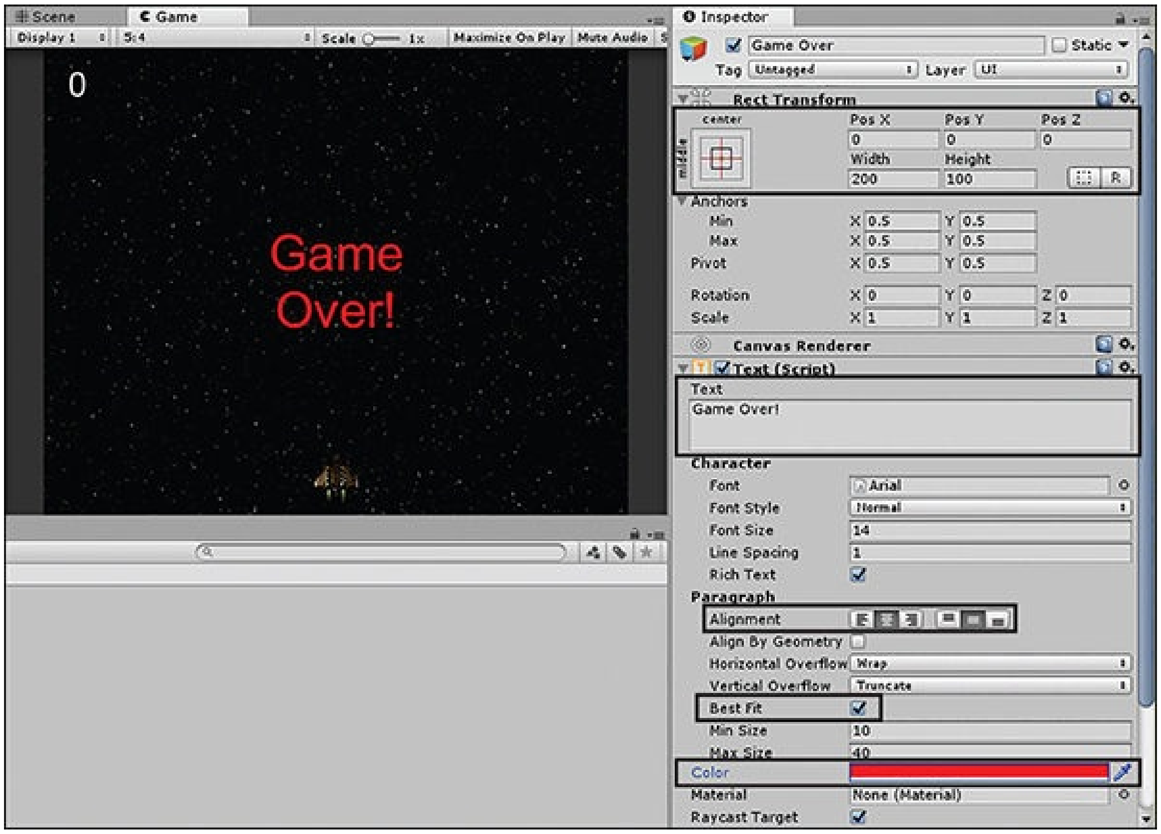


```
Game Over                         Static
Tag  Untagged          Layer  UI
Rect Transform
center        Pos X     Pos Y     Pos Z
middle        0         0         0
              Width     Height
              200       100                R
Anchors
  Min         X 0.5     Y 0.5
  Max         X 0.5     Y 0.5
Pivot         X 0.5     Y 0.5
Rotation      X 0       Y 0       Z 0
Scale         X 1       Y 1       Z 1
Canvas Renderer
Text (Script)
Text
Game Over!
Character
  Font                 Arial
  Font Style           Normal
  Font Size            14
  Line Spacing         1
  Rich Text            ✓
Paragraph
  Alignment
  Align By Geometry
  Horizontal Overflow  Wrap
  Vertical Overflow    Truncate
  Best Fit             ✓
  Min Size             10
  Max Size             40
Color                  🟥
Material               None (Material)
Raycast Target         ✓
```

FIGURE 15.7
The Game Over! sign settings.

Later you will connect this score display to your GameManager script so that it can be updated. Now all your entities are in place, and it is time to begin turning this scene into a game.

Controls

Various script components need to be assembled to make this game work. The player needs to be able to move the ship and shoot bullets. The bullets and meteors need to be able to move automatically. A meteor spawn object must keep the meteors flowing. The shredders need to be able to clean up objects, and a manager needs to keep track of all the action.

The Game Manager

The game manager is basic in this game, and you can add it first. To create the game manager, follow these steps:

1. Create an empty game object and name it **GameManager**.

2. Because the only asset you have for the game manager is a script, create a new folder called Scripts so that you have a place for any simple scripts you create.

3. Create a new script named **GameManager** in the Scripts folder and attach it to the GameManager game object. Overwrite the contents of the script with the following code:

Click here to view code image

```
using UnityEngine;
using UnityEngine.UI; // Note this new line is needed for UI

public class GameManager : MonoBehaviour
{
    public Text scoreText;
    public Text gameOverText;

    int playerScore = 0;

    public void AddScore()
    {
        playerScore++;
        // This converts the score (a number) into a string
        scoreText.text = playerScore.ToString();
    }

    public void PlayerDied()
    {
        gameOverText.enabled = true;
        // This freezes the game
        Time.timeScale = 0;
    }
}
```

In this code, you can see that the manager is responsible for keeping the score and knowing when the game is running. The manager has two public methods: `PlayerDied()` and `AddScore()`. `PlayerDied()` is called by the player when a meteor hits it. `AddScore()` is called by a bullet when it kills a meteor.

Remember to drag the Score and Game Over elements onto the GameManager script (see Figure 15.8).

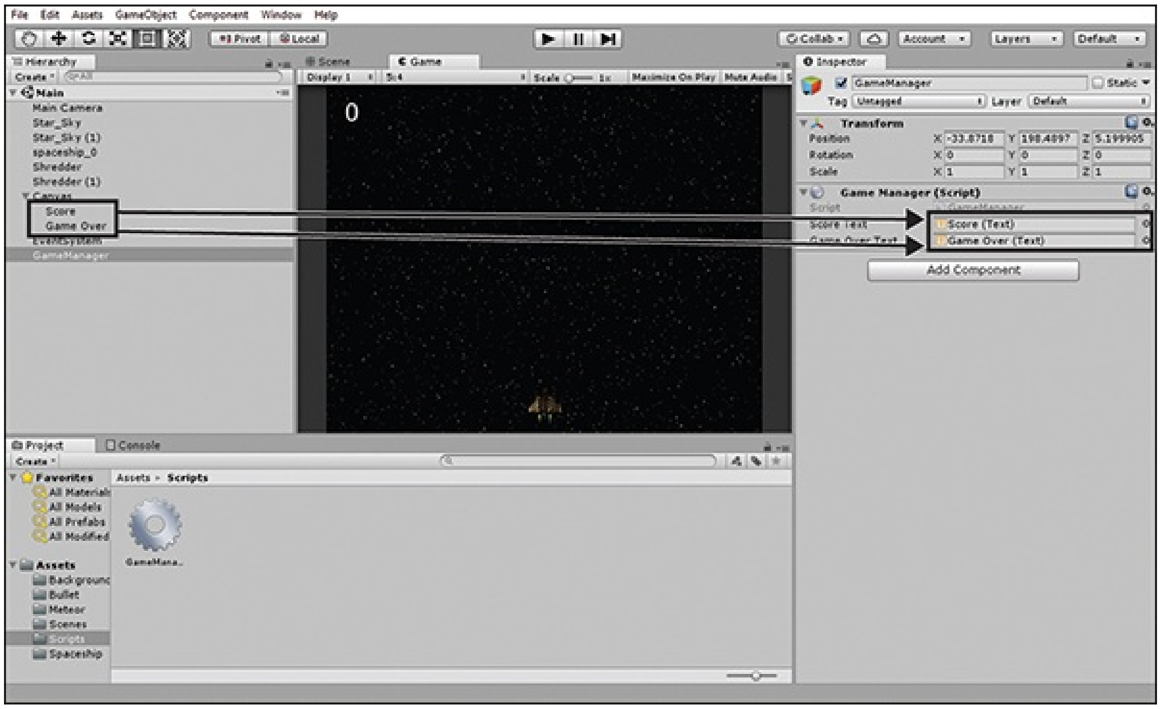


FIGURE 15.8
Attaching the text elements to the game manager.

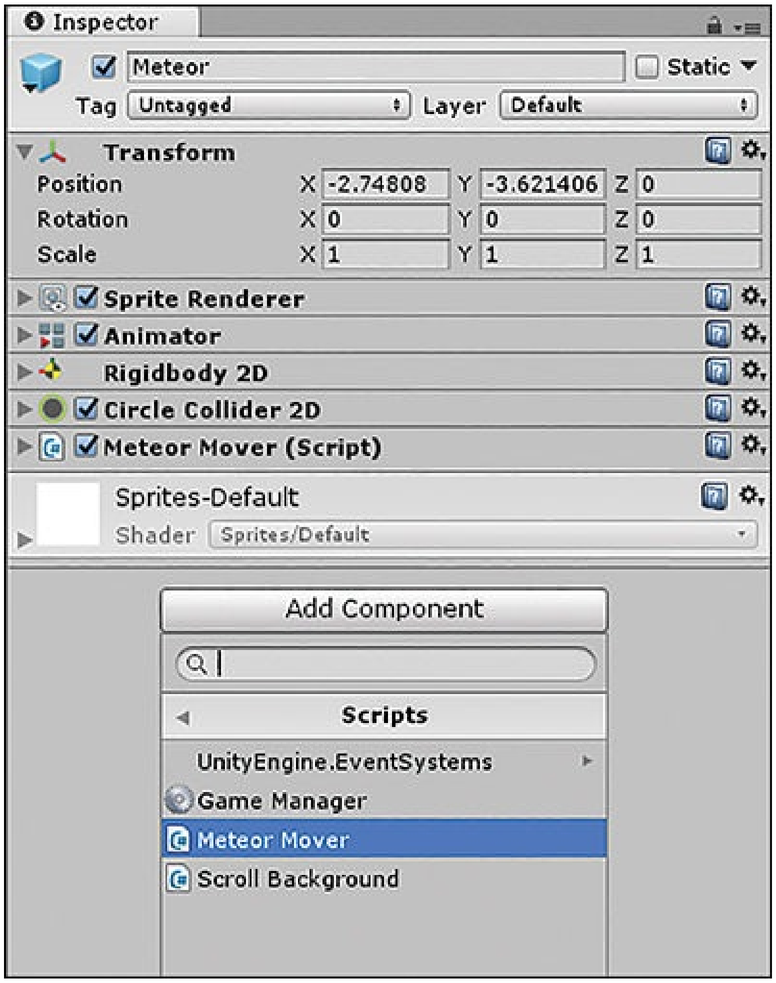
The Meteor Script

Meteors are basically going to fall from the top of the screen and get in the player’s way. To create the meteor script:

1. Create a new script in your Meteor folder and call it **MeteorMover**.
2. Select your meteor prefab. In the Inspector view, locate the Add Component button (see Figure 15.9) and select **Add Component > Scripts > MeteorMover**.
3. Overwrite the code in the MeteorMover script with the following:

Click here to view code image

```
using UnityEngine;

public class MeteorMover : MonoBehaviour
{
    public float speed = -2f;

    Rigidbody2D rigidBody;

    void Start()
    {
        rigidBody = GetComponent<Rigidbody2D>();
        // Give meteor an initial downward velocity
        rigidBody.velocity = new Vector2(0, speed);
    }
}
```



Meteor
Tag Untagged
Layer Default

	X	Y	Z
Position	-2.74808	-3.621406	0
Rotation	0	0	0
Scale	1	1	1

Sprites-Default
Shader Sprites/Default

FIGURE 15.9
Adding the Meteor script to the meteor prefab.

The meteor is very basic and contains only a single public variable to represent its downward velocity. In the `Start()` method, you get a reference to the Rigidbody 2D component on the meteor. You then use that rigidbody to set the velocity of the meteor; it moves downward because the speed has a negative value. Notice that the meteor is not responsible for determining collision.

Feel free to drag the meteor prefab into the Hierarchy view and click **Play** so you can see the effect of the code you just wrote. The meteor should slide down and spin slightly. If you do this, be sure to delete the Meteor instance from the

Hierarchy view when you’re done.

The Meteor Spawn

So far, the meteors are just prefabs with no way of getting into the scene. You need an object to be responsible for spawning the meteors at an interval. Create a new empty game object, rename it **Meteor Spawn**, and place it at (0, 8, 0). Create a new script named MeteorSpawn in your Meteor folder and place it on the Meteor Spawn object. Overwrite the code in the script with the following:

Click here to view code image

```
using UnityEngine;

public class MeteorSpawn : MonoBehaviour
{
    public GameObject meteorPrefab;
    public float minSpawnDelay = 1f;
    public float maxSpawnDelay = 3f;
    public float spawnXLimit = 6f;

    void Start()
    {
        Spawn();
    }

    void Spawn()
    {
        // Create a meteor at a random x position
        float random = Random.Range(-spawnXLimit, spawnXLimit);
        Vector3 spawnPos = transform.position + new Vector3(random, 0f, 
        Instantiate(meteorPrefab, spawnPos, Quaternion.identity);

        Invoke("Spawn", Random.Range(minSpawnDelay, maxSpawnDelay));
    }
}
```

This script is doing a few interesting things. First, it creates two variables to manage the meteor timing. It also declares a `GameObject` variable, which will be the meteor prefab. In the `Start()` method you call the function `Spawn()`. This function is responsible for creating and placing the meteors.

You can see that the meteor is spawned at the same y and z coordinate as the spawn point, but the x coordinate is offset by a number between -6 and 6. This allows the meteors to spawn across the screen and not always in the same spot. When the position for the new meteor is determined, the `Spawn()` function instantiates (creates) a meteor at that position with default rotation (`Quaternion.identity`). The last line invokes a call to the spawn function again. This method, `Invoke()`, calls the named function (in this case `Spawn()`) after a random amount of time. That random amount is controlled by the two timing variables.

In the Unity editor, click and drag your meteor prefab from the Project view onto the Meteor Prefab property of the Meteor Spawn Script component of the Meteor Spawn object. (Try saying that fast!) Run the scene, and you should see meteors spawning across the screen. (Attack, my minions!)

The DestroyOnTrigger Script

Now that you have meteors spawning everywhere, it is a good idea to begin cleaning them up. Create a new script called DestroyOnTrigger in your Scripts folder (since it is a single unrelated asset) and attach it to both the upper and the lower shredder objects you created previously. Add the following code to the script, ensuring that this code is outside a method but inside the class:

Click here to view code image

```
void OnTriggerEnter2D(Collider2D other)
{
   Destroy(other.gameObject);
}
```

This basic script simply destroys any object that enters it. Because the players cannot move vertically, you don’t need to worry about them getting destroyed. Only bullets and meteors can enter the triggers.

The ShipControl Script

Right now, meteors are falling down, and the player can’t get out of the way. You need to create a script to control the player next. Create a new script called ShipControl in your Spaceship folder and attach it to the spaceship object in your scene. Replace the code in the script with the following:

Click here to view code image

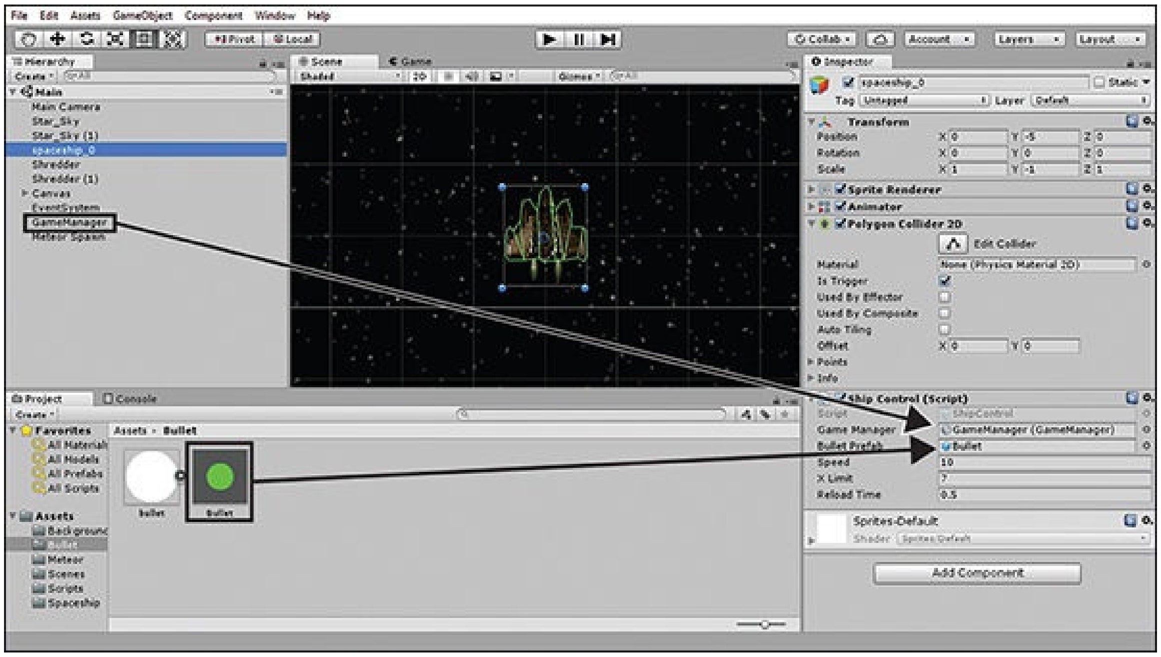
```
using UnityEngine;

public class ShipControl : MonoBehaviour
{
    public GameManager gameManager;
    public GameObject bulletPrefab;
    public float speed = 10f;
    public float xLimit = 7f;
    public float reloadTime = 0.5f;

    float elapsedTime = 0f;

    void Update()
    {
        // Keeping track of time for bullet firing
        elapsedTime += Time.deltaTime;

        // Move the player left and right
        float xInput = Input.GetAxis("Horizontal");
        transform.Translate(xInput * speed * Time.deltaTime, 0f, 0f);

        // Clamp the ship's x position
        Vector3 position = transform.position;
        position.x = Mathf.Clamp(position.x, -xLimit, xLimit);
        transform.position = position;

        // Spacebar fires. The default InputManager settings call this "
        // Only happens if enough time has elapsed since last firing. if
        {
            // Instantiate the bullet 1.2 units in front of the player
            Vector3 spawnPos = transform.position;
            spawnPos += new Vector3(0, 1.2f, 0);
            Instantiate(bulletPrefab, spawnPos, Quaternion.identity);

            elapsedTime = 0f; // Reset bullet firing timer
        }
    }

    // If a meteor hits the player
    void OnTriggerEnter2D(Collider2D other)
    {
        gameManager.PlayerDied();
    }
}
```

This script does a lot of work. It starts by creating variables for the game manager, the bullet prefab, speed, movement limitations, and bullet timing.

In the `Update()` method, the script starts by getting the elapsed time. This is used to determine whether enough time has passed to fire a bullet. Remember that according to the rules, the player can fire a bullet only every half a second. The player is then moved along the x axis based on input. The player’s x axis position is *clamped* (that is, limited) so that the player cannot go off the screen to the left or right. After that, the script determines whether the player is pressing the spacebar. Normally in Unity, the spacebar is used for a jump action. (This could be renamed in the input manager, but it was left as it is to avoid any confusion.) If it is determined that the player is pressing the spacebar, the script checks the elapsed time against the `reloadTime` (currently half a second). If the time is greater, the script creates a bullet. Notice that the script creates the bullet just a little above the ship. This prevents the bullet from colliding with the ship. Finally, the elapsed time is reset to 0 so the count for the next bullet firing can start.

The last part of the script contains the `OnTriggerEnter2D()` method. This method is called whenever a meteor hits the player. When that happens, the GameManager script is informed that the player died.

Back in the Unity editor, click and drag the bullet prefab onto the Bullet property of the Ship Control component on the spaceship. Likewise, click and drag the Game Manager object onto the Ship Control component to give it access to the GameManager script (see Figure 15.10). Run the scene and notice that you can now move the player. The player should be able to fire bullets (although they don't move). Also notice that the player can now die and end the game.



spaceship_0 Static
Tag Untagged Layer Default
Transform
Position X 0 Y -5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y -1 Z 1
Sprite Renderer
Animator
Polygon Collider 2D
Edit Collider
Material None (Physics Material 2D)
Is Trigger ✓
Used By Effector
Used By Composite
Auto Tiling
Offset X 0 Y 0
Points
Info
Ship Control (Script)
Script ShipControl
Game Manager GameManager (GameManager)
Bullet Prefab Bullet
Speed 10
X Limit 7
Reload Time 0.5
Sprites-Default
Shader Sprites/Default

FIGURE 15.10
Connecting the ShipControl script.

The Bullet Script

The last bit of interactivity you need is to make the bullets move and collide

The last bit of interactivity you need is to make the bullets move and collide. Create a new script called Bullet in your Bullet folder and add it to the bullet prefab. Replace the code in the script with the following:

Click here to view code image

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    public float speed = 10f;

    GameManager gameManager; // Note this is private this time

    void Start()
    {
        // Because the bullet doesn't exist until the game is running
        // we must find the Game Manager a different way.
        gameManager = GameObject.FindObjectOfType<GameManager>();

        Rigidbody2D rigidBody = GetComponent<Rigidbody2D>();
        rigidBody.velocity = new Vector2(0f, speed);
    }

    void OnCollisionEnter2D(Collision2D other)
    {
        Destroy(other.gameObject); // Destroy the meteor gameManager.AddS
        Destroy(gameObject); // Destroy the bullet
    }
}
```

The major difference between this script and the meteor script is that this script needs to account for collision and the player scoring. The Bullet script declares a variable to hold a reference to the GameManager script, just as the ShipControl script does. Because the bullet isn’t actually in the Scene view, however, it needs to locate the GameManager script a little differently. In the `Start()` method, the script searches for the GameControl object by type using the `GameObject.FindObjectOfType<Type>()` method. The reference to the GameManager script is then stored in the variable `gameManager`. It is worth noting that Unity’s `Find()` methods (such as the one used here) are very slow and should be used sparingly.

Because neither the bullet nor the meteor has a trigger collider on it, the use of the `OnTriggerEnter2D()` method will not work. Instead, the script uses the method `OnCollisionEnter2D()`. This method does not read in a `Collider2D` variable. Instead, it reads in a `Collision2D` variable. The differences between these two methods are irrelevant in this case. The only work being done is destroying both objects and telling the GameManager script that the player scored.

Go ahead and run the game. You should find that the game is now fully playable. Although you cannot win (that is intentional), you certainly can lose. Keep playing and see what kind of score you can get!

As a fun challenge, consider trying to make the values of `minSpawnDelay` and `maxSpawnDelay` get smaller over time, causing the meteors to spawn more quickly as the game is played.

Improvements

It is time to improve the game. As with the games you have created in earlier hours, several parts of *Captain Blaster* are left intentionally basic. Be sure to play through the game several times and see what you notice. What things are fun? What things are not fun? Are there any obvious ways to break the game? Note that a very easy cheat has been left in the game to allow players to get a high score. Can you find it?

Here are some things you could consider changing:

- Try modifying the bullet speeds, firing delay, or bullet flight path.
- Try allowing the player to fire two bullets side by side.
- Make the meteors spawn faster as time goes on.
- Try adding a different type of meteor.
- Give the player extra health—and maybe even a shield.
- Allow the player to move vertically as well as horizontally.

This is a common genre, and there are many ways you can make it unique. Try to see just how custom you can make the game. It is also worth noting that you will learn about particle systems in Hour 16, “Particle Systems,” and this game is a prime candidate for trying them out.

Summary

In this hour, you made the game *Captain Blaster*. You started by designing the game elements. Next, you built the game world. You constructed and animated a vertically scrolling background. From there, you built the various game entities. You added interactivity through scripting and controls. Finally, you examined the game and looked for improvements.

Q&A

Q. Did *Captain Blaster* really achieve the military rank of captain, or is it just a name?

A. It’s hard to say, as it is all mostly speculation. One thing is for certain: They don’t give spaceships to mere lieutenants!

Q. Why delay bullet firing by half a second?

A. Mostly it is a balance issue. If the player can fire too fast, the game has no challenge.

Q. Why use a polygon collider on the ship?

A. Because the ship has an odd size, a standard shaped collider wouldn’t be very accurate. Luckily, you can use the polygon collider to map closely to the geometry of the ship.

Workshop

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

Quiz

1. What is the win condition for the game?
2. How does the scrolling background work?
3. Which objects have rigidbodies? Which objects have colliders?
4. True or False: The meteor is responsible for detecting collision with the player.

Answers

1. This is a trick question. The player cannot win the game. The highest score, however, allows the player to “win” outside the game.
2. Two identical sprites are positioned, one above the other, on the y axis. They then leapfrog across the camera to seem endless.
3. The bullets and meteors have rigidbodies. The bullets, meteors, ship, and triggers have colliders.

4. False. The ShipControl script detects the collision.

Exercise

This exercise is a little different from the ones you have done so far. A common part of the game refinement process is to have a game playtested by people who aren’t involved with the development process. This allows people who are completely unfamiliar with the game to give honest, first-experience feedback, which is incredibly useful. For this exercise, have other people play the game. Try to get a diverse group of people—some avid gamers and some people who don’t play games, some people who are fans of this genre and some people who aren’t. Compile their feedback into groupings of good features, bad features, and things that can be improved. In addition, try to see whether there are any commonly requested features that currently aren’t in the game. Finally, see if you can implement or improve your game based on the feedback received.