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.

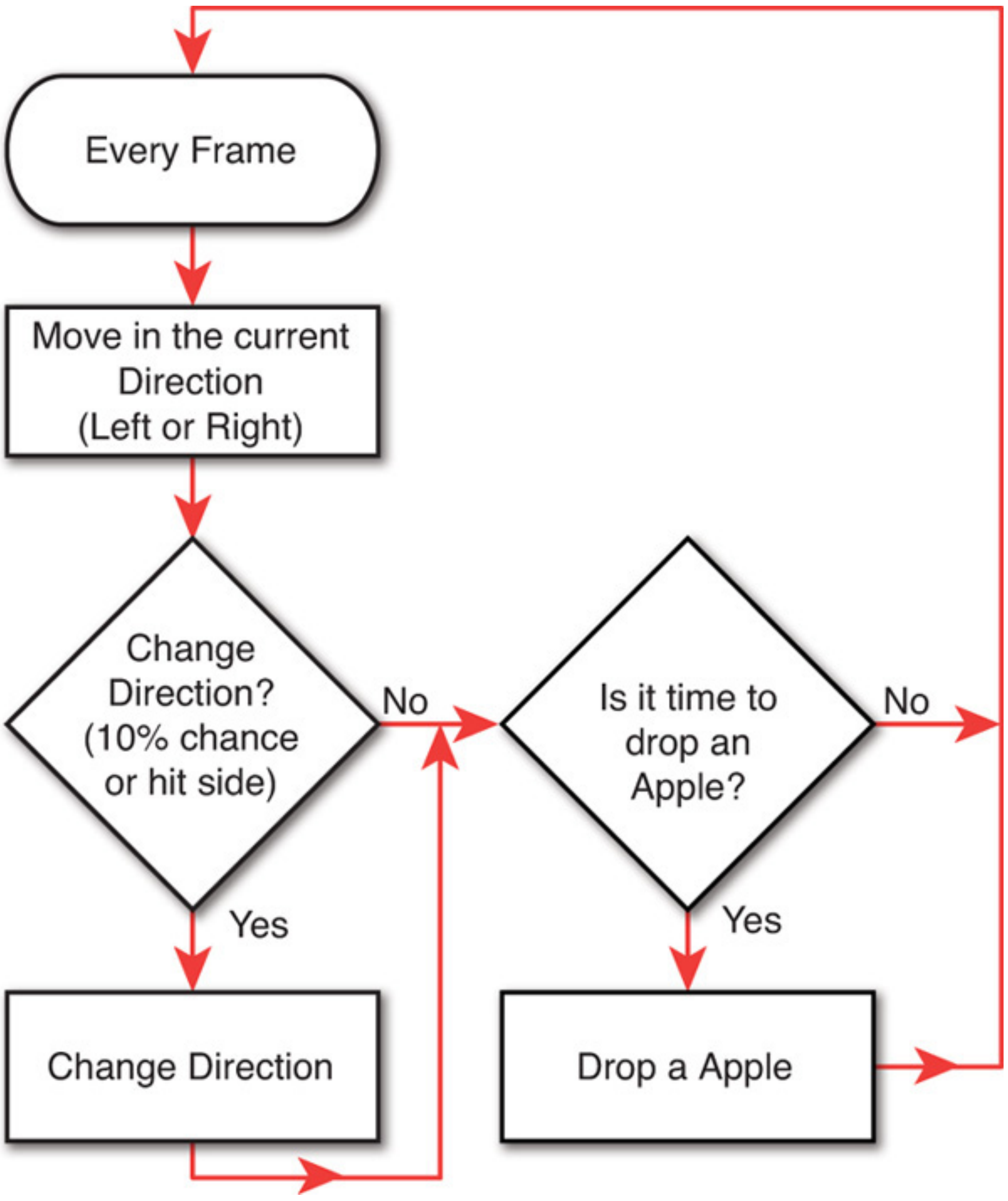

Every Frame
Move in the current Direction (Left or Right)
Change Direction? (10% chance or hit side)
No
Yes
Change Direction
Is it time to drop an Apple?
No
Yes
Drop a Apple

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 4/100ths 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 1/25th 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 (inclusive, which means that the lowest value `Random.Value` can return is 0, and the highest it can return is 1). 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 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). 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 `FixedUpdates` 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.

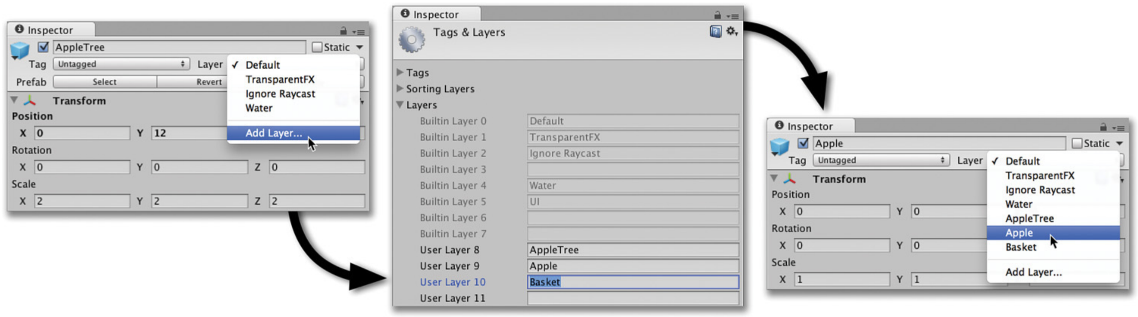
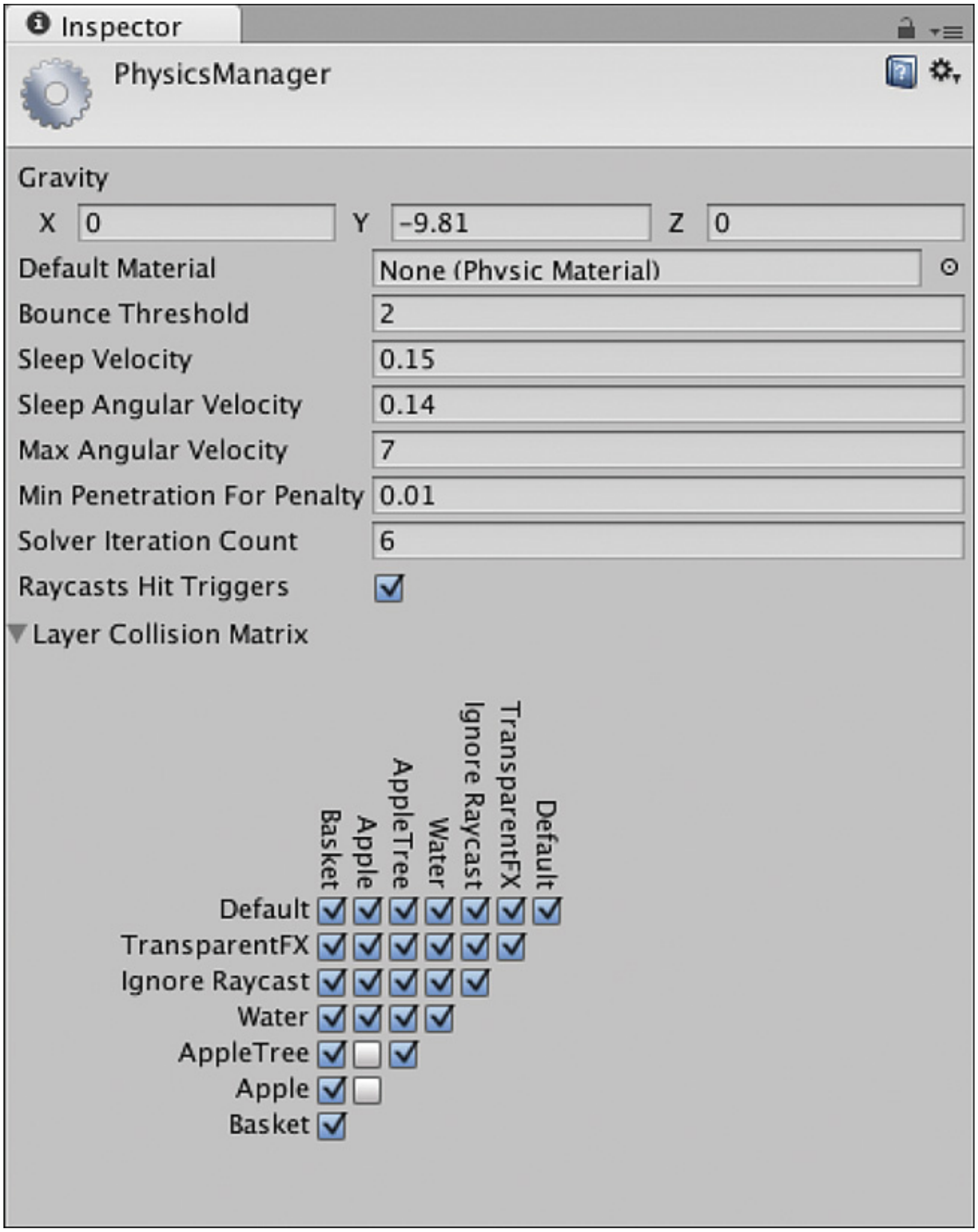


Figure 28.8 The steps required to make new physics layers and assign them

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.



PhysicsManager

Gravity

X 0 Y -9.81 Z 0

Default Material	None (Physic Material)
Bounce Threshold	2
Sleep Velocity	0.15
Sleep Angular Velocity	0.14
Max Angular Velocity	7
Min Penetration For Penalty	0.01
Solver Iteration Count	6
Raycasts Hit Triggers	✓

Layer Collision Matrix

	Basket	Apple	AppleTree	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓	✓		
Water	✓	✓	✓	✓			
AppleTree	✓	☐	✓				
Apple	✓	☐					
Basket	✓						

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.mousePositon` 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.