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 is probably a good time to think again about the purpose of a digital prototype. The first part of the book provided 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 you could spend hours describing game mechanics to someone in detail, having them just play the game and see how it feels is much more efficient (and more interesting). This is discussed at length in the book *Game Feel* by Steve Swink.[1]

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
- **C# script names:** ApplePicker, Apple, AppleTree, and Basket

Do not attach the C# scripts to anything.

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

Preparing

Happily, you've already done a lot of the preparation for this prototype in Chapter 16, "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 basic game mechanics as *Kaboom!*. Take a moment to look back at Chapter 16 and make sure that you understand the flow charts for each element: the AppleTree, the Apple, and the Basket.

As you work through these tutorials, I recommend using a pencil to check off each step as you complete them.

Getting Started: Art Assets

As a prototype, this game doesn't need fantastic art; it needs to work. The kind of art that you'll create throughout this book is 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 working prototype as quickly as possible. If your programmer art doesn't look terrible, that's nice, but it's certainly not necessary.

AppleTree

Let's start with the tree.

1. From the Unity menu bar, choose *GameObject > 3D Object > Cylinder*. This will be the trunk of the 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 Cylinder to match the settings of the Transform component shown in Figure 28.1.

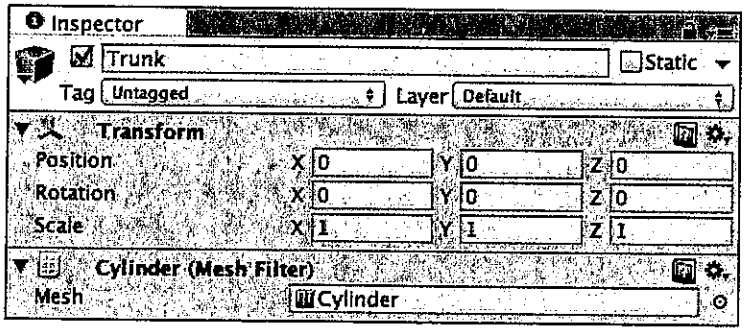

Figure 28.1 The Transform component for the Cylinder named Trunk

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].

2. Now choose *GameObject > 3D Object > Sphere*. Rename Sphere to *Leaves* and set its transform as follows:

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

The Leaves and the Trunk together should look (a bit) like a tree, but they are currently two separate objects. You need to create an empty GameObject to act as their *parent* and encapsulate the two of them under a single object.

3. 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.

4. In the Hierarchy pane, first change the name of GameObject to *AppleTree*. Another way to do this is by clicking the name *GameObject* to highlight it, waiting for a second, and either pressing Return on the keyboard (F2 on Windows) or clicking it a second time.
5. Individually drag the Trunk and Leaves GameObjects onto AppleTree (you will get the same curved arrow icon as you do when you attach a C# script to a GameObject [Figure 19.4]), 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. Your Hierarchy pane and AppleTree should now look like those shown in Figure 28.2.

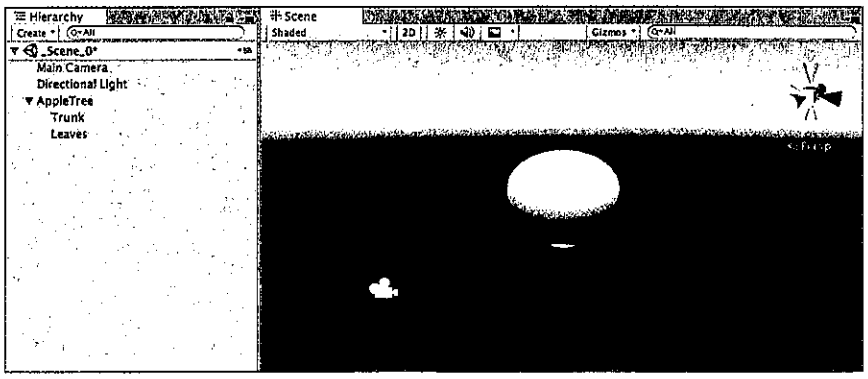

Figure 28.2 AppleTree shown in the Hierarchy and Scene panes with Leaves and Trunk as its children. The asterisk (*) next to **_Scene_0** at the top of the Hierarchy pane indicates that I have unsaved changes in my scene. I should save!

Now that the Trunk and Leaves GameObjects are parented to AppleTree, if you move, scale, or rotate AppleTree, both Trunk and Leaves will move, rotate, and scale alongside it. Give it a try by manipulating the Transform component of AppleTree.

16. 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 center the AppleTree and scale it to twice as large as it was initially.

17. Add a Rigidbody component to AppleTree by selecting it in the Hierarchy and choosing *Component > Physics > Rigidbody* from the Unity menu.

18. In the Rigidbody component Inspector of AppleTree, uncheck *Use Gravity*. If you left it checked, the tree would fall out of the sky when you played the scene.

As covered in Chapter 20, "Variables and Components," the Rigidbody component ensures that the colliders of the Trunk and Sphere are properly updated in the physics simulation when you move the AppleTree across the stage.

Simple Materials for AppleTree

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

1. From the menu bar, choose *Assets > Create > Material*. This makes a new material in the Project pane.
   a. Rename this material to *Mat_Wood*.
   b. Drag the *Mat_Wood* material onto Trunk in your scene or Hierarchy pane.
   c. Select *Mat_Wood* in the Project pane again.
   d. Set the *Albedo* color under *Main Maps* in the Inspector for Mat_Wood to a brown of your liking.[2] You can also adjust the Metallic and Smoothness sliders to your liking.[3]

2. Do the same to create a material named *Mat_Leaves*.
   a. Drag *Mat_Leaves* onto Leaves in either the Hierarchy or Scene pane.
   b. Set the *Albedo* color of Mat_Leaves to a leafy-looking green.

2. When you work with the Unity Standard Shader, Albedo is the main color of the shader. To learn more about the Standard Shader, search for "Standard Shader" in the Unity Manual (http://docs.unity.com). This is a pretty deep topic with several subsections in the table of contents at the left side of the Unity documentation. If you want to learn about Albedo directly, you can search for "Albedo Color and Transparency."
3. I chose Metallic=0 and Smoothness=0.25. As you adjust these, you can see the effect on the Trunk in the Scene pane. You can also look at the material preview sphere at the bottom of the Inspector. If you don't see the preview, click the dark gray Mat_Wood bar at the bottom of the Inspector.

3. 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.
4. By default, Unity scenes come with a *Directional Light* already included. Set the position, rotation, and scale of Directional Light in the Hierarchy to the following:

   Directional Light P:[0, 20, 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, 20, 0] to move it out of the middle of the scene view because its *gizmo* (that is, icon) would be in the middle of the Scene pane otherwise. If you play with the rotation of Directional Light, you will see that the first directional light in the scene is tied to the sun in Unity's default skybox. This isn't used in Apple Picker, but it can be a great effect in 3D games.

5. To move AppleTree up and out of the way a bit, select *AppleTree* in the Hierarchy and change its position to P:[0, 10, 0]. This might move it out of the view of the Scene pane, but you can zoom out to see it by scrolling your mouse wheel.

Apple

Now that you have the AppleTree, you need to make the *Apple* GameObject prefab that it will drop.

1. From the menu bar, choose *GameObject > 3D Object > 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]

2. Create a new material named *Mat_Apple* and set its albedo color to red (or light green, if you prefer green apples).
3. Drag *Mat_Apple* onto Apple in the Hierarchy.

Adding Physics to the Apple

As you might remember from Chapter 17, "Introducing the Unity Development Environment," the *Rigidbody* component enables an object to react to physics (for example, falling or colliding with other objects).

1. Select *Apple* in the Hierarchy pane. From the Unity menu bar, choose *Component > Physics > Rigidbody*.
2. Click the Unity *Play* button, and the Apple will fall off screen due to gravity.
3. Click the *Play* button again to stop playback, and the Apple will return to its start location.

Giving Apples the Tag "Apple"

Eventually you will want to query the scene for an array of all the Apple GameObjects on screen, and giving the Apples a specific tag can help with this.

1. With Apple selected in the Hierarchy, click the pop-up menu button in the Inspector next to *Tag* (that currently displays "Untagged") and choose *Add Tag* from the pop-up menu, as shown in section A of Figure 28.3. This will open Unity's *Tags & Layers Manager*.
2. You might need to click the disclosure triangle next to *Tags* to see the view shown in section B of Figure 28.3. Click the + symbol to add a new tag.
3. Type *Apple* into the *New Tag Name* field (C) and click *Save*. Apple is now in the Tags list (D).

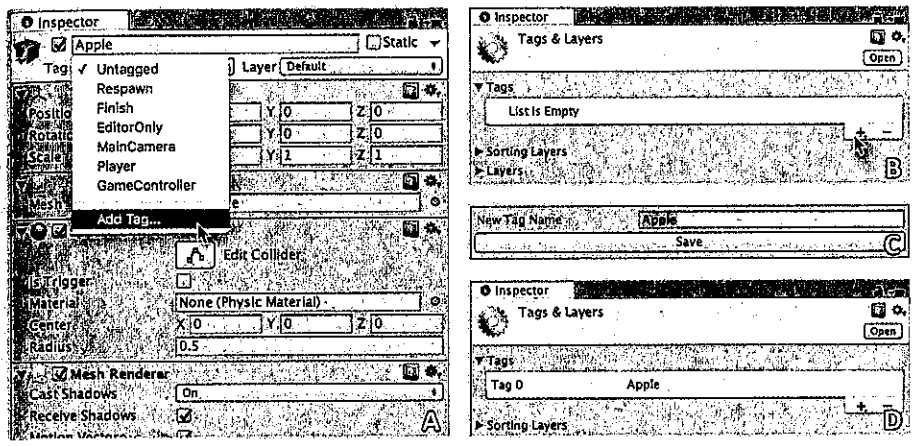

Figure 28.3 Steps 1, 2, and 3 of adding the Apple tag to the list of tags

4. Click *Apple* in the Hierarchy again to return to the Inspector for Apple.
5. Clicking the *Tag* pop-up menu once more now gives you *Apple* as a tag option. Choose *Apple* from the list of tags. Apple GameObjects will now have the tag *Apple*, which makes them easier to identify and select.

Making the Apple into a Prefab

To make the Apple into a prefab, follow these steps:

1. Drag *Apple* from the Hierarchy pane to the Project pane to make it a prefab.[4]
2. After you're sure an Apple prefab is 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 the game will be instantiated from the Apple prefab in the Project pane, you don't need to start with one in the scene.

4. See Chapter 19, "Hello World: Your First Program," for a discussion of prefabs.

Basket

Like the other art assets, the programmer art for the basket is very simple.

1. Choose *GameObject* > *3D Object* > *Cube* from the Unity 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.

2. Create a new material named *Mat_Basket*, color it a light, desaturated yellow (like straw), and apply it to the basket.
3. Add a Rigidbody component to Basket. Select *Basket* in the Hierarchy and choose *Component* > *Physics* > *Rigidbody* from the Unity menu.
   a. Set *Use Gravity* to false (unchecked) in Basket's Rigidbody Inspector.
   b. Set *Is Kinematic* to true (checked) in Basket's Rigidbody Inspector.
4. 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 (just as you did for Apple).
5. Be sure to save your scene.

Your Project and Hierarchy panes should now look like Figure 28.4.

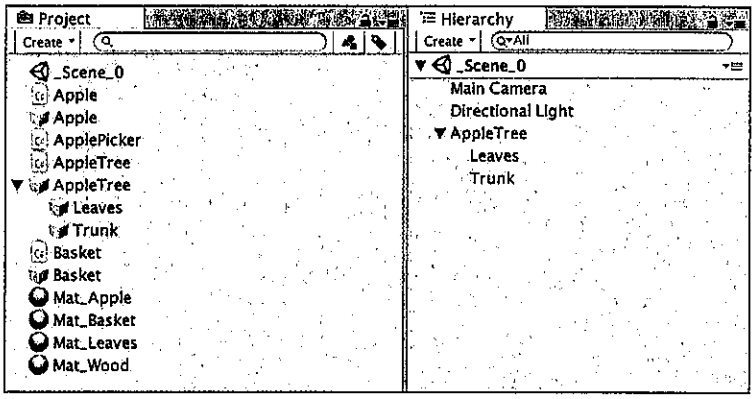

Figure 28.4 The Project and Hierarchy panes at this point in the prototype. You should have created the Apple, ApplePicker, AppleTree, and Basket scripts as part of the project setup at the beginning of this chapter.

Camera Setup

One of the most important things to get right in your games is the position of the camera. For Apple Picker, you want a camera that shows a decent-sized play area. Because the gameplay in this game is entirely two dimensional, you also want an orthographic camera instead of a perspective one.

ORTHOGRAPHIC VERSUS PERSPECTIVE CAMERAS

Orthographic and *perspective* are two types of virtual 3D cameras in games; see Figure 28.5.

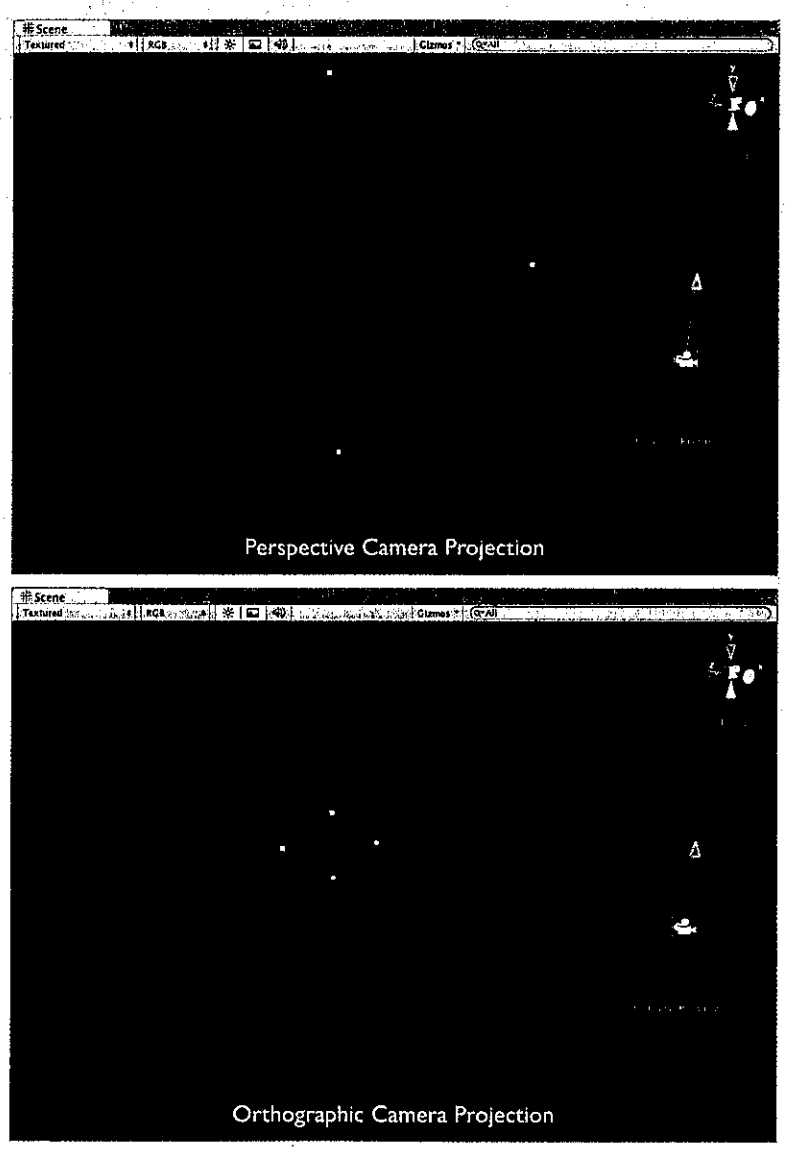

Figure 28.5 Comparison of perspective and orthographic camera projections

A *perspective* camera works like the human eye; because light comes in through a lens, objects that are close to the camera appear large, and objects that are far away appear smaller. This gives a perspective camera a *field of view* (a.k.a. projection) shaped like a square frustum (or more simply, a square pyramid). 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.

Setting the Scene pane to be orthogonal rather than perspective is also sometimes useful. 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). Click the *<Persp* under the axes gizmo to switch between perspective and *isometric* (abbreviated *=Iso*) scene views (isometric being another word for orthographic).

Camera Settings for Apple Picker

Now, establish the camera settings for Apple Picker:

1. Select *Main Camera* in the Hierarchy pane and set its transform as follows:

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

This position moves 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 Unity unit" as 1m in this book.

2. In the Camera component of the Inspector, set the following (as shown in Figure 28.6):
   a. Set the Projection to *Orthographic*.
   b. Set the Size to *16*.

This makes the AppleTree a good size in the Game pane and leaves room for the apples to fall and be caught by the player. Often, you can make a good first guess at things like camera settings and then refine them after you've had a chance to play the game. Just like everything else in game development, finding the right settings for the camera is an iterative process. Your final Main Camera Inspector should now look like what is shown in Figure 28.6.

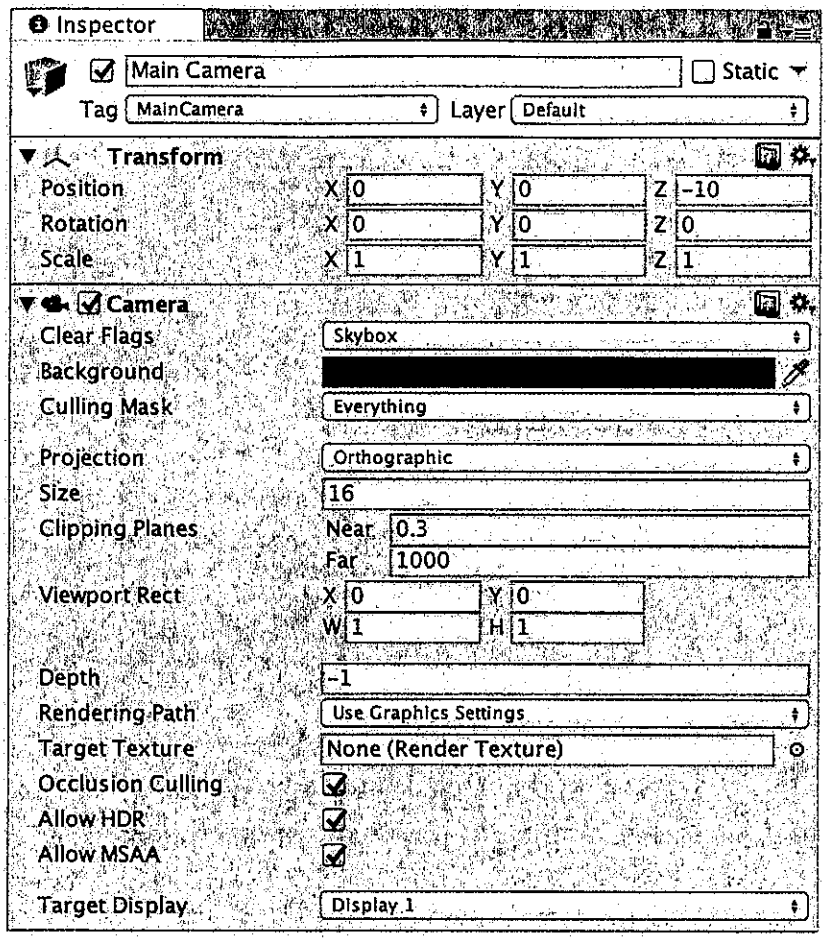

Figure 28.6 Main Camera Inspector settings

Game Panel Settings

Another contributing factor to your game view is the aspect ratio of the Game pane.

1. At the top of the Game pane is a pop-up menu that currently displays *Free Aspect*. This is the aspect ratio pop-up menu.
2. Click the aspect ratio pop-up menu and choose *16:9*. This is the standard format for widescreen televisions and computer monitors, so it will look nice when you play the game full-screen. You also should uncheck the *Low Resolution Aspect Ratios* option if you're on macOS.

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 16, "Thinking in Digital Systems."

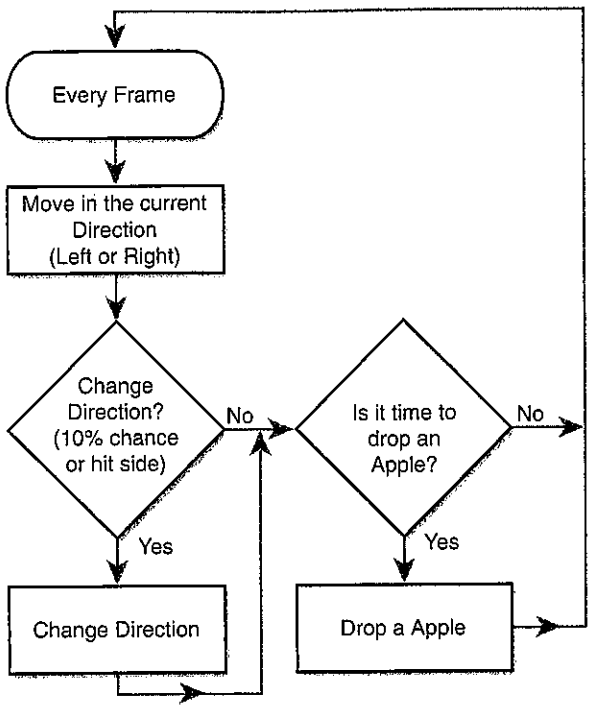

Figure 28.7 AppleTree flow chart

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

- Move at a certain speed every frame.
- Change directions upon hitting the edge of the play area.
- Change directions based on random chance.
- 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.

1. You need some configuration variables, so open the AppleTree class in MonoDevelop and enter code to match the following. The code you need to change is bolded.

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

```
public class AppleTree : MonoBehaviour {
    [Header("Set in Inspector")]
    // Prefab for instantiating apples
    public GameObject   applePrefab;

    // Speed at which the AppleTree moves
    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 might have noticed that the preceding code does not include the line numbers that were shown at the beginnings of lines in some prior chapters. The code listings in this part of the book will generally not have line numbers because there will likely be variance between your line numbers and mine due to differences in carriage returns over the course of many lines of code.[5] Save the AppleTree script in MonoDevelop and return to Unity.

2. To see this code actually do something, you need to attach it to the AppleTree GameObject.
   a. Drag the *AppleTree* C# script from the Project pane onto the AppleTree prefab that is also in the Project pane.
   b. Click the AppleTree instance in the Hierarchy pane; the script has been added not only to the AppleTree prefab but also to all of its instances.
   c. With the AppleTree selected in the Hierarchy, you should see all the variables you just declared appear in the Inspector under the *AppleTree (Script)* component.
3. 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

5. Another reason for omitting line numbers is that I needed every single character I could get to fit some long lines of code on a single line of the printed page.

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 seen well in the Game pane.

a. Set the position of AppleTree to P:[0, *12*, 0]

b. 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 opens your web browser and 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 on the Scripting Reference web page and press Return. Click the top result to see all the methods built in to every MonoBehaviour script (and by extension, built in to every class script you will write and 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 want to learn more about and then choose *Help > Unity API Reference* from the menu bar. This launches an Internet 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.

The first time you visit the Scripting Reference, you might be asked to choose between C# and JS from a couple of rectangular buttons near the top-right of the window. Make sure that you click the *C#* button in the top-right of the page to select C# as your preferred language. Most Unity code examples are available in both C# and JavaScript, though some very old examples might still only exist in JavaScript.

Basic Movement

Now make the following changes to add movement:

1. Make the following bolded changes to the `Update()` method in the AppleTree script. Note the ellipses in the code listing (...). These indicate where I've skipped over lines in this listing to conserve space. Please do not delete those lines!

```
public class AppleTree : MonoBehaviour {
    ...                                                          // a
    void Update () {
        // Basic Movement
        Vector3 pos = transform.position;                        // b
        pos.x += speed * Time.deltaTime;                         // c
        transform.position = pos;                                // d

        // Changing Direction
    }
```

The // indicators at the right side of lines reference the following additional info.

a. 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 any lines of code between the AppleTree class declaration and the `Update()` method, so I have used ellipses to skip those unchanged lines.

b. This line defines the Vector3 `pos` to be the current position of the AppleTree.

c. 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 "Making Your Games Time Based" sidebar).

d. Assigns this modified `pos` back to `transform.position` (which moves AppleTree to a new position). If you don't set `transform.position` to `pos`, AppleTree will not move.

You might be wondering why the preceding code changes were three lines instead of just one. Why couldn't the code just be this?

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

The answer is that `transform.position` is a *property*, a method that is masquerading as a field (i.e., a function masquerading as a variable) through the use of `get{}` and `set{}` accessors (see Chapter 26, "Classes"). Although reading the value of a property's subcomponent is possible, setting a subcomponent of a property is not. 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`.

2. Save the script, return to Unity, and 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 `speed` to 10, which makes it move at 10m/s (10 meters per second or 10 Unity units per second). Stop Unity playback and set `speed` to 10 in the Inspector.

MAKING YOUR GAMES TIME BASED

When movement in a game is *time based*, it happens at the same rate 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 of a second to display. If the `// b` line of code were run at 25 fps, the result would resolve like this:

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

So, in 1/25 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.

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

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

So, in 1/100 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 this consistency enables 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 vary broadly.

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:

```
public class AppleTree : MonoBehaviour {
    …
    void Update () {
        // Basic Movement
        …
        // Changing Direction
```

```
    if ( pos.x < -leftAndRightEdge ) {                              // a
        speed = Mathf.Abs(speed);  // Move right                    // b
    } else if ( pos.x > leftAndRightEdge ) {                        // c
        speed = -Mathf.Abs(speed); // Move left                     // c
    }
}
```

a. Test whether the new `pos.x` that was just set in the previous lines is less than the negative side-to-side limit that is set by `leftAndRightEdge`.

b. 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.

c. 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.

Save the script, return to Unity, and click *Play* to see what happens.

Changing Direction Randomly

To introduce random changes in direction, follow these steps:

1. Add the bolded lines shown here:

```
public class AppleTree : MonoBehaviour {
    …
    void Update () {
        // Basic Movement
        …
        // 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 ) {     // a
            speed *= -1;  // Change direction                       // b
        }
    }
}
```

   a. `Random.value` returns a random float value between 0 and 1 (including 0 and 1 as possible values). If this random number is less than `chanceToChangeDirections`, …

   b. …the AppleTree will change directions by setting `speed` to the negative of itself.

2. If you click Play, you'll see that the default `chanceToChangeDirections` of 0.1 changes direction far too often. In the Inspector, change the value of `chanceToChangeDirections` to 0.02, and it should feel a lot better.

To continue the discussion of time based games from the "Making Your Games Time Based" sidebar, this chance to change directions is actually *not* time based. Every frame, a 2% chance exists that the AppleTree will change directions. On a very fast computer, that chance could happen 400 times per second (yielding an average of 8 directions 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).

3. 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).

```
public class AppleTree : MonoBehaviour {
    …
    void Update () {
        // Basic Movement
        …
        // Changing Direction
        if ( pos.x < -leftAndRightEdge ) {
            speed = Mathf.Abs(speed);  // Move right
        } else if ( pos.x > leftAndRightEdge ) {
            speed = -Mathf.Abs(speed); // Move left
        }                                                                    // a
    }

    void FixedUpdate() {
        // Changing Direction Randomly is now time-based because of FixedUpdate()
        if ( Random.value < chanceToChangeDirections ) {                     // b
            speed *= -1;  // Change direction
        }
    }
}
```

   a. Cut the two lines that were marked `// a` and `// b` in the code listing for step 1, replace them with the closing brace, …

   b. …and paste them here.

This causes the AppleTree to randomly change directions an average of 1 time every second (50 `FixedUpdates` per second * a random chance of 0.02 = an average of 1 time per second).

Dropping Apples

Next comes dropping apples:

1. 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 in either of two ways:

- Click the tiny target to the right of *Apple Prefab None (Game Object)* in the Inspector and select *Apple* from the Assets tab in the window that appears.

  or

- Drag the *Apple* GameObject prefab from the Project pane onto the *ApplePrefab* value in the Inspector pane. This process is shown graphically in Figure 19.4 of Chapter 19, "Hello World: Your First Program."

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

```
ublic class AppleTree : MonoBehaviour {
  …
  void Start () {
      // Dropping apples every second
      Invoke( "DropApple", 2f );                                    // a
  }

  void DropApple() {                                                // b
      GameObject apple = Instantiate<GameObject>( applePrefab );    // c
      apple.transform.position = transform.position;                // d
      Invoke( "DropApple", secondsBetweenAppleDrops );              // e
  }

  void Update () { … }                                              // f
  …
```

a. The `Invoke()` function calls a named function in a certain number of seconds. In this case, it is calling the new function `DropApple()`. The second parameter, `2f`, tells `Invoke()` to wait 2 seconds before it calls `DropApple()`.

b. `DropApple()` is a custom function to instantiate an Apple at the AppleTree's location.

c. `DropApple()` creates an instance of `applePrefab` and assigns it to the GameObject variable `apple`.

d. The position of this new `apple` GameObject is set to the position of the AppleTree.

e. `Invoke()` is called again. This time, it will call the `DropApple()` function in `secondsBetweenAppleDrops` seconds (in this case, in 1 second based on the default settings in the Inspector). Because `DropApple()` invokes itself every time it is called, the effect will be for an Apple to be dropped every second that the game runs.

f. The { … } on this line indicates that I've omitted the content of the `Update()` method in this code listing. You do not need to change anything about the `Update()` method when you see ellipses like this.

3. Save the AppleTree script, return to Unity, click *Play*, and see what happens.

Did you expect the Apples and the AppleTree to go flying off? The same thing occurs here as did in the Chapter 19, "Hello World" example with the cubes flying all over the place. I did this on purpose here to show you how to fix this issue if you encounter it in your games. The first thing to do is to set the AppleTree's Rigidbody to be *kinematic*, meaning that we can move it via code, but it will not react to collisions with other objects.

4. In the Rigidbody component Inspector for AppleTree, check *Is Kinematic*.
5. Click *Play* again, and you can see that you still have an issue with the Apples.

Though this fixes the AppleTree, the Apples are still colliding with the AppleTree, causing them to fly off to the left, right or down faster than they would normally due to gravity. To fix this, you need to put them in a *physics layer* that doesn't collide with the AppleTree. Physics 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 physics layers are set to ignore each other, then the AppleTree and Apples will cease colliding with each other.

Setting Physics Layers

First, you need to make some new physics layers. These steps are shown in Figure 28.8.

1. Click the *AppleTree* in the Hierarchy and in the Inspector choose *Add Layer...* from the pop-up menu next to *Layer*. This opens the *Tags & Layers Manager* in the Inspector, which allows you to set the names of physics 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 Layers 8 through 31.
2. Name layer 8 *AppleTree*, layer 9 *Apple*, and layer 10 *Basket*.

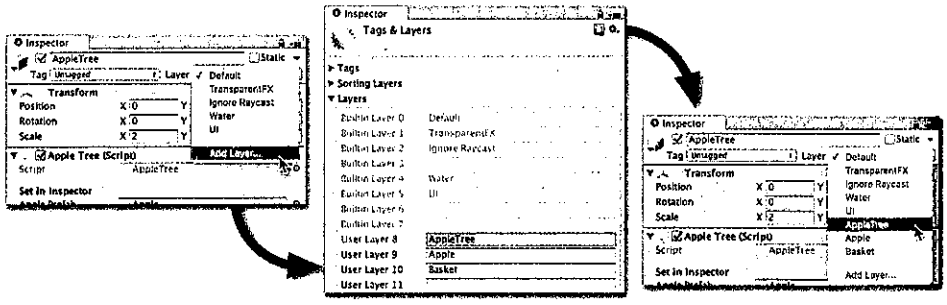

Figure 28.8 The steps required to make new physics layers (steps 1 and 2) and assign them (step 5)

3. From the Unity menu bar, choose *Edit* > *Project Settings* > *Physics*. This sets the Inspector to the *Physics Manager* (see Figure 28.9). The *Layer Collision Matrix* grid of check boxes at the bottom of the Physics Manager sets which physics layers will collide with each other (and whether GameObjects in the same Physics Layer will collide with each other as well).

4. You want the Apple to collide with the Basket and to not collide with either the AppleTree or other Apples. To do this, your Layer Collision Matrix grid should look like what is shown in Figure 28.9.

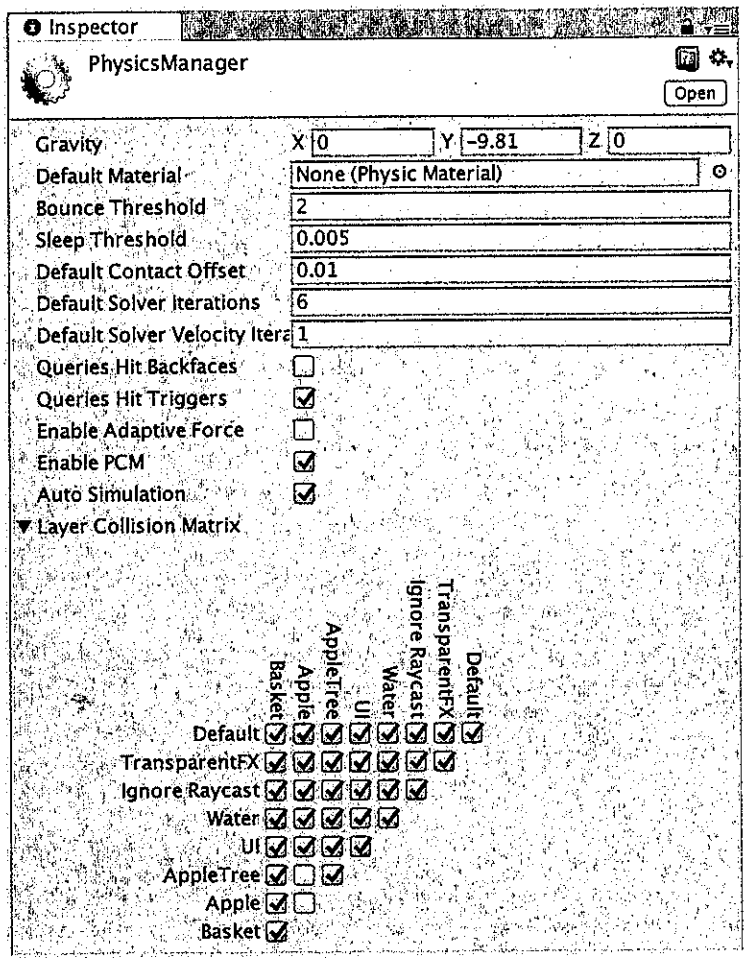

Figure 28.9 The required Layer Collision Matrix settings in the Physics Manager

5. Now that the Layer Collision Matrix is set properly, it's time to assign physics layers to the important GameObjects in the game.
   a. Click *Apple* in the Project pane, and then select *Apple* from the *Layer* pop-up menu at the top of the Inspector pane.
   b. Select the *Basket* in the Project pane and set its Layer to *Basket*.
   c. Select the *AppleTree* in the Project pane and set its Layer to *AppleTree* (refer to Figure 28.8).

When you choose the physics layer for AppleTree, Unity asks whether you want to change the layer for just AppleTree or for both AppleTree and its children. You definitely want to choose *Yes, change children* because you need the Trunk and Sphere child objects of AppleTree 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 click 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 the apples.

1. Open the Apple C# script and add the following code to kill the apples when they reach a depth of `transform.position.y == -20` (which is comfortably off screen). Here's the code:

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

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

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

   a. The bolded `public static float` line declares and defines a *static variable* named `bottomY`. As was mentioned in Chapter 26, "Classes," 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.

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

2. Save the Apple script.
3. You must attach the Apple C# script to the Apple GameObject prefab in the Project window for this code to function in the game. You already know about dragging a script onto a GameObject to attach the script, so here's another way to do this:
   a. Select *Apple* in the Project pane.
   b. Scroll to the bottom of the Inspector, and click the *Add Component* button.
   c. From the pop-up menu that appears, select *Scripts* > *Apple*.

Now, if you click Play in Unity and zoom out in the scene, you can see that apples drop for a ways and then disappear when they reach a Y position of -20.

This is all you need to do for the apples.

Instantiating the Baskets

To make the Baskets work, I am going to introduce a concept that will recur throughout these prototype tutorials. Although object-oriented thinking encourages designers 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.

1. Attach the *ApplePicker* script to *Main Camera* in the Hierarchy. I often attach these game management scripts to Main Camera because I am guaranteed that there is a Main Camera in every scene.
2. Open the ApplePicker script in MonoDevelop, type the following code, and then save:

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

public class ApplePicker : MonoBehaviour {
    [Header("Set in Inspector")]                                    // a
    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<GameObject>( basketPrefab );
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
        }
    }
```

a. This line adds a header to the Inspector in Unity so that you can see which variables should be set in the Inspector. In later code listings, it will be accompanied by a "Set Dynamically" header for variables that are calculated while the game is running.

This code instantiates three copies of the Basket prefab that are spaced out vertically.

3. In Unity, click *Main Camera* in the Hierarchy pane and set the `basketPrefab` in the Inspector to be the Basket GameObject prefab from the Project pane. Click Play, and you'll see that this code creates three baskets at the bottom of the screen.

Moving the Baskets with the Mouse

Next, you need to write some code to get each Basket moving along with the mouse.

1. Attach the *Basket* script to the *Basket* prefab in the Project pane.
2. Open the Basket C# script in MonoDevelop, enter this code, and save:

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

public class Basket : MonoBehaviour {

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

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

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

        // 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;
    }
}
```

a. `Input.mousePosition` is 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.mousePositon` is always 0 because it is essentially a two-dimensional measurement.

b. This line sets the z coordinate of `mousePos2D` to the negative of the Main Camera's Z position. In the 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, placing the final world point on the Z=0 plane.

c. `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 pushed into the 3D space 10 meters away from the Main Camera position, resulting in a `mousePos3D.z` of 0. This matters little in Apple Picker, but it will become much more important in future games. If this is at all confusing, I recommend looking at `Camera.ScreenToWorldPoint()` in the Unity Scripting Reference.[6]

Now the Baskets will move when you press *Play* in Unity, and you can use them to collide with apples, though the Apples aren't really being caught yet.

Catching Apples

Next up—catching apples:

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

```
public class Basket : MonoBehaviour {

    void Update () { … }

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

   a. 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.

   b. This line assigns this colliding GameObject to the local variable `collidedWith`.

   c. 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.

2. Save the Basket script, return to Unity, and click *Play*.

6. The reference is located at https://docs.unity3d.com/ScriptReference/. Be sure to click the C# button there so that you're looking C# documentation (as opposed to JavaScript).

At this point, the game functions very similarly to the classic game *Kaboom!* However, 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.

3. Save your scene.
4. Make a duplicate of the current scene to use for testing game balance tweaks.
   a. Click the *_Scene_0* in the Project pane to select it.
   b. Press *Command-D* on the keyboard (*Ctrl+D* on Windows) to duplicate the scene or choose *Edit > Duplicate* from the menu bar. This creates a new scene named *_Scene_1*.
   c. Double-click *_Scene_1* to open it.

As an exact duplicate of _Scene_0, the game in this new scene will work without any changes.

Click *AppleTree* in the Hierarchy to tweak variables in _Scene_1 while leaving the variables in _Scene_0 unchanged (any changes made to GameObjects in the Project pane apply to both scenes). Try making the game more difficult. 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 opened, just look at the title at the top of the Unity window or the top of the Hierarchy pane. Each 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 levels and lives.

Score Counter

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

1. Open _Scene_0 by double-clicking it in the Project pane.
2. From the Unity menu bar choose *GameObject > UI > Text*.[7]

Because this is the first *uGUI* (Unity Graphical User Interface) element to be added to this scene, it will add several things to the Hierarchy pane. The first you'll see is a *Canvas*. The Canvas is

7. Once, when I was testing this section of the tutorial, the *GameObject > UI > Text* option of the Unity menu was grayed out for some reason. If that happens for you, right-click in the empty part of the Hierarchy pane and choose *UI > Text* from the right-click menu.

the two-dimensional board on which the GUI will be arranged. Looking in the Scene pane, you should also see a *very* large 2D box extending from the origin out very far in the x and y directions.

3. Double-click on *Canvas* in the Hierarchy to zoom out and see the whole thing. This will be scaled to match your Game pane, so if you have the Game pane set to a 16:9 aspect ratio, the Canvas will follow suit. You might also want to click the *2D* button atop the Scene pane to switch to a two-dimensional view that can make working with the Canvas easier.

The other GameObject added at the top level of the Hierarchy is the *EventSystem*. The EventSystem is what allows buttons, sliders, and other interactive GUI elements that you build in uGUI to work; however, you will not be making use of it in this prototype.

As a child of the Canvas, you will see a *Text* GameObject. If you don't see it there, click the disclosure triangle in the Hierarchy next to Canvas to show its child objects. Double-click on the *Text* GameObject in the Hierarchy pane to zoom in on it. It is very likely that the text color defaulted to black, which might be difficult to see over the background of the Scene pane.

4. Select the *Text* GameObject in the Hierarchy and use the Inspector pane to change its name to *HighScore*.
5. Follow these directions to make the HighScore Inspector match that shown in Figure 28.10:
   a. In the *RectTransform* component of the HighScore Inspector:
      - Set *Anchors* Min X=0, Min Y=1, Max X=0, and Max Y=1.
      - Set *Pivot* X=0 and Y=1.
      - Set Pos X=10, Pos Y = –6, and Pos Z = 0.
      - Set Width=256 and Height=32.

      After doing this, you should double-click HighScore in the Hierarchy again to re-center the view of it in the Scene pane.
   b. In the *Text (Script)* component of the HighScore Inspector:
      - Set the *Text* section to "High Score: 1000" (without the quotes around it).
      - Set the *Font Style* to Bold.
      - Set the *Font Size* to 28.
      - Set the *Color* to white, which will make it much more visible in the Game pane.
6. Right-click on HighScore in the Hierarchy and choose *Duplicate*.[8]
7. Select the new *HighScore (1)* GameObject and change its name to *ScoreCounter*.
8. Alter the RectTransform and Text values of ScoreCounter in the Inspector to match those shown in Figure 28.10. Don't forget to set the *Anchors* and *Pivot* in the RectTransform component and the *Alignment* in the Text component. Notice that when you change the

8. You might need to click once in the Hierarchy pane to see the *HighScore (1)* duplicate appear.

Anchors or Pivot in the RectTransform, Unity automatically changes the *Pos X* to keep the ScoreCounter in the same place within the Canvas. To prevent Unity from doing this, click the *R* button in the RectTransform that is shown under the mouse cursor in the ScoreCounter Inspector in Figure 28.10.

As you've seen here, the coordinates for uGUI GameObjects differ completely from those for regular GameObjects and use a RectTransform instead of a regular Transform. The coordinates for a RectTransform are all relative to the Canvas parent of the uGUI GameObject. Clicking the help icon for the RectTransform component (circled in Figure 28.10) can give you more information about how this works. Be sure to save your scene before moving on.

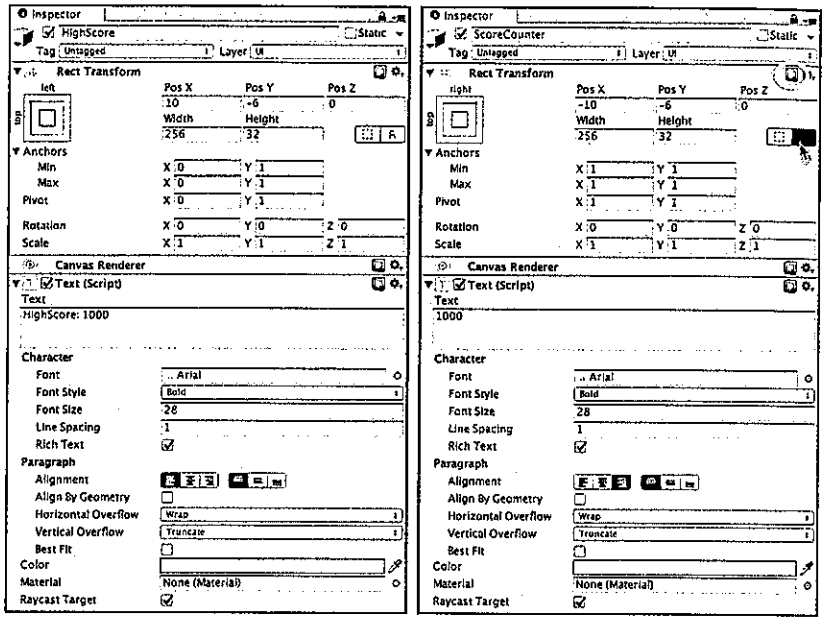

Figure 28.10 RectTransform and Text component settings for HighScore and ScoreCounter

Add Points for Each Caught Apple

When a collision occurs between an apple and a basket, two scripts are notified: the Apple and Basket scripts. In this game, there is already an `OnCollisionEnter()` method on the Basket C# script, so in the following steps you modify the Basket script to give the player points for each Apple that is caught. One hundred 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).

1. Open the Basket script in MonoDevelop and add the bolded lines shown here:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;      // This line enables use of uGUI features.          // a

public class Basket : MonoBehaviour {
    [Header("Set Dynamically")]
    public Text           scoreGT;                                              // a

    void Start() {
        // Find a reference to the ScoreCounter GameObject
        GameObject scoreGO = GameObject.Find("ScoreCounter");                   // b
        // Get the Text Component of that GameObject
        scoreGT = scoreGO.GetComponent<Text>();                                 // c
        // Set the starting number of points to 0
        scoreGT.text = "0";
    }

    void Update () { … }

    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 );                              // d
            // Add points for catching the apple
            score += 100;
            // Convert the score back to a string and display it
            scoreGT.text = score.ToString();
        }
    }
}
```

a. Be sure you don't neglect to enter these lines. They are separated from the others.

b. `GameObject.Find("ScoreCounter")` searches through all the GameObjects in the scene for one named `"ScoreCounter"` and assigns it to the local variable `scoreGO`. Make sure "ScoreCounter" does not contain a space in code or in the Hierarchy.

c. `scoreGO.Getcomponent<Text>()` searches for a Text 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. Without the earlier `using UnityEngine.UI;` line, the Text component would not be defined for C# within Unity. As Unity Technology's coding practices get stronger, they are moving to more of a model like this where you must include the code libraries for their new features manually.

d. `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 it can end the round and destroy the rest of the apples. This involves one script calling a function on another.

1. Start by making these modifications to the Apple C# script in MonoDevelop:

```
public class Apple : MonoBehaviour {
    [Header("Set in Inspector")]
    public static float     bottomY = -20f;

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

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

a. Note that all of these added lines are within this `if` statement.

b. This 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, using `GameObject.Find("Main Camera")` to obtain a reference to Main Camera is not necessary. `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, accessing public variables and methods of the ApplePicker class instance attached to Main Camera becomes possible.

c. This calls a non-existent `AppleDestroyed()` method of the ApplePicker class. Because it doesn't exist yet, MonoDevelop will color it red, and you will not be able to play the game in Unity until `AppleDestroyed()` is defined.

2. A public `AppleDestroyed()` method does not yet exist in the ApplePicker script, so open the ApplePicker C# script in MonoDevelop and make the following bolded changes:

```
public class ApplePicker : MonoBehaviour {
    ...
```

```
void Start () { … }

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

a. 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.

b. `GameObject.FindGameObjectsWithTag("Apple")` will return an array of all existing Apple GameObjects.[9] The subsequent `foreach` loop iterates through each of these and destroys them.

Save All scripts in MonoDevelop. With `AppleDestroyed()` now defined, the game is once again playable in Unity.

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 (this time, the entire code is listed, just in case):

```
using System.Collections;
using System.Collections.Generic;                                       // a
using UnityEngine;
using UnityEngine.SceneManagement;                                      // b

public class ApplePicker : MonoBehaviour {
    [Header("Set in Inspector")]
    public GameObject       basketPrefab;
    public int              numBaskets = 3;
    public float            basketBottomY = -14f;
    public float            basketSpacingY = 2f;
    public List<GameObject> basketList;

    void Start () {
        basketList = new List<GameObject>();                            // c
        for (int i=0; i<numBaskets; i++) {
            GameObject tBasketGO = Instantiate<GameObject>( basketPrefab );
            Vector3 pos = Vector3.zero;
```

9. `GameObject.FindGameObjectsWithTag()` is actually a rather processor-intensive function, so I wouldn't recommend using it inside an `Update()` or `FixedUpdate()`. However, because this is only happening when the player loses a Basket (and the gameplay is already slowed here), in this case, it's fine to use.

```
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
            basketList.Add( tBasketGO );                                    // d
        }
    }

    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                                       // e
        // 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 );
    }
}
```

a. You will be storing the Basket GameObjects in a List, so it is necessary to use the `System.Collections.Generic` code library, which as of Unity 5.5 is included in all new scripts. (For more information about lists, see Chapter 23, "Collections in C#.") 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()`.

b. This will be used in step 3 of *Adding a High Score* later in the chapter.

c. This line defines `basketList` as a new List<GameObject>. Though this was already declared by the line at `// b`, the value of `basketList` after declaration is `null`. The initialization on this line makes it an actual List that can be used.

d. 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.

e. 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).

If you play the game now and run out of baskets, Unity will throw an IndexOutOfRange exception.

Adding a High Score

Now you'll make use of the *HighScore* Text GameObject that you created earlier:

1. Create a new C# script named *HighScore*, and attach it to the HighScore GameObject in the Hierarchy pane.
2. Open the HighScore script in MonoDevelop and give it the following code:

```
ing System.Collections;
ing System.Collections.Generic;
ing UnityEngine;
ing UnityEngine.UI;    // Remember, we need this line for uGUI to work.

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

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

   a. Making the int `score` not only `public` but also `static` gives you the ability to access it from any other script by simply typing `HighScore.score`. This is one of the powers of static variables that you will use throughout the prototypes in this book.

   b. The lines in `Update()` simply display the value of `score` in the Text component. Calling `ToString()` on the `score` is not necessary in this instance because when the + operator is used to concatenate a string with another data type (the `"High Score: "` string literal is concatenated with the int `score` in this case), `ToString()` is called implicitly (that is, automatically).

3. Open the Basket C# script and add the following bolded lines to see how this is used:

```
public class Basket : MonoBehaviour {
    …
    void OnCollisionEnter( Collision coll ) {
        …
        if ( collidedWith.tag == "Apple" ) {
            …
            // 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` will be set any time the current score exceeds it.

4. Open the ApplePicker C# script and add the following lines to reset the game when the player runs out of baskets. This code avoids the IndexOutOfRange exception mentio

```
public class ApplePicker : MonoBehaviour {
    …
    public void AppleDestroyed() {
        …
        // Remove the Basket from the list and destroy the GameObje
        basketList.RemoveAt( basketIndex );
        Destroy( tBasketGO );

        // If there are no Baskets left, restart the game
        if ( basketList.Count == 0 ) {
            SceneManager.LoadScene( "_Scene_0" );
        }
    }
}
```

   a. `SceneManager.LoadScene( "_Scene_0" )` will reload _Scene_0 unless you added the line `using UnityEngine.SceneManagement` and *Destroying a Basket When an Apple Is Dropped* heading earlier. Reloading effectively resets the game to its beginning state.[10]

5. You've changed a number of scripts now. Did you remember to save after changing one? If not—or if you're not sure, as I often am—you can choose *File > Save All* MonoDevelop menu bar to save all modified but unsaved scripts. If *Save All* is then congratulations—all your scripts are already saved.

Preserving the High Score in PlayerPrefs

Because `HighScore.score` is a static variable, it is *not* reset along with the rest of the means that high scores will remain from one round to the next. However, whenever game (by clicking the Play button again), `HighScore.score` *will* reset. You can fix this the use of Unity's *PlayerPrefs*. PlayerPrefs variables store information from Unity scripts computer so that the information can be recalled later and isn't destroyed when playing PlayerPrefs also work across the Unity editor, compiled builds, and WebGL builds, so the score you get in one will carry over to the others, as long as they're run on the same ma

10. In all recent versions of Unity up to at least Unity 2017, a known bug often occurs when a level. If you see the scene get much darker when it reloads using the SceneManager encountering this issue. Until Unity fixes the bug, one interim fix for this is to disable auto baking (which pre-computes some of the lighting for the scene). To do so, open the Light by selecting *Window > Lighting > Settings* from the Unity menu bar. Uncheck *Auto Generate* bottom of the Lighting pane (next to the *Generate Lighting* button). Then click the Gen button once to manually rebuild lighting, wait for it to complete, and close the Lighting This bug should only occur inside the Unity editor and shouldn't affect any WebGL or standalone application builds that you make.

Open the HighScore C# script and add the following bolded changes:

```
ng System.Collections;
ng System.Collections.Generic;
ng UnityEngine;
ng UnityEngine.UI;    // Remember, we need this line for uGUI to work.

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

void Awake() {                                                    // a
    // If the PlayerPrefs HighScore already exists, read it
    if (PlayerPrefs.HasKey("HighScore")) {                        // b
        score = PlayerPrefs.GetInt("HighScore");
    }
    // Assign the high score to HighScore
    PlayerPrefs.SetInt("HighScore", score);                       // c
}

void Update () {
    Text gt = this.GetComponent<Text>();
    gt.text = "High Score: "+score;
    // Update the PlayerPrefs HighScore if necessary
    if (score > PlayerPrefs.GetInt("HighScore")) {                // d
        PlayerPrefs.SetInt("HighScore", score);
    }
}
```

a. `Awake()` is a built-in Unity MonoBehaviour method (like `Start()` or `Update()`) that happens when this instance of the HighScore class is first created (so `Awake()` always occurs before `Start()`).

b. PlayerPrefs is a dictionary of values that are referenced through keys (that is, unique strings). In this case, you're referencing the key *HighScore*. This line checks to see whether a HighScore int already exists in PlayerPrefs and reads it in if it does exist.

   PlayerPrefs are stored separately for each project/application, so naming this HighScore is okay; it won't conflict with a HighScore stored in PlayerPrefs by a different project.

c. The last line of `Awake()` assigns the current value of `score` to the HighScore PlayerPrefs key. If a HighScore int already exists, this will rewrite the value back to PlayerPrefs; if the key does not already exist, however, this ensures that an HighScore key is created.

d. 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.

2. *Save All* in MonoDevelop again, just to be sure. Switch back to Unity and click Play.

Now, you should be able to play the game complete with score and high score. If you attain a new high score, try stopping the game and restarting it, and you'll see that your high score is saved.

Summary

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

Next Steps

Here are some ideas for additional elements that you could add to the prototype in the future. One of the best ways to learn coding is to follow a tutorial like this chapter and then try to add your own modifications to it.

- **Start screen:** You could build a start screen in its own scene and give it a splash image and a Start button. The Start button could then call `SceneManager.LoadScene( "_Scene_0" );` to start the game. Remember that you need to enable SceneManager by adding the line `using UnityEngine.SceneManagement;` to the top of your script.
- **Game Over screen:** You could also create 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 `SceneManager.LoadScene( "_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.

If you choose to add either a Start screen or Game Over scene to the game, you will need to add every scene in your game to the Build Settings scene list one at a a time. To do so, open each scene in Unity and then choose *File > Build Settings...* from the Unity menu. In the Build Settings window that opens, click *Add Open Scenes*, and the name of the currently open scene will be added to the *Scenes in Build* list. If you do create a build of this game, the scene numbered zero will be the one that loads when the game first runs.

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 other physics games that came before it, such as *Crossbows and Catapults, Worms, Scorched Earth,* and so on.

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