP. [Figure 30.1](#) shows two images of what the finished prototype will look like. In both images, the player has powered-up her weapons and already taken out some enemy ships (which left behind the power-up cubes marked B, O, and S). In the left image, she is using the blaster weapon, and in the right, she is using the spread weapon.

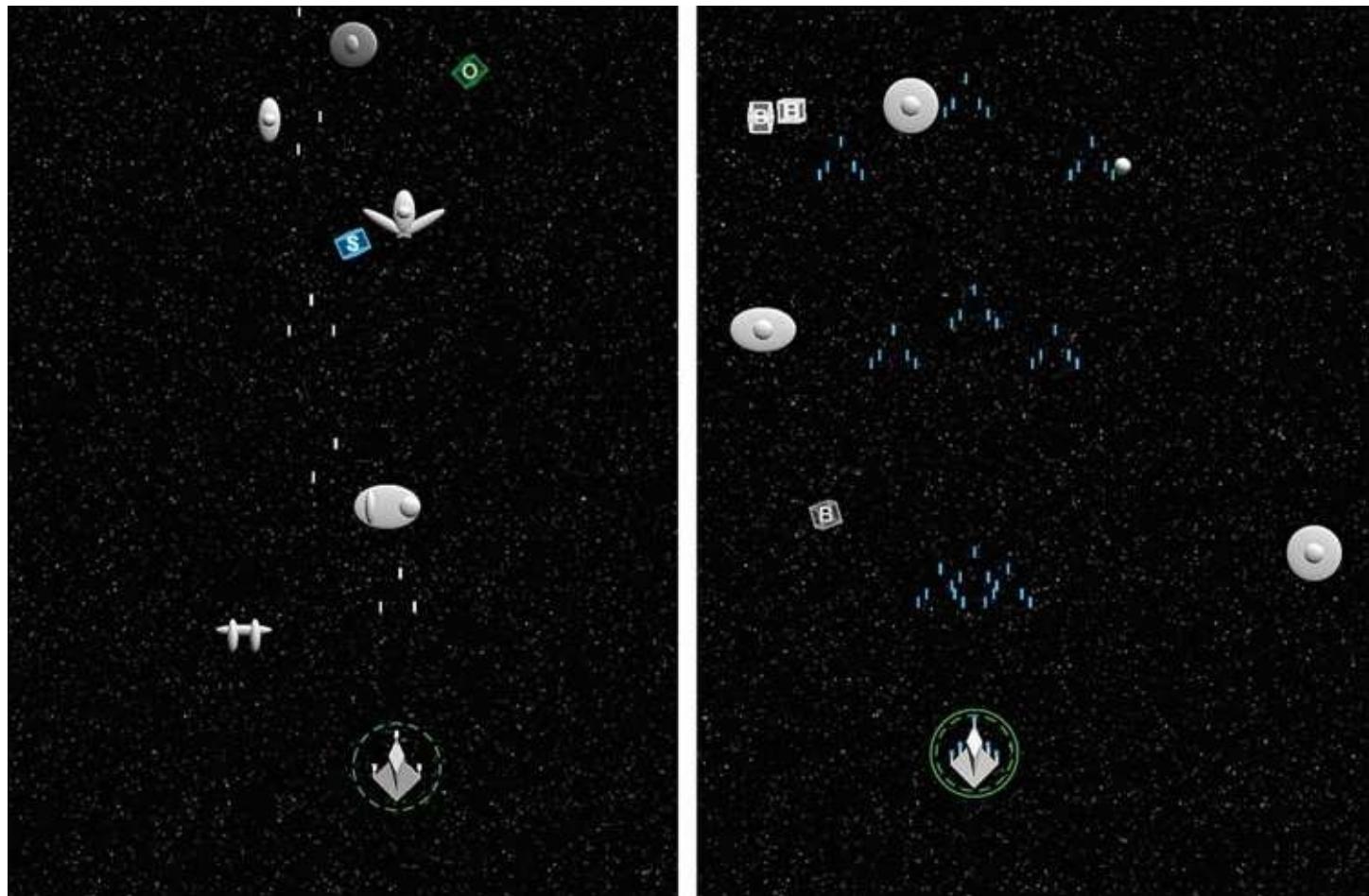


Figure 30.1 Two views of the Space SHMUP game prototype

Importing a Unity Asset Package

One new thing in the setup for this prototype is that you will be asked to download and import a custom Unity asset package. The creation of complex art and imagery for games is beyond the scope of this book, but I've created a package of some simple assets for you that will allow you to create all the visual effects required for this game. Of course, as mentioned several times throughout this book, when you're making a prototype, how it plays and feels are much more important than how it looks, but it's still important to have an understanding of how to work with art assets.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on this procedure, see [Appendix A, “Standard Project Setup Procedure.”](#)

- **Project name:** Space SHMUP Prototype
- **Scene name:** __Scene_0
- **Project folders:** __Scripts, __Materials, __Prefabs
- **Download and import package:** Find [Chapter 30](#) at <http://book.prototools.net>
- **C# script names:** (none yet)
- **Rename:** Change Main Camera to __MainCamera

To download and install the package mentioned in the sidebar “Set Up the Project for This Chapter,” first follow the URL listed (<http://book.prototools.net>) and search for this chapter. Download *Chapter30.unitypackage* to your machine, which will usually place it in your Downloads folder. Open your project in Unity and select *Assets > Import Package > Custom Package* from the menu bar. Navigate to and select *Chapter30.unitypackage* from your Downloads folder. This will open the import dialog box shown in [Figure 30.2](#).

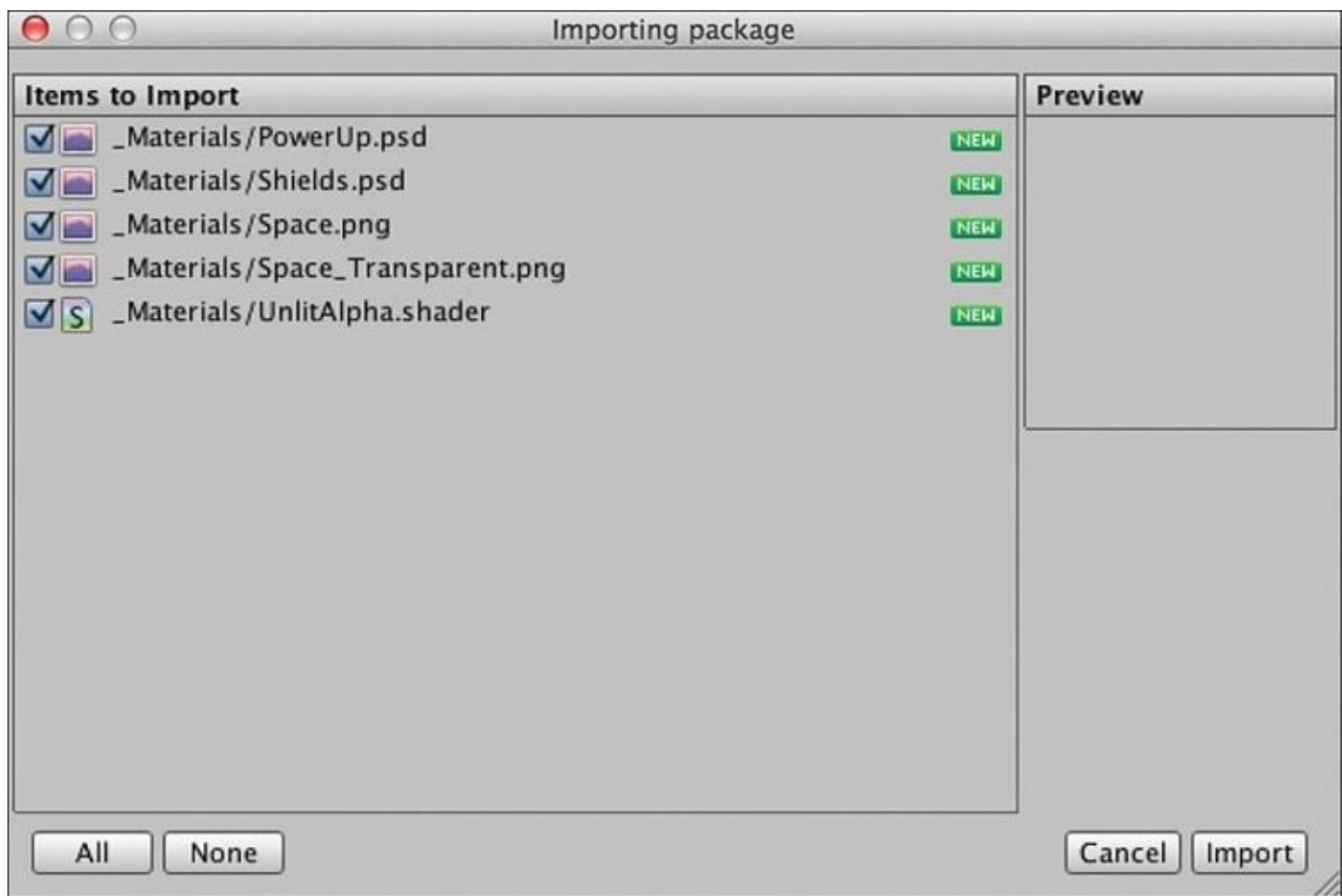


Figure 30.2 The .unitypackage import dialog box

Select all the files as shown in [Figure 30.2](#), and click *Import*. This will place four new textures and one new shader into the `_Materials` folder. The creation of textures is beyond the scope of this book, but many books and online tutorials cover texture creation. Adobe Photoshop is probably the most commonly used image editing tool, but it's very expensive. A common open source alternative is Gimp (<http://www.gimp.org>).

The creation of shaders is also far beyond the scope of this book. Shaders are programs that tell your computer how to render a texture on a `GameObject`. They can make a scene look realistic, cartoony, or however else you like, and they are an important part of the graphics of any modern game. Unity uses its own unique shader language called `ShaderLab`. If you want to learn more about it, a good place to start is the Unity Shader Reference documentation (<http://docs.unity3d.com/Documentation/Components/SL-Reference.html>).

The included shader is a simple one that bypasses most of the things a shader can do to simply render a colored, unlit shape on the screen. For on-screen elements that you want to be a specific bright color, the imported `UnlitAlpha.shader` is perfect. `UnlitAlpha` also allows for alpha blending and transparency, which will be very useful for the power-up cubes in this game.

Setting the Scene

Add a directional light to the scene (`GameObject > Create Other > Directional Light` from the menu bar). Set its transform to `P:[0,20,0] R:[50,330,0] S:[1,1,1]`.

Select `_MainCamera` and set its transform to `P:[0,0,-10] R:[0,0,0] S:[1,1,1]`. In the Camera component, set the *Background color* to black. Set *Projection* to Orthographic and *Size* to 40. Set the *Near* and *Far Clipping Planes* to 0.3 and 100, respectively.

Because this game will be a vertical, top-down shooter, we need to set an aspect ratio for the Game pane that is in portrait orientation. In the Game pane, click the pop-up menu list of aspect ratios directly underneath the tab for the pane. At the bottom of the list, you will see a + symbol. Click this to add a new aspect ratio preset. Set the values to those shown in [Figure 30.3](#), and then click *Add OK*. Set the Game pane to this new *Portrait (3:4)* aspect ratio.

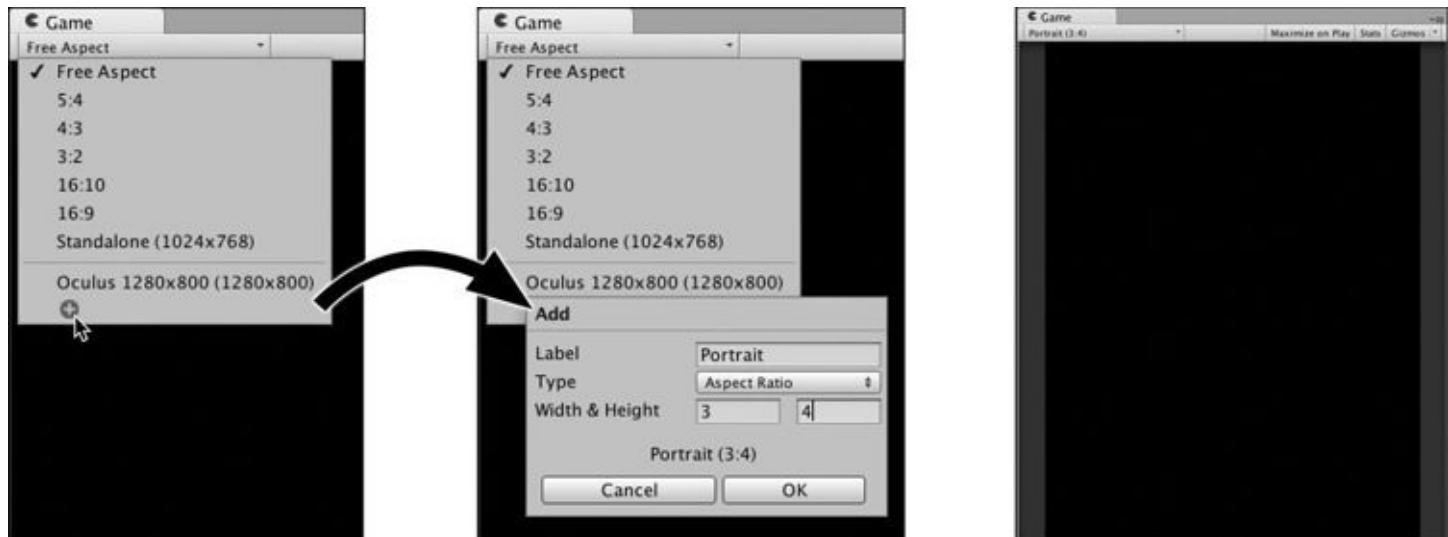


Figure 30.3 Adding a new aspect ratio preset to the Game pane

Making the Hero Ship

In this chapter, we interleave the construction of artwork and code rather than building all the art first. To make the player's spaceship, complete these steps:

1. Create an empty GameObject and name it *_Hero* (*GameObject > Create Empty*). Set its transform to P:[0,0,0] R:[0,0,0] S:[1,1,1].
2. Create a cube (*GameObject > Create Other > Cube*) and drag it onto *_Hero* in the Hierarchy, making it a child of *_Hero*. Name the cube *Wing* and set its transform to P:[0,-1,0] R:[0,0,45] S:[3,3,0.5].
3. Create an empty GameObject, name it *Cockpit*, and make it a child of *_Hero*.
4. Create a cube and make it a child of *Cockpit*. Set the Cube's transform to P:[0,0,0] R:[315,0,45] S:[1,1,1].
5. Make *Cockpit*'s transform P:[0,0,0] R:[0,0,180] S:[1,3,1]. This uses the same trick as in [Chapter 26, “Object-Oriented Thinking,”](#) to make a quick, angular ship.
6. Create a new C# script and name it *Hero* (*Assets > Create > C# Script* from the menu bar). Be sure to place this script into the *_Scripts* folder. Drag the *Hero* script onto the *_Hero* GameObject to attach it.
7. Add a Rigidbody component to *_Hero* by selecting *_Hero* in the Hierarchy and then choosing *Component > Physics > Rigidbody* from the menu bar. Set *Use Gravity* to false and *isKinematic* to true. Open the disclosure triangle for *Constraints* and freeze z position and x, y, and z rotation.

You'll add more to *_Hero* later, but this will suffice for now.

Save your scene! Remember that you should be saving your scene every time you make a change to it. I'll quiz you later.

Hero.Update()

In the code listing that follows, the `Update()` method first reads the horizontal and vertical axes from the `InputManager` (see the “[Input.GetAxis\(\) and The InputManager](#)” sidebar), placing values between -1 and 1 into the floats `xAxis` and `yAxis`. The second chunk of `Update()` code moves the ship in a time-based way, taking into account the speed setting.

The last line (marked `// 2`) rotates the ship based on the input. Although we earlier froze rotation in the `Rigidbody` component, it is still possible for us to manually set the rotation on a `Rigidbody` with `isKinematic` set to `true`. (As discussed in an earlier chapter, `isKinematic=true` means that the `Rigidbody` will be tracked by the physics system but that it will not move automatically due to `Rigidbody.velocity`.) This rotation will make the movement of the ship feel more dynamic and expressive, or “juicy.”¹

¹ *Juiciness*, as a term that relates to gameplay, was coined in 2005 by Kyle Gabler and the other members of the Experimental Gameplay Project at Carnegie Mellon University’s Entertainment Technology Center. To them, a juicy element had “constant and bountiful user feedback.” You can read about it more in their Gamasutra article “How to Prototype a Game in Under 7 Days.” http://www.gamasutra.com/view/feature/130848/how_to_prototype_a_game_in_under_7.php.

Open the `Hero` C# script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Hero : MonoBehaviour {
    static public Hero S; // Singleton

    // These fields control the movement of the ship
    public float speed = 30;
    public float rollMult = -45;
    public float pitchMult = 30;

    // Ship status information
    public float shieldLevel = 1;

    public bool _____;

    void Awake() {
        S = this; // Set the Singleton
    }

    void Update () {
        // Pull in information from the Input class
        float xAxis = Input.GetAxis("Horizontal"); // 1
        float yAxis = Input.GetAxis("Vertical"); // 1

        // Change transform.position based on the axes
        Vector3 pos = transform.position;
        pos.x += xAxis * speed * Time.deltaTime;
        pos.y += yAxis * speed * Time.deltaTime;
        transform.position = pos;

        // Rotate the ship to make it feel more dynamic // 2
        transform.rotation = Quaternion.Euler(yAxis*pitchMult, xAxis*rollMult, 0);
    }
}
```

1. These two lines use Unity’s `Input` class to pull information from the Unity `InputManager`. See the sidebar for more information.

2. The `transform.rotation...` line below this comment is used to give the ship a bit of rotation based on the speed at which it is moving, which can make the ship feel more reactive and juicy.

Input.GetAxis() and The InputManager

Much of the code in the `Hero.Update()` code listing should look familiar to you, though this is the first time in the book that we've used the `Input.GetAxis()` method. Various axes are configured in Unity's InputManager, and `Input.GetAxis()` allows them to be read. To view the default Input axes, choose *Edit > Project Settings > Input* from the menu bar.

One thing to note about the settings in [Figure 30.4](#) is that there are several that are listed twice (for example, Horizontal, Vertical, Jump). As you can see in the expanded view of the horizontal axes in the figure, this allows the horizontal axis to be controlled by either presses on the keyboard (shown in the left image of [Figure 30.4](#)) or a joystick axis (shown in the right image). This is one of the great strengths of the input axes; several different types of input can control a single axis. As a result, your games only need one line to read the value of an axis rather than a line to handle joystick input, a line for each arrow key, and a line each for the A and D keys.

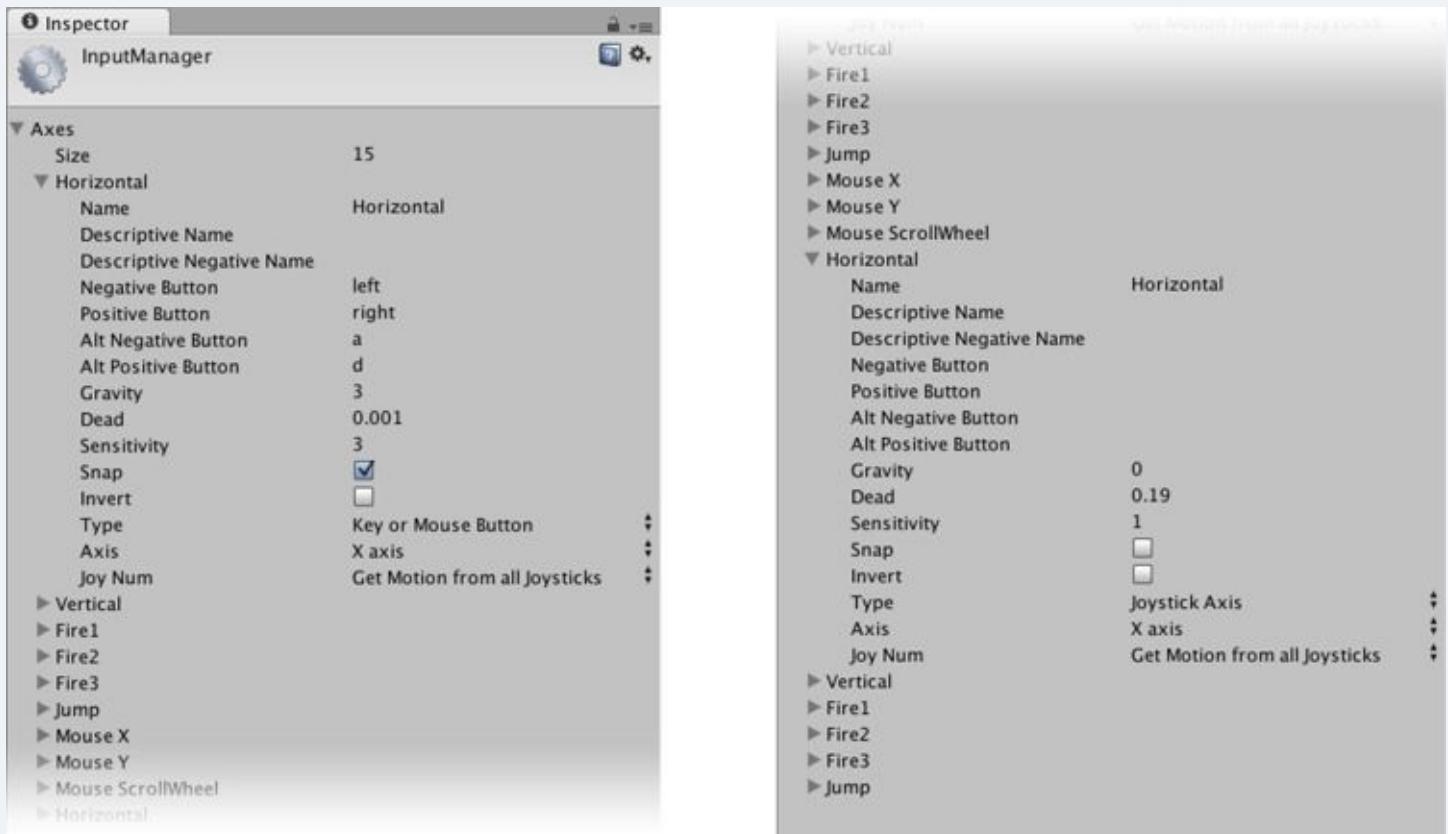


Figure 30.4 Unity's InputManager showing default settings (shown in two halves)

Every call to `Input.GetAxis()` will return a float between -1 and 1 in value (with a default of 0). Each axis in the InputManager also includes values for *Sensitivity* and *Gravity*, though these are only used for *Key or Mouse Button* input (see the left image of [Figure 30.4](#)). Sensitivity and gravity cause the axis value to interpolate smoothly when a key is pressed or released. (That is, instead of immediately jumping to the final value, the axis value will blend from the original value to the final value over time.) In the horizontal axis shown, a sensitivity of 3 means that when the right-arrow key is pressed, it will take

1/3 of a second for the value to interpolate from 0 to 1. A gravity of 3 means that when the right-arrow key is released, it will take 1/3 of a second for the axis value to interpolate back to 0. The higher the sensitivity or gravity, the faster the interpolation will take place.

As with almost anything in Unity, you can find out a lot more about the `InputManager` by clicking the Help button (that looks like a blue book with a question mark and is between the name `InputManager` and the gear at the top of the Inspector).

Try playing the game and see how the ship feels to you. The settings for `speed`, `rollMult`, and `pitchMult` work for me, but this is your game, and you should have settings that feel right to you. Make changes as necessary in the Inspector for `_Hero`.

Part of what makes this feel nice is the apparent inertia that the ship carries. When you release the movement key, it takes the ship a little while to slow down. Similarly, upon pressing a movement key, it takes the ship a little while to get up to speed. This apparent movement inertia is caused by the *sensitivity* and *gravity* axis settings that are described in the sidebar. Changing these settings in the `InputManager` will affect the movement and maneuverability of `_Hero`.

The Hero Shield

The shield for `_Hero` will be a combination of a transparent, textured quad (to provide the visuals) and a `Sphere Collider` (for collision handling).

Create a new quad (`GameObject > Create Other > Quad`). Rename the quad `Shield` and make it a child of `_Hero`. Set the transform of `Shield` to `P:[0,0,0] R:[0,0,0], S:[8,8,8]`.

Select `Shield` in the Hierarchy and add a `Sphere Collider` component (`Component > Physics > Sphere Collider`). Then delete the existing `Mesh Collider` component by clicking the tiny gear to the right of the `Mesh Collider` name in the Inspector and choosing *Remove Component* from the pop-up menu.

Create a new material (`Assets > Create > Material`), name it `Mat Shield`, and place it in the `_Materials` folder in the Project pane. Drag `Mat Shield` onto `Shield` (under `_Hero` in the Hierarchy) to assign it to the `Shield` quad.

Select `Shield` in the Hierarchy, and you will now see `Mat Shield` in the Inspector for `Shield`. Set the *Shader* of `Mat Shield` to `ProtoTools > UnlitAlpha`. Below the shader selection pop-up for `Mat Shield`, there should be an area that allows you to choose the main color for the material as well as the texture. (If you don't see this, click once on the name `Mat Shield` in the Inspector, and it should appear.) Click *Select* in the bottom-right corner of the texture square and select the texture named `Shields`. Click the color swatch next to *Main Color* and choose a bright green (`RGBA:[0,255,0,255]`). Then set the *Tiling.x* to 0.2 and the *Offset.x* to 0.4. The x Tiling of 0.2 causes `Mat Shield` to only use 1/5 of the total `Shield` texture in the x direction, and the x Offset chooses which fifth. Try x Offsets of 0, 0.2, 0.4, 0.6, and 0.8 to see the different levels of shield strength. *Tiling.y* should remain 1.0, and *Offset.y* should remain 0. This is because the texture was designed to be split into five sections horizontally but only one vertically.

Create a new C# script named `Shield` (`Asset > Create > C# Script`). Drop it into the `_Scripts` folder in the Project pane and then drag it onto `Shield` in the Hierarchy to assign it as a component of the `Shield` `GameObject`. Open the `Shield` script in MonoDevelop and enter the following code:

[Click here to view code image](#)

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

```

public class Shield : MonoBehaviour {
    public float rotationsPerSecond = 0.1f;
    public bool levelShown;
    public int levelShown = 0;

    void Update () {
        // Read the current shield level from the Hero Singleton
        int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel ); // 1
        // If this is different from levelShown...
        if (levelShown != currLevel) {
            levelShown = currLevel;
            Material mat = this.renderer.material;
            // Adjust the texture offset to show different shield level
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0 ); // 2
        }
        // Rotate the shield a bit every second
        float rz = (rotationsPerSecond*Time.time*360) % 360f; // 3
        transform.rotation = Quaternion.Euler( 0, 0, rz );
    }
}

```

1. currLevel is set to the floor of the current Hero.S.shieldLevel float. By flooring the shieldLevel, we make sure that the shield jumps to the new x Offset rather than showing an Offset between two shield icons.
2. This line adjusts the x Offset of Mat Shield to show the proper shield level.
3. This line and the next cause the Shield GameObject to rotate slowly around the z axis every frame.

Keeping _Hero On Screen

The motion of your _Hero ship should feel pretty good now, and the rotating shield looks pretty nice, but at this point, you can easily drive the ship off the screen. This is going to be a bit more complex than some of the other things we've done, but you're now going to write some reusable code to keep the ship constrained to the screen.

Bounds

Both renderers and colliders have a bounds field that is of the type Bounds. Bounds are defined by a center and a size, each of which are Vector3s. In [Figure 30.5](#), this is explained in two dimensions; just remember that there is also a z dimension when working in Unity.

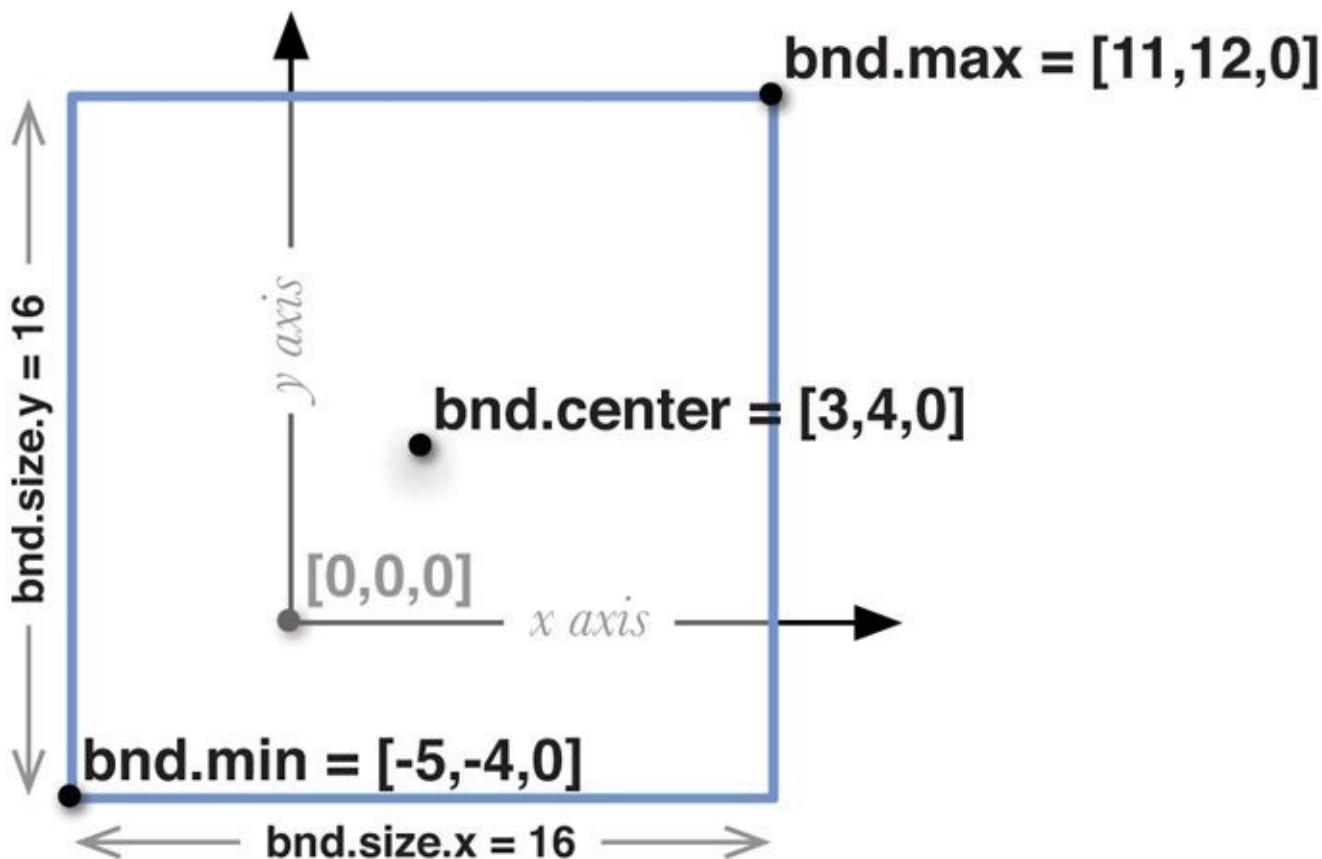


Figure 30.5 Diagram showing the various fields of Bounds bnd, defined as Bounds bnd = new Bounds(new Vector3(3, 4, 0), new Vector3(16, 16, 0));

Compositing the Bounds of a Complex GameObject

_Hero is a complex GameObject with several children, however _Hero itself has no colliders. To find the collision bounds of _Hero, it is necessary to find the bounds of each of the children of _Hero and then to create a Bounds variable that encompasses all of them. Unity doesn't include any function for expanding Bounds to envelope other Bounds, so we'll need to write one called BoundsUnion (named so because it returns the mathematical union of the two Bounds). This seems like something that might also be useful in later games, so we'll make this part of a new *Utils* C# class that we will fill with reusable game code. The Utils class is going to be almost entirely composed of static functions so that the functions can easily be called from anywhere in your code.

Create a new C# script named *Utils* and place it in the __Scripts folder. Open *Utils* in MonoDevelop and enter the following code:

[Click here to view code image](#)

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

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

// Creates bounds that encapsulate the two Bounds passed in.
public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
    // If the size of one of the bounds is Vector3.zero, ignore that one
    if ( b0.size == Vector3.zero && b1.size != Vector3.zero ) { // 1
        return( b1 );
    } else if ( b0.size != Vector3.zero && b1.size == Vector3.zero ) {
```

```

        return( b0 );
    } else if ( b0.size == Vector3.zero && b1.size == Vector3.zero ) {
        return( b0 );
    }
    // Stretch b0 to include the b1.min and b1.max
    b0.Encapsulate(b1.min); // 2
    b0.Encapsulate(b1.max);
    return( b0 );
}
}

```

1. This `if` clause ensures that neither of the bounds have a size of 0. If the size of one is `Vector3.zero`, then the other is returned. If both have a size of zero, `b0` is returned.
2. Though the Unity `Bounds` class doesn't include a function to expand to encompass other `Bounds`, it does have one to encompass a `Vector3`. `b0.Encapsulate(b1.min)` will expand `Bounds b0` to include the `Vector3 b1.min`, and if both `b1.min` and `b1.max` are inside the newly expanded `b0`, then `b0` has expanded to surround `b1` as well.

Add the following bold code to the `Utils` class after `BoundsUnion()`:

[Click here to view code image](#)

```

public class Utils : MonoBehaviour {

//===== Bounds Functions =====\\

// Creates bounds that encapsulate of the two Bounds passed in.
public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
    ...
}

public static Bounds CombineBoundsOfChildren(GameObject go) {
    // Create an empty Bounds b
    Bounds b = new Bounds(Vector3.zero, Vector3.zero);
    // If this GameObject has a Renderer Component...
    if (go.renderer != null) {
        // Expand b to contain the Renderer's Bounds
        b = BoundsUnion(b, go.renderer.bounds);
    }
    // If this GameObject has a Collider Component...
    if (go.collider != null) {
        // Expand b to contain the Collider's Bounds
        b = BoundsUnion(b, go.collider.bounds);
    }
    // Recursively iterate through each child of this gameobject.transform
    foreach( Transform t in go.transform ) { // 1
        // Expand b to contain their Bounds as well
        b = BoundsUnion( b, CombineBoundsOfChildren( t.gameObject ) ); // 2
    }
    return( b );
}
}

```

1. The `Transform` class supports enumerators (by implementing the `IEnumerable` interface), which makes it possible to loop over each child of a `Transform` using a `foreach` loop.
2. Because `CombineBoundsOfChildren()` calls itself (actually another instance of itself), this is also another example of a recursive function. (Recursive functions were first covered in [Chapter 23, “Functions and Parameters.”](#))

Now it's possible to get the combined bounds of any GameObject and its children by passing it into the `CombineBoundsOfChildren()` method. Open the Hero C# script and add the following bold lines of code to get the combined bounds of `_Hero`:

[Click here to view code image](#)

```
public bool _____;
public Bounds bounds;
void Awake() {
    S = this; // Set the Singleton
    bounds = Utils.CombineBoundsOfChildren(this.gameObject);
}
```

Making `CombineBoundsOfChildren()` a static method of `Utils` makes it very easy to call from anywhere in your code. The call to `Utils.CombineBoundsOfChildren()` has the potential to take a good amount of processing power and time if it's called on a `GameObject` with many children, so it's called only once. Later, we'll update the center of the bounds every frame to keep it up-to-date as the ship moves across the screen.

Finding the Bounds of the Camera

To keep `_Hero` on screen, it's also necessary to know the bounds of the camera's field of view. With a perspective camera, this would be pretty tricky, but orthographic cameras are much easier as long as the orthographic camera is not rotated. To find the bounds of the camera, we'll create two `Vector3s` (`boundTLN` and `boundBRF` for top-left near and bottom-right far, respectively). These will be defined by passing the top-left and bottom-right coordinates of the screen into `Camera.ScreenToWorldPoint` and replacing the `z` value of the resultant `Vector3s` with the `z` plane of the near and far settings of the camera.

Open `Utils` in MonoDevelop and add the following bold code after the static method `CombineBoundsOfChildren()`:

[Click here to view code image](#)

```
public class Utils : MonoBehaviour {
    //===== Bounds Functions =====\\
    // Creates bounds that encapsulate of the two Bounds passed in.
    public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
        ...
    }

    public static Bounds CombineBoundsOfChildren(GameObject go) {
        ...
    }

    // Make a static read-only public property camBounds
    static public Bounds camBounds { // 1
        get {
            // if _camBounds hasn't been set yet
            if (_camBounds.size == Vector3.zero) {
                // SetCameraBounds using the default Camera
                SetCameraBounds();
            }
            return( _camBounds );
        }
    }
}
```

```

}

// This is the private static field that camBounds uses
static private Bounds _camBounds; // 2

// This function is used by camBounds to set _camBounds and can also be
// called directly.
public static void SetCameraBounds(Camera cam=null) { // 3
    // If no Camera was passed in, use the main Camera
    if (cam == null) cam = Camera.main;
    // This makes a couple of important assumptions about the camera!:
    // 1. The camera is Orthographic
    // 2. The camera is at a rotation of R:[0,0,0]

    // Make Vector3s at the topLeft and bottomRight of the Screen coords
    Vector3 topLeft = new Vector3( 0, 0, 0 );
    Vector3 bottomRight = new Vector3( Screen.width, Screen.height, 0 );

    // Convert these to world coordinates
    Vector3 boundTLN = cam.ScreenToWorldPoint( topLeft );
    Vector3 boundBRF = cam.ScreenToWorldPoint( bottomRight );

    // Adjust their zs to be at the near and far Camera clipping planes
    boundTLN.z += cam.nearClipPlane;
    boundBRF.z += cam.farClipPlane;

    // Find the center of the Bounds
    Vector3 center = (boundTLN + boundBRF)/2f;
    _camBounds = new Bounds( center, Vector3.zero );
    // Expand _camBounds to encapsulate the extents.
    _camBounds.Encapsulate( boundTLN );
    _camBounds.Encapsulate( boundBRF );
}

}

```

1. `Utils.camBounds` is a public static read-only property. As a property, it runs the code in the `get{}` clause whenever it is accessed. If the private static field `_camBounds` has not yet been set, the `get{}` clause will call `Utils.SetCameraBounds()` to set `_camBounds` before returning. This technique is used to make sure that `_camBounds` is set just in time to be read by `camBounds` and to make sure that `Utils.SetCameraBounds()` is called only once.
2. Note that the order in which `camBounds` and `_camBounds` are declared does not matter to C#. The compiler reads everything that is declared in the `Utils` class before interpreting any code.
3. The static public `SetCameraBounds()` method has a default `Camera cam` value of `null`. If nothing is passed into `SetCameraBounds()` as an argument, it will replace the `null` value in `cam` with `Camera.main` (the main camera of the scene, or `_MainCamera` in this scene). If a programmer wishes to use a camera other than `_MainCamera` for `_camBounds`, she can call `Utils.SetCameraBounds()` directly instead.

Testing and Responding to the Overlap of Two Bounds

The last component we will need to keep `_Hero` on screen is the ability to test whether two `Bounds` overlap. The following code will make use of an `enum`, which you can learn more about in the [“Enum” sidebar](#).

Start by adding the bold code in the following listing *before* the declaration of the `Utils` class in the

Utils C# script:

[Click here to view code image](#)

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

// This is actually OUTSIDE of the Utils Class
public enum BoundsTest {
    center,      // Is the center of the GameObject on screen?
    onScreen,    // Are the bounds entirely on screen?
    offScreen    // Are the bounds entirely off screen?
}

public class Utils : MonoBehaviour {
    ...
}
```

Enum

An *enum* (or enumeration) is a way of defining specific, named numbers in C#. The enum definition at the top of the Utils C# script declares an enum type `BoundsTest` with three potential values: `center`, `onScreen`, and `offScreen`. Once an enum is defined, a variable can then be declared that uses the defined enum as its type.

```
public BoundsTest testMode = BoundsTest.center;
```

The preceding line will create a new variable named `testMode` that is of the type `BoundsTest` and has the value `BoundsTest.center`.

Enums are often used in code when there are only a few known options for a variable yet you want the variables to be easily readable by humans. Alternatively, it would be possible to pass the type of bounds test as a string (for example, “center”, “onScreen”, or “offScreen”), but the enum is a much cleaner way of doing this that isn’t as susceptible to misspelling and that allows for autocomplete while typing.

For more information about enums, see [Appendix B, “Useful Concepts.”](#)

Now, add the following bold methods as part of the `Utils` class. Note how the `BoundsTest` enum is used in the `switch` statement. In the code that follows, there are a few uses of the code continuation character (➡). Remember that this character represents the continuation of the previous line (which was too long to fit the width of the pages in this book).

[Click here to view code image](#)

```
public class Utils : MonoBehaviour {

    //===== Bounds Functions =====\\

    ...

    public static void SetCameraBounds(Camera cam=null) {
        ...
    }

    // Checks to see whether the Bounds bnd are within the camBounds
    public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest
    test = BoundsTest.center) {
```

```

        return( BoundsInBoundsCheck( camBounds, bnd, test ) );
    }

    // Checks to see whether Bounds lilB are within Bounds bigB
    public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
= BoundsTest.onScreen ) {

        // The behavior of this function is different based on the BoundsTest
        // that has been selected.

        // Get the center of lilB
        Vector3 pos = lilB.center;

        // Initialize the offset at [0,0,0]
        Vector3 off = Vector3.zero;

        switch (test) {
        // The center test determines what off (offset) would have to be applied
        // to lilB to move its center back inside bigB
        case BoundsTest.center:
            if ( bigB.Contains( pos ) ) {
                return( Vector3.zero );
            }

            if (pos.x > bigB.max.x) {
                off.x = pos.x - bigB.max.x;
            } else if (pos.x < bigB.min.x) {
                off.x = pos.x - bigB.min.x;
            }
            if (pos.y > bigB.max.y) {
                off.y = pos.y - bigB.max.y;
            } else if (pos.y < bigB.min.y) {
                off.y = pos.y - bigB.min.y;
            }
            if (pos.z > bigB.max.z) {
                off.z = pos.z - bigB.max.z;
            } else if (pos.z < bigB.min.z) {
                off.z = pos.z - bigB.min.z;
            }
            return( off );
        }

        // The onScreen test determines what off would have to be applied to
        // keep all of lilB inside bigB
        case BoundsTest.onScreen:
            if ( bigB.Contains( lilB.min ) && bigB.Contains( lilB.max ) ) {
                return( Vector3.zero );
            }

            if (lilB.max.x > bigB.max.x) {
                off.x = lilB.max.x - bigB.max.x;
            } else if (lilB.min.x < bigB.min.x) {
                off.x = lilB.min.x - bigB.min.x;
            }
            if (lilB.max.y > bigB.max.y) {
                off.y = lilB.max.y - bigB.max.y;
            } else if (lilB.min.y < bigB.min.y) {
                off.y = lilB.min.y - bigB.min.y;
            }
            if (lilB.max.z > bigB.max.z) {
                off.z = lilB.max.z - bigB.max.z;
            } else if (lilB.min.z < bigB.min.z) {
                off.z = lilB.min.z - bigB.min.z;
            }
    }
}

```

```

        return( off );

    // The offScreen test determines what off would need to be applied to
    // move any tiny part of lilB inside of bigB
    case BoundsTest.offScreen:
        bool cMin = bigB.Contains( lilB.min );
        bool cMax = bigB.Contains( lilB.max );
        if ( cMin || cMax ) {
            return( Vector3.zero );
        }

        if ( lilB.min.x > bigB.max.x ) {
            off.x = lilB.min.x - bigB.max.x;
        } else if ( lilB.max.x < bigB.min.x ) {
            off.x = lilB.max.x - bigB.min.x;
        }
        if ( lilB.min.y > bigB.max.y ) {
            off.y = lilB.min.y - bigB.max.y;
        } else if ( lilB.max.y < bigB.min.y ) {
            off.y = lilB.max.y - bigB.min.y;
        }
        if ( lilB.min.z > bigB.max.z ) {
            off.z = lilB.min.z - bigB.max.z;
        } else if ( lilB.max.z < bigB.min.z ) {
            off.z = lilB.max.z - bigB.min.z;
        }
        return( off );
    }

    return( Vector3.zero );
}
}
}

```

These two functions will return a Vector3 value that represents how far the lilB bounds are outside of the bigB bounds (or the camera bounds) according to the type of test passed into BoundsTest test.

Add the bold lines that follow to the Hero class to see how this works:

[Click here to view code image](#)

```

public class Hero : MonoBehaviour {
    ...

    void Update () {
        ...
        transform.position = pos;

        bounds.center = transform.position; // 1

        // Keep the ship constrained to the screen bounds
        Vector3 off = Utils.ScreenBoundsCheck(bounds, BoundsTest.onScreen); // 2
        if ( off != Vector3.zero ) { // 3
            pos -= off;
            transform.position = pos;
        }

        // Rotate the ship to make it feel more dynamic
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
    }
}

```

1. This line moves the center of bounds to line up with the position of `_Hero` after it's been moved by the existing code in the `Update()` method.
2. This line uses `Utils.ScreenBoundsCheck()` to determine whether the ship is off screen.
3. If the `off` ship offset value is not zero, then move the `_Hero` back on screen.

Try altering the line labeled `// 2` to use the other `BoundsTest` options (`BoundsTest.center` and `BoundsTest.offScreen`) and see how they change the behavior of the `ScreenBoundsCheck()`. With `BoundsTest.center`, the ship should stop halfway off screen. With `BoundsTest.offScreen`, you should see that just a tiny sliver of the shield remains on screen.

Adding Some Enemies

The enemies for a game like this were covered a bit in [Chapter 25, “Classes.”](#) There you learned about setting up a superclass for all enemies that can be extended by subclasses. For this game, we extend that further, but first, let's create the artwork.

Enemy Artwork

Because the hero ship has such an angular aesthetic, all the enemies will be constructed of spheres as shown in [Figure 30.6](#).



Figure 30.6 Each of the five enemy ship types

Enemy_0

Create an empty `GameObject` and name it `Enemy_0`. Create a sphere named `Cockpit`, make it a child of `Enemy_0`, and set its transform to `P:[0,0,0] R:[0,0,0] S:[2,2,1]`. Create a second sphere named `Wing`, make it a child of `Enemy_0`, and set its transform to `P:[0,0,0] R:[0,0,0] S:[5,5,0.5]`. Another way of writing this would be:

Enemy_0 (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Cockpit (Sphere)	P:[0,0,0]	R:[0,0,0]	S:[2,2,1]
Wing (Sphere)	P:[0,0,0]	R:[0,0,0]	S:[5,5,0.5]

Follow this formatting to make the remaining four enemies. When finished, they should look like the enemies in [Figure 30.6](#).

Enemy_1

Enemy_1 (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Cockpit (Sphere)	P:[0,0,0]	R:[0,0,0]	S:[2,2,1]
Wing (Sphere)	P:[0,0,0]	R:[0,0,0]	S:[6,4,0.5]

Enemy_2

Enemy_2 (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Cockpit (Sphere)	P:[-1.5,0,0]	R:[0,0,0]	S:[1,3,1]
Sphere	P:[2,0,0]	R:[0,0,0]	S:[2,2,1]
Wing (Sphere)	P:[0,0,0]	R:[0,0,0]	S:[6,4,0.5]

Enemy_3

Enemy_3 (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
CockpitL (Sphere)	P:[-1,0,0]	R:[0,0,0]	S:[1,3,1]
CockpitR (Sphere)	P:[1,0,0]	R:[0,0,0]	S:[1,3,1]
Wing (Sphere)	P:[0,0.5,0]	R:[0,0,0]	S:[5,1,0.5]

Enemy_4

Enemy_4 (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Cockpit (Sphere)	P:[0,1,0]	R:[0,0,0]	S:[1.5,1.5,1.5]
Fuselage (Sphere)	P:[0,1,0]	R:[0,0,0]	S:[2,4,1]
Wing_L (Sphere)	P:[-1.5,0,0]	R:[0,0,-30]	S:[5,1,0.5]
Wing_R (Sphere)	P:[1.5,0,0]	R:[0,0,30]	S:[5,1,0.5]

You must add a Rigidbody component to each of the enemy GameObjects (that is, Enemy_0, Enemy_1, Enemy_2, Enemy_3, and Enemy_4). To add a Rigidbody, complete these steps:

1. Select each enemy in the Hierarchy and choose *Component > Physics > Rigidbody* from the menu bar to add the Rigidbody component.
2. In the Rigidbody component for the enemy, set *Use Gravity* to false.
3. Set *isKinematic* to true.
4. Open the disclosure triangle for *Constraints* and freeze z position and x, y, and z rotation.

Be sure to do this for all five enemies. If a moving GameObject doesn't have a Rigidbody component, the GameObject's collider location will not move with the GameObject, but if a moving GameObject does have a Rigidbody, the colliders of both it and all of its children are updated every frame (which is one of the reasons that you don't need to add a Rigidbody component to any of the children of the enemies).

Drag each of these enemies to the *_Prefabs* folder of the Project pane to create a prefab for each, and then delete all of the enemy instances from the Hierarchy except for Enemy_0.

The Enemy C# Script

Create a new C# script named *Enemy*. Drag the *Enemy* script onto *Enemy_0* in the Project pane. When you click on *Enemy_0* in either the Project or Hierarchy panes, you should see the *Enemy (Script)* component attached. Open the *Enemy* script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;           // Required for Unity
using System.Collections;   // Required for Arrays & other Collections

public class Enemy : MonoBehaviour {
    public float speed = 10f;  // The speed in m/s
```

```

public float      fireRate = 0.3f; // Seconds/shot (Unused)
public float      health = 10;
public int        score = 100; // Points earned for destroying this

public bool _____;

public Bounds     bounds; // The Bounds of this and its children
public Vector3    boundsCenterOffset; // Dist of bounds.center from position

// Update is called once per frame
void Update() {
    Move();
}

public virtual void Move() {
    Vector3 tempPos = pos;
    tempPos.y -= speed * Time.deltaTime;
    pos = tempPos;
}

// This is a Property: A method that acts like a field
public Vector3 pos {
    get {
        return( this.transform.position );
    }
    set {
        this.transform.position = value;
    }
}

}

```

Press Play, and the instance of `Enemy_0` in the scene should move toward the bottom of the screen. However, as it is, this instance will continue off screen and exist until you stop your game. We need to have the enemy destroy itself once it has moved entirely off screen. This is another place where we'll use `Utils.ScreenBoundsTest()`.

In the following code listing, the single line in the `Awake()` method creates a repeated call to the `CheckOffscreen()` method. `InvokeRepeating()` is a built-in Unity function that is used to schedule repeated calls to the same function. The first parameter is the name of the function as a string, the second parameter is the delay (in seconds) before the named function is called the first time, and the last parameter is the delay between each subsequent call. In the `Awake()` function, `CheckOffscreen()` will first be called immediately (0 seconds) after the enemy is instantiated, and then it will be called again every 2 seconds until the object is destroyed.

The `CheckOffscreen()` method first checks to see whether `bounds.size` is `[0,0,0]`. Because `Bounds` is a value type (not a reference type), it is initially a default value of `center:[0,0,0]` `size:[0,0,0]` rather than null. (Value types can never be set to null.) To check to see whether it has been set, we check to see whether the `size` is something other than the default. If the `bounds` really haven't yet been set, `Utils.CombineBoundsOfChildren()` is called to do so. Unlike `_Hero`, it's very possible that the center of the bounds of one of the `Enemy` ships could be offset from the actual center of the `GameObject`, so `Vector3 boundsCenterOffset` is set to the value of this offset (with the `enemy` ships defined previously, this is necessary for `Enemy_4`).

Add the following bold code to the `Enemy` script:

[Click here to view code image](#)

```
public class Enemy : MonoBehaviour {
```

```
...
public Vector3    boundsCenterOffset; // Dist of bounds.center from position

void Awake() {
    InvokeRepeating( "CheckOffscreen", 0f, 2f );
}

...
// This is a Property: A method that acts like a field
public Vector3 pos {
    ...
}

void CheckOffscreen() {
    // If bounds are still their default value...
    if (bounds.size == Vector3.zero) {
        // then set them
        bounds = Utils.CombineBoundsOfChildren(this.gameObject);
        // Also find the diff between bounds.center & transform.position
        boundsCenterOffset = bounds.center - transform.position;
    }

    // Every time, update the bounds to the current position
    bounds.center = transform.position + boundsCenterOffset;
    // Check to see whether the bounds are completely offscreen
    Vector3 off = Utils.ScreenBoundsCheck( bounds, BoundsTest.offScreen );
    if ( off != Vector3.zero ) {
        // If this enemy has gone off the bottom edge of the screen
        if (off.y < 0) {
            // then destroy it
            Destroy( this.gameObject );
        }
    }
}
```

Now, when you play the scene, you should see that the `Enemy_0` ship moves down the screen, then off screen, and within a couple of seconds of moving off screen, it is destroyed.

Spawning Enemies at Random

With all of this in place, it's now possible to instantiate a number of `Enemy_0s` randomly. Create a new C# script called *Main* and attach it to `MainCamera`. Enter the following code:

[Click here to view code image](#)

```
using UnityEngine; // Required for Unity
using System.Collections; // Required for Arrays & other Collections
using System.Collections.Generic; // Required to use Lists or Dictionaries

public class Main : MonoBehaviour {
    static public Main S;

    public GameObject[] prefabEnemies;
    public float enemySpawnPerSecond = 0.5f; // # Enemies/second
    public float enemySpawnPadding = 1.5f; // Padding for position

    public bool _____;

    public float enemySpawnRate; // Delay between Enemy spawns

    void Awake() {
```

```

S = this;
// Set Utils.camBounds
Utils.SetCameraBounds(this.camera);
// 0.5 enemies/second = enemySpawnRate of 2
enemySpawnRate = 1f/enemySpawnPerSecond; // 1
// Invoke call SpawnEnemy() once after a 2 second delay
Invoke( "SpawnEnemy", enemySpawnRate ); // 2
}

public void SpawnEnemy() {
    // Pick a random Enemy prefab to instantiate
    int ndx = Random.Range(0, prefabEnemies.Length);
    GameObject go = Instantiate( prefabEnemies[ ndx ] ) as GameObject;
    // Position the Enemy above the screen with a random x position
    Vector3 pos = Vector3.zero;
    float xMin = Utils.camBounds.min.x+enemySpawnPadding;
    float xMax = Utils.camBounds.max.x-enemySpawnPadding;
    pos.x = Random.Range( xMin, xMax );
    pos.y = Utils.camBounds.max.y + enemySpawnPadding;
    go.transform.position = pos;
    // Call SpawnEnemy() again in a couple of seconds
    Invoke( "SpawnEnemy", enemySpawnRate ); // 3
}
}

```

1. The public field to set the spawn rate of enemies is `enemySpawnPerSecond`, storing the number of enemies that will spawn every second. By default, it is set to 0.5f (or half an enemy every second). The line here converts this into the number of seconds of delay between each enemy spawn (2 seconds in this case) and assigns that value to `enemySpawnRate`.
2. The `Invoke()` function works much like `InvokeRepeating()`, except that it only calls the invoked function once.
3. The reason that `Invoke()` is used instead of `InvokeRepeating()` is that we want to be able to dynamically adjust the amount of time between each enemy spawn. Once `InvokeRepeating()` is called, the invoked function is always called at the rate specified. Adding an `Invoke()` call at the end of `SpawnEnemy()` allows the game to adjust `enemySpawnRate` on the fly and have it affect how frequently `SpawnEnemy()` is called.

Once you've typed this code and saved the file, switch back to Unity and follow these instructions:

1. Delete the instance of `Enemy_0` from the Hierarchy (leaving the prefab in the Project pane alone, of course).
2. Select `_MainCamera` in the Hierarchy.
3. Open the disclosure triangle next to `prefabEnemies` in the *Main (Script)* component of `_MainCamera` and set the *Size* of `prefabEnemies` to 1.
4. Drag `Enemy_0` from the Project pane into *Element 0* of the `prefabEnemies` array.
5. *Save your scene!* Have you been remembering? If you didn't save your scene after creating all of those enemies, you really should have. There are all sorts of things beyond your control that could cause Unity to crash, and you really don't want to have to redo work. Getting into a habit of saving your scene frequently can save you a ton of wasted time and sorrow as a developer.

Play your scene. You should now see an `Enemy_0` spawn about once every 2 seconds, travel down to the bottom of the screen, and then disappear within a few seconds of exiting the bottom of the screen.

However, right now, when the `_Hero` collides with an enemy, nothing happens. This needs to be fixed,

and to do so, we're going to have to look at layers.

Setting Tags, Layers, and Physics

As was presented in [Chapter 28](#), “[Prototype 1: Apple Picker](#),” one of the things that layers control in Unity is which objects may or may not collide with each other. First, let’s think about the Space SHMUP prototype. In this game, several different types of GameObjects could be placed on different layers and interact with each other in different ways:

- **Hero:** The `_Hero` ship should collide with enemies, enemy projectiles, and power-ups but should not collide with hero projectiles.
- **ProjectileHero:** Projectiles fired by `_Hero` should only collide with enemies.
- **Enemy:** Enemies should collide with `_Hero` and hero projectiles but not with power-ups.
- **ProjectileEnemy:** Projectiles fired by enemies should only collide with `_Hero`.
- **PowerUp:** Power-ups should only collide with `_Hero`.

To create these five layers, complete these steps:

1. Open the Tags and Layers Manager in the Inspector pane (*Edit > Project Settings > Tags and Layers*). Tags and layers are different from each other, but both are set in the Tags and Layers Manager.
2. Open the disclosure triangle next to Tags. Set the *Size* of Tags to 7 and enter the tags shown in the left image of [Figure 30.7](#). Note that in the middle of your typing the name of *Tags Element 5*, `PowerUpBox`, you may receive a console message (“Default GameObject Tag: PowerUp already registered”), which you can safely ignore.

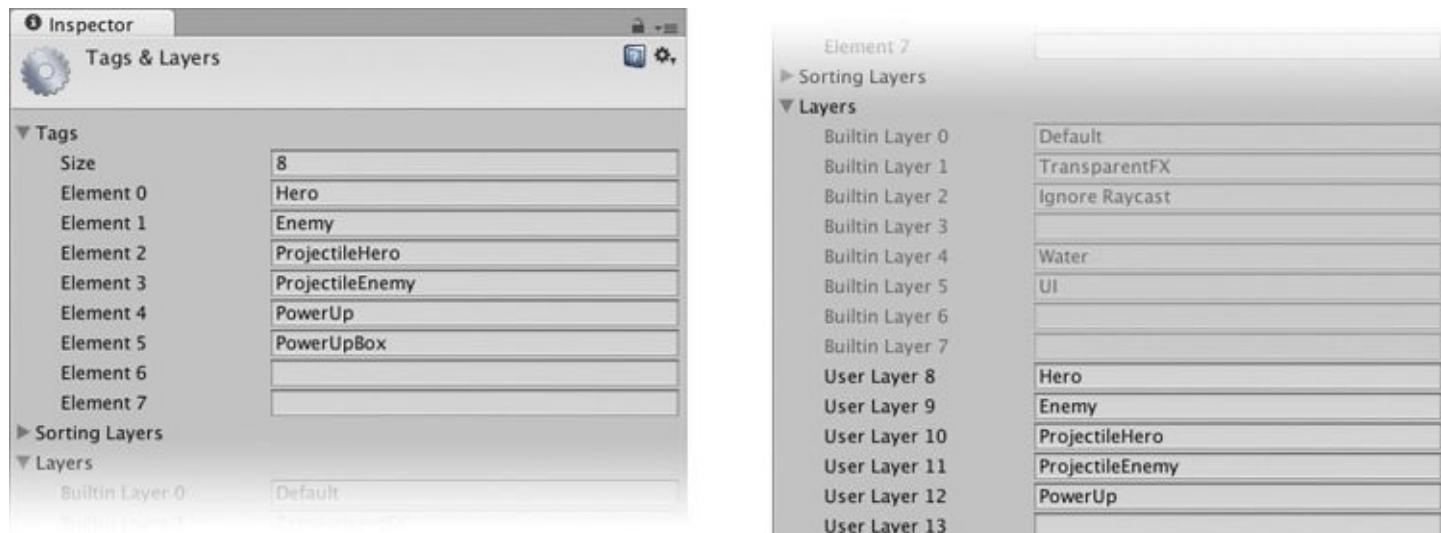


Figure 30.7 TagManager showing tags and layer names for this prototype

3. Open the disclosure triangle next to *Layers*. Starting with *User Layer 8*, enter the layer names shown in the right image of [Figure 30.7](#). *Builtin Layers 0–7* are reserved by Unity, but you can set the names of *User Layers 8–31*.
4. Open the PhysicsManager (*Edit > Project Settings > Physics*) and set it as shown in [Figure 30.8](#).

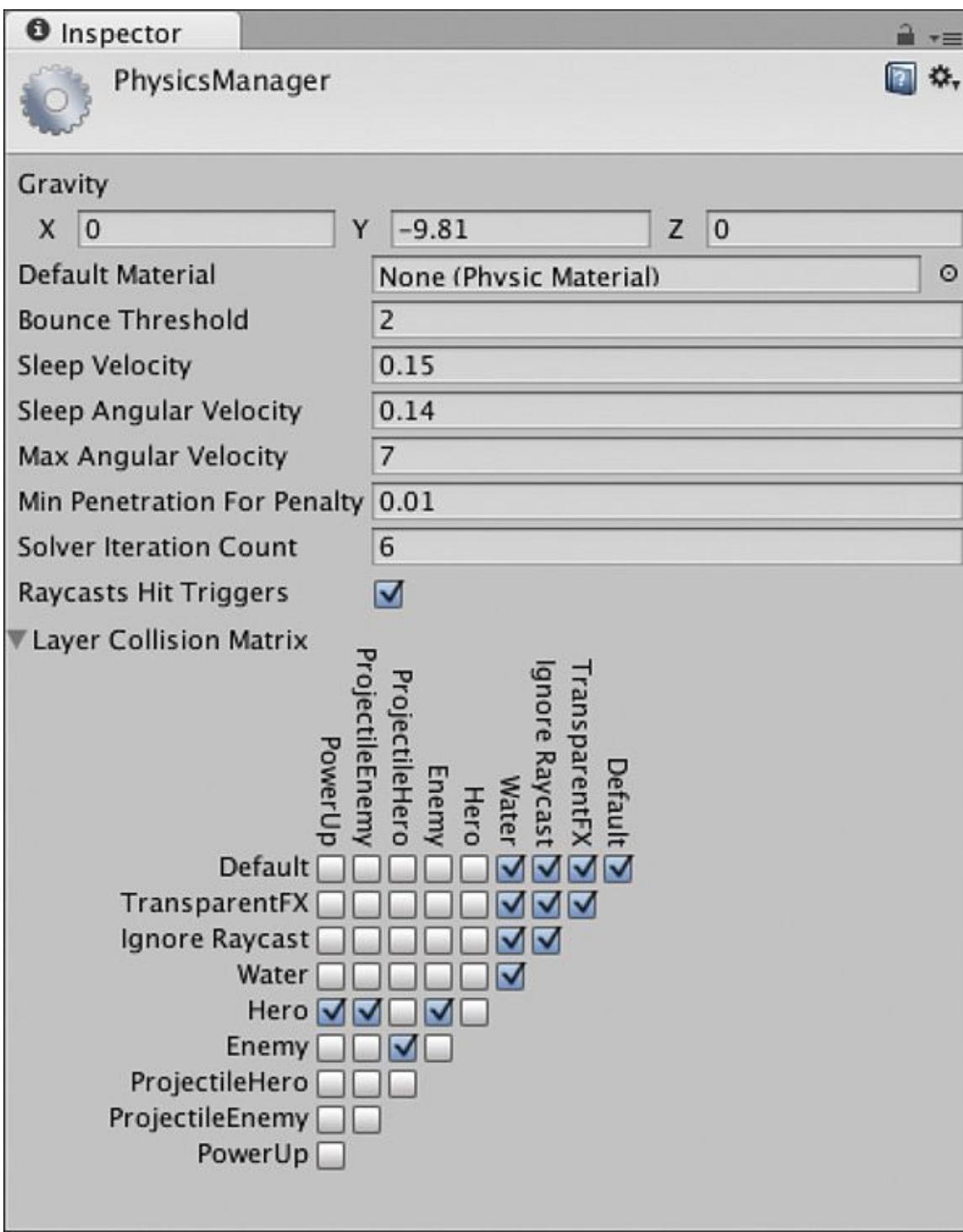


Figure 30.8 PhysicsManager with proper settings for this prototype

Note

As of Unity 4.3, there are settings for both Physics and Physics2D. In this chapter, you should be setting Physics (the standard 3D PhysX physics library), not Physics2D.

The grid at the bottom of the PhysicsManager sets which layers collide with each other. If there is a check, objects in the two layers are able to collide, if there is no check, they won't. Removing checks can speed the execution of your game because it will test fewer objects versus each other for collision. As you can see in [Figure 30.8](#), the layers and collision we've chosen achieve the collision behavior we specified earlier.

Assign the Proper Layers to GameObjects

Now that the layers have been defined, you must assign the GameObjects you've created to the correct layer, as follows:

1. Select `_Hero` in the Hierarchy and choose `Hero` from the Layer pop-up menu in the Inspector. When Unity asks if you'd like to also assign the children of `_Hero` to this new layer, choose `Yes, change children`.
2. Set the tag of `_Hero` to `Hero` using the Tag pop-up menu in the Inspector. You do not need to change the tags of the children of `_Hero`.
3. Select each of the `Enemy` prefabs in the Project pane and set each to the `Enemy` layer. When asked, elect to change the layer of their children as well.
4. Also set the tag of each `Enemy` prefab to `Enemy`. You do not need to set the tags of the children of each enemy.

Making the Enemies Damage the Player

Now that the enemies and hero have colliding layers, we need to make them react to the collisions.

Open the disclosure triangle next to `_Hero` in the Hierarchy and select its child `Shield`. In the Inspector, set the Sphere Collider of `Shield` to be a trigger (check the box next to `Is Trigger`). We don't need things to bounce off of `Shield`; we just need to know when they've hit.

Add the following bolded method to the end of the `Hero` C# script:

[Click here to view code image](#)

```
public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        print("Triggered: "+other.gameObject.name);
    }
}
```

Play the scene and try running into some enemies. You will see that you get a trigger event for the children GameObjects of the `Enemy` (for example, `Cockpit` and `Wing`) but not for the `Enemy` itself. Add this pair of methods to the `Utils` class to enable you to move up the `transform.parent` tree to find the parent with a tag (in this case, `Enemy`):

[Click here to view code image](#)

```
public class Utils : MonoBehaviour {

    //===== Bounds Functions =====\\

    ...
    // Checks to see whether Bounds lilB are within Bounds bigB
    public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
    = BoundsTest.onScreen ) {
        ...
    }

    //===== Transform Functions =====\\
```

```

// This function will iteratively climb up the transform.parent tree
// until it either finds a parent with a tag != "Untagged" or no parent
public static GameObject FindTaggedParent(GameObject go) { // 1
    // If this gameObject has a tag
    if (go.tag != "Untagged") { // 2
        // then return this gameObject
        return(go);
    }
    // If there is no parent of this Transform
    if (go.transform.parent == null) { // 3
        // We've reached the top of the hierarchy with no interesting tag
        // So return null
        return( null );
    }
    // Otherwise, recursively climb up the tree
    return( FindTaggedParent( go.transform.parent.gameObject ) ); // 4
}
// This version of the function handles things if a Transform is passed in
public static GameObject FindTaggedParent(Transform t) { // 5
    return( FindTaggedParent( t.gameObject ) );
}
}

```

1. `FindTaggedParent()` searches for a `GameObject` that is in the transform hierarchy above `GameObject go` and is tagged (that is, it has a tag other than the default tag `Untagged`).
2. If `GameObject go` has a tag, `go` is returned.
3. If `go.transform.parent` is `null`, then `GameObject go` has no parent. This means that neither the original `GameObject` nor any of its parents had a tag, so `null` is returned.
4. Because `go.transform.parent` is not `null`, `Utils.FindTaggedParent()` is called recursively with the parent `GameObject` of `go`.
5. This is an overload of the function `FindTaggedParent()` that takes a `Transform` as its initial argument rather than the `GameObject` required by the other version of `FindTaggedParent()`.

Next, modify the `OnTriggerEnter()` method in the `Hero` class as follows to take advantage of the `Utils.FindTaggedParent()` method:

[Click here to view code image](#)

```

public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        // Find the tag of other.gameObject or its parent GameObjects
        GameObject go = Utils.FindTaggedParent(other.gameObject);
        // If there is a parent with a tag
        if (go != null) {
            // Announce it
            print("Triggered: "+go.name);
        } else {
            // Otherwise announce the original other.gameObject
            print("Triggered: "+other.gameObject.name); // Move this line here!
        }
    }
}

```

```
}
```

Now when you play the scene and run the ship into enemies, you should see that `OnTriggerEnter()` announces it has hit *Enemy_0(Clone)*, an instance of *Enemy_0*.

Tip

Iterative Code Development When prototyping on your own, this kind of announcement test is something that you will do often to test whether the code you've written is working properly. I find that it is much better to do small tests along the way like this than to work on code for hours only to find at the end that something is causing a bug. Testing incrementally makes things *a lot* easier to debug because you know that you've only made slight changes since the last test that worked, so it's easier to find the place where you added a bug.

Another key element of this approach is using the debugger. Throughout the authoring of this book, any time I ran into something that worked a little differently than I expected, I used the debugger to understand what was happening. If you don't remember how to use the MonoDevelop debugger, I highly recommend rereading [Chapter 24, “Debugging.”](#)

Using the debugger effectively is often the difference between solving your code problems and just staring at pages of code blankly for several hours. Try putting a debug breakpoint into the `OnTriggerEnter()` method you just modified and watching how code is called and variables change. The recursive calling of `Utils.FindTaggedParent()` in particular should be interesting.

Iterative code development has the same strengths as the iterative process of design, and it is the key to the agile development methodology discussed in [Chapter 27, “The Agile Mentality.”](#)

Next, modify the `OnTriggerEnter()` method of the *Hero* class to make a collision with an enemy decrease the player's shield by 1 and destroy the *Enemy* that was hit. It's also very important to make sure that the same parent *GameObject* doesn't trigger the *Shield* collider twice (which could happen with very fast-moving objects if two child colliders of one object hit the *Shield* trigger in the same frame).

[Click here to view code image](#)

```
public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    // This variable holds a reference to the last triggering GameObject
    public GameObject lastTriggerGo = null; // 1

    void OnTriggerEnter(Collider other) {
        ...
        if (go != null) {
            // Make sure it's not the same triggering go as last time
            if (go == lastTriggerGo) { // 2
                return;
            }
            lastTriggerGo = go; // 3
        }
    }
}
```

```

    if (go.tag == "Enemy") {
        // If the shield was triggered by an enemy
        // Decrease the level of the shield by 1
        shieldLevel--;
        // Destroy the enemy
        Destroy(go);                                // 4
    } else {
        print("Triggered: "+go.name);              // Move this line here!
    }
} else {
    ...
}

```

1. This field holds a reference to the last GameObject that triggered Shield collider. It is initially set to null. Though we usually declare fields at the top of the class, they can actually be declared anywhere throughout the class, as we have done with this line.
2. If `lastTriggerGo` is the same as `go` (the current triggering GameObject), this collision is ignored as a duplicate, which can happen if two children GameObjects of the same Enemy trigger the Shield collider at the same time (that is, in the same single frame).
3. Assign `go` to `lastTriggerGo` so that it is updated the next time `OnTriggerEnter()` is called.
4. `go`, the enemy GameObject, is destroyed by hitting the shield. Because the actual GameObject `go` that we're testing is the Enemy GameObject found by `Utils.FindTaggedParent()`, this will delete the entire Enemy (and by extension, all of its children), and not just one of the Enemy's child GameObjects.

Play the scene and try running into some ships. After running into more than a few, you may notice a strange shield behavior. The shield will loop back around to full strength after being completely drained. What do you think is causing this? Try selecting `_Hero` in the Hierarchy while playing the scene to see what's happening to the `shieldLevel` field.

Because there is no bottom limit to `shieldLevel`, it continues past 0 into negative territory. The Shield C# script then translates this into negative x offset values for Mat Shield, and because the material's texture is set to loop, it looks like the shield is returning to full strength.

To fix this, we will convert `shieldLevel` to a property that insulates and limits a new private field named `_shieldLevel`. The `shieldLevel` property will watch the value of the `_shieldLevel` field and make sure that `_shieldLevel` never gets above 4 and that the ship is destroyed if `_shieldLevel` ever drops below 0. An insulated field like `_shieldLevel` should be set to private because it does not need to be accessed by other classes; however, in Unity, private fields are not viewable in the Inspector. To remedy this, the line `[SerializeField]` is added above the declaration of `_shieldLevel` to instruct Unity to show it in the Inspector even though it is a private field. Properties are never visible in the Inspector, even if they're public.

First, change the name of the public variable `shieldLevel` to `_shieldLevel` near the top of the `Hero` class, set it to private, and add the `[SerializeField]` line:

[Click here to view code image](#)

```

// Ship status information
[SerializeField]
private float      _shieldLevel = 1;           // Add the underscore!

```

Next, add the `shieldLevel` property to the end of the `Hero` class.

[Click here to view code image](#)

```
public class Hero : MonoBehaviour {  
    ...  
    void OnTriggerEnter(Collider other) {  
        ...  
    }  
  
    public float shieldLevel {  
        get {  
            return( _shieldLevel );  
        }  
        set {  
            _shieldLevel = Mathf.Min( value, 4 );  
            // If the shield is going to be set to less than zero  
            if (value < 0) {  
                Destroy(this.gameObject);  
            }  
        }  
    }  
}
```

1. The `get` clause just returns the value of `_shieldLevel`.
2. `Mathf.Min()` ensures that `_shieldLevel` is never set to a number higher than 4.
3. If the value passed into the `set` clause is less than 0, `_Hero` is destroyed.

Restarting the Game

From your testing, you can see that the game gets exceedingly boring once `_Hero` has been destroyed. We'll now modify both the `Hero` and `Main` classes to call a method when `_Hero` is destroyed that waits for 2 seconds and then restarts the game.

Add a `gameRestartDelay` field to the top of the `Hero` class:

[Click here to view code image](#)

```
static public Hero      S; // Singleton  
  
public float           gameRestartDelay = 2f;  
  
// These fields control the movement of the ship
```

Then add the following lines to the `shieldLevel` property definition in the `Hero` class:

[Click here to view code image](#)

```
if (value < 0) {  
    Destroy(this.gameObject);  
    // Tell Main.S to restart the game after a delay  
    Main.S.DelayedRestart( gameRestartDelay );  
}
```

Finally, add the following methods to the `Main` class to make this work.

[Click here to view code image](#)

```
public class Main : MonoBehaviour {  
    ...
```

```

public void SpawnEnemy() {
    ...
}

public void DelayedRestart( float delay ) {
    // Invoke the Restart() method in delay seconds
    Invoke("Restart", delay);
}

public void Restart() {
    // Reload _Scene_0 to restart the game
    Application.LoadLevel("_Scene_0");
}

}

```

Now, once the player ship has been destroyed, the game waits a couple of seconds and then restarts by reloading the scene.

Shooting (Finally)

Now that the enemy ships can hurt the player, it's time to give `_Hero` a way to fight back.

Artwork

Create an empty GameObject, name it *Weapon*, and give it the following structure and children:

Weapon (Empty)	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Barrel (Cube)	P:[0,0.5,0]	R:[0,0,0]	S:[0.25,1,0.1]
Collar (Cube)	P:[0,1,0]	R:[0,0,0]	S:[0.375,0.5,0.2]

Remove the Collider component from both Barrel and Collar by selecting them individually and then right-clicking on the name of the Box Collider component and choosing *Remove Component* from the pop-up menu. You can also click the gear to the right of the Box Collider name to get the same menu.

Now, create a new material named *Mat Collar*. Drag this material on to Collar to assign it. In the Inspector, choose *ProtoTools > UnlitAlpha* from the Shader pop-up menu. The Collar should now be a bright white (see [Figure 30.9](#)).

Now, create a new C# script named *Weapon* and drag it onto the Weapon GameObject in the Hierarchy. Then drag the Weapon GameObject into the `_Prefabs` folder in the Project pane to make it a prefab. Make the Weapon instance in the Hierarchy a child of `_Hero` and set its position to [0,2,0]. This should place the Weapon on the nose of the `_Hero` ship, as is shown in [Figure 30.9](#).

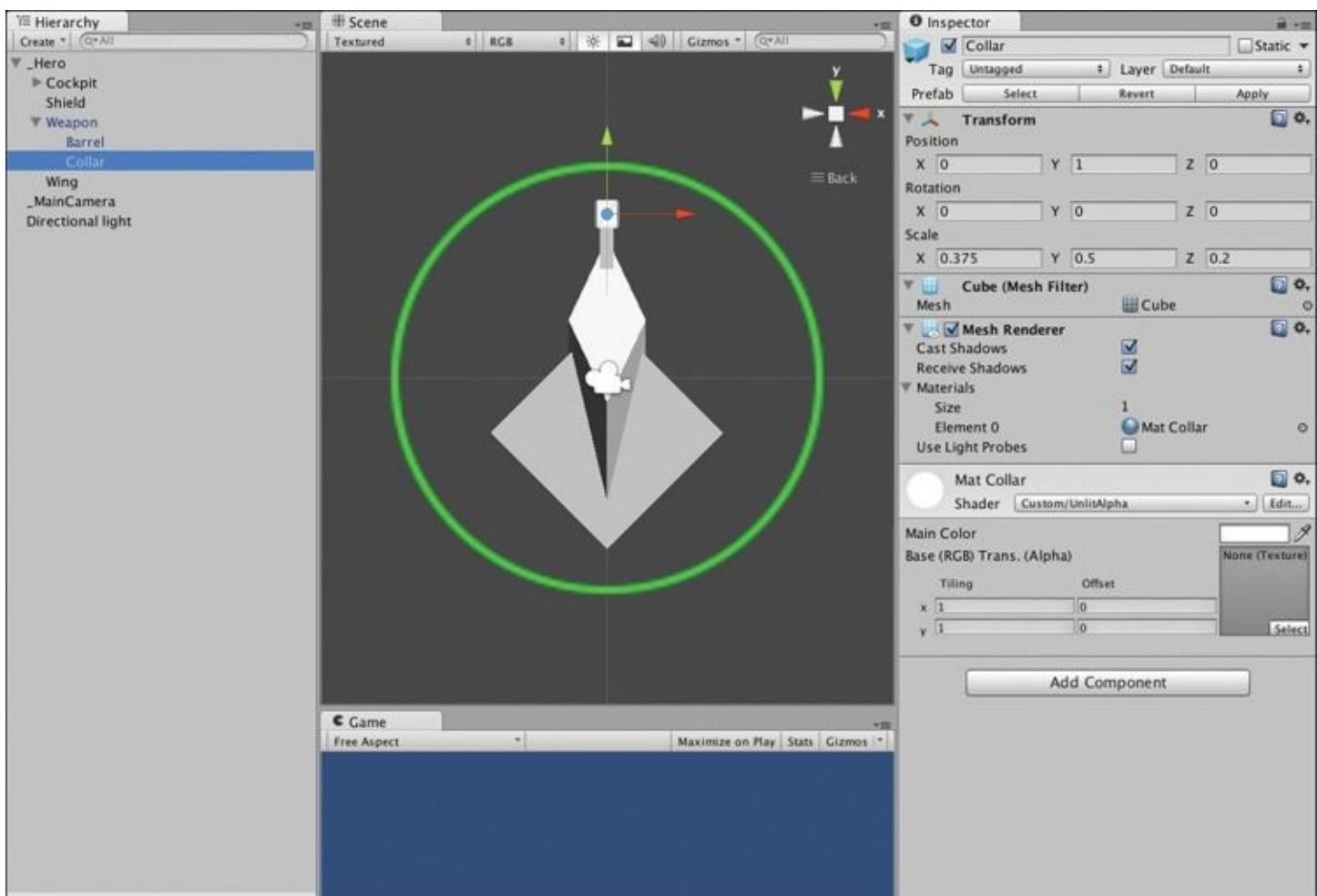


Figure 30.9 Weapon with the Collar selected and proper material and shader selected

Save your scene! Are you remembering to save constantly?

Next, create a cube named *ProjectileHero* in the Hierarchy as follows:

ProjectileHero (Cube) P:[10,0,0] R:[0,0,0] S:[0.25,1,0.5]

Set both the tag and layer of ProjectileHero to *ProjectileHero*. Create a new material named *Mat Projectile*, give it the *ProtoTools > UnlitAlpha* shader, and assign it to the ProjectileHero GameObject. Add a Rigidbody component to the ProjectileHero GameObject with the settings shown in [Figure 30.10](#). (The transform.position of ProjectileHero doesn't actually matter because it will be a prefab that is positioned via code.) Create a new C# script named *Projectile* and drag it onto ProjectileHero. We'll edit the script later.

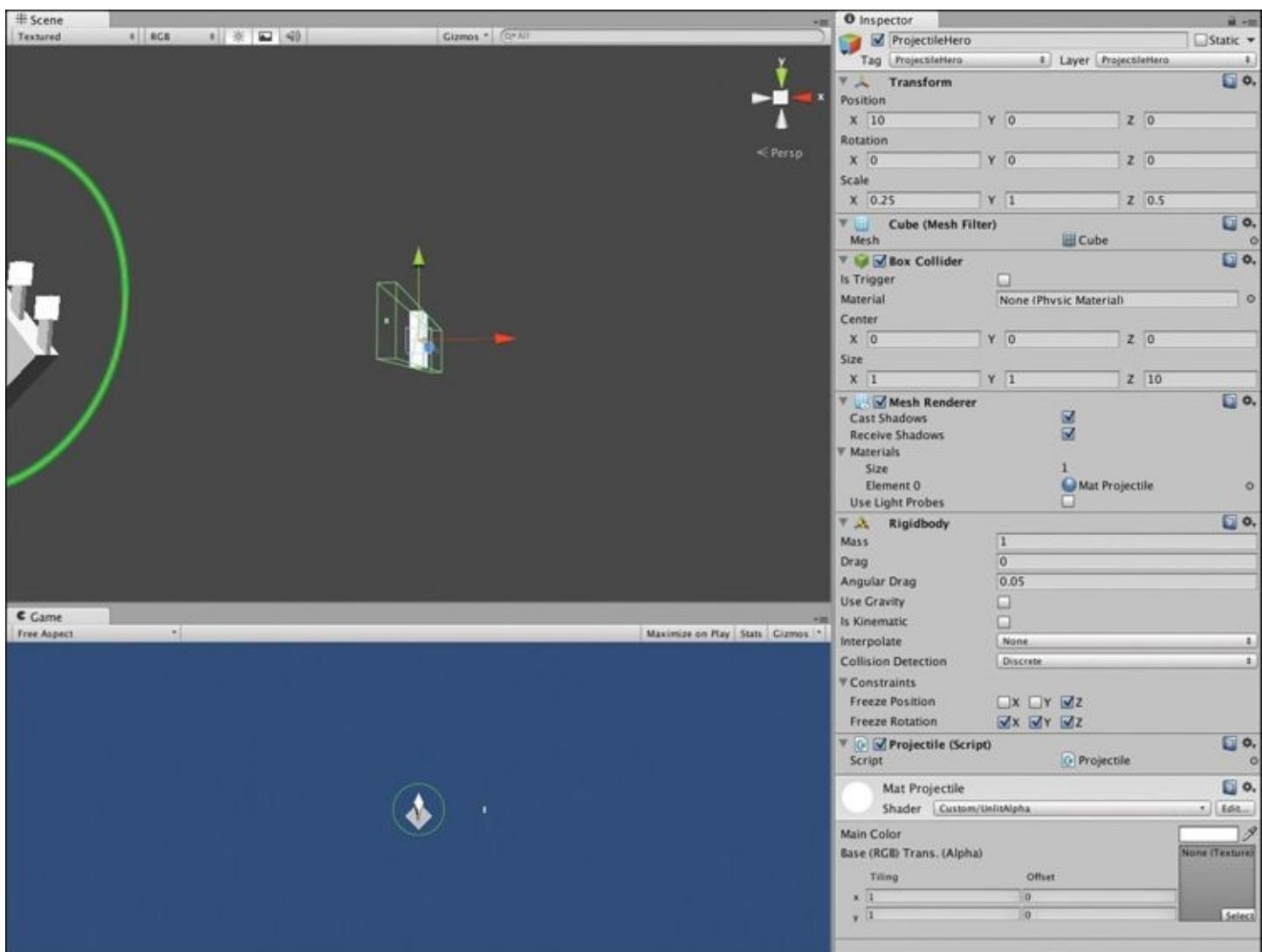


Figure 30.10 ProjectileHero with the proper settings showing the large `Size.z` of the Box Collider

In the Box Collider component of the ProjectileHero GameObject, set `Size.z` to 10. This will make sure that the projectile is able to hit anything that is slightly off of the $z=0$ plane.

Save your scene.

Drag ProjectileHero into the `_Prefabs` folder in the Project pane to make it a prefab and delete the instance remaining in the Hierarchy.

Save your scene. As I've said, you want to save as often as you can.

The Serializable WeaponDefinition Class

Open the Weapon script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

// This is an enum of the various possible weapon types
// It also includes a "shield" type to allow a shield power-up
// Items marked [NI] below are Not Implemented in this book
public enum WeaponType {
    none,           // The default / no weapon
    blaster,        // A simple blaster
    spread,         // Two shots simultaneously
    phaser,         // Shots that move in waves [NI]
    missile,        // Homing missiles [NI]
```

```

        laser,           // Damage over time [NI]
        shield          // Raise shieldLevel
    }

    // The WeaponDefinition class allows you to set the properties
    // of a specific weapon in the Inspector. Main has an array
    // of WeaponDefinitions that makes this possible.
    // [System.Serializable] tells Unity to try to view WeaponDefinition
    // in the Inspector pane. It doesn't work for everything, but it
    // will work for simple classes like this!
    [System.Serializable]
    public class WeaponDefinition {
        public WeaponType type = WeaponType.none;
        public string letter;           // The letter to show on the power-up
        public Color color = Color.white; // Color of Collar & power-up
        public GameObject projectilePrefab; // Prefab for projectiles
        public Color projectileColor = Color.white;
        public float damageOnHit = 0;    // Amount of damage caused
        public float continuousDamage = 0; // Damage per second (Laser)
        public float delayBetweenShots = 0;
        public float velocity = 20;      // Speed of projectiles
    }

    // Note: Weapon prefabs, colors, and so on. are set in the class Main.

    public class Weapon : MonoBehaviour {
        // The Weapon class will be filled in later.
    }
}

```

As described in the code comments, the enum `WeaponType` defines all the possible weapon types and power-up types. `WeaponDefinition` is a class that combines a `WeaponType` with several other fields that will be useful for defining each weapon. Add the following code to the `Main` class:

[Click here to view code image](#)

```

public class Main : MonoBehaviour {
    ...
    public float enemySpawnPadding = 1.5f; // Padding for position
    public WeaponDefinition[] weaponDefinitions;

    public bool _____;

    public WeaponType[] activeWeaponTypes;
    public float enemySpawnRate; // Delay between Enemies

    void Awake() { ... }

    void Start() {
        activeWeaponTypes = new WeaponType[weaponDefinitions.Length];
        for ( int i=0; i<weaponDefinitions.Length; i++ ) {
            activeWeaponTypes[i] = weaponDefinitions[i].type;
        }
    }
    ...
}

```

Save this and then select `_MainCamera` in the Hierarchy. You should now see a `weaponDefinitions` array in the `Main` (Script) component Inspector. Click the disclosure triangle next to it and set the *Size* of the array to 3. Enter settings for the three `WeaponDefinitions` as shown in [Figure 30.11](#). The colors don't have to be exactly right, but it is important that the alpha value of each color is set to fully opaque (which appears as a white bar beneath the color swatch).

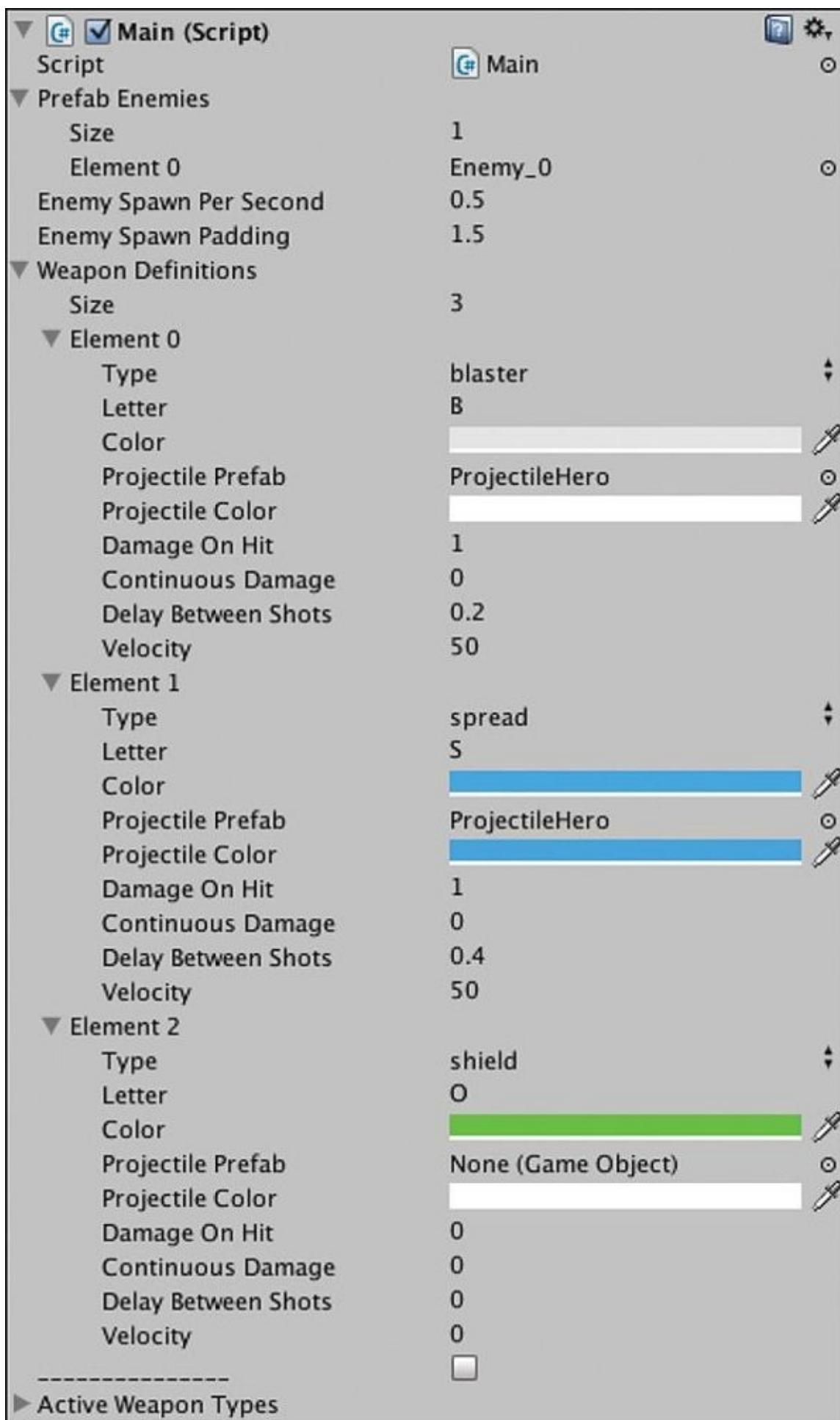


Figure 30.11 Settings for the WeaponDefinitions of blaster, spread, and shield on Main

Warning

Colors Sometimes Default to an Invisible Alpha When you create a serializable class like `WeaponDefinition` that includes color fields, the alpha values of those colors will default to 0 (i.e., invisible). To fix this, make sure that the white bar under each of your color definitions is actually white (and not black). If you click on the color itself, you will be presented with four values to set (R, G, B, and A). Make sure that A is set to 255 (i.e., fully opaque) or your shots will be invisible.

If you are using OS X and have chosen to use the OS X color picker in Unity instead of the default one, the A value is set by the Opacity slider at the bottom of the color picker window (which should be set to 100% for these colors).

A Generic Dictionary for `WeaponDefinitions`

Now, open the `Main` script in MonoDevelop and enter the following bold code. This code uses a `Dictionary`, which is another type of generic collection like `List`. Dictionaries have a *key* type and *value* type, and the key is used to retrieve the value. Here, the Dictionary has the enum `WeaponType` as the key and the class `WeaponDefinition` as the value. We will create the static public `W_DEFS` dictionary to hold the `WeaponDefinition` information that we just entered into the array in the `Main` (Script) Inspector. Unfortunately, Dictionaries do not appear in the Inspector, or we would have just used one from the start. Instead, the `W_DEFS` Dictionary is defined in the `Awake()` method of `Main` and then used by the static function `Main.GetWeaponDefinition()`.

[Click here to view code image](#)

```
public class Main : MonoBehaviour {
    static public Main S;
    static public Dictionary<WeaponType, WeaponDefinition> W_DEFS;
    ...
    void Awake() {
        ...
        Invoke( "SpawnEnemy", enemySpawnRate );

        // A generic Dictionary with WeaponType as the key
        W_DEFS = new Dictionary<WeaponType, WeaponDefinition>();
        foreach( WeaponDefinition def in weaponDefinitions ) {
            W_DEFS[def.type] = def;
        }
    }

    static public WeaponDefinition GetWeaponDefinition( WeaponType wt ) {
        // Check to make sure that the key exists in the Dictionary
        // Attempting to retrieve a key that didn't exist, would throw an error,
        // so the following if statement is important.
        if (W_DEFS.ContainsKey(wt)) {
            return( W_DEFS[wt]);
        }
        // This will return a definition for WeaponType.none,
        // which means it has failed to find the WeaponDefinition
        return( new WeaponDefinition() );
    }

    void Start() { ... }
}
```

Now, open the `Projectile` class in MonoDevelop and enter this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {
    [SerializeField]
    private WeaponType _type;

    // This public property masks the field _type & takes action when it is set
    public WeaponType type {
        get {
            return( _type );
        }
        set {
            SetType( value );
        }
    }

    void Awake() {
        // Test to see whether this has passed off screen every 2 seconds
        InvokeRepeating( "CheckOffscreen", 2f, 2f );
    }

    public void SetType( WeaponType eType ) {
        // Set the _type
        _type = eType;
        WeaponDefinition def = Main.GetWeaponDefinition( _type );
        renderer.material.color = def.projectileColor;
    }

    void CheckOffscreen() {
        if ( Utils.ScreenBoundsCheck( collider.bounds, BoundsTest.offScreen ) != Vector3.zero ) {
            Destroy( this.gameObject );
        }
    }
}
```

Whenever the `type` property of this `Projectile` is set, `SetType()` will be called, and the `Projectile` will automatically set its own color based on the `WeaponDefinitions` in `Main`.

Using a Function Delegate to Fire

Before continuing, read the “[Function Delegates](#)” section of [Appendix B](#).

In this game prototype, the `Hero` class will have a function delegate `fireDelegate` that is called to fire all weapons, and each `Weapon` attached to it will add its individual `Fire()` target method to `fireDelegate`.

Add the following bold code to the `Hero` class:

[Click here to view code image](#)

```
public class Hero : MonoBehaviour {
    ...
    public Bounds bounds;

    // Declare a new delegate type WeaponFireDelegate
    public delegate void WeaponFireDelegate();
    // Create a WeaponFireDelegate field named fireDelegate.
    public WeaponFireDelegate fireDelegate;
```

```

void Awake() {
    ...
}

void Update () {
    ...
    // Rotate the ship to make it feel more dynamic
    transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);

    // Use the fireDelegate to fire Weapons
    // First, make sure the Axis("Jump") button is pressed
    // Then ensure that fireDelegate isn't null to avoid an error
    if (Input.GetAxis("Jump") == 1 && fireDelegate != null) {           // 1
        fireDelegate();
    }
}
...
}

```

1. If `fireDelegate` is called when it has no methods assigned to it, it will throw an error. To avoid this, `fireDelegate != null` is tested to see whether it is null before calling it.

Open the Weapon C# script in MonoDevelop and add the following code:

[Click here to view code image](#)

```

public class Weapon : MonoBehaviour {
    static public Transform    PROJECTILE_ANCHOR;

    public bool _____;
    [SerializeField]
    private WeaponType      _type = WeaponType.none;
    public WeaponDefinition def;
    public GameObject        collar;
    public float             lastShot; // Time last shot was fired

    void Start() {
        collar = transform.Find("Collar").gameObject;
        // Call SetType() properly for the default _type
        SetType( _type );

        if (PROJECTILE_ANCHOR == null) {
            GameObject go = new GameObject("_Projectile_Anchor");
            PROJECTILE_ANCHOR = go.transform;
        }
        // Find the fireDelegate of the parent
        GameObject parentGO = transform.parent.gameObject;
        if (parentGO.tag == "Hero") {
            Hero.S.fireDelegate += Fire;
        }
    }

    public WeaponType type {
        get {      return( _type );      }
        set {      SetType( value );    }
    }

    public void SetType( WeaponType wt ) {
        _type = wt;
        if (type == WeaponType.none) {
            this.gameObject.SetActive(false);
            return;
        } else {

```

```

        this.gameObject.SetActive(true);
    }
    def = Main.GetWeaponDefinition(_type);
    collar.renderer.material.color = def.color;
    lastShot = 0; // You can always fire immediately after _type is set.
}

public void Fire() {
    // If this.gameObject is inactive, return
    if (!gameObject.activeInHierarchy) return;
    // If it hasn't been enough time between shots, return
    if (Time.time - lastShot < def.delayBetweenShots) {
        return;
    }
    Projectile p;
    switch (type) {
        case WeaponType.blaster:
            p = MakeProjectile();
            p.rigidbody.velocity = Vector3.up * def.velocity;
            break;

        case WeaponType.spread:
            p = MakeProjectile();
            p.rigidbody.velocity = Vector3.up * def.velocity;
            p = MakeProjectile();
            p.rigidbody.velocity = new Vector3( -.2f, 0.9f, 0 ) * def.velocity;
            p = MakeProjectile();
            p.rigidbody.velocity = new Vector3( .2f, 0.9f, 0 ) * def.velocity;
            break;
    }
}

public Projectile MakeProjectile() {
    GameObject go = Instantiate( def.projectilePrefab ) as GameObject;
    if ( transform.parent.gameObject.tag == "Hero" ) {
        go.tag = "ProjectileHero";
        go.layer = LayerMask.NameToLayer("ProjectileHero");
    } else {
        go.tag = "ProjectileEnemy";
        go.layer = LayerMask.NameToLayer("ProjectileEnemy");
    }
    go.transform.position = collar.transform.position;
    go.transform.parent = PROJECTILE_ANCHOR;
    Projectile p = go.GetComponent<Projectile>();
    p.type = type;
    lastShot = Time.time;
    return( p );
}

}

```

Most of this code should make sense to you. Note that the various kinds of projectiles and weapons are handled with a switch statement inside of the `Fire()` method.

Now, it's important to make projectiles actually damage enemies. Open the `Enemy` C# script in MonoDevelop and add the following `OnCollisionEnter()` method:

[Click here to view code image](#)

```

public class Enemy : MonoBehaviour {
    ...
    void CheckOffscreen() {
        ...

```

```

        }

        void OnCollisionEnter( Collision coll ) {
            GameObject other = coll.gameObject;
            switch (other.tag) {
                case "ProjectileHero":
                    Projectile p = other.GetComponent<Projectile>();
                    // Enemies don't take damage unless they're onscreen
                    // This stops the player from shooting them before they are visible
                    bounds.center = transform.position + boundsCenterOffset;
                    if (bounds.extents == Vector3.zero || Utils.ScreenBoundsCheck(bounds,
BoundsTest.offScreen) != Vector3.zero) {
                        Destroy(other);
                        break;
                    }
                    // Hurt this Enemy
                    // Get the damage amount from the Projectile.type & Main.W_DEFS
                    health -= Main.W_DEFS[p.type].damageOnHit;
                    if (health <= 0) {
                        // Destroy this Enemy
                        Destroy(this.gameObject);
                    }
                    Destroy(other);
                    break;
            }
        }
    }
}

```

Now when you play the scene, it is possible to destroy an Enemy, but each one takes 10 shots to take down, and it's difficult to tell that they're being damaged. We will add code that makes an Enemy blink red for a couple of frames every time it is hit, but to do so, we're going to need to have access to all the materials of all the children of each Enemy. This sounds like something that may be useful in later prototypes, so we will add it to the Utils script. Open the Utils script in MonoDevelop and add the following static method to achieve this:

[Click here to view code image](#)

```

public class Utils : MonoBehaviour {

    //===== Bounds Functions =====\\
    ...

    //===== Transform Functions =====\\
    ...

    public static GameObject FindTaggedParent(Transform t) {
        return( FindTaggedParent( t.gameObject ) );
    }

}

//===== Materials Functions =====\\

// Returns a list of all Materials on this GameObject or its children
static public Material[] GetAllMaterials( GameObject go ) {
    List<Material> mats = new List<Material>();
    if (go.renderer != null) {
        mats.Add(go.renderer.material);
    }
    foreach( Transform t in go.transform ) {
        mats.AddRange( GetAllMaterials( t.gameObject ) );
    }
}

```

```
        }  
        return( mats.ToArray() );  
    }  
}
```

Now, add the following bold code to the `Enemy` class:

[Click here to view code image](#)

```

public class Enemy : MonoBehaviour {
    ...
    public int score = 100; // Points earned for destroying this

    public int showDamageForFrames = 2; // # frames to show damage

    public bool _____;

    public Color[] originalColors;
    public Material[] materials; // All the Materials of this & its children
    public int remainingDamageFrames = 0; // Damage frames left

    public Bounds bounds; // The Bounds of this and its children

    void Awake() {
        materials = Utils.GetAllMaterials( gameObject );
        originalColors = new Color[materials.Length];
        for (int i=0; i<materials.Length; i++) {
            originalColors[i] = materials[i].color;
        }
        InvokeRepeating( "CheckOffscreen", 0f, 2f );
    }

    // Update is called once per frame
    void Update() {
        Move();
        if (remainingDamageFrames>0) {
            remainingDamageFrames--;
            if (remainingDamageFrames == 0) {
                UnShowDamage();
            }
        }
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
        case "ProjectileHero":
            ...
            // Hurt this Enemy
            ShowDamage();
            // Get the damage amount from the Projectile.type & Main.W_DEFS
            ...
            break;
        }
    }

    void ShowDamage() {
        foreach (Material m in materials) {
            m.color = Color.red;
        }
        remainingDamageFrames = showDamageForFrames;
    }

    void UnShowDamage() {
    }
}

```

```

        for ( int i=0; i<materials.Length; i++ ) {
            materials[i].color = originalColors[i];
        }
    }
}

```

Now, when an Enemy is struck by a projectile from the `_Hero`, it will turn entirely red for `showDamageForFrames` frames by setting the color of all materials to red and setting `remainingDamageFrames` to `showDamageForFrames`. Each update, if `remainingDamageFrames` is greater than 0, it is decremented until it reaches 0, at which time, the enemy ship and children revert to their original colors.

Now it's possible to see that the player is damaging the ship, but it still takes many hits to destroy one. Let's make some power-ups that will increase the power and number of the player's weapons.

Adding Power-Ups

At this point, there will be three power-ups in the game:

- **blaster [B]**: If the player weapon type is not blaster, switches to blaster and resets to 1 gun. If the player weapon type is already blaster, increases the number of guns.
- **spread [S]**: If the player weapon type is not spread, switches to spread and resets to 1 gun. If the player weapon type is already spread, increases the number of guns.
- **shield [O]**: Increases the player's `shieldLevel` by 1.

Artwork for Power-Ups

The power-ups will be made of a letter rendered as 3D text with a spinning cube behind it. (You can see some of them in [Figure 30.1](#) at the beginning of the chapter.) To make the power-ups, complete these steps:

1. Create a new 3D text (*GameObject > Create Other > 3D Text* from the menu bar). Name it `PowerUp` and give it the following settings:

PowerUp (3D Text)	P:[10,0,0]	R:[0,0,0]	S:[1,1,1]
Cube (Cube)	P:[0,0,0]	R:[0,0,0]	S:[2,2,2]
2. Create a cube that is a child of `PowerUp` as described in the preceding settings.
3. Select the `PowerUp`.
4. Set the *Text Mesh* component properties of `PowerUp` to those shown in [Figure 30.12](#).
5. Add a *Rigidbody* component to `PowerUp` (*Component > Physics > Rigidbody*) and set it as shown in [Figure 30.12](#).
6. Set both the tag and the layer of `PowerUp` to `PowerUp`. When asked, click *Yes, change children*.

Next, you will create a custom material for the `PowerUp` cube, as follows:

1. Create a new Material named *Mat PowerUp*.
2. Drag it on to the cube that is a child of `PowerUp`.
3. Select the cube that is a child of `PowerUp`.
4. Set the Shader of `Mat PowerUp` to *ProtoTools > UnlitAlpha*.
5. Click the *Select* button at the bottom right of the texture for `Mat PowerUp` and choose the texture named `PowerUp` from the *Assets* tab.

6. Set the main color of Mat PowerUp to cyan (a light blue that is RGBA:[0,255,255,255]), and you can see how the PowerUp will look when colored.

7. Set the *Box Collider* of cube to be a trigger (check the box next to *Is Trigger*).

Double-check that all the settings for PowerUp and its child Cube match those in [Figure 30.12](#) and save your scene.

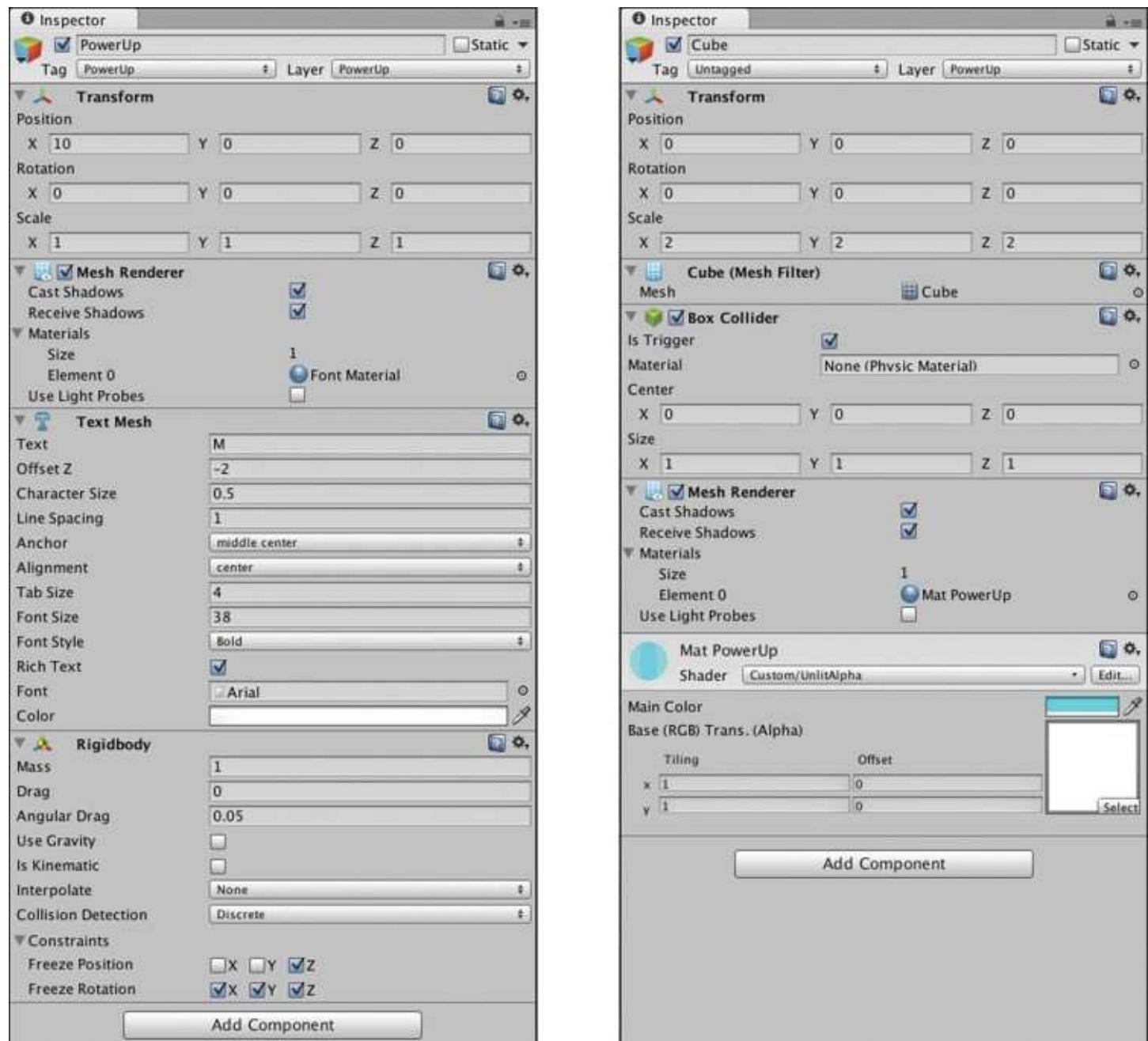


Figure 30.12 Settings for PowerUp and its child Cube prior to attaching any scripts

PowerUp Code

Now create a new C# script named *PowerUp* and assign it to the PowerUp GameObject in the Hierarchy. Open the PowerUp script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class PowerUp : MonoBehaviour {
    // This is an unusual but handy use of Vector2s. x holds a min value
    // and y a max value for a Random.Range() that will be called later
```

```

public Vector2          rotMinMax = new Vector2(15,90);
public Vector2          driftMinMax = new Vector2(.25f,2);
public float             lifeTime = 6f; // Seconds the PowerUp exists
public float             fadeTime = 4f; // Seconds it will then fade
public bool _____;
public WeaponType        type; // The type of the PowerUp
public GameObject        cube; // Reference to the Cube child
public TextMesh          letter; // Reference to the TextMesh
public Vector3           rotPerSecond; // Euler rotation speed
public float              birthTime;

void Awake() {
    // Find the Cube reference
    cube = transform.Find("Cube").gameObject;
    // Find the TextMesh
    letter = GetComponent<TextMesh>();

    // Set a random velocity
    Vector3 vel = Random.onUnitSphere; // Get Random XYZ velocity
    // Random.onUnitSphere gives you a vector point that is somewhere on
    // the surface of the sphere with a radius of 1m around the origin
    vel.z = 0; // Flatten the vel to the XY plane
    vel.Normalize(); // Make the length of the vel 1
    // Normalizing a Vector3 makes it length 1m
    vel *= Random.Range(driftMinMax.x, driftMinMax.y);
    // Above sets the velocity length to something between the x and y
    // values of driftMinMax
    rigidbody.velocity = vel;

    // Set the rotation of this GameObject to R:[0,0,0]
    transform.rotation = Quaternion.identity;
    // Quaternion.identity is equal to no rotation.

    // Set up the rotPerSecond for the Cube child using rotMinMax x & y
    rotPerSecond = new Vector3( Random.Range(rotMinMax.x,rotMinMax.y),
                               Random.Range(rotMinMax.x,rotMinMax.y),
                               Random.Range(rotMinMax.x,rotMinMax.y) );

    // CheckOffscreen() every 2 seconds
    InvokeRepeating( "CheckOffscreen", 2f, 2f );

    birthTime = Time.time;
}

void Update () {
    // Manually rotate the Cube child every Update()
    // Multiplying it by Time.time causes the rotation to be time-based
    cube.transform.rotation = Quaternion.Euler( rotPerSecond*Time.time );

    // Fade out the PowerUp over time
    // Given the default values, a PowerUp will exist for 10 seconds
    // and then fade out over 4 seconds.
    float u = (Time.time - (birthTime+lifeTime)) / fadeTime;
    // For lifeTime seconds, u will be <= 0. Then it will transition to 1
    // over fadeTime seconds.
    // If u >= 1, destroy this PowerUp
    if (u >= 1) {
        Destroy( this.gameObject );
        return;
    }
    // Use u to determine the alpha value of the Cube & Letter
    if (u>0) {
        Color c = cube.renderer.material.color;

```

```

        c.a = 1f-u;
        cube.renderer.material.color = c;
        // Fade the Letter too, just not as much
        c = letter.color;
        c.a = 1f - (u*0.5f);
        letter.color = c;
    }

}

// This SetType() differs from those on Weapon and Projectile
public void SetType( WeaponType wt ) {
    // Grab the WeaponDefinition from Main
    WeaponDefinition def = Main.GetWeaponDefinition( wt );
    // Set the color of the Cube child
    cube.renderer.material.color = def.color;
    //letter.color = def.color; // We could colorize the letter too
    letter.text = def.letter; // Set the letter that is shown
    type = wt; // Finally actually set the type
}

public void AbsorbedBy( GameObject target ) {
    // This function is called by the Hero class when a PowerUp is collected
    // We could tween into the target and shrink in size,
    // but for now, just destroy this.gameObject
    Destroy( this.gameObject );
}

void CheckOffscreen() {
    // If the PowerUp has drifted entirely off screen...
    if ( Utils.ScreenBoundsCheck( cube.collider.bounds,
        BoundsTest.offScreen ) != Vector3.zero ) {
        // ...then destroy this GameObject
        Destroy( this.gameObject );
    }
}
}

```

When you press Play, you should see the power-up drifting and rotating. If you fly `_Hero` into the power-up, you will get the console message “Triggered: Cube,” letting you know that the Trigger Collider on the PowerUp cube is working properly.

Drag the PowerUp GameObject from the Hierarchy into the `_Prefabs` folder in the Project pane to make it into a prefab. Delete the remaining PowerUp instance from the Hierarchy.

Now, make the following changes to the Hero C# script to enable the Hero to collide with and collect power-ups:

[Click here to view code image](#)

```

public class Hero : MonoBehaviour {
    ...
    private float           _shieldLevel = 1;

    // Weapon fields
    public Weapon[]          weapons;

    public bool _____;

    void Awake() {
        S = this; // Set the Singleton
        bounds = Utils.CombineBoundsOfChildren(this.gameObject);

        // Reset the weapons to start _Hero with 1 blaster
    }
}

```

```

clearWeapons();
weapons[0].SetType(WeaponType.blaster);
}

void OnTriggerEnter(Collider other) {
    ...
    if (go != null) {
        ...

        if (go.tag == "Enemy") {
            // If the shield was triggered by an enemy
            // Decrease the level of the shield by 1
            shieldLevel--;
            // Destroy the enemy
            Destroy(go);
        } else if (go.tag == "PowerUp") {
            // If the shield was triggered by a PowerUp
            AbsorbPowerUp(go);
        } else {
            print("Triggered: "+go.name); // Move this line here!
        }
    }
    ...
}

public void AbsorbPowerUp( GameObject go ) {
    PowerUp pu = go.GetComponent<PowerUp>();
    switch (pu.type) {
        case WeaponType.shield: // If it's the shield
            shieldLevel++;
            break;

        default: // If it's any Weapon PowerUp
            // Check the current weapon type
            if (pu.type == weapons[0].type) {
                // then increase the number of weapons of this type
                Weapon w = GetEmptyWeaponSlot(); // Find an available weapon
                if (w != null) {
                    // Set it to pu.type
                    w.SetType(pu.type);
                }
            } else {
                // If this is a different weapon
                ClearWeapons();
                weapons[0].SetType(pu.type);
            }
            break;
    }
    pu.AbsorbedBy( this.gameObject );
}

Weapon GetEmptyWeaponSlot() {
    for (int i=0; i<weapons.Length; i++) {
        if ( weapons[i].type == WeaponType.none ) {
            return( weapons[i] );
        }
    }
    return( null );
}

void ClearWeapons() {
    foreach (Weapon w in weapons) {
        w.SetType(WeaponType.none);
    }
}

```

}

}

Now that the code is set up, you need to make a couple of changes to `_Hero` in Unity. Open the disclosure triangle next to the `GameObject _Hero` in the Hierarchy. Select the `Weapon` child of `_Hero`. Press Command-D (or Control+D on Windows) four times to make four duplicates of `Weapon`. These should all still be children of `_Hero`. Rename the five weapons `Weapon_0` through `Weapon_4` and configure their transforms as follows:

_Hero	P:[0,0,0]	R:[0,0,0]	S:[1,1,1]
Weapon_0	P:[0,2,0]	R:[0,0,0]	S:[1,1,1]
Weapon_1	P:[-2,-1,0]	R:[0,0,0]	S:[1,1,1]
Weapon_2	P:[2,-1,0]	R:[0,0,0]	S:[1,1,1]
Weapon_3	P:[-1.25,-0.25,0]	R:[0,0,0]	S:[1,1,1]
Weapon_4	P:[1.25,-0.25,0]	R:[0,0,0]	S:[1,1,1]

Next, select `_Hero` and open the disclosure triangle for the `weapons` field in the `Hero (Script)` component Inspector. Set the `Size` of `weapons` to 5 and assign `Weapon_0` through `Weapon_4` to the five `Weapon` slots in order (either by dragging them in from the Hierarchy or by clicking the target to the right of the `Weapon` slot and selecting each `Weapon_#` from the Scene tab). [Figure 30.13](#) shows the final setup.

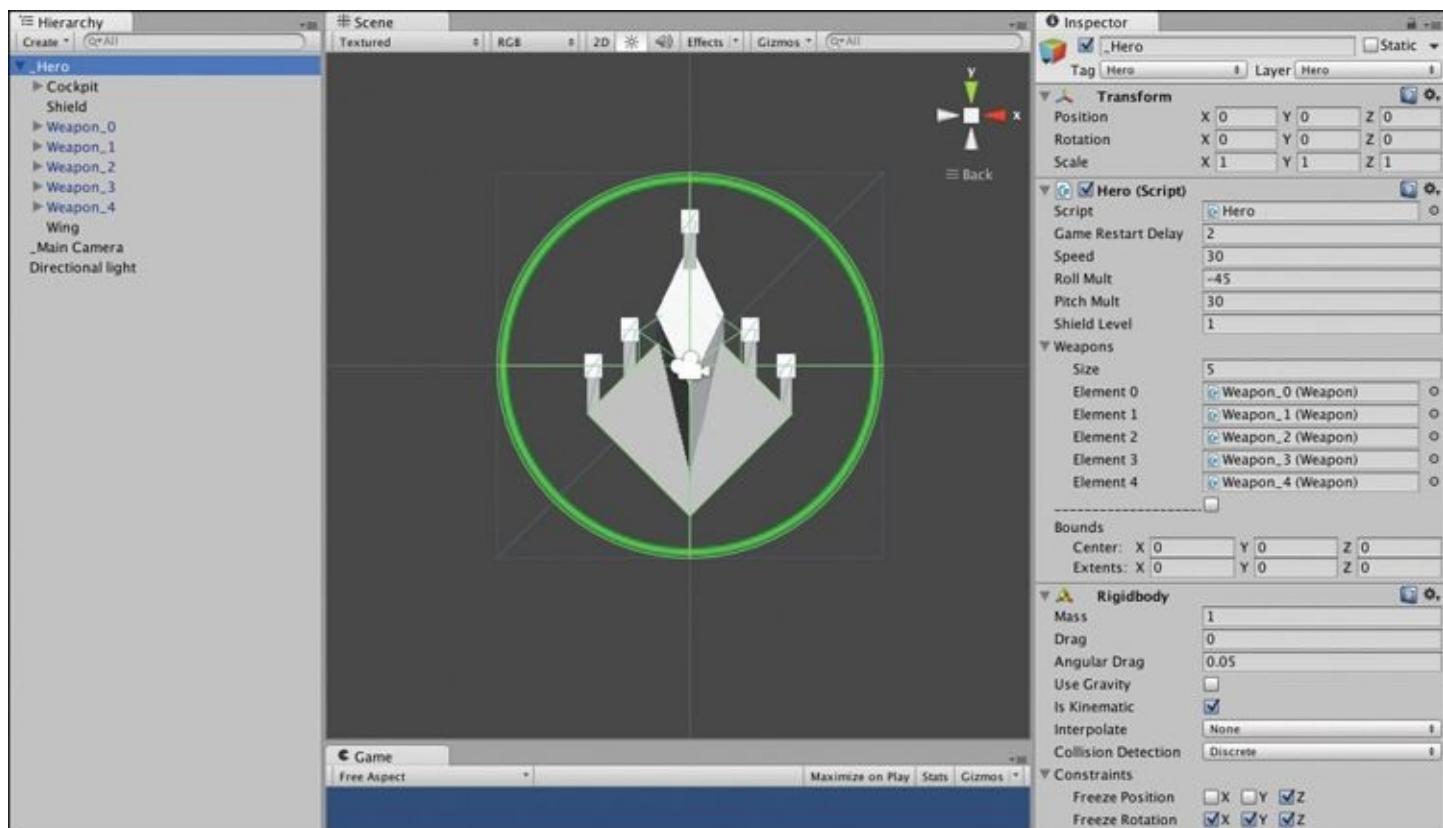


Figure 30.13 The `_Hero` ship showing five Weapons as children and assigned to the weapons field

Resolving Race Conditions in Code

Now, when you try to play the scene as we've created it, you may encounter an error message in the Console pane. It's also possible that you will not get this error. In this code, I've tried to intentionally create a *race condition* to show you how to resolve them. A race condition occurs when one piece of code must be executed before another piece of code, but it's possible that they will execute in the wrong order. The two pieces of code end up racing against each other. The thing about race conditions is that they're unpredictable, so you might not get the error that I tried to create. Regardless, please read this section. Race conditions are an important kind of error that you should understand. The error you may encounter is as follows:

```
NullReferenceException: Object reference not set to an instance of an object
Main.GetWeaponDefinition (WeaponType wt) (at Assets/_Scripts/Main.cs:38)
Weapon.SetType (WeaponType wt) (at Assets/_Scripts/Weapon.cs:77) Hero.Awake () (at
Assets/_Scripts/Hero.cs:35)
```

If you double-click the error message, it should take you to line 38 of Main.cs. (Your line number might differ slightly.) Line 38 is:

```
if (W_DEFS.ContainsKey(wt)) {
```

Let's use the debugger to learn more about what's causing the error. (Please do this even if you're not getting the error.) Add a breakpoint next to this line in Main.cs and attach the debugger to Unity (by clicking the Play icon in the top-left corner of the MonoDevelop window or selecting *Run > Attach to Process* from the MonoDevelop menu bar). If you need a refresher on the debugger, reread [Chapter 24](#). Unfortunately, in this case, you will need to have the debugger attached when the scene first starts playing, so the trick described in [Chapter 24](#) where you start the game paused and then attach the debugger later won't work for these bugs.

When you run the project (in Unity) with the debugger attached, it will freeze on your line 38 breakpoint immediately before executing that line. We know that something's wrong with this line, and as a NullReferenceException, we know that the code is trying to access some variable that isn't yet defined. Let's look at each variable and see what's happening.

1. Open the Watch panel in MonoDevelop (*View > Debug Windows > Watch* from the menu bar; there should be already a check mark next to it, and selecting it again will bring the Watch panel to the front).
2. The two variables used in this line are W_DEFS (a static variable of the Main class) and wt (a local variable of the method GetWeaponDefinition ()).
3. Type each of these into a line of the Watch window, and you'll be able to see their individual values.
4. As expected, W_DEFS isn't defined (its value is null). (That is, if you're experiencing the race condition error on your machine.) But we know that W_DEFS is properly defined in Main.Awake (). You can see the code that does so just a few lines above. The only way that W_DEFS could not be defined is if Main.Awake () hasn't run yet.

This is the race condition. Main.Awake () defines W_DEF, and Hero.Awake () is trying to use that value. We know that Awake () is called on each GameObject as it comes into being, but it is unclear in what order they are called. I believe that it probably happens in the order that the objects are listed in the Hierarchy, but I'm not certain of that. It's possible that your Awake () methods may be called in a different order than mine.

This is the major problem with race conditions. The two `Awake()` functions are racing against each other. When one is called first, your code works fine, but when the other is called first, everything breaks. Regardless of whether your code happens to be working, this is an issue that you need to resolve, because even on the same computer, the two `Awake()` functions could be called in different orders from one time to the next..

This is one reason that there are both `Awake()` and `Start()` methods in Unity. `Awake()` is called immediately when a `GameObject` is instantiated, while `Start()` is called immediately before the first `Update()` that the `GameObject` ever receives. This can be a difference of several milliseconds, which for a computer program is a very long time. If you have a number of objects in your scene, you can be guaranteed that `Awake()` will be called on all of them before `Start()` is called on any of them. `Awake()` will always happen before `Start()`.

Knowing this, take a look back at the original error. If you look at the Call Stack pane in MonoDevelop (`View > Debug Windows > Call Stack` from the menu bar), it looks like `Hero.Awake()` on line 35 called `Weapon.SetType()`, which in turn called `Main.GetWeaponDefinition()`. To start fixing this issue, we will choose to delay the call from `Hero.Awake()` by moving it into `Hero.Start()`. Make the following changes to the `Hero` C# script. You should click the Stop sign in the MonoDevelop debugger (or select `Run > Stop` from the menu bar) as well as stop playback in Unity before changing the `Hero` script code:

[Click here to view code image](#)

```
public class Hero : MonoBehaviour {
    ...
    void Awake() {
        S = this; // Set the Singleton
        bounds = Utils.CombineBoundsOfChildren(this.gameObject);
    }
    void Start() {
        // Reset the weapons to start _Hero with 1 blaster
        ClearWeapons();
        weapons[0].SetType(WeaponType.blaster);
    }
    ...
}
```

However after doing so, playing the project will expose yet another race condition error!

`UnassignedReferenceException`: The variable `collar` of `Weapon` has not been assigned. You probably need to assign the `collar` variable of the `Weapon` script in the Inspector. `Weapon.SetType(WeaponType wt)` (at Assets/_Scripts/Weapon.cs:78) `Hero.Start()` (at Assets/_Scripts/Hero.cs:38)

Attach the MonoDevelop debugger to the Unity process again to get more information on this error. Place a breakpoint on line 78 of `Weapon.cs` and then press Play in Unity. Because the `Start()` functions are called at different times, I sometimes saw the code first stop on line 38 of `Hero.cs` (where the breakpoint still remains from the previous debug) and sometimes saw it first stop on line 78 of `Weapon`. This is happening because both `Hero.Start()` and `Weapon.Start()` call `Weapon.SetType()`. If `Weapon.Start()` happens to be called before `Hero.Start()`, this is fine, but if `Hero.Start()` is called first, we get an error due to the race condition. The issue here is

that all Weapons need to define `Weapon.collider` before `Hero.Start()` is run. To resolve this, move the definition of `collar` from the `Start()` method to an `Awake()` method in the the Weapon C# script.

[Click here to view code image](#)

```
void Awake() {
    collar = transform.Find("Collar").gameObject;
}

void Start() {
    // Call SetType() properly for the default _type
    SetType( _type );

    ...
}
```

Now, the race conditions should finally be resolved. `Weapon.Awake()` will define `collar` before either `Weapon.Start()` or `Hero.Start()` are called. Also, `Main.Awake()` will set the value of `Main.W_DEFS` before `Hero.Start()` is called. Race conditions are a common error for new game developers to step into, and it's important to be able to recognize when you may be encountering one. This is why I have lead you into this one and shown you how to discover and resolve the problem.

Making Enemies Drop Power-Ups

Getting back to the power-ups. Let's make enemies have the potential to drop a random power-up when they are destroyed. This gives the player a lot more incentive to try to destroy enemies rather than just avoid them, and it gives the player a path to improving her ship.

Add the following code to the `Enemy` and `Main` C# scripts:

[Click here to view code image](#)

```
public class Enemy : MonoBehaviour {
    ...

    public int showDamageForFrames = 2; // # frames to show damage
    public float powerUpDropChance = 1f; // Chance to drop a power-up

    public bool _____;

    ...

    void OnCollisionEnter( Collision coll ) {
        ...
        case "ProjectileHero":
            ...
            if (health <= 0) {
                // Tell the Main singleton that this ship has been destroyed
                Main.S.ShipDestroyed( this );
                // Destroy this Enemy
                Destroy(this.gameObject);
            }
            ...
    }
    ...
}

public class Main : MonoBehaviour {
    ...
    public WeaponDefinition[] weaponDefinitions;
```

```

public GameObject prefabPowerUp;
public WeaponType[] powerUpFrequency = new WeaponType[] {
    WeaponType.blaster, WeaponType.blaster,
    WeaponType.spread,
    WeaponType.shield
};

public bool _____;
...

public void ShipDestroyed( Enemy e ) {
    // Potentially generate a PowerUp
    if (Random.value <= e.powerUpDropChance) {
        // Random.value generates a value between 0 & 1 (though never == 1)
        // If the e.powerUpDropChance is 0.50f, a PowerUp will be generated
        // 50% of the time. For testing, it's now set to 1f.

        // Choose which PowerUp to pick
        // Pick one from the possibilities in powerUpFrequency
        int ndx = Random.Range(0,powerUpFrequency.Length);
        WeaponType puType = powerUpFrequency[ndx];

        // Spawn a PowerUp
        GameObject go = Instantiate( prefabPowerUp ) as GameObject;
        PowerUp pu = go.GetComponent<PowerUp>();
        // Set it to the proper WeaponType
        pu.SetType( puType );

        // Set it to the position of the destroyed ship
        pu.transform.position = e.transform.position;
    }
}
}

```

Before this code will work, you need to select `_MainCamera` in the Unity Hierarchy and set the `prefabPowerUp` field of the Main Script component to be the PowerUp prefab in the `_Prefabs` folder of the Project pane. `powerUpFrequency` should already be set in the Inspector, but just in case, [Figure 30.14](#) shows the correct settings. Note that enums appear in the Unity Inspector as pop-up menus.

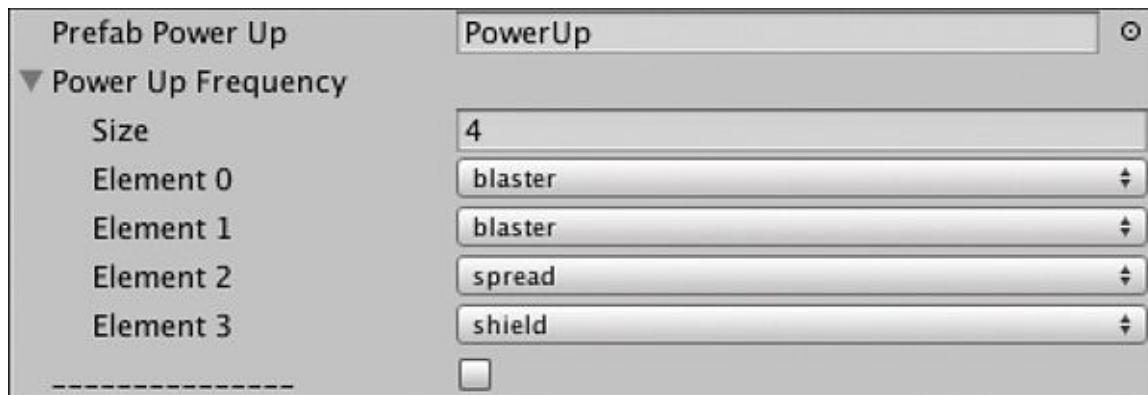


Figure 30.14 `prefabPowerUp` and `powerUpFrequency` on the Main (Script) component of `_MainCamera`

Now play the scene and destroy some enemies. They should drop power-ups that will now improve your ship!

You should notice over time that the blaster [B] power-up is more common than spread [S] or shield [O]. This is because there are two occurrences of blaster in `powerUpFrequency` and only one each

of spread and shield. By adjusting the relative numbers of occurrences of each of these in `powerUpFrequency`, you can determine the chance that each will be chosen relative to the others. This same trick could also be used to set the frequency of different types of enemies spawning by assigning some enemies to the `prefabEnemies` array more times than other enemy types.

Programming Other Enemies

Now that the core elements of the game are each working, it's time to expand the different offerings of enemies. Create new C# scripts named `Enemy_1`, `Enemy_2`, `Enemy_3`, and `Enemy_4` and assign them each to their respective `Enemy_#` prefab in the Project pane.

Enemy_1

Open `Enemy_1` scripts in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

// Enemy_1 extends the Enemy class
public class Enemy_1 : Enemy {
    // Because Enemy_1 extends Enemy, the _____ bool won't work           // 1
    // the same way in the Inspector pane. :/

    // # seconds for a full sine wave
    public float    waveFrequency = 2;
    // sine wave width in meters
    public float    waveWidth = 4;
    public float    waveRotY = 45;

    private float    x0 = -12345; // The initial x value of pos
    private float    birthTime;

    void Start() {
        // Set x0 to the initial x position of Enemy_1
        // This works fine because the position will have already
        // been set by Main.SpawnEnemy() before Start() runs
        // (though Awake() would have been too early!).
        // This is also good because there is no Start() method
        // on Enemy.
        x0 = pos.x;

        birthTime = Time.time;
    }

    // Override the Move function on Enemy
    public override void Move() {                                         // 2
        // Because pos is a property, you can't directly set pos.x
        // so get the pos as an editable Vector3
        Vector3 tempPos = pos;
        // theta adjusts based on time
        float age = Time.time - birthTime;
        float theta = Mathf.PI * 2 * age / waveFrequency;
        float sin = Mathf.Sin(theta);
        tempPos.x = x0 + waveWidth * sin;
        pos = tempPos;

        // rotate a bit about y
        Vector3 rot = new Vector3(0, sin*waveRotY, 0);
        this.transform.rotation = Quaternion.Euler(rot);
```

```
// base.Move() still handles the movement down in y
base.Move(); // 3
}
```

1. The bool _____ that is used to divide the elements you should set in the Inspector from those you should not won't work that way in these subclasses of `Enemy` because when the Inspector sees a subclass like this, it will first list all the public fields of the superclass and then all the public fields of the subclass. This will place `waveFrequency`, `waveWidth`, and `waveRotY` below the line, even though you should feel free to manipulate them in the Inspector.
2. Because the method `Move()` is marked as a virtual method in the `Enemy` superclass, we are able to override it here and replace it with another function.
3. `base.Move()` calls the `Move()` function on the superclass `Enemy`.

Back in Unity, select `_MainCamera` in the Hierarchy and change *Element 0* of `prefabEnemies` from `Enemy_0` to `Enemy_1` (which is the `Enemy_1` prefab in the Project pane) in the *Main (Script)* component. Now, when you press Play, the `Enemy_1` ship will appear instead of `Enemy_0`, and it will move in a wave.

Tip

Sphere Colliders Only Scale Uniformly You might have noticed that the collision with `Enemy_1` actually occurs before the projectile reaches the wing. If you select `Enemy_1` in the Project pane and drag an instance into the scene, you will see that the green collider spheres around `Enemy_1` don't scale to match the flat ellipse of the wing. This isn't a huge problem, but it is something to be aware of. A Sphere Collider will scale with the largest single component of scale in the transform. (In this case, because wing has a `Scale.x` of 6, the Sphere Collider scales up to that.)

If you want, you can try other types of colliders to see whether one of them will scale to match the wing more accurately. A Box Collider will scale nonuniformly. You can also approximate one direction being much longer than the others with a Capsule Collider. A Mesh Collider will match the scaling most exactly, but Mesh Colliders are much slower than other types. This shouldn't be a problem on a modern high-performance PC, but Mesh Colliders are often too slow for mobile platforms like iOS or Android.

If you choose to give `Enemy_1` a Box Collider or Mesh Collider, then when it rotates about the `y` axis, it will move the edges of the wing out of the XY (that is, `z=0`) plane. This is why the `ProjectileHero` prefab has a Box Collider `Size.z` of 10 (to make sure that it can hit the wingtips of `Enemy_1` even if they are not in the XY plane).

Preparing for the Other Enemies

The remaining enemies make use of *linear interpolation*, an important development concept that is described in [Appendix B](#). You saw a very simple interpolation in [Chapter 29](#), “[Prototype 2: Mission Demolition](#),” but these will be a bit more interesting. Take a moment to read the “[Interpolation](#)” section of [Appendix B](#), before tackling the remaining enemies.

Enemy_2

Enemy_2 will move via a linear interpolation that is heavily eased by a sine wave. It will rush in from the side of the screen, slow, reverse direction for a bit, slow, and then fly off the screen along its initial velocity. Only two points will be used in this interpolation, but the u value will be drastically curved by a sine wave. The easing function for the u value of Enemy_2 will be along the lines of

$$u = u + 0.6 * \sin(2\pi * u)$$

This is one of the easing functions depicted in the “[Interpolation](#)” section of [Appendix B](#).

Open the Enemy_2 C# script and enter the following code. After you have the code working, you’re welcome to adjust the easing curve and see how it affects the motion.

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Enemy_2 : Enemy {
    // Enemy_2 uses a Sin wave to modify a 2-point linear interpolation
    public Vector3[] points;
    public float birthTime;
    public float lifeTime = 10;
    // Determines how much the Sine wave will affect movement
    public float sinEccentricity = 0.6f;

    void Start () {
        // Initialize the points
        points = new Vector3[2];

        // Find Utils.camBounds
        Vector3 cbMin = Utils.camBounds.min;
        Vector3 cbMax = Utils.camBounds.max;

        Vector3 v = Vector3.zero;
        // Pick any point on the left side of the screen
        v.x = cbMin.x - Main.S.enemySpawnPadding;
        v.y = Random.Range( cbMin.y, cbMax.y );
        points[0] = v;

        // Pick any point on the right side of the screen
        v = Vector3.zero;
        v.x = cbMax.x + Main.S.enemySpawnPadding;
        v.y = Random.Range( cbMin.y, cbMax.y );
        points[1] = v;

        // Possibly swap sides
        if (Random.value < 0.5f) {
            // Setting the .x of each point to its negative will move it to the
            // other side of the screen
            points[0].x *= -1;
            points[1].x *= -1;
        }
    }

    // Set the birthTime to the current time
    birthTime = Time.time;
}

public override void Move() {
    // Bézier curves work based on a u value between 0 & 1
    float u = (Time.time - birthTime) / lifeTime;
```

```

// If u>1, then it has been longer than lifeTime since birthTime
if (u > 1) {
    // This Enemy_2 has finished its life
    Destroy( this.gameObject );
    return;
}

// Adjust u by adding an easing curve based on a Sine wave
u = u + sinEccentricity*(Mathf.Sin(u*Mathf.PI*2));

// Interpolate the two linear interpolation points
pos = (1-u)*points[0] + u*points[1];
}
}
}

```

Swap the Enemy_2 prefab into the Element 0 slot of Main.S.prefabEnemies using the _MainCamera Inspector and press Play. As you can see the easing function causes each Enemy_2 to have very smooth movement that waves between the points it has selected on either side of the screen.

Enemy_3

Enemy_3 will use a Bézier curve to swoop down from above, slow, and fly back up off the top of the screen. For this example, we will use a simple version of the three-point Bézier curve function. In the “[Interpolation](#)” section of [Appendix B](#) you can find a recursive version of the Bézier curve function that can use any number of points (not just three).

Open the Enemy_3 script and enter the following code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;

// Enemy_3 extends Enemy
public class Enemy_3 : Enemy {

    // Enemy_3 will move following a Bezier curve, which is a linear
    // interpolation between more than two points.
    public Vector3[] points;
    public float birthTime;
    public float lifeTime = 10;

    // Again, Start works well because it is not used by Enemy
    void Start () {
        points = new Vector3[3]; // Initialize points

        // The start position has already been set by Main.SpawnEnemy()
        points[0] = pos;

        // Set xMin and xMax the same way that Main.SpawnEnemy() does
        float xMin = Utils.camBounds.min.x+Main.S.enemySpawnPadding;
        float xMax = Utils.camBounds.max.x-Main.S.enemySpawnPadding;

        Vector3 v;
        // Pick a random middle position in the bottom half of the screen
        v = Vector3.zero;
        v.x = Random.Range( xMin, xMax );
        v.y = Random.Range( Utils.camBounds.min.y, 0 );
        points[1] = v;

        // Pick a random final position above the top of the screen
        v = Vector3.zero;
    }
}

```

```

v.y = pos.y;
v.x = Random.Range( xMin, xMax );
points[2] = v;

// Set the birthTime to the current time
birthTime = Time.time;
}

public override void Move() {
    // Bezier curves work based on a u value between 0 & 1
    float u = (Time.time - birthTime) / lifeTime;

    if (u > 1) {
        // This Enemy_3 has finished its life
        Destroy( this.gameObject );
        return;
    }

    // Interpolate the three Bezier curve points
    Vector3 p01, p12;
    p01 = (1-u)*points[0] + u*points[1];
    p12 = (1-u)*points[1] + u*points[2];
    pos = (1-u)*p01 + u*p12;
}

}
}

```

Now try swapping Enemy_3 into the Element 0 of `prefabEnemies` on `_MainCamera`. These have a very different movement than the previous enemies. After playing for a bit, you'll notice a couple of things about Bézier curves:

1. Even though the midpoint is at or below the bottom of the screen, no `Enemy_3` ever gets that far down. That is because a Bézier curve touches both the start and end points but is only influenced by the midpoint.
2. `Enemy_3` slows down a lot in the middle of the curve. This is also a feature of Bézier curves. If you want, you can correct this by adding the following bold line to the `Enemy_3 Move()` method just before the curve points are interpolated. This will add easing to the `Enemy_3` movement that will speed up the middle of the curve to make the movement feel more consistent:

[Click here to view code image](#)

```

Vector3 p01, p12;
u = u - 0.2f * Mathf.Sin(u * Mathf.PI * 2);
p01 = (1-u)*points[0] + u*points[1];

```

Enemy_4

As somewhat of a boss type, `Enemy_4` will have more health than other `Enemy` types and will have destructible parts (rather than all the parts being destroyed at the same time). It will also stay on screen, moving from one position to another, until the player destroys it completely.

Collider Modifications

Before getting into code issues, you need to make a few adjustments to the colliders of `Enemy_4`. First, drag an instance of `Enemy_4` into the Hierarchy and make sure that it's positioned away from other `GameObjects` in the scene.

Open the disclosure triangle next to `Enemy_4` in the Hierarchy and select `Enemy_4.Fuselage`. Replace

the Sphere Collider with a Capsule Collider by selecting *Component > Physics > Capsule Collider* from the menu bar. If Unity asks you, choose to replace the Sphere Collider with the Capsule Collider, if it doesn't ask you, you will need to manually remove the Sphere Collider. Set the Capsule Collider as follows in the Fuselage Inspector:

Center	[0,0,0]
Radius	0.5
Height	1
Direction	Y-Axis

Feel free to play with the values somewhat to see how they affect things. As you can see, the Capsule Collider is a much better approximation of Fuselage than the Sphere Collider was.

Now, select Wing_L in the Hierarchy and replace its Sphere Collider with a Capsule Collider as well. The settings for this collider are as follows:

Center	[0,0,0]
Radius	0.1
Height	5
Direction	X-Axis

The Direction setting chooses which is the long axis of the capsule. This is determined in local coordinates, so the Capsule Collider height of 5 along the X-axis matches the Transform scale of 5 in the X dimension. The radius of 0.1 states that the radius should be 1/10th of the height ($5 * 1/10^{th} = 0.5$, which is the Z Scale dimension). You can see that the capsule does not perfectly match the wing, but it is a much better approximation than a sphere.

Select Wing_R, replace its collider with a Capsule Collider, and give that collider the same settings as used on Wing_L. Once these changes have been made, click the *Prefab > Apply* button at the top of the Inspector pane to commit these changes to the Enemy_4 prefab in the Project pane. To double-check that this worked successfully, drag a second instance of the Enemy_4 prefab into the Hierarchy pane and check to make sure that the colliders all look correct. Once this is done, delete both instances of Enemy_4 from the Hierarchy pane.

This same Capsule Collider strategy could also be applied to Enemy_3 if you want.

Movement of Enemy_4

Enemy_4 will start in the standard position off the top of the screen, pick a random point on screen, and move to it over time using a linear interpolation. Each time Enemy_4 reaches the end of an interpolation, it will pick a new point and start moving toward it. Open the Enemy_4 script and input this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Enemy_4 : Enemy {
    // Enemy_4 will start offscreen and then pick a random point on screen to
    // move to. Once it has arrived, it will pick another random point and
    // continue until the player has shot it down.

    public Vector3[] points; // Stores the p0 & p1 for interpolation
    public float timeStart; // Birth time for this Enemy_4
    public float duration = 4; // Duration of movement
```

```

void Start () {
    points = new Vector3[2];
    // There is already an initial position chosen by Main.SpawnEnemy()
    // so add it to points as the initial p0 & p1
    points[0] = pos;
    points[1] = pos;

    InitMovement();
}

void InitMovement() {
    // Pick a new point to move to that is on screen
    Vector3 p1 = Vector3.zero;
    float esp = Main.S.enemySpawnPadding;
    Bounds cBounds = Utils.camBounds;
    p1.x = Random.Range(cBounds.min.x + esp, cBounds.max.x - esp);
    p1.y = Random.Range(cBounds.min.y + esp, cBounds.max.y - esp);

    points[0] = points[1]; // Shift points[1] to points[0]
    points[1] = p1; // Add p1 as points[1]

    // Reset the time
    timeStart = Time.time;
}

public override void Move () {
    // This completely overrides Enemy.Move() with a linear interpolation

    float u = (Time.time-timeStart)/duration;
    if (u>=1) { // if u >=1...
        InitMovement(); // ...then initialize movement to a new point
        u=0;
    }

    u = 1 - Mathf.Pow( 1-u, 2 ); // Apply Ease Out easing to u

    pos = (1-u)*points[0] + u*points[1]; // Simple linear interpolation
}
}

```

Swap the Enemy_4 prefab into the Element 0 slot of Main.S.prefabEnemies using the _MainCamera Inspector and save your scene. Did you remember to save after altering the colliders?

Press Play. You can see that the spawned Enemy_4s stay on screen until you destroy them. However, they're currently just as simple to take down as any of the other enemies. Now we'll break the Enemy_4 ship into four different parts with the central Cockpit protected by the others.

Open the Enemy_4 C# script and start by adding a new serializable class named Part to the top of Enemy_4.cs. Also be sure to add a Part [] array to the Enemy_4 class named parts.

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;

// Part is another serializable data storage class just like WeaponDefinition
[System.Serializable]
public class Part {
    // These three fields need to be defined in the Inspector pane
    public string name; // The name of this part
    public float health; // The amount of health this part has
    public string[] protectedBy; // The other parts that protect this
}

```

```

// These two fields are set automatically in Start().
// Caching like this makes it faster and easier to find these later
public GameObject    go;           // The GameObject of this part
public Material      mat;         // The Material to show damage
}

public class Enemy_4 : Enemy {
    ...
    public float           duration = 4; // Duration of movement

public Part[]           parts; // The array of ship Parts

void Start() {
    ...
}

...
}

```

The Part class will store individual information about the four parts of Enemy_4: Cockpit, Fuselage, Wing_L, and Wing_R.

Switch back to Unity and do the following:

1. Select the Enemy_4 prefab in the Project pane.
2. Expand the disclosure triangle next to parts in the *Inspector* > *Enemy_4 (Script)*.
3. Enter the settings shown in [Figure 30.15](#). The GameObject go and Material mat of each Part will be set automatically by code.

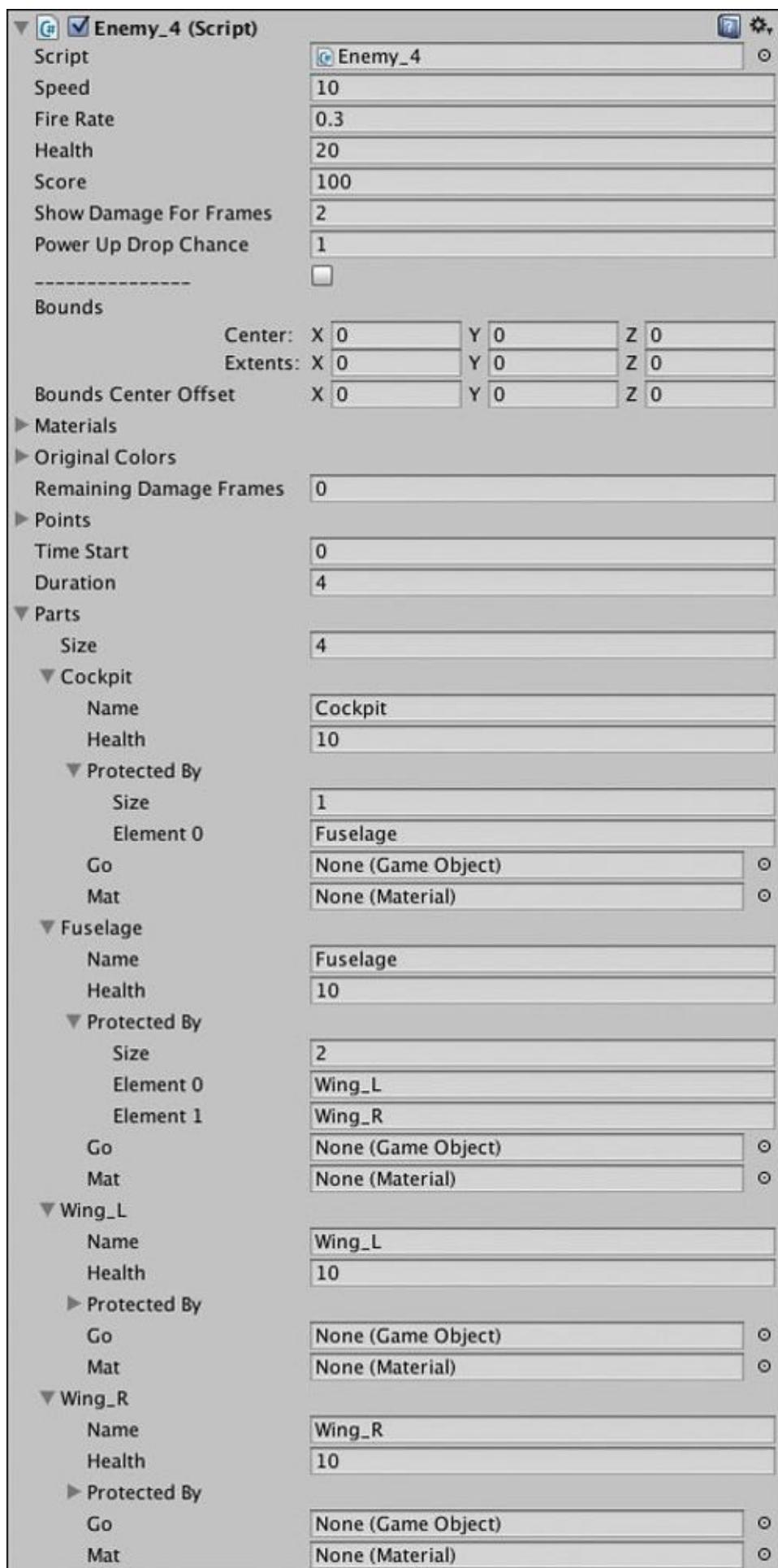


Figure 30.15 The settings for the `Parts` array of `Enemy_4`

As you can see in [Figure 30.15](#), each part has 10 health, and there is a tree of protection. Cockpit is

protected by Fuselage, and Fuselage is protected by both Wing_L and Wing_R. Now, switch back to MonoDevelop and make the following additions to the Enemy_4 class to make this protection work:

[Click here to view code image](#)

```
public class Enemy_4 : Enemy {
    ...
    void Start () {
        ...
        InitMovement ();

        // Cache GameObject & Material of each Part in parts
        Transform t;
        foreach(Part prt in parts) {
            t = transform.Find(prt.name);
            if (t != null) {
                prt.go = t.gameObject;
                prt.mat = prt.go.renderer.material;
            }
        }
    }

    ...
    public override void Move() {
        ...
    }

    // This will override the OnCollisionEnter that is part of Enemy.cs
    // Because of the way that MonoBehaviour declares common Unity functions
    // like OnCollisionEnter(), the override keyword is not necessary.
    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
            case "ProjectileHero":
                Projectile p = other.GetComponent<Projectile>();
                // Enemies don't take damage unless they're on screen
                // This stops the player from shooting them before they are visible
                bounds.center = transform.position + boundsCenterOffset;
                if (bounds.extents == Vector3.zero || Utils.ScreenBoundsCheck(bounds,
BoundsTest.offScreen) != Vector3.zero) {
                    Destroy(other);
                    break;
                }

                // Hurt this Enemy
                // Find the GameObject that was hit
                // The Collision coll has contacts[], an array of ContactPoints
                // Because there was a collision, we're guaranteed that there is at
                // least a contacts[0], and ContactPoints have a reference to
                // thisCollider, which will be the collider for the part of the
                // Enemy_4 that was hit.
                GameObject goHit = coll.contacts[0].thisCollider.gameObject;
                Part prtHit = FindPart(goHit);
                if (prtHit == null) { // If prtHit wasn't found
                    // ...then it's usually because, very rarely, thisCollider on
                    // contacts[0] will be the ProjectileHero instead of the ship
                    // part. If so, just look for otherCollider instead
                    goHit = coll.contacts[0].otherCollider.gameObject;
                    prtHit = FindPart(goHit);
                }

                // Check whether this part is still protected
                if (prtHit.protectedBy != null) {
                    foreach( string s in prtHit.protectedBy ) {

```

```

        // If one of the protecting parts hasn't been destroyed...
        if (!Destroyed(s)) {
            // ...then don't damage this part yet
            Destroy(other); // Destroy the ProjectileHero
            return; // return before causing damage
        }
    }
}

// It's not protected, so make it take damage
// Get the damage amount from the Projectile.type & Main.W_DEFS
prtHit.health -= Main.W_DEFS[p.type].damageOnHit;
// Show damage on the part
ShowLocalizedDamage(prtHit.mat);
if (prtHit.health <= 0) {
    // Instead of Destroying this enemy, disable the damaged part
    prtHit.go.SetActive(false);
}
// Check to see if the whole ship is destroyed
bool allDestroyed = true; // Assume it is destroyed
foreach( Part prt in parts ) {
    if (!Destroyed(prt)) { // If a part still exists
        allDestroyed = false; // ...change allDestroyed to false
        break; // and break out of the foreach loop
    }
}
if (allDestroyed) { // If it IS completely destroyed
    // Tell the Main singleton that this ship has been destroyed
    Main.S.ShipDestroyed( this );
    // Destroy this Enemy
    Destroy(this.gameObject);
}
Destroy(other); // Destroy the ProjectileHero
break;
}
}

// These two functions find a Part in this.parts by name or GameObject
Part FindPart(string n) {
    foreach( Part prt in parts ) {
        if (prt.name == n) {
            return( prt );
        }
    }
    return( null );
}
Part FindPart(GameObject go) {
    foreach( Part prt in parts ) {
        if (prt.go == go) {
            return( prt );
        }
    }
    return( null );
}

// These functions return true if the Part has been destroyed
bool Destroyed(GameObject go) {
    return( Destroyed( FindPart(go) ) );
}
bool Destroyed(string n) {
    return( Destroyed( FindPart(n) ) );
}
bool Destroyed(Part prt) {
    if (prt == null) { // If no real Part was passed in

```

```
        return(true); // Return true (meaning yes, it was destroyed)
    }
    // Returns the result of the comparison: prt.health <= 0
    // If prt.health is 0 or less, returns true (yes, it was destroyed)
    return (prt.health <= 0);
}

// This changes the color of just one Part to red instead of the whole ship
void ShowLocalizedDamage(Material m) {
    m.color = Color.red;
    remainingDamageFrames = showDamageForFrames;
}
```

Now when you play the scene, you should be overwhelmed by many `Enemy_4`s, each of which has two wings that protect the fuselage and a fuselage that protects the cockpit. If you want more of a chance against these, you can change the value of `Main (Script).enemySpawn PerSecond` on the `_MainCamera` to something lower, which will give you more time between `Enemy_4` spawns (though it will also delay the initial spawn).

Adding Particle Effects and Background

After all of that coding, here are a couple of things you can do just for fun to make the game look a little better.

Starfield Background

Create a two-layer starfield background to make things look more like outer space.

Create a quad in the Hierarchy (*GameObject > Create Other > Quad*). Name it *StarfieldBG*.

StarfieldBG (Quad) P:[0,0,10] R:[0,0,0] S:[80,80,1]

This will place `StarfieldBG` in the center of the camera's view and fill the view entirely. Now, create a new material named `Mat Starfield` and set its shader to `ProtoTools > UnlitAlpha`. Set the texture of `Mat Starfield` to the `Space Texture2D` that you imported at the beginning of this tutorial. Now drag `Mat Starfield` onto `StarfieldBG`, and you should see a starfield behind your `_Hero` ship.

Select *Mat Starfield* in the Project pane and duplicate it (Command-D on Mac or Control+D on PC). Name the new material *Mat Starfield Transparent*. Select *Space_Transparent* as the texture for this new material.

Select `StarfieldBG` in the Hierarchy and duplicate it. Name the duplicate `StarfieldFG_0`. Drag the `Starfield Transparent` material onto `StarfieldFG_0` and set its transform.

StarfieldFG_0 P:[0,0,5] R:[0,0,0] S:[160,160,1]

Now if you drag `StarfieldFG_0` around a bit, you'll see that it moves some stars in the foreground past stars in the background, creating a nifty parallax scrolling effect. Now duplicate `Starfield_FG_0` and name the duplicate `Starfield_FG_1`. You will need two copies of the foreground for the scrolling trick that we're going to employ.

Create a new C# script named *Parallax* and edit it in MonoDevelop.

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Parallax : MonoBehaviour {
    public GameObject poi; // The player ship
```

```

public GameObject[] panels; // The scrolling foregrounds
public float scrollSpeed = -30f;
// motionMult controls how much panels react to player movement
public float motionMult = 0.25f;

private float panelHt; // Height of each panel
private float depth; // Depth of panels (that is, pos.z)

// Use this for initialization
void Start () {
    panelHt = panels[0].transform.localScale.y;
    depth = panels[0].transform.position.z;

    // Set initial positions of panels
    panels[0].transform.position = new Vector3(0,0,depth);
    panels[1].transform.position = new Vector3(0,panelHt,depth);
}

// Update is called once per frame
void Update () {
    float tY, tX=0;
    tY= Time.time * scrollSpeed % panelHt + (panelHt*0.5f);

    if (poi != null) {
        tX = -poi.transform.position.x * motionMult;
    }

    // Position panels[0]
    panels[0].transform.position = new Vector3(tX, tY, depth);
    // Then position panels[1] where needed to make a continuous starfield
    if (tY >= 0) {
        panels[1].transform.position = new Vector3(tX, tY-panelHt, depth);
    } else {
        panels[1].transform.position = new Vector3(tX, tY+panelHt, depth);
    }
}

```

Save the script, return to Unity, and assign the script to `_MainCamera`. Select `_MainCamera` in the Hierarchy and find the *Parallax (Script)* component in the Inspector. There, set the `poi` to `_Hero` and add `StarfieldFG_0` and `StarfieldFG_1` to the `panels` array. Now press Play, and you should see the starfield moving in response to the player.

And of course, remember to save your scene.

Summary

This was a long chapter, but it introduced a lot of important concepts that I hope will help you with your own game projects in the future. Over the years, I have made extensive use of linear interpolation and Bézier curves to make the motion in my games and other projects smooth and refined. Just a simple easing function can make the movement of an object look graceful, excited, or lethargic, which is a powerful when you're trying to balance and tune the feel of a game.

In the next chapter, we move on to a very different kind of game: a solitaire card game (actually, my favorite solitaire card game). The next chapter demonstrates how to read information from an XML file to construct an entire deck of cards out of just a few art assets and also how to use XML to lay out the game itself. And, at the end, you'll have a fun digital card game to play.

Next Steps

From your experience in the previous tutorials, you already understand how to do many of the things listed in this section. These are just some recommendations on what you can do if you want to keep going with this prototype.

Tune Variables

As you have learned in both paper and digital games, tuning of numbers is critically important and has a significant effect on experience. The following is a list of variables you should consider tuning to change the feel of the game:

- **_Hero:** Change how movement feels
- Adjust the speed.
- Modify the gravity and sensitivity of the horizontal and vertical axes in the InputManager.
- **Weapons:** Differentiate weapons more
 - **Spread:** The spread gun could shoot five projectiles instead of just three but have a much longer `delayBetweenShots`.
 - **Blaster:** The blaster could fire more rapidly (smaller `delayBetweenShots`) but do less damage with each shot (reduced `damageOnHit`).
- **Power-ups:** Adjust drop rate
 - Each Enemy class has a `powerUpDropChance` field that can be set to any number between 0 (never drop a power-up) to 1 (always drop a power-up). These were set to 1 for testing, but you can adjust them to whatever you want.
 - It's also possible now for multiple Projectiles to hit an Enemy on the same turn that the Enemy's health drops to 0. This will cause multiple PowerUps to be spawned. Try to change the code to stop this from happening.

Add Additional Elements

While this prototype has so far shown five kinds of enemies and two kinds of weapons, there are infinite possibilities for either open to you:

- **Weapons:** Add additional weapons
 - **Phaser:** Shoots two projectiles that move in a sine wave pattern (similar to the movement of `Enemy_1`).
 - **Laser:** Instead of doing all of its damage at once, the laser does continuous damage over time.
 - **Missiles:** Missiles could have a lock-on mechanic and have a very slow fire-rate but would track enemies and always hit. Perhaps missiles could be a different kind of weapon with limited ammunition that were fired using a different button (that is, not the space bar).
 - **Swivel Gun:** Like the blaster but actually shoots at the nearest enemy. However, the gun is very weak.
- **Enemies:** Add additional enemies. There are countless kinds of enemies that could be created for this game.
 - Add additional enemy abilities
 - Allow some enemies to shoot.

- Some enemies could track and follow the player, possibly acting like missiles homing in on the player.
- Add level progression
 - Make specific, timed waves instead of the randomized infinite attack in the existing prototype. This could be accomplished using a [System.Serializable] Wave class as defined here:

[Click here to view code image](#)

```
[System.Serializable]
public class Wave {
    float delayBeforeWave=1; // secs to delay after the prev wave
    GameObject[] ships; // array of ships in this wave
    // Delay the next wave until this wave is completely killed?
    bool delayNextWaveUntilThisWaveIsDead=false;
}
```

- Add a Level class to contain the Wave [] array:

[Click here to view code image](#)

```
[System.Serializable]
public class Level {
    Wave[] waves; // Holder for waves
    float timeLimit=-1; // If -1, there is no time limit
    string name = ""; // If the name is left blank (i.e., ""), // the name could appear as "Level #1"
}
```

However, this will cause issues because even if Level is serializable, the Wave [] array won't appear properly because the Unity Inspector won't allow nested serializable classes. This means that you should probably try something like an XML document to define levels and waves which can then be read into Level and Wave classes. XML is covered in the “[XML](#)” section of [Appendix B](#) and is used in the next prototype, Prospector Solitaire.

- Add more game structure and GUI (graphical user interface) elements:
 - Give the player a score and a specific number of lives (both of these were covered in [Chapter 29](#)).
 - Add difficulty settings.
 - Track high scores (as covered in the Apple Picker and Mission Demolition prototypes).
 - Create a title screen scene that welcomes the player to the game and allows her to choose the difficulty setting. This could also show high scores.

Chapter 31. Prototype 4: Prospector Solitaire

In this chapter, you make your first digital card game. The game you create is a version of the popular *Tri-Peaks Solitaire* game that has been updated for the digital age.

This chapter includes several new techniques, including using XML configuration files, designing for mobile devices, and your first look at the 2D sprite tools that are included in Unity 4.3 and above.

By the end of the chapter, you'll have not only a working card game but also a great framework for future card games you wish to create.

Getting Started: Prototype 4

As with Prototype 3, this starts with you being asked to download and import a unitypackage of assets for this game. The art assets we'll be using are constructed from parts of the publicly available Vectorized Playing Cards 1.3 by Chris Aguilar.¹

¹ Chris Aguilar, “Vectorized Playing Cards 1.3,” <http://code.google.com/p/vectorized-playing-cards>. Copyright 2011—Chris Aguilar. Licensed under LGPL 3—www.gnu.org/copyleft/lesser.html.

It's also important to note that this project will work only with Unity 4.3 and later. In this chapter, we make extensive use of the Unity 2D tools that were introduced in version 4.3.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup Procedure.”](#) When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 2D for this project.

- **Project name:** Prospector.
- **Scene name:** _Prospector_Scene_0.
- **Project folders:** __Scripts, __Prefabs, __Sprites.
- **Download and import package:** Go to [Chapter 31](#) at <http://book.prototools.net>
- **C# script names:** (none yet).
- **Rename:** Change Main Camera to _MainCamera.

Importing the unitypackage should set _MainCamera correctly, but just in case, here are the settings:

_MainCamera (Camera)	P:[0,0,-40]	R:[0,0,0]	S:[1,1,1]
Projection: Orthographic			
Size: 10			

Note that this unitypackage includes a version of the Utils script that has additional functions beyond what you wrote in the previous chapter.

Build Settings

This will be the first project designed to be able to be compiled on mobile devices. As an example, I'll be using settings for the Apple iPad, but it's perfectly fine to use Android, BlackBerry, Windows 8 Phone, or even a Web Player build instead if you prefer. This project is designed for the 4:3 aspect ratio screen of an iPad in portrait mode. Though this project is designed to be able to be compiled for a mobile device, the actual build process for mobile devices is beyond the scope of this book (and would differ widely based on which device you own), but you can find a lot of information about doing so on Unity's website. The link for iOS development is <http://docs.unity3d.com/Documentation/Manual/iphone-GettingStarted.html>. There is also more information about compiling for mobile at the end of this chapter

1. Double-click the `__Prospector_Scene_0` scene in the Project pane to open it.
2. From the menu bar choose *File > Build Settings*, which will bring up the window shown in [Figure 31.1](#).

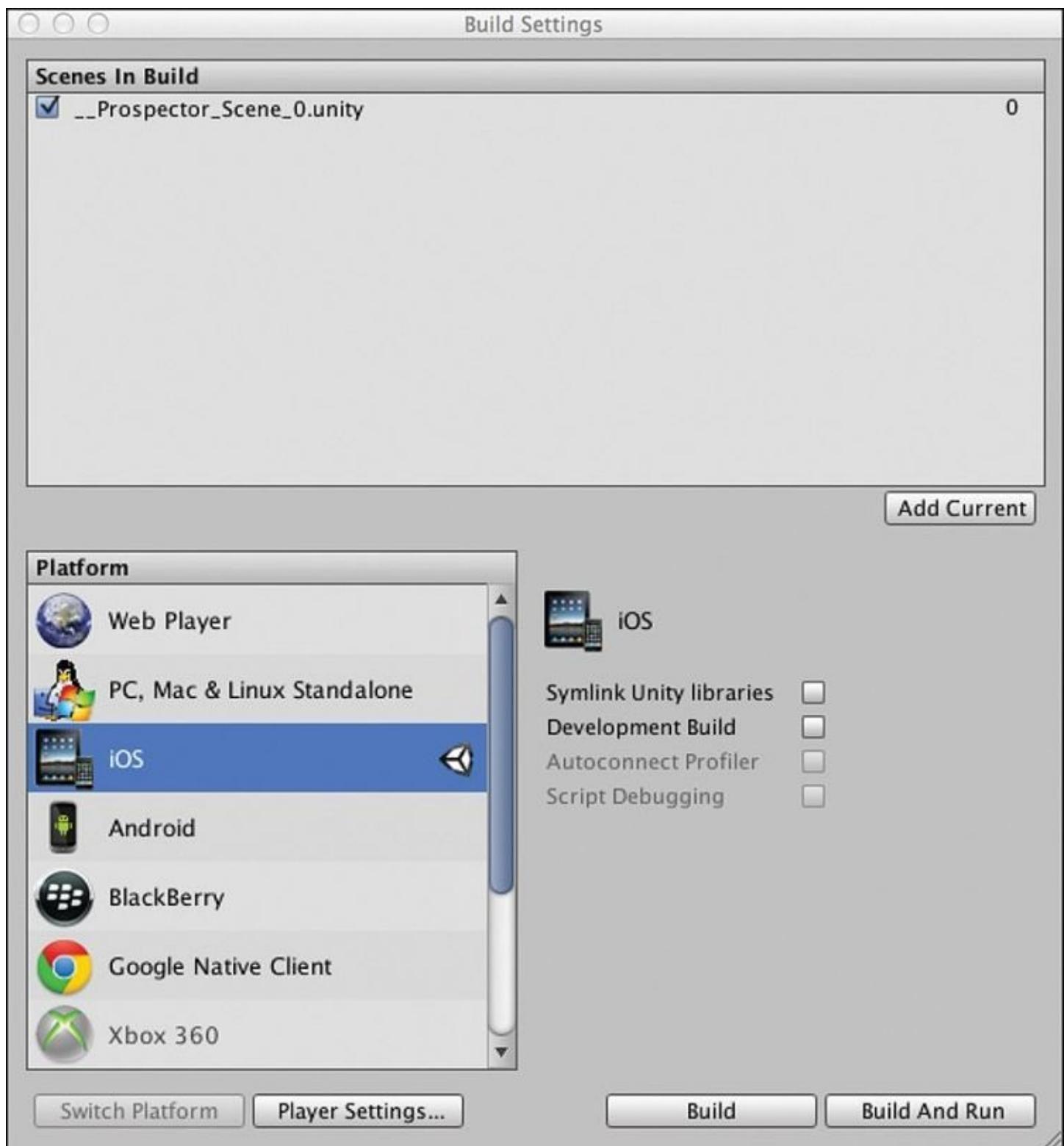


Figure 31.1 The Build Settings window

3. Click *Add Current* to add `__Prospector_Scene_0` to the list of scenes for this build.
4. Select iOS from the list of platforms and click *Switch Platform*. Unity will reimport all of your images to match the default iOS settings. The *Switch Platform* button will turn gray once the switch is complete. Once your Build Settings look like the image in [Figure 31.1](#), you can close this window. (Don't click *Build* yet; that would happen after actually making the game.)

Importing Images as Sprites

Next, we'll need to properly import the images we're using as *sprites*. A sprite is a 2D image that can be moved around the screen and rotated, and they are very common in 2D games:

1. Open the `_Sprites` folder in your Project pane and select all the images therein. (Click the top image and then Shift-click the bottom image in the `_Sprites` folder.) Looking at the Preview in the Inspector pane, you can see that all of them are currently imported as square images with no transparency. We're going to change that and make them usable sprites.
2. In the `21 Texture 2Ds Import Settings` section of the Inspector pane, set the *Texture Type* to *Sprite*. Then click *Apply*, and Unity will reimport all the images at their proper aspect ratio; however, you will see a warning in the console that “Only square textures can be compressed to PVRTC format.” Now change the *Format* in the Inspector from Compressed to *TrueColor* and click *Apply*. This will eliminate that warning, and your images should now be ready to be used as sprites. [Figure 31.2](#) shows the final import settings.



Figure 31.2 Import settings for the Texture 2Ds that will become sprites

3. Looking at the Project pane, you will see that each of the images now has a disclosure triangle

next to it. If you open the disclosure triangle, you'll find a sprite with the same name as the image under each image.

4. Select the Letters image in the Project pane. For most of the images that were imported, a Sprite Mode of Single is appropriate because each image becomes a single sprite. However, the Letters image is actually a sprite atlas (a series of sprites saved as a single image), so it requires different settings. In *Letters Import Settings* in the Inspector pane, change the Sprite Mode to *Multiple* and click *Apply*. This will add a new *Sprite Editor* button under the *Pixels to Units* field. Click this button to open the *Sprite Editor*. You'll see the Letters image there with a single blue box around it defining the bounds of the Letters sprite. Click the small icon with either a rainbow or a letter A on it in the Sprite Editor (circled in [Figure 31.3](#)) to switch between viewing the actual image and the alpha channel of the image. Because Letters is an image of white letters over a transparent background, it may be easier to see what's happening if you look at the alpha channel.
5. Now, click the *Slice* pop-up menu in the top-left corner of the Sprite Editor and change the *Type* from Automatic to Grid (see [Figure 31.3](#)). Set the *Pixel size* to X:32 Y:32 and click the *Slice* button. This will chop Letters horizontally into 16 sprites that are each 32x32 pixels in size. Click *Apply* (in the top-right corner of the Sprite Editor) to generate these sprites in the Project pane. Now instead of one Letters sprite, there are 16 sprites named Letters_0 to Letters_15. In this game, you will use Letters_1 to Letters_13 for each of the 13 ranks of cards (Ace through King). Now all the sprites are set up and ready to be used.

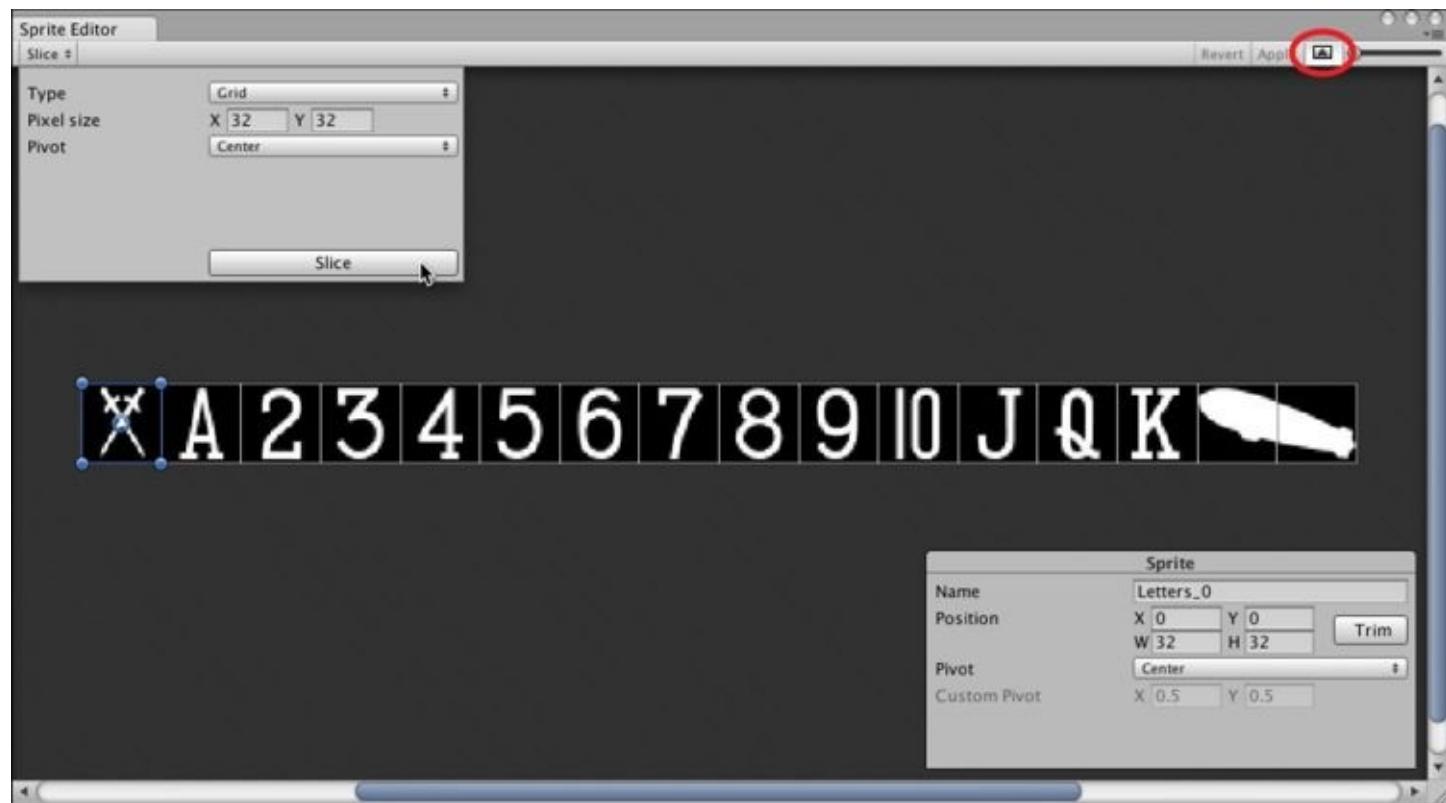


Figure 31.3 The Sprite Editor showing the correct settings for the grid slicing of Letters. The button circled in the top right switches between viewing the color channels and the alpha channel of Letters.

6. Save your scene. You haven't actually altered the scene yet, but it's good practice to save your scene all the time, so you should be in the habit of saving your scene any time you change anything.

Constructing Cards from Sprites

One of the most important aspects of this project is that we're going to procedurally construct an entire deck of cards from the 21 images that were imported. This will make the final build for mobile smaller and will give you a chance to see how XML works.

The image in [Figure 31.4](#) shows an example of how this will be done. The 10 of Spades in the image is constructed from Card_Front, 12 copies of Spade, and 2 copies of the Letters_10 sprite.

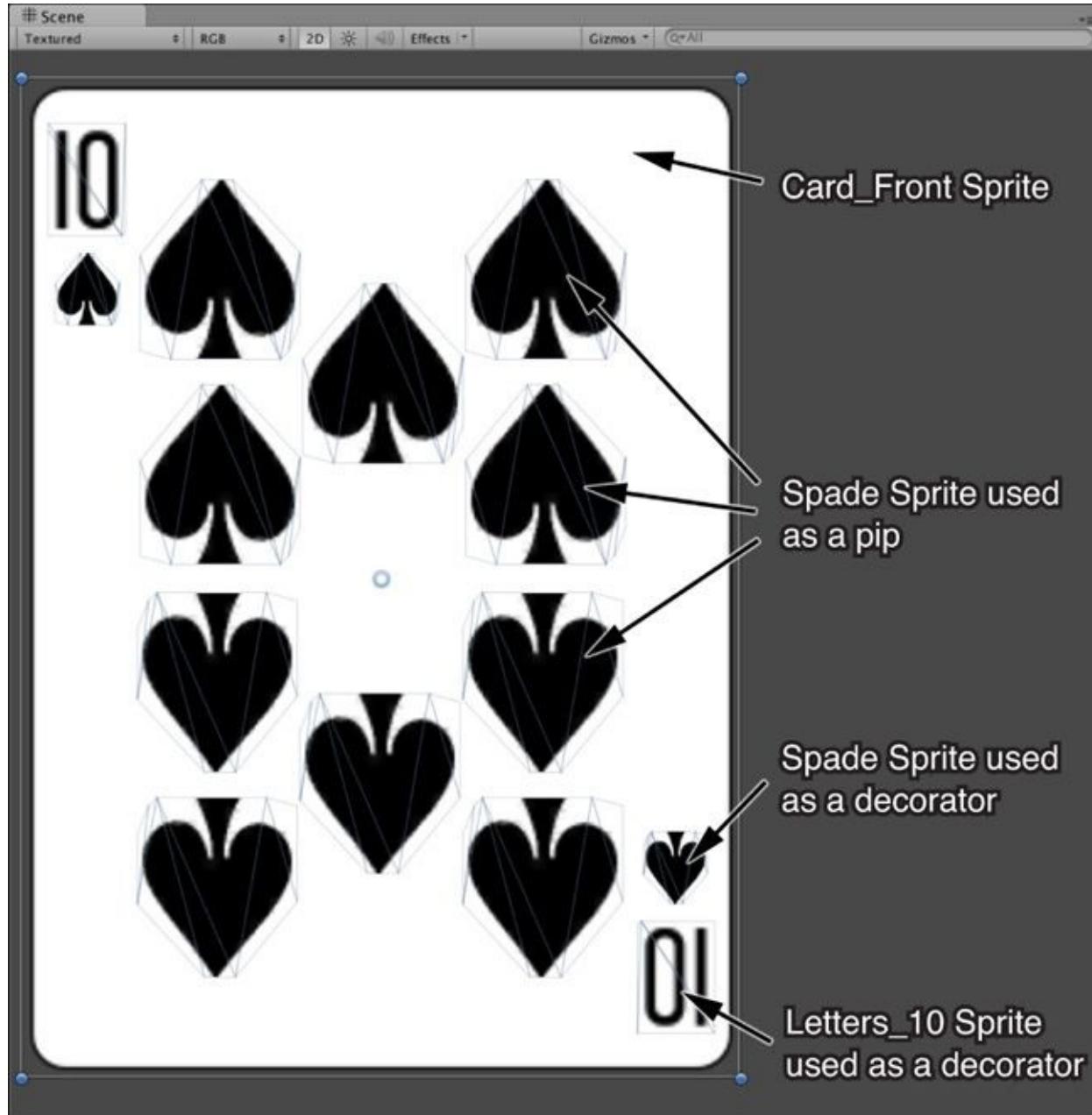


Figure 31.4 The 10 of Spades showing autogenerated borders around each of the sprites from which it is made. The visible part of this card is composed of 15 different sprites (12 Spades, 2 Letter_10s, and 1 Card_Front).

This is defined through the use of an XML file. Read the “[XML](#)” section of [Appendix B](#), “[Useful Concepts](#),” now to learn more about XML and how it can be read using the PT_XMLReader that was part of the imported unitypackage. The structure of the DeckXML.xml file used in this project is also shown in that section of [Appendix B](#).

Making Use of XML Through Code

For the first part of this project, create three C# files named *Card*, *Deck*, and *Prospector*.

- Card: The class for each individual card in the deck. Card will also contain the CardDefinition class (which holds information about where sprites are to be positioned on each rank of card) and the Decorator class (which holds information about the decorators and pips described in the XML document).
- Deck: The Deck class interprets the information in DeckXML.xml and uses that information to create an entire deck of cards.
- Prospector: The Prospector class manages the overall game. While Deck handles the creation of cards, Prospector turns those cards into a game. Prospector collects the cards into various piles (like the draw pile and discard pile) and manages game logic.

1. Start by creating the Card C# script and opening it in MonoDevelop. Enter the following code:

[Click here to view code image](#)

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

public class Card : MonoBehaviour {
    // This will be defined later
}

[System.Serializable]
public class Decorator {
    // This class stores information about each decorator or pip from DeckXML
    public string type; // For card pips, type = "pip"
    public Vector3 loc; // The location of the Sprite on the Card
    public bool flip = false; // Whether to flip the Sprite vertically
    public float scale = 1f; // The scale of the Sprite
}

[System.Serializable]
public class CardDefinition {
    // This class stores information for each rank of card
    public string face; // Sprite to use for each face card
    public int rank; // The rank (1-13) of this card
    public List<Decorator> pips = new List<Decorator>(); // Pips used
    // Because decorators (from the XML) are used the same way on every card in
    // the deck, pips only stores information about the pips on numbered cards
}
```

2. These small classes in Card.cs will store the information created when Deck reads the XML file. Open the Deck C# script in MonoDevelop and enter the following code:

[Click here to view code image](#)

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

public class Deck : MonoBehaviour {

    public bool _____;
    public PT_XMLReader xmlr;

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        ReadDeck(deckXMLText);
    }
}
```

```

// ReadDeck parses the XML file passed to it into CardDefinitions
public void ReadDeck(string deckXMLText) {
    xmlr = new PT_XMLReader(); // Create a new PT_XMLReader
    xmlr.Parse(deckXMLText); // Use that PT_XMLReader to parse DeckXML

    // This prints a test line to show you how xmlr can be used.
    // For more information read about XML in the Useful Concepts Appendix
    string s = "xml[0] decorator[0] ";
    s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
    s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
    s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
    s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
    print(s);
}

}

```

The lines that grab data from the parsed XML may look a little strange at first. `xmlr` is the `PT_XMLReader` instance, and `xmlr.xml` is the parsed XML. Bracket access is then used to delve into the XML in a nested way. `xmlr.xml["xml"]` grabs a collection of all the `<xml>` elements at the top level of the XML file, and `xmlr.xml["xml"][0]` gets the 0th element of that collection. This continues, digging down into the various elements of the XML file to get the attributes (e.g., `att("type")`) of the decorators.

3. Now open the `Prospector` class and enter this code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Prospector : MonoBehaviour {
    static public Prospector S;

    public Deck deck;
    public TextAsset deckXML;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }

    void Start () {
        deck = GetComponent<Deck>(); // Get the Deck
        deck.InitDeck(deckXML.text); // Pass DeckXML to it
    }
}

```

4. Now that the code is ready, go back to Unity and attach both the `Prospector` and `Deck` classes to `_MainCamera`. (Drag each script from the Project pane onto `_MainCamera` in the Hierarchy pane.) Then select `_MainCamera` in the Hierarchy. You should see both scripts attached as Script components. Drag `DeckXML` from the Resources folder in the Project pane into the `deckXML` `TextAsset` variable in the Inspector for the `Prospector` (Script) component.

5. Save your scene and click Play. You should see the following output in the console:

[Click here to view code image](#)

```
xml[0] decorator[0] type=letter x=-1.05 y=1.42 scale=1.25
```

This line comes from the test code in `Deck:ReadDeck()` and shows that `ReadDeck()` is properly reading the type, x, y, and scale attributes from the 0th decorator of the 0th xml in the XML file, as shown in the following lines from `DeckXML.xml`. (You can double-click the `DeckXML` file in the Resources folder to open it in MonoDevelop and view the entire thing.)

[Click here to view code image](#)

```
<xml>
  <decorator type="letter" x="-1.05" y="1.42" z="0" flip="0" scale="1.25"/>
  ...
</xml>
```

Now, let's actually do something with this information.

1. Make the following bolded changes to the `Deck` class:

[Click here to view code image](#)

```
public class Deck : MonoBehaviour {

  public bool _____;

  public PT_XMLReader
  public List<string> _____;
  public List<Card> _____;
  public List<Decorator> _____;
  public List<CardDefinition> _____;
  public Transform _____;
  public Dictionary<string,Sprite> _____;

  // InitDeck is called by Prospector when it is ready
  public void InitDeck(string deckXMLText) {
    ReadDeck(deckXMLText);
  }

  // ReadDeck parses the XML file passed to it into CardDefinitions
  public void ReadDeck(string deckXMLText) {
    xmlr = new PT_XMLReader(); // Create a new PT_XMLReader
    xmlr.Parse(deckXMLText); // Use that PT_XMLReader to parse DeckXML

    // This prints a test line to show you how xmlr can be used.
    // For more information read about XML in the Useful Concepts Appendix
    string s = "xml[0] decorator[0] ";
    s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
    s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
    s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
    s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
    //print(s); // Comment out this line, since we're done with the test

    // Read decorators for all Cards
    decorators = new List<Decorator>(); // Init the List of Decorators
    // Grab a PT_XMLHashList of all <decorator>s in the XML file
    PT_XMLHashList xDecos = xmlr.xml["xml"][0]["decorator"];
    Decorator deco;
    for (int i=0; i<xDecos.Count; i++) {
      // For each <decorator> in the XML
      deco = new Decorator(); // Make a new Decorator
      // Copy the attributes of the <decorator> to the Decorator
      deco.type = xDecos[i].att("type");
      // Set the bool flip based on whether the text of the attribute is
      // "1" or something else. This is an atypical but perfectly fine
      // use of the == comparison operator. It will return a true or
      // false, which will be assigned to deco.flip.
```

```

deco.flip = ( xDecos[i].att ("flip") == "1" );
// floats need to be parsed from the attribute strings
deco.scale = float.Parse( xDecos[i].att ("scale") );
// Vector3 loc initializes to [0,0,0], so we just need to modify it
deco.loc.x = float.Parse( xDecos[i].att ("x") );
deco.loc.y = float.Parse( xDecos[i].att ("y") );
deco.loc.z = float.Parse( xDecos[i].att ("z") );
// Add the temporary deco to the List decorators
decorators.Add (deco);
}

// Read pip locations for each card number
cardDefs = new List<CardDefinition>(); // Init the List of Cards
// Grab a PT_XMLHashList of all the <card>s in the XML file
PT_XMLHashList xCardDefs = xmlr.xml["xml"][0]["card"];
for (int i=0; i<xCardDefs.Count; i++) {
    // For each of the <card>s
    // Create a new CardDefinition
    CardDefinition cDef = new CardDefinition();
    // Parse the attribute values and add them to cDef
    cDef.rank = int.Parse( xCardDefs[i].att ("rank") );
    // Grab a PT_XMLHashList of all the <pip>s on this <card>
    PT_XMLHashList xPips = xCardDefs[i]["pip"];
    if (xPips != null) {
        for (int j=0; j<xPips.Count; j++) {
            // Iterate through all the <pip>s
            deco = new Decorator();
            // <pip>s on the <card> are handled via the Decorator Class
            deco.type = "pip";
            deco.flip = ( xPips[j].att ("flip") == "1" );
            deco.loc.x = float.Parse( xPips[j].att ("x") );
            deco.loc.y = float.Parse( xPips[j].att ("y") );
            deco.loc.z = float.Parse( xPips[j].att ("z") );
            if ( xPips[j].HasAtt("scale") ) {
                deco.scale = float.Parse( xPips[j].att ("scale") );
            }
            cDef.pips.Add(deco);
        }
    }
    // Face cards (Jack, Queen, & King) have a face attribute
    // cDef.face is the base name of the face card Sprite
    // e.g., FaceCard_11 is the base name for the Jack face Sprites
    // the Jack of Clubs is FaceCard_11C, hearts is FaceCard_11H, etc.
    if (xCardDefs[i].HasAtt("face")) {
        cDef.face = xCardDefs[i].att ("face");
    }
    cardDefs.Add(cDef);
}
}
}

```

2. Now, the `ReadDeck()` method will parse the XML and turn it into a list of Decorators (the suit and rank in the corners of the card) and CardDefinitions (a class containing information about each of the ranks of card (Ace through King). Switch back to Unity and press Play. Then click on the `_MainCamera` and look at the Inspector for the Deck (Script) component. Because both `Decorator` and `CardDefinition` were set to `[System.Serializable]`, they appear properly in the Unity Inspector, as shown in [Figure 31.5](#). (Note that this is due to an improvement in the Inspector for Unity 4.3.)

3. Stop playback and save your scene.

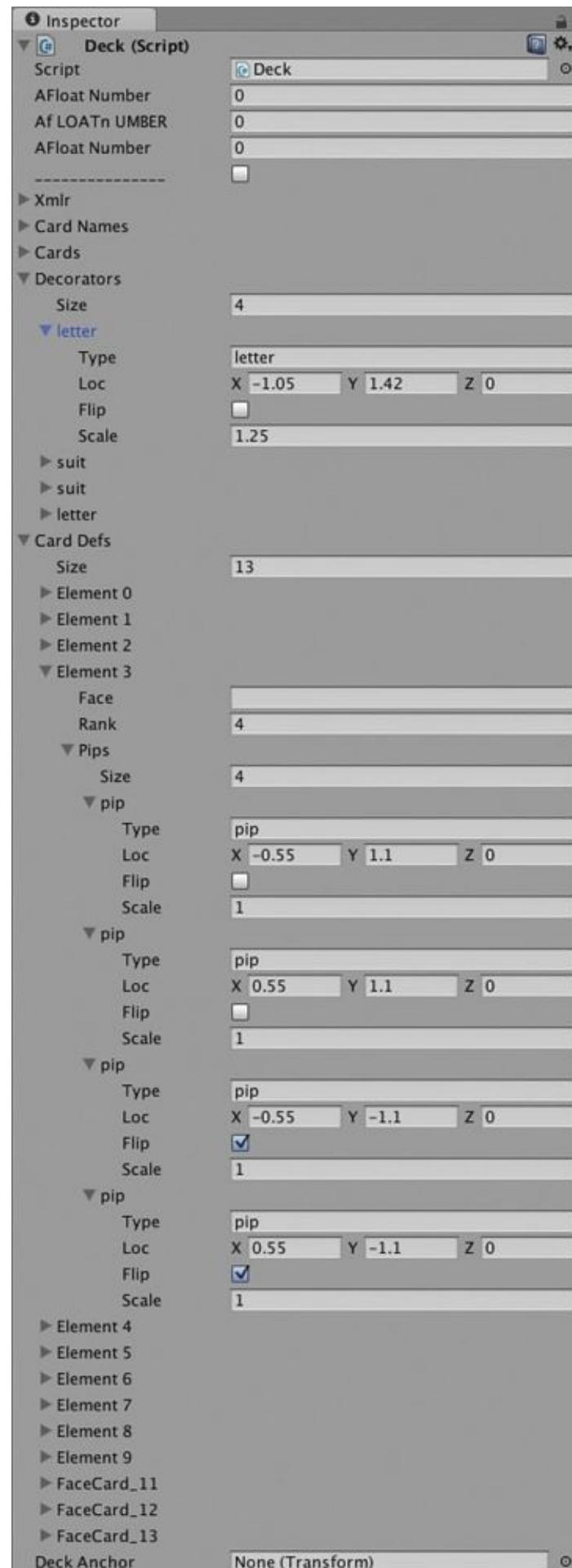


Figure 31.5 The Inspector for the Deck (Script) component of _MainCamera showing Decorators and Card Defs that have been read from the DecXML.xml file

Assigning the Sprites That Become Cards

Now that the XML has been properly read and parsed into usable Lists, it's time to make some cards. The first step in doing so is to get references to all of those sprites that we made earlier in the chapter:

1. Add the following variables to the top of the Deck class to hold these sprites:

[Click here to view code image](#)

```
public class Deck : MonoBehaviour {
    // Suits
    public Sprite             suitClub;
    public Sprite             suitDiamond;
    public Sprite             suitHeart;
    public Sprite             suitSpade;

    public Sprite[]           faceSprites;
    public Sprite[]           rankSprites;

    public Sprite             cardBack;
    public Sprite             cardBackGold;
    public Sprite             cardFront;
    public Sprite             cardFrontGold;

    // Prefabs
    public GameObject         prefabSprite;
    public GameObject         prefabCard;

    public bool _____;
}
```

When you switch back to Unity, there will now be many new public variables that need to be defined in the Deck (Sprite) Inspector.

2. Drag the Club, Diamond, Heart, and Spade textures from the _Sprites folder in the Project pane into their respective variables under Deck (suitClub, suitDiamond, suitHeart, and suitSpade). Unity will automatically assign the sprite to the variable (as opposed to attempting to assign the Texture2D to a sprite variable).
3. The next bit is a touch trickier. Lock the Inspector on _MainCamera by selecting _MainCamera in the Hierarchy pane and then clicking the tiny lock icon at the top of the Inspector pane (circled in red in the [Figure 31.6](#)). Locking the Inspector pane ensures that it won't change which object is displayed when you select something new.
4. Disclose the triangle next to the variable faceSprites in the Inspector for Deck (Script) and set its Size to 12. Now, drag each of the Texture2Ds with names that start with *FaceCard_* into an element of faceSprites. The order doesn't matter as long as there is exactly one of each when you're done (see [Figure 31.6](#)).

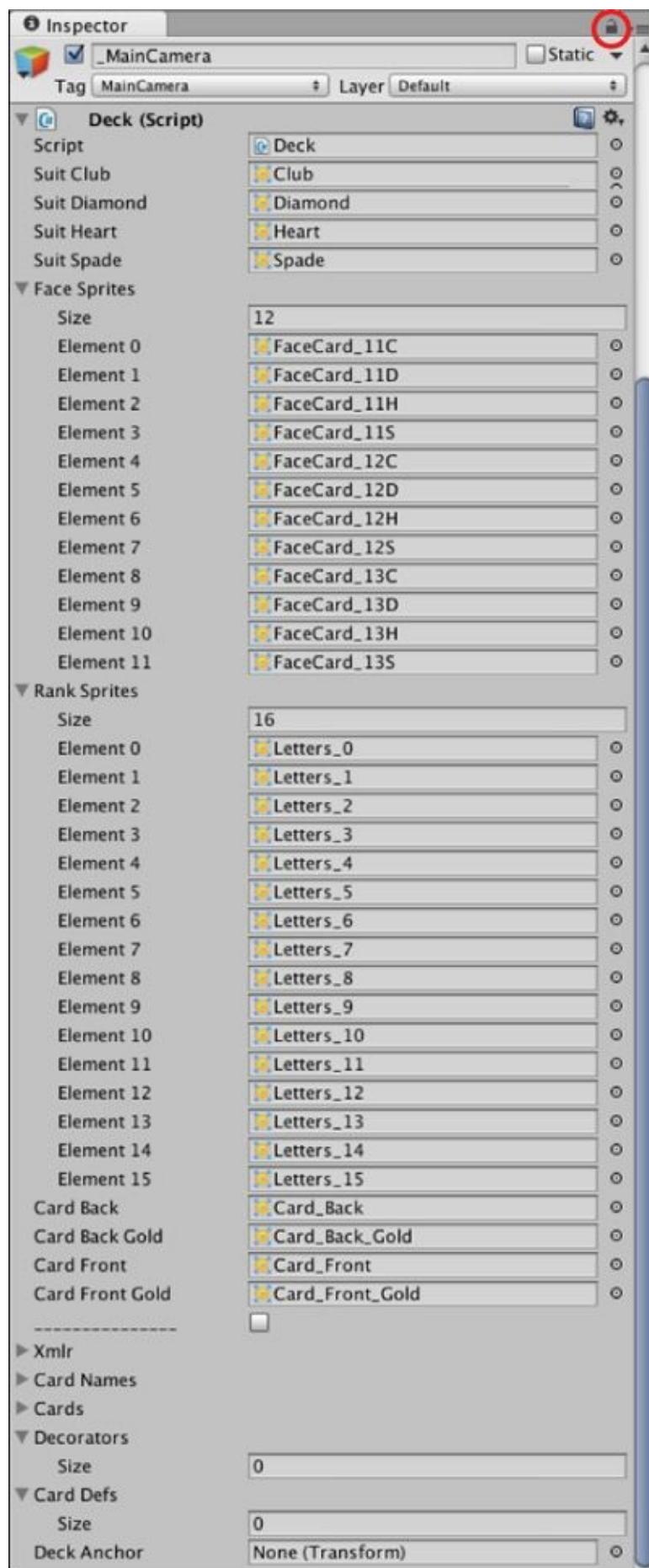


Figure 31.6 The Inspector for the Deck (Script) Component of _MainCamera showing the correct sprites assigned to each public sprite variable

5. Open the disclosure triangle next to Letters in the Project pane. Select Letters_0 and then Shift-click Letters_15. You should now have all 16 sprites under Letters selected. Drag this group of

sprites onto the `rankSprites` variable in `Deck` (Script). You should see a plus icon appear next to the word `<multiple>` when hovering over the variable name `rankSprites` (on PC, you may only see the + icon). Release the mouse button, and if done correctly, the `rankSprites` list should now be full of 16 Letters sprites named `Letters_0` through `Letters_15`. Double-check to make sure that they're in the correct order with `Letters_0` in Element 0 and `Letters_15` in Element 15; if not, you may have to add them one at a time.

6. Drag the sprites `Card_Back`, `Card_Back_Gold`, `Card_Front`, and `Card_Front_Gold` from the Project pane to their respective variable slots in the `Deck` (Script) Inspector.
7. Unlock the Inspector pane by clicking the tiny lock icon again (circled in red in [Figure 31.6](#)). Your Inspector for `Deck` (Script) should now look like what is shown in the figure.

Creating Prefab GameObjects for Sprites and Cards

Just like anything else on screen, sprites need to be enclosed in `GameObjects`. For this project, you will need two prefabs: a generic `PrefabSprite` that will be used for all decorators and pips, and a `PrefabCard` that will form the basis of all the cards in the deck.

PrefabSprite

To create `PrefabSprite`, do the following:

1. From the menu bar, choose `GameObject > Create Other > Sprite`.
2. Name this `GameObject` `PrefabSprite` and drag it into the `_Prefabs` folder of the Project pane to make it a prefab.
3. Delete the remaining `PrefabSprite` instance from the Hierarchy.

PrefabCard

To create `PrefabCard`, do the following:

1. From the menu bar, choose `GameObject > Create Other > Sprite`. Name this `GameObject` `PrefabCard`.
2. Drag `Card_Front` from the Project pane into the `sprite` variable of the `Sprite Renderer` in the `PrefabCard` Inspector. Now you should see the `Card_Front` sprite in the Scene pane.
3. Drag the `Card` script from the Project pane onto `PrefabCard` in the Hierarchy. This will assign the `Card` script to `PrefabCard` (and the `Card` (Script) component will now appear in the Inspector for `PrefabCard`).
4. In the Inspector for `PrefabCard`, click the `Add Component` button. Choose `Physics > Box Collider` from the menu that appears. (This is the same as choosing `Component > Physics > Box Collider` from the menu bar.) The `Size` of the `Box Collider` should automatically set itself to `[2.56, 3.56, 0.2]`, but if not, set the `Size` to those values.
5. Drag `PrefabCard` from the Hierarchy into the `_Prefabs` folder to make a prefab from it. Delete the remaining instance of `PrefabCard` from the Hierarchy.

Now, you need to assign these two prefabs to their respective public variables in the Inspector for the `Deck` (Script) Component on `_MainCamera`.

1. Select `_MainCamera` in the hierarchy, and drag `PrefabCard` and `PrefabSprite` from the Project pane into their respective variables in the `Deck` (Script) Inspector.
2. Save your scene.

Building the Cards in Code

Before actually adding the method to the Deck class to make the cards, we need to add variables to Card, as follows:

1. Add the following code to the Card C# script:

[Click here to view code image](#)

```
public class Card : MonoBehaviour {
    public string suit; // Suit of the Card (C, D, H, or S)
    public int rank; // Rank of the Card (1-14)
    public Color color = Color.black; // Color to tint pips
    public string colS = "Black"; // or "Red". Name of the Color

    // This List holds all of the Decorator GameObjects
    public List<GameObject> decoGOs = new List<GameObject>();
    // This List holds all of the Pip GameObjects
    public List<GameObject> pipGOs = new List<GameObject>();

    public GameObject back; // The GameObject of the back of the card
    public CardDefinition def; // Parsed from DeckXML.xml
}
```

2. And now, add this code to Deck:

[Click here to view code image](#)

```
public class Deck : MonoBehaviour {
    ...

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        // This creates an anchor for all the Card GameObjects in the Hierarchy
        if (GameObject.Find("_Deck") == null) {
            GameObject anchorGO = new GameObject("_Deck");
            deckAnchor = anchorGO.transform;
        }

        // Initialize the Dictionary of SuitSprites with necessary Sprites
        dictSuits = new Dictionary<string, Sprite>() {
            { "C", suitClub },
            { "D", suitDiamond },
            { "H", suitHeart },
            { "S", suitSpade }
        };

        ReadDeck(deckXMLText);
        MakeCards();
    }

    // ReadDeck parses the XML file passed to it into CardDefinitions
    public void ReadDeck(string deckXMLText) {
        ...
    }

    // Get the proper CardDefinition based on Rank (1 to 14 is Ace to King)
    public CardDefinition GetCardDefinitionByRank(int rnk) {
        // Search through all of the CardDefinitions
        foreach (CardDefinition cd in cardDefs) {
            // If the rank is correct, return this definition
            if (cd.rank == rnk) {
                return( cd );
            }
        }
    }
}
```

```

        }
    }
    return( null );
}

// Make the Card GameObjects
public void MakeCards() {
    // cardNames will be the names of cards to build
    // Each suit goes from 1 to 13 (e.g., C1 to C13 for Clubs)
    cardNames = new List<string>();
    string[] letters = new string[] {"C", "D", "H", "S"};
    foreach (string s in letters) {
        for (int i=0; i<13; i++) {
            cardNames.Add(s+(i+1));
        }
    }
}

// Make a List to hold all the cards
cards = new List<Card>();
// Several variables that will be reused several times
Sprite tS = null;
GameObject tGO = null;
SpriteRenderer tSR = null;

// Iterate through all of the card names that were just made
for (int i=0; i<cardNames.Count; i++) {
    // Create a new Card GameObject
    GameObject cgo = Instantiate(prefabCard) as GameObject;
    // Set the transform.parent of the new card to the anchor.
    cgo.transform.parent = deckAnchor;
    Card card = cgo.GetComponent<Card>(); // Get the Card Component

    // This just stacks the cards so that they're all in nice rows
    cgo.transform.localPosition = new Vector3( (i%13)*3, i/13*4, 0 );

    // Assign basic values to the Card
    card.name = cardNames[i];
    card.suit = card.name[0].ToString();
    card.rank = int.Parse( card.name.Substring(1) );
    if (card.suit == "D" || card.suit == "H") {
        card.colS = "Red";
        card.color = Color.red;
    }
    // Pull the CardDefinition for this card
    card.def = GetCardDefinitionByRank(card.rank);

    // Add Decorators
    foreach( Decorator deco in decorators ) {
        if (deco.type == "suit") {
            // Instantiate a Sprite GameObject
            tGO = Instantiate( prefabSprite ) as GameObject;
            // Get the SpriteRenderer Component
            tSR = tGO.GetComponent<SpriteRenderer>();
            // Set the Sprite to the proper suit
            tSR.sprite = dictSuits[card.suit];
        } else { //if it's not a suit, it's a rank deco
            tGO = Instantiate( prefabSprite ) as GameObject;
            tSR = tGO.GetComponent<SpriteRenderer>();
            // Get the proper Sprite to show this rank
            ts = rankSprites[ card.rank ];
            // Assign this rank Sprite to the SpriteRenderer
        }
    }
}

```

```
    tSR.sprite = tS;
    // Set the color of the rank to match the suit
    tSR.color = card.color;
}
// Make the deco Sprites render above the Card
tSR.sortingOrder = 1;
// Make the decorator Sprite a child of the Card
tGO.transform.parent = cgo.transform;
// Set the localPosition based on the location from DeckXML
tGO.transform.localPosition = deco.loc;
// Flip the decorator if needed
if (deco.flip) {
    // An Euler rotation of 180° around the Z-axis will flip it
    tGO.transform.rotation = Quaternion.Euler(0,0,180);
}
// Set the scale to keep decos from being too big
if (deco.scale != 1) {
    tGO.transform.localScale = Vector3.one * deco.scale;
}
// Name this GameObject so it's easy to find
tGO.name = deco.type;
// Add this deco GameObject to the List card.decoGOs
card.decoGOs.Add(tGO);

}
// Add the card to the deck
cards.Add (card);
}
}
```

3. Press Play. You should see 52 cards lined up. They don't yet have pips, but they do appear, and the correct decorators and coloring are on them. Now let's add the code for pips and faces. Add the following to the `MakeCards()` method of the `Deck` class:

[Click here to view code image](#)

```
// Make the Card GameObjects
public void MakeCards() {
    ...
    // Iterate through all of the card names that were just made
    for (int i=0; i<cardNames.Count; i++) {
        ...
        // Add Decorators
        foreach( Decorator deco in decorators ) {
            ...
        }

        // Add Pips
        // For each of the pips in the definition
        foreach( Decorator pip in card.def.pips ) {
            // Instantiate a Sprite GameObject
            tGO = Instantiate( prefabSprite ) as GameObject;
            // Set the parent to be the card GameObject
            tGO.transform.parent = cgo.transform;
            // Set the position to that specified in the XML
            tGO.transform.localPosition = pip.loc;
            // Flip it if necessary
            if (pip.flip) {
                tGO.transform.rotation = Quaternion.Euler(0,0,180);
            }
        }
    }
}
```

```

        // Scale it if necessary (only for the Ace)
        if (pip.scale != 1) {
            tGO.transform.localScale = Vector3.one * pip.scale;
        }
        // Give this GameObject a name
        tGO.name = "pip";
        // Get the SpriteRenderer Component
        tSR = tGO.GetComponent<SpriteRenderer>();
        // Set the Sprite to the proper suit
        tSR.sprite = dictSuits[card.suit];
        // Set sortingOrder so the pip is rendered above the Card_Front
        tSR.sortingOrder = 1;
        // Add this to the Card's list of pips
        card.pipGOs.Add(tGO);
    }

    // Handle Face Cards
    if (card.def.face != "") { // If this has a face in card.def
        tGO = Instantiate( prefabSprite ) as GameObject;
        tSR = tGO.GetComponent<SpriteRenderer>();
        // Generate the right name and pass it to GetFace()
        tS = GetFace( card.def.face+card.suit );
        tSR.sprite = tS;          // Assign this Sprite to tSR
        tSR.sortingOrder = 1; // Set the sortingOrder
        tGO.transform.parent = card.transform;
        tGO.transform.localPosition = Vector3.zero;
        tGO.name = "face";
    }

    // Add the card to the deck
    cards.Add (card);

}

} // This is the closing brace for MakeCards()

// Find the proper face card Sprite
public Sprite GetFace(string faceS) {
    foreach (Sprite tS in faceSprites) {
        // If this Sprite has the right name...
        if (tS.name == faceS) {
            // ...then return the Sprite
            return( tS );
        }
    }
    // If nothing can be found, return null
    return( null );
}

```

4. Pressing Play now, you should see all 52 cards laid out properly with pips and faces for face cards. The next thing to do is add a back to the cards. The back will have a higher sorting order than anything else on the card, and it will be visible when the card is face-down and invisible when the card is face-up.

To accomplish this visibility toggle, add the following `faceUp` property to the `Card` class. As a property, `faceUp` is actually two functions (a get and a set) masquerading as a single field:

[Click here to view code image](#)

```

public class Card : MonoBehaviour {
    ...
    public CardDefinition def; // Parsed from DeckXML.xml

    public bool faceUp {

```

```

        get {
            return( !back.activeSelf );
        }
        set {
            back.SetActive(!value);
        }
    }
}

```

5. Now, the back can be added to the card in `MakeCards()`. Add the following lines to the `MakeCards()` method of the `Deck` class:

[Click here to view code image](#)

```

// Make the Card GameObjects
public void MakeCards() {
    ...

    // Iterate through all of the card names that were just made
    for (int i=0; i<cardNames.Count; i++) {
        ...
        // Handle Face Cards
        if (card.def.face != "") { // If this has a face in card.def
            ...
        }

        // Add Card Back
        // The Card_Back will be able to cover everything else on the Card
        tGO = Instantiate( prefabSprite ) as GameObject;
        tSR = tGO.GetComponent<SpriteRenderer>();
        tSR.sprite = cardBack;
        tGO.transform.parent = card.transform;
        tGO.transform.localPosition = Vector3.zero;
        // This is a higher sortingOrder than anything else
        tSR.sortingOrder = 2;
        tGO.name = "back";
        card.back = tGO;

        // Default to face-up
        card.faceUp = false; // Use the property faceUp of Card

        // Add the card to the deck
        cards.Add (card);
    }
}

```

6. Press Play, and you'll see that all the cards are now flipped face-down. If you change the last added line to `card.faceUp = true;`, all of them will be face-up.

Shuffling the Cards

Now that cards can be built and displayed on screen, the last generic thing that we will need from the `Deck` class is the ability to shuffle cards.

1. Add the following `Shuffle()` method to end of the `Deck` class:

[Click here to view code image](#)

```

public class Deck : MonoBehaviour {
    ...
    // Shuffle the Cards in Deck.cards
}

```

```

static public void Shuffle(ref List<Card> oCards) { // 1
    // Create a temporary List to hold the new shuffle order
    List<Card> tCards = new List<Card>();

    int ndx; // This will hold the index of the card to be moved
    tCards = new List<Card>(); // Initialize the temporary List
    // Repeat as long as there are cards in the original List
    while (oCards.Count > 0) {
        // Pick the index of a random card
        ndx = Random.Range(0, oCards.Count);
        // Add that card to the temporary List
        tCards.Add (oCards[ndx]);
        // And remove that card from the original List
        oCards.RemoveAt(ndx);
    }
    // Replace the original List with the temporary List
    oCards = tCards;
    // Because oCards is a reference variable, the original that was
    // passed in is changed as well.
}
}

```

At the line marked by // 1 in the preceding code listing, the `ref` keyword is used to make sure that the `List<Card>` that is passed to `List<Card> oCards` is passed as a reference rather than copied into `oCards`. This means that anything that happens to `oCards` is actually happening to the variable that is passed in. In other words, if the cards of a `Deck` are passed in via reference, those cards will be shuffled without requiring a return variable.

2. Make the following change to the `Prospector` script to see this work:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    static public Prospector S;

    public Deck deck;
    public TextAsset deckXML;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }

    void Start () {
        deck = GetComponent<Deck>(); // Get the Deck
        deck.InitDeck(deckXML.text); // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards); // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which allows
        // deck.cards to be modified by Deck.Shuffle()
    }
}

```

3. If you play the scene now, you can select `_MainCamera` in the scene Hierarchy and look at the `Deck.cards` variable to see a shuffled deck of cards.

Now that the `Deck` class can shuffle any list of cards, you have the basic tools to create *any* card game. The game you will make in this prototype is called `Prospector`.

The Prospector Game

The code up till now has given you the basic tools to make any card game. Now let's talk about the specific game we're going to make.

Prospector is based on the classic solitaire card game *Tri-Peaks*. The rules of both are the same, except for two things:

1. The premise of *Prospector* is that the player is digging down for gold, whereas the premise of *Tri-Peaks* is that the player is trying to climb three mountains.
2. The objective of *Tri-Peaks* is just to clear all of the cards. The objective of *Prospector* is to earn points by having long runs of cards played without having to draw a card from the draw pile, and each gold card in the run doubles the value of the whole run.

Prospector Rules

To try this out, grab a normal deck of playing cards (like a physical, real deck, not the virtual one we just made). Remove the Jokers and shuffle the remaining 52 cards:

1. Lay out 28 of the cards as shown in the [Figure 31.7](#). The bottom three rows of cards should be face-down, and the top row is face-up. The card edges don't need to be touching, but the lower cards do need to be covered by the upper cards. This sets up the initial tableau of cards for the "mine" that our prospector will be excavating.

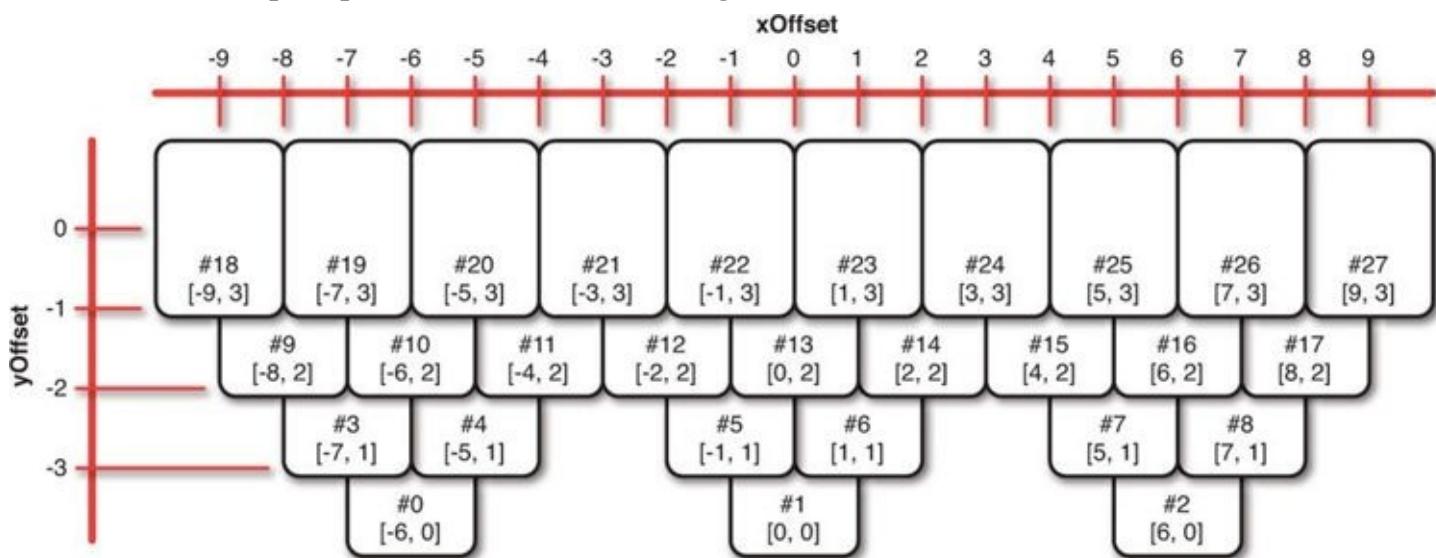


Figure 31.7 The initial tableau layout for the mine in Prospector

2. The rest of the deck forms a draw pile. Place it above the top row of cards face-down.
3. Draw the top card from the draw deck and place it face-up and centered above the top row of cards. This is the target card.
4. Any card that is either exactly one rank above or below the target card may be moved from the tableau onto the target card, making it the new target. Aces and Kings wrap around, so an Ace can be played on a King and vice versa.
5. If a face-down card is no longer covered by a card from a higher row, it can be turned face-up.
6. If none of the face-up cards can be played on the target card, draw a new target card from the draw pile.
7. If the tableau is emptied before the draw pile, you win! (Scoring and gold cards will be saved for the digital version of the game.)

Example of Play

The image in [Figure 31.8](#) shows an example initial layout for Prospector. In the situation shown, the player can initially play either the 9C (9 of Clubs) or the 7S (7 of Spades) onto the 8H.

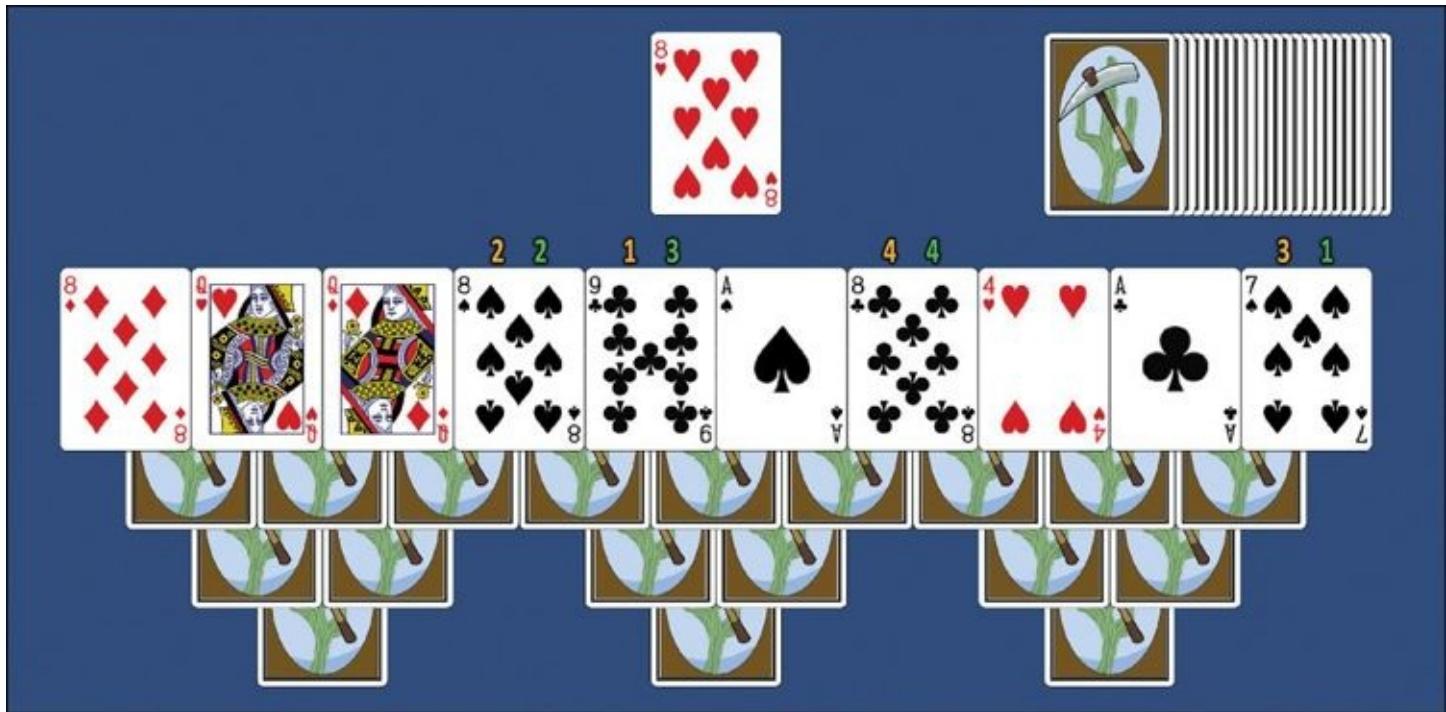


Figure 31.8 An example initial layout for Prospector

The amber and green numbers show two possible four-card runs. In the amber run, the 9C is played, becoming the new target card. This allows the play of 8S, 8D, or 8C. The player chooses 8S because it will then reveal the card that was hidden by 9C and 8S. Then the amber run continues with 7S and finally 8C. This results in the layout shown in [Figure 31.9](#).

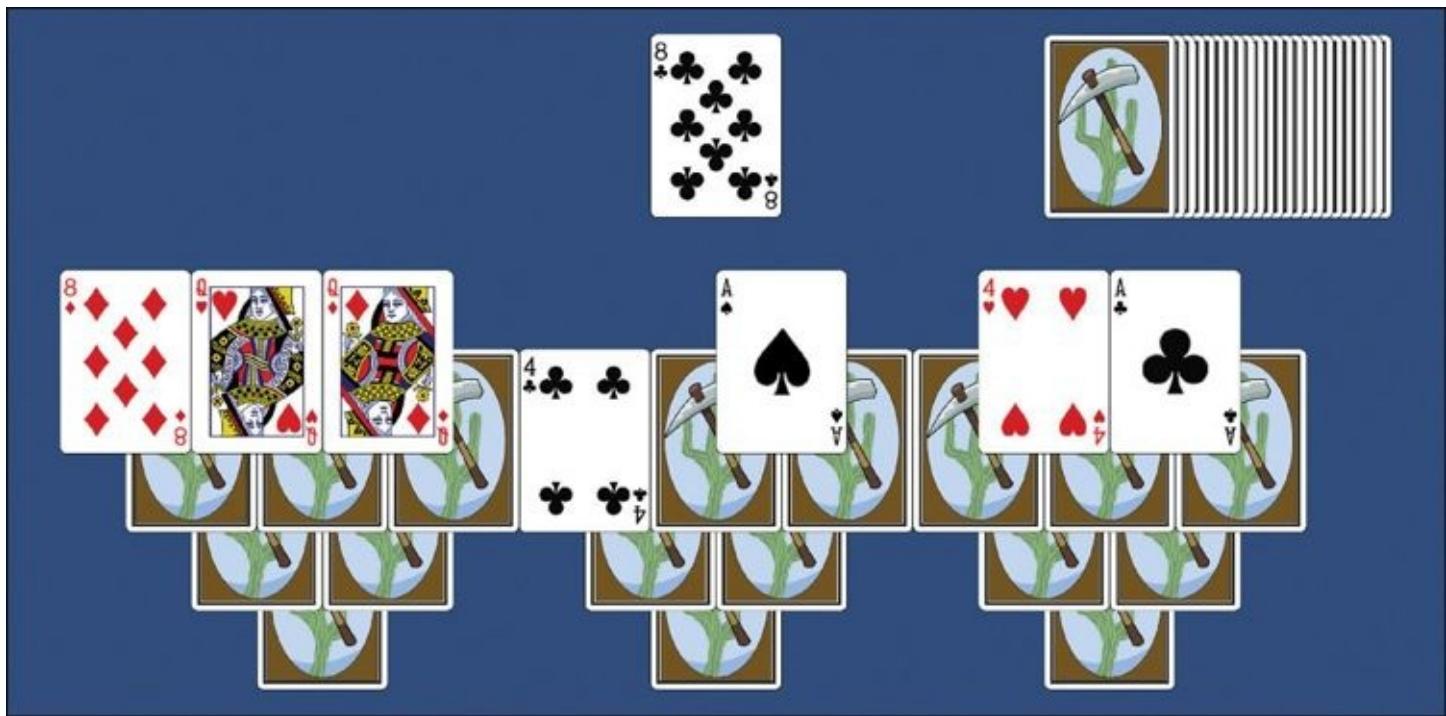


Figure 31.9 The Prospector example game after the first run

Now, because there are no more valid face-up cards to play from the tableau, the player must draw a card from the draw pile to become the next target card. This ends the first run of the game.

Try playing the game a few times to get a feel for it.

Implementing Prospector in Code

As you have seen from playing, Prospector is a pretty simple game, but it's also pretty fun. We can add to that fun later with some nice visuals and scoring tweaks, but for now, let's just get the basic game working.

Laying Out the Mine Tableau

We'll need to implement the same tableau layout for the mine cards in the digital version of Prospector as we did with the paper prototype you just played. To do this, we'll generate some XML code from the layout diagram in [Figure 31.7](#).

1. In Unity, open the LayoutXML.xml file in the Resources folder, and you'll see this layout information:

[Click here to view code image](#)

```
<xml>
  <!-- This file holds info for laying out the Prospector card game. -->

  <!-- The multiplier is multiplied by the x and y attributes below. -->
  <!-- This determines how loose or tight the layout is. -->
  <multiplier x="1.25" y="1.5" />

  <!-- In the XML below, id is the number of the card -->
  <!-- x and y set position -->
  <!-- faceup is 1 if the card is face-up -->
  <!-- layer sets the depth layer so cards overlap properly -->
  <!-- hiddenby is the ids of cards that keep a card face-down -->

  <!-- Layer0, the deepest cards. -->
  <slot id="0" x="-6" y="-5" faceup="0" layer="0" hiddenby="3,4" />
  <slot id="1" x="0" y="-5" faceup="0" layer="0" hiddenby="5,6" />
  <slot id="2" x="6" y="-5" faceup="0" layer="0" hiddenby="7,8" />

  <!-- Layer1, the next level. -->
  <slot id="3" x="-7" y="-4" faceup="0" layer="1" hiddenby="9,10" />
  <slot id="4" x="-5" y="-4" faceup="0" layer="1" hiddenby="10,11" />
  <slot id="5" x="-1" y="-4" faceup="0" layer="1" hiddenby="12,13" />
  <slot id="6" x="1" y="-4" faceup="0" layer="1" hiddenby="13,14" />
  <slot id="7" x="5" y="-4" faceup="0" layer="1" hiddenby="15,16" />
  <slot id="8" x="7" y="-4" faceup="0" layer="1" hiddenby="16,17" />

  <!-- Layer2, the next level. -->
  <slot id="9" x="-8" y="-3" faceup="0" layer="2" hiddenby="18,19" />
  <slot id="10" x="-6" y="-3" faceup="0" layer="2" hiddenby="19,20" />
  <slot id="11" x="-4" y="-3" faceup="0" layer="2" hiddenby="20,21" />
  <slot id="12" x="-2" y="-3" faceup="0" layer="2" hiddenby="21,22" />
  <slot id="13" x="0" y="-3" faceup="0" layer="2" hiddenby="22,23" />
  <slot id="14" x="2" y="-3" faceup="0" layer="2" hiddenby="23,24" />
  <slot id="15" x="4" y="-3" faceup="0" layer="2" hiddenby="24,25" />
  <slot id="16" x="6" y="-3" faceup="0" layer="2" hiddenby="25,26" />
  <slot id="17" x="8" y="-3" faceup="0" layer="2" hiddenby="26,27" />

  <!-- Layer3, the top level. -->
  <slot id="18" x="-9" y="-2" faceup="1" layer="3" />
  <slot id="19" x="-7" y="-2" faceup="1" layer="3" />
  <slot id="20" x="-5" y="-2" faceup="1" layer="3" />
  <slot id="21" x="-3" y="-2" faceup="1" layer="3" />
  <slot id="22" x="-1" y="-2" faceup="1" layer="3" />
  <slot id="23" x="1" y="-2" faceup="1" layer="3" />
  <slot id="24" x="3" y="-2" faceup="1" layer="3" />
```

```

<slot id="25" x="5" y="-2" faceup="1" layer="3" />
<slot id="26" x="7" y="-2" faceup="1" layer="3" />
<slot id="27" x="9" y="-2" faceup="1" layer="3" />

<!-- This positions the draw pile and staggers it -->
<slot type="drawpile" x="6" y="5" xstagger="0.15" layer="4"/>

<!-- This positions the discard pile and target card -->
<slot type="discardpile" x="0" y="1" layer="5"/>

</xml>

```

As you can see, this has layout information for each of the cards in the tableau (which is formed of `<slot>`s without a `type` attribute) as well as two special slots (that do have `type` attributes), the `drawpile` and `discardpile`.

2. Now, let's write some code to parse this LayoutXML into useful information. Create a new class named `Layout` and enter the following code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

// The SlotDef class is not a subclass of MonoBehaviour, so it doesn't need a
// separate C# file.
[System.Serializable] // This makes SlotDefs visible in the Unity Inspector pane
public class SlotDef {
    public float          x;
    public float          y;
    public bool           faceUp=false;
    public string          layerName="Default";
    public int            layerID = 0;
    public int            id;
    public List<int>      hiddenBy = new List<int>();
    public string          type="slot";
    public Vector2         stagger;
}

public class Layout : MonoBehaviour {
    public PT_XMLReader    xmlr; // Just like Deck, this has a PT_XMLReader
    public PT_XMLHashtable  xml; // This variable is for easier xml access
    public Vector2          multiplier; // Sets the spacing of the tableau
    // SlotDef references
    public List<SlotDef>    slotDefs; // All the SlotDefs for Row0-Row3
    public SlotDef          drawPile;
    public SlotDef          discardPile;
    // This holds all of the possible names for the layers set by layerID
    public string[]          sortingLayerNames = new
    string[] { "Row0", "Row1", "Row2", "Row3", "Discard", "Draw" };

    // This function is called to read in the LayoutXML.xml file
    public void ReadLayout(string xmlText) {
        xmlr = new PT_XMLReader();
        xmlr.Parse(xmlText); // The XML is parsed
        xml = xmlr.xml["xml"][0]; // And xml is set as a shortcut to the XML

        // Read in the multiplier, which sets card spacing
        multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
        multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

        // Read in the slots

```

```

SlotDef tSD;
// slotsX is used as a shortcut to all the <slot>s
PT_XMLHashList slotsX = xml["slot"];

for (int i=0; i<slotsX.Count; i++) {
    tSD = new SlotDef(); // Create a new SlotDef instance
    if (slotsX[i].HasAtt("type")) {
        // If this <slot> has a type attribute parse it
        tSD.type = slotsX[i].att("type");
    } else {
        // If not, set its type to "slot"; it's a tableau card
        tSD.type = "slot";
    }
    // Various attributes are parsed into numerical values
    tSD.x = float.Parse( slotsX[i].att("x") );
    tSD.y = float.Parse( slotsX[i].att("y") );
    tSD.layerID = int.Parse( slotsX[i].att("layer") );
    // This converts the number of the layerID into a text layerName
    tSD.layerName = sortingLayerNames[ tSD.layerID ];
    // The layers are used to make sure that the correct cards are
    // on top of the others. In Unity 2D, all of our assets are
    // effectively at the same Z depth, so the layer is used
    // to differentiate between them.

    switch (tSD.type) {
        // pull additional attributes based on the type of this <slot>
        case "slot":
            tSD.faceUp = (slotsX[i].att("faceup") == "1");
            tSD.id = int.Parse( slotsX[i].att("id") );
            if (slotsX[i].HasAtt("hiddenby")) {
                string[] hiding = slotsX[i].att("hiddenby").Split(',');
                foreach( string s in hiding ) {
                    tSD.hiddenBy.Add ( int.Parse(s) );
                }
            }
            slotDefs.Add(tSD);
            break;

        case "drawpile":
            tSD.stagger.x = float.Parse( slotsX[i].att("xstagger") );
            drawPile = tSD;
            break;
        case "discardpile":
            discardPile = tSD;
            break;
    }
}
}

```

At this point, most of the preceding syntax should look familiar to you. The `SlotDef` class is created to store information read in from the XML `<slot>`s in a more accessible way. Then, the `Layout` class is defined, and the `ReadLayout()` method is created, which will take an XML-formatted string as input and turn it into a series of `SlotDefs`.

3. Open the Prospector class and add the following bolded lines:

```
k here to view code image  
public class Prospector : MonoBehaviour {  
    static public Prospector S;  
  
    public Deck deck;
```

```

public TextAsset          deckXML;

public Layout           layout;
public TextAsset         layoutXML;

void Awake() {
    S = this; // Set up a Singleton for Prospector
}

void Start () {
    deck = GetComponent<Deck>();           // Get the Deck
    deck.InitDeck(deckXML.text);            // Pass DeckXML to it
    Deck.Shuffle(ref deck.cards);          // This shuffles the deck
    // The ref keyword passes a reference to deck.cards, which allows
    // deck.cards to be modified by Deck.Shuffle()

    layout = GetComponent<Layout>(); // Get the Layout
    layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it
}
}
}

```

4. Once this is done, you will need to set up a couple of things in Unity. Switch to Unity and select `_MainCamera` in the Hierarchy. From the menu bar, choose *Component > Scripts > Layout* to attach a Layout script to `_MainCamera` (this is just another different way to attach a script to a `GameObject`). You should now be able to scroll down in the Inspector pane and see the *Layout (Script)* component at the bottom.
5. Find the *Prospector (Script)* component of `_MainCamera`. You'll see that the public fields `layout` and `layoutXML` have appeared there. Click the target next to `layoutXML` and choose *LayoutXML* from the Assets tab. (You may need to click the Assets button at the top of the window that appeared.)
6. *Save your scene.*
7. Now, press Play. If you select `_MainCamera` in the Hierarchy and scroll down to the *Layout (Script)* component, you should be able to open the disclosure triangle next to `slots` and see that all of the `<slot>`s have been parsed from the XML.

Working with `CardProspector`, a Subclass of `Card`

Before we can position the cards in the tableau, it will be necessary to add some features to the `Card` class that are specific to the Prospector game. Because `Card` and `Deck` are intended to be reused on other card games, we will choose to create a `CardProspector` class as a subclass of `Card` rather than modifying `Card` directly. Create a new C# script named *CardProspector* and enter this code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

// This is an enum, which defines a type of variable that only has a few
// possible named values. The CardState variable type has one of four values:
// drawpile, tableau, target, & discard
public enum CardState {
    drawpile,
    tableau,
    target,
    discard
}

```

```

public class CardProspector : Card { // Make sure CardProspector extends Card
    // This is how you use the enum CardState
    public CardState state = CardState.drawpile;
    // The hiddenBy list stores which other cards will keep this one face down
    public List<CardProspector> hiddenBy = new List<CardProspector>();
    // LayoutID matches this card to a Layout XML id if it's a tableau card
    public int layoutID;
    // The SlotDef class stores information pulled in from the LayoutXML <slot>
    public SlotDef slotDef;
}

```

These extensions to Card will handle things like the four types of places that the card can be in the tableau (drawpile, tableau [one of the initial 28 cards in the mine], discard, or target [the active card on top of the discard pile]), the storage of layout information (slotDef), and the information that determines when a card should be face-up or face-down (hiddenBy and layoutID).

Now that this subclass is available, it'll be necessary to convert the cards in the deck from Cards to CardProspects. This will be done by adding the following code to the Prospector class:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...
    public List<CardProspector> drawPile;

    void Start () {
        deck = GetComponent<Deck>(); // Get the Deck
        deck.InitDeck(deckXML.text); // Pass DeckXML to it

        layout = GetComponent<Layout>(); // Get the Layout
        layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it

        drawPile = ConvertListCardsToListCardProspects( deck.cards );
    }

    List<CardProspector> ConvertListCardsToListCardProspects(List<Card> lCD) {
        List<CardProspector> lCP = new List<CardProspector>();
        CardProspector tCP;
        foreach( Card tCD in lCD ) {
            tCP = tCD as CardProspector; // 1
            lCP.Add( tCP );
        }
        return( lCP );
    }
}

```

Once this code is in, try running it and then look at the drawPile in the Inspector pane. You'll notice that all the cards in the drawPile are null. (You can also see this happen by placing a break point on the line marked // 1 in the preceding code.) When we try to treat the Card tCD as a CardProspector, the as returns null instead of a converted Card. This is because of how object-oriented coding works in C# (see the [“On Superclasses and Subclasses” sidebar](#)).

On Superclasses and Subclasses

You're familiar, of course, with superclasses and subclasses from [Chapter 25, “Classes.”](#) However, you might wonder why attempting to cast a superclass to a subclass doesn't work.

In Prospector, Card is the superclass, and the subclass is CardProspector. You could just as easily think of this as a superclass Animal and a subclass Scorpion. All Scorpions are Animals, but not all Animals are Scorpions. You can always refer to a Scorpion as “that Animal”, but you can’t refer to any Animal as a Scorpion. Along the same lines, a Scorpion might have a Sting () function, but a Cow would not. This is why it’s not possible to treat any Animal as a Scorpion, because trying to call Sting () on any other Animal might cause an error.

In Prospector, we want to use a bunch of cards that are created by the Deck script as if they were CardProspectors. This is akin to having a bunch of Animals that we want to treat like Scorpions (but we’ve already decided this is impossible). However, it’s always possible to refer to a Scorpion as an Animal, so the solution that we use in Prospector is to make PrefabCard have a *CardProspector (Script)* component instead of just a *Card (Script)* component. If we just create Scorpions from the beginning, and then treat them as Animals through several functions (which we can do because Scorpion is a subclass of Animal), when we choose to call Scorpion s = Animal as Scorpion; later, that will work perfectly because the Animal was always secretly a Scorpion.

The solution in this case is to make sure that the CardProspector was always a CardProspector and was just masquerading as a Card for all of the code in the Deck class. To do this, select PrefabCard in the Project pane, and you’ll see that it appears in the Inspector with a *Card (Script)* component. If you click the target next to the *Script* variable (that is currently set to Card), you can choose a different script for this component. Select CardProspector, and the PrefabCard will now have a *CardProspector (Script)* component instead of just a Card. If you select _MainCamera from the Hierarchy and play the scene now, you will see that all of the entries in drawPile are now full of CardProspectors instead of null.

When the Deck script instantiates PrefabCard and gets the Card component of it, this will still work perfectly fine because a CardPrefab can always be referred to as a Card. Then, when the ConvertListCardsToListCardProspectors () function attempts to call tCP = tCD as CardProspector;, it works just fine.

Save your scene.

Positioning Cards in the Tableau

Now that everything is ready, it’s time to add some code to the Prospector class to actually lay out the game:

[Click here to view code image](#)

```
public class Prospector : MonoBehaviour {
    ...
    public Layout           layout;
    public TextAsset        layoutXML;
    public Vector3          layoutCenter;
    public float             xOffset = 3;
    public float             yOffset = -2.5f;
    public Transform         layoutAnchor;

    public CardProspector   target;
    public List<CardProspector> tableau;
    public List<CardProspector> discardPile;
```

```

public List<CardProspector> drawPile;

void Start () {
    ...
    drawPile = ConvertListCardsToListCardProspectors( deck.cards );
    LayoutGame();
}

// The Draw function will pull a single card from the drawPile and return it
CardProspector Draw() {
    CardProspector cd = drawPile[0]; // Pull the 0th CardProspector
    drawPile.RemoveAt(0);           // Then remove it from List<CardProspector> drawPile
    return(cd);                   // And return it
}

// LayoutGame() positions the initial tableau of cards, a.k.a. the "mine"
void LayoutGame() {
    // Create an empty GameObject to serve as an anchor for the tableau //1
    if (layoutAnchor == null) {
        GameObject tGO = new GameObject("_LayoutAnchor");
        // ^ Create an empty GameObject named _LayoutAnchor in the Hierarchy
        layoutAnchor = tGO.transform; // Grab its Transform
        layoutAnchor.transform.position = layoutCenter; // Position it
    }

    CardProspector cp;
    // Follow the layout
    foreach (SlotDef tSD in layout.slotDefs) {
        // ^ Iterate through all the SlotDefs in the layout.slotDefs as tSD
        cp = Draw(); // Pull a card from the top (beginning) of the drawPile
        cp.faceUp = tSD.faceUp; // Set its faceUp to the value in SlotDef
        cp.transform.parent = layoutAnchor; // Make its parent layoutAnchor
        // This replaces the previous parent: deck.deckAnchor, which appears
        // as _Deck in the Hierarchy when the scene is playing.
        cp.transform.localPosition = new Vector3(
            layout.multiplier.x * tSD.x,
            layout.multiplier.y * tSD.y,
            -tSD.layerID );
        // ^ Set the localPosition of the card based on slotDef
        cp.layoutID = tSD.id;
        cp.slotDef = tSD;
        cp.state = CardState.tableau;
        // CardProspectors in the tableau have the state CardState.tableau

        tableau.Add(cp); // Add this CardProspector to the List<CardProspector> tableau
    }
}
}

```

When you play this, you will see that the cards are indeed laid out in the mine tableau layout described in LayoutXML.xml, and the right ones are face-up and face-down, but there are some serious issues with sorting layers (see [Figure 31.10](#)).

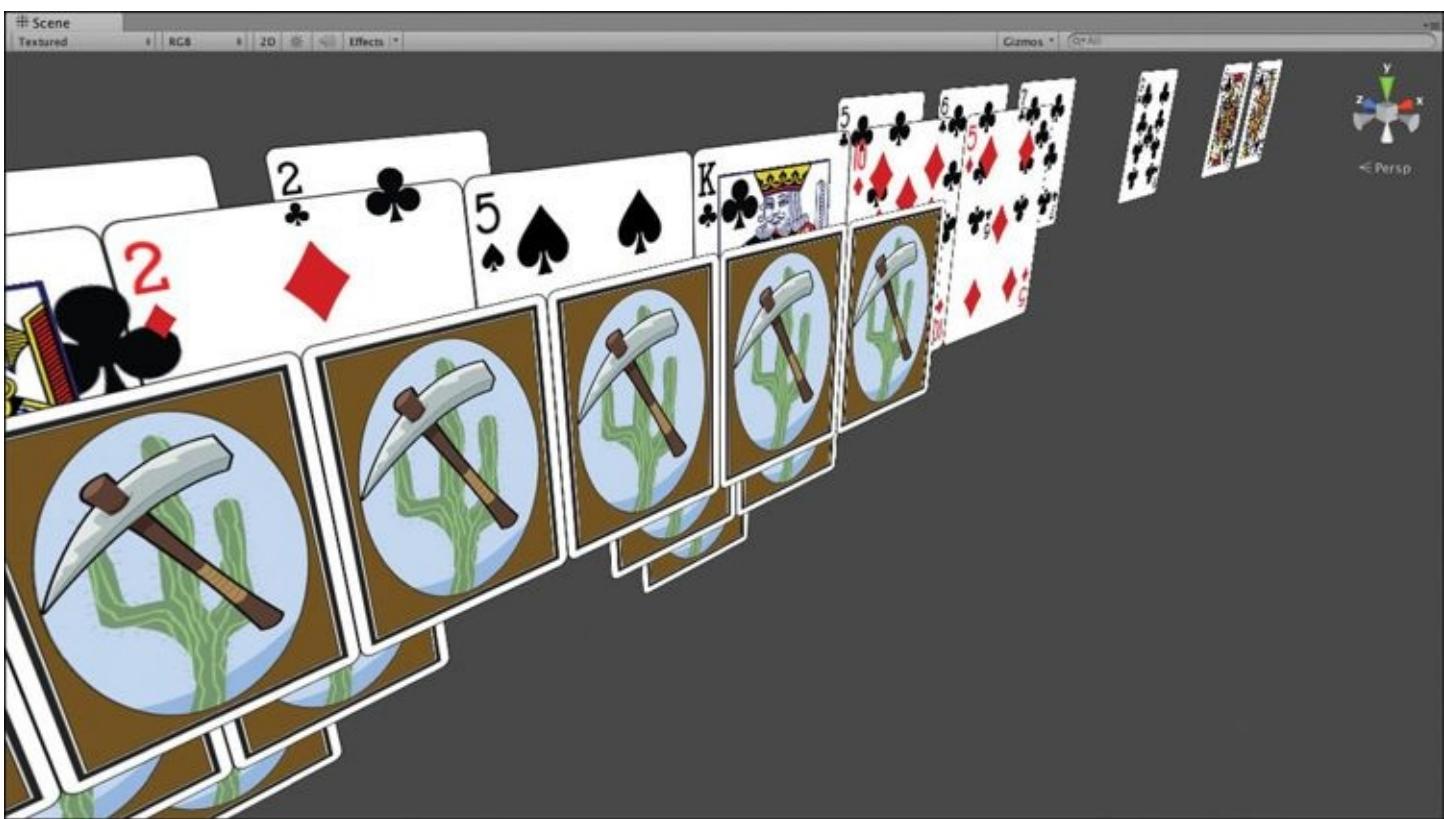


Figure 31.10 Cards are laid out, but there are several sorting layer issues (and remaining cards from the initial grid layout that existed previously)

Hold the Option/Alt key down and use the left mouse button in the Scene window to look around, and you will see that when using Unity's 2D tools, the distance of the 2D object to the camera has nothing to do with the depth sorting of the objects (that is, which objects are rendered on top of each other). We actually got a little lucky with the construction of the cards because we built them from back to front so that all the pips and decorators showed up on top of the card face. However, here we're actually going to have to be more careful about it for the layout of the game to avoid the problems you can see in [Figure 31.10](#).

Unity 2D has two methods of dealing with depth sorting:

- **Sorting Layers:** Sorting layers are used to group 2D objects. Everything in a lower sorting layer is rendered behind everything in a higher sorting layer. Each `SpriteRenderer` component has a `sortingLayerName` string variable that can be set to the name of a sorting layer.
- **Sorting Order:** Each `SpriteRenderer` component also has a `sortingOrder` variable that can be set. This is used to position elements within each sorting layer relative to each other.

In the absence of sorting layers and `sortingOrder`, sprites are often rendered from back to front in the order that they were created, but this is not at all reliable.

Setting Up Sorting Layers

From the menu bar, choose `Edit > Project Settings > Tags and Layers`. You've used tags and layers for physics layers and tags before, but we haven't yet touched sorting layers. Open the disclosure triangle next to *Sorting Layers*, and enter the layers as shown in [Figure 31.11](#). You will need to click the `+` button at the bottom-right of the list to add new sorting layers.

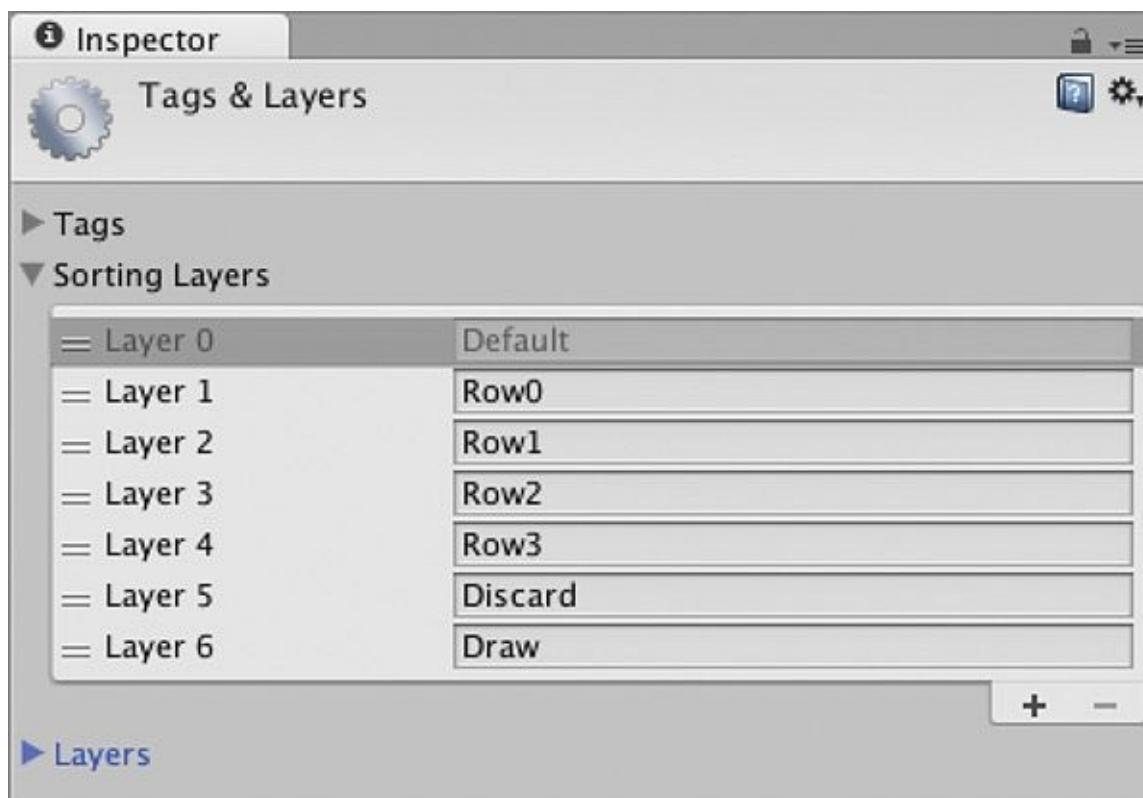


Figure 31.11 The sorting layers required for Prospector

Because SpriteRenderers and depth sorting are something that will be necessary for any card game built using our code base, the code to deal with depth sorting should be added to the Card class. Open the Card script and add the following code:

[Click here to view code image](#)

```

public class Card : MonoBehaviour {
    ...
    public CardDefinition def; // Parsed from DeckXML.xml

    // List of the SpriteRenderer Components of this GameObject and its children
    public SpriteRenderer[] spriteRenderers;

    void Start() {
        SetSortOrder(0); // Ensures that the card starts properly depth sorted
    }

    public bool faceUp {
        ...
    }

    // If spriteRenderers is not yet defined, this function defines it
    public void PopulateSpriteRenderers() {
        // If spriteRenderers is null or empty
        if (spriteRenderers == null || spriteRenderers.Length == 0) {
            // Get SpriteRenderer Components of this GameObject and its children
            spriteRenderers = GetComponentsInChildren<SpriteRenderer>();
        }
    }

    // Sets the sortingLayerName on all SpriteRenderer Components
    public void SetSortingLayerName(string tSLN) {
        PopulateSpriteRenderers();

        foreach (SpriteRenderer tSR in spriteRenderers) {
            tSR.sortingLayerName = tSLN;
        }
    }
}

```

```

        }

    // Sets the sortingOrder of all SpriteRenderer Components
    public void SetSortOrder(int s0rd) {
        PopulateSpriteRenderers();

        // The white background of the card is on bottom (s0rd)
        // On top of that are all the pips, decorators, face, etc. (s0rd+1)
        // The back is on top so that when visible, it covers the rest (s0rd+2)

        // Iterate through all the spriteRenderers as tSR
        foreach (SpriteRenderer tSR in spriteRenderers) {
            if (tSR.gameObject == this.gameObject) {
                // If the gameObject is this.gameObject, it's the background
                tSR.sortingOrder = s0rd; // Set its order to s0rd
                continue; // And continue to the next iteration of the loop

            }
            // Each of the children of this GameObject are named
            // switch based on the names
            switch (tSR.gameObject.name) {
                case "back": // if the name is "back"
                    tSR.sortingOrder = s0rd+2;
                    // ^ Set it to the highest layer to cover everything else
                    break;
                case "face": // if the name is "face"
                default: // or if it's anything else
                    tSR.sortingOrder = s0rd+1;
                    // ^ Set it to the middle layer to be above the background
                    break;
            }
        }
    }
}

```

Now, Prospector needs one line added to make sure that the cards in the initial mine layout are placed in the proper sorting layer. Add this line near the end of `Prospector.LayoutGame()`:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...
    // LayoutGame() positions the initial tableau of cards, the "mine"
    void LayoutGame() {
        ...
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
            cp.state = CardState.tableau;
            // CardProspectors in the tableau have the state CardState.tableau

            cp.SetSortingLayerName(tSD.layerName); // Set the sorting layers

            tableau.Add(cp); // Add this CardProspector to the List<> tableau
        }
    }
}

```

Now, when you run the scene, you'll see that the cards stack properly on top of each other in the mine.

Implementing Draw, Discard, and Game Logic

Before we move cards into place for the draw pile, let's start by delineating the possible actions that can happen in the game:

1. If the target card is replaced by any other card, the replaced target card then moves to the discard pile.
2. A card can move from the drawPile to become the target card.
3. A card in the mine tableau that is one higher or one lower in rank than the target card can move to become the target card.
4. If a face-down card has no more cards hiding it, it becomes face-up.
5. The game is over when either the mine is empty (win) or the draw pile is empty and there are no more possible plays (loss).

Actions number 2 and 3 here are the possible move actions, where a card is physically moved, and numbers 1, 4, and 5 are passive actions that happen as a result of either number 2 or 3.

Making Cards Clickable

Because all of these actions are instigated by a click on one of the cards, we first need to make the cards clickable. This is something that will be needed for every card game, so add the following method to the Card class:

[Click here to view code image](#)

```
public class Card : MonoBehaviour {  
    ...  
  
    // Virtual methods can be overridden by subclass methods with the same name  
    virtual public void OnMouseUpAsButton() {  
        print (name); // When clicked, this outputs the card name  
    }  
}
```

Now, when you press Play, you can click any card in the scene, and it will output its name. However, in Prospector, we need card clicks to do more than that, so add the following method to the end of the CardProspector class:

[Click here to view code image](#)

```
public class CardProspector : Card {  
    ...  
  
    // This allows the card to react to being clicked  
    override public void OnMouseUpAsButton() {  
        // Call the CardClicked method on the Prospector singleton  
        Prospector.S.CardClicked(this);  
        // Also call the base class (Card.cs) version of this method  
        base.OnMouseUpAsButton();  
    }  
}
```

Now, the CardClicked method must be written in the Prospector script. For now, let's just tackle moving a card from the drawPile to the target (number 2 from the action list):

[Click here to view code image](#)

```
public class Prospector : MonoBehaviour {  
    ...
```

```

// CardClicked is called any time a card in the game is clicked
public void CardClicked(CardProspector cd) {
    // The reaction is determined by the state of the clicked card
    switch (cd.state) {
        case CardState.target:
            // Clicking the target card does nothing
            break;
        case CardState.drawpile:
            // Clicking any card in the drawPile will draw the next card
            MoveToDiscard(target); // Moves the target to the discardPile
            MoveToTarget(Draw()); // Moves the next drawn card to the target
            UpdateDrawPile(); // Restacks the drawPile
            break;
        case CardState.tableau:
            // Clicking a card in the tableau will check if it's a valid play
            break;
    }
}
}

```

Of course, you also will need to add the `MoveToDiscard()`, `MoveToTarget()`, and `UpdateDrawPile()` methods to the `Prospector` class, as well as bits to the end of `Prospector.LayoutGame()` to draw the initial target card and arrange the `drawPile`:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...

    // LayoutGame() positions the initial tableau of cards, the "mine"
    void LayoutGame() {
        ...
        foreach (SlotDef tSD in layout.slotDefs) {
            ...

            // Set up the initial target card
            MoveToTarget(Draw ());

            // Set up the Draw pile
            UpdateDrawPile();
        }
    }

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        ...
    }

    // Moves the current target to the discardPile
    void MoveToDiscard(CardProspector cd) {
        // Set the state of the card to discard
        cd.state = CardState.discard;
        discardPile.Add(cd); // Add it to the discardPile List<>
        cd.transform.parent = layoutAnchor; // Update its transform parent
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID+0.5f );
        // ^ Position it on the discardPile
        cd.faceUp = true;
        // Place it on top of the pile for depth sorting
        cd.SetSortingLayerName(layout.discardPile.layerName);
    }
}

```

```

        cd.SetSortOrder(-100+discardPile.Count);
    }

    // Make cd the new target card
    void MoveToTarget(CardProspector cd) {
        // If there is currently a target card, move it to discardPile
        if (target != null) MoveToDiscard(target);
        target = cd; // cd is the new target
        cd.state = CardState.target;
        cd.transform.parent = layoutAnchor;
        // Move to the target position
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID );
        cd.faceUp = true; // Make it face-up
        // Set the depth sorting
        cd.SetSortingLayerName(layout.discardPile.layerName);
        cd.SetSortOrder(0);
    }

    // Arranges all the cards of the drawPile to show how many are left
    void UpdateDrawPile() {
        CardProspector cd;
        // Go through all the cards of the drawPile
        for (int i=0; i<drawPile.Count; i++) {
            cd = drawPile[i];
            cd.transform.parent = layoutAnchor;
            // Position it correctly with the layout.drawPile.stagger
            Vector2 dpStagger = layout.drawPile.stagger;
            cd.transform.localPosition = new Vector3(
                layout.multiplier.x * (layout.drawPile.x + i*dpStagger.x),
                layout.multiplier.y * (layout.drawPile.y + i*dpStagger.y),
                -layout.drawPile.layerID+0.1f*i );
            cd.faceUp = false; // Make them all face-down
            cd.state = CardState.drawpile;
            // Set depth sorting
            cd.SetSortingLayerName(layout.drawPile.layerName);
            cd.SetSortOrder(-10*i);
        }
    }
}

```

Now, when you play the scene, you will see that you can click on the drawPile to draw a new target card. We're getting close to having a game now!

Matching Cards from the Mine

To make the card in the mine work, we need to have a little code that checks to make sure that the clicked card is either one higher or one lower than the target card (and, of course, also handles A-to-King wraparound). Add these bolded lines to the Prospector script:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
            ...

```

```

        case CardState.tableau:
            // Clicking a card in the tableau will check if it's a valid play
            bool validMatch = true;
            if (!cd.faceUp) {
                // If the card is face-down, it's not valid
                validMatch = false;
            }
            if (!AdjacentRank(cd, target)) {
                // If it's not an adjacent rank, it's not valid
                validMatch = false;
            }
            if (!validMatch) return; // return if not valid
            // Yay! It's a valid card.
            tableau.Remove(cd); // Remove it from the tableau List
            MoveToTarget(cd); // Make it the target card
            break;
        }
    }
}

...
// Return true if the two cards are adjacent in rank (A & K wrap around)
public bool AdjacentRank(CardProspector c0, CardProspector c1) {
    // If either card is face-down, it's not adjacent.
    if (!c0.faceUp || !c1.faceUp) return(false);

    // If they are 1 apart, they are adjacent
    if (Mathf.Abs(c0.rank - c1.rank) == 1) {
        return(true);
    }
    // If one is A and the other King, they're adjacent
    if (c0.rank == 1 && c1.rank == 13) return(true);
    if (c0.rank == 13 && c1.rank == 1) return(true);

    // Otherwise, return false
    return(false);
}
}

```

Now, you can play the game and actually play the top row correctly. However, as you play more, you'll notice that the face-down cards are never flipping to face-up. This is what the `List<CardProspector> CardProspector.hiddenBy` field is for. We have the information about which cards hide others in `List<int> SlotDef.hiddenBy`, but we need to be able to convert from the integer IDs in `SlotDef.hiddenBy` to the actual `CardProspectors` that have that ID. Add this code to `Prospector` to do so:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...

    CardProspector Draw() {
        ...
    }

    // Convert from the layoutID int to the CardProspector with that ID
    CardProspector FindCardByLayoutID(int layoutID) {
        foreach (CardProspector tCP in tableau) {
            // Search through all cards in the tableau List<CardProspector>
            if (tCP.layoutID == layoutID) {
                // If the card has the same ID, return it
                return( tCP );
            }
        }
    }
}

```

```

        }

        // If it's not found, return null
        return( null );
    }

    // LayoutGame() positions the initial tableau of cards, the "mine"
    void LayoutGame() {
        ...
        // Follow the layout
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
        }

        // Set which cards are hiding others
        foreach (CardProspector tCP in tableau) {
            foreach( int hid in tCP.slotDef.hiddenBy ) {
                cp = FindCardByLayoutID(hid);
                tCP.hiddenBy.Add(cp);
            }
        }

        // Set up the target card
        MoveToTarget(Draw ());
        ...
    }

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
            ...
            case CardState.tableau:
                ...
                MoveToTarget(cd);    // Make it the target card
                SetTableauFaces();  // Update tableau card face-ups
                break;
        }
    }

    ...

    // This turns cards in the Mine face-up or face-down
    void SetTableauFaces() {
        foreach( CardProspector cd in tableau ) {
            bool fup = true; // Assume the card will be face-up
            foreach( CardProspector cover in cd.hiddenBy ) {
                // If either of the covering cards are in the tableau
                if (cover.state == CardState.tableau) {
                    fup = false; // then this card is face-down
                }
            }
            cd.faceUp = fup; // Set the value on the card
        }
    }
}

```

Now, an entire round of the game is playable! Next up is making the game know when it's over. This only needs to be checked once after each time the player has clicked a card, so the check will be called from the end of `Prospector.CardClicked()`. Add the following to the `Prospector` class:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
            ...
        }
        // Check to see whether the game is over or not
        CheckForGameOver();
    }

    ...

    // Test whether the game is over
    void CheckForGameOver() {
        // If the tableau is empty, the game is over
        if (tableau.Count==0) {
            // Call GameOver() with a win
            GameOver(true);
            return;
        }
        // If there are still cards in the draw pile, the game's not over
        if (drawPile.Count>0) {
            return;
        }
        // Check for remaining valid plays
        foreach ( CardProspector cd in tableau ) {
            if (AdjacentRank(cd, target)) {
                // If there is a valid play, the game's not over
                return;
            }
        }
        // Since there are no valid plays, the game is over
        // Call GameOver with a loss
        GameOver (false);
    }

    // Called when the game is over. Simple for now, but expandable
    void GameOver(bool won) {
        if (won) {
            print ("Game Over. You won! :)");
        } else {
            print ("Game Over. You Lost. :( ");
        }
        // Reload the scene, resetting the game
        Application.LoadLevel("__Prospector_Scene_0");
    }
}

```

Now the game is playable and repeatable, and it knows when it has won or lost. Next up, it's time to add some scoring.

Adding Scoring to Prospector

The original card game of *Prospector* (or *Tri-Peaks*, on which it was based) had no scoring mechanism beyond the player winning or losing. But as a digital game, it's really helpful to have scores and a high score so that players have a reason to keep playing (to beat their high score).

Ways to Earn Points in the Game

We will implement several ways to earn points in Prospector:

1. Moving a card from the mine to the target card earns 1 point.
2. Every subsequent card removed from the mine without drawing from the draw pile increases the points awarded per card by 1, so a *run* of five cards removed without a draw would be worth 1, 2, 3, 4, and 5 points, respectively, for a total of 15 points for the run ($1 + 2 + 3 + 4 + 5 = 15$).
3. If the player wins the round, she carries her score on to the next round. Whenever a round is lost, her score for all rounds is totaled and checked against the high score list.
4. The number of points earned for a run will double for each special gold card in the run. If two of the cards in the example run from #2 were gold, then the run would be worth 60 points ($15 \times 2 \times 2 = 60$).

The scoring will be handled by the `Prospector` class because it is aware of all the conditions that could contribute to the score. We will also create a script named `Scoreboard` to handle all the visual elements of showing the score to the player.

We'll implement numbers 1 through 3 in this chapter, and I'll leave number 4 for you to implement on your own later.

Making the Run Scoring Work

For now, let's just make some changes to `Prospector` to track the score. Because we're enabling runs and eventually will add run score doubling via gold cards, it makes sense to store the score for the run separately and then apply that to the total score for the round once the run has been ended (by drawing a card from the `drawPile`). Add the following code to `Prospector` to implement this:

[Click here to view code image](#)

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

// An enum to handle all the possible scoring events
public enum ScoreEvent {
    draw,
    mine,
    mineGold,
    gameWin,
    gameLoss
}

public class Prospector : MonoBehaviour {
    static public Prospector S;
    static public int SCORE_FROM_PREV_ROUND = 0;
    static public int HIGH_SCORE = 0;

    ...

    public List<CardProspector> drawPile;

    // Fields to track score info
    public int chain = 0; // of cards in this run
    public int scoreRun = 0;
    public int score = 0;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }
}
```

```

// Check for a high score in PlayerPrefs
if (PlayerPrefs.HasKey ("ProspectorHighScore")) {
    HIGH_SCORE = PlayerPrefs.GetInt("ProspectorHighScore");
}
// Add the score from last round, which will be >0 if it was a win
score += SCORE_FROM_PREV_ROUND;
// And reset the SCORE_FROM_PREV_ROUND
SCORE_FROM_PREV_ROUND = 0;
}

...
// CardClicked is called any time a card in the game is clicked
public void CardClicked(CardProspector cd) {
    // The reaction is determined by the state of the clicked card
    switch (cd.state) {
        ...
        case CardState.drawpile:
            ...
            ScoreManager(ScoreEvent.draw);
            break;
        case CardState.tableau:
            ...
            ScoreManager(ScoreEvent.mine);
            break;
    }
    ...
}

...
// Called when the game is over. Simple for now, but expandable
void GameOver(bool won) {
    if (won) {
        ScoreManager(ScoreEvent.gameWin);           // This replaces the old line
    } else {
        ScoreManager(ScoreEvent.gameLoss);           // This replaces the old line
    }
    // Reload the scene, resetting the game
    Application.LoadLevel("__Prospector_Scene_0");
}

// ScoreManager handles all of the scoring
void ScoreManager(ScoreEvent sEvt) {
    switch (sEvt) {
        // Same things need to happen whether it's a draw, a win, or a loss
        case ScoreEvent.draw: // Drawing a card
        case ScoreEvent.gameWin: // Won the round
        case ScoreEvent.gameLoss: // Lost the round
            chain = 0;           // resets the score chain
            score += scoreRun; // add scoreRun to total score
            scoreRun = 0;        // reset scoreRun
            break;
        case ScoreEvent.mine: // Remove a mine card
            chain++;           // increase the score chain
            scoreRun += chain; // add score for this card to run
            break;
    }

    // This second switch statement handles round wins and losses
    switch (sEvt) {
        case ScoreEvent.gameWin:
            // If it's a win, add the score to the next round
    }
}

```

```
// static fields are NOT reset by Application.LoadLevel()
Prospector.SCORE_FROM_PREV_ROUND = score;
print ("You won this round! Round score: "+score);
break;
case ScoreEvent.gameLoss:
    // If it's a loss, check against the high score
    if (Prospector.HIGH_SCORE <= score) {
        print("You got the high score! High score: "+score);
        Prospector.HIGH_SCORE = score;
        PlayerPrefs.SetInt("ProspectorHighScore", score);
    } else {
        print ("Your final score for the game was: "+score);
    }
    break;
default:
    print ("score: "+score+"  scoreRun:"+scoreRun+"  chain:"+chain);
    break;
}
```

Now, as you play the game, you'll see little notes in the Console pane that tell you your score. This works fine for testing, but let's make things look a little better for our players.

Showing the Score to the Players

For this game, we'll make a couple of reusable components that can show the score. One will be a Scoreboard class that will manage all of the score display. The other will be FloatingScore, which will be an on-screen number that can move around the screen on its own. We'll also make use of Unity's `SendMessage()` feature, which can call a method by name with one parameter on any GameObject:

1. Create a new C# script named *FloatingScore* and enter the following code:

Click here to view code image

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

// An enum to track the possible states of a FloatingScore
public enum FSState {
    idle,
    pre,
    active,
    post
}

// FloatingScore can move itself on screen following a Bézier curve
public class FloatingScore : MonoBehaviour {
    public FSState state = FSState.idle;
    [SerializeField]
    private int _score = 0; // The score field
    public string scoreString;

    // The score property also sets scoreString when set
    public int score {
        get {
            return(_score);
        }
        set {
            score = value;
        }
    }
}
```

```

        scoreString = Utils.AddCommasToNumber(_score);
        GetComponent<GUIText>().text = scoreString;
    }

    public List<Vector3> bezierPts; // Bezier points for movement
    public List<float> fontSizes; // Bezier points for font scaling
    public float timeStart = -1f;
    public float timeDuration = 1f;
    public string easingCurve = Easing.InOut; // Uses Easing in Utils.cs

    // The GameObject that will receive the SendMessage when this is done moving
    public GameObject reportFinishTo = null;

    // Set up the FloatingScore and movement
    // Note the use of parameter defaults for eTimeS & eTimeD
    public void Init(List<Vector3> ePts, float eTimeS = 0, float eTimeD = 1) {
        bezierPts = new List<Vector3>(ePts);

        if (ePts.Count == 1) { // If there's only one point
            // ...then just go there.
            transform.position = ePts[0];
            return;
        }

        // If eTimeS is the default, just start at the current time
        if (eTimeS == 0) eTimeS = Time.time;
        timeStart = eTimeS;
        timeDuration = eTimeD;

        state = FSState.pre; // Set it to the pre state, ready to start moving
    }

    public void FSCallback(FloatingScore fs) {
        // When this callback is called by SendMessage,
        // add the score from the calling FloatingScore
        score += fs.score;
    }

    // Update is called once per frame
    void Update () {
        // If this is not moving, just return
        if (state == FSState.idle) return;

        // Get u from the current time and duration
        // u ranges from 0 to 1 (usually)
        float u = (Time.time - timeStart)/timeDuration;
        // Use Easing class from Utils to curve the u value
        float uC = Easing.Ease (u, easingCurve);
        if (u<0) { // If u<0, then we shouldn't move yet.
            state = FSState.pre;
            // Move to the initial point
            transform.position = bezierPts[0];
        } else {
            if (u>=1) { // If u>=1, we're done moving
                uC = 1; // Set uC=1 so we don't overshoot
                state = FSState.post;
                if (reportFinishTo != null) { //If there's a callback GameObject
                    // Use SendMessage to call the FSCallback method
                    // with this as the parameter.
                    reportFinishTo.SendMessage("FSCallback", this);
                    // Now that the message has been sent,
                    // Destroy this gameObject
                    Destroy (gameObject);
                }
            }
        }
    }
}

```

```
        } else { // If there is nothing to callback
            // ...then don't destroy this. Just let it stay still.
            state = FSState.idle;
        }
    } else {
        // 0<=u<1, which means that this is active and moving
        state = FSState.active;
    }
    // Use Bezier curve to move this to the right point
    Vector3 pos = Utils.Bezier(uC, bezierPts);
    transform.position = pos;
    if (fontSizes != null && fontSizes.Count>0) {
        // If fontSizes has values in it
        // ...then adjust the fontSize of this GUIText
        int size = Mathf.RoundToInt( Utils.Bezier(uC, fontSizes) );
        GetComponent<GUIText>().fontSize = size;
    }
}
}
```

2. Create a new C# script named *Scoreboard* and enter this code into it:

[Click here to view code image](#)

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

// The Scoreboard class manages showing the score to the player
public class Scoreboard : MonoBehaviour {
    public static Scoreboard S; // The singleton for Scoreboard

    public GameObject    prefabFloatingScore;

    public bool _____;
    [SerializeField]
    private int          _score = 0;
    public string         _scoreString;

    // The score property also sets the scoreString
    public int score {
        get {
            return(_score);
        }
        set {
            _score = value;
            scoreString = Utils.AddCommasToNumber(_score);
        }
    }

    // The scoreString property also sets the GUIText.text
    public string scoreString {
        get {
            return(_scoreString);
        }
        set {
            _scoreString = value;
            GetComponent<GUIText>().text = _scoreString;
        }
    }

    void Awake() {
        S = this;
    }
}
```

```

}

// When called by SendMessage, this adds the fs.score to this.score
public void FSCallback(FloatingScore fs) {
    score += fs.score;
}

// This will Instantiate a new FloatingScore GameObject and initialize it.
// It also returns a pointer to the FloatingScore created so that the
// calling function can do more with it (like set fontSizes, etc.)
public FloatingScore CreateFloatingScore(int amt, List<Vector3> pts) {
    GameObject go = Instantiate(prefabFloatingScore) as GameObject;
    FloatingScore fs = go.GetComponent<FloatingScore>();
    fs.score = amt;
    fs.reportFinishTo = this.gameObject; // Set fs to call back to this
    fs.Init(pts);
    return(fs);
}
}

```

}

3. Now, you need to make the GameObjects for both the Scoreboard and the FloatingScore. In Unity, from the menu bar, choose *GameObject* > *Create Other* > *GUIText*. Rename the *GUIText* to *PrefabFloatingScore* and give it the settings shown in [Figure 31.12](#).

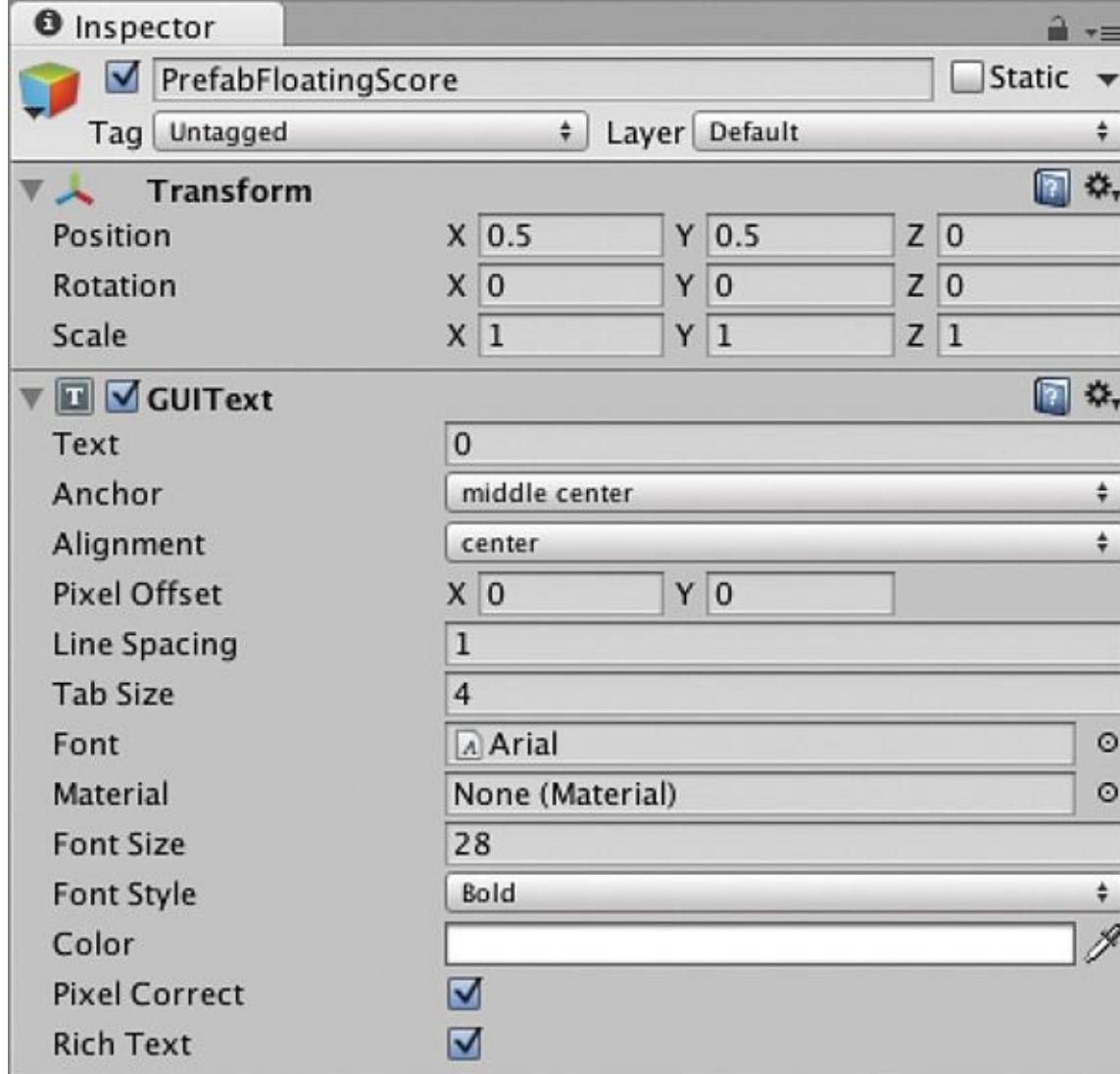


Figure 31.12 The settings for PrefabFloatingScore

4. Now attach the script *FloatingScore* to the *GameObject* *PrefabFloatingScore* (by dragging the

script onto FloatingScore in the Hierarchy). Then convert PrefabFloatingScore to a prefab by dragging it from the Hierarchy into the _Prefabs folder in the Project pane. Finally, delete the instance of PrefabFloatingScore that remains in the Hierarchy pane.

5. To make the scoreboard, create another GUIText GameObject in the scene (*GameObject > Create Other > GUIText*). Rename this GUIText GameObject to *_Scoreboard*. (The underscore at the beginning of the name will keep it sorted to the top of the Hierarchy pane.) Attach the Scoreboard C# script to the *_Scoreboard* GameObject and give *_Scoreboard* the settings shown in [Figure 31.13](#). This includes dragging the PrefabFloatingScore prefab into the public *prefabFloatingScore* field of the *Scoreboard (Script)* component.

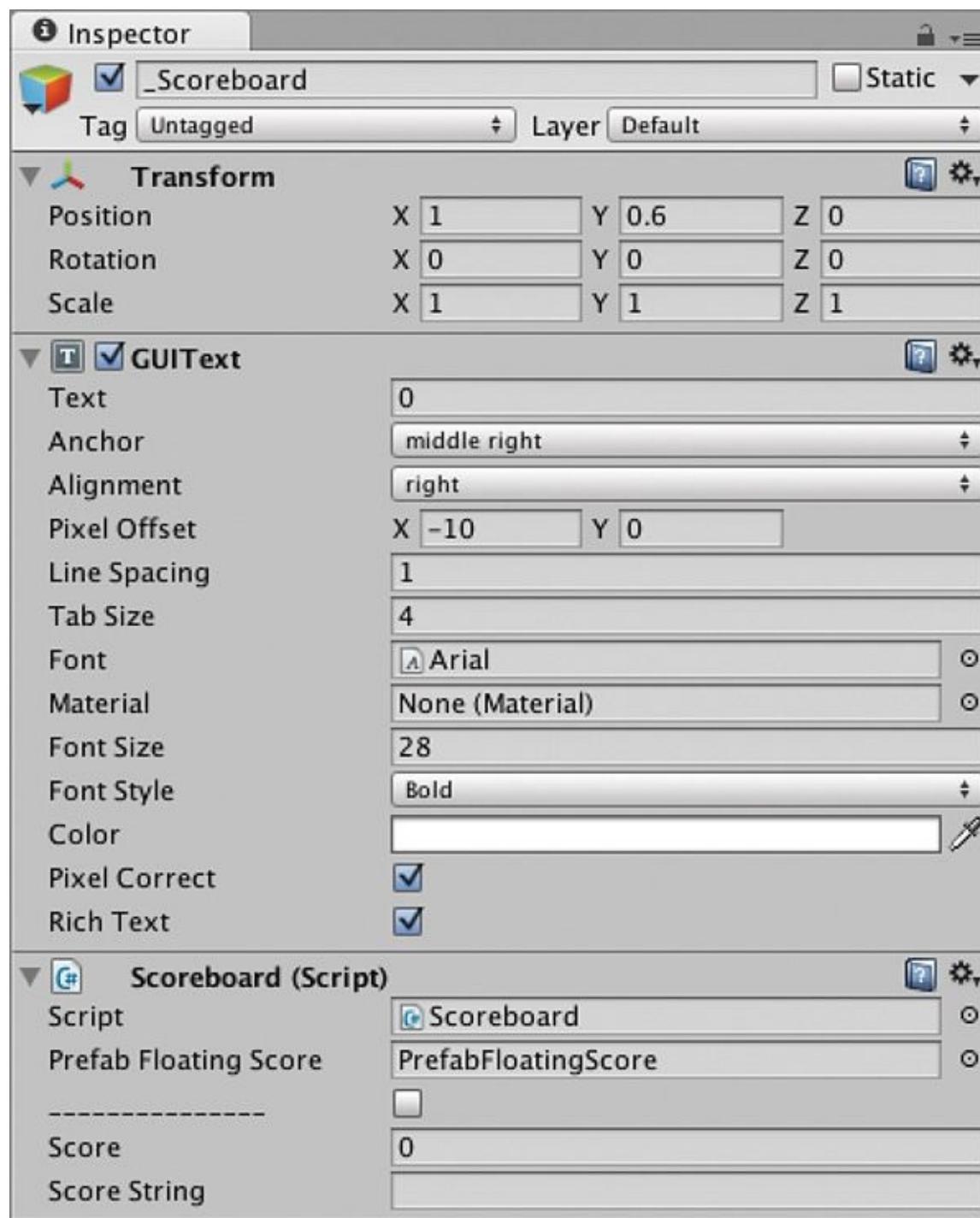


Figure 31.13 The settings for *_Scoreboard*

6. Now, all you need to do is make a few changes to the *Prospector* class to incorporate the new code and GameObjects. Add the following bolded code to the *Prospector* class:

[Click here to view code image](#)

```

public class Prospector : MonoBehaviour {
    ...
    static public int HIGH_SCORE = 0;

    public Vector3 fsPosMid = new Vector3(0.5f, 0.90f, 0);
    public Vector3 fsPosRun = new Vector3(0.5f, 0.75f, 0);
    public Vector3 fsPosMid2 = new Vector3(0.5f, 0.5f, 0);
    public Vector3 fsPosEnd = new Vector3(1.0f, 0.65f, 0);

    public Deck deck;
    ...
    // Fields to track score info
    public int chain = 0;
    public int scoreRun = 0;
    public int score = 0;
    public FloatingScore fsRun;

    void Start () {
        Scoreboard.S.score = score;

        deck = GetComponent<Deck>(); // Get the Deck
        ...
    }

    ...

    // ScoreManager handles all of the scoring
    void ScoreManager(ScoreEvent sEvt) {
        List<Vector3> fsPts;
        switch (sEvt) {
            case ScoreEvent.draw: // Drawing a card
            case ScoreEvent.gameWin: // Won the round
            case ScoreEvent.gameLoss: // Lost the round
                // The same things need to happen whether it's a draw, win, or loss
                chain = 0; // resets the score chain
                score += scoreRun; // add scoreRun to total score
                scoreRun = 0; // reset scoreRun
                // Add fsRun to the _Scoreboard score
                if (fsRun != null) {
                    // Create points for the Bezier curve
                    fsPts = new List<Vector3>();
                    fsPts.Add( fsPosRun );
                    fsPts.Add( fsPosMid2 );
                    fsPts.Add( fsPosEnd );
                    fsRun.reportFinishTo = Scoreboard.S.gameObject;
                    fsRun.Init(fsPts, 0, 1);
                    // Also adjust the fontSize
                    fsRun.fontSizes = new List<float>(new float[] {28,36,4});
                    fsRun = null; // Clear fsRun so it's created again
                }
                break;
            case ScoreEvent.mine: // Remove a mine card
                chain++; // increase the score chain
                scoreRun += chain; // add score for this card to run
                // Create a FloatingScore for this score
                FloatingScore fs;
                // Move it from the mousePosition to fsPosRun
                Vector3 p0 = Input.mousePosition;
                p0.x /= Screen.width;
                p0.y /= Screen.height;
                fsPts = new List<Vector3>();
                fsPts.Add( p0 );
                fsPts.Add( fsPosMid );
                fsPts.Add( fsPosRun );
        }
    }
}

```

```

        fs = Scoreboard.S.CreateFloatingScore(chain, fsPts);
        fs.fontSizes = new List<float>(new float[] {4,50,28});
        if (fsRun == null) {
            fsRun = fs;
            fsRun.reportFinishTo = null;
        } else {
            fs.reportFinishTo = fsRun.gameObject;
        }
        break;
    }
    ...
}
}

```

Now when you play the game, you should see the score flying around. This is actually pretty important because it helps your players understand where the score is coming from and helps reveal the mechanics of the game to them through play (rather than requiring them to read instructions).

Adding Some Art to the Game

Let's add some theming to the game by adding a background. In the `_Sprites` folder with the various card elements are a PNG named `ProspectorBackground` and a material named `ProspectorBackground Mat`. These are already set up for you, since you already learned how to do so in previous chapters.

In Unity, add a quad to the scene (`GameObject > Create Other > Quad`). Drag the `ProspectorBackground Mat` onto the quad. Rename the quad `ProspectorBackground` and set its transform as follows:

ProspectorBackground (Quad) P:[0,0,0] R:[0,0,0] S:[26.667,20,1]

Because `_MainCamera`'s orthographic size is 10, that means that it is 10 units between the center of the screen and the nearest edge (which in this case is the top and bottom), for a total height of 20 units visible on screen. The `ProspectorBackground` quad is 20 units high (y-scale) because of this. And, because the screen is at a 4:3 aspect ratio, $20 / 3 * 4 = 26.667$ is the width (x-scale) that we need to set the background to.

When you play the game now, it should look something like [Figure 31.14](#).



Figure 31.14 The Prospector game with a background

Announcing the Beginning and End of Rounds

I'm sure you've noticed that the rounds of the game end rather abruptly. Let's do something about that. First off, we'll delay the actual reloading of the level using an `Invoke()` function. Add the following bolded code to Prospector:

[Click here to view code image](#)

```
public class Prospector : MonoBehaviour {
    static public Prospector S;
    static public int SCORE_FROM_PREV_ROUND = 0;
    static public int HIGH_SCORE = 0;

    public float reloadDelay = 1f; // The delay between rounds

    public Vector3 fsPosMid = new Vector3(0.50f, 0.90f, 0);
    ...

    // Called when the game is over. Simple for now, but expandable
    void GameOver(bool won) {
        if (won) {
            ScoreManager(ScoreEvent.gameWin);
        } else {
            ScoreManager(ScoreEvent.gameLoss);
        }
        // Reload the scene in reloadDelay seconds
        // This will give the score a moment to travel
    }
}
```

```

        Invoke ("ReloadLevel", reloadDelay); //1
        // Application.LoadLevel("__Prospector_Scene_0"); // Now commented out
    }

    void ReloadLevel() {
        // Reload the scene, resetting the game
        Application.LoadLevel("__Prospector_Scene_0");
    }
    ...
}

```

1. The `Invoke()` at `//1` works by calling a function named `ReloadLevel` in `reloadDelay` seconds. This is similar to how `SendMessage()` works, but it does so with a delay. Now when you play the game, it will wait for the score for the final run to move all the way to the `_Scoreboard` before the game reloads.

Giving the Player Feedback on Her Score

We also want to tell the player how she did at the end of each round. Add two new `GUIText`s to the scene and name them `GameOver` and `RoundResult`. Give them the settings shown in [Figure 31.15](#).

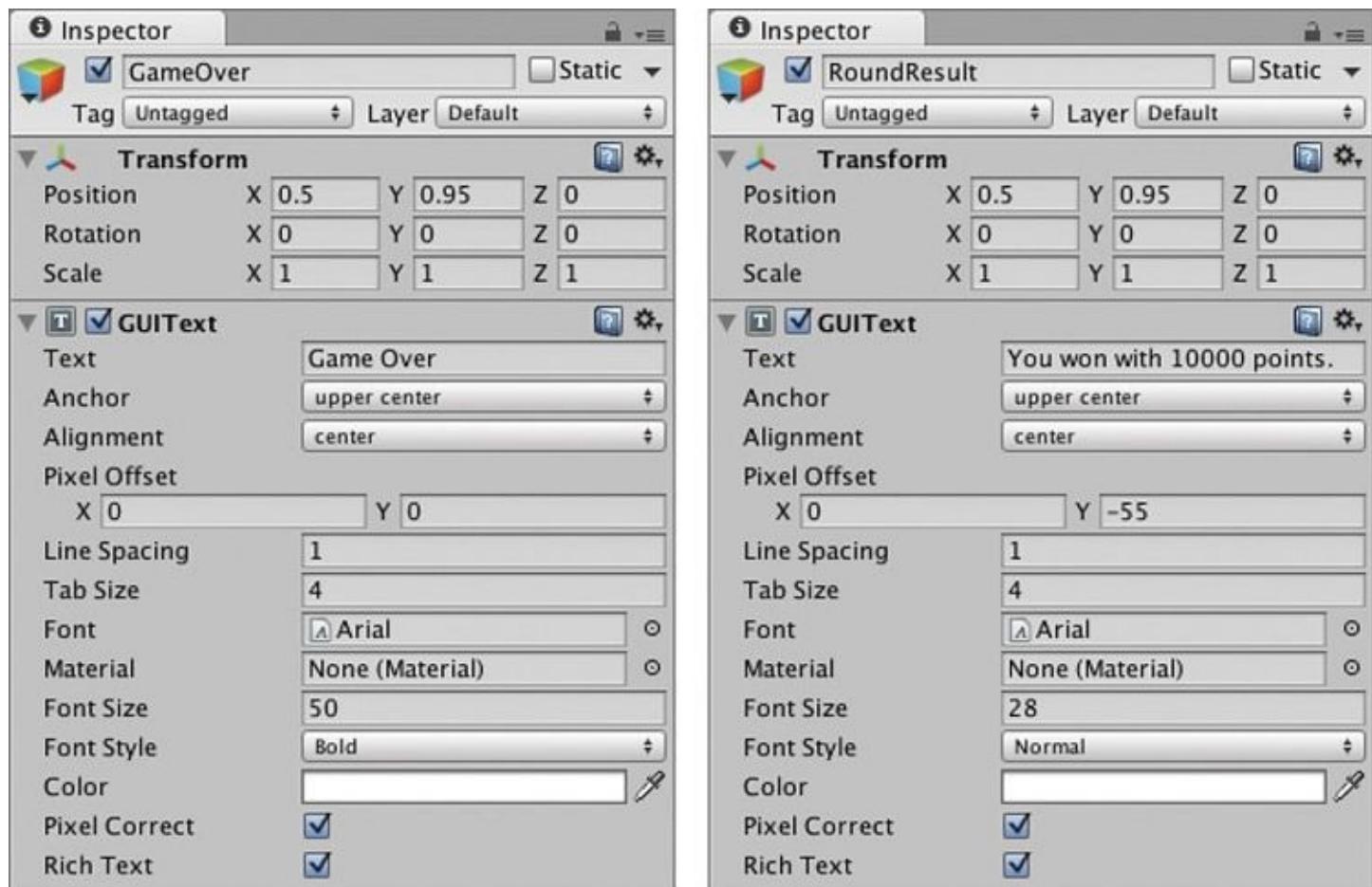


Figure 31.15 The settings for the `GameOver` and `RoundResult` `GUIText`s

At this time, you should also add another `GUIText` named `HighScore` to display the high score information to the player. Give `HighScore` the settings shown in [Figure 31.16](#).

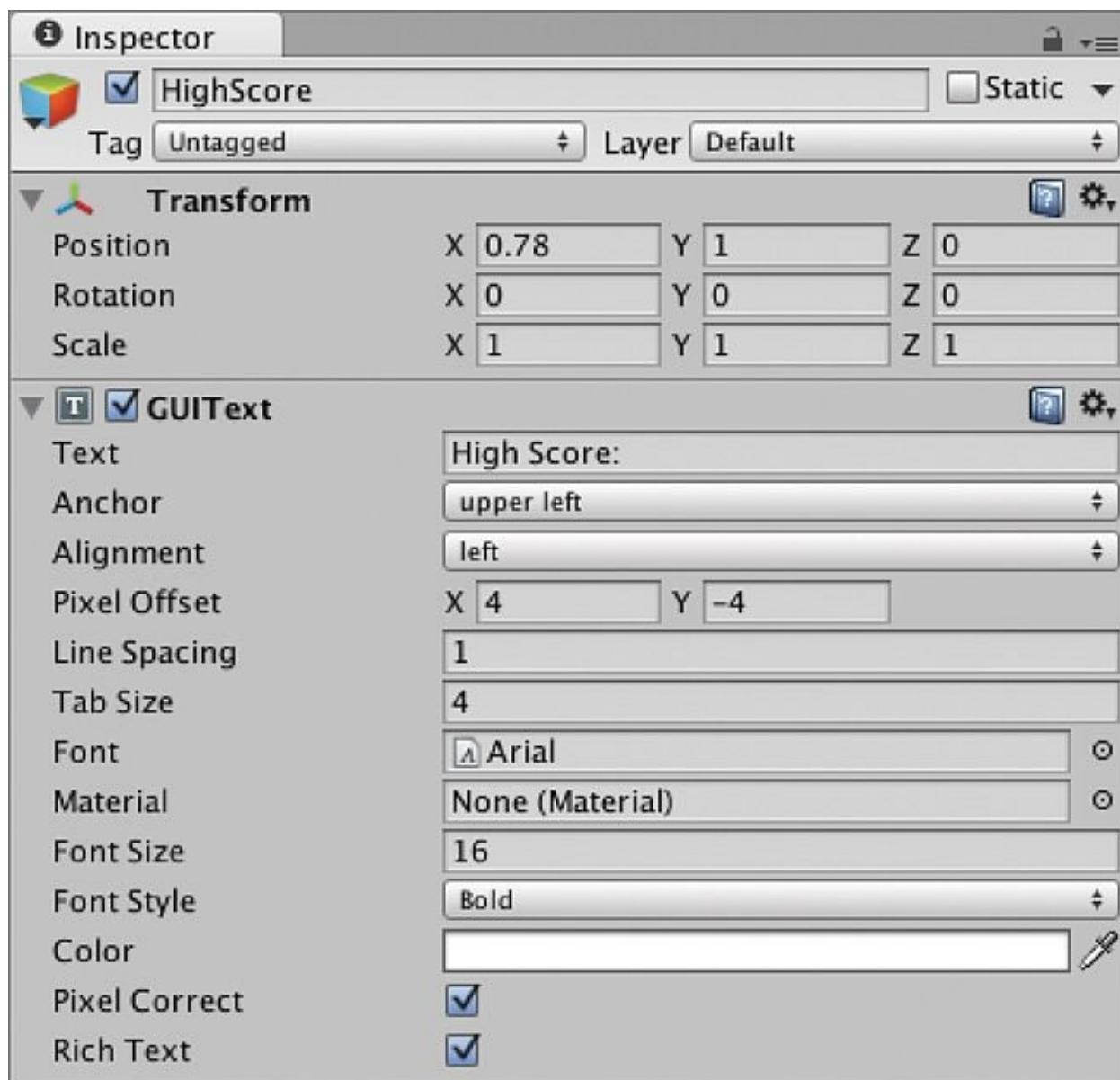


Figure 31.16 The settings for the HighScore GUIText

The numbers in these settings were determined by trial and error, and you should feel free to adjust them as you see fit.

To make these GUITexts functional, add the following bolded code to the `Prospector` class:

[Click here to view code image](#)

```
public class Prospector : MonoBehaviour {
    ...
    public FloatingScore fsRun;
    public GUIText GTGameOver;
    public GUIText GTRoundResult;

    void Awake() {
        ...
        // Set up the GUITexts that show at the end of the round
        // Get the GUIText Components
        GameObject go = GameObject.Find ("GameOver");
        if (go != null) {
            GTGameOver = go.GetComponent<GUIText>();
        }
        go = GameObject.Find ("RoundResult");
        if (go != null) {
```

```

    GTRoundResult = go.GetComponent<GUIText>();
}
// Make them invisible
ShowResultGTs(false);

go = GameObject.Find("HighScore");
string hScore = "High Score: "+Utils.AddCommasToNumber(HIGH_SCORE);
go.GetComponent<GUIText>().text = hScore;
}

void ShowResultGTs(bool show) {
    GTGameOver.gameObject.SetActive(show);
    GTRoundResult.gameObject.SetActive(show);
}
...

// ScoreManager handles all of the scoring
void ScoreManager(ScoreEvent sEvt) {
    ...

    // This second switch statement handles round wins and losses
    switch (sEvt) {
        case ScoreEvent.gameWin:
            GTGameOver.text = "Round Over";
            // If it's a win, add the score to the next round
            // static fields are NOT reset by Application.LoadLevel()
            Prospector.SCORE_FROM_PREV_ROUND = score;
            print ("You won this round! Round score: "+score);
            GTRoundResult.text = "You won this round!\nRound Score: "+score;
            ShowResultGTs(true);
            break;
        case ScoreEvent.gameLoss:
            GTGameOver.text = "Game Over";
            // If it's a loss, check against the high score
            if (Prospector.HIGH_SCORE <= score) {
                print("You got the high score! High score: "+score);
                string sRR = "You got the high score!\nHigh score: "+score;
                GTRoundResult.text = sRR;
                Prospector.HIGH_SCORE = score;
                PlayerPrefs.SetInt("ProspectorHighScore", score);
            } else {
                print ("Your final score for the game was: "+score);
                GTRoundResult.text = "Your final score was: "+score;
            }
            ShowResultGTs(true);
            break;
    ...
}
}
}

```

Now, when you finish a round or game, you should see messages like those in [Figure 31.17](#).



Figure 31.17 Example game over messages

Summary

In this chapter, you created a complete card game that constructs itself from XML files and that contains scoring, background images, and theming. One of the purposes of the tutorials in this book is to give you a framework on which to build your own games. In the next chapter, we do just that. I'll guide you through building the *Bartok* game from the first chapter of the book based on this project.

Next Steps

The following are some possible directions that you can take this game yourself.

Gold Cards

We mentioned this as number 4 in the list of ways to add scoring to the game, but gold cards were not implemented in the chapter. There are graphics in the package you imported for gold cards (both *Card_Back_Gold* and *Card_Front_Gold*). The purpose of the gold cards is to double the value of any run that they are part of. Gold cards can only start in the mine, and any card in the mine has a 10% chance of being a gold card. Try implementing the gold cards on your own.

Compile This Game on a Mobile Device

Though the build settings in this game were designed for an iPad, it's not within the scope of this book to instruct you on actual compilation for a mobile device. Unity has several pages that document this; however, and I recommend that you look at the proper one for the device that you own. In order to keep the information here as current as possible, my best recommendation for you is to do a web search for *Unity getting started* and the name of the mobile platform on which you want to develop (e.g., *Unity getting started iOS*). Right now, that could be *iOS*, *Android*, *BlackBerry*, or *Windows Phone*. The Unity documentation includes “getting started” pages for all of these platforms.

In my personal experience, I have found compilation on Android devices to be the easiest. Including the time to install and configure the additional software to do so, compiling this game for iOS took about two hours (most of which was spent setting up and paying for my Apple iOS developer account and provisioning profile), and compiling this game for Android took about 20 minutes.

I also highly recommend looking into some of the tools out there that can help you with mobile development. Test Flight (<http://testflightapp.com>) is a tool that helps you to distribute test builds of your game to iOS devices easily over the Internet, and there are many developers who use it. A newer

service, Tsugi (<http://tsugi.co>) takes this a step further by also managing and automating the Unity build process for both iOS and Android through a cloud service. As different members of your team make changes to the game, Tsugi will automatically build it and distribute it to all team members.

Chapter 32. Prototype 5: Bartok

This chapter differs somewhat from the other prototypes because instead of creating an entirely new project, this one shows you how you can build a different game on top of the prototype projects that you've developed while reading this book.

Before starting this project, you should have first completed [Chapter 31](#), “[Prototype 4: Prospector Solitaire](#)” so that you understand the inner workings of the card game framework developed in that chapter.

Bartok is the game you first encountered in [Chapter 1](#), “[Thinking Like a Designer](#).” Now you’ll build it yourself.

Getting Started: Prototype 5

This time, instead of downloading a unitypackage as you did before, just make a duplicate of your entire project folder for Prospector (or you can download it from <http://book.prototools.net> under [Chapter 32](#)). Again, the art assets we’ll be using are constructed from parts of the Vectorized Playing Cards 1.3 by Chris Aguilar.¹

¹ Chris Aguilar, “Vectorized Playing Cards 1.3,” <http://code.google.com/p/vectorized-playing-cards/> ©2011–Chris Aguilar. Licensed under LGPL 3—www.gnu.org/copyleft/lesser.html

Note that *this project will only work with Unity version 4.3 and later*.

Understanding Bartok

For a description of *Bartok* and how to play, see [Chapter 1](#), where it is used extensively as a design exercise. In short, *Bartok* is very similar to the commercial game *Uno*, except that it is played with a standard deck of cards, and in the traditional Bartok card game, the winner of each round is able to add a rule to the game. In the [Chapter 1](#) example, we also included three variations of the rules, but those will not be created in this chapter; I’ll leave that to you to accomplish later.

To play an online version of the Bartok game, visit <http://book.prototools.net> and look under [Chapter 1](#).

Making a New Scene

As with much of this project, the scene we will use will be based on the scene from *Prospector*. Click `_Prospector_Scene_0` in the Project pane and then choose *Edit > Duplicate* from the menu bar. This will make a new Scene named `_Prospector_Scene_1`. Rename this to `_Bartok_Scene_0` and double-click it to open it. You can tell that it has opened because the title bar of the Unity window will change to reflect the new scene name.

Cleaning the Scene

Let’s get rid of some of the things we don’t need. Select `_Scoreboard` and `HighScore` in the Hierarchy pane and delete them (*Edit > Delete* from the menu bar). This game won’t be scored, so we don’t need either of those.

Similarly, you can delete both `GameOver` and `RoundResult` from this scene. We’ll be making use of them later but can always grab copies from `_Prospector_Scene_0` when we need them.

Select `_MainCamera` and remove the *Prospector (Script)* and *Layout (Script)* components (right-click

the name of each [or click the gear to the right of the name of each] and choose *Remove Component*). You should be left with a `_MainCamera` that has all the proper settings for Transform and Camera and also still has a *Deck (Script)* component.

Lastly, let's change the background. Start by selecting the `ProspectorBackground` GameObject in the Hierarchy pane (not the `Texture2D` `ProjectPane`) and renaming it *BartokBackground*. Then create a new Material in the `_Sprites` folder (`Assets > Create > Material` from the menu bar) and name it *BartokBackground Mat*. Drag this new material on to `BartokBackground`. You'll notice in the Game pane that this made things very dark. (This is because the new material has a Diffuse shader while the previous material used the `UnlitAlpha` shader.) To remedy this, add a directional light to the scene (`GameObject > Create Other > Directional Light`). The transform for the `BartokBackground` and directional light should be as follows:

BartokBackground (Quad)	P:[0,0,1]	R:[0,0,0]	S:[26.667,20,1]
Directional Light	P:[0,0,0]	R:[50,-30,0]	S:[1,1,1]

This should set the scene properly.

The Importance of Adding Card Animation

This will be a game for a single human player, but the game of Bartok works best with four players, so three of the players will be AIs (artificial intelligences). Because Bartok is such a simple game, they won't have to be good AIs; they just need to act. The other thing that will need to happen is that we will have to let the player know whose turn it is and what the other players are doing. For this to work, we're going to make the cards animate from place to place in this game. This wasn't necessary in `Prospector` because the player was taking all of the actions herself, and it was obvious to her what the result should be. Because the player of Bartok is presented with three other hands that will be face-down to her, the animation can be used as an important way to message what actions are being taken by the AI players.

Much of the challenge in designing this tutorial was in creating good animations and in making sure that the game waited properly for each animation to end before moving on to the next thing. Because of that, you will see use of `SendMessage()` and `Invoke()` in this project as well as the use of more specific callback messages than `SendMessage()` allows. Instead, we will be passing a C# class instance to an object to and then call a *callback function* on the instance when the object is done moving, which is less flexible than `SendMessage()` but faster and more specific and can also be used for C# classes that don't extend `MonoBehaviour`.

Build Settings

Whereas the last project was designed as a mobile app, this will be a standalone application for Mac or PC, so the build settings will need to change. From the menu bar, choose `File > Build Settings`, which will bring up the window shown in [Figure 32.1](#).

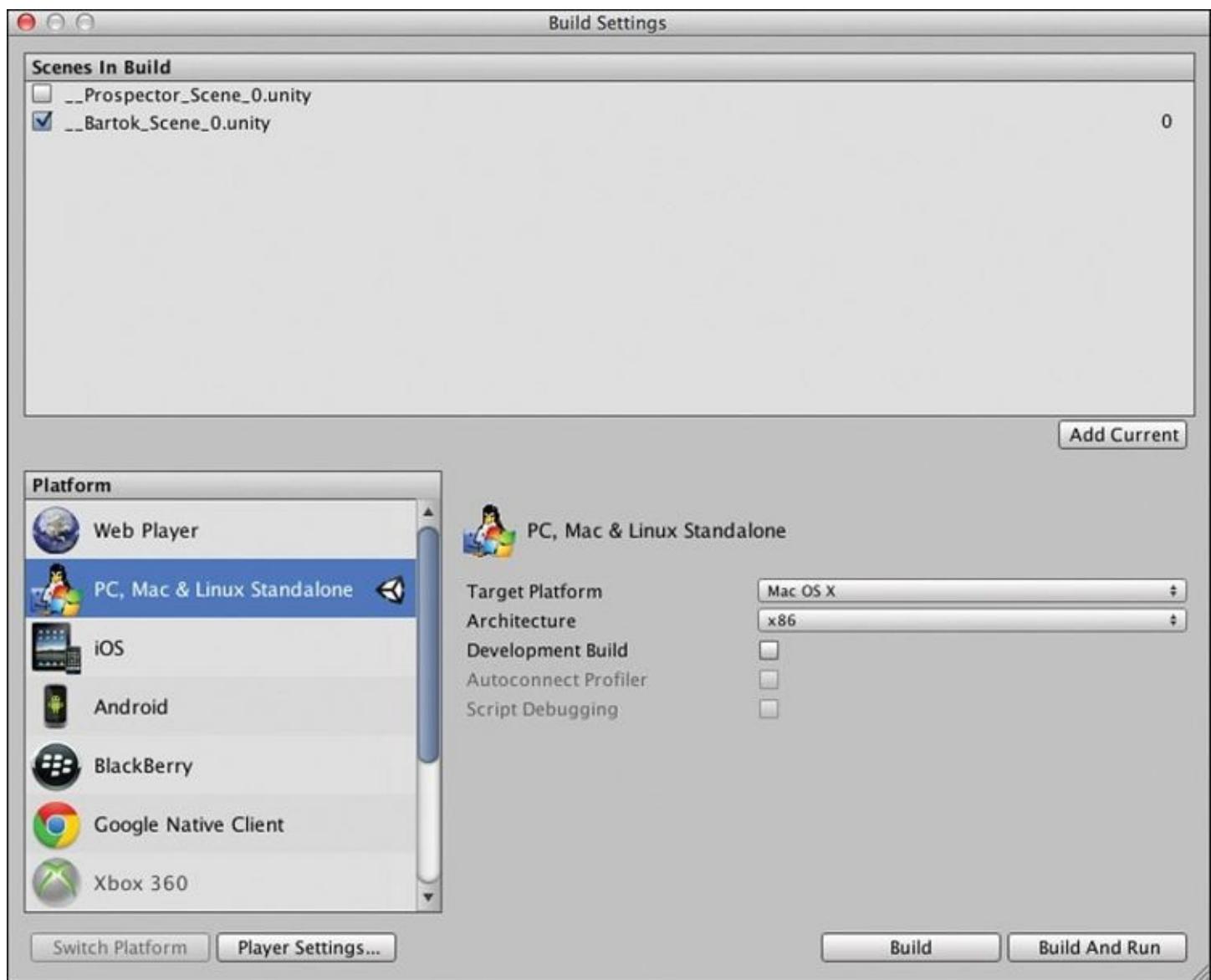


Figure 32.1 The Build Settings window

You'll see that `_Prospector_Scene_0` is currently in the list of *Scenes In Build*, but `_Bartok_Scene_0` is not. Click the *Add Current* button to add `_Bartok_Scene_0` to the list of scenes for this build. Then, uncheck the box next to `_Prospector_Scene_0` to remove it from the list of scenes. Next, select *PC, Mac & Linux Standalone* from the list of platforms and click *Switch Platform*. The *Switch Platform* button will turn gray once the switch is complete. This may take a second, but it should be pretty fast. When you've done this, double-check to make sure that your target platform is set to the type for your machine (for example, *Mac OS X* if you're working on an OS X and *Windows* if you're working on a PC). All the other settings should be fine as they are.

Once your build settings look like the image in [Figure 32.1](#), you can close this window. (Don't click *Build* yet; that will happen after actually making the game.)

Once you've closed the window, look at the pop-up menu under the title of the Game pane. From that list of aspect ratios, change it to *Standalone (1024x768)*. This will ensure that your game aspect ratio looks the same as the examples that you'll see throughout this tutorial.

Coding Bartok

Just as we had a `Prospector` class to manage the game and a `CardProspector:Card` class to extend `Card` and add game-specific capabilities, we will need a `Bartok` and `CardBartok` class in this game. Create both a `Bartok` and a `CardBartok` C# script in the `_Scripts` folder of the Project pane (`Assets > Create > C# Script`). Drag `Bartok` on to `_MainCamera` in the Hierarchy (or assign it some other way; you should know what you're doing by now). Double-click `CardBartok` to open it in MonoDevelop and enter the following code. (If you want, you can copy some of this from `CardProspector`.)

[Click here to view code image](#)

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

// CBState includes both states for the game and to___ states for movement
public enum CBState {
    drawpile,
    toHand,
    hand,
    toTarget,
    target,
    discard,
    to,
    idle
}

// CardBartok extends Card just as CardProspector did.
public class CardBartok : Card {
    // These static fields are used to set values that will be the same
    // for all instances of CardBartok
    static public float      MOVE_DURATION = 0.5f;
    static public string     MOVE_EASING = Easing.InOut;
    static public float      CARD_HEIGHT = 3.5f;
    static public float      CARD_WIDTH = 2f;

    public CBState           state = CBState.drawpile;

    // Fields to store info the card will use to move and rotate
    public List<Vector3>    bezierPts;
    public List<Quaternion>  bezierRots;
    public float              timeStart, timeDuration; // declares 2 fields

    // When the card is done moving, it will call reportFinishTo.SendMessage()
    public GameObject        reportFinishTo = null;

    // MoveTo tells the card to interpolate to a new position and rotation
    public void MoveTo(Vector3 ePos, Quaternion eRot) {
        // Make new interpolation lists for the card.
        // Position and Rotation will each have only two points.
        bezierPts = new List<Vector3>();
        bezierPts.Add ( transform.localPosition ); // Current position
        bezierPts.Add ( ePos );                      // New position
        bezierRots = new List<Quaternion>();
        bezierRots.Add ( transform.rotation );        // Current rotation
        bezierRots.Add ( eRot );                      // New rotation

        // If timeStart is 0, then it's set to start immediately,
        // otherwise, it starts at timeStart. This way, if timeStart is
        // already set, it won't be overwritten.
```

```

if (timeStart == 0) {
    timeStart = Time.time;
}
// timeDuration always starts the same but can be altered later
timeDuration = MOVE_DURATION;

// Setting state to either toHand or toTarget will be handled by the
// calling method
state = CBState.to;
}
// This overload of MoveTo doesn't require a rotation argument
public void MoveTo(Vector3 ePos) {
    MoveTo(ePos, Quaternion.identity);
}

void Update() {
    switch (state) {
        // All the to__ states are ones where the card is interpolating
        case CBState.toHand:
        case CBState.toTarget:
        case CBState.to:
            // Get u from the current time and duration
            // u ranges from 0 to 1 (usually)
            float u = (Time.time - timeStart)/timeDuration;

            // Use Easing class from Utils to curve the u value
            float uC = Easing.Ease (u, MOVE_EASING);

            if (u<0) { // If u<0, then we shouldn't move yet.
                // Stay at the initial position
                transform.localPosition = bezierPts[0];
                transform.rotation = bezierRots[0];
                return;
            } else if (u>=1) { // If u>=1, we're finished moving
                uC = 1; // Set uC=1 so we don't overshoot
                // Move from the to__ state to the following state
                if (state == CBState.toHand) state = CBState.hand;
                if (state == CBState.toTarget) state = CBState.toTarget;
                if (state == CBState.to) state = CBState.idle;
                // Move to the final position
                transform.localPosition = bezierPts[bezierPts.Count-1];
                transform.rotation = bezierRots[bezierPts.Count-1];
                // Reset timeStart to 0 so it gets overwritten next time
                timeStart = 0;

                if (reportFinishTo != null) { //If there's a callback GameObject
                    // ... then use SendMessage to call the CBCallback method
                    // with this as the parameter.
                    reportFinishTo.SendMessage("CBCallback", this);
                    // After calling SendMessage(), reportFinishTo must be set
                    // to null so that it the card doesn't continue to report
                    // to the same GameObject every subsequent time it moves.
                    reportFinishTo = null;
                } else { // If there is nothing to callback
                    // Do nothing
                }
            } else { // 0<=u<1, which means that this is interpolating now
                // Use Bezier curve to move this to the right point
                Vector3 pos = Utils.Bezier(uC, bezierPts);
                transform.localPosition = pos;
                Quaternion rotQ = Utils.Bezier(uC, bezierRots);
                transform.rotation = rotQ;
            }
        }
    }
}

```

```
    }  
    break;  
}  
}
```

A lot of this is an adaptation and expansion on the code that you saw in the preceding chapter for the `FloatingScore` class. The CardBartok version of interpolation also interpolates Quaternions (a class that handles rotations), which will be important because we want the cards in Bartok to fan as if they were being held by a player.

Now, open the `Bartok` class and enter this code. The first thing we want to do is to make sure that the `Deck` class is working properly to create all 52 cards:

[Click here to view code image](#)

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

public class Bartok : MonoBehaviour {
    static public Bartok S;

    public TextAsset             deckXML;
    public TextAsset             layoutXML;
    public Vector3               layoutCenter = Vector3.zero;

    public bool _____;

    public Deck                  deck;
    public List<CardBartok>    drawPile;
    public List<CardBartok>    discardPile;

    void Awake() {
        S = this;
    }

    void Start () {
        deck = GetComponent<Deck>();          // Get the Deck
        deck.InitDeck(deckXML.text);           // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards);          // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which allows
        // deck.cards to be modified by Deck.Shuffle()
    }
}
```

As you can see, most of this is the same as what you saw in Prospector, except that you're now dealing with the `CardBartok` class for cards rather than the `CardProspector` class. At this time, you should also adjust other aspects of `PrefabCard` in the Inspector. Select `PrefabCard` in the `_Prefabs` folder of the Project pane and follow these instructions:

1. Set the Box Collider component's *Is Trigger* field to true.
 2. Set the *Size.z* of the Box Collider component to 0.1.
 3. Add a Rigidbody component to PrefabCard (*Component > Physics > Rigidbody*).

4. Set the Rigidbody's *Use Gravity* field to false.

5. Set the Rigidbody's *Is Kinematic* field to true.

When finished, the Box Collider and Rigidbody components on PrefabCard should look like those shown in [Figure 32.2](#).

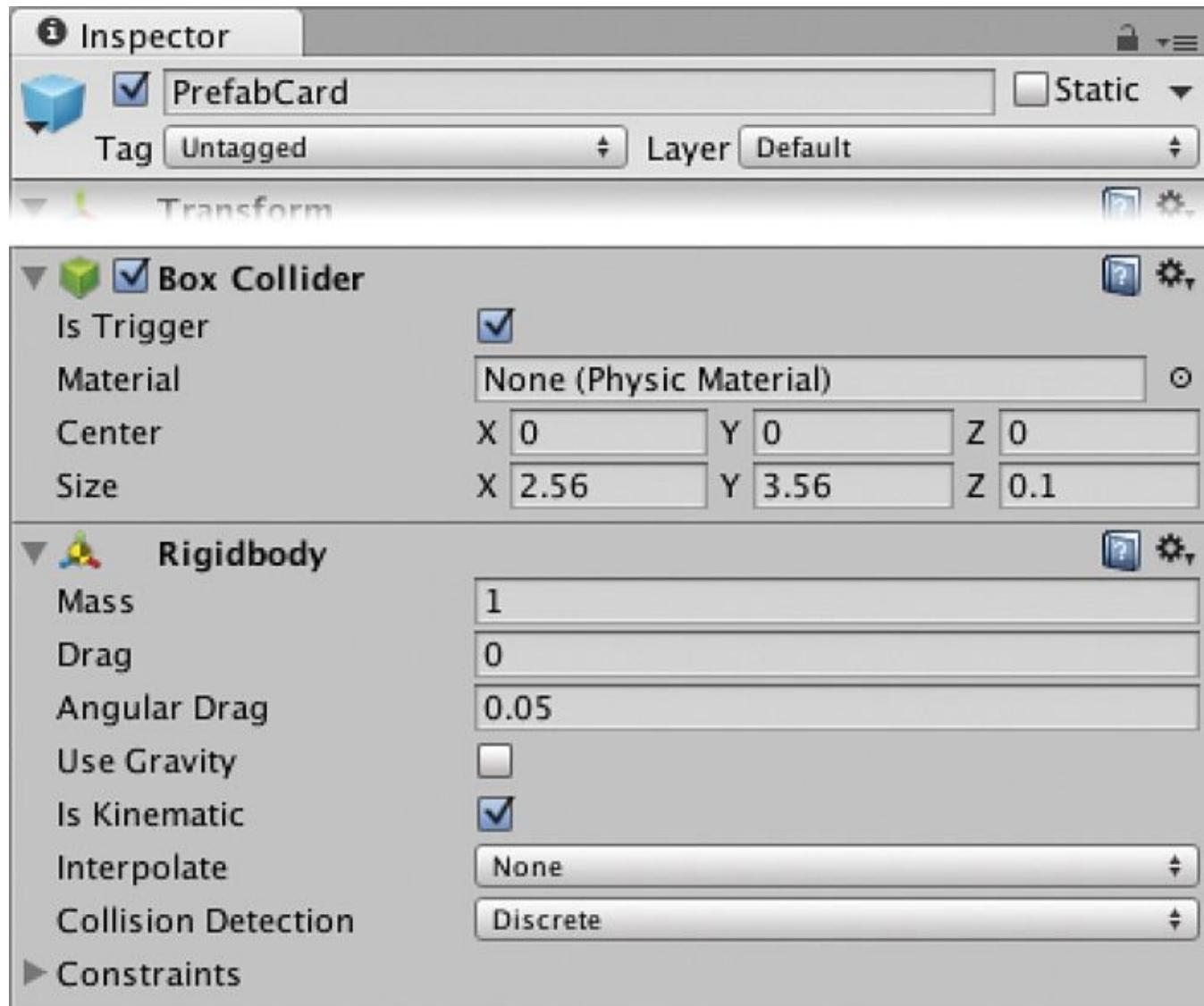


Figure 32.2 Box Collider and Rigidbody settings for PrefabCard

You will also need to make some changes inside the Unity editor before the code you just wrote will work. In the Hierarchy pane, select `_MainCamera`. The attached `Bartok (Script)` component is at the bottom of the inspector. (If you want to move it up, you can click the gear next to its name and choose `Move Up`.) Set the `DeckXML` field of `Bartok (Script)` to the `DeckXML` file that is in the `Resources` folder of the Project pane. (Because the deck remains unchanged [still 13 cards of 4 suits], this is the same file that was used by `Prospector`.)

Now select `PrefabCard` in the `_Prefabs` folder of the Project pane. You're going to need to swap a new `CardBartok (Script)` component for the existing `CardProspector` one. Next to the `Script` variable in the `CardProspector (Script)` component, click the `Target` button and choose `CardBartok` to replace it. Alternatively, you could just attach the `CardBartok` script to `PrefabCard` and then remove the `CardProspector (Script)` component.

Now when you press `Play`, you should see a grid of cards just as you saw in the early stages of `Prospector`.

The Game Layout

The layout for Bartok differs significantly from Prospector. In Bartok, there will be a draw pile and discard pile in the middle of the screen as well as four hands of cards distributed to the top, left, bottom, and right sides of the screen. The hands should be fanned as if they were being held by a player (see [Figure 32.3](#)).

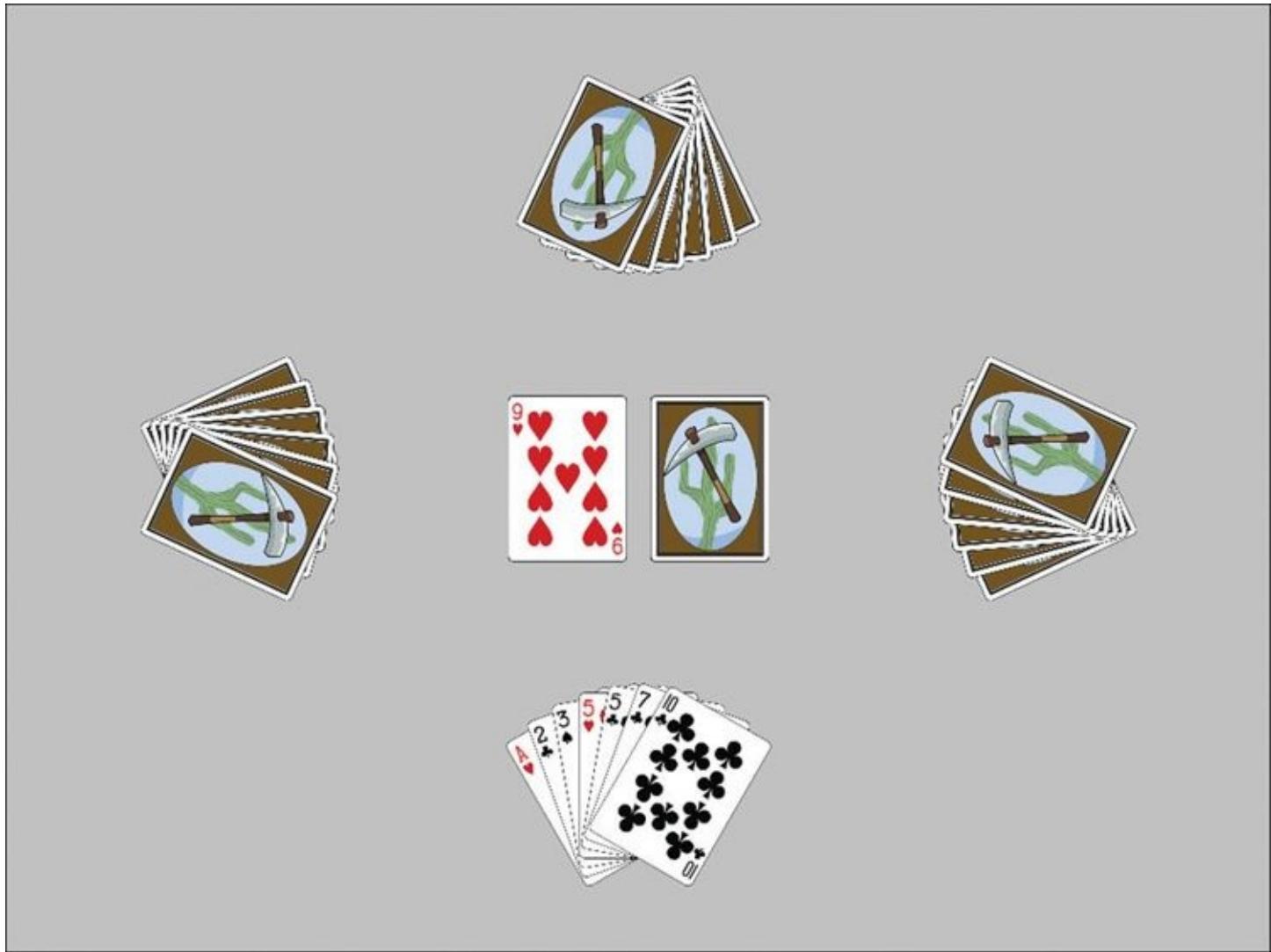


Figure 32.3 The eventual layout of Bartok

This will require a somewhat different layout XML document than was used for Prospector. Select LayoutXML in the Resources folder of the Project pane and duplicate it (*Edit > Duplicate*). Name the duplicate *BartokLayoutXML* and enter the following text. (Bold text differs from the original LayoutXML text.)

[Click here to view code image](#)

```
<xml>
  <!-- This file includes info for laying out the Bartok card game. -->
  <!-- The multiplier is multiplied by the x and y attributes below. -->
  <!-- This determines how loose or tight the layout is. -->
  <multiplier x="1" y="1" />

  <!-- This positions the draw pile and staggers it -->
  <slot type="drawpile" x="1.5" y="0" xstagger="0.05" layer="1"/>

  <!-- This positions the discard pile -->
  <slot type="discardpile" x="-1.5" y="0" layer="2"/>
```

```

<!-- This positions the target card -->
<slot type="target" x="-1.5" y="0" layer="4"/>

<!-- These slots are for the four hands held by the four players -->
<slot type="hand" x="0" y="-8" rot="0" player="1" layer="3"/>
<slot type="hand" x="-10" y="0" rot="270" player="2" layer="3"/>
<slot type="hand" x="0" y="8" rot="180" player="3" layer="3"/>
<slot type="hand" x="10" y="0" rot="90" player="4" layer="3"/>

</xml>

```

Now, the class that does the layout must also be rewritten to both fan the cards properly and to take advantage of the new ability to interpolate cards. Create a new C# script named *BartokLayout* and enter this code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

// SlotDef class is not based on MonoBehaviour, so it doesn't need its own file.
[System.Serializable] // Makes SlotDef able to be seen in the Unity Inspector
public class SlotDef {
    public float x;
    public float y;
    public bool faceUp=false;
    public string layerName="Default";
    public int layerID = 0;
    public int id;
    public List<int> hiddenBy = new List<int>(); // Unused in Bartok
    public float rot; // rotation of hands
    public string type="slot";
    public Vector2 stagger;
    public int player; // player number of a hand
    public Vector3 pos; // pos derived from x, y, & multiplier
}

public class BartokLayout : MonoBehaviour {
    ...
}

```

Save this code and return to Unity. You'll notice that this causes an error in the console:

“error CS0101: The namespace ‘global::’ already contains a definition for ‘SlotDef’.”

This is because the public class *SlotDef* in the Layout script (from Prospector) is conflicting with the public class *SlotDef* in the new *BartokLayout* script. Either delete the Layout script entirely or open the Layout script in MonoDevelop and comment out the section defining *SlotDef*. To comment out a large chunk of code, just place a /* before the code and a */ after the code you wish to comment. You can also comment out a large section by selecting the lines of code in MonoDevelop and choosing *Edit > Format > Toggle Line Comment(s)* from the menu bar, which will place a single line comment (//) before each line you have selected. After you have eliminated the *SlotDef* class from the Layout script, return to the *BartokLayout* script and continue editing it by adding the bolded lines in the following code listing:

[Click here to view code image](#)

```

public class BartokLayout : MonoBehaviour {
    public PT_XMLReader xmlr; // Just like Deck, this has an PT_XMLReader
    public PT_XMLHashtable xml; // This variable is for faster xml access
}

```

```

public Vector2 multiplier; // Sets the spacing of the tableau
// SlotDef references
public List<SlotDef> slotDefs; // The SlotDefs hands
public SlotDef drawPile;
public SlotDef discardPile;
public SlotDef target;

// This function is called to read in the LayoutXML.xml file
public void ReadLayout(string xmlText) {
    xmldr = new PT_XMLReader();
    xmldr.Parse(xmlText); // The XML is parsed
    xml = xmldr.xml["xml"][0]; // And xml is set as a shortcut to the XML

    // Read in the multiplier, which sets card spacing
    multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
    multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

    // Read in the slots
    SlotDef tSD;
    // slotsX is used as a shortcut to all the <slot>s
    PT_XMLHashList slotsX = xml["slot"];

    for (int i=0; i<slotsX.Count; i++) {
        tSD = new SlotDef(); // Create a new SlotDef instance
        if (slotsX[i].HasAtt("type")) {
            // If this <slot> has a type attribute parse it
            tSD.type = slotsX[i].att("type");
        } else {
            // If not, set its type to "slot"
            tSD.type = "slot";
        }

        // Various attributes are parsed into numerical values
        tSD.x = float.Parse( slotsX[i].att("x") );
        tSD.y = float.Parse( slotsX[i].att("y") );
        tSD.pos = new Vector3( tSD.x*multiplier.x, tSD.y*multiplier.y, 0 );

        // Sorting Layers
        tSD.layerID = int.Parse( slotsX[i].att("layer") );
        // In this game, the Sorting Layers are named 1, 2, 3, ...through 10
        // This converts the number of the layerID into a text layerName
        tSD.layerName = tSD.layerID.ToString();
        // The layers are used to make sure that the correct cards are
        // on top of the others. In Unity 2D, all of our assets are
        // effectively at the same Z depth, so sorting layers are used
        // to differentiate between them.

        // pull additional attributes based on the type of each <slot>
        switch (tSD.type) {
            case "slot":
                // ignore slots that are just of the "slot" type
                break;

            case "drawpile":
                // The drawPile xstagger is read but not actually used in Bartok
                tSD.stagger.x = float.Parse( slotsX[i].att("xstagger") );
                drawPile = tSD;
                break;

            case "discardpile":
                discardPile = tSD;
                break;

            case "target":

```

```
// The target card has a different layer from discardPile
target = tSD;
break;

case "hand":
    // Information for each player's hand
    tSD.player = int.Parse( slotsX[i].att("player") );
    tSD.rot = float.Parse( slotsX[i].att("rot") );
    slotDefs.Add (tSD);
    break;

    }

}

}
```

To use this code, you'll need to attach the BartokLayout script to `_MainCamera`. (Drag the `BartokLayout` script from the Project pane onto `_MainCamera` in the Hierarchy pane.) You will also need to assign the `BartokLayoutXML` to the `layoutXML` field of the *Bartok (Script)* component on `_MainCamera`.

Now, add the following bolded code to the Bartok script to make use of BartokLayout:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {
    static public Bartok S;
    ...

    public List<CardBartok>      discardPile;
    public BartokLayout           layout;
    public Transform               layoutAnchor;
    ...

    void Start () {
        deck = GetComponent<Deck>();           // Get the Deck
        deck.InitDeck(deckXML.text);             // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards);           // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which allows
        // deck.cards to be modified by Deck.Shuffle()

        layout = GetComponent<BartokLayout>();    // Get the Layout
        layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it

        drawPile = UpgradeCardsList( deck.cards );
    }

    // UpgradeCardsList casts the Cards in LCD to be CardBartoks
    // Of course, they were all along, but this lets Unity know it
    List<CardBartok> UpgradeCardsList(List<Card> LCD) {
        List<CardBartok> lCB = new List<CardBartok>();
        foreach( Card tCD in LCD ) {
            lCB.Add ( tCD as CardBartok );
        }
        return( lCB );
    }
}
```

When you run the project now, you should be able to select `_MainCamera` from the Hierarchy pane

and expand the variables in the *BartokLayout (Script)* component to see that they're being read in correctly. You should also look at the `drawPile` field of *Bartok (Script)* to see that it is properly filled with 52 shuffled *CardBartok* instances.

The Player Class

Because this game has four players, I've chosen to create a class to represent players that can do things like gather cards into a hand and eventually choose what to play using simple artificial intelligence. One thing that is unique about the `Player` class relative to others that you've written is that the `Player` class does *not* extend `MonoBehaviour` (or any other class). This means that it doesn't receive calls from `Awake()`, `Start()`, or `Update()` and that you can't call some functions like `print()` from within it or attach it to a `GameObject` as a component. However, none of that is necessary for the `Player` class, so it is actually easier in this case to work without it.

Create a new C# script named `Player` and enter this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Linq; // Enables LINQ queries, which will be explained soon

// The player can either be human or an ai
public enum PlayerType {
    human,
    ai
}

// The individual player of the game
// Note: Player does NOT extend MonoBehaviour (or any other class)

[System.Serializable] // Make the Player class visible in the Inspector pane
public class Player {

    public PlayerType           type = PlayerType.ai;
    public int                  playerNum;

    public List<CardBartok>    hand; // The cards in this player's hand

    public SlotDef              handSlotDef;

    // Add a card to the hand
    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Add the card to the hand
        hand.Add (eCB);

        return( eCB );
    }

    // Remove a card from the hand
    public CardBartok RemoveCard(CardBartok cb) {
        hand.Remove(cb);
        return(cb);
    }
}
```

Now, add the following code to Bartok to make use of the Player:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {
    ...
    public Vector3 layoutCenter = Vector3.zero;
    // The number of degrees to fan each card in a hand
    public float handFanDegrees = 10f;
    public bool _____;
    ...
    public Transform layoutAnchor;
    public List<Player> players;
    public CardBartok targetCard;
    ...

    void Start () {
        ...
        drawPile = UpgradeCardsList( deck.cards );
        LayoutGame();
    }

    List<CardBartok> UpgradeCardsList(List<Card> lCD) {
        ...
    }

    // Position all the cards in the drawPile properly
    public void ArrangeDrawPile() {
        CardBartok tCB;

        for (int i=0; i<drawPile.Count; i++) {
            tCB = drawPile[i];
            tCB.transform.parent = layoutAnchor;
            tCB.transform.localPosition = layout.drawPile.pos;
            // Rotation should start at 0
            tCB.faceUp = false;
            tCB.SetSortingLayerName(layout.drawPile.layerName);
            tCB.SetSortOrder(-i*4); // Order them front-to-back
            tCB.state = CBState.drawpile;
        }
    }

    // Perform the initial game layout
    void LayoutGame() {
        // Create an empty GameObject to serve as an anchor for the tableau
        if (layoutAnchor == null) {
            GameObject tGO = new GameObject("_LayoutAnchor");
            // ^ Create an empty GameObject named _LayoutAnchor in the Hierarchy
            layoutAnchor = tGO.transform; // Grab its Transform
            layoutAnchor.transform.position = layoutCenter; // Position it
        }

        // Position the drawPile cards
        ArrangeDrawPile();

        // Set up the players
        Player pl;
        players = new List<Player>();
        foreach (SlotDef tSD in layout.slotDefs) {
```

```
pl = new Player();
pl.handSlotDef = tSD;
players.Add(pl);
pl.playerNum = players.Count;
}
players[0].type = PlayerType.human; // Make the 0th player human

}

// The Draw function will pull a single card from the drawPile and return it
public CardBartok Draw() {
    CardBartok cd = drawPile[0];      // Pull the 0th CardProspector
    drawPile.RemoveAt(0);           // Then remove it from List<CardBartok> drawPile
    return(cd);                     // And return it
}

// This Update method is used to test adding cards to players' hands
void Update() {
    if (Input.GetKeyDown(KeyCode.Alpha1)) {
        players[0].AddCard(Draw ());
    }
    if (Input.GetKeyDown(KeyCode.Alpha2)) {
        players[1].AddCard(Draw ());
    }
    if (Input.GetKeyDown(KeyCode.Alpha3)) {
        players[2].AddCard(Draw ());
    }
    if (Input.GetKeyDown(KeyCode.Alpha4)) {
        players[3].AddCard(Draw ());
    }
}
```

Now run the game again. Select `_MainCamera` in the Hierarchy and find the `Players` field on the `Bartok (Script)` component. Open the disclosure triangle for `Players`, and you'll see four elements, one for each player. Open those disclosure triangles, as well, and then open up the disclosure triangles for each hand. Because of the test code in the new `Update()` method, if you click in the Game pane (which gives the game focus and allows it to react to keyboard input), you can press the number keys 1 to 4 on your keyboard (across the top of the keyboard, not the keypad) and watch cards be added to the players' hands. The Inspector for the `Bartok (Script)` component should show cards being added to hands as shown in [Figure 32.4](#).

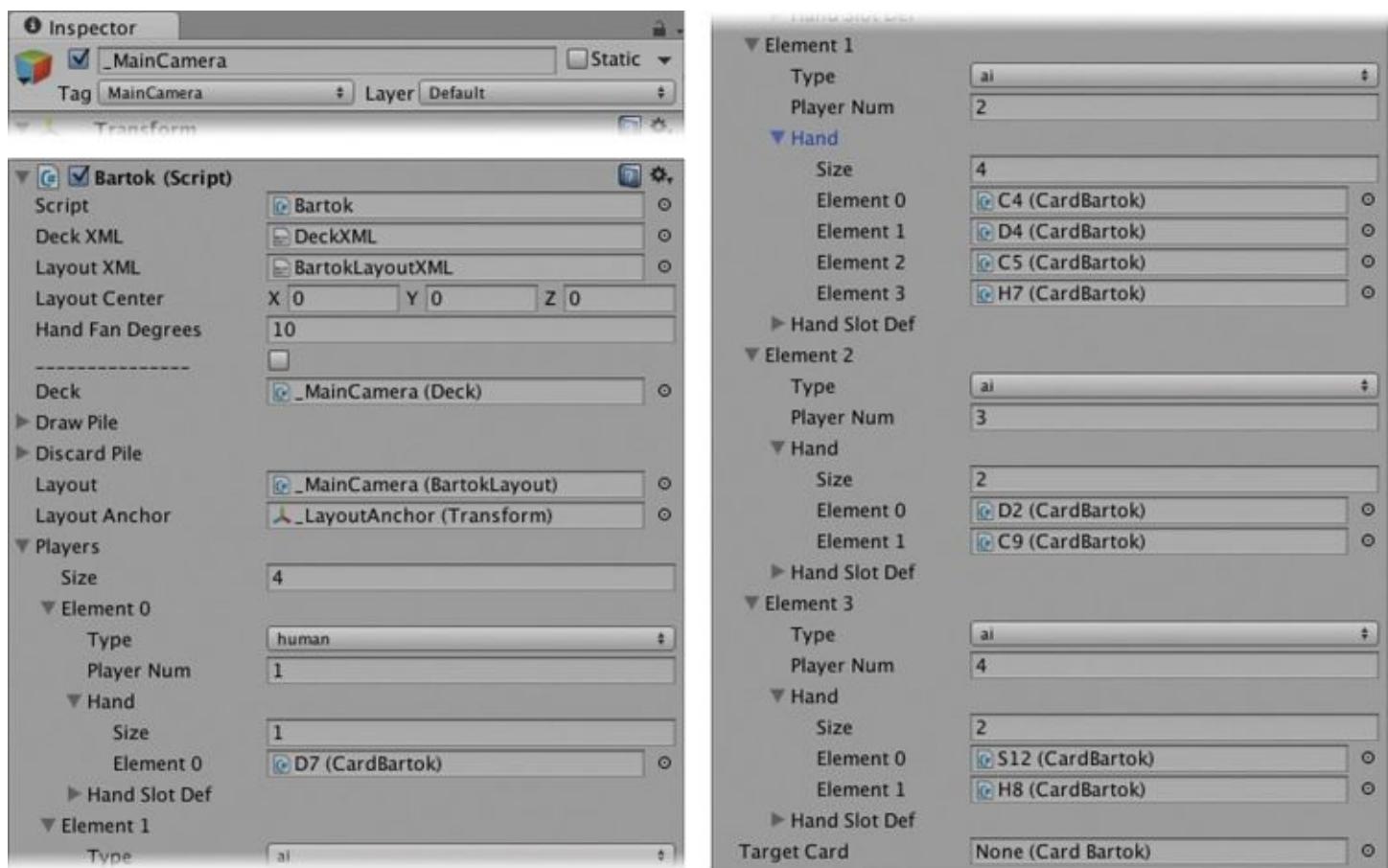


Figure 32.4 Bartok (Script) component showing players and their hands

This `Update()` method of course won't be used in the final version of the game, but it is often useful to build little functions like this that allow you to test features before other aspects of the game are ready. In this case, we needed a way to test whether the `Player.AddCard()` method worked properly, and this was a quick way to do so.

Fanning the Hands

Now that cards are being moved from the `drawPile` into players' hands, it's time to graphically represent them there. Add the following code to `Player` to make this happen:

[Click here to view code image](#)

```
public class Player {
    ...

    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Add the card to the hand
        hand.Add (eCB);
        FanHand();
        return( eCB );
    }

    public CardBartok RemoveCard(CardBartok cb) {
        hand.Remove(cb);
        FanHand();
        return(cb);
    }

    public void FanHand() {
        // startRot is the rotation about Z of the first card
    }
}
```

```

float startRot = 0;
startRot = handSlotDef.rot;
if (hand.Count > 1) {
    startRot += Bartok.S.handFanDegrees * (hand.Count-1) / 2;
}
// Then each card is rotated handFanDegrees from that to fan the cards

// Move all the cards to their new positions
Vector3 pos;
float rot;
Quaternion rotQ;
for (int i=0; i<hand.Count; i++) {
    rot = startRot - Bartok.S.handFanDegrees*i; // Rot about the z axis
    // ^ Also adds the rotations of the different players' hands
    rotQ = Quaternion.Euler( 0, 0, rot );
    // ^ Quaternion representing the same rotation as rot

    // pos is a V3 half a card height above [0,0,0] (i.e., [0,1.75,0])
    pos = Vector3.up * CardBartok.CARD_HEIGHT / 2f;

    // Multiplying a Quaternion by a Vector3 rotates that Vector3 by
    // the rotation stored in the Quaternion. The result gives us a
    // vector above [0,0,0] that has been rotated by rot degrees
    pos = rotQ * pos;

    // Add the base position of the player's hand (which will be at the
    // bottom-center of the fan of the cards)
    pos += handSlotDef.pos;
    // This staggers the cards in the z direction, which isn't visible
    // but which does keep their colliders from overlapping
    pos.z = -0.5f*i;

    // Set the localPosition and rotation of the ith card in the hand
    hand[i].transform.localPosition = pos;
    hand[i].transform.rotation = rotQ;
    hand[i].state = CBState.hand;

    // This uses a comparison operator to return a true or false bool
    // So, if (type == PlayerType.human), hand[i].faceUp is set to true
    hand[i].faceUp = (type == PlayerType.human);

    // Set the SortOrder of the cards so that they overlap properly
    hand[i].SetSortOrder(i*4);
}

}
}

```

Now if you play the scene and press the numbers 1, 2, 3, and 4 on your keyboard, you should see cards jumping into the players' hands and being fanned correctly. However, you probably noticed that the cards aren't sorted by rank in the human player's hand and look kind of sloppy. Luckily, we can do something about that.

A Tiny Introduction to LINQ

LINQ, which stands for Language INtegrated Query, is a fantastic extension to C# that has had many books written about it. Fully 24 pages of Joseph and Ben Albahari's fantastic *C# 5.0 Pocket Reference*² are devoted to LINQ (wherein they only devote 4 pages to arrays). Most of LINQ is far beyond the scope of this book, but it's important that you know that it exists and what it can do.

LINQ has the capability to do database-like queries within a single line of C#, allowing you to select and order specific elements in an array. This is how we will sort the cards in the human player's hand. Add the following bolded lines to `Player.AddCard()`:

[Click here to view code image](#)

```
public class Player {  
    ...  
  
    public CardBartok AddCard(CardBartok eCB) {  
        if (hand == null) hand = new List<CardBartok>();  
  
        // Add the card to the hand  
        hand.Add (eCB);  
  
        // Sort the cards by rank using LINQ if this is a human  
        if (type == PlayerType.human) {  
            CardBartok[] cards = hand.ToArray(); // Copy hand to a new array  
  
            // Below is the LINQ call that works on the array of CardBartoks.  
            // It is similar to doing a foreach(CardBartok cd in cards)  
            // and sorting them by rank. It then returns a sorted array  
            cards = cards.OrderBy( cd => cd.rank ).ToArray();  
  
            // Convert the array CardBartok[] back to a List<CardBartok>  
            hand = new List<CardBartok>(cards);  
            // Note: LINQ operations can be a bit slow (like it could take a  
            // couple of milliseconds), but since we're only doing it once  
            // every turn, it isn't a problem.  
        }  
  
        FanHand();  
        return( eCB );  
    }  
  
    ...  
}
```

As you can see, in very few lines, we were able to sort the list. LINQ has tremendous capabilities that are beyond the scope of this book, but I highly recommend you look them up if you need to do sorting or other query-like operations on elements in an array (for example, if you had an array of people and needed to find all of them between the ages of 18 and 25).

Play the scene now, and you'll see that the cards in the human player's hand are always in order by rank.

The cards are going to need to animate into position for the game to be intelligible to the player, so it's time to make the cards move.

Making Cards Move!

Now comes the fun part where we make the cards actually interpolate from one position and rotation to the next. This will make the card game look much more like it's actually being played, and as you'll see, it makes it easier for the player to understand what is happening in the game.

A lot of the interpolation that we'll do here is based on that which was done for `FloatingScore` in `Prospector`. Just like `FloatingScore`, we'll start an interpolation that will be handled by the card itself, and when the card is done moving, it will send a callback message to notify the game that it's done.

Let's start by moving the cards smoothly into the players' hands. CardBartok already has a lot of the movement code written, so let's take advantage of it. Modify the following bolded code of the `Player.FanHand()` method:

[Click here to view code image](#)

```
public class Player {  
    ...  
    public void FanHand() {  
        ...  
        for (int i=0; i<hand.Count; i++) {  
            ...  
  
            // Set the localPosition and rotation of the ith card in the hand  
            hand[i].MoveTo(pos, rotQ); // Tell CardBartok to interpolate  
            hand[i].state = CBState.toHand;  
            // ^ After the move, CardBartok will set the state to CBState.hand  
  
            /* <= This "/*" begins a multiline comment // 1  
            hand[i].transform.localPosition = pos;  
            hand[i].transform.rotation = rotQ;  
            hand[i].state = CBState.hand;  
            */ // 1  
  
            ...  
        }  
    }  
}
```

1. The `/*` begins a multiline comment, so all lines of code between it and the following `*/` are considered to be commented out (and are ignored by C#). This is the same way that you could have commented out the `SlotDef` class in the `Layout` script at the beginning of this chapter.

Now, when you play the scene and press the number keys (1, 2, 3, 4), you will see the cards actually move into place! Because most of the heavy lifting is done by CardBartok, this took very little code to implement. This is one of the great advantages of object-oriented code. We trust that CardBartok knows how to move on its own so we can just call `MoveTo()` with a position and rotation, and CardBartok will do the rest.

Managing the Initial Card Deal

In the beginning of a round of Bartok, seven cards are dealt to each player, and then a single card is turned up from the `drawPile` to become the first target card. Add the following code to `Bartok` to make this happen:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {  
    ...  
    public float handFanDegrees = 10f;  
    public int numStartingCards = 7;  
    public float drawTimeStagger = 0.1f;  
    ...  
  
    void LayoutGame() {  
        ...  
        players[0].type = PlayerType.human; // Make the 0th player human  
  
        CardBartok tCB;  
        // Deal 7 cards to each player
```

```

for (int i=0; i<numStartingCards; i++) {
    for (int j=0; j<4; j++) { // There are always 4 players
        tCB = Draw (); // Draw a card
        // Stagger the draw time a bit. Remember order of operations.
        tCB.timeStart = Time.time + drawTimeStagger * ( i*4 + j );
        // ^ By setting the timeStart before calling AddCard, we
        // override the automatic setting of timeStart in
        // CardBartok.MoveTo().

        // Add the card to the player's hand. The modulus (%4)
        // results in a number from 0 to 3
        players[ (j+1)%4 ].AddCard(tCB);
    }
}

// Call Bartok.DrawFirstTarget() when the hand cards have been drawn.
Invoke("DrawFirstTarget", drawTimeStagger * (numStartingCards*4+4) );
}

public void DrawFirstTarget() {
    // Flip up the first target card from the drawPile
    CardBartok tCB = MoveToTarget( Draw () );
}

// This makes a new card the target
public CardBartok MoveToTarget(CardBartok tCB) {
    tCB.timeStart = 0;
    tCB.MoveTo(layout.discardPile.pos+Vector3.back);
    tCB.state = CBState.toTarget;
    tCB.faceUp = true;

    targetCard = tCB;

    return(tCB);
}
...
}

```

Upon playing the scene, you will see that the distribution of the seven cards and the draw of the first target happen properly on schedule; however, the human player's cards are overlapping each other in strange ways. Just as we did with Prospector, we need to very carefully manage both the `sortingLayerName` and the `sortingOrder` of each element of the cards.

Managing 2D Depth-Sorting Order

In addition to the standard issue of depth-sorting 2D objects, we now have to deal with the fact that the cards are moving, and there will be some times that we want them in one sort order at the beginning of the move and a different sort order when they arrive. To enable that, we will add fields for an `eventualSortLayer` and `eventualSortOrder` to `CardBartok`. This way, when a card is moving, it will switch to the `eventualSortLayer` and `eventualSortOrder` partway through the move.

The first thing you need to do is rename all of the sorting layers. Open the Tags & Layers manager by choosing *Edit > Project Settings > Tags & Layers* from the menu bar. Then set the names of Sorting Layers 1 through 10 to 1 through 10, as shown in [Figure 32.5](#).

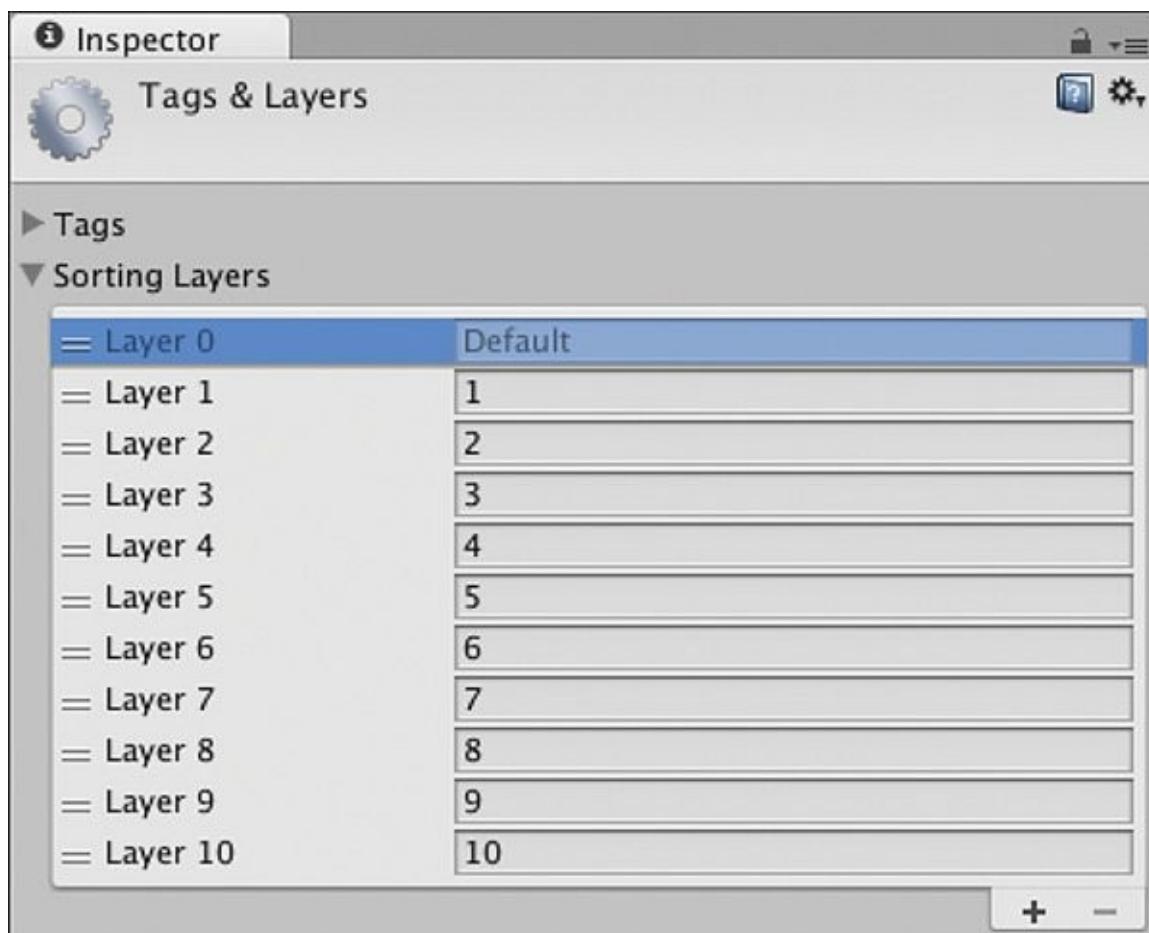


Figure 32.5 Simply named sorting layers for Bartok

Once this is done, add the following bolded code to CardBartok:

[Click here to view code image](#)

```

public class CardBartok : Card {
    ...
    public float           timeStart, timeDuration;
    public int           eventualSortOrder;
    public string        eventualSortLayer;
    ...

    void Update() {
        switch (state) {
            // All the to... states are ones where the card is interpolating
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
                ...
            } else { // 0<=u<1, which means that this is interpolating now
                ...
                transform.rotation = rotQ;

                if (u>0.5f && spriteRenderers[0].sortingOrder != eventualSortOrder) {
                    // Jump to the proper sort order
                    SetSortOrder(eventualSortOrder);
                }
                if (u>0.75f && spriteRenderers[0].sortingLayerName !=
eventualSortLayer) {
                    // Jump to the proper sort layer
                    SetSortingLayerName(eventualSortLayer);
                }
            }
        }
    }
}

```

```
        }  
    }  
    break;  
}
```

Now that the `eventualSortOrder` and `eventualSortLayer` fields exist, we need to use them throughout the code that has already been written. In Bartok, we'll make this change and also add a `MoveToDiscard()` function that moves the target card into the `discardPile`:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {
    ...
    public CardBartok MoveToTarget(CardBartok tCB) {
        tCB.timeStart = 0;
        tCB.MoveTo(layout.discardPile.pos+Vector3.back);
        tCB.state = CBState.toTarget;
        tCB.faceUp = true;
        tCB.SetSortingLayerName("10");//layout.target.layerName;
        tCB.eventualSortLayer = layout.target.layerName;
        if (targetCard != null) {
            MoveToDiscard(targetCard);
        }
        targetCard = tCB;

        return(tCB);
    }

    public CardBartok MoveToDiscard(CardBartok tCB) {
        tCB.state = CBState.discard;
        discardPile.Add ( tCB );
        tCB.SetSortingLayerName(layout.discardPile.layerName);
        tCB.SetSortOrder( discardPile.Count*4 );
        tCB.transform.localPosition = layout.discardPile.pos + Vector3.back/2;

        return(tCB);
    }
    ...
}
```

And there are a couple of changes to be made in Player as well:

[Click here to view code image](#)

```
public class Player {  
    ...  
    public CardBartok AddCard(CardBartok eCB) {  
        ...  
        // Sort the cards by rank using LINQ if this is a human  
        if (type == PlayerType.human) {  
            ...  
        }  
  
        eCB.SetSortingLayerName("10"); // This sorts the moving card to the top  
        eCB.eventualSortLayer = handSlotDef.layerName;  
  
        FanHand();  
        return( eCB );  
    }  
    ...  
}
```

```
public void FanHand() {  
    ...  
    for (int i=0; i<hand.Count; i++) {  
        ...  
  
        // Set the SortOrder of the cards so that they overlap properly  
        hand[i].eventualSortOrder = i*4;  
        //hand[i].SetSortOrder(i*4);  
    }  
}
```

Handling Turns

In this game, players will need to take turns, and the human player will have to know whose turn it is. We will accomplish this by highlighting the background behind the current player with a light.

In Unity, choose *GameObject > Create Other > Point Light* from the menu bar. Name the new light *TurnLight* and set its transform to the following:

TurnLight (Point Light) P:[0,0,-5] R:[0,0,0] S:[1,1,1]

As you can see, this casts a nice, obvious light on the background. We also need to add code to manage the light and the turns. Open the Bartok script and add the bolded code shown here:

[Click here to view code image](#)

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

// This enum contains the different phases of a game turn
public enum TurnPhase {
    idle,
    pre,
    waiting,
    post,
    gameOver
}

public class Bartok : MonoBehaviour {
    static public Bartok S;
    // This field is static to enforce that there is only 1 current player
    static public Player CURRENT_PLAYER;
    ...

    public CardBartok targetCard;

    public TurnPhase phase = TurnPhase.idle;
    public GameObject turnLight;

    void Awake() {
        S = this;

        // Find the TurnLight by name
        turnLight = GameObject.Find ("TurnLight");
    }

    ...

    public void DrawFirstTarget() {
        // Flip up the target card in the middle
        CardBartok tCB = MoveToTarget( Draw () );
        // Set the CardBartok to call CBCallback on this Bartok when it is
```

```

        tcb.reportFinishTo = this.gameObject;
    }

    // This callback is used by the last card to be dealt at the beginning
    // It is only used once per game.
    public void CBCallback(CardBartok cb) {
        // You sometimes want to have reporting of method calls like this // 1
        Utils.tr(Utils.RoundToPlaces(Time.time), "Bartok.CBCallback()", cb.name);

        StartGame(); // Start the Game
    }

    public void StartGame() {
        // Pick the player to the left of the human to go first.
        // (players[0] is the human)
        PassTurn(1);
    }

    public void PassTurn(int num=-1) {
        // If no number was passed in, pick the next player
        if (num == -1) {
            int ndx = players.IndexOf(CURRENT_PLAYER);
            num = (ndx+1)%4;
        }
        int lastPlayerNum = -1;
        if (CURRENT_PLAYER != null) {
            lastPlayerNum = CURRENT_PLAYER.playerNum;
        }
        CURRENT_PLAYER = players[num];
        phase = TurnPhase.pre;

        CURRENT_PLAYER.TakeTurn();

        // Move the TurnLight to shine on the new CURRENT_PLAYER
        Vector3 lPos = CURRENT_PLAYER.handSlotDef.pos + Vector3.back*5;
        turnLight.transform.position = lPos;

        // Report the turn passing
        Utils.tr(Utils.RoundToPlaces(Time.time), "Bartok.PassTurn()", 
"Old: "+lastPlayerNum,"New: "+CURRENT_PLAYER.playerNum);
    }

    // ValidPlay verifies that the card chosen can be played on the discard pile
    public bool ValidPlay(CardBartok cb) {
        // It's a valid play if the rank is the same
        if (cb.rank == targetCard.rank) return(true);

        // It's a valid play if the suit is the same
        if (cb.suit == targetCard.suit) {
            return(true);
        }

        // Otherwise, return false
        return(false);
    }

    ...
}

/* Now is a good time to comment out this testing code // 2
// This Update method is used to test passing cards to players
void Update() {
    if (Input.GetKeyDown(KeyCode.Alpha1)) {
        players[0].AddCard(Draw ());
    }
}

```

```

        if (Input.GetKeyDown(KeyCode.Alpha2)) {
            players[1].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha3)) {
            players[2].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha4)) {
            players[3].AddCard(Draw ());
        }
    }
}
*/ // 2
}

```

1. The line following the // 1 is the first use of the static public `Utils.tr()` method (`tr` is short for “trace”, another term for outputting to the console). This method takes any number of arguments (via the `params` keyword), concatenates them, and outputs them to the Console pane. It is one of the elements that was added to the `Utils` class in the version contained in the `unitypackage` that you imported into `Prospector`.

2. Make sure that you add both the opening and closing lines of this multiline comment.

Press Play, and you will see the scene deal out the cards and then move the `TurnLight` to hover over the left player, signifying that it is that player’s turn. Now, let’s make the AI players able to take turns. Open the `Player` script and add the bolded code:

[Click here to view code image](#)

```

public class Player {
    ...

    public void FanHand() {
        ...
        for (int i=0; i<hand.Count; i++) {
            ...
            pos.z = -0.5f*i;

            // The line below makes sure that the card starts moving immediately
            // if it's not the initial deal at the beginning of the game.
            if (Bartok.S.phase != TurnPhase.idle) {
                hand[i].timeStart = 0;
            }

            // Set the localPosition and rotation of the ith card in the hand
            ...
        }
    }

    // The TakeTurn() function enables the AI of the computer Players
    public void TakeTurn() {
        Utils.tr (Utils.RoundToPlaces(Time.time), "Player.TakeTurn");

        // Don't need to do anything if this is the human player.
        if (type == PlayerType.human) return;

        Bartok.S.phase = TurnPhase.waiting;

        CardBartok cb;

        // If this is an AI player, need to make a choice about what to play
        // Find valid plays
        List<CardBartok> validCards = new List<CardBartok>();
        foreach (CardBartok tcb in hand) {

```

```

        if (Bartok.S.ValidPlay(tCB)) {
            validCards.Add ( tCB );
        }
    }
    // If there are no valid cards
    if (validCards.Count == 0) {
        // ...then draw a card
        cb = AddCard( Bartok.S.Draw () );
        cb.callbackPlayer = this;
        return;
    }

    // Otherwise, if there is a card or more to play, pick one
    cb = validCards[ Random.Range (0,validCards.Count) ];
    RemoveCard(cb);
    Bartok.S.MoveToTarget(cb);
    cb.callbackPlayer = this;

}

public void CBCallback(CardBartok tCB) {
    Utils.tr (Utils.RoundToPlaces(Time.time),
    "Player.CBCallback()",tCB.name,"Player "+playerNum);
    // The card is done moving, so pass the turn
    Bartok.S.PassTurn();
}
}
}

```

The last method you just added is a `CBCallback` function that a `CardBartok` should call when it's done moving; however, because `Player` does not extend `MonoBehaviour`, we need to use a method other than `SendMessage()` to do so. Instead, we'll pass the `CardBartok` a reference to this `Player`, and then the `CardBartok` can call `CBCallback` directly on the `Player` instance when it's done moving. This `Player` reference will be stored on `CardBartok` as the field `callbackPlayer`. Open `CardBartok` and add this code:

[Click here to view code image](#)

```

public class CardBartok : Card {
    ...
    // When the card is done moving, it will call reportFinishTo.SendMessage()
    public GameObject           reportFinishTo = null;
    public Player                callbackPlayer = null;

    void Awake() {
        callbackPlayer = null; // Just to be sure.
    }

    // MoveTo tells the card to interpolate to a new position and rotation
    ...

    void Update() {
        switch (state) {
            // All the to... states are ones where the card is interpolating
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
            ...
            } else if (u>=1) { // If u>=1, we're finished moving
                uC = 1; // Set uC=1 so we don't overshoot
                ...
            }
        }
    }
}

```

```
        if (reportFinishTo != null) { //If there's a callback GameObject
            // ... then use SendMessage to call the CBCCallback method
            // with this as the parameter.
            reportFinishTo.SendMessage("CBCCallback", this);
            // After calling SendMessage(), reportFinishTo must be set
            // to null so that it the card doesn't continue to report
            // to the same GameObject every subsequent time it moves.
            reportFinishTo = null;
        } else if (callbackPlayer != null) {
            // If there's a callback Player
            // then call CBCCallback directly on the Player
            callbackPlayer.CBCCallback(this);
            callbackPlayer = null;
        } else { // If there is nothing to callback
            // Just let it stay still.
        }
    } else {
        ...
    }
    break;
}
}
```

Now, you'll see that when you play the scene, the other three players each play. It's time to make the human able to play too by making the cards clickable.

Add the `OnMouseUpAsButton()` method to the end of `CardBartok`:

[Click here to view code image](#)

```
public class CardBartok : Card {
    ...
    void update() {...}

    // This allows the card to react to being clicked
    override public void OnMouseUpAsButton() {
        // Call the CardClicked method on the Bartok singleton
        Bartok.S.CardClicked(this);
        // Also call the base class (Card.cs) version of this method
        base.OnMouseUpAsButton();
    }
}
```

And now add the `CardClicked()` method to the end of the Bartok script:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {
    ...
    public void CardClicked(CardBartok tCB) {
        // If it's not the human's turn, don't respond
        if (CURRENT_PLAYER.type != PlayerType.human) return;
        // If the game is waiting on a card to move, don't respond
        if (phase == TurnPhase.waiting) return;

        // Act differently based on whether it was a card in hand
        // or on the drawPile that was clicked
        switch (tCB.state) {
            case CBState.drawpile:
                // Draw the top card, not necessarily the one clicked.
                CardBartok cb = CURRENT_PLAYER.AddCard( Draw() );
                cb.callbackPlayer = CURRENT_PLAYER;
```

```
        Utils.tr (Utils.RoundToPlaces(Time.time),
"Bartok.CardClicked()", "Draw", cb.name);
        phase = TurnPhase.waiting;
        break;
    case CBState.hand:
        // Check to see whether the card is valid
        if (ValidPlay(tCB)) {
            CURRENT_PLAYER.RemoveCard(tCB);
            MoveToTarget(tCB);
            tCB.callbackPlayer = CURRENT_PLAYER;
            Utils.tr(Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()", "Play", tCB.name, targetCard.name+ " is target");
            phase = TurnPhase.waiting;
        } else {
            // Just ignore it
            Utils.tr(Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()", "Attempted to Play", tCB.name, targetCard.name+ " is target");
        }
        break;
    }
}
```

Now, you can play as well, and the game works fine. But right now there is no logic to end the game when it's over. Just a few more additions, and this prototype will be playable!

Adding Game Logic

Just as with Prospector, we want to message the player when she finishes the game. Create two new GUITexts and name them *GTGameOver* and *GTRoundResult*. Their settings should be those shown in [Figure 32.6](#).

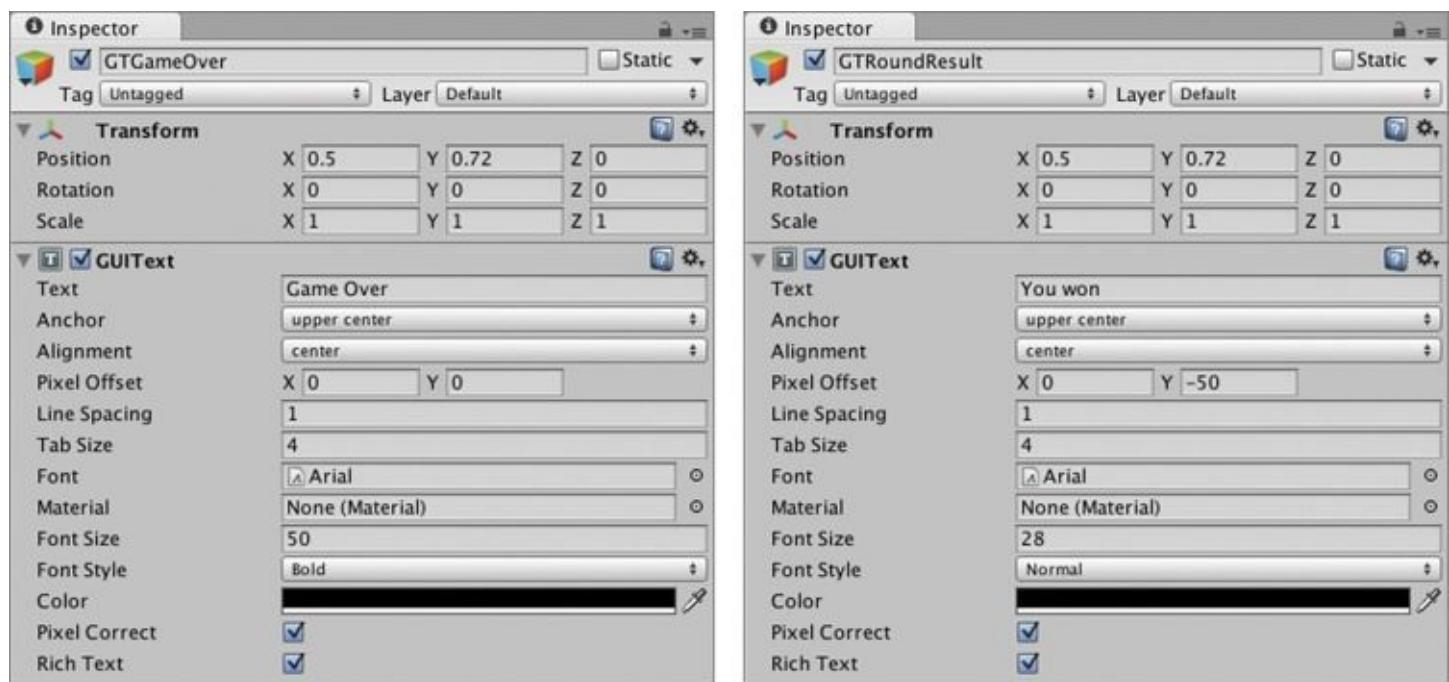


Figure 32.6 Settings for GTGameOver and GTRoundResult

Once you have these, add the following code to Bartok to both manage these GUITexts and to test whether the game is over and restart it after one second if it is:

[Click here to view code image](#)

```
public class Bartok : MonoBehaviour {  
    ...  
}
```

```

public GameObject turnLight;

public GameObject GTGameOver;
public GameObject GTRoundResult;

void Awake() {
    S = this;

    // Find the TurnLight by name
    turnLight = GameObject.Find ("TurnLight");
    GTGameOver = GameObject.Find("GTGameOver");
    GTRoundResult = GameObject.Find("GTRoundResult");
    GTGameOver.SetActive(false);
    GTRoundResult.SetActive(false);
}

...
}

public void PassTurn(int num=-1) {
    ...
    if (CURRENT_PLAYER != null) {
        lastPlayerNum = CURRENT_PLAYER.playerNum;
        // Check for Game Over and need to reshuffle discards
        if (CheckGameOver() ) {
            return;
        }
    }
    ...
}

...
}

public bool CheckGameOver() {
    // See if we need to reshuffle the discard pile into the draw pile
    if (drawPile.Count == 0) {
        List<Card> cards = new List<Card>();
        foreach (CardBartok cb in discardPile) {
            cards.Add (cb);
        }
        discardPile.Clear();
        Deck.Shuffle( ref cards );
        drawPile = UpgradeCardsList(cards);
        ArrangeDrawPile();
    }

    // Check to see if the current player has won
    if (CURRENT_PLAYER.hand.Count == 0) {
        // The current player has won!
        if (CURRENT_PLAYER.type == PlayerType.human) {
            GTGameOver.guiText.text = "You Won!";
            GTRoundResult.guiText.text = "";
        } else {
            GTGameOver.guiText.text = "Game Over";
            GTRoundResult.guiText.text = "Player "+CURRENT_PLAYER.playerNum + " won";
        }
        GTGameOver.SetActive(true);
        GTRoundResult.SetActive(true);
        phase = TurnPhase.gameOver;
        Invoke("RestartGame", 1);
        return(true);
    }

    return(false);
}

```

```
public void RestartGame() {
    CURRENT_PLAYER = null;
    Application.LoadLevel("__Bartok_Scene_0");
}

}
```

Now the game will play properly, it will end when it's over, and it will restart properly as well.

Summary

The goal of this chapter was to demonstrate how possible it is to take the digital prototypes that you make in this book and adapt them to your own games. Once you finish all the tutorial chapters, you will have the framework for a classic arcade game (*Apple Picker*), a physics-based casual game (*Mission Demolition*), a space shooter (*Space SHMUP*), a card game (*Prospector and Bartok*), a word game (*Word Game*), a first-person shooter (*Quick Snap*), and a third-person adventure game (*Omega Mage*). As prototypes, none of these are finished games, but any of them could serve as a foundation on which to build your own games.

Next Steps

The classic paper version of the Bartok card game included the ability for the winner of any round to add additional rules to the game. While it's not possible to allow the player to just make up rules for this digital game, it is certainly possible to add your own optional rules through code just as I did for the version you played with in [Chapter 1](#).

If you visit the <http://book.prototools.net> website, you can look under [Chapter 32](#) for the Unity project of the expanded version of Bartok that includes all the optional rules you were able to play with in [Chapter 1](#). That should be a good starting point for you to use to add your own rules to the game.

Chapter 33. Prototype 6: Word Game

In this chapter, you learn how to create a simple word game. This game uses several concepts that you have already learned, and it introduces the concept of *coroutines*, methods that can yield during execution to allow the processor to handle other methods.

By the end of this chapter, you'll have a simple word game that you can expand yourself.

Getting Started: Word Game Prototype

As usual, you'll import a unitypackage to start this chapter. This package contains a few art assets and some C# Scripts that you created in previous chapters.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup Procedure.”](#) When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 3D for this project.

For this project, we will import the main scene from the unitypackage, so you do not need to set up the `_MainCamera`.

- **Project name:** Word Game
- **Download and import package:** See [Chapter 33](#) at <http://book.prototools.net>
- **Scene name:** `_WordGame_Scene_0` (imported in unitypackage)
- **Project folders:** `_Scripts`, `_Prefabs`, Materials & Textures, Resources
- **C# script names:** Just the imported scripts in the ProtoTools folder

Open the scene `_WordGame_Scene_0`, and you will find a `_MainCamera` that is already set up for an orthographic game. You'll also notice that some of the reusable C# scripts that were created in previous chapters have been moved into a ProtoTools folder to keep them separate from the new scripts you'll create for this project. I find this is useful because it enables me to just place a copy of the ProtoTools folder into the `_Scripts` folder of any new project and have all that functionality ready to go.

In your Build Settings, make sure that this one is set to *PC, Mac, & Linux Standalone*. Set the aspect ratio of the Game pane to *Standalone (1024 × 768)*.

About the Word Game

This game is a classic form of word game. Commercial examples of this game include *Word Whomp* by [Pogo.com](#), *Jumbline 2* by Branium, *Pressed for Words* by Words and Maps, and many others. The player will be presented with six letters that spell at least one six-letter word, and she is tasked with finding all of the words that can be created with those six letters. Our version of the game will include some slick animations (using Bézier interpolations) and a scoring paradigm that encourages the player to find long words before short ones. [Figure 33.1](#) shows an image of the game you'll create in this chapter. As you can see in the image, this game can handle words up to 8 letters in length, though 6 is the standard for this kind of game.



Figure 33.1 An image of the Word Game created in this chapter

In this image, you can see that each of the words are divided into individual letters, and there are two sizes of letters. For the sake of object orientation, we'll create a `Letter` class that handles each letter and a `Word` class to collect them into words. We'll also create a `WordList` class to read the large dictionary of possible words that we have and turn it into usable data for the game. The game will be controlled by a `WordGame` class, and the `Scoreboard` and `FloatingScore` classes from previous prototypes will be used to show the score to the player. In addition, the `Utils` class will be used for interpolation and easing. The `PT_XMLReader` class is imported with this project, but is unused. I left this script in the `unitypackage` because I want to encourage you to start building your own collection of useful scripts that you can import into any project to help you get started (just as the `ProtoTools` folder is for the projects in this book). Feel free to add any useful scripts that you create to this collection, and think about importing it as the first thing you do for each new game prototype that you start.

Parsing the Word List

This game uses a modified form of the 2of12inf word list created by Alan Beale.¹ I've removed some offensive words and attempted to correct others. You are more than welcome to use this word list however you wish in the future, as long as you follow the copyright wishes of both Alan Beale and Kevin Atkinson (as listed in the footnote). I also modified the list by shifting all of the letters to uppercase and by changing the line ending from `\r\n` (a carriage return and a line feed, which is the standard Windows text file format) to `\n` (just a line feed, the standard Macintosh text format). This was done because it makes it easier to split the file into individual words based on line feed, and it will work on Windows just as well as Mac.

¹ Alan Beale has released all of his word lists into the public domain apart from the aspects of the 2of12inf list that were based on the AGID word list, Copyright 2000 by Kevin Atkinson. Permission to use, copy, modify, distribute and sell this [the AGID] database, the associated scripts, the output created from the scripts and its documentation for any purpose is hereby granted [by Atkinson] without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Kevin Atkinson makes no representations about the suitability of this array for any purpose. It is provided "as is" without express or implied warranty.

The decision to remove offensive words was based on the kind of game this is. In a game like *Scrabble* or *Letterpress*, the player is given a series of letter tiles, and she is able to choose which words she wishes to spell with those tiles. If this game were of that ilk, I would not have removed any words from the word list. However, in this game, the player is forced to spell every word in the list that can be made from the collection of letters that she is given. This means that the game could force players to spell some terms that would be very offensive to them. In this game, the decision of which words are chosen has shifted from the player to the computer, and I did not feel comfortable forcing players to spell offensive words. However, in the over 75,000 words in the list, there are probably some words that I missed, so if you find any words in the game that you feel I should omit (or ones I should add), please let me know by sending me a message via the website <http://book.prototools.net>. Thanks.

To read the word list file, we need to pull its text into an array of strings and split it by `\n`. The following code listing includes the first instance of a *Coroutine* in this book. Coroutines are functions that can execute progressively with yields in between to allow other functions to run. You can learn more about them in [Appendix B, "Useful Concepts."](#) Create a new C# script named *WordList* and enter the following code:

[Click here to view code image](#)

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

public class WordList : MonoBehaviour {
    public static WordList S;

    public TextAsset wordListText;
    public int numToParseBeforeYield = 10000;
    public int wordLengthMin = 3;
    public int wordLengthMax = 7;

    public bool _____;

    public int currLine = 0;
    public int totalLines;
    public int longWordCount;
    public int wordCount;
```

```

// Some variables below are private to keep them from appearing in the
// Inspector. Because these variables will be so long, it can drastically
// slow playback if the Inspector is trying to display them. Private
// variables are restricted so that only this instance of the WordList
// class can see them.
private string[]          lines;                                // 1
private List<string>      longWords;
private List<string>      words;

void Awake() {
    S = this; // Set up the singleton
}

void Start () {
    // Split the text of wordListText on line feeds, which creates a large,
    // populated string[] with all the words from the list
    lines = wordListText.text.Split('\n');
    totalLines = lines.Length;

    // This starts the coroutine ParseLines(). Coroutines can be paused in
    // the middle to allow other code to execute.
    StartCoroutine( ParseLines() );                                // 2
}

// All coroutines have IEnumerator as their return type.
public IEnumerator ParseLines() {                                // 3
    string word;
    // Init the Lists to hold the longest words and all valid words
    longWords = new List<string>();
    words = new List<string>();

    for (currLine = 0; currLine < totalLines; currLine++) {
        word = lines[currLine];

        // If the word is as long as wordLengthMax
        if (word.Length == wordLengthMax) {
            // ...then store it in longWords
            longWords.Add(word);
        }
        // If it's between wordLengthMin and wordLengthMax in length
        if ( word.Length>=wordLengthMin && word.Length<=wordLengthMax ) {
            // ...then add it to the list of all valid words
            words.Add(word);
        }
    }

    // Determine whether the coroutine should yield
    // This uses a modulus (%) function to yield every 10,000th record
    // (or whatever you have numToParseBeforeYield set to)
    if (currLine % numToParseBeforeYield == 0) {
        // Count the words in each list to show that the parsing is
        // progressing
        longWordCount = longWords.Count;
        wordCount = words.Count;
        // This yields execution until the next frame
        yield return null;                                         // 4
    }

    // The yield will cause the execution of this method to wait
    // here while other code executes and then continue from this
    // point.
}
}

```

```

// These methods allow other classes to access the private List<string>s
public List<string> GetWords() {
    return(words);
}

public string GetWord(int ndx) {
    return( words[ndx] );
}

public List<string> GetLongWords() {
    return( longWords );
}

public string GetLongWord(int ndx) {
    return( longWords[ndx] );
}
}

```

1. Another way to hide variables from the Inspector without making them private is to use the *attribute* `[System.NonSerialized]`. An attribute sends an instruction to Unity about how to treat the next line in the code. If you were to make a variable public and then place `[System.NonSerialized]` on the line preceding it, then the public variable would not appear in the Inspector. Private variables were used in this code to show how methods like `GetWord()` can be used to access them, but if all you really need to do is hide a variable from the Inspector, `[System.NonSerialized]` may be a better way to go.
2. This starts the *Coroutine* `ParseLines()`. Coroutines can be *yielded* in the middle to allow other code to execute. When a coroutine yields, it transfers execution back to other code and then continues the coroutine after a certain amount of time has passed. So, if a coroutine had a yield in the middle of an infinite `while` loop, other code could still execute even though the loop never exited.
3. All coroutines have `IEnumerator` as their return type. This enables them to yield their execution and allow other methods to run before returning to the coroutine. This is extremely important for processes like loading large files or like parsing a large amount of data (as we're doing in this case).
4. This is the yield statement. When a coroutine yields, it effectively pauses on the `yield` line until a certain amount of time has passed and then continues from there. The coroutine and yield statements are necessary in this code because the `for` loop will be iterating more than 75,000 times to interpret all 75,000+ lines of `WordList`. On a slower computer, this could make it appear that the program had frozen, so the coroutine allows us to keep Unity updating and interactive while in the middle of a time-consuming process.

In this code, we want to have the coroutine yield for as little time as possible (a single frame), so the yield statement returns null. It is possible to have coroutines yield for a specific amount of time by entering code like `yield return new WaitForSeconds(5);`, which would cause the coroutine to yield for about 5 seconds (coroutine yield times are not exact). See the [“Using Coroutines”](#) sidebar for more information.

Once the code is written and saved, switch back to Unity and attach the `WordList` C# script to `_MainCamera`. Then, select `_MainCamera` in the Hierarchy and set the `wordListText` variable of the `WordList (Script)` component in the Inspector to be the file `2of12inf`, which you can find in the Resources folder of the Project pane. Once this is set, press Play. You will see that the `currLine`,

`longWordCount`, and `wordCount` will count up progressively. This is happening because the numbers are allowed to update every time the coroutine `ParseLines()` yields.

If you use the Inspector to change `numToParseBeforeYield` to 100, you will see that these numbers build much more slowly because the coroutine is yielding every 100 words. However, if you change it to something like 100000, these numbers will update only once because there are fewer than 100 thousand words in the word list. If you’re interested in seeing how much time each pass through the `ParseLines()` coroutine is taking, try using the profiler, as described in the sidebar titled “[The Unity Profiler](#).”

Using Coroutines

While the coroutine in this chapter probably isn’t strictly necessary as long as you have a fast computer, this kind of thing becomes much more important when you’re developing for mobile devices (or other devices with slower processors). Parsing this same word list on an older iPhone can take as much as 10 to 20 seconds, so it’s important to include breaks in the parsing where the app can handle other tasks and not appear frozen.

An important thing to note about the required `StartCoroutine()` method is that it can only be called within a class that extends `MonoBehaviour`.

You can learn more about coroutines in the Unity documentation or [Appendix B](#).

The Unity Profiler

The Unity profiler is one of the most powerful tools for optimizing the performance of your games, though sadly, it’s only available in Unity Pro. For every frame of your game, the profiler maintains stats on the amount of time spent on each C# function, calls to the graphics engine, handling user input, and so on. You can see a great example of how this works by running the profiler on this project.

First, make sure that the `WordList` code from the preceding pages is working properly. Next, we’ll add a Profiler pane to the same group as the Scene pane. That will ensure that you can see both the Game pane and the Profiler pane simultaneously. To add the Profiler pane, click the pop-up menu button at the top right of the current Scene pane and choose *Add Tab > Profiler* (as shown in [Figure 33.2](#)).

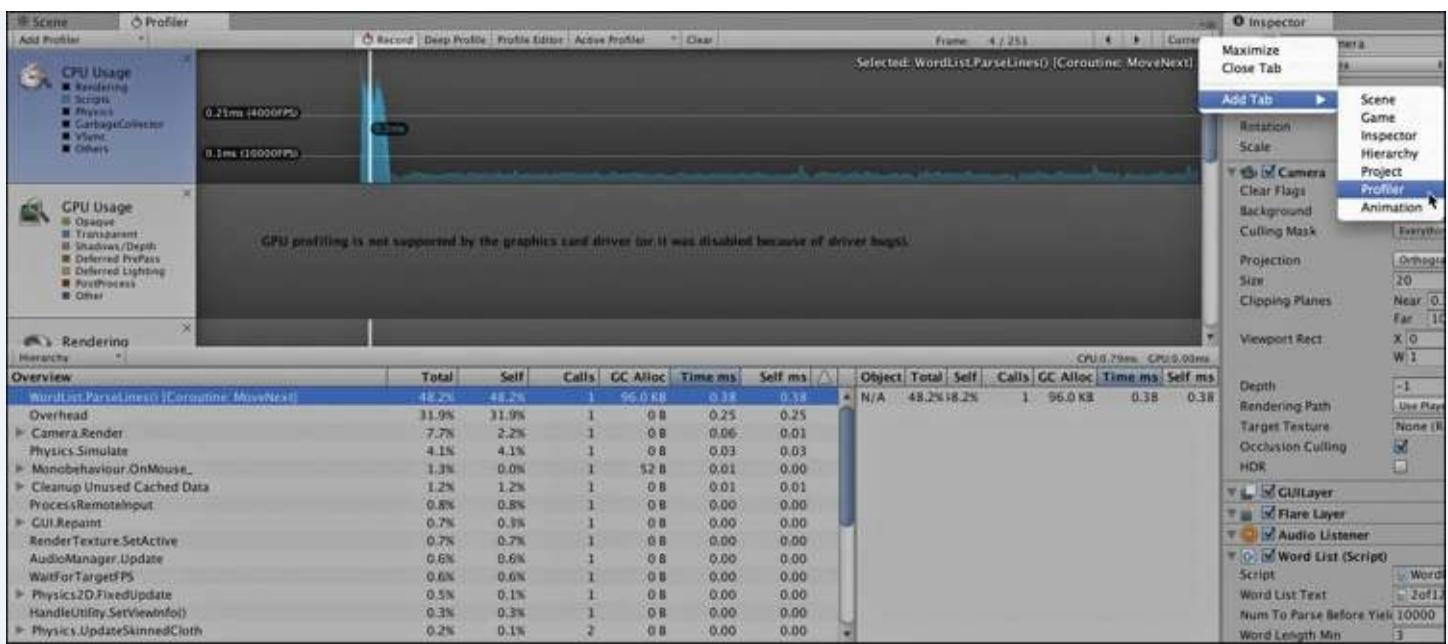


Figure 33.2 The Profiler pane

To see the profiler in action, first click the Pause button at the top of the Unity window and then click Play. This will cause Unity to prepare itself to run your game but to be paused before the first frame. If you click Pause again, you will see a graph start to appear in the profiler. Pause the game again before the graph completely reaches the left side of the screen.

With the game paused, the profiler should stop graphing yet maintain the graph of the frames that have already passed. Each of the colors in the graph next to CPU Usage covers a different aspect of things for which the CPU (the main processor in your computer) is used. In the later frames, if you’re on a fast computer, you should see that most of the chart is yellow; the yellow represents the time Unity spends on VSync (that is, waiting for the screen to be ready to display another frame). This is blocking our view of how much time is taken by the scripts (which are light blue), so we’ll hide it from the graph. The little colored boxes below CPU Usage on the left side of the profiler each represent a different kind of process that runs on the CPU. You want to turn all of them off except for the Scripts box (which is blue). To do this, click the colored box next to everything except for Scripts. This should leave you with a blue graph like the one shown in [Figure 33.2](#).

Now, click and drag the mouse along the blue graph, and you should see a white line following the mouse. This white line represents a single frame in the graph. As you move, the text in the bottom half of the profiler will update to show how much processing time was taken by each function or background process during that frame. The function we’re interested in is the `WordList.ParseLines()` coroutine. This only runs in the first few frames, so you won’t see it on the right side of the graph; however, you should see a spike of script activity at the beginning of the graph (as shown in [Figure 33.2](#)), which is the time taken by the `ParseLines()` coroutine. Move the white line to that part of the graph and click `WordList.ParseLines()` in the Overview column. This will highlight the graph contribution of that one routine and dim the others. If you use the left and right arrows at the top-right corner of the Profiler pane, you can step one frame back or forward (respectively) and see the CPU resources used by `ParseLines()` in each frame. In my profiling, I found that for the first several frames, `ParseLines()` took up nearly 50% of the CPU time spent on each frame (although your numbers may vary due to computer

type and processing speed).

In addition to script profiling, the profiler can also help you find what aspects of rendering or physics simulation are taking the most time in your game. If you ever run into frame rate issues in one of your games, try checking the profiler to see what's happening. (You'll want to be sure to turn all of the other types of CPU profiling back on when you do [that is, re-check all the boxes that we unchecked to isolate scripts]).

To see a very different profiler graph, you can try running the profiler on the Hello World project from [Chapter 18, “Hello World: Your First Program.”](#) You'll see that in Hello World, much more time is spent on physics than scripts. (You may need to turn the VSync element of the graph off again to see this clearly.)

You can learn more about the profiler in the Unity documentation.

Setting Up the Game

We're going to create a WordGame class to manage the game, but before we do so, we need to make a couple of changes to WordList. First, we need to make it not start parsing the words on `Start()` but instead wait until an `Init()` method is called by another class. Second, we need to make WordList notify the upcoming WordGame script when the parsing is complete. To do this, we will have the WordList send a message to the `_MainCamera` GameObject using the `SendMessage()` command. This message will be interpreted by WordGame as you'll soon see. Change the name of the `void Start()` method in WordList to `public void Init()` and add the following bold code to the end of the `ParseLines` method in WordList:

[Click here to view code image](#)

```
public class WordList : MonoBehaviour {
    ...
    void Awake() {
        S = this; // Set up the singleton
    }
    public void Init() { // This line replaces "void Start()"
        // Split the text of wordListText on line feeds, which creates a large,
        // populated string[] with all the words from the list
        lines = wordListText.text.Split('\n');
        totalLines = lines.Length;

        // This starts a coroutine of ParseLines. Coroutines can be paused in
        // the middle to allow other code to execute.
        StartCoroutine( ParseLines() );
    }
    ...
    public IEnumerator ParseLines() {
        ...
        for (currLine = 0; currLine < totalLines; currLine++) {
            ...
        }
        // Send a message to this gameObject to let it know the parse is done
        gameObject.SendMessage("WordListParseComplete");
    }
}
```

The `SendMessage()` command is executed on the `GameObject _MainCamera` (because `WordList` is a Script Component of `_MainCamera`). This command will call a `WordListParseComplete()` method on any script that is attached to the `GameObject` on which it is called (that is, `_MainCamera`). Now, create a `WordGame` C# script and attach it to `_MainCamera` as a Script component. Then enter the following code to take advantage of the changes just made to `WordList`:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;    // We'll be using List<> & Dictionary<>
using System.Linq;                // We'll be using LINQ

public enum GameMode {
    preGame,      // Before the game starts
    loading,      // The word list is loading and being parsed
    makeLevel,    // The individual WordLevel is being created
    levelPrep,    // The level visuals are Instantiated
    inLevel       // The level is in progress
}

public class WordGame : MonoBehaviour {
    static public WordGame S; // Singleton

    public bool _____;

    public GameMode mode = GameMode.preGame;

    void Awake() {
        S = this; // Assign the singleton
    }

    void Start () {
        mode = GameMode.loading;
        // Tells WordList.S to start parsing all the words
        WordList.S.Init();
    }

    // Called by the SendMessage() command from WordList
    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
    }
}
```

Select `_MainCamera` in the Hierarchy pane, and look at the `WordGame (Script)` component in the Inspector. When you press Play, you'll see the value of the `mode` field initially move from `preGame` to `loading`. Then, after all the words have been parsed, it will change from `loading` to `makeLevel`. This shows us that everything is working as we had hoped.

Now, it's time to take the words in the `WordList` and make a level from them. The `Level` class will include the following:

- The long word on which the level is based. (If `maxWordLength` is 6, this is the six-letter word whose letters will be reshuffled into the other words.)
- The index number of that word in `longWords`.
- The level number as the `int levelNum`. In this chapter, every time the game starts, we'll choose a random word, but later you could use a pseudo-random function to ensure that the eighth level

would always be the same word.

- A Dictionary<,> of each character in the word and how many times it is used. Dictionaries are part of System.Collections.Generic along with Lists.
- A List<> of all the other words that can be formed from the characters in the Dictionary above.

A Dictionary<,> is a generic collection type that holds a series of key value pairs. In each level, the Dictionary<,> will use char keys and int values to hold information about how many times each char is used in the long word. For example, this is how the long word MISSISSIPPI would look:

[Click here to view code image](#)

```
Dictionary<char,int> charDict = new Dictionary<char,int>();  
charDict.Add('M',1); // MISSISSIPPI has 1 M  
charDict.Add('I',4); // MISSISSIPPI has 4 Is  
charDict.Add('S',4); // MISSISSIPPI has 4 Ss  
charDict.Add('P',2); // MISSISSIPPI has 2 Ps
```

WordLevel will also contain two useful static methods:

- MakeCharDict (): Creates a charDict like the one in the preceding code from any string
- CheckWordInLevel (): Checks to see whether a word can be spelled using the chars in a WordLevel's charDict

Create a new C# script named *WordLevel* and enter the following code. Note that WordLevel does not extend MonoBehaviour, so it is not a class that can be attached to a GameObject as a Script component, and it cannot have StartCoroutine () called within it.

[Click here to view code image](#)

```
using UnityEngine;  
using System.Collections;  
using System.Collections.Generic;  
  
[System.Serializable] // WordLevels can be viewed in the Inspector  
public class WordLevel { // WordLevel does NOT extend MonoBehaviour  
    public int                  levelNum;  
    public int                  longWordIndex;  
    public string                word;  
    // A Dictionary<,> of all the letters in word  
    public Dictionary<char,int>  charDict;  
    // All the words that can be spelled with the letters in charDict  
    public List<string>          subWords;  
  
    // A static function that counts the instances of chars in a string and  
    // returns a Dictionary<char,int> that contains this information  
    static public Dictionary<char,int> MakeCharDict(string w) {  
        Dictionary<char,int> dict = new Dictionary<char,int>();  
        char c;  
        for (int i=0; i<w.Length; i++) {  
            c = w[i];  
            if (dict.ContainsKey(c)) {  
                dict[c]++;  
            } else {  
                dict.Add (c,1);  
            }  
        }  
        return(dict);  
    }  
}
```

```

// This static method checks to see whether the word can be spelled with the
// chars in level.charDict
public static bool CheckWordInLevel(string str, WordLevel level) {
    Dictionary<char,int> counts = new Dictionary<char, int>();
    for (int i=0; i<str.Length; i++) {
        char c = str[i];
        // If the charDict contains char c
        if (level.charDict.ContainsKey(c)) {
            // If counts doesn't already have char c as a key
            if (!counts.ContainsKey(c)) {
                // ...then add a new key with a value of 1
                counts.Add (c,1);
            } else {
                // Otherwise, add 1 to the current value
                counts[c]++;
            }
            // If this means that there are more instances of char c in str
            // than are available in level.charDict
            if (counts[c] > level.charDict[c]) {
                // ... then return false
                return(false);
            }
        } else {
            // The char c isn't in level.word, so return false
            return(false);
        }
    }
    return(true);
}
}

```

Now, to make use of this, make the following bolded changes to WordGame:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...

    public GameMode mode = GameMode.preGame;
    public WordLevel currLevel;

    ...

    // Called by the SendMessage() command from WordList
    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
        // Make a level and assign it to currLevel, the current WordLevel
        currLevel = MakeWordLevel();
    }

    // With the default value of -1, this method will generate a level from
    // a random word.
    public WordLevel MakeWordLevel(int levelNum = -1) {
        WordLevel level = new WordLevel();
        if (levelNum == -1) {
            // Pick a random level
            level.longWordIndex = Random.Range(0,WordList.S.longWordCount);
        } else {
            // This can be added later
        }
        level.levelNum = levelNum;
        level.word = WordList.S.GetLongWord(level.longWordIndex);
        level.charDict = WordLevel.MakeCharDict(level.word);
    }
}

```

```

// Call a coroutine to check all the words in the WordList and see
// whether each word can be spelled by the chars in level.charDict
StartCoroutine( FindSubWordsCoroutine(level) );

// This returns the level before the coroutine finishes, so
// SubWordSearchComplete() is called when the coroutine is done
return( level );
}

// A coroutine that finds words that can be spelled in this level
public IEnumerator FindSubWordsCoroutine(WordLevel level) {
    level.subWords = new List<string>();
    string str;

    List<string> words = WordList.S.GetWords();
    // ^ This is very fast because List<string> is passed by reference

    // Iterate through all the words in the WordList
    for (int i=0; i<WordList.S.wordCount; i++) {
        str = words[i];
        // Check whether each one can be spelled using level.charDict
        if (WordLevel.CheckWordInLevel(str, level)) {
            level.subWords.Add(str);
        }
        // Yield if we've parsed a lot of words this frame
        if (i%WordList.S.numToParseBeforeYield == 0) {
            // yield until the next frame
            yield return null;
        }
    }
}

// List<string>.Sort() sorts alphabetically by default
level.subWords.Sort ();
// Now sort by length to have words grouped by number of letters
level.subWords = SortWordsByLength(level.subWords).ToList();

// The coroutine is complete, so call SubWordSearchComplete()
SubWordSearchComplete();
}

public static IEnumerable<string> SortWordsByLength(IEnumerable<string> e)
{
    // Use LINQ to sort the array received and return a copy
    // The LINQ syntax is different from regular C# and is beyond
    // the scope of this book
    var sorted = from s in e
                orderby s.Length ascending
                select s;
    return sorted;
}

public void SubWordSearchComplete() {
    mode = GameMode.levelPrep;
}

}

```

1. There is a good explanation of LINQ at the Unity Gems website. This link is from the Internet Archive to ensure that it remains valid. <https://web.archive.org/web/20140209060811/http://unitygems.com/linq-1-time-linq/>

This code creates the level, chooses a goal word, and populates it with subWords that can be spelled

using the characters in the goal word. When you press Play, you should now see the `currLevel` field populate in the `_MainCamera` Inspector.

Save your scene! If you haven't been saving your scene all along—and this served as a reminder to do so—you need to be reminding yourself to save more often.

Laying Out the Screen

Now that the level has been created, it's time to generate on-screen visuals to represent both the big letters that can be used to spell words and the regular letters of the words. To start, you need to create a `PrefabLetter` to be instantiated for each letter.

Making PrefabLetter

Follow these steps to make `PrefabLetter`:

1. From the menu bar, choose `GameObject > Create Other > Quad`. Rename the quad to `PrefabLetter`.
2. From the menu bar, choose `Assets > Create > Material`. Name the material `LetterMat` and place it in the Materials & Textures folder.
3. Drag `LetterMat` onto `PrefabLetter` to assign it. Click on `PrefabLetter`, and set the `shader` of `LetterMat` to `ProtoTools > UnlitAlpha`. Then select `Rounded Rect 256` as the texture for the `LetterMat` material.
4. If you double-click `PrefabLetter` in the Hierarchy, you should now see a nice rounded rectangle there. If you can't see it, you may need to orbit the camera around to the other side. (Backface culling makes quads visible only from one side and invisible from the other.)
5. From the menu bar, choose `GameObject > Create Other > 3D Text` and rename it `3D Text`. Drag the `3D Text` onto `PrefabLetter` in the Hierarchy to make it a child of `PrefabLetter`. Then select `3D Text` in the Hierarchy and set it to the settings shown in [Figure 33.3](#).

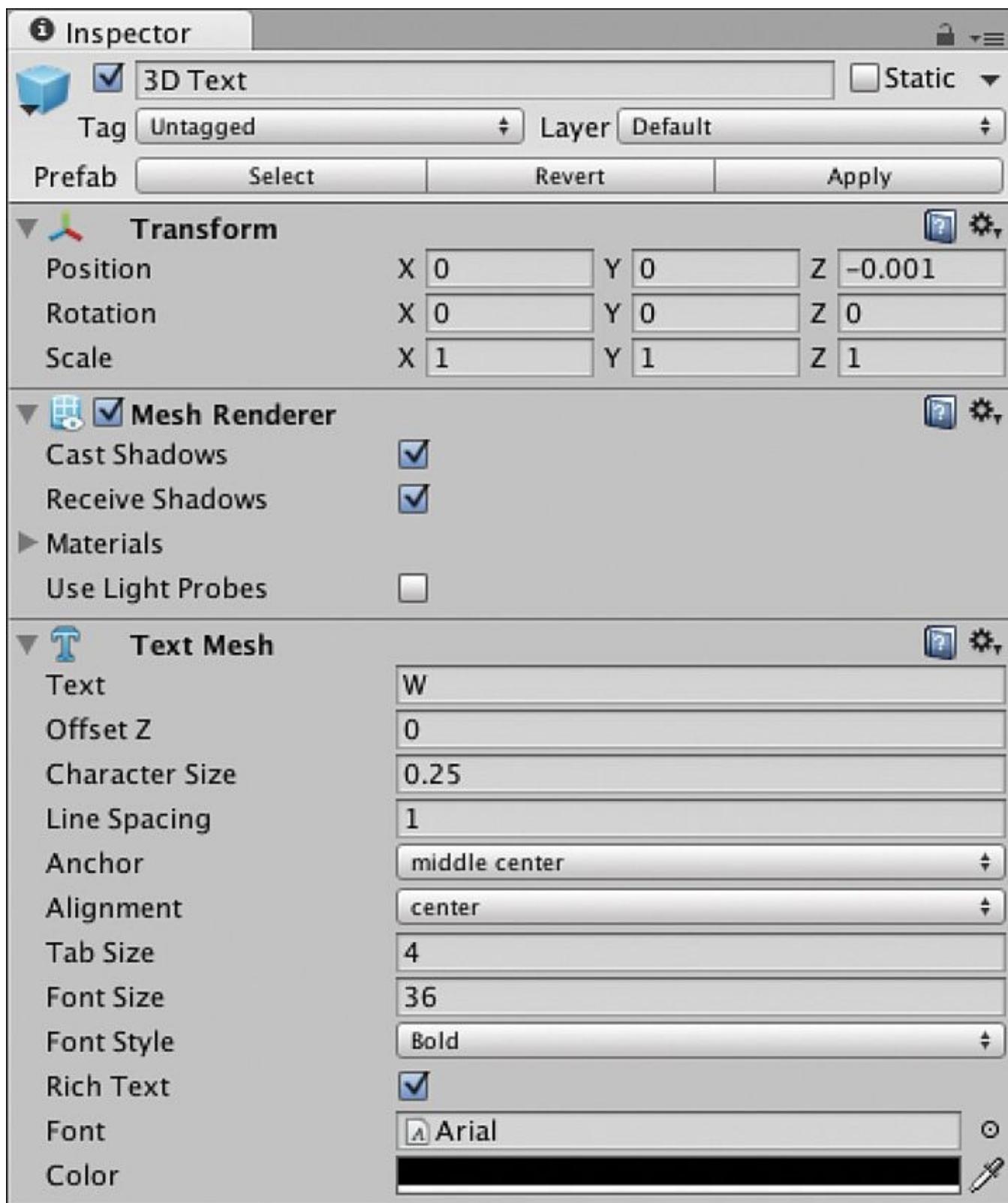


Figure 33.3 The Inspector settings for 3D Text, a child of PrefabLetter

6. Once PrefabLetter is ready, drag it into the `_Prefabs` folder in the Project pane and delete the remaining instance from the Hierarchy.

The Letter C# Script

PrefabLetter will have its own C# script to handle setting the character it shows, its color, and various other things. Create a new C# script named *Letter* and attach it to PrefabLetter. Then open it in MonoDevelop and enter the following code:

[Click here to view code image](#)

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

public class Letter : MonoBehaviour {

    private char           _c;      // The char shown on this Letter
    public TextMesh         tMesh;   // The TextMesh shows the char
    public Renderer         tRend;   // The Renderer of 3D Text. This will
                                    // determine whether the char is visible
    public bool             big = false; // Big letters act differently

    void Awake() {
        tMesh = GetComponentInChildren<TextMesh>();
        tRend = tMesh.renderer;
        visible = false;
    }

    // Used to get or set _c and the letter shown by 3D Text
    public char             c {
        get {
            return( _c );
        }
        set {
            _c = value;
            tMesh.text = _c.ToString();
        }
    }

    // Gets or sets _c as a string
    public string           str {
        get {
            return( _c.ToString() );
        }
        set {
            c = value[0];
        }
    }

    // Enables or disables the renderer for 3D Text, which causes the char to be
    // visible or invisible respectively.
    public bool             visible {
        get {
            return( tRend.enabled );
        }
        set {
            tRend.enabled = value;
        }
    }

    // Gets or sets the color of the rounded rectangle
    public Color             color {
        get {
            return(renderer.material.color);
        }
        set {
            renderer.material.color = value;
        }
    }

    // Sets the position of the Letter's gameObject
    public Vector3           pos {
        set {
            transform.position = value;
        }
    }
}
```

```
    }
}
```

This class makes use of several properties (faux fields with `get {}` and `set {}`) to perform various actions when variables are set. This enables, for instance, `WordGame` to set the `char c` of a `Letter` without worrying about how that gets converted to a string and then shown by 3D Text. This kind of encapsulation of functionality within a class is central to object-oriented programming.

The Wyrd Class: A Collection of Letters

Create a new C# script named `Wyrd`. The `Wyrd` class will act as a collection of `Letters`, and its name is spelled with a `y` to differentiate it from the other instances of the word `word` throughout the code and the text of this book. `Wyrd` is another class that does not extend `MonoBehaviour` and cannot be attached to a `GameObject`, but it can still contain `List<>`s of classes that are attached to `GameObjects`.

Enter the following code:

[Click here to view code image](#)

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

public class Wyrd {
    public string      str; // A string representation of the word
    public List<Letter> letters = new List<Letter>();
    public bool        found = false; // True if the player has found this word

    // A property to set visibility of the 3D Text of each Letter
    public bool visible {
        get {
            if (letters.Count == 0) return(false);
            return(letters[0].visible);
        }
        set {
            foreach( Letter lett in letters) {
                lett.visible = value;
            }
        }
    }

    // A property to set the rounded rectangle color of each Letter
    public Color color {
        get {
            if (letters.Count == 0) return(Color.black);
            return(letters[0].color);
        }
        set {
            foreach( Letter lett in letters) {
                lett.color = value;
            }
        }
    }

    // Adds a Letter to letters
    public void Add(Letter lett) {
        letters.Add(lett);
        str += lett.c.ToString();
    }
}
```

}

WordGame Layout

The Layout() function will generate Wyrds and Letters for the game as well as big Letters that the player can use to spell words (shown as large gray letters in the screenshot at the beginning of this chapter). We'll start with the small letters, and for this phase of the prototype, we'll make the letters visible to begin with (rather than hiding them as we'll do in the final version). Add the following code to WordGame:

[Click here to view code image](#)

```
public class WordGame : MonoBehaviour {
    static public WordGame     S; // Singleton

    public GameObject         prefabLetter;
    public Rect                wordArea = new Rect(-24, 19, 48, 28);
    public float               letterSize = 1.5f;
    public bool                showAllWyrds = true;
    public float               bigLetterSize = 4f;

    public bool _____;

    public GameMode            mode = GameMode.preGame;
    public WordLevel           currLevel;
    public List<Wyrd>          wyrds;

    ...

    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;
        Layout(); // Call the Layout() function after SubWordSearch
    }

    void Layout() {
        // Place the letters for each subword of currLevel on screen
        wyrds = new List<Wyrd>();

        // Declare a lot of variables that will be used in this method
        GameObject go;
        Letter lett;
        string word;
        Vector3 pos;
        float left = 0;
        float columnWidth = 3;
        char c;
        Color col;
        Wyrd wyrd;

        // Determine how many rows of Letters will fit on screen
        int numRows = Mathf.RoundToInt(wordArea.height/letterSize);

        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            wyrd = new Wyrd();
            word = currLevel.subWords[i];

            // if the word is longer than columnWidth, expand it
            columnWidth = Mathf.Max( columnWidth, word.Length );

            // Instantiate a PrefabLetter for each letter of the word
        }
    }
}
```

```
        for (int j=0; j<word.Length; j++) {
            c = word[j]; // Grab the jth char of the word
            go = Instantiate(prefabLetter) as GameObject;
            lett = go.GetComponent<Letter>();
            lett.c = c; // Set the c of the Letter
            // Position the Letter
            pos = new Vector3(wordArea.x+left+j*letterSize, wordArea.y, 0);
            // The % here makes multiple columns line up
            pos.y -= (i%numRows)*letterSize;
            lett.pos = pos;
            go.transform.localScale = Vector3.one*letterSize;
            wyrd.Add(lett);
        }

        if (showAllWyrds) wyrd.visible = true; // This line is for testing

        wyrds.Add(wyrd);

        // If we've gotten to the numRows(th) row, start a new column
        if (i%numRows == numRows-1) {
            left += (columnWidth+0.5f)*letterSize;
        }
    }
}
```

Before pressing Play, you need to assign the `PrefabLetter` prefab from the Project pane to the `prefabLetter` field of the *WordGame (Script)* component of `_MainCamera`. After doing so, press Play, and you should see a list of words pop up on screen, as shown in [Figure 33.4](#).



Figure 33.4 An example of the current state of the game: the level for the word TORNADO

Adding the Big Letters at the Bottom

The next step in `Layout()` is to place the large letters at the bottom of the screen. Add the following code to do so:

[Click here to view code image](#)

```
public class WordGame : MonoBehaviour {
    ...
    public float bigLetterSize = 4f;
    public Color bigColorDim = new Color(0.8f, 0.8f, 0.8f);
    public Color bigColorSelected = Color.white;
    public Vector3 bigLetterCenter = new Vector3(0, -16, 0);

    public bool _____;
    public GameMode mode = GameMode.preGame;
    public WordLevel currLevel;
    public List<Wyrd> wyrds;
    public List<Letter> bigLetters;
    public List<Letter> bigLettersActive;
    ...
    void Layout() {
        ...
    }
}
```

```

// Make a Wyrd of each level.subWord
for (int i=0; i<currLevel.subWords.Count; i++) {
    ...
}

// Place the big letters
// Initialize the List<>s for big Letters
bigLetters = new List<Letter>();
bigLettersActive = new List<Letter>();

// Create a big Letter for each letter in the target word
for (int i=0; i<currLevel.word.Length; i++) {
    // This is similar to the process for a normal Letter
    c = currLevel.word[i];
    go = Instantiate(prefabLetter) as GameObject;
    lett = go.GetComponent<Letter>();
    lett.c = c;
    go.transform.localScale = Vector3.one*bigLetterSize;

    // Set the initial position of the big Letters below screen
    pos = new Vector3( 0, -100, 0 );
    lett.pos = pos;

    col = bigColorDim;
    lett.color = col;
    lett.visible = true; // This is always true for big letters
    lett.big = true;
    bigLetters.Add(lett);
}
// Shuffle the big letters
bigLetters = ShuffleLetters(bigLetters);
// Arrange them on screen
ArrangeBigLetters();

// Set the mode to be in-game
mode = GameMode.inLevel;
}

// This shuffles a List<Letter> randomly and returns the result
List<Letter> ShuffleLetters(List<Letter> letts) {
    List<Letter> newL = new List<Letter>();
    int ndx;
    while(letts.Count > 0) {
        ndx = Random.Range(0,letts.Count);
        newL.Add(letts[ndx]);
        letts.RemoveAt(ndx);
    }
    return(newL);
}

// This arranges the big Letters on screen
void ArrangeBigLetters() {
    // The halfWidth allows the big Letters to be centered
    float halfWidth = ( (float) bigLetters.Count )/2f-0.5f;
    Vector3 pos;
    for (int i=0; i<bigLetters.Count; i++) {
        pos = bigLetterCenter;
        pos.x += (i-halfWidth)*bigLetterSize;
        bigLetters[i].pos = pos;
    }
    // bigLettersActive
    halfWidth = ( (float) bigLettersActive.Count )/2f-0.5f;
    for (int i=0; i<bigLettersActive.Count; i++) {
        pos = bigLetterCenter;

```

```

        pos.x += (i-halfWidth)*bigLetterSize;
        pos.y += bigLetterSize*1.25f;
        bigLettersActive[i].pos = pos;
    }
}

}

```

Now, in addition to the Letters up top, you should also see big Letters below, the shuffled form of the goal word. It's time to add some interactivity.

Adding Interactivity

For this game, we want the player to be able to type words from the available big Letters on her keyboard and press Return/Enter to submit them. She can also press Backspace/Delete to remove a letter from the end of what she has typed and press the space bar to shuffle the remaining unselected letters.

When she presses Enter, the word she typed will be compared with the possible words in the WordLevel. If the word she typed is in the WordLevel, she will get a point for each letter in the word. In addition, if the word she typed contains any smaller words that are in the WordLevel, she will also get points for those plus a multiplier for each word. Looking at the TORNADO example earlier, if a player typed TORNADO as her first word and hit Return, she would get 36 total points as follows:

TORNADO	7 x 1 points	1 point for each letter x 1 for first word
TORN	4 x 2 points	1 point per letter x 2 for second word = 8 points
TOR	3 x 3 points	1 point per letter x 3 = 9 points
ADO	<u>+ 3 x 4 points</u>	1 point per letter x 4 = 12 points
36 total points		

All of this interactivity will be handled by the `Update()` function and will be based on `Input.inputString`, a string of all the keyboard input that occurred this frame. Add the following `Update()` method and supporting methods to `WordGame`:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...

    public List<Letter> bigLettersActive;
    public string testWord;
    private string upperCase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    ...

    void Update() {
        // Declare a couple of useful local variables
        Letter lett;
        char c;

        switch (mode) {
        case GameMode.inLevel:
            // Iterate through each char input by the player this frame
            foreach (char cIt in Input.inputString) {
                // Shift cIt to UPPERCASE
                c = System.Char.ToUpperInvariant(cIt);

                // Check to see if it's an uppercase letter
                if (upperCase.Contains(c)) { // Any uppercase letter

```

```

        // Find an available Letter in bigLetters with this char
        lett = FindNextLetterByChar(c);
        // If a Letter was returned
        if (lett != null) {
            // ... then add this char to the testWord and move the
            // returned big Letter to bigLettersActive
            testWord += c.ToString();
            // Move it from the inactive to the active List<>
            bigLettersActive.Add(lett);
            bigLetters.Remove(lett);
            lett.color = bigColorSelected; // Make it the active color
            ArrangeBigLetters();           // Rearrange the big Letters
        }
    }

    if (c == '\b') { // Backspace
        // Remove the last Letter in bigLettersActive
        if (bigLettersActive.Count == 0) return;
        if (testWord.Length > 1) {
            // Clear the last char of testWord
            testWord = testWord.Substring(0,testWord.Length-1);
        } else {
            testWord = "";
        }

        lett = bigLettersActive[bigLettersActive.Count-1];
        // Move it from the active to the inactive List<>
        bigLettersActive.Remove(lett);
        bigLetters.Add (lett);
        lett.color = bigColorDim;      // Make it the inactive color
        ArrangeBigLetters();           // Rearrange the big Letters
    }

    if (c == '\n' || c == '\r') { // Return/Enter
        // Test the testWord against the words in WordLevel
        Checkword();
    }

    if (c == ' ') { // Space
        // Shuffle the bigLetters
        bigLetters = ShuffleLetters(bigLetters);
        ArrangeBigLetters();
    }
}

break;
}

}

// This finds an available Letter with the char c in bigLetters.
// If there isn't one available, it returns null.
Letter FindNextLetterByChar(char c) {
    // Search through each Letter in bigLetters
    foreach (Letter l in bigLetters) {
        // If one has the same char as c
        if (l.c == c) {
            // ...then return it
            return(l);
        }
    }
    // Otherwise, return null
    return( null );
}

```

```

public void CheckWord() {
    // Test testWord against the level.subWords
    string subWord;
    bool foundTestWord = false;

    // Create a List<int> to hold the indices of other subWords that are
    // contained within testWord
    List<int> containedWords = new List<int>();

    // Iterate through each word in currLevel.subWords
    for (int i=0; i<currLevel.subWords.Count; i++) {

        // If the ith Wyrd on screen has already been found
        if (wyrd[i].found) {
            // ...then continue & skip the rest of this iteration
            continue;
            // This works because the Wyrds on screen and the words in the
            // subWords List<> are in the same order
        }

        subWord = currLevel.subWords[i];
        // if this subWord is the testWord
        if (string.Equals(testWord, subWord)) {
            // ...then highlight the subWord
            HighlightWyrd(i);
            foundTestWord = true;
        } else if (testWord.Contains(subWord)) {
            // ^else if testWord contains this subWord (e.g., SAND contains AND)
            // ...then add it to the list of containedWords
            containedWords.Add(i);
        }
    }

    // If the test word was found in subWords
    if (foundTestWord) {
        // ...then highlight the other words contained in testWord
        int numContained = containedWords.Count;
        int ndx;
        // Highlight the words in reverse order
        for (int i=0; i<containedWords.Count; i++) {
            ndx = numContained-i-1;
            HighlightWyrd( containedWords[ndx] );
        }
    }

    // Clear the active big Letters regardless of whether testWord was valid
    clearBigLettersActive();
}

// Highlight a Wyrd
void HighlightWyrd(int ndx) {
    // Activate the subWord
    wyrd[ndx].found = true;    // Let it know it's been found
    // Lighten its color
    wyrd[ndx].color = (wyrd[ndx].color+Color.white)/2f;
    wyrd[ndx].visible = true; // Make its 3D Text visible
}

// Remove all the Letters from bigLettersActive
void ClearBigLettersActive() {
    testWord = "";           // Clear the testWord
}

```

```

        foreach (Letter l in bigLettersActive) {
            bigLetters.Add(l); // Add each Letter to bigLetters
            l.color = bigColorDim; // Set it to the inactive color
        }
        bigLettersActive.Clear(); // Clear the List<>
        ArrangeBigLetters(); // Rearrange the Letters on screen
    }
}

```

After you've entered all this, you will also need to set `showAllWyrds` to false in the Inspector for the `WordGame (Script)` component of `_MainCamera`. Then, press Play.

You should be presented with a working version of the game and a random level.

Adding Scoring

Because of the Scoreboard and FloatingScore code that we've already written and imported into this project, adding scoring to this game should be very easy. Start by dragging Scoreboard from the `_Prefab` folder in the Project pane into the Hierarchy pane. Both the Scoreboard instance now in the Hierarchy and the `PrefabFloatingScore` should be preset for this game. (If you want to learn more about how they work, refer to [Chapter 31, “Prototype 4: Prospector Solitaire.”](#))

Now, you must add scoring code to the `WordGame C#` script by making the following edits:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...
    public Vector3 bigLetterCenter = new Vector3(0, -16, 0);
    public List<float> scoreFontSizes = new List<float> { 24, 36, 36, 1 };
    public Vector3 scoreMidPoint = new Vector3(1,1,0);
    public float scoreComboDelay = 0.5f;
    ...

    public void CheckWord() {
        ...
        // Iterate through each word in currLevel.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // if this subWord is the testWord
            if (string.Equals(testWord, subWord)) {
                // ...then highlight the subWord
                HighlightWyrd(i);
                Score( wyrds[i], 1 ); // Score the testWord
                foundTestWord = true;
            }
            ...
        }

        // If the test word was found in subWords
        if (foundTestWord) {
            ...
            // Highlight the words in reverse order
            for (int i=0; i<containedWords.Count; i++) {
                ndx = numContained-i-1;
                HighlightWyrd( containedWords[ndx] );
                Score( wyrds[ containedWords[ndx] ], i+2 ); // Score other words
                // The second parameter (i+2) is the # of this word in the combo
            }
        }
    }
}

```

```

}

...
}

// Add to the score for this word
// int combo is the number of this word in a combo
void Score(Wyrd wyrd, int combo) {
    // Get the position of the first Letter in the wyrd
    Vector3 pt = wyrd.letters[0].transform.position;
    // Create a List<Vector3> of Bezier points for the FloatingScore
    List<Vector3> pts = new List<Vector3>();

    // Convert the pt to a ViewportPoint. ViewportPoints range from 0 to 1
    // across the screen and are used for GUI coordinates
    pt = Camera.main.WorldToViewportPoint(pt);
    pt.z = 0;

    // Make pt the first Bezier point
    pts.Add(pt);

    // Add a second Bezier point
    pts.Add(scoreMidPoint);

    // Make the Scoreboard the last Bezier point
    pts.Add(Scoreboard.S.transform.position);

    // Set the value of the Floating Score
    int value = wyrd.letters.Count * combo;
    FloatingScore fs = Scoreboard.S.CreateFloatingScore(value, pts);

    fs.timeDuration = 2f;
    fs.fontSizes = scoreFontSizes;

    // Double the InOut Easing effect
    fs.easingCurve = Easing.InOut+Easing.InOut;

    // Make the text of the FloatingScore something like "3 x 2"
    string txt = wyrd.letters.Count.ToString();
    if (combo > 1) {
        txt += " x "+combo;
    }
    fs.guiText.text = txt;
}

}

```

After you've entered this code and run the game, you'll see that you get a score for each correct word you enter, and you get a multiplier for each additional valid word contained in the word you type. However, all the scores happen at the same time. Happily, this is something else that we can use coroutines to fix. Make the following changes to make the `CheckWord()` function into a coroutine that pauses for 0.5 seconds between each word in the combo:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...

    void Update() {
        ...
        switch (mode) {

```

```

case GameMode.inLevel:
    ...
    // Iterate through each char input by the player this frame
    foreach (char c in Input.inputString) {
        ...
        if (c == '\n') { // Return/Enter
            // Test the testWord against the words in WordLevel
            StartCoroutine( CheckWord() );
        }
        ...
    }
    break;
}

...
}

public IEnumerator CheckWord() {
    ...

    // If the test word was found in subWords
    if (foundTestWord) {
        // ...then highlight the other words contained in testWord
        int numContained = containedWords.Count;
        int ndx;
        // Highlight the words in reverse order
        for (int i=0; i<containedWords.Count; i++) {

            // yield for a bit before highlighting each word
            yield return( new WaitForSeconds(scoreComboDelay) );

            ndx = numContained-i-1;
            HighlightWyrd( containedWords[ndx] );
            Score( wyrds[ containedWords[ndx] ], i+2 ); // Score other words
            // The second parameter (i+2) is the # of this word in the combo
        }
    }
    ...
}
...
}

```

Now, the scores will fly and the words will show with a separation of about 0.5 seconds.

Adding Animation

In a similar manner to scoring, we can very easily add smooth animation of Letters by taking advantage of the interpolation functions that we imported in the Utils script.

Add the following code to the Letter C# script:

[Click here to view code image](#)

```

public class Letter : MonoBehaviour {
    ...
    public bool big = false; // Big letters are a little different
    // Linear interpolation fields
    public List<Vector3> pts = null;
    public float timeDuration = 0.5f;
    public float timeStart = -1;
    public string easingCurve = Easing.InOut; // Easing from Utils.cs
    ...
}

```

```

// Now set up a Bezier curve to move to the new position
public Vector3 pos {
    set {
        // transform.position = value; // This line is now commented out

        // Find a midpoint that is a random distance from the actual
        // midpoint between the current position and the value passed in
        Vector3 mid = (transform.position + value)/2f;
        // The random distance will be within 1/4 of the magnitude of the
        // line from the actual midpoint
        float mag = (transform.position - value).magnitude;
        mid += Random.insideUnitSphere * mag*0.25f;
        // Create a List<Vector3> of Bezier points
        pts = new List<Vector3>() { transform.position, mid, value };
        // If timeStart is at the default -1, then set it
        if (timeStart == -1 ) timeStart = Time.time;
    }
}

// Moves immediately to the new position
public Vector3 position {
    set {
        transform.position = value;
    }
}

// Interpolation code
void Update() {
    if (timeStart == -1) return;

    // Standard linear interpolation code
    float u = (Time.time-timeStart)/timeDuration;
    u = Mathf.Clamp01(u);
    float u1 = Easing.Ease(u,easingCurve);
    Vector3 v = Utils.Bezier(u1, pts);
    transform.position = v;

    // If the interpolation is done, set timeStart back to -1
    if (u == 1) timeStart = -1;
}
}

```

Now, if you play the scene, you'll see the Letters all interpolate to their new positions. However, it looks a little strange for all the Letters to move at the same time and start from the center of the screen. Let's add some small changes to `WordGame.Layout()` to improve this:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...

    void Layout() {
        ...

        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...

            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                ...
                // The % here makes multiple columns line up
                pos.y -= (i%numRows)*letterSize;
            }
        }
    }
}

```

```

        // Move the lett immediately to a position above the screen
        lett.position = pos+Vector3.up*(20+i%numRows);
        // Then set the pos for it to interpolate to
        lett.pos = pos;
        // Increment lett.timeStart to move wyrds at different times
        lett.timeStart = Time.time + i*0.05f;

        go.transform.localScale = Vector3.one*letterSize;
        wyrd.Add(lett);
    }

    ...

}

...

// Create a big Letter for each letter in the target word
for (int i=0; i<currLevel.word.Length; i++) {
    ...
    go.transform.localScale = Vector3.one*bigLetterSize;

    // Set the initial position of the big Letters below screen
    pos = new Vector3( 0, -100, 0 );
    lett.pos = pos;

    // Increment lett.timeStart to have big Letters come in last
    lett.timeStart = Time.time + currLevel.subWords.Count*0.05f;
    lett.easingCurve = Easing.Sin+-0.18; // Bouncy easing

    col = bigColorDim;
    lett.color = col;

    ...
}

...
}
...
}

```

With this code, the game should now layout with nice smooth motions.

Adding Color

Now that the game moves well, it's time to add a little color:

1. Add the following code to WordGame to color the wyrds based on their length:

[Click here to view code image](#)

```

public class WordGame : MonoBehaviour {
    ...
    public float scoreComboDelay = 0.5f;
    public Color[] wyrdPalette;

    public bool _____;
    ...

    void Layout() {
        ...
        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                ...
            }
        }
    }
}

```

```
        wyrd.Add(lett);
    }

    if (showAllWyrd) wyrd.visible = true; // This line is for testing

    // Color the wyrd based on length
    wyrd.color = wyrdPalette[word.Length-WordList.S.wordLengthMin];

    wyrds.Add(wyrd);
    ...
}

...
}
```

These last few code changes have been so simple because we already had supporting code in place (for example, the `Wyrd.color` and `Letter.color` properties as well as the `Easing` code in the `Utils` class).

2. Now, you need to set about eight colors for wyrdPalette. To do this, we'll use the Color Palette image included in the import at the beginning of the project. We're going to be using the eye dropper to set color, which may leave you wondering how to see both the Color Palette image and the _MainCamera Inspector at the same time. To do this, we'll take advantage of Unity's capability to have more than one Inspector window open at the same time.
 3. As shown in [Figure 33.5](#), click the pane options button (circled in red) and choose *Add Tab > Inspector* to add an Inspector to the Game tab. Then select the Color Palette image in the Project pane. It will appear in both Inspectors. (You might need to drag the edge of the image preview part of the Inspector to make it look like the [Figure 33.6](#).) Click the lock icon on one inspector (circled in red in the [Figure 33.6](#)), and then select _MainCamera in the Hierarchy pane. You'll see that the unlocked Inspector changes to _MainCamera, but the locked one is still showing Color Palette.

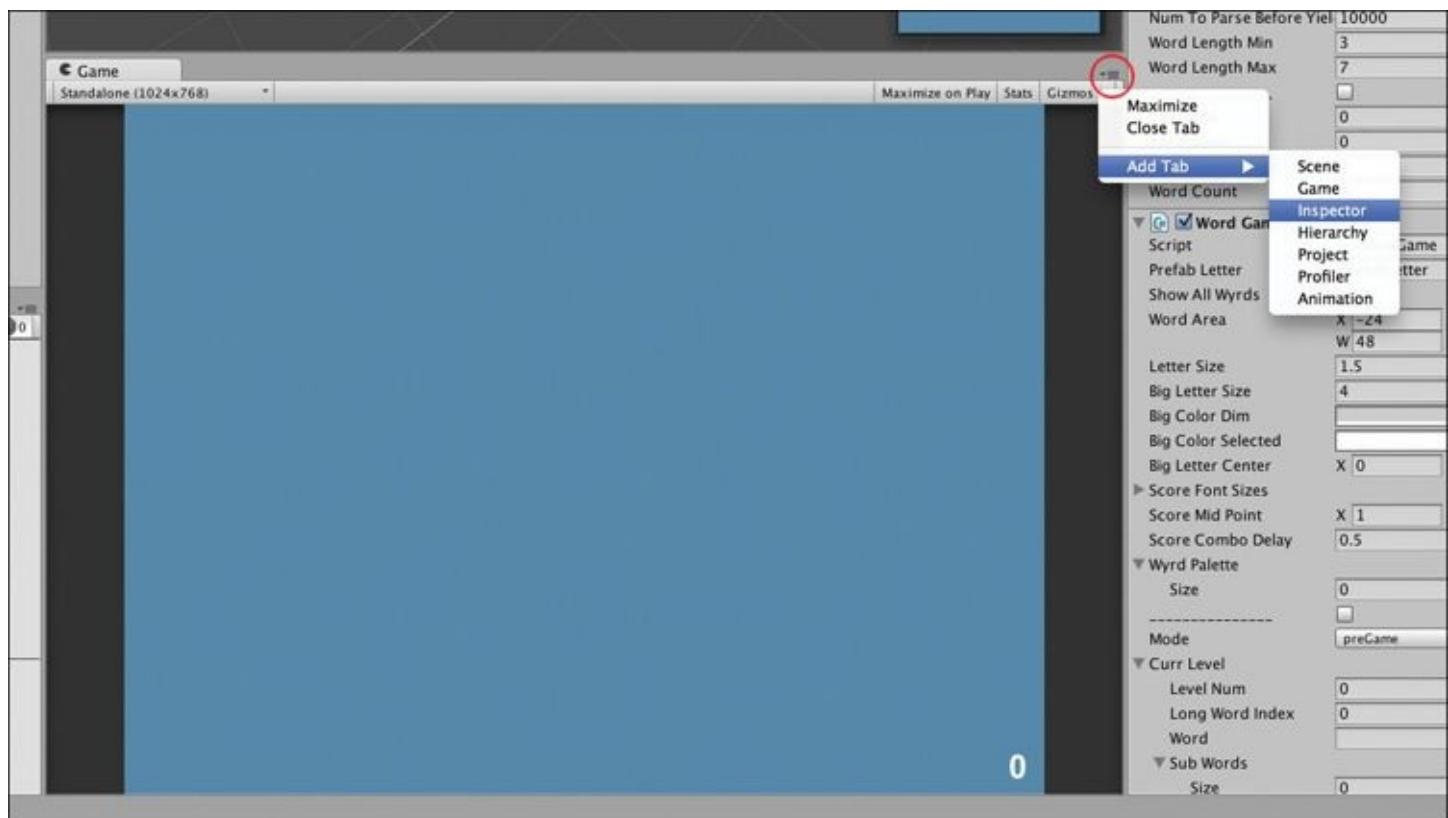


Figure 33.5 Using the pane options button to add an Inspector to the Game pane

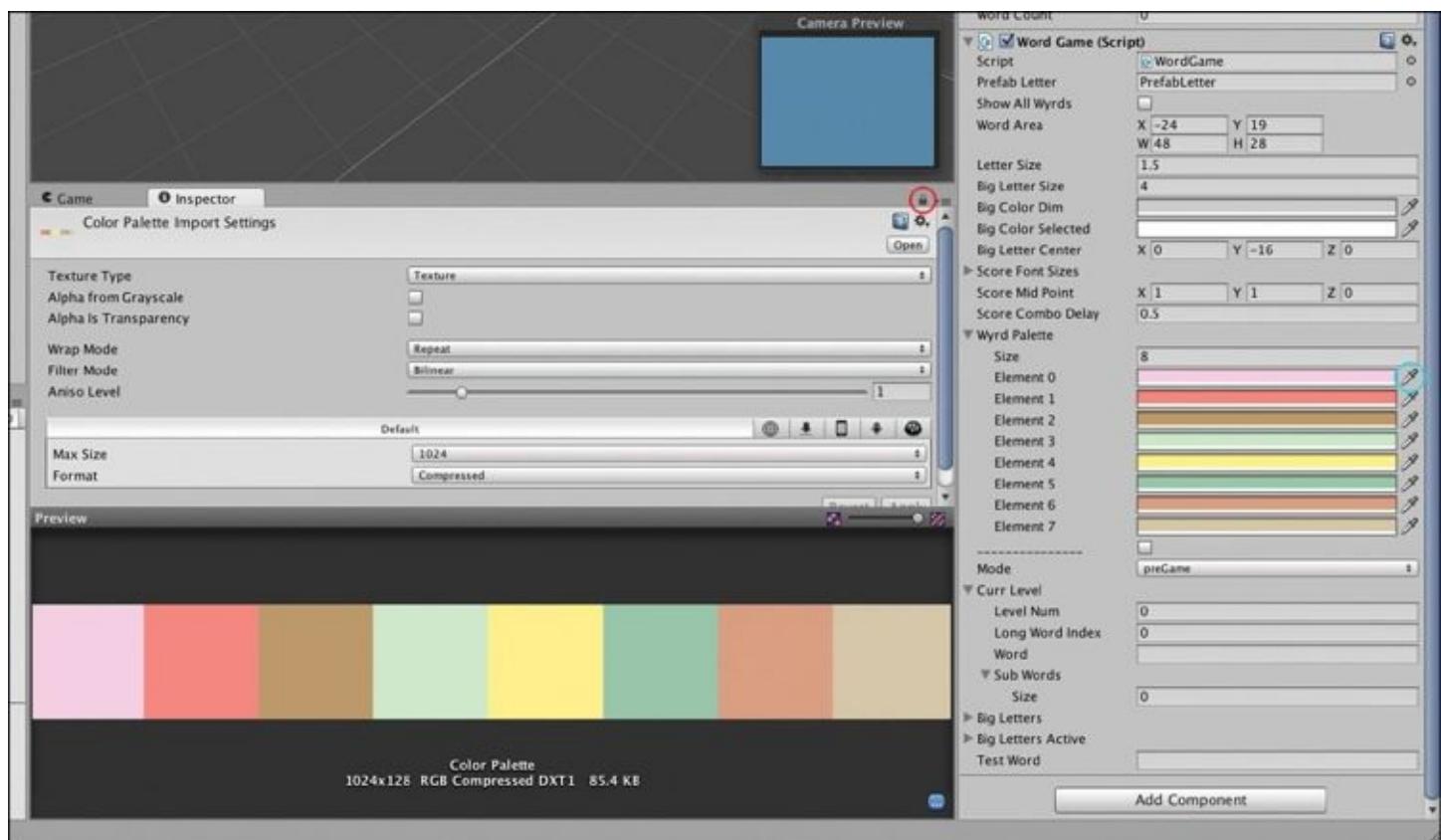


Figure 33.6 The lock icon on one Inspector (circled in red) and the eye dropper in the other inspector (circled in light blue)

4. Expand the disclosure triangle next to `wyrdPalette` in the `_MainCamera` Inspector and set its size to 8. Click the eye dropper next to each `wyrdPalette` element (circled in light blue in [Figure 33.6](#)), and then click one of the colors in the Color Palette image. Doing this for each element of `wyrdPalette` will give you the eight different colors of the Color Palette image, but they will all default to having an alpha of 0 (and therefore being invisible). Click each color bar in the `wyrdPalette` array and set each one's alpha (or A) to 255 to make it fully opaque, as indicated by the white bar below the color swatch.

Now when you play the scene, you should see something that looks like the screen shot from the beginning of the chapter.

Summary

In this chapter, you created a simple word game and added a little flair to it with some nice interpolated movement. If you've been following these tutorials in order, you may have realized that the process of making them is getting a little bit easier. With the expanded understanding of Unity that you now have and the capabilities of readymade utility scripts like Scoreboard, FloatingScore, and Utils, you're able to focus more of the coding effort on the things that are new and different in each game and less on reinventing the wheel.

Next Steps

In the previous prototypes, you saw examples of how to set up a series of game states to handle the different phases of the game and transition from one level to the next. Right now, this prototype doesn't have any of that. On your own, you should add that kind of control structure to this game. Here are some things to think about as you do so:

- When should the player be able to move on to the next level? Must she guess every single word,

or can she move on once she has either reached a specific point total or has guessed the target word.

- How will you handle levels? Will you just pick a completely random word as we are now, or will you modify the randomness to make sure that level 5 is always the same word (therefore making it fair for players to compare their scores on level 5)? Here's a hint if you decide to try for a modified randomness:

[Click here to view code image](#)

```
int PickNthRandom(int n, int range) {  
    // If 0 is passed in, 0 is returned.  
  
    int seed = Random.seed; // Store the current Random.seed  
  
    // The Random.seed sets the starting point for the Random function.  
    // With the same Random.seed, Random.value will generate the same  
    // results, in the same order, every time.  
  
    Random.seed = 0; // Set a new Random.seed  
  
    // Get the nth random number between 0 and range-1  
    int j=0;  
    for (int i=0; i<n; i++) {  
        j = Random.Range(0,range);  
    }  
  
    Random.seed = seed; // Restore the original Random.seed  
  
    return(j);  
}
```

- How do you want to handle levels with too many or too few subWords? Some collections of seven letters have so many words that they extend off the screen to the right, whereas others have so few that there's only one column. Do you want to make the game ask for the next word in this case? If so, how do you then instruct something like the PickNthRandom function to skip certain numbers?

You should have enough knowledge of programming and prototyping now that you can take these questions and make this into a real game. You've got the skills, now go for it!

Chapter 34. Prototype 7: QuickSnap

In this chapter, you make a puzzle game based on first-person shooter (FPS) movement and control. The player is shown a complex environment and tasked with taking specific photos.

Although this book certainly could have included a more traditional FPS (first-person shooter), you could probably already make one yourself using the skills you've learned from previous chapters. Instead, this prototype focuses on a different style of gameplay and enables me to present some concepts such as XML creation and light baking.

Getting Started: QuickSnap Prototype

You'll be importing a much larger unitypackage for this project than you have in the past. This is because the game will be built inside a modified version of the environment that Unity Technologies created for their *Stealth* game prototype. It's an environment that contains a number of interesting objects and complex lighting, so it will look much better than any of our earlier prototypes.

You have two choices for the unitypackage to import. One contains high-resolution graphics and is about 200 MB in size, whereas the other uses lower-resolution graphics and is only about 88 MB. However, both will work equally well for the development of the prototype, and you shouldn't notice any real difference between the two unless you play at very high resolution and are looking for it. Baking lightmaps will probably take longer on the 200 MB version, but it will also be a little bit prettier.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup.”](#) When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 3D for this project.

- **Project name** QuickSnap.
- **unitypackage:** Find and download one of the two packages for [Chapter 34](#) at <http://book.prototools.net>. Check your build settings *before* importing the package.
- **Scene name:** __QuickSnap_Scene_0.
- **Project folders:** These are all imported from the unitypackage.
- **C# script names:** Just the imported scripts in the ProtoTools folder.

Build and Aspect Ratio Settings

If you have recently completed one of the other projects in the book, your build settings and the aspect ratio of your Game pane may need to be updated.

Build Settings

Open the Build Settings window in Unity (*File > Build Settings* from the menu bar) and make sure that the Platform is set to *PC, Mac & Linux Standalone*. If it is not, click that selection in the Platforms list and then click the *Switch Platform* button. The *Switch Platform* button will turn gray once the switch is complete. Close the Build Settings window.

Aspect Ratio

In the Game pane, choose 16:9 from the pop-up menu of aspect ratio choices (located directly beneath the heading tab of the Game pane).

Importing the unitypackage

Choose which package you want to import and do so. The package import might take a while—especially if you chose the high-res version—because Unity needs to import and compress all the image files for the environment.

Building the Scene

Once the import is done, drag *environment* from the *_Prefabs* folder into the Hierarchy pane. You'll see that this is a very complex environment, and it may even render somewhat slowly on your machine. If so, click the button that looks like a sun at the top of the Scene pane; this will toggle lighting calculations. The darker-looking scene is actually lit, whereas the brighter rendering of the scene is just the raw textures and should render faster.

Adding a First-Person Controller

Unity Technologies has built a couple of character controller scripts that they include with every install of Unity. To import them, choose *Assets > Import Package > Character Controller* from the menu bar. This will pop up a list of all the assets in the package. Go ahead and import them all.

The import will create a new folder in the Project pane called *Standard Assets*. Open the disclosure triangle next to *Standard Assets* and then the one next to *Character Controllers*, and you'll find two character controller prefabs. Drag the *First Person Controller* into the Hierarchy pane, click it, and change the name of the instance to *_FPC*. *_FPC* should have a Transform of:

_FPC (GameObject) P:[-2,1,0] R:[0,-90,0] S:[1,1,1]

Opening the disclosure triangle next to *_FPC*, you'll see that there are two GameObjects: *Graphics* and *Main Camera*. *Graphics* holds the white capsule that you can see in the Scene pane, and *Main Camera* holds the camera that's attached to *_FPC*. Because there is a Camera on *_FPC*, we no longer need the original *Main Camera* that came with the scene. Select the black *Main Camera* at the top level of the Hierarchy and delete it (Command-Delete on Mac or just Delete on PC). Next, select the child of *_FPC* named *Graphics* and disable it by clicking the check box in the top left of the Inspector pane. This will keep the white capsule from appearing in the Game or Scene panes.

Press Play, and you should be able to move around the space using standard First-Person Shooter (FPS) controls (WASD or arrows for movement, Space to jump, and mouse movement to look around). You should notice a couple of things while walking around: The eye point of the camera seems a bit high, and the scene is pretty dull and dark.

The first of these issues is easy to fix. Select *_FPC* in the Hierarchy and change the *Height* value within the *Character Controller* component of the Inspector from 2 to 1. Now, the camera will be at a lower height that is more fitting for the environment.

The second is going to take more work, and it's going to take some time.

Lightmapping in Unity

Lightmapping is the process in Unity of creating textures for objects that have complicated lighting calculations *baked* into them. This means that rather than having the same texture repeat across an entire wall, a lightmap texture is made that also includes the effects of various lights on the wall. Unity 4 lightmapping works through an included piece of software called Autodesk Beast, while Unity 5 uses Geomerics Enlighten engine. Because of the differences in interface and settings between these lightmapping solutions, if you are using Unity 5 you should download the Unity 5 version of this chapter from <http://book.prototools.net> that covers the Enlighten engine instead of Beast.

Beast calculates information about all of the lights in your scene and where they would hit various objects that are marked as *static* (that is, they will never move during the course of the game). If you open the disclosure triangle next to *environment* in the Hierarchy and select the child *env_stealth_static*, you'll see that *Static* is checked in the top-right corner of the Inspector for both *env_stealth_static* and all of its children. Clicking the downward triangle pop-up menu in the Inspector next to *Static* will also reveal that it is set to static for all types of lighting calculations.

At high quality levels, lightmapping can create beautiful shadows and even handles reflections of light so that, for instance, the area around a shiny red object would be tinted red by the light bouncing off of it. All of this information is collected and baked into the textures for static objects in the scene, meaning that the original texture for an object is replaced by a texture that includes the additional lighting information.

However, lightmapping is one of the few areas of difference between Unity Free and Unity Pro. Pro allows you to take advantage of deferred rendering and dual lightmaps, both of which improve the visuals that Unity is able to produce in real time. Take a look at [Figure 34.1](#).



Figure 34.1 Various levels of lightmapping and rendering in Unity

The three series of images in [Figure 34.1](#) show the same scene rendered first with forward rendering and no lightmapping. The second shows the scene with forward rendering and single lightmaps. The third column shows deferred rendering and directional lightmaps. The changes between the first two

columns are most obvious, but there are subtle differences in the third column that make it look even better. As you can see in the top and middle rows of images, the deferred rendering of the third column enables images and light to be correctly mapped to the floor (the blue glow beneath the large, floating security card in the corner of the room in the top row, and the shadows cast by the barrel in the second row). In addition, in the third column, you can see that the walls are casting subtle shadows on the ground that are missing in the middle column. Though deferred rendering and directional lightmaps are not available in Unity Free, the good news is that you can still get all of the benefits from lightmap baking that are shown in the second column.

Light Probes

One of the limitations of lightmapping is that it can only be used on static objects, so a character or other object that moves through the scene cannot be lightmapped. However, with Unity Pro, it is possible to fake lightmapping on a dynamic (that is, nonstatic) object through the use of *light probes*. Light probes are a way for Unity to map what the light would look like in the open spaces of a scene, as is shown in [Figure 34.2](#). Each spherical light probe stores information about the lighting at a certain location in the scene (though they are, of course, invisible when the scene is playing). If a dynamic object is set to use light probes, the shading on that object will be interpolated based on its position relative to the position of the light probes. For more information, search for “light probes” in the Unity Manual.

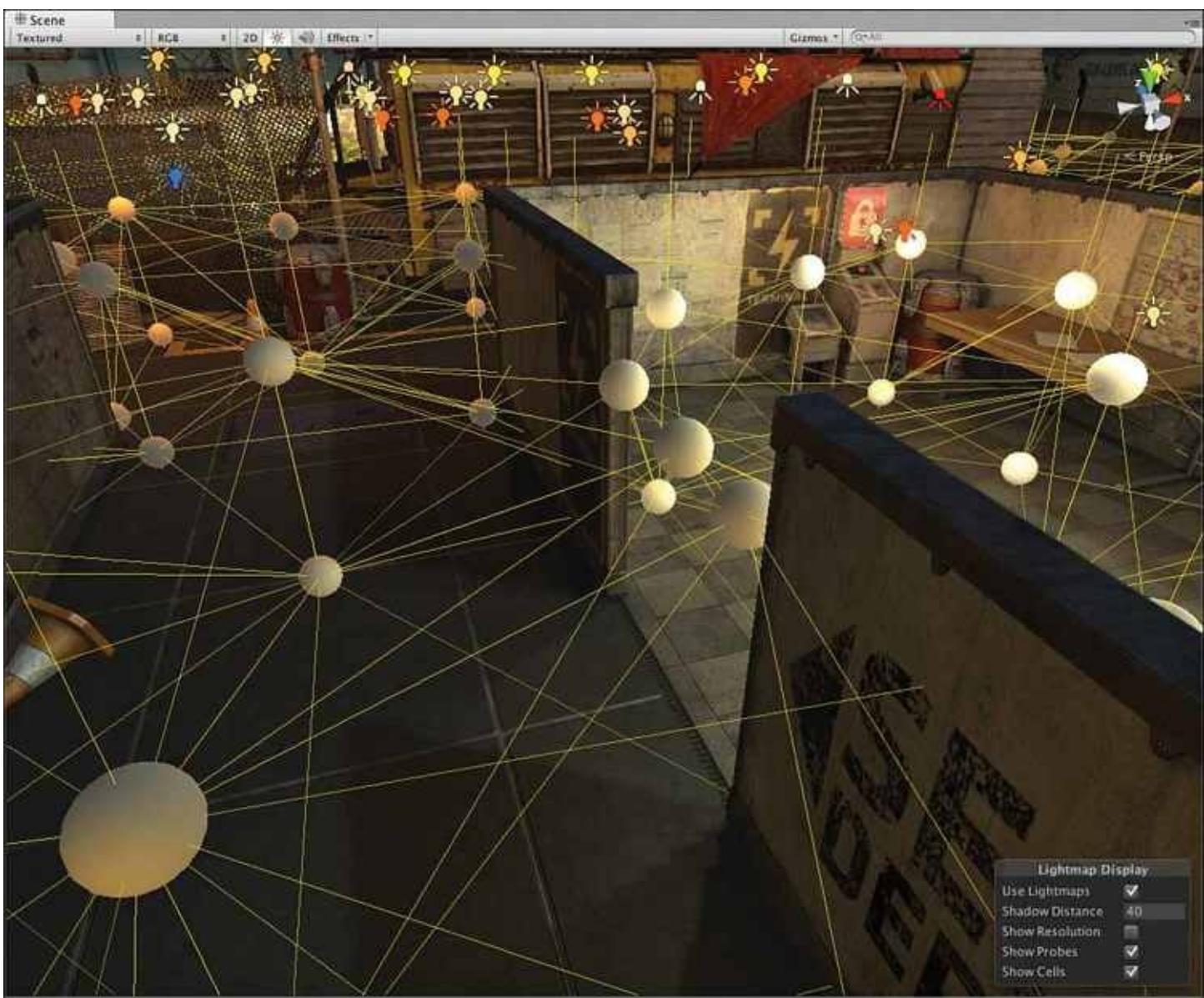


Figure 34.2 Network of light probes in the scene (Unity Pro feature)

Baking Lightmaps

As you can see in the preceding figures, lightmapping can create incredibly realistic scenes that will still render at real-time speeds. Unfortunately for you as the developer, this means that all the processing savings seen by the player are passed back to you. Therefore, lightmapping can take hours to complete on a complex scene, and the baking for the small scene in this prototype could take you from 30 minutes to a few hours, during which you can still work in Unity, but you can't quit, and you can't play the scene. Please make sure that you have sufficient time for your computer to do the baking before starting the process.

To implement lightmapping in your prototype, choose *Window > Lightmapping* from the menu bar and click the *Bake* button at the top. The settings you choose will be different for Unity Pro and Unity Free, as shown in [Figure 34.3](#). You can choose to use the Unity Free settings in Unity Pro to speed the baking process (at the expense of better visuals).

Unity FreeSettings



Unity Pro Settings

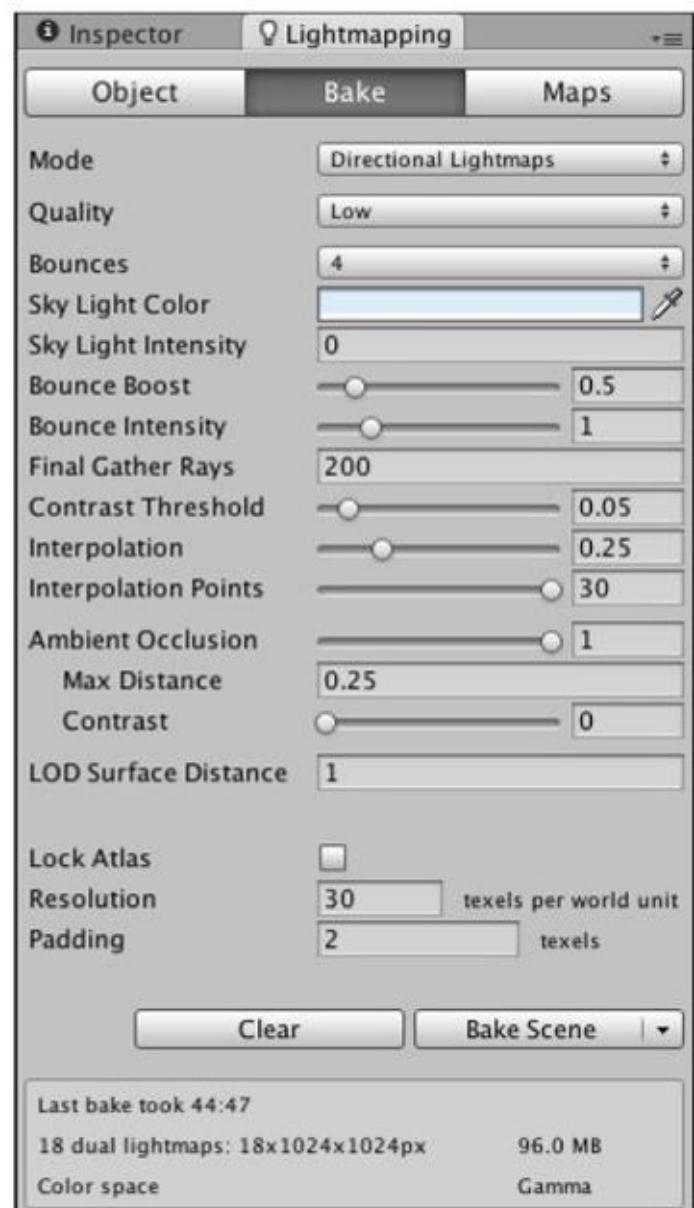


Figure 34.3 Lightmapping settings for Beast in Unity Free and Unity Pro

In both cases, you have two options to choose from for the quality of your lightmaps: low quality will generate shadows that have obvious jagged edges, but it will bake much faster; whereas high quality will produce better shadows and lighting but will take much longer to bake. I highly recommend setting the Quality pop-up menu to *Low* for your first bake. That will let you know how long it will take, and you can multiply that by at least four when you're estimating the time for the high-quality bake. Enter the rest of the settings shown in [Figure 34.3](#) based on whether you want to use single lightmaps or the directional lightmaps available in Unity Pro.

After you've entered these settings, you can click the *Bake Scene* button at the bottom of the Lightmapping pane (if you don't see *Bake Scene* on that button, you may need to choose it from the attached pop-up menu triangle) and get a coffee or maybe get yourself a snowcone; you may be waiting for a while. Otherwise, you can choose to continue working on this prototype and just bake the scene when you take a break. I tend to bake scenes overnight when I can, but if you have a fast machine, it may only take 10 to 20 minutes for single lightmaps on low quality. When the baking is processing, a progress bar will appear in the bottom-right corner of the Unity window and the *Bake Scene* button will be replaced by a *Cancel* button. As mentioned before, you can still work in Unity while the scene is baking, but some of the things you can do are limited.

Quality Settings

The kind of tricks we're using to improve the look of the game are highly dependent on the Unity quality settings that are chosen by the player, but as a developer, you can choose the default quality settings for your games. From the Unity menu bar, choose *Edit > Project Settings > Quality*. This will open the *QualitySettings* pane in the Inspector. By default, there are six quality levels listed, with *Good* selected (as is shown by the darker highlighting of the *Good* row). When a quality level is selected, the bottom half of the Inspector shows the values for that quality level. Underneath the list of quality levels are pop-up triangles that allow you to set the default quality level on a specific platform. For now, the only platform we care about is *PC, Mac & Linux Standalone*, which is represented by the second column. Click the default setting triangle (circled in black in [Figure 34.4](#)) at the bottom of the second column, and choose *Fantastic*. This will change the color of the check box for *Fantastic* in the second column to green, as is shown in [Figure 34.4](#).

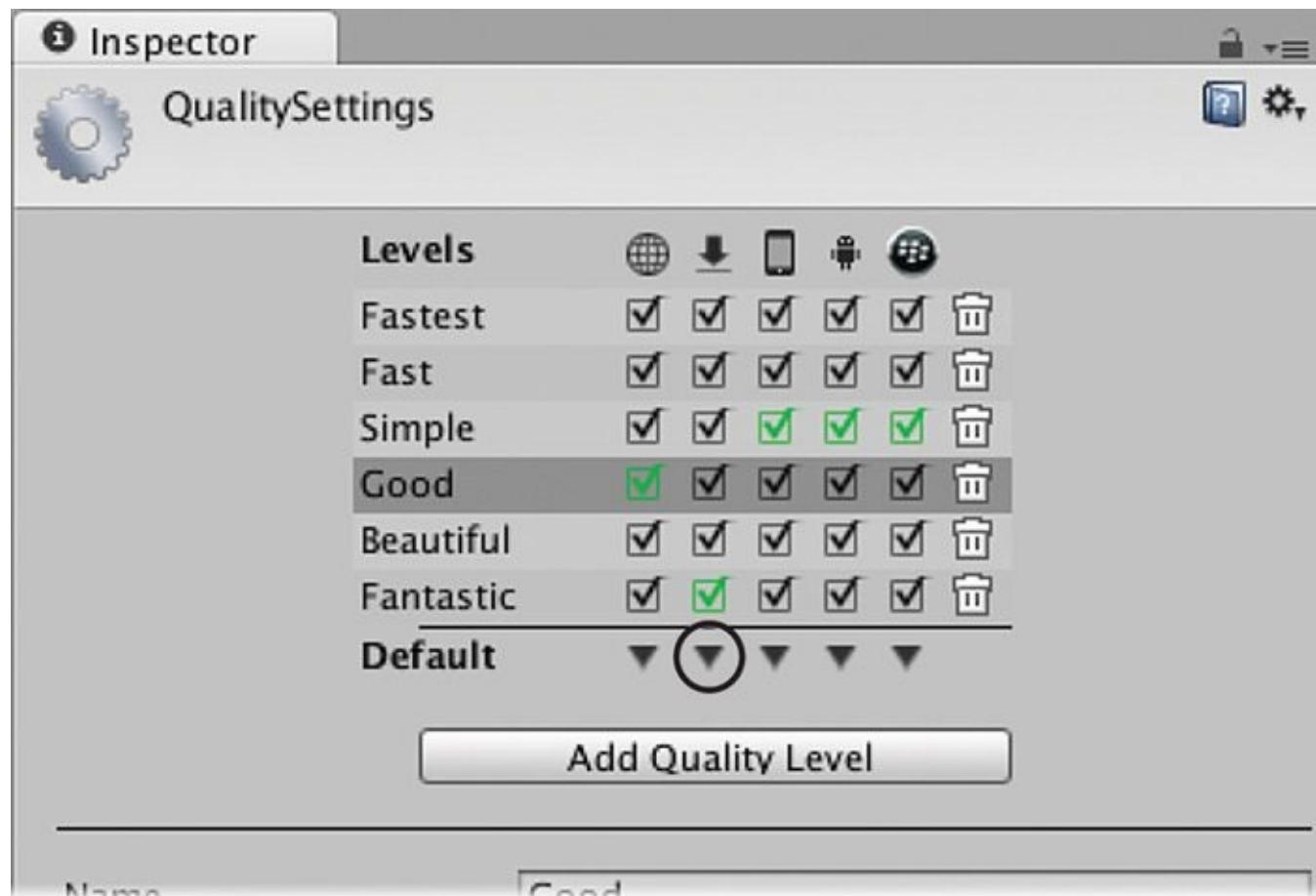


Figure 34.4 Quality settings in the Inspector with *Fantastic* as the default for standalone builds

Creating the *_TargetCamera*

In this game, we'll use a *_TargetCamera* to show the player the shot that she's trying to mimic. Before doing so, let's change the name of the *Main Camera* inside of *_FPC* for clarity's sake. Open the disclosure triangle next to *_FPC* and rename *Main Camera* to *FPCamera*. If you're using Unity Pro, look in the *FPCamera* Inspector under the *Camera* component and set the *Rendering Path* to *Deferred Lighting*. (If you're using Unity Free, leave the *Rendering Path* set to *Use Player Settings*.)

Next, choose *GameObject > Create Other > Camera* from the menu bar. Rename this new camera to *_TargetCamera* and select it in the Hierarchy. In the *_TargetCamera* Inspector pane, click the gear for the *Audio Listener* component and choose *Remove Component*. (Unity only allows one *Audio Listener* in the scene, and *FPCamera* already has one.) Set the *_TargetCamera* transform to the following:

_TargetCamera (Camera)

P:[0,1,0]

R:[0,0,0]

S:[1,1,1]

Right now, the images from the two cameras in the scene are directly on top of each other in the Game pane. In the *Camera* component of the *_TargetCamera* Inspector, set the *Depth* to 1. This will layer the image from *_TargetCamera* above that from *FPCamera*. Then, set the *Viewport Rect* to [x:0, y:0.8, w:0.2, h:0.2], which will shrink the image from *_TargetCamera* to the top-left corner of the screen. If you are using Unity Pro, set the *Rendering Path* of *_TargetCamera* to *Deferred Lighting* as you did for *FPCamera* (or *Use Player Settings* in Unity Free).

GUIs, Layers, and Cameras (Oh My!)

The layers you've used before for physics calculations can also be used to cause various objects to render exclusively to one camera or the other.

1. First, we'll make a GUI component to demonstrate this. Inside the Textures folder in the Project pane you'll find a texture named *_Crosshairs*. Select it and choose *GameObject > Create Other > GUI Texture* from the menu bar. (Selecting *_Crosshairs* ahead of time causes the new GUI Texture to be made from it.) Set the *Color* of *_Crosshairs* to [r:64, g:64, b:64, a:128].

Notice that the *_Crosshairs* appear in both cameras, and their size on screen is not scaled to the display size of the camera. This is where layers come in.

2. From the menu bar, choose *Edit > Project Settings > Tags and Layers*. The list of Layers should already be disclosed, but if not, click the triangle next to it. Type *FPCamera* into User Layer 8 and *_TargetCamera* into User Layer 9.

3. We want the *_Crosshairs* to only appear on the *FPCamera*, so select *_Crosshairs* in the Hierarchy and use the Inspector to set its layer to *FPCamera*.

4. Now, select *_TargetCamera* in the Hierarchy. The pop-up menu for *Culling Mask* in the Camera component of *_TargetCamera* is used to exclude certain layers from being rendered to the camera. Click the pop-up menu (that currently says *Everything*) and select *FPCamera* to toggle the check mark (turning it off). The text on the pop-up menu will change to *Mixed...* to show that some layers are disabled, and the *_Crosshairs* image will disappear from *_TargetCamera*.

5. Select *FPCamera* and disable the layer *_TargetCamera* in its culling mask. This will make the *_TargetCamera* layer work in the opposite manner.

When you finish, the camera *FPCamera* should not show the layer *_TargetCamera*, and the camera *_TargetCamera* should not show the layer *FPCamera*.

Additional GUI Elements

As you can see in [Figure 34.5](#), we want to have several GUI elements in the scene.

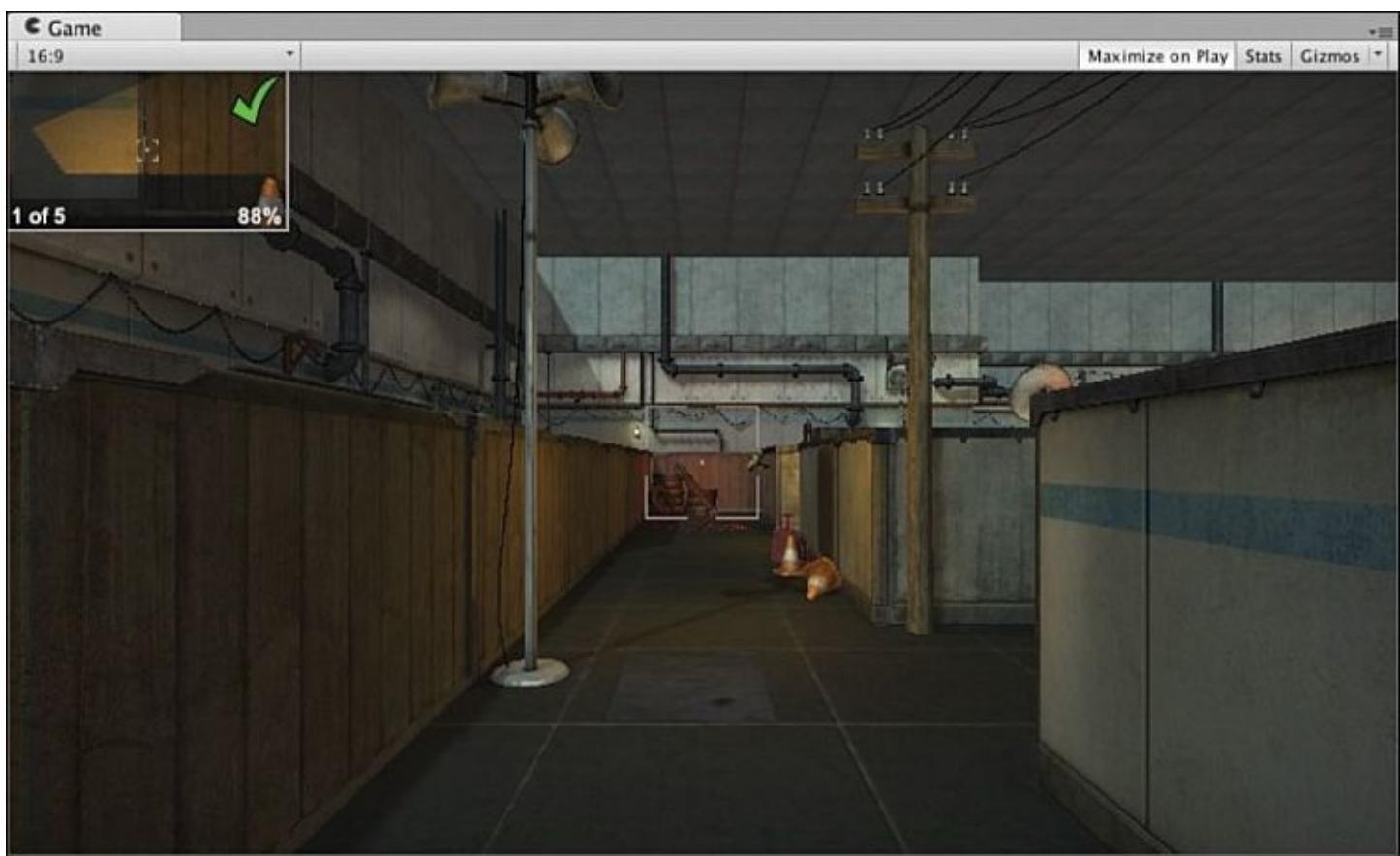


Figure 34.5 Scene showing various GUI elements

There are two `GUIText`s that need to be added to the scene. Create two `GUI Text`s (`GameObject > Create Other > GUI Text` from the menu bar). Name one `ShotCounter` and the other `ShotRating`. In each, set the values shown in [Figure 34.6](#). Be sure to set the *Layer* as well.

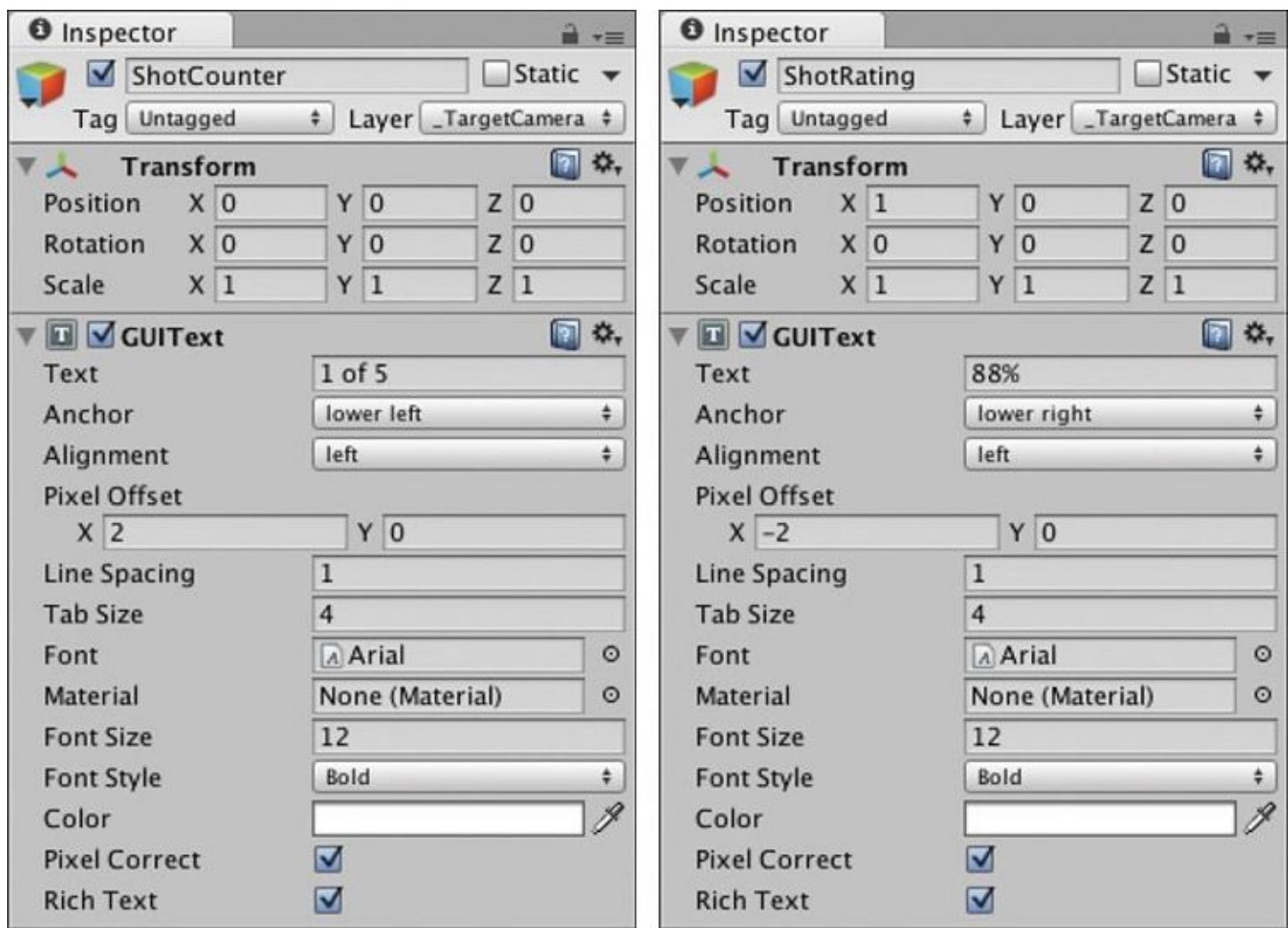


Figure 34.6 GUIText settings for ShotCounter and ShotRating

This should place two bits of text in the left and right bottom corners of the `_TargetCamera` image in the Game pane (as shown in [Figure 34.5](#)).

In the Textures folder of the Project pane, you'll find the following textures. Select each in turn and then make a new GUI Texture from it (*GameObject > Create Other > GUI Texture*):

- `_Check_64`
- `_Crosshairs_12`
- `_White`

Once these are in the Hierarchy, give them the settings in [Figure 34.7](#).

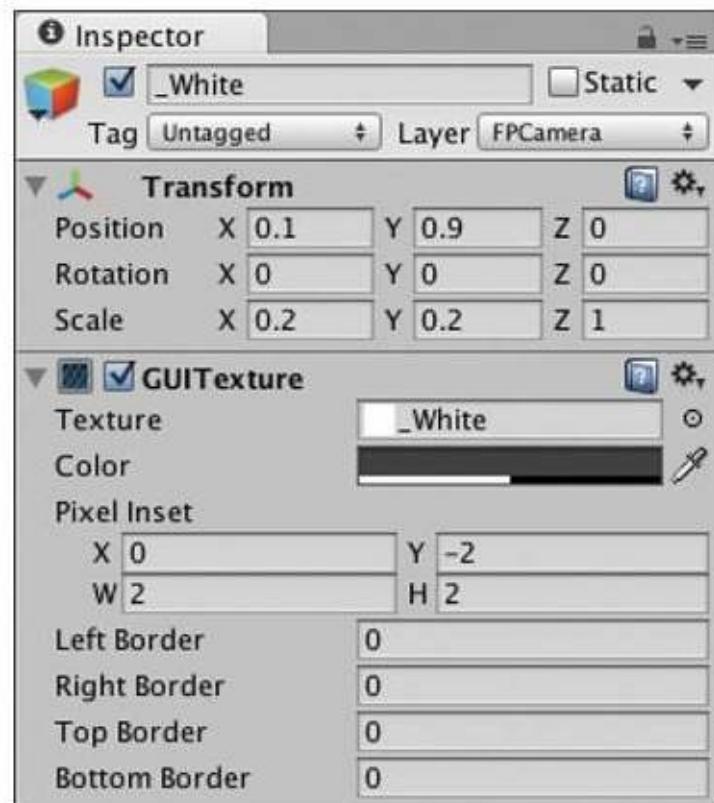
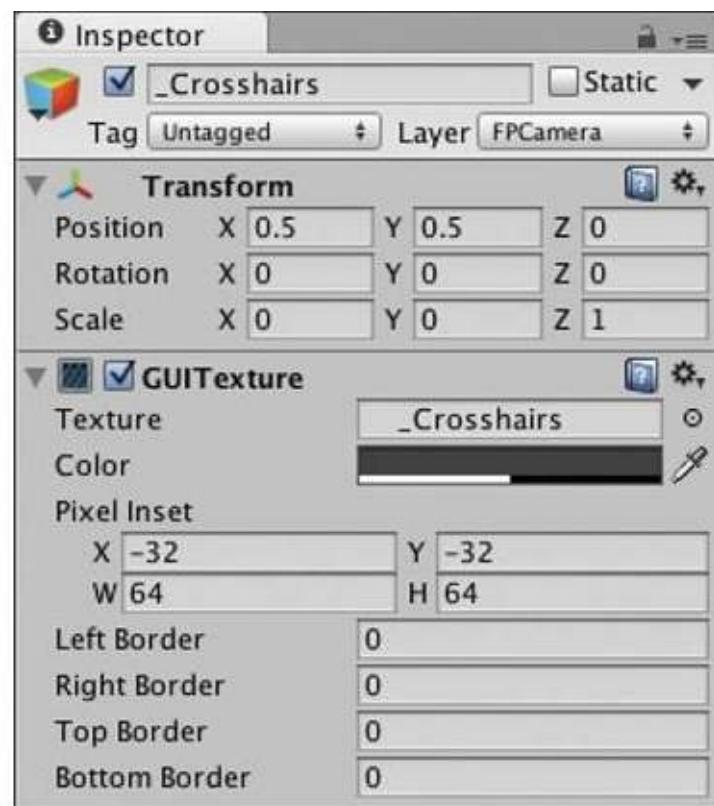
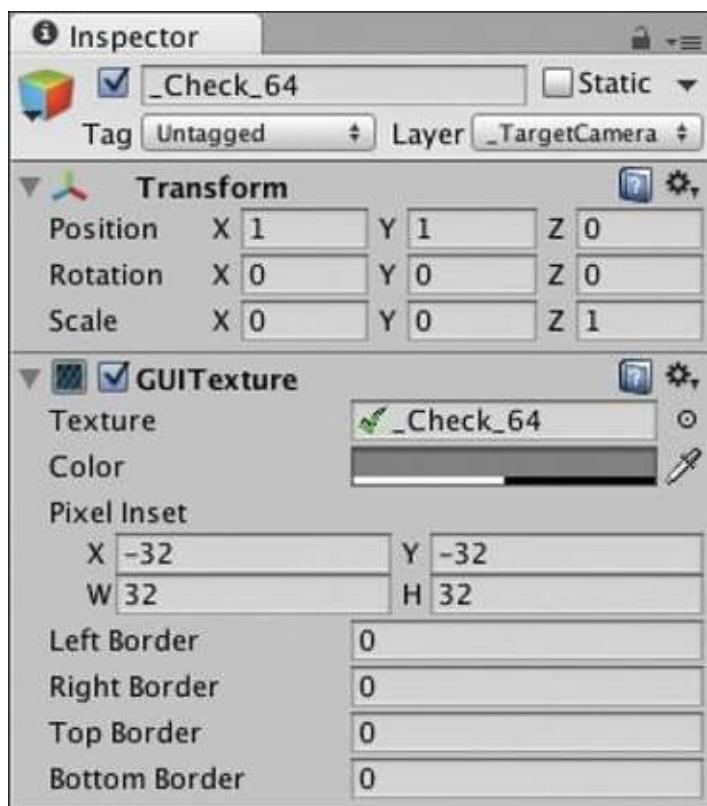


Figure 34.7 GUITexture settings for **_Check_64**, **_Crosshairs**, **_Crosshairs_12**, and **_White**

Again, be sure to set the *Layer* properly. Also, set the color of **_Crosshairs_12** to [r:64, g:64, b:64, a:128] like **_Crosshairs**. Your Game window should now look like that shown in [Figure 34.5](#).

Coding the Game

With the experience you now have, coding this game will actually seem pretty simple, but like the others, you'll learn things here that can serve as a foundation for further projects.

1. Create a new C# script named *Shot* in the **_Scripts** folder. Open it and replace all the default text in the *Shot* class with the following code:

[Click here to view code image](#)

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

[System.Serializable] // Make this visible in the Inspector
public class Shot { // Shot does not extend MonoBehaviour

    public Vector3 position; // Position of the Camera
    public Quaternion rotation; // Rotation of the Camera
    public Vector3 target; // Where the Camera is pointed

}
```

This Shots class will keep track of the information about the photographic shots taken in the game. Because we want the game to be able to work with Unity Free, we can't record the images from a camera to an image (which is a Unity Pro-only feature), but we can still record the position and orientation of the camera (and the position of its target as well). Shot will handle this for us.

2. Create a second new C# script named *TargetCamera*, place it in the *_Scripts* folder, and attach it to *_TargetCamera* in the Hierarchy. The TargetCamera class will be responsible for both creating the list of shots to be taken (in an edit mode) and checking the player's shots against the target shots in play mode. Enter the following code:

[Click here to view code image](#)

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

public class TargetCamera : MonoBehaviour {

    public GameObject fpCamera; // First-person Camera
    public bool _____;

    void Update () {
        Shot sh;

        // Mouse Input
        if (Input.GetMouseButtonDown(0)) { // Left mouse button
            sh = new Shot();
            // Grab the position and rotation of fpCamera
            sh.position = fpCamera.transform.position;
            sh.rotation = fpCamera.transform.rotation;
            // Shoot a ray from the camera and see what it hits
            Ray ray = new Ray(sh.position, fpCamera.transform.forward);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit) ) {
                sh.target = hit.point;
            }

            // Position _TargetCamera with the Shot
            ShowShot(sh);
        }
    }

    public void ShowShot(Shot sh) {
        // Position _TargetCamera with the Shot
        transform.position = sh.position;
```

```
        transform.rotation = sh.rotation;
    }
}
```

3. Assign the FPCamera child of _FPC in the Hierarchy to the fpCamera field of TargetCamera in the Inspector for _TargetCamera.

4. Save your scene!

Press Play, and you should see that when you click the left mouse button, the _TargetCamera image shifts to show the shot you took when you clicked. Now that we know this works, we need a way to store the shots from one play session to the next.

Using PlayerPrefs to Store Shots

As we saw in the *Apple Picker* prototype, PlayerPrefs is a great place to store things long term; however, it can only store floats, ints, and strings. To handle this, we'll convert the Shot into XML and then we can read it back in using the PT_XMLReader (that was used in the card game prototypes of [Chapter 31](#), “[Prototype 4: Prospector Solitaire](#),” and [Chapter 32](#), “[Prototype 5: Bartok](#)”) when we start the game the next time:

1. The first step in doing so is converting each Shot into XML. Add the following code to Shot:

[Click here to view code image](#)

```
public class Shot { // Shot does not extend MonoBehaviour

    public Vector3 position; // Position of the Camera
    public Quaternion rotation; // Rotation of the Camera
    public Vector3 target; // Where the Camera is pointed

    // Generates a single-line <shot> entry for an XML document
    public string ToXML() {
        string ss = "<shot ";
        ss += "x=\"" + position.x + "\" ";
        ss += "y=\"" + position.y + "\" ";
        ss += "z=\"" + position.z + "\" ";
        ss += "qx=\"" + rotation.x + "\" ";
        ss += "qy=\"" + rotation.y + "\" ";
        ss += "qz=\"" + rotation.z + "\" ";
        ss += "qw=\"" + rotation.w + "\" ";
        ss += "tx=\"" + target.x + "\" ";
        ss += "ty=\"" + target.y + "\" ";
        ss += "tz=\"" + target.z + "\" ";
        ss += " />";

        return(ss);
    }
}
```

- 1.1. In this line and those that follow, you see \" inside of the double quotes that define a string literal (for example, "x=\"\"). The backslash (\) is used as an *escape character* in this situation, and as such, it causes the character that immediately follows it to be interpreted as part of the string literal, regardless of what character it is. Normally, a double quote would end the string literal, but with the backslash before it, a double quote is added to the string literal instead.

2. And add a line to TargetCamera to test it:

[Click here to view code image](#)

```

public class TargetCamera : MonoBehaviour {
    ...
    void Update () {
        ...
        if (Input.GetMouseButtonDown(0)) { // Left mouse button
            ...
            ShowShot( sh );
            Utils.tr( sh.ToXML() );
        }
    }
    ...
}

```

3. Press Play and take some shots. You should see output in the Console window that is formatted like the following:

[Click here to view code image](#)

```

<shot
x="-9.014837" y="1.457083" z="24.45312" qx="0.02179807" qy="0.0392502" qz="-0.0008564426"

```

This lets you know that it's working properly.

4. Now, expand this further by adding a static public List<Shot> to Shot so that it can keep track of multiple shots at a time. We'll also add the ability to read and write XML to Player-Prefs now. Add the following to Shot:

[Click here to view code image](#)

```

public class Shot { // Shot does not extend MonoBehaviour
    static public List<Shot> shots = new List<Shot>(); // List of all shots
    static public string prefsName = "QuickSnap_Shots";
    public Vector3 position; // Position of the Camera

    ...
    public string ToXML() {
        ...
    }

    // Takes a PT_XMLHashtable from PT_XMLReader of a <shot> entry in XML and
    // parses it into a Shot
    static public Shot ParseShotXML( PT_XMLHashtable xHT ) {
        Shot sh = new Shot();

        sh.position.x = float.Parse(xHT.att("x"));
        sh.position.y = float.Parse(xHT.att("y"));
        sh.position.z = float.Parse(xHT.att("z"));
        sh.rotation.x = float.Parse(xHT.att("qx"));
        sh.rotation.y = float.Parse(xHT.att("qy"));
        sh.rotation.z = float.Parse(xHT.att("qz"));
        sh.rotation.w = float.Parse(xHT.att("qw"));
        sh.target.x = float.Parse(xHT.att("tx"));
        sh.target.y = float.Parse(xHT.att("ty"));
        sh.target.z = float.Parse(xHT.att("tz"));

        return( sh );
    }

    // Loads all of the Shots from PlayerPrefs
    static public void LoadShots() {
        // Empty the shots List<Shot>
        shots = new List<Shot>();
    }
}

```

```

if (!PlayerPrefs.HasKey(prefsName)) {
    // If there are no shots, return
    return;
}

// Get the full XML and parse it
string shotsXML = PlayerPrefs.GetString(prefsName);
PT_XMLReader xmlr = new PT_XMLReader();
xmlr.Parse(shotsXML);

// Pull the PT_XMLHashList of all <shot>s
PT_XMLHashList hl = xmlr.xml["xml"][0]["shot"];
for (int i=0; i<hl.Count; i++) {
    // Parse each <shot> in the PT_XMLHashlist into a Shot
    PT_XMLHashtable ht = hl[i];
    Shot sh = ParseShotXML(ht);
    // Add it to the List<shot> shots
    shots.Add(sh);
}
}

// Save List<Shot> shots to PlayerPrefs
static public void SaveShots() {
    string xs = Shot.XML;

    Utils.tr(xs); // Trace all the XML to the Console

    // Set the PlayerPrefs
    PlayerPrefs.SetString(prefsName, xs);

    Utils.tr("PlayerPrefs."+prefsName+" has been set.");
}

// Convert all Shot.shots to XML
static public string XML {
    get {
        // Start an XML string
        string xs = "<xml>\n";
        // Add each of the Shots as a <shot> in XML
        foreach( Shot sh in shots ) {
            xs += sh.ToXML()+"\n";
        }
        // Add the closing XML tag
        xs += "</xml>";
        return(xs);
    }
}
}
}

```

5. Next, you need to add some code to the TargetCamera class to take advantage of this:

[Click here to view code image](#)

```

public class TargetCamera : MonoBehaviour {
    public bool           editMode = true;
    public GameObject     fpCamera; // First-person Camera

    public bool           _____;

    public int            shotNum;
    public GUIText        shotCounter, shotRating;
    public GUITexture     checkMark;
}

```

```

void Start() {
    // Find the GUI components
    GameObject go = GameObject.Find("ShotCounter");
    shotCounter = go.GetComponent<GUIText>();
    go = GameObject.Find("ShotRating");
    shotRating = go.GetComponent<GUIText>();
    go = GameObject.Find("_Check_64");
    checkMark = go.GetComponent<GUITexture>();
    // Hide the checkMark
    checkMark.enabled = false;

    // Load all the shots from PlayerPrefs
    Shot.LoadShots();
    // If there were shots stored in PlayerPrefs
    if (Shot.shots.Count>0) {
        shotNum = 0;
        ShowShot(Shot.shots[shotNum]);
    }

    // Hide the cursor (Note: this doesn't work in the Unity Editor unless
    // the Game pane is set to Maximize on Play.)
    Screen.showCursor = false;
}

void Update () {
    ...
    if (Input.GetMouseButtonDown(0)) { // Left mouse button
        ...
        Utils.tr( sh.ToXML() );
        // Record a new shot
        Shot.shots.Add(sh);
        shotNum = Shot.shots.Count-1;
    }

    // Keyboard Input
    // Use Q and E to cycle Shots
    // Note: Either of these will throw an error if Shot.shots is empty.
    if (Input.GetKeyDown(KeyCode.Q)) {
        shotNum--;
        if (shotNum < 0) shotNum = Shot.shots.Count-1;
        ShowShot(Shot.shots[shotNum]);
    }
    if (Input.GetKeyDown(KeyCode.E)) {
        shotNum++;
        if (shotNum >= Shot.shots.Count) shotNum = 0;
        ShowShot(Shot.shots[shotNum]);
    }
    // If in editMode & Left Shift is held down...
    if (editMode && Input.GetKey(KeyCode.LeftShift)) {
        // Use Shift-S to Save
        if (Input.GetKeyDown(KeyCode.S)) {
            Shot.SaveShots();
        }
        // Use Shift-X to output XML to Console
        if (Input.GetKeyDown(KeyCode.X)) {
            Utils.tr(Shot.XML);
        }
    }

    // Update the GUITexts
    shotCounter.text = (shotNum+1).ToString()+" of "+Shot.shots.Count;
    if (Shot.shots.Count == 0) shotCounter.text = "No shots exist";
}

```

```
// ^ Shot.shots.Count doesn't require .ToString() because it is assumed
// when the left side of the + operator is a string
shotRating.text = ""; // This line will be replaced later
```

```
}
```

```
...
```

6. Now, upon pressing Play, you will see that no shots yet exist. Walk around the scene and use the left mouse button to take about eight interesting shots. You can switch between shots by tapping the *Q* and *E* keys.

Players will find the shots easier to match if you give them a specific object to focus on and if you give them some hints about framing (like objects that line up with the sides or corners of the image). After you have some shots that you like, press Shift+S (using the left Shift key), and the Console window should tell you that you have saved the shots to PlayerPrefs. Press the Play button to stop playback, and then press it again to start the game anew. You'll see that this second time it properly loads all of the shots from Player Prefs.

[Figure 34.8](#) shows the eight shots that I took.



Figure 34.8 Eight shots from around the scene

7. Save your Scene. Remember, *always* save your scene.

Two Uses of `OnDrawGizmos()`

Gizmos are the various on-screen icons and tools that you see in Unity's Scene pane. These include the axes gizmo (that shows the orientation of the Scene view) and all of the icons for lights and such. Gizmos can also be used as a debug tool by allowing developers to draw simple shapes using them. We'll add a `DEBUG` bool to `TargetCamera`, and if it's checked, `OnDrawGizmos()` will draw information about each shot in the Scene pane.

Add the following code to the end of the `TargetCamera` class:

[Click here to view code image](#)

```
public class TargetCamera : MonoBehaviour {
    ...
    // OnDrawGizmos() is called ANY time Gizmos need to be drawn, even when
    // Unity isn't playing!
    public void OnDrawGizmos() {
        List<Shot> shots = Shot.shots;
        for (int i=0; i<shots.Count; i++) {
            Gizmos.color = Color.green;
            Gizmos.DrawWireSphere(shots[i].position, 0.5f);
            Gizmos.color = Color.yellow;
            Gizmos.DrawLine( shots[i].position, shots[i].target );
```

```
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(shots[i].target, 0.25f);
    }
}
```

Now, when you play the scene, you should see little gizmos pop up in the Scene pane like the ones shown in [Figure 34.9](#) showing a green wireframe sphere where the camera was positioned when the shot was taken and a smaller red sphere where the `Physics.Raycast()` from the camera hit an object. If you see a shot that has a red sphere back at the point 0,0,0 (in the bottom-right corner of the map near where `_FPC` starts), that means that the `Physics.Raycast()` didn't hit anything, and that shot should be replaced.



Figure 34.9 Gizmos showing the location of the eight shots from around the scene. Note that the image has been rotated 90° clockwise to better fit the page.

Speaking of replacing shots, right now, we don't have a way to delete the entry in PlayerPrefs. To do this, we'll use another feature of `OnDrawGizmos()`. As mentioned in the code listing comments, `OnDrawGizmos()` is called any time that the scene draws gizmos, so we can take advantage of that to run a little code even when Unity is not playing.

1. Add the following `DeleteShots()` method to the end of the `Shot` class:

[Click here to view code image](#)

```
public class Shot { // Shot does not extend MonoBehaviour
    ...
    // Delete Shots from Shot.shots and PlayerPrefs
    static public void DeleteShots() {
        shots = new List<Shot>();
```

```

        if (PlayerPrefs.HasKey(prefsName)) {
            PlayerPrefs.DeleteKey(prefsName);
            Utils.tr("PlayerPrefs."+prefsName+" has been deleted.");
        } else {
            Utils.tr("There was no PlayerPrefs."+prefsName+" to delete.");
        }
    }
}

```

2. Next, add a `checkToDeletePlayerPrefs` bool field to the top of `TargetCamera` and the bold lines to the end of `TargetCamera.OnDrawGizmos()`:

[Click here to view code image](#)

```

public class TargetCamera : MonoBehaviour {
    ...

    public GameObject fpCamera; // First-person Camera

public bool checkToDeletePlayerPrefs = false;

    public bool _____;
    ...

    // OnDrawGizmos() is called ANY time Gizmos need to be drawn, even when
    // Unity isn't playing!
    public void OnDrawGizmos() {

        // If checkToDeletePlayerPrefs is checked
        if (checkToDeletePlayerPrefs) {
            Shot.DeleteShots(); // Delete all the shots
            // Uncheck checkToDeletePlayerPrefs
            checkToDeletePlayerPrefs = false;
            shotNum = 0; // Set shotNum to 0
        }
    }
}

```

Now, even when Unity is not playing, if you check the box in the Inspector next to the `TargetCamera` field `checkToDeletePlayerPrefs`, Unity will run the code to delete the prefs from `PlayerPrefs`, notify you that it has been done in the console, and then uncheck `checkToDeletePlayerPrefs` in the Inspector. You need to be very careful if you choose to use this kind of feature, but it can be really useful in situations like this.

If you tested `checkToDeletePlayerPrefs` (as you should), you now need to go take some more shots and save them before continuing.

Replacing Individual Shots

Now, you can delete all the shots together, but what if you want to just replace a single shot?

1. Add the following static method to the end of the `Shot` class:

[Click here to view code image](#)

```

public class Shot { // Shot does not extend MonoBehaviour
    ...

    // Replace the shot
    static public void ReplaceShot(int ndx, Shot sh) {
        // Make sure there's a Shot at that index to replace
        if (shots==null || shots.Count <= ndx) return;
        // Remove the old Shot

```

```

shots.RemoveAt(ndx);
// List<>.Insert() adds something to the list at a specific index
shots.Insert(ndx, sh);

        Utils.tr("Replaced shot:", ndx, "with", sh.ToXML());
    }
}

```

2. Now make the changes to TargetCamera shown here. This will involve making several changes to the // Mouse Input section of the code so I have included that entire section:

[Click here to view code image](#)

```

public class TargetCamera : MonoBehaviour {
    ...

    void Update () {
        Shot sh;

        // Mouse Input
        // If Left or Right mouse button is pressed this frame...
        if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)) {
            sh = new Shot();
            // Grab the position and rotation of fpCamera
            sh.position = fpCamera.transform.position;
            sh.rotation = fpCamera.transform.rotation;
            // Shoot a ray from the camera and see what it hits
            Ray ray = new Ray(sh.position, fpCamera.transform.forward);
            RaycastHit hit;
            if ( Physics.Raycast(ray, out hit) ) {
                sh.target = hit.point;
            }

            if (editMode) {
                if (Input.GetMouseButtonDown(0)) {
                    // Left button records a new shot
                    Shot.shots.Add(sh);
                    shotNum = Shot.shots.Count-1;
                } else if (Input.GetMouseButtonDown(1)) {
                    // Right button replaces the current shot
                    Shot.ReplaceShot(shotNum, sh);
                    ShowShot(Shot.shots[shotNum]);
                }
            }
        }

        // Position _TargetCamera with the Shot
        // ShowShot(sh); // Comment out or delete this line
    }

    // Keyboard Input
    ...
}
...
}

```

3. Play the scene and walk around.

Now, as long as `editMode` is checked in the `_TargetCamera` Inspector, you can right-click to replace any of the shots you don't particularly like. Then, when you want to save the updated list of shots, press Shift+S.

Maximizing the Target Window

You may have noticed in your development that it's somewhat difficult to see the Target window at its small size. You can adjust a camera's on-screen window at runtime by changing its Camera.rect value. Try adding the following code to TargetCamera. Note that this will still show the tiny _Crosshairs_12 in the middle of the screen; can you think of a way to hide it when the target window is enlarged?

[Click here to view code image](#)

```
public class TargetCamera : MonoBehaviour {  
    ...  
  
    public bool _____;  
  
    public Rect camRectNormal; // Pulled from camera.rect  
  
    ...  
  
    void Start() {  
        ...  
        Screen.showCursor = false;  
  
        camRectNormal = camera.rect;  
    }  
  
    void Update () {  
        ...  
  
        // Keyboard Input  
        ...  
        if (editMode && Input.GetKey(KeyCode.LeftShift)) { ... }  
        // Hold Tab to maximize the Target window  
        if (Input.GetKeyDown(KeyCode.Tab)) {  
            // Maximize when Tab is pressed  
            camera.rect = new Rect(0,0,1,1);  
        }  
        if (Input.GetKeyUp(KeyCode.Tab)) {  
            // Return to normal when Tab is released  
            camera.rect = camRectNormal;  
        }  
    }  
}
```

Now it's time to add the mode that players will see.

Comparing Shots

Because we're asking players to take the same shots as we have, it's critical to write a good Shot comparison function to compare the player's Shot to the target Shot. The method we will use compares both the location of the camera when taking the Shot and the location of the Physics.Raycast() hit that is generated by the ray shooting from the camera. We will not compare the rotation of the two Shots because if the player is not standing in exactly the right position, she will have to rotate differently from the original Shot to get the correct objects in view. This is shown in [Figure 34.10](#).

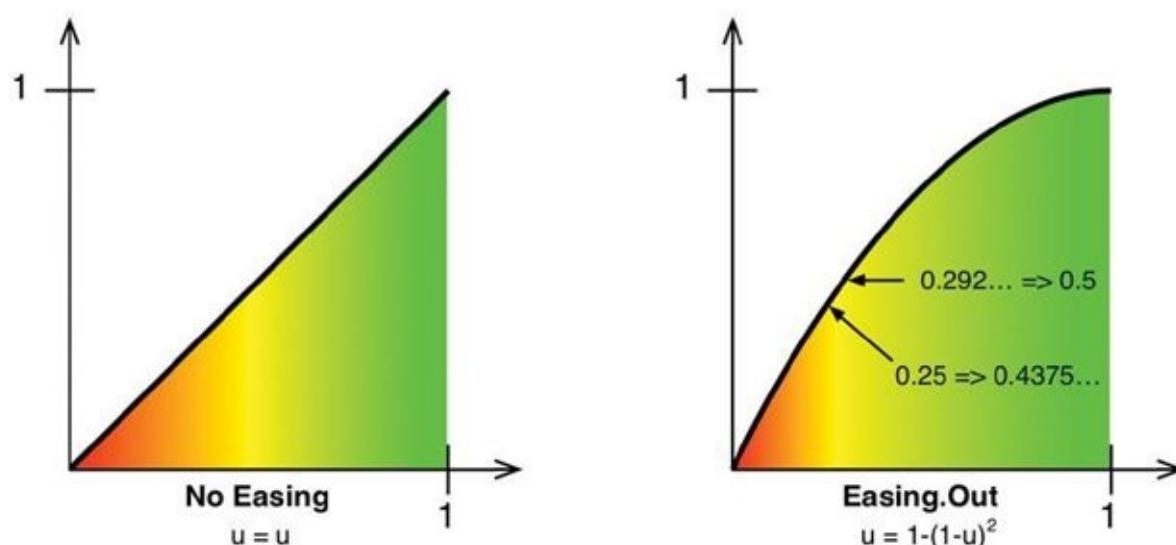
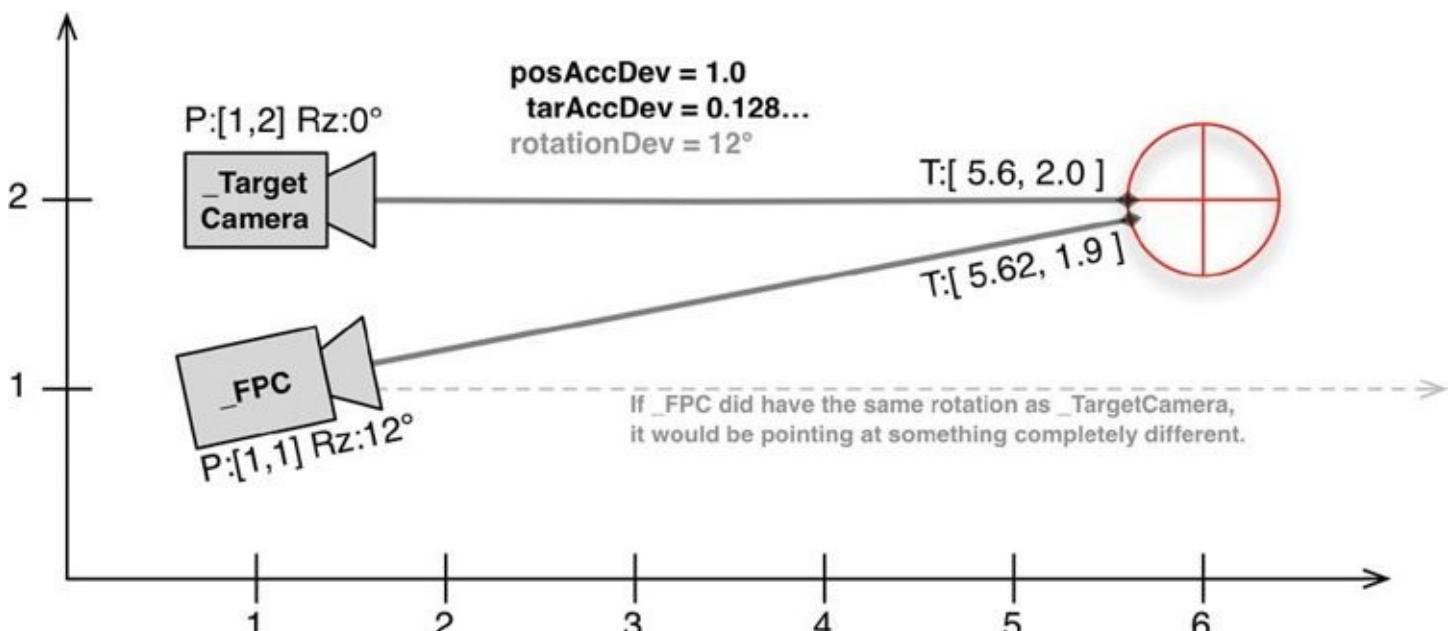


Figure 34.10 A 2D demonstration of why we are comparing camera position and target position rather than camera rotation. Also shows the effects of using Easing.Out on accuracy percentage.

You'll see in the following code that we also make use of the `Easing.Ease()` function to perform an `Easing.Out` on the accuracy percentage before it is returned. While `Easing` is most often used for interpolation and movement, it can be used to curve or alter any number that tends to sit between 0 and 1 (like the percentage accuracy that we're calculating). As shown in [Figure 34.10](#), using `Easing.Out` (which is the function $u = 1 - (1 - u)^2$) will curve the results as shown in the bottom right of the figure. This will make a larger area get a good percentage score while still dropping the percentage to 0 at the same point as the original, accommodating the extreme difficulty in getting the camera in the exact right position while still limiting the radius of the deviation from the correct position to `maxPosDeviation` and `maxTarDeviation`. I personally use easing a *lot* in my game design work. You can read more about it in the “[Interpolation](#)” section of [Appendix B](#), “[Useful Concepts](#).”

Add the following static method to the end of the `Shot` class to compare the two shots. This code is going to reference some fields that we'll set on `_TargetCamera` in the Inspector, so later we'll add a singleton to `TargetCamera` to make this easy:

[Click here to view code image](#)

```
public class Shot { // Shot does not extend MonoBehaviour
```

```

...
// Compare two Shots. 1 is a perfect match, while <0 is not valid
public static float Compare(Shot target, Shot test) {
    // Get the positinal deviation of both the camera and the Raycast hit
    float posDev = (test.position - target.position).magnitude;
    float tarDev = (test.target - target.target).magnitude;

    float posAccPct, tarAccPct, posAP2, tarAP2; // Accuracy percentages
    TargetCamera tc = TargetCamera.S;

    // Get a value for accuracy where 1 is perfect and 0 is barely ok
    posAccPct = 1-(posDev/tc.maxPosDeviation);
    tarAccPct = 1-(tarDev/tc.maxTarDeviation);

    // Curve the value so that it's more forgiving. This uses the same
    // Easing that we do for motion. You can curve ANY value between
    // 0 and 1, not just Interpolation values.
    posAP2 = Easing.Ease(posAccPct, tc.deviationEasing);
    tarAP2 = Easing.Ease(tarAccPct, tc.deviationEasing);

    float accuracy = (posAP2+tarAP2)/2f;

    // Remember that you can use Utils to format numbers nicely as strings
    string accText = Utils.RoundToPlaces(accuracy*100).ToString()+"%";
    Utils.tr("Position:",posAccPct,posAP2,"Target:",tarAccPct,tarAP2,"Accuracy",accText);

    return(accuracy);
}
}

```

Now, add the bolded code to TargetCamera to make use of the Slot.Compare() function:

[Click here to view code image](#)

```

public class TargetCamera : MonoBehaviour {
    static public TargetCamera S;

    public bool editMode = true;
    public GameObject fpCamera; // First-person Camera
    // Maximum deviation in Shot.position allowed
    public float maxPosDeviation = 1f;
    // Maximum deviation in Shot.target allowed
    public float maxTarDeviation = 0.5f;
    // Easing for these deviations
    public string deviationEasing = Easing.Out;

    public bool checkToDeletePlayerPrefs = false;
    public bool _____;

    public Rect camRectNormal; // Pulled from camera.rect
    public int shotNum;
    public GUIText shotCounter, shotRating;
    public GUITexture checkMark;
    public Shot lastShot;

    void Awake() {
        S = this;
    }

    ...
}

void Update () {

```

```
if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)) {
    ...
    if (editMode) {
        if (Input.GetMouseButtonDown(0)) {
            // Left button records a new shot
            Shot.shots.Add(sh);
            shotNum = Shot.shots.Count-1;
        } else {
            // Right button replaces the current shot
            Shot.ReplaceShot(shotNum, sh);
            ShowShot(Shot.shots[shotNum]);
        }
        // Reset information about the player when editing shots
        ResetPlayerShotsAndRatings();
    } else {
        // Test this shot against the current Shot
        float acc = Shot.Compare( Shot.shots[shotNum], sh );
        lastShot = sh;
    }
}
...
}

// OnDrawGizmos() is called ANY time Gizmos need to be drawn, even when
// Unity isn't playing!
public void OnDrawGizmos() {
    ...
    // Show the player's last shot attempt
    if (lastShot != null) {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(lastShot.position, 0.25f);
        Gizmos.color = Color.white;
        Gizmos.DrawLine( lastShot.position, lastShot.target );
        Gizmos.color = Color.red;
        Gizmos.DrawSphere(lastShot.target, 0.125f);
    }
}
```

Be sure to set `TargetCamera.editMode` to false in the Inspector for `_TargetCamera`, and then press Play. You should be able to walk around and select various target shots with the Q and E keys. Line yourself up with the target shot and click the left mouse button to take a test shot. The Console pane will show you information on your accuracy, and a new gizmo should appear in the Scene pane that shows the last shot you took. If you are consistently inaccurate on a specific shot, you can use this to see whether it's your position or your aim that is causing the problem. You can then adjust `maxPosDeviation` and `maxTarDeviation` to something that works better for you.

Recording and Displaying Player Progress

Now that shots are being compared properly, it's time to show the player how she is doing in the game. Add the bolded code shown here to TargetCamera:

[Click here to view code image](#)

```
public class TargetCamera : MonoBehaviour {  
    ...
```

```

public string deviationEasing = Easing.Out;
public float passingAccuracy = 0.7f;

...
public Shot lastShot;
public int numShots;
public Shot[] playerShots;
public float[] playerRatings;

...

void Start() {
    ...
    // Load all the shots from PlayerPrefs
    Shot.LoadShots();
    // If there were shots stored in PlayerPrefs
    if (Shot.shots.Count>0) {
        shotNum = 0;
        ResetPlayerShotsAndRatings();
        ShowShot(Shot.shots[shotNum]);
    }
    ...
}

void ResetPlayerShotsAndRatings() {
    numShots = Shot.shots.Count;
    // Initialize playerShots & playerRatings with default values
    playerShots = new Shot[numShots];
    playerRatings = new float[numShots];
}

void Update () {
    ...
    if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)) {
        ...
        if (editMode) {
            if (Input.GetMouseButtonDown(0)) {
                ...
                ShowShot(Shot.shots[shotNum]);
            }
            // Reset information about the player when editing shots
            ResetPlayerShotsAndRatings();
        } else {
            // Test this shot against the current Shot
            float acc = Shot.Compare( Shot.shots[shotNum], sh );
            lastShot = sh;
            playerShots[shotNum] = sh;
            playerRatings[shotNum] = acc;
        }
    }
}

// Update the GUITexts
shotCounter.text = (shotNum+1).ToString()+" of "+Shot.shots.Count;
if (Shot.shots.Count == 0) shotCounter.text = "No shots exist";
// ^ Shot.shots.Count doesn't require .ToString() because it is assumed
// when the left side of the + operator is a string
// shotRating.text = ""; // This line is now commented out

if (playerRatings.Length > shotNum && playerShots[shotNum] != null) {
    float rating = Mathf.Round(playerRatings[shotNum]*100f);
    if (rating < 0) rating = 0;
    shotRating.text = rating.ToString()+"%";
    checkMark.enabled = (playerRatings[shotNum] > passingAccuracy);
    // ^ the > comparison is used to generate true or false

```

```
    } else {
        shotRating.text = "";
        checkMark.enabled = false;
    }
}
```

Now, when you press Play, taking a shot will show you a shot rating in the lower-right corner of the target window, and if that rating is above the `_TargetCamera`'s `passingAccuracy`, a check mark will appear in the upper-right corner as well.

Adding a Snapshot Sound and Visual Effect

To make this seem more like the player is taking pictures, we're going to add a shutter sound and visual effect.

Adding the Audio Source

Select `_TargetCamera` in the Hierarchy, and from the menu bar choose *Component > Audio > Audio Source*. As its name would imply, an Audio Source component generates sound in your scene. In the Audio Source component that has now appeared in the Inspector, click the target icon to the right of the Audio Clip field and choose the only audio clip in our Project Assets: *camera-shutter-click-03*. You will see now that below the Audio Clip field, Unity tells you “This is a 3D Sound.”

Unity has both 3D and 2D sounds. 2D are normal stereo sounds that you're used to dealing with (like when listening to music). 2D sounds will sound the same to the player regardless of where the Audio Source is in the scene. 3D sounds, in contrast, change based on how far they are from the Audio Listener (which is the component on FPCamera that acts like a microphone in this game), and they will even have a Doppler effect when moving toward or away from the Audio Listener (like how a car siren will change pitch as it is moving toward or away from you). 3D sounds are fantastic in some games, but we really don't want one for this prototype; we want the camera shutter to sound the same regardless of where the player is.

So, click the name *camera-shutter-click-03* in the Inspector to highlight it in the Project pane. Then select it in the Project pane to show import settings for *camera-shutter-click-03*. Uncheck the box next to *3D Sound* (setting it to false) and click *Apply*. Now, select *_TargetCamera* in the Hierarchy, and the Audio Source component will now tell you that *camera-shutter-click-03* is a 2D sound. In the Audio Source component, make sure that *Bypass Effects* is checked and *Play On Awake* is not checked. (If *Play On Awake* were true, it would cause the camera sound to play immediately when the scene started.)

Adding the Shutter Visual Effect

In the Project pane, select the Texture named `_White` in the Textures folder. Then choose *Game Object* > *Create Other* > *GUI Texture* from the menu bar. This will make a new `GUITexture` in the Hierarchy named `_White`. Rename it to `WhiteOut` so that it is not confused with the `_White` already in the Hierarchy. Then, set it as follows:

WhiteOut (GUITexture) P:[0.5,0.5,0] R:[0,0,0] S:[1,1,0]
Layer: _TargetCamera
Pixel Inset: [x:0, y:0, w:0, h:0]

This should completely fill the Target window with white. Any time the shot in the Target window

changes, this WhiteOut will flash to draw the player's attention.

Audio and Visual Effect Code

Add the following code to the TargetCamera class to use the Audio Source and WhiteOut. Note the two different strategies that we use for delaying an action: `Invoke()` and yielding a coroutine.

[Click here to view code image](#)

```
public class TargetCamera : MonoBehaviour {  
    ...  
    public Shot[] playerShots;  
    public float[] playerRatings;  
    public GUITexture whiteOut;  
  
    void Start() {  
        // Find the GUI Components  
        ...  
        checkMark = go.GetComponent<GUITexture>();  
        go = GameObject.Find ("WhiteOut");  
        whiteOut = go.GetComponent<GUITexture>();  
        // Hide the checkMark and whiteOut  
        checkMark.enabled = false;  
        whiteOut.enabled = false;  
        ...  
    }  
  
    void Update () {  
        Shot sh;  
  
        // Mouse Input  
        // If Left or Right mouse button is pressed this frame...  
        if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)) {  
            ...  
  
            if (editMode) {  
                ...  
            } else {  
                // Test this shot against the current Shot  
                float acc = Shot.Compare( Shot.shots[shotNum], sh );  
                lastShot = sh;  
                playerShots[shotNum] = sh;  
                playerRatings[shotNum] = acc;  
                // Show the shot just taken by the player  
                ShowShot(sh);  
                // Return to the current shot after waiting 1 second  
                Invoke("ShowCurrentShot",1);  
            }  
  
            // Play the shutter sound  
            this.GetComponent< AudioSource >().Play();  
        }  
        ...  
    }  
  
    public void ShowShot(Shot sh) {  
        // Call WhiteOutTargetWindow() and let it handle its own timing  
        StartCoroutine( WhiteOutTargetWindow() );  
        // Position _TargetCamera with the Shot  
        this.transform.position = sh.position;  
        this.transform.rotation = sh.rotation;  
    }  
}
```

```

public void ShowCurrentShot() {
    ShowShot(Shot.shots[shotNum]);
}

// Another use for coroutines is to have a fire-and-forget function with a
// delay in it as we've done here. WhiteOutTargetWindow() will enable
// whiteOut, yield for 0.05 seconds, and then disable it. Compare this
// method of delay to the Invoke("ShowCurrentShot",1f) used above
public IEnumerator WhiteOutTargetWindow() {
    whiteOut.enabled = true;
    yield return new WaitForSeconds(0.05f);
    whiteOut.enabled = false;
}

...
}

```

Now, when you play the scene, it should feel more like taking photos. When you snap a shot, the shutter sound is played, WhiteOut blinks, the shot you just took is shown for a second, WhiteOut blinks again, and then the target shot is shown with a rating for how well you did.

Summary

Although the coding in this tutorial was simpler than previous chapters, I hope you appreciated the chance to see the kind of powerful visual technology that is included in Unity (even in Unity Free). Most of the tutorial projects in this book are meant to be prototypes, and as such, they look like prototypes. With this project, I wanted to give you the chance to make something beautiful.

The tutorial in the next chapter returns to prototype-style graphics, but it includes the most complex user interaction and reading of XML files. It also makes further use of the multicamera tricks that you learned in this chapter.

Next Steps

This prototype introduced you to a lot of new concepts and showed how to get beautiful graphics out of Unity. Here are some additional things you can add to make it a more interesting game:

1. Add a timer in the top right of the main window that counts down the amount of time the player has to find all eight photographs.
2. Change the environment or create a new environment for the game. The current environment has a flat floor, but there's no need for that to be the case.
3. If you do change the environment (or even in this environment), you might want to consider adjusting the `maxTarDeviation` of a shot comparison based on the distance of the `Shot.target` from the `Shot.postion`. If the two are far apart, you should allow more `maxTarDeviation`. This would be particularly important in a large outdoor environment.
4. Put moving creatures in the environment that the player must find and photograph. Give the player points for the location and size of the creature in the shot. Something similar to this was a really fun photography side quest in *Beyond Good & Evil* by Ubisoft.

Hint: Use `GameObject.Renderer.Bounds` to get the 3D bounding volume around the creature, and then call `Camera.WorldToViewportPoint()` on the Bound's center and extents to find where they would be in the viewport. If the object is centered and fills a decent amount of the frame, give the player points. You might even give them more points for following the rule of

thirds (search for “photography rule of thirds” online).

If you do add dynamic objects moving through the scene, and you’re using Unity Pro, be sure to look at the Mesh Renderer component of the Inspector for the dynamic objects and set *Use Light Probes* to true. This will enable you to use the light probe information shown in [Figure 34.2](#) on the dynamic objects.

5. Add roving guards or lights that the player must hide from. Information about this can be found in the original Unity Stealth tutorial that this environment was pulled from. Search for “Unity Stealth tutorial” online, and you should find it easily.

Chapter 35. Prototype 8: Omega Mage

Omega Mage is a game prototype that mixes the dungeon exploration of *The Legend of Zelda* or *Rogue* with element-based spell-casting and an interface that works for either mouse or touchscreen (Android, iOS, and so on).

This is the last prototype of the book and therefore the most complex. At the end, you'll have a nice skeleton for an action-adventure game.

Getting Started: Omega Mage Prototype

The unitypackage for this project includes a number of assets, materials, and scripts. Because you already have experience with building objects in Unity out of simple shapes (as you did in the SHMUP prototype), I do not ask you to do so in this chapter. Instead, you import a series of prefabs that will serve as the artwork in this game.

Game Overview

Omega Mage is a dungeon crawler in the vein of classics like *The Legend of Zelda* and *Rogue*. However, in this game, the main character is a magic user who can call upon the four elements to defeat her enemies.

[Figure 35.1](#) shows what Omega Mage will look like at the end of the prototype. The `_MainCamera` on the left 3/4 of the screen shows the action from a top-down camera angle, and the right 1/4 of the screen contains a simple inventory for selecting elements.



Figure 35.1 An example of what Omega Mage will look like

Omega Mage was designed with touch-based tablets like the iPad and Android in mind, so the interactions will work equally well through either mouse or touch interface.

The player will click or tap on the ground in the main area to cause her character to walk to the tapped point. Tapping on one of the four elements in the inventory area will select it (and the player will see it orbiting around the Mage). Tapping the black “none” element in the middle of the others clears the element selection.

Clicking and dragging with no element selected causes the Mage to continuously move toward the player’s cursor (or touch) until she releases the mouse or lifts her finger, at which point, the Mage will stop. If she has an element selected, clicking and dragging on the ground will cast a ground spell (like the fire spell shown in [Figure 35.1](#)).

Tapping on an enemy will attack it. If the player has an element selected, the Mage will cast a spell of that element type. If she has no element selected, the tap will push the enemy away slightly.

Changes to ProtoTools in the Unitypackage

For this prototype, I’ve added a couple of new scripts to the ProtoTools directory:

- **PT_MonoBehaviour:** This simple script just adds some shortcuts to nested fields that we change often, including position, localPosition, rotation, scale, material, and color. If your scripts extend this instead of MonoBehaviour, they’ll gain this small added functionality.
- **PT_Mover:** This more complex script allows an object to move in a time-based way using Bézier curves that affect position, rotation, scale, and color of the primary material. This is a child of PT_MonoBehaviour, so if your scripts extend this, they will gain both the PT_MonoBehaviour and PT_Mover functionalities.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see [Appendix A, “Standard Project Setup Procedure.”](#) When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 3D for this project.

- **Project name:** OmegaMage
- **Download and import package:** Find [Chapter 35](#) at <http://book.prototools.net>
- **Scene name:** __OmegaMage_Scene_0
- **Project folders:** These are all imported from the unitypackage
- **C# script names:** Just the imported scripts in the ProtoTools folder
- **Remember:** Change the name of Main Camera to _MainCamer

The texture images for the ground and wall blocks that are in the _Textures & Materials/cartoon6r.free.fr folder were created by Philippe Cizaire and are used and included in the unitypackage with his permission. To see more of his work, check out his website at <http://cartoon6r.free.fr>.

Building the Scene

Unlike the previous prototype, this game uses Extensible Markup Language (XML) and a series of 3D tiles to build the game environment. Double-click the *Rooms.xml* file inside the Resources folder to see what I mean. The layout of these rooms using text was inspired by the classic adventure game *Rogue*, which became one of the first “graphical” games by displaying the player, enemies, and dungeons as ASCII art that looked much like the text in the *Rooms.xml* file.

In this prototype, we need to read the XML data from Rooms.xml and parse that into information to build a room from 3D tiles (using TilePrefab in the _Prefabs folder). This will be accomplished by a LayoutTiles script on _MainCamera in concert with a small Tile script on the TilePrefab. The textures for the tiles will be defined using fields in the _MainCamera:LayoutTiles Inspector.

To start, create a new C# script in the __Scripts folder named *Tile* and attach it to TilePrefab in the Prefabs folder. Open the Tile script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Tile : PT_MonoBehaviour {
    // public fields
    public string type;

    // Hidden private fields
    private string _tex;
    private int _height = 0;
    private Vector3 _pos;

    // Properties with get{} and set{}

    // height moves the Tile up or down. Walls have height=1
    public int height {
        get { return( _height ); }
        set {
            _height = value;
            AdjustHeight();
        }
    }

    // Sets the texture of the Tile based on a string
    // It requires LayoutTiles, so it's commented out for now
    /* */ // 1
    public string tex {
        get {
            return( _tex );
        }
        set {
            _tex = value;
            name = "TilePrefab_"+_tex; // Sets the name of this GameObject
            Texture2D t2D = LayoutTiles.S.GetTileTex(_tex);
            if (t2D == null) {
                Utils.tr("ERROR","Tile.type{set}=",value,
                    "No matching Texture2D in LayoutTiles.S.tileTextures!");
            } else {
                renderer.material.mainTexture = t2D;
            }
        }
    }
    /* */ // 2

    // Uses the "new" keyword to replace the pos inherited from PT_MonoBehaviour
    // Without the "new" keyword, the two properties would conflict
    new public Vector3 pos {
        get { return( _pos ); }
        set {
            _pos = value;
            AdjustHeight();
        }
    }
}
```

```

// Methods
    public void AdjustHeight() {
        // Moves the block up or down based on _height
        Vector3 vertOffset = Vector3.back*(_height-0.5f);
        // The -0.5f shifts the Tile down 0.5 units so that it's top surface is
        // at z=0 when pos.z=0 and height=0
        transform.position = _pos+vertOffset;
    }
}

```

1. This is the beginning of a multiline comment that hides the property `tex` from the compiler for now.

2. This is the end of the multiline comment that hides `tex`.

You'll notice that the `tex` property is commented out. This is because it requires the `LayoutTiles` script to be in place to compile properly. After you have entered the preceding code and made sure that it all compiles, create a new script named `LayoutTiles` in the `__Scripts` folder and attach it to `_MainCamera`. Then, remove the multiline comments (`/*` and `*/`) from before and after the `tex` property. Unity will throw a compiler error (Assets/`__Scripts`/Tile.cs(31,53): error CS0117: 'LayoutTiles' does not contain a definition for 'S'), but we'll rectify that now by writing the `LayoutTiles` script. Open `LayoutTiles` in `MonoBehaviour` and add the following code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[System.Serializable]
public class TileTex {
    // This class enables us to define various textures for tiles
    public string str;
    public Texture2D tex;
}

public class LayoutTiles : MonoBehaviour {
    static public LayoutTiles S;

    public TextAsset roomsText; // The Rooms.xml file
    public string roomNumber = "0"; // Current room # as a string
    // ^ roomNumber as string allows encoding in the XML & rooms 0-F
    public GameObject tilePrefab; // Prefab for all Tiles
    public TileTex[] tileTextures; // A list of named textures for Tiles

    public bool _____;

    public PT_XMLReader roomsXMLR;
    public PT_XMLHashList roomsXML;
    public Tile[,] tiles;
    public Transform tileAnchor;

    void Awake() {
        S = this; // Set the Singleton for LayoutTiles

        // Make a new GameObject to be the TileAnchor (the parent transform of
        // all Tiles). This keeps Tiles tidy in the Hierarchy pane.
        GameObject tAnc = new GameObject("TileAnchor");
        tileAnchor = tAnc.transform;
    }
}

```

```

// Read the XML
roomsXMLR = new PT_XMLReader(); // Create a PT_XMLReader
roomsXMLR.Parse(roomsText.text); // Parse the Rooms.xml file
roomsXML = roomsXMLR.xml["xml"][0]["room"]; // Pull all the <room>s

// Build the 0th Room
BuildRoom(roomNumber);
}

// This is the GetTileTex() method that Tile uses
public Texture2D GetTileTex(string tStr) {
    // Search through all the tileTextures for the proper string
    foreach (TileTex tTex in tileTextures) {
        if (tTex.str == tStr) {
            return(tTex.tex);
        }
    }
    // Return null if nothing was found
    return(null);
}

// Build a room from an XML <room> entry
public void BuildRoom(PT_XMLHashtable room) {
    // Get the texture names for the floors and walls from <room> attributes
    string floorTexStr = room.att("floor");
    string wallTexStr = room.att("wall");
    // Split the room into rows of tiles based on carriage returns in the
    // Rooms.xml file
    string[] roomRows = room.text.Split('\n');
    // Trim tabs from the beginnings of lines. However, we're leaving spaces
    // and underscores to allow for non-rectangular rooms.
    for (int i=0; i<roomRows.Length; i++) {
        roomRows[i] = roomRows[i].Trim('\t');
    }
    // Clear the tiles Array
    tiles = new Tile[ 100, 100 ]; // Arbitrary max room size is 100x100

    // Declare a number of local fields that we'll use later
    Tile ti;
    string type, rawType, tileTexStr;
    GameObject go;
    int height;
    float maxY = roomRows.Length-1;

    // These loops scan through each tile of each row of the room
    for (int y=0; y<roomRows.Length; y++) {
        for (int x=0; x<roomRows[y].Length; x++) {
            // Set defaults
            height = 0;
            tileTexStr = floorTexStr;

            // Get the character representing the tile
            type = rawType = roomRows[y][x].ToString();
            switch (rawType) {
                case " ": // empty space
                case "_": // empty space
                    // Just skip over empty space
                    continue;
                case ".": // default floor
                    // Keep type="."
                    break;
                case "|": // default wall
                    height = 1;
            }
        }
    }
}

```

```
        break;
    default:
        // Anything else will be interpreted as floor
        type = ".";
        break;
    }

    // Set the texture for floor or wall based on <room> attributes
    if (type == ".") {
        tileTexStr = floorTexStr;
    } else if (type == "|") {
        tileTexStr = wallTexStr;
    }

    // Instantiate a new TilePrefab
    go = Instantiate(tilePrefab) as GameObject;
    ti = go.GetComponent<Tile>();
    // Set the parent Transform to tileAnchor
    ti.transform.parent = tileAnchor;
    // Set the position of the tile
    ti.pos = new Vector3( x, maxY-y, 0 );
    tiles[x,y] = ti; // Add ti to the tiles 2D Array

    // Set the type, height, and texture of the Tile
    ti.type = type;
    ti.height = height;
    ti.tex = tileTexStr;

    // More to come here...
}

}
```

This should all compile fine (and now you'll no longer have the error from Tile), but you need to add some things in the `_MainCamera:LayoutTiles` Inspector before it will actually work. Click `_MainCamera` in the Hierarchy and enter the data shown in [Figure 35.2](#).

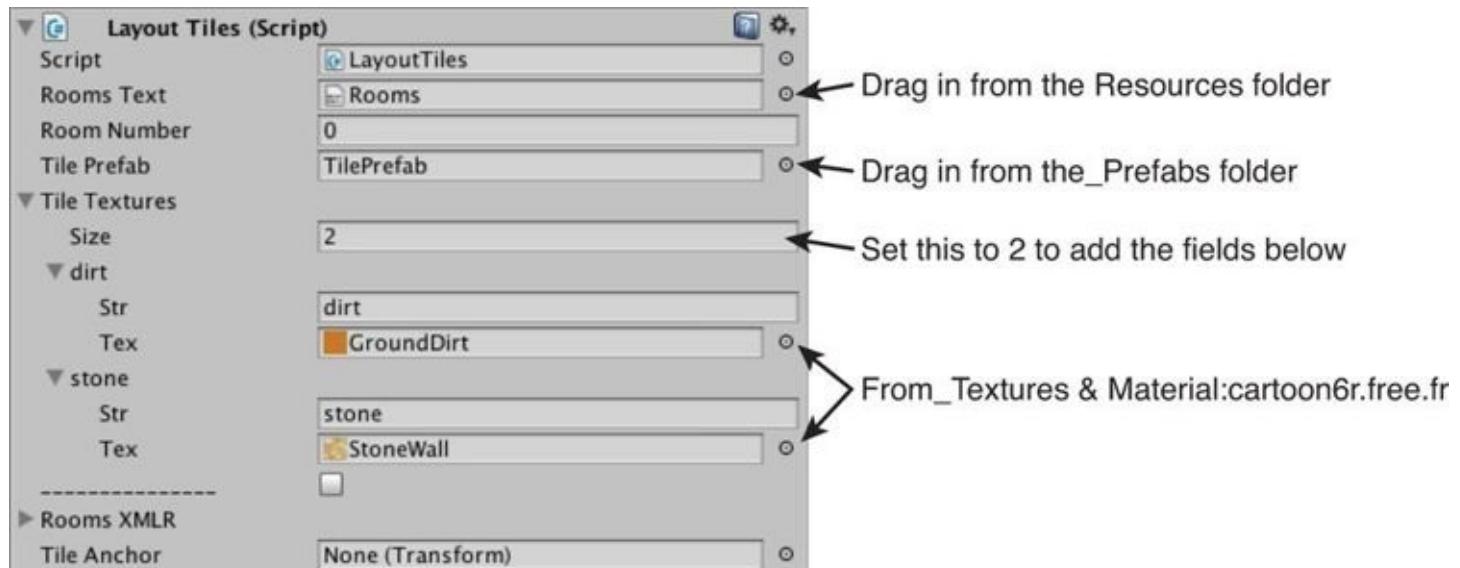


Figure 35.2 Inspector settings MainCamera:LayoutTiles

After you have done this, press Play, and a room will be built for you. If you like, you can stop, change `_MainCamera:LayoutTiles.roomNumber` in the Inspector to another valid number (0–8), and press Play again to see other rooms in the dungeon. Be sure to stop playback and set `roomNumber` back to 0 when you’re done looking around.

You'll notice that the room is pretty dark now, but that's okay; our Mage character will carry a torch. Oh, and save your scene. Always remember to save your scene!

The Mage Character

Drag `_Mage` from the `_Prefabs` folder into the Hierarchy pane. This is the `GameObject` for the player character in *Omega Mage*. You can see that `_Mage` comes complete with a `Rigidbody` and `Capsule Collider` as well as a placeholder human model and an overhead spotlight.

Create a new C# script in the `__Scripts` folder named *Mage* and attach it to `_Mage` in the Hierarchy. Open the *Mage* script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;      // Enables List<>s
using System.Linq;                  // Enables LINQ queries

// Mage is a subclass of PT_MonoBehaviour
public class Mage : PT_MonoBehaviour {
    static public Mage S;

    void Awake() {
        S = this; // Set the Mage Singleton
    }
}
```

Of course, we'll add a lot more to this later, but for now, we need to properly position `_Mage` in the room. Open `LayoutTiles` in `MonoBehaviour` and add the following bold code to the `BuildRoom()` method just above where it currently has " // More to come here...":

[Click here to view code image](#)

```
public class LayoutTiles : MonoBehaviour {
    ...

    // Build a room from an XML <room> entry
    public void BuildRoom(PT_XMLHashtable room) {
        ...

        // These loops scan through each tile of each row of the room
        for (int y=0; y<roomRows.Length; y++) {
            for (int x=0; x<roomRows[y].Length; x++) {
                ...
                ti.tex = tileTexStr;

                // If the type is still rawType, continue to the next iteration
                if (rawType == type) continue;

                // Check for specific entities in the room
                switch (rawType) { // 1
                    case "X": // Starting position for the Mage
                        Mage.S.pos = ti.pos; // Uses the Mage Singleton
                        break;
                }

                // More to come here...
            }
        }
    }
}
```

1. In the previous version of this code listing, it would treat any `rawType` char it didn't recognize as a '.' (ground). Here, the code checks to see if that unrecognized `rawType` char was an 'X' (the char for the start position of the Mage).

This will properly position the `_Mage` where the `X` is in Room 0 of the `Rooms.xml` file. Now, it's time to make `_MainCamera` follow `_Mage`. Create a new C# script in the `_Scripts` folder named `CameraFollow` and attach it to `_MainCamera`. Open `CameraFollow` in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class CameraFollow : PT_MonoBehaviour {
    public static CameraFollow S;

    public Transform targetTransform;
    public float camEasing = 0.1f;
    public Vector3 followOffset = new Vector3(0,0,-2);

    void Awake() {
        S = this;
    }

    void FixedUpdate() {
        Vector3 pos1 = targetTransform.position+followOffset;
        pos = Vector3.Lerp(pos, pos1, camEasing);
    }
}
```

Now, return to Unity and select `_MainCamera` in the Inspector. You need to set `targetTransform` in the `_MainCamera:CameraFollow` Inspector. Click the disclosure triangle next to `_Mage` in the Hierarchy; then open the disclosure triangle next to `CharacterTrans`. You'll see that `Spotlight` is a child of `CharacterTrans`. Drag `Spotlight` into the `targetTransform` field in the `_MainCamera:CameraFollow` Inspector. This will cause the `_MainCamera` to be slightly ahead of `_Mage` because `Spotlight` is always positioned slightly ahead of where the `_Mage` is looking. Press Play, and you should see this result. If you select `_Mage` in the Hierarchy while the scene is playing and adjust the `transform.rotation.z` value in the `_Mage` Inspector, you should see the camera follow the pool of light that is just in front of the `_Mage`.

Mouse Interaction

Because we want this game to be playable on either a computer or touch-based mobile device, all of the interactions are going to be based on simple mouse gestures. This works for tablets because Unity automatically converts single touches on a tablet into mouse gestures.

The mouse interaction code will be added to the `Mage` script, and it's going to take a decent amount of coding to get the initial pieces working. Add the following to the `Mage` script:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic; // Enables List<>s
using System.Linq; // Enables LINQ queries

// The MPhase enum is used to track the phase of mouse interaction
public enum MPhase {
```

```

idle,
down,
drag
}

// MouseInfo stores information about the mouse in each frame of interaction
[System.Serializable]
public class MouseInfo {
    public Vector3      loc;          // 3D loc of the mouse near z=0
    public Vector3      screenLoc;    // Screen position of the mouse
    public Ray          ray;          // Ray from the mouse into 3D space
    public float        time;         // Time this mouseInfo was recorded
    public RaycastHit  hitInfo;      // Info about what was hit by the ray
    public bool         hit;          // Whether the mouse was over any collider

    // These methods see if the mouseRay hits anything
    public RaycastHit Raycast() {
        hit = Physics.Raycast(ray, out hitInfo);
        return(hitInfo);
    }

    public RaycastHit Raycast(int mask) {
        hit = Physics.Raycast(ray, out hitInfo, mask);
        return(hitInfo);
    }
}

// Mage is a subclass of PT_MonoBehaviour
public class Mage : PT_MonoBehaviour {
    static public Mage S;
    static public bool DEBUG = true;

    public float          mTapTime = 0.1f; // How long is considered a tap
    public float          mDragDist = 5; // Min dist in pixels to be a drag

    public float          activeScreenWidth = 1; // % of the screen to use

    public bool _____;

    public MPhase          mPhase = MPhase.idle;
    public List<MouseInfo> mouseInfos = new List<MouseInfo>();

    void Awake() {
        S = this; // Set the Mage Singleton
        mPhase = MPhase.idle;
    }

    void Update() {
        // Find whether the mouse button 0 was pressed or released this frame
        bool b0Down = Input.GetMouseButtonDown(0);
        bool b0Up = Input.GetMouseButtonUp(0);

        // Handle all input here (except for Inventory buttons)
        /*
        There are only a few possible actions:                                // 1
        1. Tap on the ground to move to that point
        2. Drag on the ground with no spell selected to move to the Mage
        3. Drag on the ground with spell to cast along the ground
        4. Tap on an enemy to attack (or force-push away without an element)
        */

        // An example of using < to return a bool value
        bool inActiveArea = (float) Input.mousePosition.x / Screen.width <
activeScreenWidth;
    }
}

```

```

// This is handled as an if statement instead of switch because a tap
// can sometimes happen within a single frame
if (mPhase == MPhase.idle) { // If the mouse is idle
    if (b0Down && inActiveArea) {
        mouseInfos.Clear(); // Clear the mouseInfos
        AddMouseInfo(); // And add a first MouseInfo

        // If the mouse was clicked on something, it's a valid MouseDown
        if (mouseInfos[0].hit) { // Something was hit!
            MouseDown(); // Call MouseDown()
            mPhase = MPhase.down; // and set the mPhase
        }
    }
}

if (mPhase == MPhase.down) { // if the mouse is down
    AddMouseInfo(); // Add a MouseInfo for this frame
    if (b0Up) { // The mouse button was released
        MouseTap(); // This was a tap
        mPhase = MPhase.idle;
    } else if (Time.time - mouseInfos[0].time > mTapTime) {
        // If it's been down longer than a tap, this may be a drag, but
        // to be a drag, it must also have moved a certain number of
        // pixels on screen.
        float dragDist = (lastMouseInfo.screenLoc -
                           mouseInfos[0].screenLoc).magnitude;
        if (dragDist >= mDragDist) {
            mPhase = MPhase.drag;
        }
    }
}

if (mPhase == MPhase.drag) { // if the mouse is being drug
    AddMouseInfo();
    if (b0Up) {
        // The mouse button was released
        MouseDragUp();
        mPhase = MPhase.idle;
    } else {
        MouseDrag(); // Still dragging
    }
}

// Pulls info about the Mouse, adds it to mouseInfos, and returns it
MouseInfo AddMouseInfo() {
    MouseInfo mInfo = new MouseInfo();
    mInfo.screenLoc = Input.mousePosition;
    mInfo.loc = Utils.mouseLoc; // Gets the position of the mouse at z=0
    mInfo.ray = Utils.mouseRay; // Gets the ray from the Main Camera through
                                // the mouse pointer
    mInfo.time = Time.time;
    mInfo.Raycast(); // Default is to raycast with no mask

    if (mouseInfos.Count == 0) {
        // If this is the first mouseInfo
        mouseInfos.Add(mInfo); // Add mInfo to mouseInfos
    } else {
        float lastTime = mouseInfos[mouseInfos.Count-1].time;
        if (mInfo.time != lastTime) {
            // if time has passed since the last mouseInfo
            mouseInfos.Add(mInfo); // Add mInfo to mouseInfos
        }
    }
}

```

```

        }
        // This time test is necessary because AddMouseInfo() could be
        // called twice in one frame
    }
    return(mInfo); // Return mInfo as well
}

public MouseInfo lastMouseInfo {
    // Access to the latest MouseInfo
    get {
        if (mouseInfos.Count == 0) return( null );
        return( mouseInfos[mouseInfos.Count-1] );
    }
}

void MouseDown() {
    // The mouse was pressed on something (it could be a drag or tap)
    if (DEBUG) print("Mage.MouseDown()");
}

void MouseTap() {
    // Something was tapped like a button
    if (DEBUG) print("Mage.MouseTap()");
}

void MouseDrag() {
    // The mouse is being drug across something
    if (DEBUG) print("Mage.MouseDrag()");
}

void MouseDragUp() {
    // The mouse is released after being drug
    if (DEBUG) print("Mage.MouseDragUp()");
}

}

```

1. As you can see in the multiline comment, only a few different kinds of possible mouse interactions are possible:

1. Tap on the ground to move to that point.
2. Drag on the ground with no spell selected to move to the Mage.
3. Drag on the ground with a spell selected to cast along the ground.
4. Tap on an enemy to attack (or force-push away without an element).

After you have added this code, you can press Play and see the results. If you click the mouse on nothing (the background), nothing happens. Clicking any object will produce a “Mage.MouseTap()” message in the Console window. Clicking and holding (or dragging) will produce “Mage.MouseDrag()” messages followed by a “Mage.MouseDragUp()” message when you release the button.

The `if (DEBUG) print();` statements will only print if the static `Mage.DEBUG` bool is set to true. That way, you can easily turn these comments off when you no longer need them.

Of the four mouse interactions listed in note // 1, we’ll handle the two types of movement first.

Movement

When the player taps on the ground, the Mage should move there. First we'll add code to control the Mage movement using the Rigidbody (which will make collision happen properly). Add the following bold code to the Mage class. This will allow a tap to move the _Mage:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...
    public float activeScreenWidth = 1; // % of the screen to use

public float speed = 2; // The speed at which _Mage walks

    public bool _____;

    public MPhase mPhase = MPhase.idle;
    public List<MouseInfo> mouseInfos = new List<MouseInfo>();

    public bool walking = false;
    public Vector3 walkTarget;
    public Transform characterTrans;

    void Awake() {
        S = this; // Set the Mage Singleton
        mPhase = MPhase.idle;

        // Find the characterTrans to rotate with Face()
        characterTrans = transform.Find("CharacterTrans");
    }

    ...

    void MouseTap() {
        // Something was tapped like a button
        if (DEBUG) print("Mage.MouseTap()");

        WalkTo(lastMouseInfo.loc); // Walk to the latest mouseInfo pos
    }

    void MouseDragUp() {
        // The mouse is released after being dragged
        print("Mage.MouseDragUp()");
    }

// Walk to a specific position. The position.z is always 0
public void WalkTo(Vector3 xTarget) {
    walkTarget = xTarget; // Set the point to walk to
    walkTarget.z = 0; // Force z=0
    walking = true; // Now the Mage is walking
    Face(walkTarget); // Look in the direction of the walkTarget
}

public void Face(Vector3 poi) { // Face toward a point of interest
    Vector3 delta = poi-pos; // Find vector to the point of interest
    // Use Atan2 to get the rotation around Z that points the X-axis of
    // _Mage:CharacterTrans toward poi
    float rZ = Mathf.Rad2Deg * Mathf.Atan2(delta.y, delta.x);
    // Set the rotation of characterTrans (doesn't actually rotate _Mage)
    characterTrans.rotation = Quaternion.Euler(0,0,rZ);
}

public void StopWalking() { // Stops the _Mage from walking
```

```
walking = false;
rigidbody.velocity = Vector3.zero;
}

void FixedUpdate () { // Happens every physics step (i.e., 50 times/second)
    if (walking) { // If Mage is walking
        if ( (walkTarget-pos).magnitude < speed*Time.fixedDeltaTime ) {
            // If Mage is very close to walkTarget, just stop there
            pos = walkTarget;
            StopWalking();
        } else {
            // Otherwise, move toward walkTarget
            rigidbody.velocity = (walkTarget-pos).normalized * speed;
        }
    } else {
        // If not walking, velocity should be zero
        rigidbody.velocity = Vector3.zero;
    }
}

void OnCollisionEnter( Collision coll ) {
    GameObject otherGO = coll.gameObject;

    // Colliding with a wall can also stop walking
    Tile ti = otherGO.GetComponent<Tile>();
    if (ti != null) {
        if (ti.height > 0) { // If ti.height is > 0
            // Then this ti is a wall, and Mage should stop
            StopWalking();
        }
    }
}
```

Now, when you press Play, your Mage will walk to a point on the ground that you click. However, it would be nice to give the player a little more feedback about the tap.

Adding a Tap Indicator

Create a new C# script in the `_Scripts` folder named `TapIndicator` and drag it onto the `TapIndicator` in the `_Prefabs` folder. Then open the `TapIndicator` script in MonoDevelop and enter this code:

Click here to view code image

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

/*
TapIndicator makes use of the PT_Mover class from ProtoTools. This allows it to
use a Bezier curve to alter position, rotation, scale, etc.

You'll also notice that this adds several public fields to the Inspector.
*/

public class TapIndicator : PT_Mover {

    public float      lifeTime = 0.4f; // How long will it last
    public float[]    scales; // The scales it interpolates
    public Color[]    colors; // The colors it interpolates

    void Awake() {
        scale = Vector3.zero; // This initially hides the indicator
    }
}
```

```

}

void Start () {
    // PT_Mover works based on the PT_Loc class, which contains information
    // about position, rotation, and scale. It's similar to a Transform but
    // simpler (and Unity won't let us create Transforms at will).

    PT_Loc pLoc;
    List<PT_Loc> locs = new List<PT_Loc>();

    // The position is always the same and always at z=-0.1f
    Vector3 tPos = pos;
    tPos.z = -0.1f;

    // You must have an equal number of scales and colors in the Inspector
    for (int i=0; i<scales.Length; i++) {
        pLoc = new PT_Loc();
        pLoc.scale = Vector3.one * scales[i];      // Each scale
        pLoc.pos = tPos;
        pLoc.color = colors[i];                    // and each color

        locs.Add(pLoc);                          // is added to locs
    }

    // callback is a function delegate that can call a void function() when
    // the move is done
    callback = CallbackMethod;      // Call CallbackMethod() when finished

    // Initiate the move by passing in a series of PT_Locs and duration for
    // the Bézier curve.
    PT_StartMove(locs, lifeTime);
}

void CallbackMethod() {
    Destroy(gameObject);      // When the move is done, Destroy(gameObject)
}
}

```

As you can see from the code, this creates a fire-and-forget object. It is instantiated, sets up a Bézier curve to interpolate both scale and color (we'll set the values for this in the Inspector next), and when the PT_Mover is done, it calls the `CallbackMethod()` and destroys its own GameObject.

Click on TapIndicator in the `_Prefab` folder of the Project pane and enter the values shown in [Figure 35.3](#) into the Inspector.

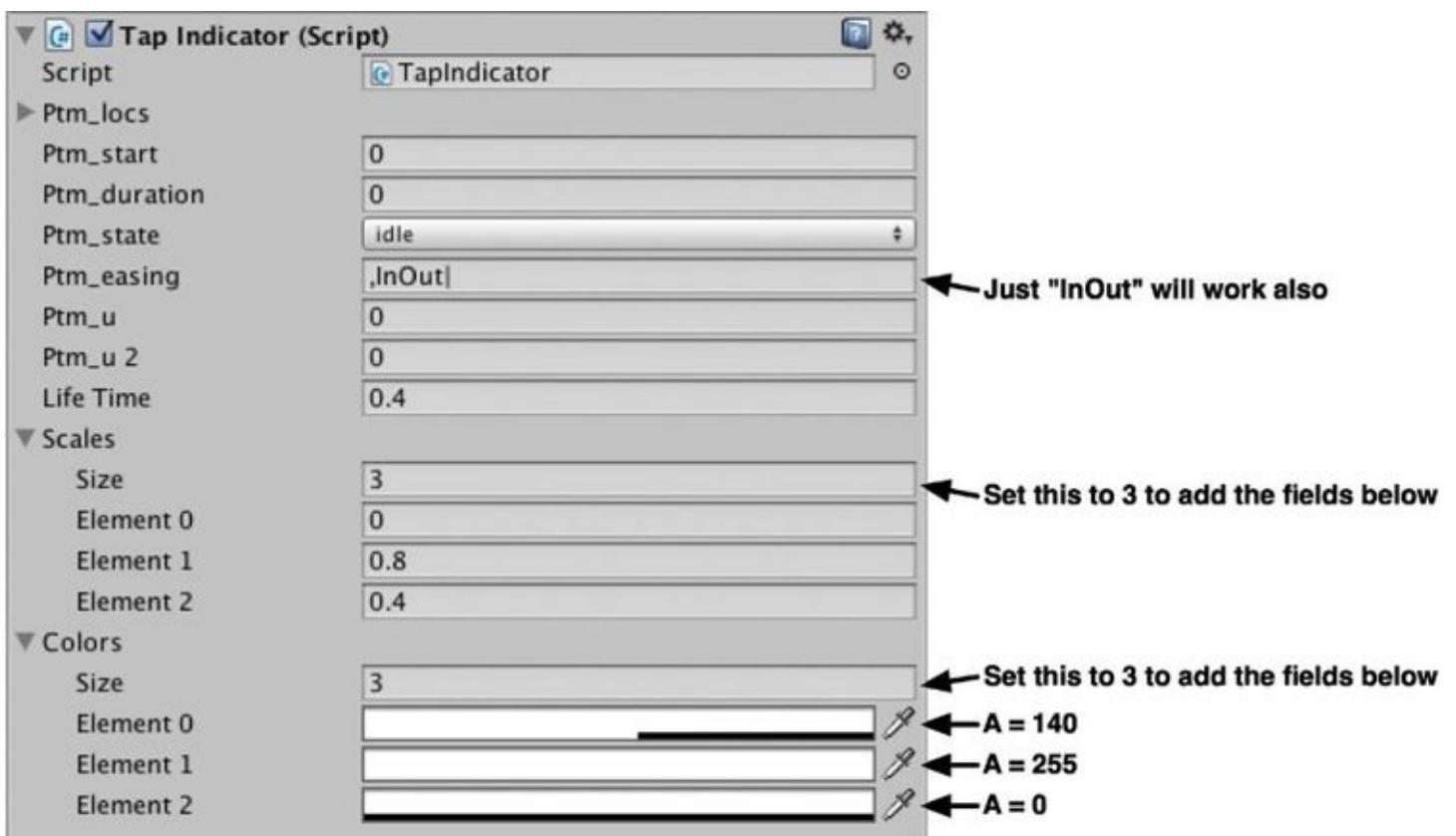


Figure 35.3 Inspector settings TapIndicator:TapIndicator

Now, open Mage in MonoDevelop and add the following code to instantiate the TapIndicator:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...

    public float mTapTime = 0.1f; // How long is considered a tap
    public GameObject tapIndicatorPrefab; // Prefab of the tap indicator

    ...

    void MouseTap() {
        // Something was tapped like a button
        if (DEBUG) print("Mage.MouseTap()");

        WalkTo(lastMouseInfo.loc); // Walk to the latest mouseInfo pos
        ShowTap(lastMouseInfo.loc); // Show where the player tapped
    }

    ...

    void OnCollisionEnter( Collision coll ) {
        ...
    }

    // Show where the player tapped
    public void ShowTap(Vector3 loc) {
        GameObject go = Instantiate(tapIndicatorPrefab) as GameObject;
        go.transform.position = loc;
    }
}
```

The last step in implementing the TapIndicator is to set the `tapIndicatorPrefab` field in the `_Mage:Mage` Inspector. Drag TapIndicator from the `_Prefabs` folder into the

tapIndicatorPrefab field on _Mage: Mage, and then press Play. You should now see an indicator whenever you tap, as shown in [Figure 35.4](#).

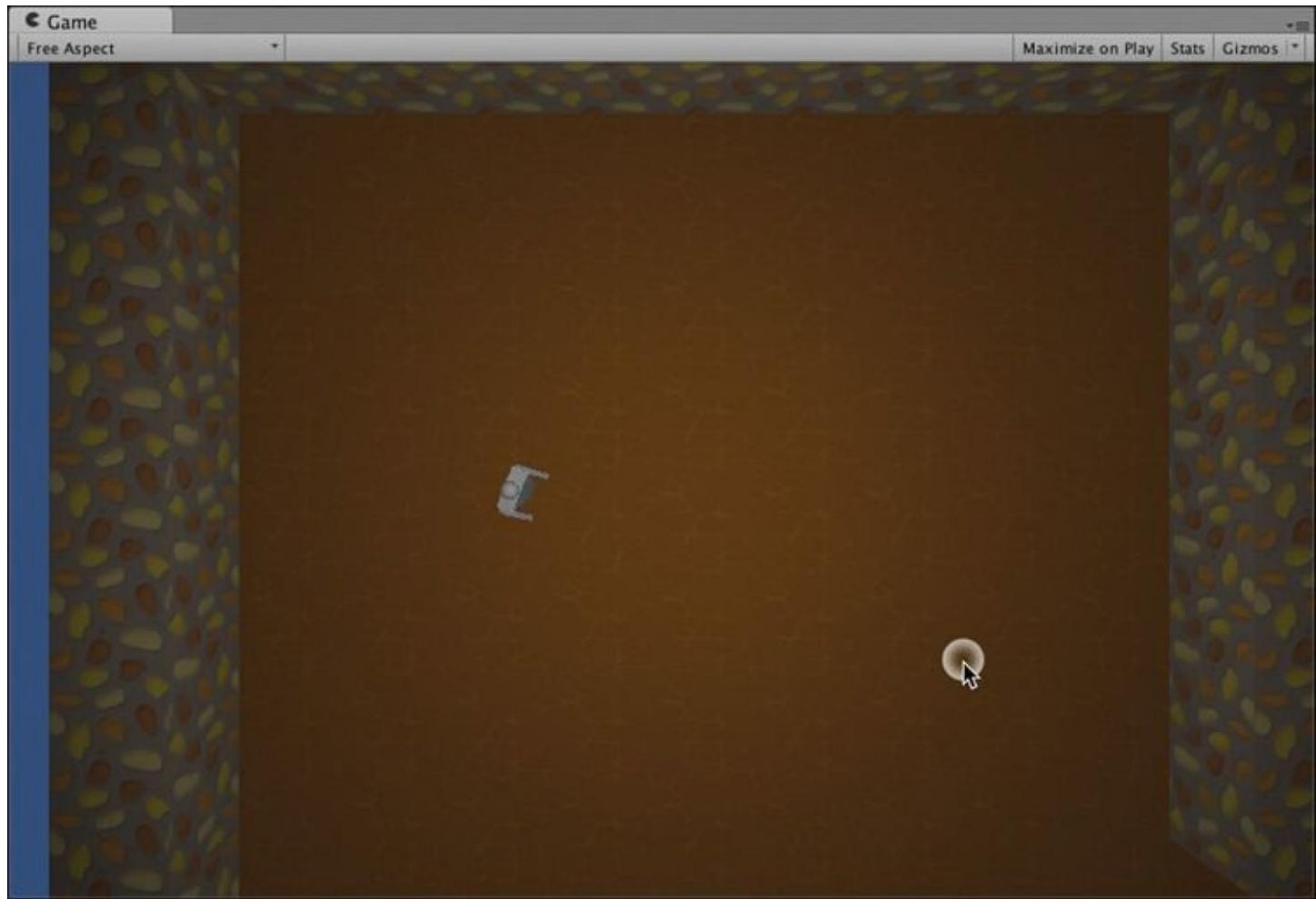


Figure 35.4 The Game pane showing current progress on Omega Mage

Moving When Dragging

The second type of mouse interaction previously listed was “Drag on the ground with no spell selected to move to the Mage.” This should also be easy to add. All we need to do is to call `WalkTo()` with a new location whenever the player is dragging the mouse. We also need to tell the Mage to `StopWalking()` when `MouseDragUp()` is called. Otherwise, the Mage will continue moving toward the last location of the players mouse, which doesn’t feel right following the continuous motion of leading the Mage with the drag. Add the following bolded code to Mage in MonoDevelop:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...
    void MouseDrag() {
        // The mouse is being drug across something
        if (DEBUG) print("Mage.MouseDrag()");
        // Continuously walk toward the current mouseInfo pos
        WalkTo(mouseInfos[mouseInfos.Count-1].loc);
    }

    void MouseDragUp() {
        // The mouse is released after being dragged
        if (DEBUG) print("Mage.MouseDragUp()");
    }
}
```

```

        // Stop walking when the drag is stopped
        StopWalking();
    }
    ...
}

```

This mouse interaction was easy, but the remaining two are a little more difficult because they have to do with casting spells. For casting spells, we need an inventory.

The Inventory and Selecting Elements

In [Figure 35.1](#) (at the beginning of the chapter), you saw that the right 1/4 of the screen contains an interface for selecting elements. It's time to build that interface.

Start by adding an `ElementType` enum to the top of the `Mage` script between `enum MPhase` and `class MouseInfo`. This will allow us to refer to elements by name or number. (You can convert from an enum to an int with a typecast, as you'll see in the next script.)

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;           // Enables List<>s
using System.Linq;                      // Enables LINQ queries

// The MPhase enum is used to track the phase of Mouse interaction
public enum MPhase {
    ...
}

// The ElementType enum
public enum ElementType {
    earth,
    water,
    air,
    fire,
    aether,
    none
}

// MouseInfo stores information about the mouse in each frame of interaction
[System.Serializable]
public class MouseInfo {
    ...
}

// Mage is a subclass of PT_MonoBehaviour
public class Mage : PT_MonoBehaviour {
    ...
}

```

Create a new C# script in the `__Scripts` folder named `ElementInventoryButton`. Open it in MonoDevelop and enter the following code:

[Click here to view code image](#)

```

using UnityEngine;
using System.Collections;

public class ElementInventoryButton : MonoBehaviour {
    public ElementType type;

```

```

void Awake() {
    // Parse the first character of the name of this GameObject into an int
    char c = gameObject.name[0];
    string s = c.ToString();
    int typeNum = int.Parse(s);

    // typecast that int to an ElementType
    type = (ElementType) typeNum;
}

void OnMouseUpAsButton() {
    // Tell the Mage to add this element type
    //Mage.S.SelectElement(type);
}

}

```

Once that is saved and compiles properly, switch back to Unity and drag `_InventoryCamera` from the `_Prefabs` folder into the Hierarchy. It should situate itself at `P:[-100,0,0]` `R:[0,0,0]` `S:[1,1,1]`.

Open the disclosure triangle next to `_InventoryCamera` in the Hierarchy, and then attach the `ElementInventoryButton` script to the child GameObjects `0_Earth`, `1_Water`, `2_Air`, `3_Fire`, and `5_None`. When you press Play, you should see that the `ElementInventoryButton.type` of each of these buttons assigns itself due to the `Awake()` method. You could assign these yourself in the Inspector, of course, but I felt that it was important for you to know that it is possible to convert from strings to chars to integers to enums.

Adjusting the Two Cameras

As you can see, the `_InventoryCamera` includes a camera that covers the right 1/4 of the screen and is rendered above the image from `_MainCamera`. This is controlled by the `Viewport Rect` and `Depth` fields in the `_InventoryCamera`. Camera Inspector. You can see the Inspectors for both `_MainCamera` and `_InventoryCamera` in [Figure 35.5](#). `_InventoryCamera` has a depth of 0, which is greater than the default `_MainCamera` depth of -1, so the `_MainCamera` is rendered in the background. However, `_MainCamera` is still centered in the middle of the screen with 1/4 of its image hidden by `_InventoryCamera`. Change the `Viewport Rect` settings of `_MainCamera` to those shown in [Figure 35.5](#), and `_MainCamera` will now only attempt to fill the left 3/4 of the screen. This would also be a good time to choose a different background color for `_MainCamera`. (I recommend [R:48, G:64, B:48] in the color picker; the alpha setting for a background color is ignored by Unity.)

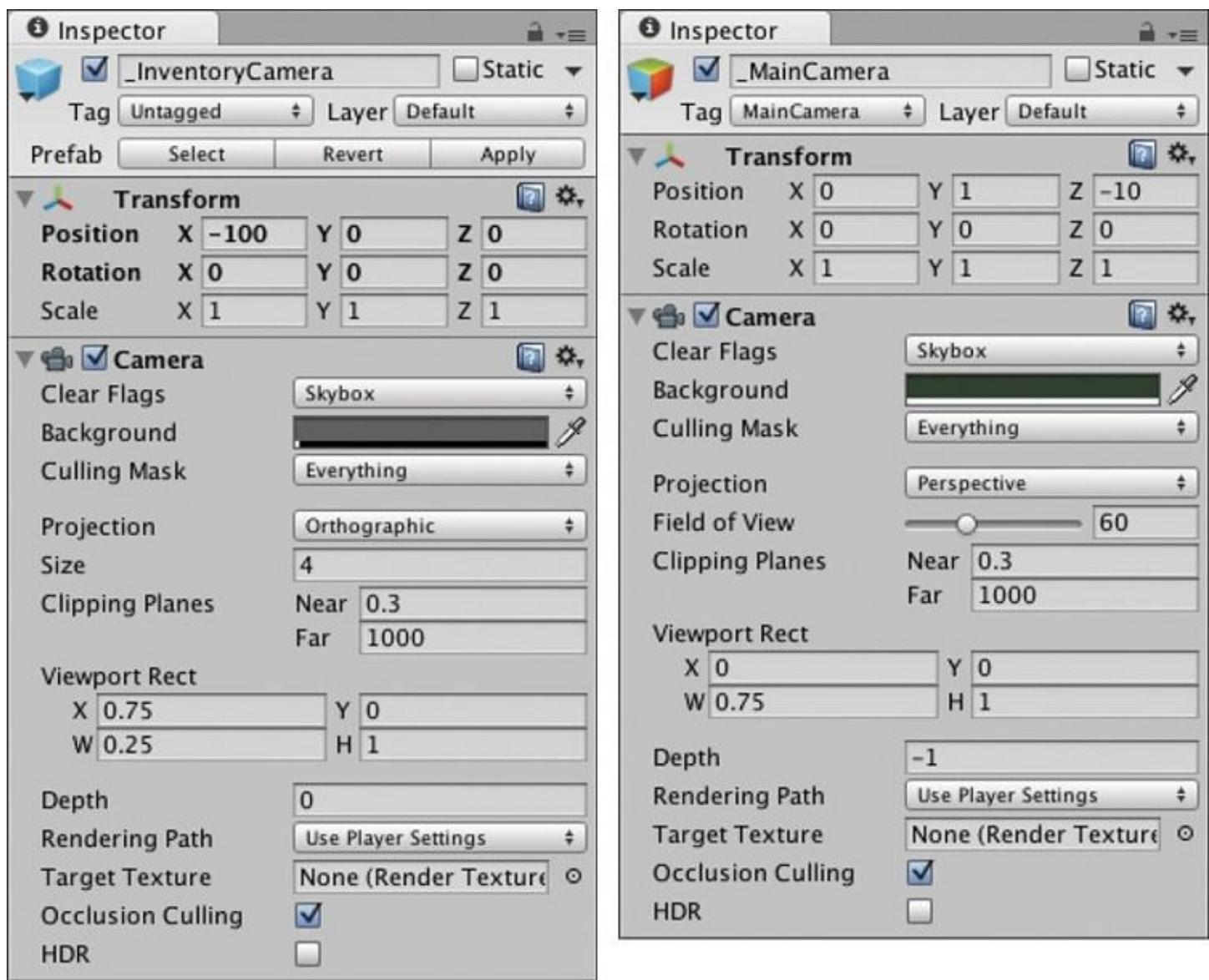


Figure 35.5 Camera Inspectors for `_InventoryCamera` and `_MainCamera`

Because this prototype is designed with an iPad in mind, the main screen is meant to be viewed at a 4x3 aspect ratio (the original iPad resolution was 1024x768, while the iPad Air and iPad Mini Retina both have a screen resolution of 2048x1536). To see what this will look like in Unity, choose 4:3 from the aspect ratio pop-up menu in the top-left corner of the Game pane. (If you choose something like 16:9, you will see gray bars on either side of the wooden inventory background.)

The final adjustment you need to do is to make Mage ignore any clicks or taps in the right 1/4 of the screen. Select Mage in the Hierarchy and set the `activeScreenWidth` field of the Mage (Script) Inspector to 0.75.

Selecting Elements

As you could see in the code for `ElementInventoryButton`, we need to add a `SelectElement()` method to Mage that will choose an element and cause it to circle the Mage's head. However, before doing that, we need to make the `Element` class and prepare the `Element_Sphere` GameObject prefabs that will circle the Mage. Create a new C# script in the `__Scripts` folder named *Element* and enter this code:

[Click here to view code image](#)

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

```
public class Element : PT_MonoBehaviour {
    public ElementType type;
}
```

As you can see, there's not much to it, but it does extend the PT_MonoBehaviour class, which gives us access to several shortcuts for modifying Element.gameObjects, and it attaches an ElementType field named type to each prefab.

Now, in Unity's Project pane, find the four Element_Spheres in the _Prefabs folder. Click *Element_Sphere_Air* and then Shift-click *Element_Sphere_Water* so that they're all four selected. From the menu bar, choose *Component > Scripts > Element*. This will simultaneously add the Element script to all four prefabs. Now, individually select each prefab and choose the appropriate Element.type in the Inspector pane for that prefab (for example, set the type of Element_Sphere_Air to air).

In further iterations of this game (beyond the scope of this book), multiple elements can be selected simultaneously to create different mixed spells as is done in the game *Magicka* by Paradox Interactive—and the code you're about to add to Mage will reflect that possible expansion—but for now, we will only allow the selection of a single element at a time. Add the following bolded code to Mage to implement element selection:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...
    public float speed = 2; // The speed at which _Mage walks

    public GameObject[] elementPrefabs; // The Element_Sphere Prefabs
    public float elementRotDist = 0.5f; // Radius of rotation
    public float elementRotSpeed = 0.5f; // Period of rotation
    public int maxNumSelectedElements = 1;

    public bool selectedElement = false;

    ...
    public Transform characterTrans;

    public List<Element> selectedElements = new List<Element>();

    ...

    void Update() {
        ...
        if (mPhase == MPhase.down) { // if the mouse is down
            AddMouseInfo(); // Add a MouseInfo for this frame
            if (b0Up) {
                ...
            } else if (Time.time - mouseInfos[0].time > mTapTime) {
                ...
                if (dragDist >= mDragDist) {
                    mPhase = MPhase.drag;
                }
            }
        // However, drag will immediately start after mTapTime if there
        // are no elements selected.
        if (selectedElements.Count == 0) {
            mPhase = MPhase.drag;
        }
    }
}
```

```

if (mPhase == MPhase.drag) {
    ...
}

OrbitSelectedElements();
}

...

// Show where the player tapped
public void ShowTap(Vector3 loc) {
    GameObject go = Instantiate(tapIndicatorPrefab) as GameObject;
    go.transform.position = loc;
}

// Chooses an Element_Sphere of elType and adds it to selectedElements
public void SelectElement(ElementType elType) {
    if (elType == ElementType.none) { // If it's the none element...
        ClearElements(); // then clear all Elements
        return; // and return
    }

    if (maxNumSelectedElements == 1) {
        // If only one can be selected, clear the existing one...
        ClearElements(); // ...so it can be replaced
    }

    // Can't select more than maxNumSelectedElements simultaneously
    if (selectedElements.Count >= maxNumSelectedElements) return;

    // It's okay to add this element
    GameObject go = Instantiate(elementPrefabs[(int) elType]) as GameObject;
    // ^ Note the typecast from ElementType to int in the line above
    Element el = go.GetComponent<Element>();
    el.transform.parent = this.transform;

    selectedElements.Add(el); // Add el to the list of selectedElements
}

// Clears all elements from selectedElements and destroys their GameObjects
public void ClearElements() {
    foreach (Element el in selectedElements) {
        // Destroy each GameObject in the list
        Destroy(el.gameObject);
    }
    selectedElements.Clear(); // and clear the list
}

// Called every Update() to orbit the elements around
void OrbitSelectedElements() {
    // If there are none selected, just return
    if (selectedElements.Count == 0) return;

    Element el;
    Vector3 vec;
    float theta0, theta;
    float tau = Mathf.PI*2; // tau is 360° in radians (i.e. 6.283...)

    // Divide the circle into the number of elements that are orbiting
    float rotPerElement = tau / selectedElements.Count;

    // The base rotation angle (theta0) is set based on time
}

```

```
theta0 = elementRotSpeed * Time.time * tau;

for (int i=0; i<selectedElements.Count; i++) {
    // Determine the rotation angle for each element
    theta = theta0 + i*rotPerElement;
    el = selectedElements[i];
    // Use simple trigonometry to turn the angle into a unit vector
    vec = new Vector3(Mathf.Cos(theta),Mathf.Sin(theta),0);
    // Multiply that unit vector by the elementRotDist
    vec *= elementRotDist;
    // Raise the element to waist height.
    vec.z = -0.5f;
    el.lPos = vec;      // Set the position of the Element_Sphere
}
}
```

Once this has been saved and compiled, return to Unity and select `_Mage` in the Hierarchy. You will see that the new fields have been added to the `Mage (Script)` Inspector. Open the disclosure triangle next to the `elementPrefabs` field and set its `Size` to 4. Then drag each of the `Element_Sphere` prefabs into the `elementPrefabs` array in the same order as the enum: Earth, Water, Air, and Fire. When finished, your `_Mage.Mage (Script)` Inspector should look like that in [Figure 35.6](#).

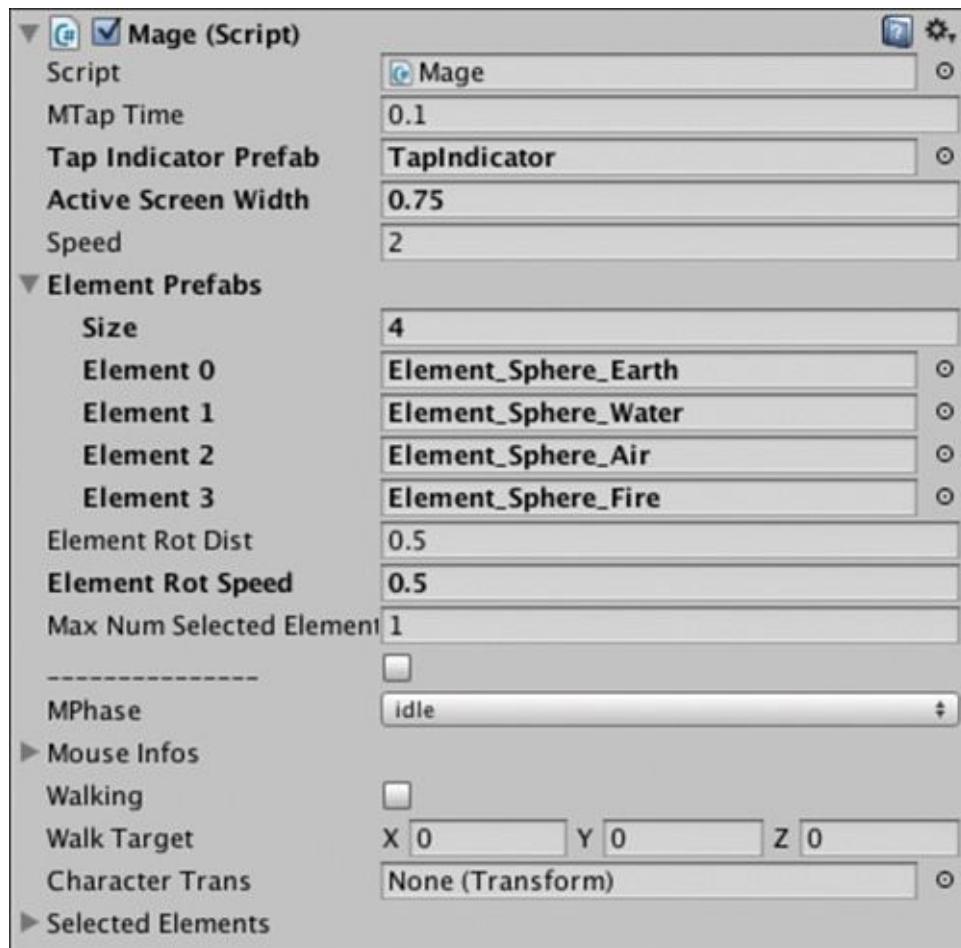


Figure 35.6 Mage (Script) Inspector for `_Mage` showing `elementPrefabs`

Open the ElementInventoryButton script in MonoDevelop and remove the comment (//) on the line that calls Mage.S.SelectElement():

[Click here to view code image](#)

```
void OnMouseUpAsButton() {  
    // Tell the Mage to add this element type  
Mage.S.SelectElement(type);
```

}

Now, when you press Play, you can select a single element, and that element will orbit around the Mage. Clicking the black none element will clear your selection. If you want, you can set `maxNumSelectedElements` in the *Mage (Script)* Inspector to a higher number (like 4) to see how multiple `Element_Spheres` would orbit the Mage, but for the purposes of this chapter, set `maxNumSelectedElements` back to 1 when you're done checking that out.

Now that you can select elements, let's do something with them. The game is designed to be able to handle two kinds of spells:

- *Ground spells* are cast on the ground itself and are used for area-of-effect damage to or protection from enemies.
- *Attack spells* are cast directly at a single enemy.

In this chapter, you will be building the fire ground spell.

Casting the Fire Ground Spell

The spell we'll be creating is the fire ground spell. To cast this spell, the player will select a single element and then draw a colored path on the ground that the spell will follow, sprouting a line of flame to discourage enemies.

For this to work, we need more information about where the player initially started the mouse interaction or touch: Was it over the ground, an enemy, or the Mage? This information will change how the rest of the mouse gesture is interpreted. This will be accomplished by adding tags to various `GameObjects` (including the Mage, the `TilePrefab`, and various enemies) and then using the `Utils.FindTaggedParent()` function from the SHMUP prototype. However, because `Utils.FindTaggedParent()` will return null if nothing in the parent Hierarchy is tagged, that case needs to be handled as well.

Add the bolded code to the Mage C# script:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...
    public MPhase mPhase = MPhase.idle;
    public List<MouseInfo> mouseInfos = new List<MouseInfo>();
    public string actionStartTag; // ["Mage", "Ground", "Enemy"]

    public bool walking = false;
    ...

    void MouseDown() {
        // The mouse was pressed on something (it could be a drag or tap)
        if (DEBUG) print("Mage.MouseDown()");
        GameObject clickedGO = mouseInfos[0].hitInfo.collider.gameObject;
        // ^ If the mouse wasn't clicked on anything, this would throw an error
        // because hitInfo would be null. However, we know that MouseDown()
        // is only called when the mouse WAS clicking on something, so
        // hitInfo is guaranteed to be defined.

        GameObject taggedParent = Utils.FindTaggedParent(clickedGO);
        if (taggedParent == null) {
            actionStartTag = "";
        } else {
```

```

        actionStartTag = taggedParent.tag;
        // ^ this should be either "Ground", "Mage", or "Enemy"
    }

}

void MouseTap() {
    // Something was tapped like a button
    if (DEBUG) print("Mage.MouseTap()");

    // Now this cares what was tapped
    switch (actionStartTag) {
        case "Mage":
            // Do nothing
            break;
        case "Ground":
            // Move to tapped point @ z=0 whether or not an element is selected
            WalkTo(lastMouseInfo.loc); // Walk to the first mouseInfo pos
            ShowTap(lastMouseInfo.loc); // Show where the player tapped
            break;
    }
}

void MouseDrag() {
    // The mouse is being drug across something
    if (DEBUG) print("Mage.MouseDrag()");

    // Drag is meaningless unless the mouse started on the ground
    if (actionStartTag != "Ground") return;

    // If there is no element selected, the player should follow the mouse
    if (selectedElements.Count == 0) {
        // Continuously walk toward the current mouseInfo pos
        WalkTo(mouseInfos[mouseInfos.Count-1].loc);
    }
}

void MouseDragUp() {
    // The mouse is released after being drug
    if (DEBUG) print("Mage.MouseDragUp()");

    // Drag is meaningless unless the mouse started on the ground
    if (actionStartTag != "Ground") return;

    // If there is no element selected, stop walking now
    if (selectedElements.Count == 0) {
        // Stop walking when the drag is stopped
        StopWalking();
    }
}

...
}

```

Once this code is done, the tags need to be added to a couple of objects. Because of the `Utils.FindTaggedParent()` method, only the topmost parent needs to have the tag, which will help a lot with complex objects—like the `Mage`—that have several colliders on them.

Select `_Mage` in the Hierarchy and choose *Add Tag* from the Tag pop-up menu at the top of the `_Mage` Inspector. In the Tags array, add tags for `Mage`, `Ground`, and `Enemy`. Once this is complete, select `_Mage` in the Hierarchy again and set its tag to `Mage`. Select `TilePrefab` in the `_Prefabs` folder of the Project pane and set its tag to `Ground`. Now save your scene and press Play.

You should see now that tapping on `_Mage` or dragging from `_Mage` to anywhere else doesn't do

anything. Tapping on the ground still moves _Mage like normal, and dragging on the ground causes _Mage to follow only if no elements are selected.

Using a LineRenderer to Cast a Ground Spell

When an element is selected, we want to draw a line on the ground in the color of that element and then cast an elemental ground spell (in this case, a fire spell). The line can be drawn through the use of a LineRenderer, which is the same standard Unity component that is used for the built-in TrailRenderer. However, the LineRenderer is a little bit lower level and more controllable.

Only one line needs to be drawn at a time, so we only need a single LineRenderer component. Select _Mage in the Hierarchy and from the menu bar select *Component > Effects > Line Renderer*. You'll see that there is now an ugly pink line segment near your _Mage. We'll soon make it look much nicer.

You'll also notice that there is now a LineRenderer component in the _Mage Inspector. Open the disclosure triangle next to *Materials* in the LineRenderer Inspector and click the little circle to the right of Element 0. This will bring up a list of all the materials in the project. Choose *Default-Particle* from the end of the list (it's one of the few default materials included in every Unity project). This material works particularly well for lines, trails, and simple particles. The line should now look a lot nicer.

Uncheck the boxes next to *Cast Shadows* and *Receive Shadows* in the LineRenderer Inspector. Open the disclosure triangle next to *Parameters* and set both *Start Width* and *End Width* to 0.2. Finally, now that the LineRenderer is ready, uncheck the box next to LineRenderer to disable it. (We'll enable it through code when needed.) When finished, your LineRenderer settings should look like those in the left image of [Figure 35.7](#).

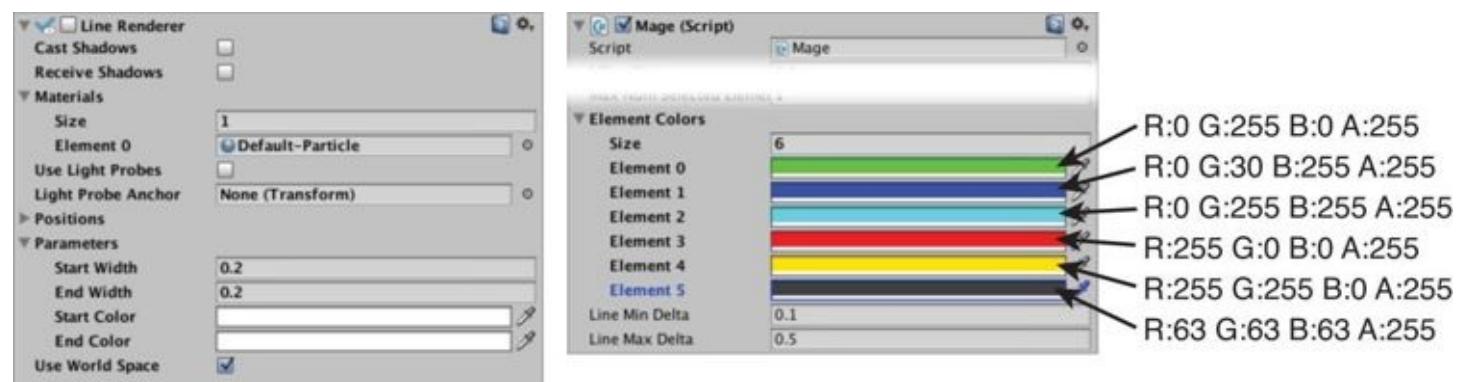


Figure 35.7 _Mage:LineRenderer settings and _Mage:ElementColor settings

Open the Mage script in MonoDevelop and add the bolded code that follows to implement the line renderer:

[Click here to view code image](#)

```
public class Mage : PT_MonoBehaviour {
    ...
    public int maxNumSelectedElements = 1;
    public Color[] elementColors;

    public bool _____;

    public List<Vector3> linePts; // Points to be shown in the line
    protected LineRenderer liner; // Ref to the LineRenderer Component
    protected float lineZ = -0.1f; // Z depth of the line
    // ^ protected variables are between public and private.
    //    public variables can be seen by everyone
    //    private variables can only be seen by this class
```

```

// protected variables can be seen by this class or any subclasses
// only public variables appear in the Inspector
// (or those with [SerializeField] in the preceding line)
public MPhase mPhase = MPhase.idle;
...

void Awake() {
    S = this; // Set the Mage Singleton
    mPhase = MPhase.idle;

    // Find the characterTrans to rotate with Face()
    characterTrans = transform.Find("CharacterTrans");

    // Get the LineRenderer component and disable it
    liner = GetComponent<LineRenderer>();
    liner.enabled = false;
}

...
void MouseDrag() {
    ...
    // If there is no element selected, the player should follow the mouse
    if (selectedElements.Count == 0) {
        // Continuously walk towards the current mouseInfo pos
        WalkTo(mouseInfos[mouseInfos.Count-1].loc);
    } else {
        // This is a ground spell, so we need to draw a line
        AddPointToLiner( mouseInfos[mouseInfos.Count-1].loc );
        // ^ add the most recent MouseInfo.loc to liner
    }
}

void MouseDragUp() {
    ...
    // If there is no element selected, stop walking now
    if (selectedElements.Count == 0) {
        // Stop walking when the drag is stopped
        StopWalking();
    } else {
        //TODO: Cast the Spell

        // Clear the liner
        ClearLiner();
    }
}

...
void OrbitSelectedElements() {
    ...
}

//----- LineRenderer Code -----
// Add a new point to the line.
void AddPointToLiner(Vector3 pt) {
    pt.z = lineZ; // Set the z of the pt to lineZ to elevate it slightly
                  // above the ground

    linePts.Add(pt);
    UpdateLiner();
}

```

```

}

// Update the LineRenderer with the new points
public void UpdateLiner() {
    // Get the type of the selectedElement
    int el = (int) selectedElements[0].type;

    // Set the line color based on that type
    liner.SetColors(elementColors[el],elementColors[el]);

    // Update the representation of the ground spell about to be cast
    liner.SetVertexCount(linePts.Count); // Set the number of vertices
    for (int i=0; i<linePts.Count; i++) {
        liner.SetPosition(i, linePts[i]); // Set each vertex
    }
    liner.enabled = true; // Enable the LineRenderer
}

public void ClearLiner() {
    liner.enabled = false; // Disable the LineRenderer
    linePts.Clear(); // and clear all linePts
}

}

```

Once you save and compile this, you're going to need to set up the `elementColors` array in the `_Mage: Mage (Script)` Inspector. Fill the 6 elements of `elementColors` with the colors shown in the right image of [Figure 35.7](#). Save your scene and press Play. Now, when you select an element and drag on the ground, a line shows where the spell will be cast. However, the line probably looks a little glitchy now. This is because some points in the line are too close together, and others are too far apart. The following changes to the Mage class code will improve this. It will also add a maximum total line length, after which point, the line will no longer draw. This will prevent players from casting ridiculously long spells.

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...
    public Color[] elementColors;

    // These set the min and max distance between two line points
    public float lineMinDelta = 0.1f;
    public float lineMaxDelta = 0.5f;
    public float lineMaxLength = 8f;

    public bool _____;

    public float totalLineLength;
    public List<Vector3> linePts; // Points to be shown in the line

    ...
    //----- LineRenderer Code -----//

    // Add a new point to the line. This ignores the point if it's too close to
    // existing ones and adds extra points if it's too far away
    void AddPointToLiner(Vector3 pt) {
        pt.z = lineZ; // Set the z of the pt to lineZ to elevate it slightly
        // above the ground
    }
}

```

```

//linePts.Add(pt);           // COMMENT OUT OR DELETE THESE TWO LINES!!!
//UpdateLiner();           // COMMENT OUT OR DELETE THESE TWO LINES!!!

// Always add the point if linePts is empty...
if (linePts.Count == 0) {
    linePts.Add (pt);
    totalLineLength = 0;
    return; // ...but wait for a second point to enable the LineRenderer
}

// If the line is too long already, return
if (totalLineLength > lineMaxLength) return;

// If there is a previous point (pt0), then find how far pt is from it
Vector3 pt0 = linePts[linePts.Count-1]; // Get the last point in linePts
Vector3 dir = pt-pt0;
float delta = dir.magnitude;
dir.Normalize();

totalLineLength += delta;

// If it's less than the min distance
if ( delta < lineMinDelta ) {
    // ...then it's too close; don't add it
    return;
}

// If it's further than the max distance then extra points...
if (delta > lineMaxDelta) {
    // ...then add extra points in between
    float numToAdd = Mathf.Ceil(delta/lineMaxDelta);
    float midDelta = delta/numToAdd;
    Vector3 ptMid;
    for (int i=1; i<numToAdd; i++) {
        ptMid = pt0+(dir*midDelta*i);
        linePts.Add(ptMid);
    }
}

linePts.Add(pt); // Add the point
UpdateLiner(); // And finally update the line
}
...
}

```

Now the line should display much more smoothly, and it should stop drawing at the point that it's about 8 meters long (that is, 8 Unity units long).

The Fire Spell

Now that we can see the line where the spell will go, it's time to cast a spell. Drag FireGroundSpellPrefab from the _Prefabs folder of the Project pane into the scene, and you can get a preview of what the spell will look like. You can also take this time to examine its *Particle System* component, which is generating all of the fire particles. Once you're done exploring it, delete the instance from the Hierarchy (though not from the Project pane, of course).

Open the Mage script and add the following code:

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...

```

```

public GameObject fireGroundSpellPrefab;

public bool _____;

protected Transform spellAnchor; // The parent transform for all spells

...

void Awake() {
    ...
    liner.enabled = false;

    GameObject saGO = new GameObject("Spell Anchor");
    // ^ Create an empty GameObject named "Spell Anchor". When you create a
    // new GameObject this way, it's at P:[0,0,0] R:[0,0,0] S:[1,1,1]
    spellAnchor = saGO.transform; // Get its transform
}

...

void MouseDragUp() {
    ...
    // If there is no element selected, stop walking now
    if (selectedElements.Count == 0) {
        // Stop walking when the drag is stopped
        StopWalking();
    } else {
        CastGroundSpell();
        // Clear the liner
        ClearLiner();
    }
}

void CastGroundSpell() {
    // There is not a no-element ground spell, so return
    if (selectedElements.Count == 0) return;

    // Because this version of the prototype only allows a single element to
    // be selected, we can use that 0th element to pick the spell.
    switch (selectedElements[0].type) {
        case ElementType.fire:
            GameObject fireGO;
            foreach( Vector3 pt in linePts ) { // For each Vector3 in linePts...
                // ...create an instance of fireGroundSpellPrefab
                fireGO = Instantiate(fireGroundSpellPrefab) as GameObject;
                fireGO.transform.parent = spellAnchor;
                fireGO.transform.position = pt;
            }
            break;
    }
    // TODO: Add other elements types later
}

    // Clear the selectedElements; they're consumed by the spell
    clearElements();
}

...
}

```

Save the Mage script and return to Unity. You will now see a `fireGroundSpellPrefab` field on the `_Mage.Mage (Script)` Inspector. Drag the `FireGroundSpellPrefab` from the `_Prefabs` folder of the Project pane into this field and save your scene. Now, when you play the scene, you can choose a fire

element and cast a fire spell on the ground. However, that spell lasts forever right now.

Fire-and-Forget Spells

One of the aspects of the way this game is designed is that spells should be able to be cast by the Mage script and then forgotten. Rather than the Mage script managing the duration, damage, and behavior of the spell, the individual spell prefabs can manage themselves. To accomplish this for the fire ground spell, create a new C# script in the `_Scripts` folder named `FireGroundSpell` and attach it to the `FireGroundSpellPrefab` in the `_Prefabs` folder. Open the script in MonoDevelop and enter this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

// Extends PT_MonoBehaviour
public class FireGroundSpell : PT_MonoBehaviour {

    public float duration = 4; // Lifetime of this GameObject
    public float durationVariance = 0.5f;
    // ^ This allows the duration to range from 3.5 to 4.5
    public float fadeTime = 1f; // Length of time to fade
    public float timeStart; // Birth time of this GameObject

    // Use this for initialization
    void Start () {
        timeStart = Time.time;
        duration = Random.Range(duration-durationVariance,
                                duration+durationVariance);
        // ^ Set the duration to a number between 3.5 and 4.5 (defaults)
    }

    // Update is called once per frame
    void Update () {
        // Determine a number [0..1] (between 0 and 1) that stores the
        // percentage of duration that has passed
        float u = (Time.time-timeStart)/duration;

        // At what u value should this start fading
        float fadePercent = 1-(fadeTime/duration);
        if (u>fadePercent) { // If it's after the time to start fading...
            // ...then sink into the ground
            float u2 = (u-fadePercent)/(1-fadePercent);
            // ^ u2 is a number [0..1] for just the fadeTime
            Vector3 loc = pos;
            loc.z = u2*2; // move lower over time
            pos = loc;
        }

        if (u>1) { // If this has lived longer than duration...
            Destroy(gameObject); // ...destroy it
        }
    }

    void OnTriggerEnter(Collider other) {
        // Announce when another object enters the collider
        GameObject go = Utils.FindTaggedParent(other.gameObject);
        if (go == null) {
            go = other.gameObject;
        }
        Utils.tr("Flame hit", go.name);
    }
}
```

```
//TODO: Actually damage the other object
```

```
}
```

Now, when you play the scene, you will see that after you cast the fire ground spell, each spurt of fire lasts for about 4 seconds and dies out. In addition, if you make the Mage walk through the fire, the “Flame hit” announcement will appear in the Console. The Mage’s spells won’t damage her of course, so let’s give her a way to look for some enemies that she can damage.

Changing Rooms

As a kindness to our Mage, the first room doesn’t have any monsters in it, but there will certainly be some throughout the dungeon. The Mage needs a way to move from room to room. This will be accomplished with the PortalPrefab in the `_Prefabs` folder and some scripting. If you inspect the PortalPrefab, note that it is in the *Ignore Raycast* layer. Unity automatically includes this layer in every project, and we’re using it here to ensure that the `Physics.Raycast()` done by `MouseInfo` ignores the PortalPrefab and falls through to the Tile beyond it.

Create a new C# script named *Portal* and place it in the `_Scripts` folder. Then, attach it to the PortalPrefab prefab in the `_Prefabs` folder. Open the Portal script in MonoDevelop and enter the following code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public class Portal : PT_MonoBehaviour {

    public string      toRoom;
    public bool        justArrived = false;
    // ^ true if _Mage has just teleported here

    void OnTriggerEnter(Collider other) {
        if (justArrived) return;
        // ^ Since the Mage has just arrived, don't teleport her back

        // Get the GameObject of the collider
        GameObject go = other.gameObject;
        // Search up for a tagged parent
        GameObject goP = Utils.FindTaggedParent(go);
        if (goP != null) go = goP;

        // If this isn't the _Mage, return
        if (go.tag != "Mage") return;

        // Go ahead and build the next room
        LayoutTiles.S.BuildRoom(toRoom);
    }

    void OnTriggerExit(Collider other) {
        // Once the Mage leaves this Portal, set justArrived to false
        if (other.gameObject.tag == "Mage") {
            justArrived = false;
        }
    }
}
```

The `justArrived` bool is important because it keeps the Mage from immediately being teleported back to the previous room. If `justArrived` wasn't there, the Mage appearing in the new room (above the Portal) would invoke `OnTriggerEnter()` and be sent back to the previous room.

Now, we need to make several changes to the `LayoutTiles` script to allow it to build rooms more than once (and to allow it to destroy the old room Tiles that are being replaced). Open `LayoutTiles` and edit the code as shown in bold here:

[Click here to view code image](#)

```
public class LayoutTiles : MonoBehaviour {
    ...
    public TileTex[] tileTextures; // A list of named textures for Tiles
    public GameObject portalPrefab; // Prefab for the portals between rooms

    public bool justArrived = false;

    private bool firstRoom = true; // Is this the first room built?
    public PT_XMLReader roomsXMLR;
    ...

    public Texture2D GetTileTex(string tStr) {
        ...
    }

    // Build a room based on room number. This is an alternative version of
    // BuildRoom that grabs roomXML based on <room> num.
    public void BuildRoom(string rNumStr) {
        PT_XMLHashtable roomHT = null;
        for (int i=0; i<roomsXML.Count; i++) {
            PT_XMLHashtable ht = roomsXML[i];
            if (ht.att("num") == rNumStr) {
                roomHT = ht;
                break;
            }
        }
        if (roomHT == null) {
            Utils.tr("ERROR", "LayoutTiles.BuildRoom()", "Room not found: "+rNumStr);
            return;
        }
        BuildRoom(roomHT);
    }

    // Build a room from an XML <room> entry
    public void BuildRoom(PT_XMLHashtable room) {
        // Destroy any old Tiles
        foreach (Transform t in tileAnchor) { // Clear out old tiles
            // ^ You can iterate over a Transform to get its children
            Destroy(t.gameObject);
        }

        // Move the Mage out of the way
        Mage.S.pos = Vector3.left * 1000;
        // This keeps the Mage from accidentally triggering OnTriggerExit() on
        // a Portal. In my testing, I found that OnTriggerExit was being called
        // at strange times.
        Mage.S.ClearInput(); // Cancel any active mouse input and drags

        string rNumStr = room.att("num");

        // Get the texture names for the floors and walls from <room> attributes
```

```

...
float maxY = roomRows.Length-1;
List<Portal> portals = new List<Portal>();

// These loops scan through each tile of each row of the room
for (int y=0; y<roomRows.Length; y++) {
    for (int x=0; x<roomRows[y].Length; x++) {
        ...
        ti.tex = tileTexStr;

        // Check for specific entities in the room
        switch (rawType) {
            case "X": // Starting position for the Mage
                // Mage.S.pos = ti.pos;           // COMMENT OUT THIS LINE!
                if (firstRoom) {
                    Mage.S.pos = ti.pos; // Uses the Mage Singleton
                    roomNumber = rNumStr;
                    // ^ Setting roomNumber now keeps any portals from
                    //     moving the Mage to them in this first room.
                    firstRoom = false;
                }
                break;

            case "0": // Numbers are room portals (up to F in hexadecimal)
            case "1": // This allows portals to be placed in the Rooms.xml file
            case "2":
            case "3":
            case "4":
            case "5":
            case "6":
            case "7":
            case "8":
            case "9":
            case "A":
            case "B":
            case "C":
            case "D":
            case "E":
            case "F":
                // Instantiate a Portal
                GameObject pGO = Instantiate(portalPrefab) as GameObject;
                Portal p = pGO.GetComponent<Portal>();
                p.pos = ti.pos;
                p.transform.parent = tileAnchor;
                // ^ Attaching this to the tileAnchor means that the Portal
                //     will be Destroyed when a new room is built
                p.toRoom = rawType;
                portals.Add(p);
                break;
        }
    }
}

// More to come here...

}

}

// Position the Mage
foreach (Portal p in portals) {
    // If p.toRoom is the same as the room number the Mage just exited,
    // then the Mage should enter this room through this Portal
    // Alternatively, if firstRoom == true and there was no X in the
    // room (as a default Mage starting point), move the Mage to this
    // Portal as a backup measure (if, for instance, you want to just
}

```

```

        // load room number "5")
        if (p.toRoom == roomNumber || firstRoom) {
            // ^ If there's an X in the room, firstRoom will be set to false
            // by the time the code gets here
            Mage.S.StopWalking(); // Stop any Mage movement
            Mage.S.pos = p.pos; // Move _Mage to this Portal location
            // _Mage maintains her facing from the previous room, so there
            // is no need to rotate her in order for her to enter this room
            // facing the right direction.
            p.justArrived = true;
            // ^ Tell the Portal that Mage has just arrived.
            firstRoom = false;
            // ^ Stops a 2nd Portal in this room from moving the Mage to it
        }
    }

    // Finally assign the roomNumber
    roomNumber = rNumStr;
}
}
}

```

For this code to compile properly, you need to add a few lines to Mage as well. Open Mage in MonoDevelop and add the `ClearInput()` method to the end of the Mage class:

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...
    public void ClearLiner() {
        liner.enabled = false;      // Disable the LineRenderer
        linePts.Clear();           // and clear all linePts
    }

    // Stop any active drag or other mouse input
    public void ClearInput() {
        mPhase = MPhase.idle;
    }
}

```

Save and switch back to Unity. Select `_MainCamera` in the Hierarchy and find the `_MainCamera.LayoutTiles (Script)` Inspector. Drag `PortalPrefab` from the `_Prefabs` folder in the Project pane into the `portalPrefab` field of the `LayoutTiles (Script)` Inspector. Now is a good time to also make sure that the `roomNumber` field is set to 0 so that your Mage starts in the correct room.

Save your scene, press Play, and you'll see that the Mage can now walk from room to room by walking into portals. Try exploring the entire dungeon if you like. Now that the Mage can explore, it's time to give her some enemies.

Spawning Enemies

In addition to storing information about the Tile layout of the room, the `Rooms.xml` file also includes information about two different kinds of enemies: bugs and spikers. These are represented in the `Rooms.xml` file by a `b` for the bug and a `^`, `v`, `{`, or `}` for the spikers (because the spikers need to have an initial direction set for them). Note that the `<` and `>` angle brackets couldn't be used because they are special reserved characters in XML files.

All Enemies

All enemies in the game will share certain traits. Each will have some sort of simple movement. Most will be damaged by spells. Each will damage the Mage if they collide with her. Each will have a collider of some sort that is used to determine what it is colliding with.

Because so many aspects of all enemies are shared, it makes sense to handle them in some sort of hierarchical, object-oriented way. In the SHMUP prototype, we implemented an `Enemy` superclass with several subclasses. For this prototype—to demonstrate something different—we'll implement an *interface* instead. Look at the “[Interfaces](#)” section in [Appendix B](#), “[Useful Concepts](#),” for an introduction to the concept.

Briefly, an interface declares methods or properties that will later be implemented by a class. Any class that implements the interface can then be referred to in code as that interface type rather than as the specific class. This differs from subclassing in several ways, one of the most interesting of which is that a class may implement several different interfaces simultaneously, whereas a class can only extend a single superclass.

We will also create a *factory* to instantiate the various enemy types for us. This will be an adaptation of the classic Factory Pattern described in the book *Design Patterns* by the “Gang of Four.”¹ When the `LayoutTiles.BuildRoom()` method encounters a character that it doesn't understand (like the `b`, `^`, `v`, `,`, or `}`), it will pass it to the `EnemyFactory.Instantiate()` method to see whether an enemy should be created. A diagram of this concept is shown in [Figure 35.8](#).

`EnemyFactory.Instantiate()` is a “factory” because it can create any number of different classes and return them to `LayoutTiles` as long as they all implement the `Enemy` interface. For more information about other design patterns, see the “[Software Design Patterns](#)” section of [Appendix B](#).

¹ Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994. The Factory Pattern is one of many described in the book. Others include the Singleton Pattern that has been used in many of the tutorials in this book.

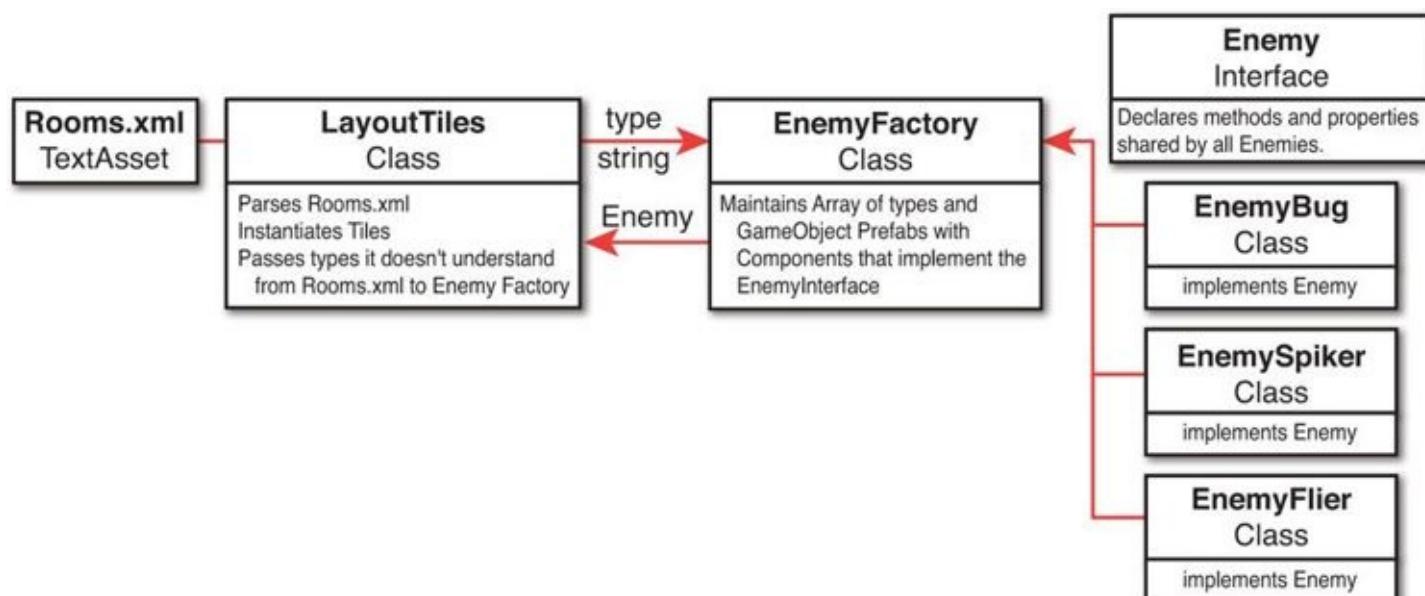


Figure 35.8 Conceptual structure of `EnemyFactory`

There are two ways to approach designing something like this. One is to think really hard about it ahead of time and try to foresee all the possible parts of each `Enemy` type's behavior that you want to abstract into the interface. The other is to just build a couple of enemies, see what methods they have in common, and then add those to the interface. The second method is much more akin to the iterative process of design that is used throughout this book, though it can lead to less flexibility and

extensibility. Most of the time, I do a mixture of the two methods with a weighting toward the second. I try to just start coding something with both the knowledge that I will have to refactor some of the code later and a weather eye toward things that I will want to abstract into an interface or superclass later. With that in mind, let's make the *EnemyBug* work.

EnemyBug

Bugs are a very basic kind of enemy. They will simply move toward the Mage if she's in the same room with them. Bugs have no knowledge of walls or the layout of the room and will therefore happily walk into a wall rather than navigating around to get to the player. (If you want to make them more intelligent later, you can add A* pathfinding (pronounced "A-star") or use Unity's navigation mesh.) Bugs are damaged by—but unafraid of—fire and will damage the Mage if they touch her.

Let's start by making the *EnemyBug* chase after the player. Create a new C# script named *EnemyBug*, and place it in the *_Scripts* folder. Then drag it on to *EnemyBug* in the *_Prefabs* folder.

Next, drag an instance of *EnemyBug* from the *_Prefabs* folder into the Hierarchy. Ensure that its position is P:[8,4,0], which will place it in the same room with the player once *LayoutTiles* has built the 0th room (but still keep it far enough away that you have lots of time to cast a spell on it before it reaches you). If you explore *EnemyBug* in the Hierarchy, you can see that it is constructed very much like *_Mage*, with a top-level GameObject (*EnemyBug*) that contains a Rigidbody and CapsuleCollider. *EnemyBug* parents a child named *CharacterTrans* that is used to parent and rotate a child named *View_Bug* that actually has the model of the Bug seen by the player. As with *View_Character*, *View_Bug* also has individual colliders on each of its body parts. This is more important with *View_Bug* than *View_Character* because *View_Bug*'s legs stick out farther than the boundary of the CapsuleCollider on *EnemyBug*.

Open the *EnemyBug* script and enter the following code. As you can see, it's almost entirely cut and pasted from *Mage*. In general, when you're coding large systems, you don't want to cut and paste code like this. However, when prototyping, it's very useful to copy and paste at first and then—once you know exactly what you want—you can abstract the code later.

[Click here to view code image](#)

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

public class EnemyBug : PT_MonoBehaviour {
    public float speed = 0.5f;

    public bool _____;

    public Vector3 walkTarget;
    public bool walking;
    public Transform characterTrans;

    void Awake() {
        characterTrans = transform.Find("CharacterTrans");
    }

    void Update() {
        WalkTo (Mage.S.pos);
    }

    // ----- Walking Code -----
}
```

```

// All of this walking code is copied directly from Mage

// Walk to a specific position. The position.z is always 0
public void WalkTo(Vector3 xTarget) {
    walkTarget = xTarget;      // Set the point to walk to
    walkTarget.z = 0;          // Force z=0
    walking = true;            // Now the EnemyBug is walking
    Face(walkTarget);         // Look in the direction of the walkTarget
}

public void Face(Vector3 poi) { // Face towards a point of interest
    Vector3 delta = poi-pos; // Find vector to the point of interest
    // Use Atan2 to get the rotation around Z that points the X-axis of
    // EnemyBug:CharacterTrans towards poi
    float rZ = Mathf.Rad2Deg * Mathf.Atan2(delta.y, delta.x);
    // Set the rotation of characterTrans (doesn't actually rotate Enemy)
    characterTrans.rotation = Quaternion.Euler(0,0,rZ);
}

public void StopWalking() { // Stops the EnemyBug from walking
    walking = false;
    rigidbody.velocity = Vector3.zero;
}

void FixedUpdate () { // Happens every physics step (i.e., 50 times/second)
    if (walking) { // If EnemyBug is walking
        if ( (walkTarget-pos).magnitude < speed*Time.fixedDeltaTime ) {
            // If EnemyBug is very close to walkTarget, just stop there
            pos = walkTarget;
            StopWalking();
        } else {
            // Otherwise, move towards walkTarget
            rigidbody.velocity = (walkTarget-pos).normalized * speed;
        }
    } else {
        // If not walking, velocity should be zero
        rigidbody.velocity = Vector3.zero;
    }
}
}

```

Press Play, and you'll see that your Mage will now be chased by an EnemyBug. If you cast a fire spell on the ground and the bug walks through it. You will also get a message about it in the Console pane. However, the fire doesn't actually hurt the bug yet.

Damaging the EnemyBug

We need to add a function that allows the bug to be hurt. Add the following code to EnemyBug:

[Click here to view code image](#)

```

public class EnemyBug : PT_MonoBehaviour {
    public float speed = 0.5f;
    public float health = 10;

    public bool _____;

    private float _____;
    public Vector3 walkTarget;
    public bool walking;
    public Transform characterTrans;

```

```

void Awake() {
    characterTrans = transform.Find("CharacterTrans");
    _maxHealth = health; // Used to put a top cap on healing
}

...
void FixedUpdate () { // Happens every physics step (i.e., 50 times/second)
    ...
}

// Damage this instance. By default, the damage is instant, but it can also
// be treated as damage over time, where the amt value would be the amount
// of damage done every second.
// NOTE: This same code can be used to heal the instance
public void Damage(float amt, bool damageOverTime=false) {
    // If it's DOT, then only damage the fractional amount for this frame
    if (damageOverTime) {
        amt *= Time.deltaTime;
    }

    health -= amt;
    health = Mathf.Min(_maxHealth, health); // Limit health if healing

    if (health <= 0) {
        Die();
    }
}

// Making Die() a separate function allows us to add things later like
// different death animations, dropping something for the player, etc.
public void Die() {
    Destroy(gameObject);
}
}
}

```

Now that the EnemyBug has the ability to be damaged, it's time to make the FireGroundSpell do so. Open FireGroundSpell and enter the following code:

[Click here to view code image](#)

```

public class FireGroundSpell : PT_MonoBehaviour {
    ...
    public float timeStart; // Birth time of this GameObject
    public float damagePerSecond = 10;

    ...

    void OnTriggerEnter(Collider other) {
        ...
    }

    void OnTriggerStay(Collider other) {
        // Actually damage the other
        // Get a reference to the EnemyBug script component of the other
        EnemyBug recipient = other.GetComponent<EnemyBug>();
        // If there is an EnemyBug component, damage it with fire
        if (recipient != null) {
            recipient.Damage(damagePerSecond, true);
        }
    }
}

```

Save, press Play, and cast a fire spell on the ground between _Mage and EnemyBug. You'll see that

the bug disappears shortly after touching the fire. However, if you pay attention, you'll notice that the bug actually dies too quickly. The fire spell does 10 damage/second, and the EnemyBug has 10 health, so it should take 1 second to die, but it's happening faster than that. This is because the bug is touching multiple individual FireGroundSpellPrefab instances, and each one is damaging it individually. The way that damage is done to the EnemyBug needs to be modified so that the bug doesn't take compound damage from multiple instances of the same spell. We need to modify the Damage () method of EnemyBug so that it also takes information on the type of damage that is done and treats different types of damage differently. First, replace the recipient.Damage () line you just added to the end of FireGroundSpell with this one:

[Click here to view code image](#)

```
// If there is an EnemyBug component, damage it with fire
if (recipient != null) {
    recipient.Damage(damagePerSecond, ElementType.fire, true);
}
```

Then, edit EnemyBug as shown in the following code. Note that you'll be replacing most of the Damage () method:

[Click here to view code image](#)

```
public class EnemyBug : PT_MonoBehaviour {
    ...
    public Transform characterTrans;
    // Stores damage for each element each frame
    public Dictionary<ElementType, float> damageDict;
    // ^ NOTE: Dictionaries do not appear in the Unity Inspector

    void Awake() {
        characterTrans = transform.Find("CharacterTrans");
        _maxHealth = health; // Always starts with max health
        ResetDamageDict();
    }

    // Resets the values for the damageDict
    void ResetDamageDict() {
        if (damageDict == null) {
            damageDict = new Dictionary<ElementType, float>();
        }
        damageDict.Clear();
        damageDict.Add(ElementType.earth, 0);
        damageDict.Add(ElementType.water, 0);
        damageDict.Add(ElementType.air, 0);
        damageDict.Add(ElementType.fire, 0);
        damageDict.Add(ElementType.aether, 0);
        damageDict.Add(ElementType.none, 0);
    }

    ...

    // Damage this instance. By default, the damage is instant, but it can also
    // be treated as damage over time, where the amt value would be the amount
    // of damage done every second.
    // NOTE: This same code can be used to heal the instance
    public void Damage(float amt, ElementType eT, bool damageOverTime=false) {
        // If it's DOT, then only damage the fractional amount for this frame
        if (damageOverTime) {
            amt *= Time.deltaTime;
        }
    }
}
```

```

// Treat different damage types differently (most are default)
switch (eT) {
    case ElementType.fire:
        // Only the max damage from one fire source affects this instance
        damageDict[eT] = Mathf.Max ( amt, damageDict[eT] );
        break;

    case ElementType.air:
        // air doesn't damage EnemyBugs, so do nothing
        break;

    default:
        // By default, damage is added to the other damage by same element
        damageDict[eT] += amt;
        break;
}

}

// LateUpdate() is automatically called by Unity every frame. Once all the
// Updates() on all instances have been called, then LateUpdate() is called
// on all instances.
void LateUpdate() {
    // Apply damage from the different element types

    // Iteration through a Dictionary uses a KeyValuePair
    // entry.Key is the ElementType, while entry.Value is the float
    float dmg = 0;
    foreach ( KeyValuePair<ElementType, float> entry in damageDict ) {
        dmg += entry.Value;
    }

    health -= dmg;
    health = Mathf.Min(_maxHealth, health); // Limit health if healing

    ResetDamageDict(); // Prepare for next frame

    if (health <= 0) {
        Die();
    }
}

...
}

```

Now, it takes a full second of fire damage for the bug to die. If you want to double-check that the timing works properly, you can increase the bug's damage to 100 and see that it does indeed take 10 seconds for it to die.

Showing Damage

Right now, there is no indication to the player that the bug is actually being damaged until it dies. Many games flash something red when it's damaged (we did this in the SHMUP prototype), but color already has a lot of meaning in this game, so we should alter something other than color when a character is damaged. Instead of flashing red, we'll scale the model slightly using the CharacterTrans. Add the following code to EnemyBug to do so:

[Click here to view code image](#)

```

public class EnemyBug : PT_MonoBehaviour {
    public float speed = 0.5f;

```

```

public float             health = 10;
public float             damageScale = 0.8f;
public float             damageScaleDuration = 0.25f;

public bool _____;

private float           damageScaleStartTime;
...

void LateUpdate() {
    // Apply damage from the different element types

    // Iteration through a Dictionary uses a KeyValuePair
    // entry.Key is the ElementType, while entry.Value is the float
    float dmg = 0;
    foreach ( KeyValuePair<ElementType, float> entry in damageDict ) {
        dmg += entry.Value;
    }

    if (dmg > 0) { // If this took damage...
        // and if it is at full scale now (& not already damage scaling)...
        if (characterTrans.localScale == Vector3.one) {
            // start the damage scale animation
            damageScaleStartTime = Time.time;
        }
    }

    // The damage scale animation
    float damU = (Time.time - damageScaleStartTime)/damageScaleDuration;
    damU = Mathf.Min(1, damU); // Limit the max localScale to 1
    float scl = (1-damU)*damageScale + damU*1;
    characterTrans.localScale = scl * Vector3.one;

    health -= dmg;
    health = Mathf.Min(_maxHealth, health); // Limit health if healing

    ResetDamageDict(); // Prepare for next frame

    if (health <= 0) {
        Die();
    }
}
...
}

```

Now, when the EnemyBug takes damage, it will pop to a smaller size (80% its original scale) and then interpolate back up to 100% over 0.25 seconds. Save your scene, press Play, and test it with the fire spell.

Damaging the Mage

The last thing for the EnemyBug to do is damage the Mage on contact. Contact with any enemy will cause the Mage to jump backward, lose some health, and blink for 1 second of invincibility. This will be done by switching the Mage into a different mode for 1 second that controls all of these factors. Open the Mage script and edit the code as shown:

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...
    public GameObject     fireGroundSpellPrefab;

```

```

public float           health = 4; // Total mage health
public float           damageTime = -100;
// ^ Time that damage occurred. It's set to -100 so that the Mage doesn't
// act damaged immediately when the scene starts
public float           knockbackDist = 1; // Distance to move backward
public float           knockbackDur = 0.5f; // Seconds to move backward
public float           invincibleDur = 0.5f; // Seconds to be invincible
public int              invTimesToBlink = 4; // # blinks while invincible

public bool _____;

private bool           invincibleBool = false; // Is Mage invincible?
private bool           knockbackBool = false; // Mage being knocked back?
private Vector3         knockbackDir; // Direction of knockback
private Transform       viewCharacterTrans;

protected Transform     spellAnchor; // The parent transform for all spells

...
void Awake() {
    ...
    // Find the characterTrans to rotate with Face()
    characterTrans = transform.Find("CharacterTrans");
    viewCharacterTrans = characterTrans.Find("View_Character");
    ...
}

...
void FixedUpdate () { // Happens every physics step (i.e., 50 times/second)
    if (invincibleBool) {
        // Get number [0..1]
        float blinkU = (Time.time - damageTime)/invincibleDur;
        blinkU *= invTimesToBlink; // Multiply by times to blink
        blinkU %= 1.0f;
        // ^ Modulo 1.0 gives us the decimal remainder left when dividing ?blinkU
        // by 1.0. For example: 3.85f % 1.0f is 0.85f
        bool visible = (blinkU > 0.5f);
        if (Time.time - damageTime > invincibleDur) {
            invincibleBool = false;
            visible = true; // Just to be sure
        }
        // Making the GameObject inactive makes it invisible
        viewCharacterTrans.gameObject.SetActive(visible);
    }

    if (knockbackBool) {
        if (Time.time - damageTime > knockbackDur) {
            knockbackBool = false;
        }
        float knockbackSpeed = knockbackDist/knockbackDur;
        vel = knockbackDir * knockbackSpeed;
        return; // Returns to avoid walking code below
    }

    if (walking) { // If Mage is walking
        ...
    }
}

void OnCollisionEnter( Collision coll ) {
    GameObject otherGO = coll.gameObject;
}

```

```

// Colliding with a wall can also stop walking
Tile ti = otherGO.GetComponent<Tile>();
if (ti != null) {
    if (ti.height > 0) { // If ti.height is > 0
        // Then this ti is a wall, and Mage should stop
        StopWalking();
    }
}

// See if it's an EnemyBug
EnemyBug bug = coll.gameObject.GetComponent<EnemyBug>();
// If otherGO is an EnemyBug, pass otherGO to CollisionDamage()
if (bug != null) CollisionDamage(otherGO);
}

void CollisionDamage(GameObject enemy) {

    // Don't take damage if you're already invincible
    if (invincibleBool) return;

    // The Mage has been hit by an enemy
    StopWalking();
    ClearInput();

    health -= 1; // Take 1 point of damage (for now)
    if (health <= 0) {
        Die();
        return;
    }

    damageTime = Time.time;
    knockbackBool = true;
    knockbackDir = (pos - enemy.transform.position).normalized;
    invincibleBool = true;
}

// The Mage dies
void Die() {
    Application.LoadLevel(0); // Reload the level
    // ^ Eventually, you'll want to do something more elegant
}

// Show where the player tapped
...
}

```

That's it for the EnemyBug at this point. Make sure that all your work gets passed to the version of EnemyBug in the `_Prefabs` folder by clicking the *Apply* button to the right of `Prefab` in the top section of the EnemyBug Inspector. This applies all the changes you've made back to the EnemyBug Prefab. Just to be sure everything worked properly:

1. Save your scene.
2. Delete EnemyBug from the Hierarchy.
3. Drag a new instance of EnemyBug into the Hierarchy from the `_Prefabs` folder.
4. Make sure that it works the same way.

If everything works fine, you can delete the new EnemyBug instance from the Hierarchy and save the scene. If anything went wrong, you can revert back to the saved version of the scene and try applying the changes again. If for some reason this still doesn't work, drag the working version of EnemyBug

from the Hierarchy into the `_Prefabs` folder. This will make a new prefab named `EnemyBug 1`. Then delete the old `EnemyBug` prefab and rename `EnemyBug 1` to `EnemyBug`.

EnemySpiker

The next kind of enemy we'll implement is the `EnemySpiker`. The spiker moves back and forth patrolling a line. It's not affected by any spells, and it damages the `Mage` on contact.

Create a new C# script named `EnemySpiker` and place it in the `_Scripts` folder. Attach this script to the `EnemySpiker` prefab in the `_Prefabs` folder. Open the `EnemySpiker` script and enter the following code:

[Click here to view code image](#)

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

public class EnemySpiker : PT_MonoBehaviour {
    public float speed = 5f;
    public string roomXMLString = "{}";

    public bool _____;

    public Vector3 moveDir;
    public Transform characterTrans;

    void Awake() {
        characterTrans = transform.Find("CharacterTrans");
    }

    void Start() {
        // Set the move direction based on the character in Rooms.xml
        switch (roomXMLString) {
            case "^":
                moveDir = Vector3.up;
                break;
            case "v":
                moveDir = Vector3.down;
                break;
            case "{":
                moveDir = Vector3.left;
                break;
            case "}":
                moveDir = Vector3.right;
                break;
        }
    }

    void FixedUpdate () { // Happens every physics step (i.e., 50 times/second)
        rigidbody.velocity = moveDir * speed;
    }

    // This has the same structure as the Damage Method in EnemyBug
    public void Damage(float amt, ElementType eT, bool damageOverTime=false) {
        // Nothing damages the EnemySpiker
    }

    void OnTriggerEnter(Collider other) {
        // Check to see if a wall was hit
        GameObject go = Utils.FindTaggedParent(other.gameObject);
        if (go == null) return; // In case nothing is tagged
    }
}
```

```

        if (go.tag == "Ground") {
            // Make sure that the ground tile is in the direction we're moving.
            // A dot product will help us with this (see the Useful Concepts
            // Reference).
            float dot = Vector3.Dot(moveDir, go.transform.position - pos);
            if (dot > 0) { // If Spiker is moving towards the block it hit
                moveDir *= -1; // Reverse direction
            }
        }
    }
}

```

Making EnemySpiker Hurt Mage

The EnemySpiker uses a trigger (instead of a normal collider) so that collisions with other objects don't cause the spiker to go off course due to collisions. However, this also means that it currently passes harmlessly through the Mage. Make the following code edits to Mage to make EnemySpiker able to hurt her:

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...

    void OnCollisionEnter( Collision coll ) {
        ...
    }

    void OnTriggerEnter(Collider other) {
        EnemySpiker spiker = other.GetComponent<EnemySpiker>();
        if (spiker != null) {
            CollisionDamage(other.gameObject);
        }
    }

    void CollisionDamage(GameObject enemy) {
        ...
    }

    ...
}

```

To test the new EnemySpiker, complete the following steps:

1. Drag an EnemySpiker from the Project pane to the Hierarchy pane to add an instance of EnemySpiker to the scene. Be sure to position it somewhere that it will be about 1 meter (1 Unity unit) above the floor and between some walls.
2. Save the scene.
3. Play the scene to test everything and see the EnemySpiker behavior.
4. If everything seems to be working fine, delete the EnemySpiker instance from the Hierarchy and save your scene again.

Now we have two different kinds of enemies in the game that have their own behaviors and can each damage the Mage; however, if we wanted them to do different amounts of damage to the Mage—for example: if we wanted the EnemySpiker to only remove 0.5 health from the Mage instead of the 1 health that's removed by the bug—there's not a way to do that right now because then even if each of the enemies had a `touchDamage` field, there is no elegant way to pass it into the

CollisionDamage method (which only takes a GameObject as input). It's times like these when an interface can help.

Abstracting the Enemy Interface

As mentioned earlier, an interface is a great way to abstract out the commonalities of two different classes. For EnemyBug and EnemySpiker to both properly interact with the Mage when they touch her, we really only need to know two things about them:

1. Where the enemy is located so the Mage can get knocked back
2. The amount of damage the enemy does to the Mage when she touches it

As you will read in the “[Interfaces](#)” section of [Appendix B](#), an interface is like a promise that any class that implements the interface will implement specific methods and properties. Although interfaces cannot contain information about fields, that functionality can easily be handled by properties.

Create a new C# script named *Enemy* and put it in the __ Scripts folder. Open it and enter this code:

[Click here to view code image](#)

```
using UnityEngine;
using System.Collections;

public interface Enemy {
    // These are declarations of properties that will be implemented by all
    // classes that implement the Enemy interface
    Vector3 pos { get; set; } // The Enemy's transform.position
    float touchDamage { get; set; } // Damage done by touching the Enemy
}
```

Interfaces are usually very short because they only declare methods and properties that will be implemented later. Now it's time to make use of the Enemy interface. Make the following changes to the beginning of the EnemyBug script:

[Click here to view code image](#)

```
public class EnemyBug : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float _touchDamage = 1;
    public float touchDamage {
        get { return(_touchDamage); }
        set { _touchDamage = value; }
    }
    // The pos Property is already implemented in PT_MonoBehaviour

    public float speed = 0.5f;
    ...
}
```

The `EnemyBug : PT_MonoBehaviour, Enemy` tells C# that EnemyBug extends the `PT_MonoBehaviour` class *and* implements the `Enemy` interface. If there is more than one interface implemented or if there is a class and one or more interfaces, they are separated by a comma. This change to `EnemyBug` implements the promised `touchDamage` property, while the promised `pos` property is already implemented by `PT_MonoBehaviour`. The `[SerializeField]` attribute causes the private field `_touchDamage` to appear in the Inspector (even though it's private).

Now, add similar code to the beginning of `EnemySpiker`:

[Click here to view code image](#)

```

public class EnemySpiker : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float _touchDamage = 0.5f;
    public float touchDamage {
        get { return( _touchDamage ); }
        set { _touchDamage = value; }
    }
    // The pos Property is already implemented in PT_MonoBehaviour

    public float speed = 5f;
    ...
}

```

Now, both EnemyBug and EnemySpiker class instances can be treated as instances of the **Enemy** interface. Make the following changes to Mage to see how this works:

[Click here to view code image](#)

```

public class Mage : PT_MonoBehaviour {
    ...

    void OnCollisionEnter( Collision coll ) {
        ...

        // See if it's an EnemyBug
        EnemyBug bug = coll.gameObject.GetComponent<EnemyBug>();
        // If otherGO is an EnemyBug, pass bug to CollisionDamage(), which will
        // interpret it as an Enemy
        if (bug != null) CollisionDamage(bug);
        // if (bug != null) CollisionDamage(otherGO); // COMMENT OUT THIS LINE!
    }

    void OnTriggerEnter(Collider other) {
        EnemySpiker spiker = other.GetComponent<EnemySpiker>();
        if (spiker != null) {
            // CollisionDamage() will see spiker as an Enemy
            CollisionDamage(spiker);
            // CollisionDamage(other.gameObject); // COMMENT OUT THIS LINE!
        }
    }

    void CollisionDamage(Enemy enemy) {
        // Don't take damage if you're already invincible
        if (invincibleBool) return;

        // The Mage has been hit by an enemy
        StopWalking();
        ClearInput();

        health -= enemy.touchDamage; // Take damage based on Enemy
        if (health <= 0) {
            Die();
            return;
        }

        damageTime = Time.time;
        knockbackBool = true;
        knockbackDir = (pos - enemy.pos).normalized;
        invincibleBool = true;
    }

    ...
}

```

Even though `EnemyBug` and `EnemySpiker` share very little code, they can both be handled by the `CollisionDamage()` method because they both implement the `Enemy` interface. It's important to note that even though `EnemyBug` and `EnemySpiker` share a lot of fields (for example, `gameObject`, `transform`, and so on); as far as `CollisionDamage()` is concerned it can only access the two properties `pos` and `touchDamage` that are declared in the `Enemy` interface.

Making an `EnemyFactory`

The abstraction of the two enemies into the `Enemy` interface also gives us the ability to make a factory that can be used to generate enemies if passed the string representation from `Rooms.xml`. As described earlier in the chapter, a factory is a class or method that can create instances of different classes that all implement the same interface. It will be useful for us in this game because after it's implemented, adding new enemies to the factory will be as simple as editing an array in the `_MainCamera.LayoutTiles` Inspector.

First, you need to add a few more lines to the `Enemy` interface. Open `Enemy` in MonoDevelop and edit it as follows:

[Click here to view code image](#)

```
public interface Enemy {
    // These are declarations of properties that will be implemented by all
    // Classes that implement the Enemy interface
    Vector3    pos { get; set; } // The Enemy's transform.position
    float      touchDamage { get; set; } // Damage done by touching the Enemy
    string     typeString { get; set; } // The type string from Rooms.xml

    // The following are already implemented by all MonoBehaviour subclasses
    GameObject  gameObject { get; }
    Transform   transform { get; }
}
```

Make these changes to the `EnemyBug` script:

[Click here to view code image](#)

```
public class EnemyBug : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float          _touchDamage = 1;
    public float            touchDamage {
        get { return( _touchDamage ); }
        set { _touchDamage = value; }
    }
    // The pos Property is already implemented in PT_MonoBehaviour
    public string           typeString {
        get { return( roomXMLString ); }
        set { roomXMLString = value; }
    }

    public string           roomXMLString;
    public float             speed = 0.5f;
    ...
}
```

And these changes to the `EnemySpiker` script:

[Click here to view code image](#)

```
public class EnemySpiker : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float          _touchDamage = 0.5f;
```

```

public float touchDamage {
    get { return( _touchDamage ); }
    set { _touchDamage = value; }
}
// The pos Property is already implemented in PT_MonoBehaviour
public string typeString {
    get { return( roomXMLString ); }
    set { roomXMLString = value; }
}
public float speed = 5f;
public string roomXMLString = "{";
public bool _____;
...
}

```

Now, open the LayoutTiles script in MonoDevelop and make the following code changes:

[Click here to view code image](#)

```

[System.Serializable]
public class TileTex {
    ...
}

[System.Serializable]
public class EnemyDef {
    // This class enables us to define various enemies
    public string str;
    public GameObject go;
}

public class LayoutTiles : MonoBehaviour {
    ...
    public GameObject portalPrefab; // Prefab for the portals between rooms
    public EnemyDef[] enemyDefinitions; // Prefabs for Enemies

    public bool _____;

    ...
}

public void BuildRoom(PT_XMLHashtable room) {
    ...
    // These loops scan through each tile of each row of the room
    for (int y=0; y<roomRows.Length; y++) {
        for (int x=0; x<roomRows[y].Length; x++) {
            ...
            // Check for specific entities in the room
            switch (rawType) {
                ...
                case "F":
                    ...
                    portals.Add(p);
                    break;
                default:
                    // Try to see if there's an Enemy for that letter
                    Enemy en = EnemyFactory(rawType);
                    if (en == null) break; // If there's not one, break out
                    // Set up the new Enemy
                    en.pos = ti.pos;
            }
        }
    }
}

```

```

        // Make en a child of tileAnchor so it's deleted when the
        // next room is loaded.
        en.transform.parent = tileAnchor;
        en.typeString = rawType;
        break;
    }

}

...
}

public Enemy EnemyFactory(string sType) {
    // See if there's an EnemyDef with that sType
    GameObject prefab = null;
    foreach (EnemyDef ed in enemyDefinitions) {
        if (ed.str == sType) {
            prefab = ed.go;
            break;
        }
    }
    if (prefab == null) {
        Utils.tr("LayoutTiles.EnemyFactory()", "No EnemyDef for: "+sType);
        return(null);
    }

    GameObject go = Instantiate(prefab) as GameObject;

    // The generic form of GetComponent (with the <>) won't work for
    // interfaces like Enemy, so we must use this form instead.
    Enemy en = (Enemy) go.GetComponent(typeof(Enemy));

    return(en);
}
}

```

All that's left to do is add the different EnemyDefs to `_MainCamera.LayoutTiles`. Select `_MainCamera` in the Hierarchy. Open the disclosure next to `enemyDefinitions` in the *LayoutTiles (Script)* Inspector and set its length to 5. Set the five *Str* fields to *b*, *^*, *v*, *{*, and *}*. Drag *EnemyBug* from the `_Prefabs` folder into the first `Go` field and *EnemySpiker* into the remaining four. When you're done, it should look like [Figure 35.9](#).

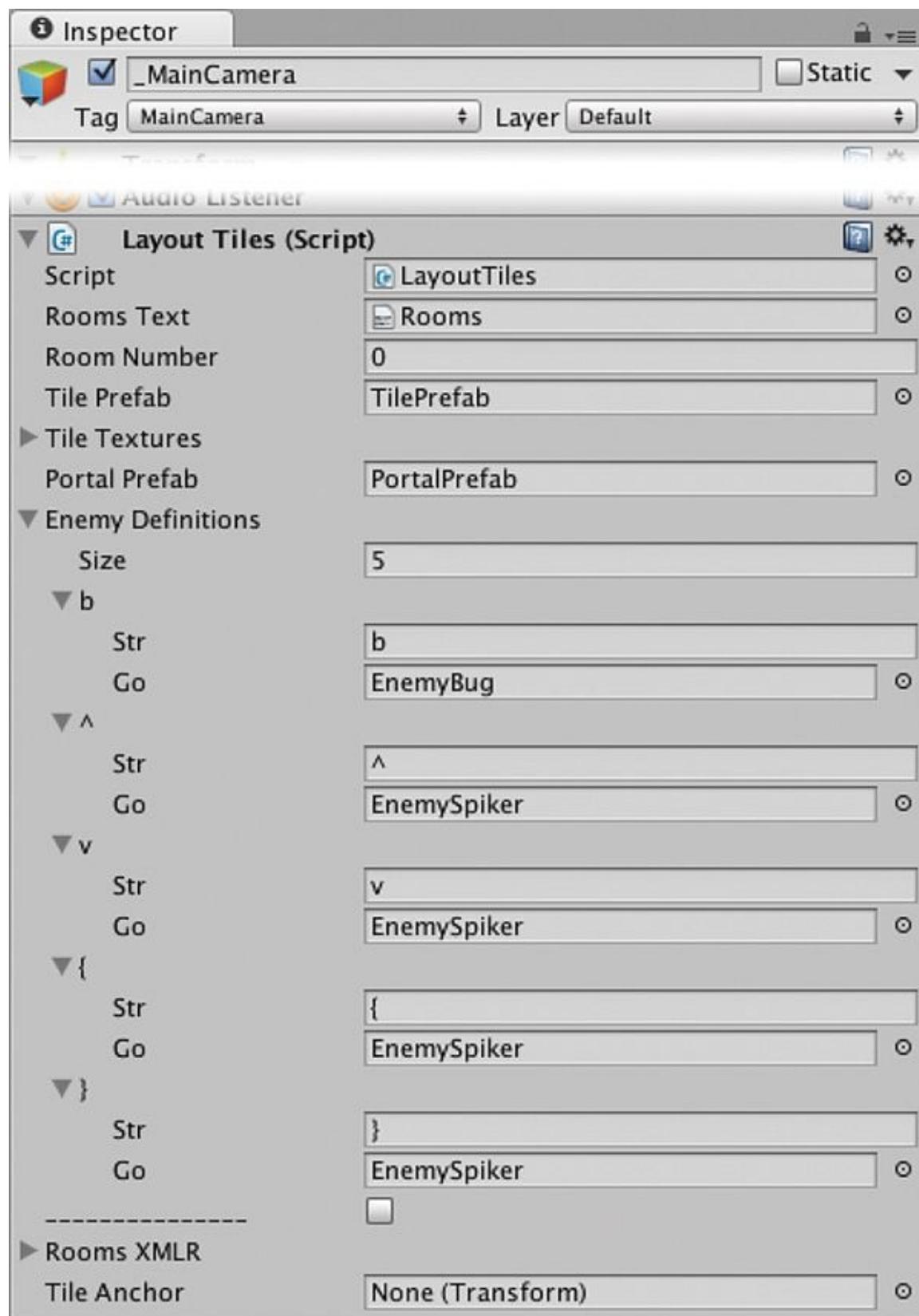


Figure 35.9 Settings for `_MainCamera:LayoutTiles.enemyDefinitions`

Now, as you walk through the dungeon, you will encounter both kinds of enemies. Because of the four different characters for an `EnemySpiker`, the spikers will be initialized facing the proper direction.

Summary

That's it for the final tutorial. This prototype introduced you to a lot of new concepts like interfaces and the Factory Pattern. I'm personally going to continue working on this prototype and see where it goes as a game. I invite you to do the same or, just like any of these, use it as the foundation on which to build your own fantastic game.

Next Steps

If you do continue with this project, here are some additional things you can add to make it a more interesting game:

1. Add direct spells. These would be cast by choosing an element and then tapping on an enemy.
2. Make more enemies. There is a prefab in the `_Prefabs` folder for an `EnemyFlier` that could hover around and then swoop to attack, but there wasn't room in the chapter for the code.
3. Add spells for each element and also a direct spell for when no element is selected. (As discussed earlier in the chapter, this could push the enemy back.)
4. Make the elements a resource. Maybe there are fountains around the dungeon where the Mage can restore her supply of elements. Maybe killing enemies releases elements. In the inventory on the right, the element buttons could show how many of that element the Mage has collected. You'd also want a top limit on how many elements of each type she could carry.
5. Add more interesting ground and wall tiles. If you don't let the player walk off a cliff, you could make a dropoff tile that was lower than the others but still had a collider up where the Mage would collide with it, and it would look a little more interesting but still act just like a wall tile.
6. Use physics layers to more specifically manage your collisions.
7. Following in the footsteps of *Rogue*, make a procedurally generated dungeon so that you had a new, never-before-seen dungeon every time you played.

Thanks!

Thank you again for reading this book. I sincerely hope that it helps you to achieve your dreams.