CHAPTER 30

PROTOTYPE 3: *SPACE SHMUP*

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

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

Getting Started: Prototype 3

In this project, you make a prototype for a classic space-based SHMUP. This chapter will get you to the same basic prototype level as the previous two chapters, and the next chapter will show you how to implement several additional features. Figure 30.1 shows an image of what the finished prototype will look like after both chapters. These images show the player ship at the bottom surrounded by a green shield as well as several enemy types and upgrades (the power-up cubes marked B, O, and S).

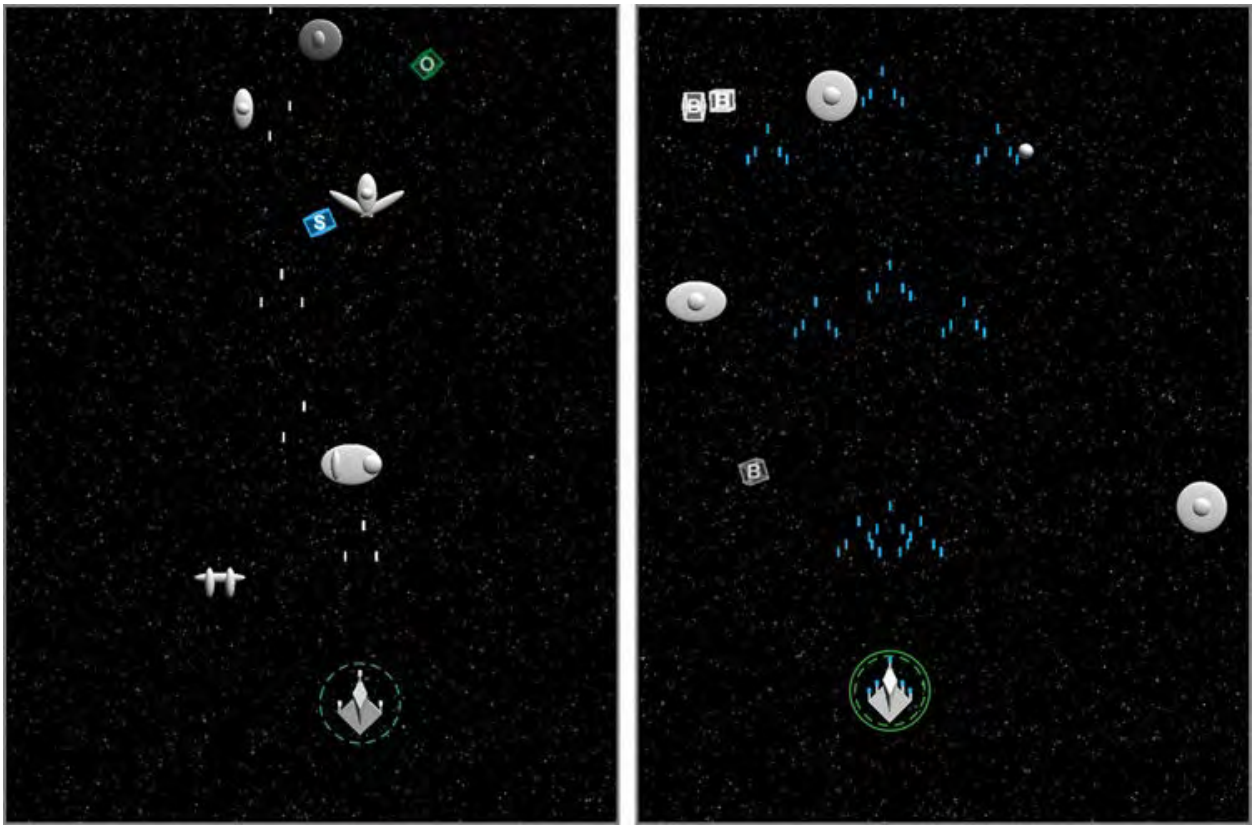

Figure 30.1 Two views of the *Space SHMUP* game prototype. The player is using the blaster weapon in the left image and the spread weapon in the right.

Importing a Unity Asset Package

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

SET UP THE PROJECT FOR THIS CHAPTER

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on this procedure, see Appendix A, "Standard Project Setup Procedure."

- **Project name:** Space SHMUP Prototype
- **Scene name:** _Scene_0

- **Project folders:** __Scripts (2 underscores before "Scripts"), _Materials, _Prefabs
- **Download and import package:** Find Chapter 30 at http://book.prototools.net
- **C# script names**: (none yet)
- **Rename:** Change Main Camera to _MainCamera

To download and install the package mentioned in the sidebar "Set Up the Project for This Chapter," first follow the URL listed (http://book.prototools.net) and search for this chapter. Download C30_Space_SHMUP_Starter.unitypackage to your machine, which will usually place it in your *Downloads* folder. Open your project in Unity and select *Assets* > *Import Package* > *Custom Package* from the menu bar. Navigate to and select *C30_Space_SHMUP_Starter.unitypackage* from your Downloads folder. The import dialog box opens, as shown in Figure 30.2.

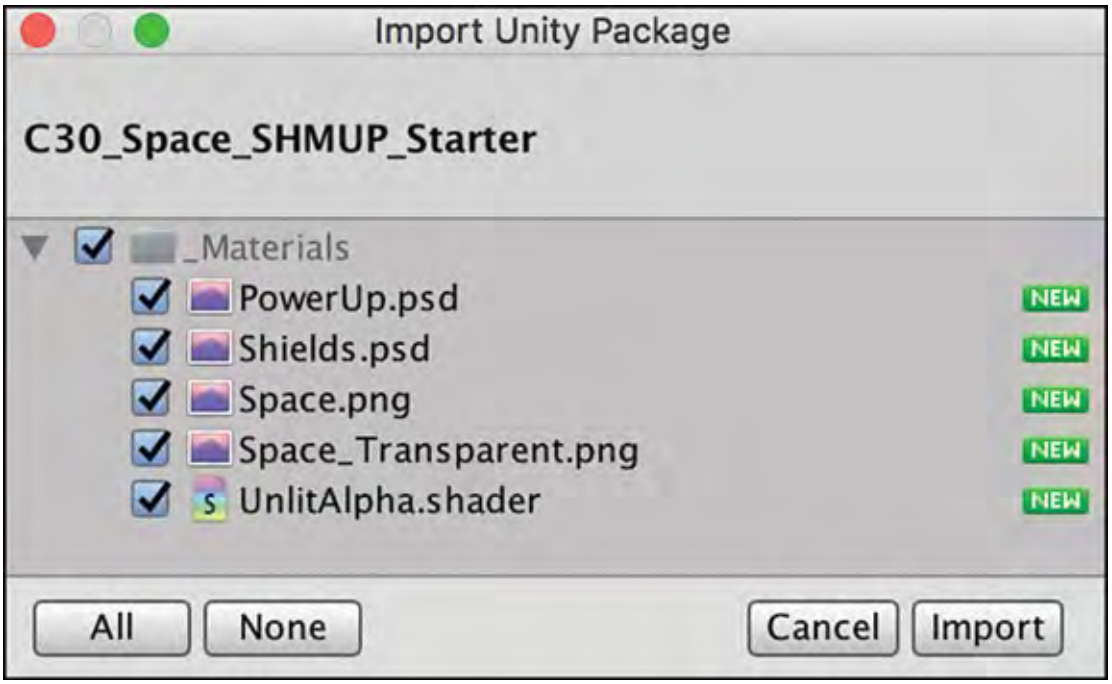

Figure 30.2 The .unitypackage import dialog box

Select all the files as shown in Figure 30.2 (by clicking the *All* button), and click *Import*. This places four new *textures* and one new *shader* into the *_Materials* folder. Textures are usually just image files. The creation of textures is beyond the scope of this book, but many books and online tutorials cover texture creation. *Adobe Photoshop* is probably the most commonly used image editing tool, but it is

very expensive. A common open source alternative is *Gimp* (http://www.gimp.org), and a very good, surprisingly cheap commercial competitor is *Affinity Photo* (https://affinity.serif.com/photo).

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

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

Setting the Scene

Follow these steps to set the scene (use a pencil to check them off as you go):

1. Select *Directional Light* in the Hierarchy and set its transform to:

   P:[0, 20, 0] R:[50, –30, 0] S:[1, 1, 1]
2. Make sure that you renamed *Main Camera* to *_MainCamera* (as you were instructed in the project setup sidebar). Select *_MainCamera* and set its transform to:

   P:[0, 0, -10] R:[0, 0, 0] S:[1, 1, 1]
3. In the Camera component of *_MainCamera*, set the following. Then save your scene.
   - *Clear Flags* to Solid Color
   - *Background* to black (with 255 alpha; RGBA:[0, 0, 0, 255])
   - *Projection* to Orthographic
   - *Size* to 40 (after setting Projection)
   - *Near Clipping Plane* to 0.3
   - *Far Clipping Plane* to 100
4. Because this game will be a vertical, top-down shooter, you need to set an

aspect ratio for the Game pane that is in portrait orientation. In the Game pane, click the pop-up menu list of aspect ratios that should currently show *Free Aspect* (see Figure 30.3). At the bottom of the list is a + symbol. Click it to add a new aspect ratio preset. Set the values to those shown in Figure 30.3, and then click *Add*. Set the Game pane to this new *Portrait (3:4)* aspect ratio.

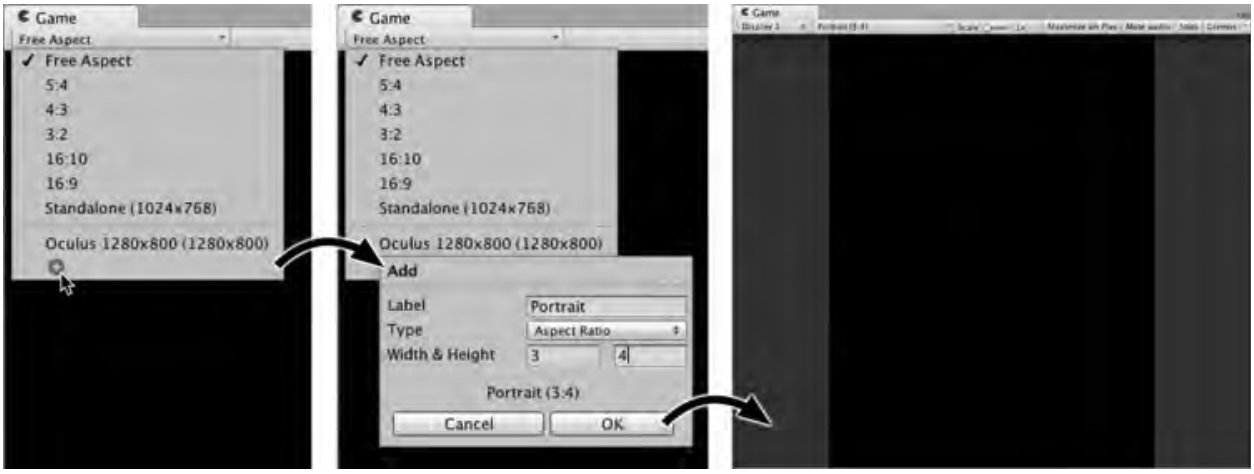

Figure 30.3 Adding a new aspect ratio preset to the Game pane

Making the Hero Ship

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

1. Create an empty GameObject and name it *_Hero* (*GameObject > Create Empty*). Set its transform to P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1].
2. Create a cube (*GameObject > 3D Object > Cube*) and drag it onto _Hero, making it a child thereof. Name the cube *Wing* and set its transform to P:[0, -1, 0] R:[0, 0, 45] S:[3, 3, 0.5].
3. Create an empty GameObject, name it *Cockpit*, and make it a child of _Hero.
4. Create a cube and make it a child of Cockpit (you can do this by right-clicking on Cockpit and choosing *3D Object > Cube*). Set the Cube's transform to P:[0, 0, 0] R:[315, 0, 45] S:[1, 1, 1].
5. Select Cockpit again and set its transform to P:[0, 0, 0] R:[0, 0, 180] S:[1, 3, 1]. This uses the same trick you learned in Chapter 27, "Object-Oriented Thinking," to make a quick, angular ship.
6. Select *_Hero* in the Hierarchy and click the *Add Component* button in the

Inspector. Choose *New Script* from the pop-up menu. Name the script *Hero*, double-check that *Language* is C Sharp, and click *Create and Add*. This is another way to make a new script and attach it to a GameObject. In the Project pane, move the Hero script into the __Scripts folder.

7. Add a Rigidbody component to _Hero by selecting *_Hero* in the Hierarchy and then choosing *Add Component* > *Physics* > *Rigidbody* from the *Add Component* button in the Inspector. Set the following on the Rigidbody component of _Hero:
   - *Use Gravity* to false (unchecked)
   - *isKinematic* to true (checked)
   - *Constraints*: freeze Z position and X, Y, and Z rotation (by checking them)

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

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

The Hero `Update()` Method

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

The last line (marked `// c`) rotates the ship based on the input. Although you earlier froze the rotation in _Hero's Rigidbody component, you can still manually set the rotation on a Rigidbody with isKinematic set to true. (As discussed in the previous chapter, isKinematic = true means that the Rigidbody will be tracked by the physics system but that it will not move automatically due to Rigidbody.velocity.) This rotation makes the movement of the ship feel more dynamic and expressive, or "juicy."[1]

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

Click here to view code image

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

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

    [Header("Set in Inspector")]
    // These fields control the movement of the ship
    public float            speed = 30;
    public float            rollMult = -45;
    public float            pitchMult = 30;

    [Header("Set Dynamically")]
    public float            shieldLevel = 1;

    void Awake() {
        if (S == null) {
        S = this; // Set the
Singleton                                       // a
    } else {
        Debug.LogError("Hero.Awake() - Attempted to assign second
Hero.S!");
    }
}

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

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

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

a. The singleton for the Hero class (see *Software Design Patterns* in Appendix B). The code in `Awake()` shows an error in the Console pane if you try to set Hero.S after it has already been set (which would happen if somehow there were two GameObjects in the same scene that both had Hero scripts attached or two Hero components attached to a single GameObject).

b. These two lines use Unity's Input class to pull information from the Unity

InputManager. See the sidebar for more information.

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

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

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

INPUT.GETAXIS() AND THE INPUTMANAGER

Much of the code in the `Hero.Update()` code listing should look familiar to you, though this is the first time in the book that you've seen the `Input.GetAxis()` method. Unity's InputManager allows you to configure various input axes, and those axes can be read through `Input.GetAxis()`. To view the default Input axes, choose *Edit* > *Project Settings* > *Input* from the menu bar.

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

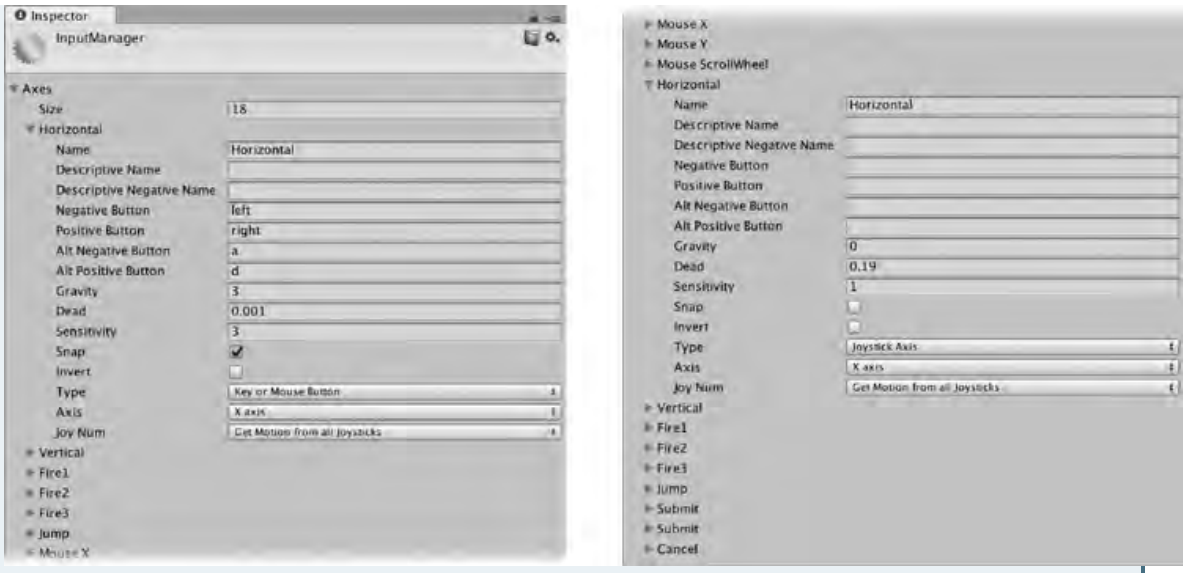

Figure 30.4 Unity's InputManager showing some of the default settings (split in two halves)

Every call to `Input.GetAxis()` returns a float between –1 and 1 in value (with a default of 0). Each axis in the InputManager also includes values for *Sensitivity* and *Gravity*, though these are only used for *Key or Mouse Button* input (see the left image of Figure 30.4). Sensitivity and gravity cause the axis value to *interpolate* smoothly when a key is pressed or released (i.e., instead of immediately jumping to the final value, the axis value will blend from the original value to the final value over time). In the Horizontal axis shown, a sensitivity of 3 means that when the right-arrow key is pressed, it takes 1/3 of a second for the value to interpolate from 0 to 1. A gravity of 3 means that when the right-arrow key is released, it takes 1/3 of a second for the axis value to interpolate back to 0. The higher the sensitivity or gravity, the faster the interpolation takes place.

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

The Hero Shield

The shield for _Hero is a combination of a transparent, textured quad (to provide

the visuals) and a Sphere Collider (for collision handling):

1. Create a new quad (*GameObject > 3D Object > Quad*). Rename the quad *Shield* and make it a child of _Hero. Set the transform of Shield to P:[0, 0, 0], R:[0, 0, 0], S:[8, 8, 8].
2. Select *Shield* in the Hierarchy and delete the existing Mesh Collider component by clicking the tiny gear to the right of the Mesh Collider name in the Inspector and choosing *Remove Component* from the pop-up menu. Add a *Sphere Collider* component (*Component > Physics > Sphere Collider*).
3. Create a new material (*Assets > Create > Material*), name it *Mat_Shield*, and place it in the _Materials folder in the Project pane. Drag *Mat_Shield* onto the *Shield* under _Hero in the Hierarchy to assign it to the Shield quad.
4. Select *Shield* in the Hierarchy, and you can now see Mat_Shield in the Inspector for Shield. Set the Shader of Mat_Shield to *ProtoTools > UnlitAlpha*. Below the shader selection pop-up for Mat_Shield, you should see an area that allows you to choose the main color for the material as well as the texture. (If you don't see the shader properties, click once on the name *Mat_Shield* in the Inspector, and it should appear.)
5. Click *Select* in the bottom-right corner of the texture square and select the texture named *Shields*. Click the color swatch next to *Main Color* and choose a bright green (RGBA:[0, 255, 0, 255]). Then set:
   - *Tiling.x* to 0.2
   - *Offset.x* to 0.4
   - *Tiling.y* should remain 1.0
   - *Offset.y* should remain 0

The Shield texture was designed to be split into five sections horizontally. The X Tiling of 0.2 causes Mat_Shield to only use 1/5 of the total Shield texture in the X direction, and the X Offset chooses which fifth. Try X Offsets of 0, 0.2, 0.4, 0.6, and 0.8 to see the different levels of shield strength.

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

Click here to view code image

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

public class Shield : MonoBehaviour {
    [Header("Set in Inspector")]
    public float    rotationsPerSecond = 0.1f;

    [Header("Set Dynamically")]
    public int      levelShown = 0;

    // This non-public variable will not appear in the Inspector
    Material        mat;
a

    void Start() {
        mat = GetComponent<Renderer>
().material;                                // b
    }

    void Update () {
        // Read the current shield level from the Hero Singleton
        int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel
);              // c
        // If this is different from levelShown...
        if (levelShown != currLevel) {
            levelShown = currLevel;
            // Adjust the texture offset to show different shield level
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown,
0);      // d
        }
        // Rotate the shield a bit every frame in a time-based way
        float rZ = -(rotationsPerSecond*Time.time*3600) %
360f;             // e
        transform.rotation = Quaternion.Euler( 0, 0, rZ );
    }
}
```

a. The Material field `mat` is *not* declared public, so it will not be visible in the Inspector, and it will not be able to be accessed outside of this Shield class.

b. In `Start()`, `mat` is defined as the material of the Renderer component on this GameObject (Shield in the Hierarchy). This allows you to quickly set the texture offset in the line marked `// d`.

c. `currLevel` is set to the *floor* of the current `Hero.S.shieldLevel` float. Flooring the `shieldLevel` ensures that the shield jumps to the new X Offset rather than show an Offset between two shield icons.

d. This line adjusts the X Offset of Mat_Shield to show the proper shield

level.

e. This line and the next cause the Shield GameObject to rotate slowly about the z axis.

Keeping _Hero on Screen

The motion of your _Hero ship should feel pretty good now, and the rotating shield looks pretty nice, but at this point, you can easily drive the ship off the screen. To resolve this, you're going to make a reusable *component* script.[2] You can read more about the component software design pattern in Chapter 27, "Object-Oriented Thinking" and under *Software Design Patterns* in Appendix B, "Useful Concepts." In brief, a component is a small piece of code that is meant to work alongside others to add functionality to a GameObject without conflicting with other code on that object. Unity's components that you've worked with in the Inspector (e.g., Renderer, Transform, and so on) all follow this pattern. Now, you'll do the same with a small script to keep _Hero on screen. Note that this script only works with orthographic cameras.

1. Select *_Hero* in the Hierarchy and using the *Add Component* button in the Inspector, choose *Add Component* > *New Script*. Name the script *BoundsCheck* and click *Create and Add*. Drag the BoundsCheck script in the Project pane into the __Scripts folder.
2. Open the BoundsCheck script and add the following code:

Click here to view code image

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

// To type the next 4 lines, start by typing /// and then Tab.
/// <summary>
/// Keeps a GameObject on screen.
/// Note that this ONLY works for an orthographic Main Camera at [ 0, 0,
0 ].
/// </summary>
public class BoundsCheck : MonoBehaviour
{                                       // a
    [Header("Set in Inspector")]
    public float  radius = 1f;

    [Header("Set Dynamically")]
    public float  camWidth;
    public float  camHeight;
```

```
    void Awake() {
        camHeight =
Camera.main.orthographicSize;                                   // b
        camWidth = camHeight *
Camera.main.aspect;                                 // c
     }

    void LateUpdate ()
{                                                                   // d
        Vector3 pos = transform.position;

        if (pos.x > camWidth - radius) {
            pos.x = camWidth - radius;
        }

        if (pos.x < -camWidth + radius) {
            pos.x = -camWidth + radius;
        }

        if (pos.y > camHeight - radius) {
            pos.y = camHeight - radius;
        }
        if (pos.y < -camHeight + radius) {
            pos.y = -camHeight + radius;
        }

        transform.position = pos;
    }

    // Draw the bounds in the Scene pane using OnDrawGizmos()
    void OnDrawGizmos ()
{                                                                   // e
        if (!Application.isPlaying) return;
        Vector3boundSize = newVector3(camWidth* 2, camHeight* 2, 0.1f);
        Gizmos.DrawWireCube(Vector3.zero, boundSize);
    }
}
```

a. Because this is intended to be a reusable piece of code, adding some internal documentation to it is useful. The lines above the class declaration that all begin with `///` are part of C#'s built-in documentation system.[3] After you've typed this, it interprets the text between the `<summary>` tags as a summary of what the class is used for. After typing it, hover your mouse over the name BoundsCheck on the line marked `// a`, and you should see a pop-up with this class summary.

b. `Camera.main` gives you access to the first camera with the tag *MainCamera* in your scene. Then, if the camera is orthographic, `.orthographicSize` gives you the Size number from the Camera

Inspector (which is 40 in this case). This makes `camHeight` the distance from the origin of the world (position [0, 0, 0]) to the top or bottom edge of the screen in world coordinates.

c. `Camera.main.aspect` is the aspect ratio of the camera in width/height as defined by the aspect ratio of the Game pane (currently set to *Portrait (3:4)*). By multiplying `camHeight` by `.aspect`, you can get the distance from the origin to the left or right edge of the screen.

d. `LateUpdate()` is called every frame after `Update()` has been called on all GameObjects. If this code were in the `Update()` function, it might happen either before or after the `Update()` call on the Hero script. Putting this code in `LateUpdate()`avoids causing a *race condition* between the two `Update()` functions and ensures that `Hero.Update()` moves the _Hero GameObject to the new position each frame before this function is called and bounds _Hero to the screen.

e. `OnDrawGizmos()` is a built-in MonoBehaviour method that can draw to the Scene pane.

A *race condition* is an instance where the order in which two pieces of code execute (i.e., A before B or B before A) matters, but you don't have control over that order. For example, in this code, if `BoundsCheck.LateUpdate()` were to execute before `Hero.Update()`, the _Hero GameObject would potentially be moved out of bounds (because it would first limit the ship to the bounds and *then* move the ship). Using `LateUpdate()` in BoundsCheck enforces the execution order of the two scripts.

3. Click *Play* and try flying your ship around. Based on the default setting for `radius`, you should see that the ship stops while it is still 1m on screen. If you set `BoundsCheck. radius` to 4 in the _Hero Inspector, the ship stays entirely on screen. If you set `radius` to -4, the ship can exit the edge of the screen but will be locked there, ready to come right back on. Stop playback and set `radius` to 4.

Adding Some Enemies

Chapter 26, "Classes," covered a bit about the Enemy class and subclasses for a game like this. There you learned about setting up a superclass for all enemies that can be extended by subclasses. For this game, will you extend that further in the next chapter, but first, let's create the artwork.

Enemy Artwork

Because the hero ship has such an angular aesthetic, all the enemies will be constructed of spheres as shown in Figure 30.5.

Figure 30.5 Each of the five enemy ship types (lighting will differ slightly in Unity)

To create the artwork for Enemy_0 do the following:

1. Create an empty GameObject, name it *Enemy_0*, and set its transform to P:[-20, 10, 0], R:[0, 0, 0], S:[1, 1, 1]. This position is to make sure it doesn't overlap with _Hero as you build it.
2. Create a sphere named *Cockpit*, make it a child of Enemy_0, and set its transform to P:[0, 0, 0], R:[0, 0, 0], S:[2, 2, 1].
3. Create a second sphere named *Wing*, make it a child of Enemy_0, and set its transform to P:[0, 0, 0], R:[0, 0, 0], S:[5, 5, 0.5].

Another way of writing the preceding three steps for Enemy_0 would be:

Enemy_0 (Empty)	P:[-20, 10, 0]	R:[0, 0, 0]	S:[1, 1, 1]
Cockpit (Sphere)	P:[0, 0, 0]	R:[0, 0, 0]	S:[2, 2, 1]
Wing (Sphere)	P:[0, 0, 0]	R:[0, 0, 0]	S:[5, 5, 0.5]

4. Follow this formatting to make the remaining four enemies. When finished, they should look like the enemies shown in Figure 30.5.

Enemy_1

Enemy_1 (Empty)	P:[-10, 10, 0]	R:[0, 0, 0]	S:[1, 1, 1]
Cockpit (Sphere)	P:[0, 0, 0]	R:[0, 0, 0]	S:[2, 2, 1]
Wing (Sphere)	P:[0, 0, 0]	R:[0, 0, 0]	S:[6, 4, 0.5]

Enemy_2

Enemy_2 (Empty)	P:[0, 10, 0]	R:[0, 0, 0]	S:[1, 1, 1]
Cockpit (Sphere)	P:[-1.5, 0, 0]	R:[0, 0, 0]	S:[1, 3, 1]
Reactor (Sphere)	P:[2, 0, 0]	R:[0, 0, 0]	S:[2, 2, 1]
Wing (Sphere)	P:[0, 0, 0]	R:[0, 0, 0]	S:[6, 4, 0.5]

Enemy_3

Enemy_3 (Empty)	P:[10, 10, 0]	R:[0, 0, 0]	S:[1, 1, 1]
CockpitL (Sphere)	P:[-1, 0, 0]	R:[0, 0, 0]	S:[1, 3, 1]
CockpitR (Sphere)	P:[1, 0, 0]	R:[0, 0, 0]	S:[1, 3, 1]
Wing (Sphere)	P:[0, 0.5, 0]	R:[0, 0, 0]	S:[5, 1, 0.5]

Enemy_4

Enemy_4 (Empty)	P:[20, 10, 0]	R:[0, 0, 0]	S:[1, 1, 1]
Cockpit (Sphere)	P:[0, 1, 0]	R:[0, 0, 0]	S:[1.5, 1.5, 1.5]
Fuselage (Sphere)	P:[0, 1, 0]	R:[0, 0, 0]	S:[2, 4, 1]
WingL	P:[	R:[0,	S:[5, 1,

(Sphere)	-1.5, 0, 0]	0, -30]	0.5]
WingR (Sphere)	P:[1.5, 0, 0]	R:[0, 0, 30]	S:[5, 1, 0.5]

5. You must add a Rigidbody component to each of the enemy GameObjects (i.e., Enemy_0, Enemy_1, Enemy_2, Enemy_3, and Enemy_4). To add a Rigidbody, complete these steps:
   a. Select *Enemy_0* in the Hierarchy and choose *Component > Physics > Rigidbody* from the menu bar to add the Rigidbody component.
   b. In the Rigidbody component for the enemy, set *Use Gravity* to false.
   c. Set *isKinematic* to true.
   d. Open the disclosure triangle for *Constraints* and freeze Z position and X, Y, and Z rotation.
6. Now copy the Rigidbody component from Enemy_0 to all four other enemies. Do the following steps for each of the four other enemies:
   a. Select *Enemy_0* in the Hierarchy and click the little *gear button* in the top-right corner of the Enemy_0 *Rigidbody* component.
   b. From the pop-up menu, choose *Copy Component*.
   c. Select the enemy that you want to add a Rigidbody to (e.g., Enemy_1).
   d. Click the gear button in the top-right of the Transform component on the enemy.
   e. Choose *Paste Component As New* from the pop-up menu.

This attaches a Rigidbody component to the enemy with the same settings as the Rigidbody that you copied from Enemy_0. Be sure to do this for all enemies. If a moving GameObject doesn't have a Rigidbody component, the GameObject's collider location will not move with the GameObject, but if a moving GameObject does have a Rigidbody, the colliders of both it and all of its children are updated every frame (which is why you don't need to add a Rigidbody component to any of the children of the enemy GameObjects).

7. Drag each of these enemies to the _Prefabs folder of the Project pane to create a prefab for each.
8. Delete all the enemy instances from the Hierarchy except for Enemy_0.

The Enemy C# Script

To create the Enemy script, follow these steps:

1. Create a new C# script named *Enemy* and place it into the __Scripts folder.
2. Select *Enemy_0* in the Project pane (not in the Hierarchy). In the Inspector for Enemy_0, click the *Add Component* button and choose *Scripts* > *Enemy* from the pop-up menu. After doing this, when you click on Enemy_0 in either the Project or Hierarchy panes, you should see an *Enemy (Script)* component attached.
3. Open the *Enemy* script in MonoDevelop and enter the following code:

Click here to view code image

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

public class Enemy : MonoBehaviour {
    [Header("Set in Inspector: Enemy")]
    public float      speed = 10f;      // The speed in m/s
    public float      fireRate = 0.3f;  // Seconds/shot (Unused)
    public float      health = 10;
    public int         score = 100;      // Points earned for destroying
this

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

    void Update() {
        Move();
    }

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

a. As was discussed in Chapter 26, "Classes," a *property* is a function masquerading as a field. This means that you can get and set the value of `pos` as if it were a class variable of Enemy.

b. The `Move()` method gets the current position of this Enemy_0, moves it in the downward Y direction, and assigns it back to `pos` (setting the position of the GameObject).

4. In Unity, click *Play*, and the instance of Enemy_0 in the scene should move toward the bottom of the screen. However, with the current code, this instance will continue off screen and exist until you stop your game. You need to have the enemy destroy itself after it has moved entirely off screen. This is a great place to reuse the BoundsCheck component.

5. To attach the BoundsCheck script to the Enemy_0 prefab, select the *Enemy_0* prefab in the Hierarchy (not the Project pane this time). In the Inspector, click *Add Component* and choose *Add Component* > *Scripts* > *BoundsCheck*. This attaches the script to the instance of Enemy_0 in the Hierarchy, but it has not yet attached it to the Enemy_0 prefab in the Project pane. You can tell this because all the text in the *BoundsCheck (Script)* component is bold.

6. To apply this change made to the Enemy_0 instance back to its prefab, click *Apply* at the top of the Inspector for the Enemy_0 instance in the Hierarchy. Now check the Enemy_0 prefab in the Project pane to see that the script is attached.

7. Select the *Enemy_0* instance in the Hierarchy and set the `radius` value in the BoundsCheck Inspector to -2.5. Note that this value is bolded because it is different from the value on the prefab. Click *Apply* at the top of the Inspector again, and the radius value will no longer be bolded, showing you that it is the same value as the one on the prefab.

8. Click *Play*, and you'll see that the Enemy_0 instance stops right after it has gone off screen. However, instead of forcing Enemy_0 to remain on screen, you really want to be able to check whether it has gone off screen and then destroy it when it has.

9. To do so, make the following bolded modifications to the BoundsCheck script.

Click here to view code image

```
/// <summary>
/// Checks whether a GameObject is on screen and can force it to stay on
screen.
/// Note that this ONLY works for an orthographic Main Camera.
/// </summary>
public class BoundsCheck : MonoBehaviour {
```

```
    [Header("Set in Inspector")]
    public float     radius = 1f;
    public bool     keepOnScreen =
true;                                    // a

    [Header("Set Dynamically")]
    public bool     isOnScreen =
true;                                      // b
    public float     camWidth;
    public float     camHeight;

    void Awake() { … }      // Remember, ellipses mean to not alter this
method.

    void LateUpdate () {
        Vector3 pos = transform.position;
              // c
        isOnScreen =
true;                                                  // d

        if ( pos.x > camWidth - radius ) {
            pos.x = camWidth - radius;
            isOnScreen =
false;                                             // e
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x = -camWidth + radius;
            isOnScreen =
false;                                             // e
        }

        if ( pos.y > camHeight - radius ) {
            pos.y = camHeight - radius;
            isOnScreen =
false;                                             // e
        }
        if ( pos.y < -camHeight + radius ) {
            pos.y = -camHeight + radius;
            isOnScreen =
false;                                             // e
        }

        if ( keepOnScreen && !isOnScreen )
{                                  // f
            transform.position =
pos;                                          // g
            isOnScreen = true;
        }
    }

    …
}
```

a. `keepOnScreen` allows you to choose whether BoundsCheck forces a GameObject to stay on screen (`true`) or allows it to exit the screen and notifies you that it has done so (`false`).

b. `isOnScreen` turns false if the GameObject exits the screen. More accurately, it turns false if the GameObject goes past the edge of the screen minus the value of `radius`. This is why `radius` is set to –2.5 for Enemy_0, so that it is completely off screen before `isOnScreen` is set to `false`.

c. Remember that ellipses in code mean you should *not* modify the `Start()` method.

d. `isOnScreen` is set to true until proven false. This allows the value of `isOnScreen` to return to true if the GameObject was off screen in the last frame but has come back on in this frame.

e. If any of these four `if` statements are true, then the GameObject is outside of the area it is supposed to be in. `isOnScreen` is set to `false`, and `pos` is adjusted to a position that would bring the GameObject back "on screen."

f. If `keepOnScreen` is `true`, then you are trying to force the GameObject to stay on screen. If `keepOnScreen` is `true` and `isOnScreen` is `false`, then the GameObject has gone out of bounds and needs to be brought back in. In this case, `transform.position` is set to the updated `pos` that is on screen, and `isOnScreen` is set to `true` because this position assignment has now moved the GameObject back on screen.

If `keepOnScreen` is `false`, then `pos` is *not* assigned back to `transform.position`, the GameObject is allowed to go off screen, and `isOnScreen` is allowed to remain `false`. The other possibility is that the GameObject was on screen the whole time, in which case, `isOnScreen` would still be `true` from when it was set on line `// d`.

g. Note that this line is now indented and inside the `if` statement on line `// f`.

Happily, all of these modifications to the code do not negatively impact the way it was used for _Hero, and everything there still works fine. You've created a reusable component that you can apply to both _Hero and the Enemy GameObjects.

Deleting the Enemy When It Goes Off Screen

Now that BoundsCheck can tell you when Enemy_0 goes off screen, you need to set it to properly do so.

1. Set `keepOnScreen` to `false` in the *BoundsCheck (Script)* component of the Enemy_0 prefab in the _Prefabs folder of the Project pane.
2. To ensure that this propagates to the Enemy_0 instance in the Hierarchy, select the instance in the Hierarchy and click the gear to the right of the *BoundsCheck (Script)* component heading in the Inspector. From the gear pop-up menu, choose *Revert to Prefab* to set the values of the instance in the Hierarchy to those on the prefab. When you've done this, the *BoundsCheck (Script)* component on both the Enemy_0 prefab in the Project pane and the Enemy_0 instance in the Hierarchy should look like Figure 30.6.

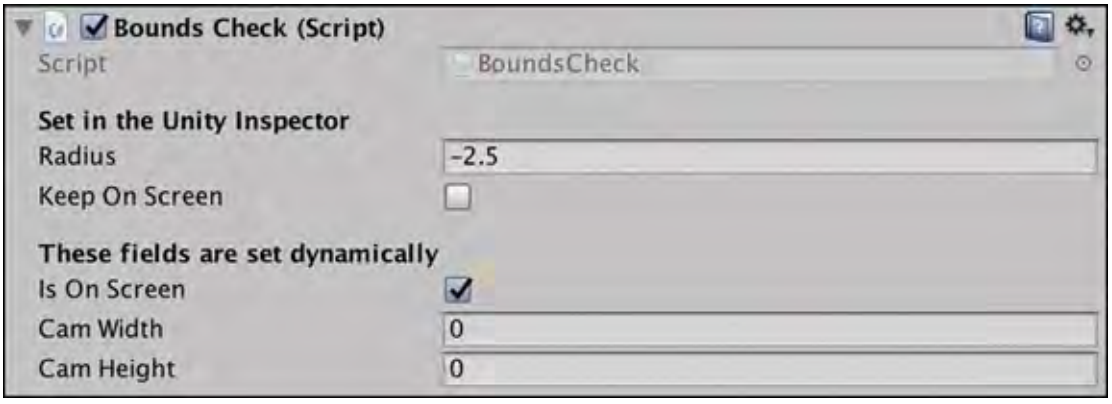

Figure 30.6 The *BoundsCheck (Script)* component settings for both the prefab and instance of Enemy_0

3. Add the following bold code to the Enemy script:

Click here to view code image

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

    private BoundsCheck
bndCheck;                                                  // a

    void Awake()
{                                                                  // b
        bndCheck = GetComponent<BoundsCheck>();
    }

    …

    void Update() {
```

```
        Move();

        if ( bndCheck != null && !bndCheck.isOnScreen )
{                   // c
            // Check to make sure it's gone off the bottom of the screen
            if ( pos.y < bndCheck.camHeight - bndCheck.radius )
{            // d
                // We're off the bottom, so destroy this GameObject
                Destroy( gameObject );
            }
        }
    }

    …
}
```

a. This private variable allows this Enemy script to store a reference to the *BoundsCheck (Script)* component attached to the same GameObject.

b. This `Awake()` method searches for the BoundsCheck script component attached to this same GameObject. If there is not one, `bndCheck` will be set to `null`. Code like this that searches for components and caches references is often placed in the `Awake()` method so that the references are ready immediately when the GameObject is instantiated.

c. First checks to make sure `bndCheck` is not `null`. If you attached the Enemy script to a GameObject without attaching a BoundsCheck script as well, this could be the case. Only if `bndCheck != null` does the script check to see whether the GameObject is not on screen (according to BoundsCheck).

d. If `isOnScreen` is `false`, this line checks to see whether it is off screen because it has a `pos.y` that is too negative (i.e., it has gone off the bottom of the screen). If this is the case, the GameObject is destroyed.

This works and does what we want, but it seems a bit messy to be doing the same comparison of `pos.y` versus the `camHeight` and `radius` both here and in BoundsCheck.

It is generally considered good programming style to let each C# class (or component) handle the job it is meant to do and not have crossover like this. As such, let's alter BoundsCheck to be able to tell you in which direction the GameObject has gone off screen.

3. Modify the BoundsCheck script by adding the bolded code that follows:

Click here to view code image

```
public class BoundsCheck : MonoBehaviour {
    …
    public float     camHeight;
    [HideInInspector]
    public bool     offRight, offLeft, offUp,
offDown;                        // a

    void Start() { … }

    void LateUpdate () {
        Vector3 pos = transform.position;
        isOnScreen = true;
        offRight = offLeft = offUp = offDown =
false;                         // b

        if ( pos.x > camWidth - radius ) {
            pos.x = camWidth - radius;
            offRight =
true;                                                  // c
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x = -camWidth + radius;
            offLeft =
true;                                                   // c
        }

        if ( pos.y > camHeight - radius ) {
            pos.y = camHeight - radius;
            offUp =
true;                                                     // c
        }

        if ( pos.y < -camHeight + radius ) {
            pos.y = -camHeight + radius;
            offDown =
true;                                                   // c
        }

        isOnScreen = !(offRight || offLeft || offUp ||
offDown);             // d
        if ( keepOnScreen && !isOnScreen ) {
            transform.position = pos;
            isOnScreen = true;
            offRight = offLeft = offUp = offDown =
false;                     // e
        }
    }

    …
}
```

a. Here you declare four variables, one for each direction in which the GameObject could go off screen. As bools, they all will default to `false`. The `[HideInInspector]` line preceding this causes these four public fields to not appear in the Inspector, though they are still public variables and can still be read (or set) by other classes. `[HideInInspector]` applies to all four `off__bools` (i.e., `offRight`, `offLeft`, and so on) because they are all declared on the line beneath it. If the `off__` bools were declared on four separate lines, a `[HideInInspector]` line would need to precede each line individually to achieve the same effect.

b. At the beginning of each `LateUpdate()` you set all four `off__` bools to `false`. In this line, `offDown` is first set to `false`, then `offUp` is set to the value of `offDown` (i.e., `false`), and so on until all `off__` bools hold the value `false`. This takes the place of the old line that set `isOnScreen` to `true`.

c. Each instance of `isOnScreen = false;` has now been replaced with an `off__ = true;` so that you know in which direction the GameObject has exited the screen. The possibility exists for two of these `off__` bools to both be true; for example, when the GameObject has exited the bottom-right corner of the screen.

d. Here, you set `isOnScreen` based on the values of all the `off__` bools. First, inside the parentheses, you take the logical OR (`||`) of all the `off__` bools. If one or more of them are `true`, this evaluates to `true`. You then take the NOT (`!`) of that and assign it to `isOnScreen`. So, if one or more of the `off__` bools are `true`, `isOnScreen` will be `false`, otherwise `isOnScreen` is set to `true`.

e. If `keepOnScreen` is true, and this GameObject is forced back on screen, `isOnScreen` is set to true, and all the `off__` bools are set to false.

4. Now, make the following bolded changes to the Enemy script to take advantage of the improvements to BoundsCheck.

Click here to view code image

```
public class Enemy : MonoBehaviour {
    …

    void Update() {
        Move();
        if ( bndCheck != null && bndCheck.offDown )
{                       // a
            // We're off the bottom, so destroy this
```

```
GameObject             // b
            Destroy( gameObject
);                                          // b
        }
    }

    …
}
```

a. Now, you just need to check against `bndCheck.offDown` to determine whether the Enemy instance has gone off the bottom of the screen.

b. These two lines have lost one tab of indentation because there is now only one `if` clause instead of two.

This is a much simpler implementation from the viewpoint of the Enemy class, and it makes good use of the BoundsCheck component, allowing it to do its job without needlessly duplicating its functionality in the Enemy class.

Now, when you play the scene, you should see that the Enemy_0 ship moves down the screen and is destroyed as soon as it exits the bottom of the screen.

Spawning Enemies at Random

With all of this in place, instantiating a number of Enemy_0s randomly is now possible.

1. Attach a *BoundsCheck* script to *_MainCamera* and set its `keepOnScreen` field to false.
2. Create a new C# script called *Main* inside the __Scripts folder. Attach it to _MainCamera, and then enter the following code:

Click here to view code image

```
using System.Collections;           // Required for Arrays & other
Collections
using System.Collections.Generic;   // Required to use Lists or
Dictionaries
using UnityEngine;                  // Required for Unity
using UnityEngine.SceneManagement;  // For loading & reloading of scenes

public class Main : MonoBehaviour {
    static public Main S;                                   // A singleton
for Main

    [Header("Set in Inspector")]
    public GameObject[]     prefabEnemies;              // Array of
```

```
Enemy prefabs
    public float            enemySpawnPerSecond = 0.5f; // #
Enemies/second
    public float            enemyDefaultPadding = 1.5f; // Padding for
position

    private BoundsCheck     bndCheck;

    void Awake() {
        S = this;
        // Set bndCheck to reference the BoundsCheck component on this
GameObject
        bndCheck = GetComponent<BoundsCheck>();

        // Invoke SpawnEnemy() once (in 2 seconds, based on default
values)
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond
);                      // a
    }

    public void SpawnEnemy() {
        // Pick a random Enemy prefab to instantiate
        int ndx = Random.Range(0,
prefabEnemies.Length);                      // b
        GameObject go = Instantiate<GameObject>( prefabEnemies[ ndx ]
);     // c

        // Position the Enemy above the screen with a random x position
        float enemyPadding =
enemyDefaultPadding;                            // d
        if (go.GetComponent<BoundsCheck>() != null)
{                       // e
            enemyPadding = Mathf.Abs( go.GetComponent<BoundsCheck>
().radius );
        }

        // Set the initial position for the spawned
Enemy                   // f
        Vector3 pos = Vector3.zero;
        float xMin = -bndCheck.camWidth + enemyPadding;
        float xMax =  bndCheck.camWidth - enemyPadding;
        pos.x = Random.Range( xMin, xMax );
        pos.y = bndCheck.camHeight + enemyPadding;
        go.transform.position = pos;

        // Invoke SpawnEnemy() again
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond
);                      // g
    }
}
```

a. This `Invoke()` function calls the `SpawnEnemy()` method in 1/0.5

seconds (i.e., 2 seconds) based on the default values.

b. Based on the length of the array `prefabEnemies`, this chooses a random number between 0 and one less than `prefabEnemies.Length`, so if four prefabs are in the `prefabEnemies` array, it will return 0, 1, 2, or 3. The `int` version of `Random.Range()` will never return a number as high as the max (i.e., second) integer passed in. The float version is able to return the max number.

c. The random `ndx` generated is used to select a GameObject prefab from `prefabEnemies`.

d. The enemyPadding is initially set to the enemyDefaultPadding set in the Inspector.

e. However, if the selected enemy prefab has a BoundsCheck component, you instead read the `radius` from that. The absolute value of the `radius` is taken because sometimes the `radius` is set to a negative value so that the GameObject must be entirely off screen before registering as isOnScreen = false, as is the case for Enemy_0.

f. This section of the code sets an initial position for the enemy that was instantiated. It uses the BoundsCheck on this _MainCamera GameObject to get the camWidth and camHeight and chooses an X position where the spawned enemy is entirely on screen horizontally. It then chooses a Y position where the enemy is just above the screen.

g. Invoke is called again. The reason that `Invoke()` is used instead of `InvokeRepeating()` is that you want to be able to dynamically adjust the amount of time between each enemy spawn. After `InvokeRepeating()` is called, the invoked function is always called at the rate specified. Adding an `Invoke()` call at the end of `SpawnEnemy()` allows the game to adjust `enemySpawnPerSecond` on the fly and have it affect how frequently `SpawnEnemy()` is called.

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

a. Delete the instance of *Enemy_0* from the Hierarchy (leaving the prefab in the Project pane alone, of course).

b. Select *_MainCamera* in the Hierarchy.

c. Open the disclosure triangle next to `prefabEnemies` in the *Main (Script)* component of _MainCamera and set the *Size* of `prefabEnemies` to *1*.

d. Drag *Enemy_0* from the Project pane into *Element 0* of the `prefabEnemies` array.

e. *Save your scene!* Have you been remembering?

If you didn't save your scene after creating all of those enemies, you really should have. All sorts of things beyond your control could cause Unity to crash, and you don't want to have to redo work. Getting into a habit of saving your scene frequently can save you a ton of wasted time and sorrow as a developer.

4. Play your scene. You should now see an Enemy_0 spawn about once every 2 seconds, travel down to the bottom of the screen, and then disappear after it exits the bottom of the screen.

However, right now, when the _Hero collides with an enemy, nothing happens. This needs to be fixed, and to do so, you have to look at layers.

Setting Tags, Layers, and Physics

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

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

To create these layers as well as some tags that will be useful later, complete these steps:

1. Open the *Tags & Layers* manager in the Inspector pane (*Edit > Project Settings > Tags and Layers*). Tags and physics layers are different from each other, but both are set here.

2. Open the disclosure triangle next to Tags. Click the + below the Tags list and enter the tag name for each of the tags shown in the left image of Figure 30.7.

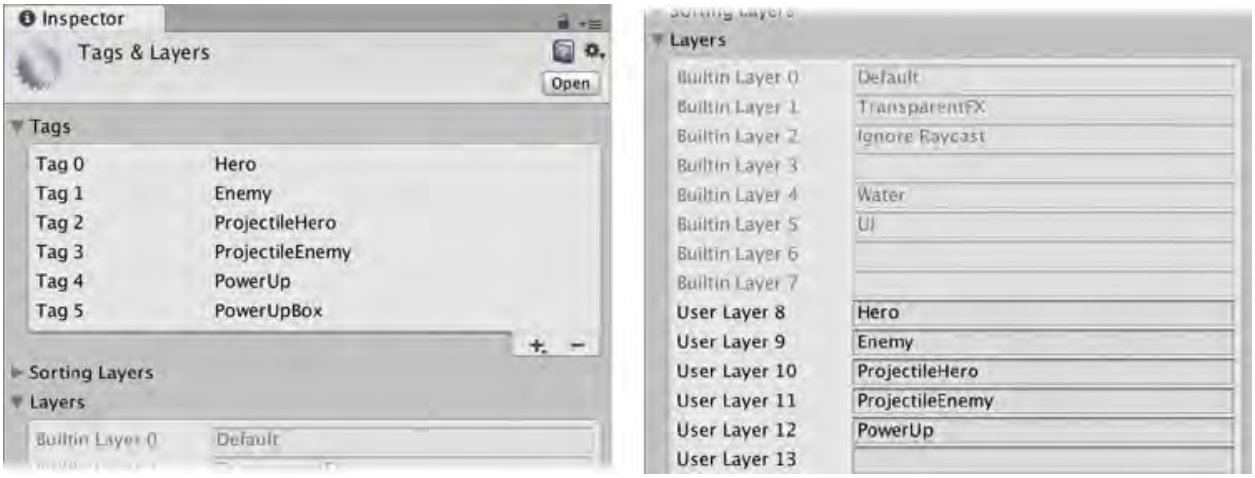

Figure 30.7 TagManager showing tags and layer names for this prototype

In case it's difficult to see, the Tag names are: Hero, Enemy, ProjectileHero, ProjectileEnemy, PowerUp, and PowerUpBox.

3. Open the disclosure triangle next to *Layers*. Starting with *User Layer 8*, enter the layer names shown in the right image of Figure 30.7. Builtin Layers 0–7 are reserved by Unity, but you can set the names of User Layers 8–31.

The Layer names are: Hero, Enemy, ProjectileHero, ProjectileEnemy, and PowerUp.

4. Open the PhysicsManager (*Edit* > *Project Settings* > *Physics*) and set the *Layer Collision Matrix* as shown in Figure 30.8.

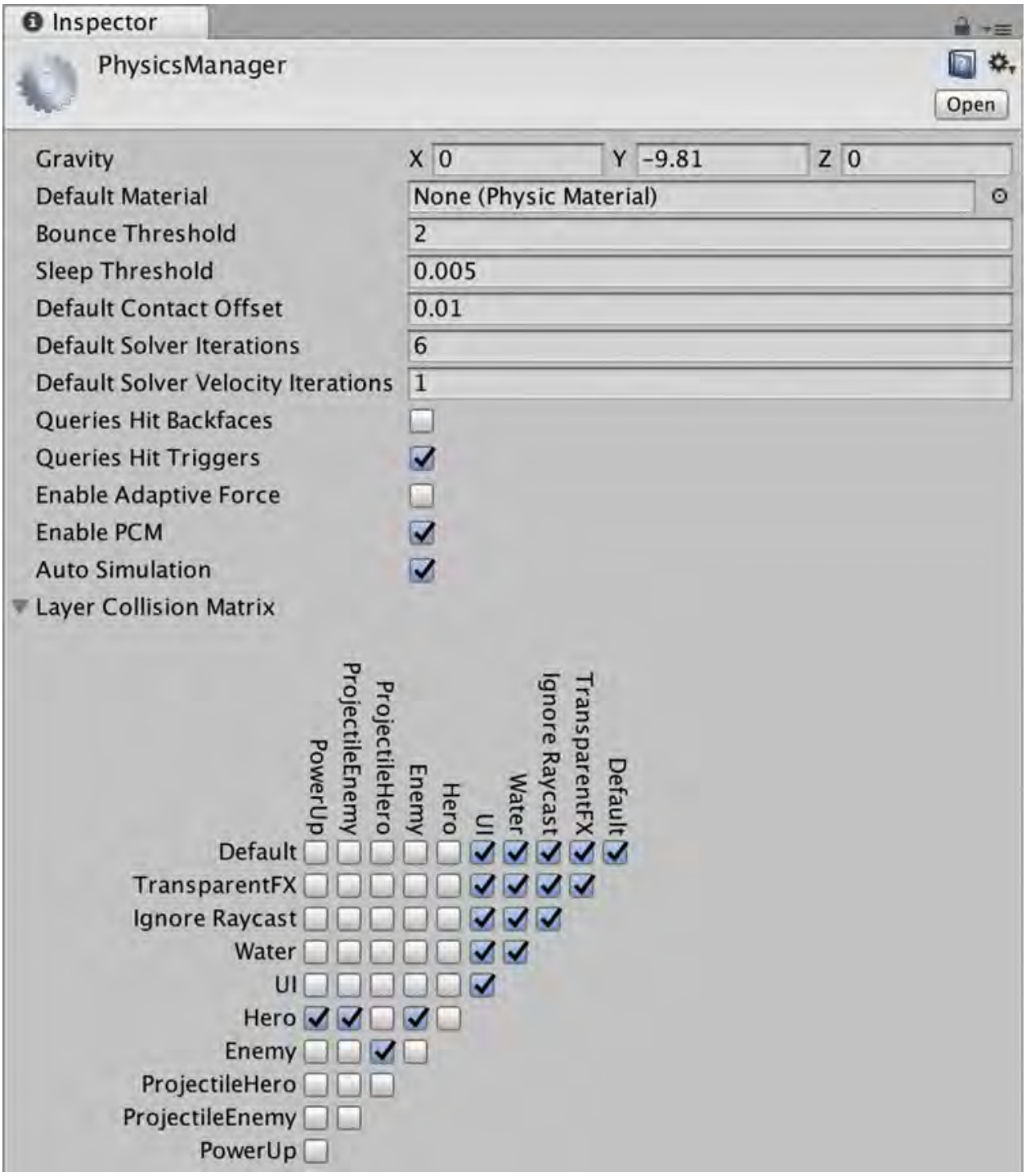

Figure 30.8 PhysicsManager with proper settings for this prototype

Note

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

Physics2D.

As you experienced in Chapter 28, the grid at the bottom of the PhysicsManager sets which layers collide with each other. If there is a check, objects in the two layers are able to collide, if there is no check, they won't. Removing checks can speed the execution of your game because it will test fewer objects versus each other for collision. As you can see in Figure 30.8, the Layer Collision Matrix that is selected achieves the collision behavior we specified earlier.

Assigning the Proper Layers to GameObjects

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

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

Making the Enemies Damage the Player

Now that the enemies and hero have colliding layers, you need to make them react to collisions with each other.

1. Open the disclosure triangle next to _Hero in the Hierarchy and select its child *Shield*. In the Inspector, set the *Sphere Collider* of Shield to be a trigger (check the box next to *Is Trigger*). You don't need things to bounce off of Shield; you just need to know when they've hit.
2. Add the following bolded method to the end of the Hero C# script:

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    void Update() {
```

```
        …
    }

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

3. Play the scene and try running into some enemies. You can see that you get a separate trigger event for each of the child GameObjects of the enemy (e.g., Cockpit and Wing) but not for the enemy itself. You need to be able to get the Enemy_0 GameObject that is the parent of Cockpit and Wing, and if you had even more deeply nested child GameObjects, you would need to still find this topmost or *root* parent.

Luckily, this is a pretty common thing to need to do, so it's part of the Transform component of any GameObject. Calling `transform.root` on any GameObject gives you the transform of the root GameObject, from which it is easy to get the GameObject itself.

4. Replace the `OnTriggerEnter()` code of the Hero C# script with these bolded lines:

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
        print("Triggered: "+go.name);
    }
}
```

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

> Tip
>
> **ITERATIVE CODE DEVELOPMENT** When prototyping on your own, this kind of *console announcement test* is something that you will do often to test whether the code you've written is working properly. I find that doing small tests along the way like this is much better than

working on code for hours only to find at the end that something is causing a bug. Testing incrementally makes things *a lot* easier to debug because you know that you've only made slight changes since the last test that worked, so finding the place where you added a bug is easier.

Another key element of this approach is using the debugger. Throughout the authoring of this book, any time I ran into something that worked a little differently than I expected, I used the debugger to understand what was happening. If you don't remember how to use the MonoDevelop debugger, I highly recommend rereading Chapter 25, "Debugging."

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

Iterative code development also has the same strengths as the iterative process of design, and it is the key to the agile development methodology discussed in Chapter 14, "The Agile Mentality."

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

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    public float             shieldLevel = 1;
    // This variable holds a reference to the last triggering GameObject
    private GameObject       lastTriggerGo =
null;                            // a

    …

    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
```

```
        //print("Triggered:
"+go.name);                                          // b

        // Make sure it's not the same triggering go as last time
        if (go == lastTriggerGo)
{                                                    // c
            return;
        }
        lastTriggerGo =
go;                                                        // d

        if (go.tag == "Enemy") {  // If the shield was triggered by an
enemy
            shieldLevel--;        // Decrease the level of the shield by
1
            Destroy(go);          // … and Destroy the
enemy                    // e
        } else {
            print( "Triggered by non-Enemy:
"+go.name);                          // f
        }
    }
}
```

a. This private field will hold a reference to the last GameObject that triggered _Hero's collider. It is initially set to `null`.

b. Comment out this line here.

c. If `lastTriggerGo` is the same as `go` (the current triggering GameObject), this collision is ignored as a duplicate, and the function returns (i.e., exits). This can happen if two child GameObjects of the same Enemy both trigger the hero collider in the same single frame.

d. Assign `go` to `lastTriggerGo` so that it is updated before the next time `OnTriggerEnter()` is called.

e. `go`, the enemy GameObject, is destroyed by hitting the shield. Because the actual GameObject `go` that you're testing is the Enemy GameObject found by `transform.root`, this will delete the entire enemy (and by extension, all of its children), and not just one of the enemy's child GameObjects.

f. If _Hero collides with something that is not tagged "Enemy", then this will print to the Console and let you know what it is.

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

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

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

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

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    [Header("Set Dynamically")]
    [SerializeField]
    private float           _shieldLevel = 1; // Remember the underscore
    // This variable holds a reference to the last triggering GameObject
    …
}
```

8. Add the `shieldLevel` property to the end of the Hero class.

Click here to view code image

```
public class Hero : MonoBehaviour {
    …

    void OnTriggerEnter(Collider other) {
        …
    }

    public float shieldLevel {
        get {
            return( _shieldLevel
```

```
);                                          // a
        }
        set {
            _shieldLevel = Mathf.Min( value,4
);                          // b
            // If the shield is going to be set to less than zero
            if (value < 0)
{                                                   // c
                Destroy(this.gameObject);
            }
        }
    }
}
```

a. The `get` clause just returns the value of `_shieldLevel`.

b. `Mathf.Min()` ensures that `_shieldLevel` is never set to a number higher than 4.

c. If the value passed into the set clause is less than 0, _Hero is destroyed.

The `shieldLevel--;` line in `OnTriggerEnter()` uses both the get and set clauses of the `shieldLevel` property. First, it uses the get clause to determine the current value of `shieldLevel`, and then it subtracts 1 from that value and calls the set clause to assign that value back.

Restarting the Game

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

1. Add a `gameRestartDelay` field near the top of the Hero class:

Click here to view code image

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

    [Header("Set in Inspector")]
    …
    public float            pitchMult = 30;
    public float            gameRestartDelay = 2f;

    [Header("Set Dynamically")]
    …
```

```
}
```

2. Add the following lines to the `shieldLevel` property definition in the Hero class:

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    public float shieldLevel {
        get { … }
        set {
            …
            if (value < 0) {
                Destroy(this.gameObject);
                // Tell Main.S to restart the game after a delay
                Main.S.DelayedRestart( gameRestartDelay
);                  // a
            }
        }
    }
}
```

a. When you initially type `DelayedRestart()` into MonoDevelop, it appears red because the `DelayedRestart()` function does not yet exist in the Main class.

3. Add the following methods to the Main class to make the delayed restart work.

Click here to view code image

```
public class Main : MonoBehaviour {
    …

    public void SpawnEnemy() { … }

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

    public void Restart() {
        // Reload _Scene_0 to restart the game
        SceneManager.LoadScene( "_Scene_0");
    }
}
```

4. Click *Play* to test the game. Now, after the player ship has been destroyed, the game waits a couple of seconds and then restarts by reloading the scene.

> **Note**
>
> If your lighting looks strange after you've reloaded the scene (e.g., your ship and enemy ships look a bit darker), then you might be experiencing a known bug with Unity's lighting system (as I also mentioned in Chapter 28). Hopefully, Unity has now resolved this, but if you are seeing issues, an interim fix is available. Follow these directions to resolve it for this project:
>
> 1. From the menu bar, choose *Window* > *Lighting* > *Settings*.
> 2. Click the *Scene* button at the top of the Lighting pane.
> 3. Uncheck the *Auto Generate* selection at the bottom of the Lighting pane (next to the *Generate Lighting* button). This stops Unity from constantly recalculating the global illumination settings.
> 4. To make sure the lighting is built properly, click the *Generate Lighting* button at the bottom of the Lighting pane to manually calculate the global illumination.
> 5. Wait a few seconds for this to finish, and then click *Play* to test. You should see that the lighting is consistent even after reloading the scene. You should not have to recalculate the lighting again in this chapter, but if you do change lighting in the game, be sure to come back and manually recalculate.

Shooting (Finally)

Now that the enemy ships can hurt the player, it's time to give _Hero a way to fight back. This chapter only includes a single type and level of projectile. In the next chapter, you will do much more interesting things with the weapons in the game.

ProjectileHero, the Hero's Bullet

Follow these steps to create the Hero's bullet:

1. Create a cube named *ProjectileHero* in the Hierarchy with the following transform values:

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

2. Set both the Tag and Layer of ProjectileHero to *ProjectileHero*.
3. Create a new material named *Mat_Projectile*, place it in the _Materials folder

of the Project pane, give it the *ProtoTools* > *UnlitAlpha* shader, and assign it to the ProjectileHero GameObject.

4. Add a *Rigidbody* component to the ProjectileHero GameObject with these settings:
   - *Use Gravity* to false (unchecked)
   - *isKinematic* to false (unchecked)
   - *Collision Detection* to Continuous
   - *Constraints*: freeze Z position and X, Y, and Z rotation (by checking them)
5. In the *Box Collider* component of the ProjectileHero GameObject, set *Size.Z* to 10. This ensures that the projectile is able to hit anything that is slightly off of the XY (i.e., Z=0) plane.
6. Create a new C# script named *Projectile* and attach it to ProjectileHero. You'll edit the script later.

When you're finished with these steps, your settings should match those shown in Figure 30.9 (though you won't see the *BoundsCheck (Script)* component until you add it in step 8).

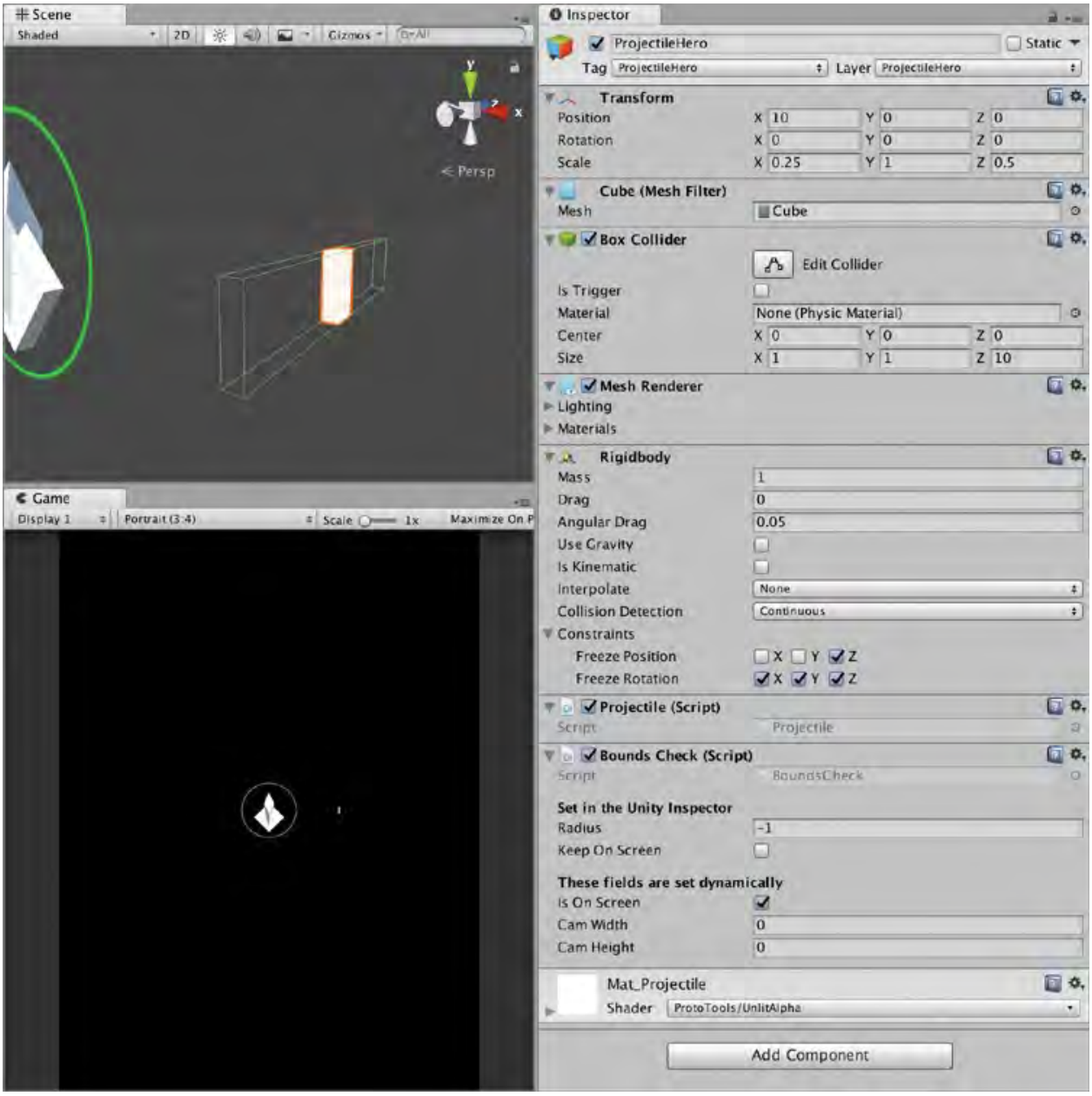

Figure 30.9 ProjectileHero with the proper settings showing the large Size.Z of the Box Collider

7. Save your scene.
8. Attach a *BoundsCheck* script component to ProjectileHero as well. Set `keepOnScreen` to `false` and `radius` to –1. The BoundsCheck `radius` will not affect collisions with other GameObjects; it only affects when the ProjectileHero thinks it has gone off screen.
9. Make ProjectileHero into a prefab by dragging it from the Hierarchy into the *_Prefabs folder* in the Project pane. Then delete the instance remaining in the

Hierarchy.

10. Save your scene—yes, save it again. As I've said, you want to save as often as you can.

Giving _Hero the Ability to Shoot

Now you add the capability for the Hero to shoot the bullet.

1. Open the Hero C# script and make the following bolded changes:

Click here to view code image

```
public class Hero : MonoBehaviour {
    …
    public float            gameRestartDelay = 2f;
    public GameObject       projectilePrefab;
    public float            projectileSpeed = 40;

    …

    void Update () {
        …
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,
xAxis*rollMult,0);

        // Allow the ship to fire
        if ( Input.GetKeyDown( KeyCode.Space ) )
{                           // a
            TempFire();
        }
    }

    void TempFire()
{                                                       // b
        GameObject projGO = Instantiate<GameObject>( projectilePrefab );
        projGO.transform.position = transform.position;
        Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
        rigidB.velocity = Vector3.up * projectileSpeed;
    }

    void OnTriggerEnter(Collider other) { … }
    …
}
```

   a. This causes the ship to fire every time the space bar is pressed.

   b. This method is named `TempFire()` because you will be replacing it in the next chapter.

2. In Unity, select *_Hero* in the Hierarchy and assign *ProjectileHero* from the

Project pane to the `projectilePrefab` of the Hero script.

3. Save and click *Play*. Now, you can fire projectiles by pressing the space bar, but they don't yet destroy enemy ships, and they continue forever when they go off screen.

Scripting the Projectile

To script the projectile, follow these steps:

1. Open the *Projectile* C# script and make the following bolded changes. All you need the Projectile to do is destroy itself when it goes off screen. You'll add more in the next chapter.

Click here to view code image

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

public class Projectile : MonoBehaviour {

    private BoundsCheck     bndCheck;

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
    }
    void Update () {
        if (bndCheck.offUp) {                                    // a
            Destroy( gameObject );
        }
    }
}
```

   a. If the Projectile goes off the top of the screen, destroy it.

2. Of course, remember to save.

Allowing Projectiles to Destroy Enemies

You also need the capability of destroying enemies with the bullets.

1. Open the Enemy C# script and add the following method to the end of the script:

Click here to view code image

```
public class Enemy : MonoBehaviour {
    …
    public virtual void Move() { … }
```

```
    void OnCollisionEnter( Collision coll ) {
      GameObject otherGO =
coll.gameObject;                                    // a
      if ( otherGO.tag == "ProjectileHero")
{                                  // b
        Destroy( otherGO );         // Destroy the Projectile
        Destroy( gameObject );      // Destroy this Enemy GameObject
        } else {
            print( "Enemy hit by non-ProjectileHero: " + otherGO.name
);     // c
        }
    }
}
```

a. Get the GameObject of the Collider that was hit in the Collision.

b. If `otherGO` has the ProjectileHero tag, then destroy it and this Enemy instance.

c. If `otherGO` doesn't have the ProjectileHero tag, print the name of what was hit to the Console for debugging purposes. If you want to test this, you can temporarily remove the ProjectileHero tag from the ProjectileHero prefab and shoot an Enemy.[4]

Now, when you click *Play*, Enemy_0s will come down the screen, and you can shoot them with projectiles. That's it for this chapter—you have a nice, simple prototype—but the next chapter, expands on it considerably by showing you how to add additional enemies, three kinds of power-ups, and two additional kinds of guns. It also offers some more interesting coding tricks.

Summary

In most chapters, I include a *next steps* section here to give you ideas of what you can do to extend the project and push yourself. However, for the *Space SHMUP* prototype, you're going to do some of those things in the next chapter and see some new coding concepts in the process. Take a break now so you can approach the next chapter fresh and congratulate yourself on a prototype well done.

1. *Juiciness*, as a term that relates to gameplay, was coined in 2005 by Kyle Gabler and the other members of the Experimental Gameplay Project at Carnegie Mellon University's Entertainment Technology Center. To them, a juicy element had "constant and bountiful user feedback." You can read about it more in their Gamasutra article by searching online for "Gamasutra How to

Prototype a Game in Under 7 Days."

2. The first edition of this book had a much more complex system for keeping GameObjects on screen that was more than was needed for this chapter and somewhat confusing. I've replaced it with this version in the second edition to both streamline the chapter and to reinforce the concept of components.
3. Search online for "C# XML documentation" to learn more.
4. You can't test this by running the _Hero into an Enemy because the collider on the Shield child of _Hero is a trigger, and triggers will not invoke a call to OnCollisionEnter().