Pro HTML5 Games

Learn to Build your Own Games
using HTML5 and JavaScript

—

Second Edition

—

Aditya Ravi Shankar

APRESS®

Pro HTML5 Games

Learn to Build your Own Games using
HTML5 and JavaScript

Second Edition

Aditya Ravi Shankar

Apress®

Contents at a Glance

Contents

v

About the Author

Aditya Ravi Shankar started programming in 1993 when he was first introduced to the world of computers. With no access to the Internet or online tutorials at the time, he wrote his first game in GW-BASIC by painstakingly retyping code from a book he found at the local library.

After graduating from the Indian Institute of Technology - Madras in 2001, Aditya spent nearly a decade working as a software consultant, developing trading and analytics systems for investment banks and large Fortune 100 companies, before eventually leaving his corporate life behind so he could focus on doing what he loved.

A self-confessed technology geek, Aditya has spent the time since then working on his own projects and experimenting, with every new language and technology that he could, including of course HTML5. During this time, he became well known for re-creating several classic games in HTML5, including the real-time strategy game *Command and Conquer* and the tactical game *Commandos: Behind Enemy Lines*. He has also worked as a consultant to develop a large variety of HTML5 games, including endless runner games, racing games, base-defense games, arcade games, puzzle games, educational games, and different types of multiplayer games.

Apart from programming, Aditya is passionate about billiards, salsa dancing, and personal development. He maintains a personal website (`http://www.adityaravishankar.com`) where he writes articles on game programming, personal development, and billiards.

About the Technical Reviewer

Gaurav Mishra is an expert in user interface development and UX design with more than 10 years of experience. He provides workshops and training in UI development, UX design, and Drupal. Gaurav has played a key role in the success of many organizations and likes to build products and services from scratch. Gaurav lives in New Delhi, India, and likes to spend his leisure time with his baby Yuvika and wife Neeti. He likes all genres of music, from Indian classical to club music. Gaurav can be reached at mr.gauravmishr@gmail.com and also tweets at @gauravmishr.

CHAPTER 1

■ ■ ■ ■

HTML5 and JavaScript Essentials

HTML5, the latest version of the HTML standard, provides us with many new features for improved interactivity and media support. These new features (such as canvas, audio, and video) have made it possible to make fairly rich and interactive applications for the browser without requiring third-party plug-ins such as Flash.

Even though the HTML5 standard continues to grow and improve as a "living standard," all the elements that we need for building some very amazing games are already supported by all modern browsers (Google Chrome, Mozilla Firefox, Internet Explorer 9+, Microsoft Edge, Safari, and Opera).

Over the past half-decade (since I wrote the first edition of this book), HTML5 support has become a standard across all modern browsers, both desktop and mobile. This means we now can make games in HTML5 that can be easily extended to work on both mobile and desktop across a wide variety of operating systems.

All you need to get started on developing your games in HTML5 are a good text editor to write your code (I currently use Visual Studio Code on both Mac and PC—https://code.visualstudio.com/) and a modern, HTML5-compatible browser (I primarily use Google Chrome). Once you have installed your preferred text editor and HTML5-compatible browser, you are ready to create your first HTML5 page.

A Basic HTML5 Page

The structure of an HTML5 document is very similar to the structure in previous versions, except that HTML5 has a much simpler DOCTYPE tag at the beginning of the document. This simpler DOCTYPE tag lets the browser know that it needs to use the latest standards when interpreting the document.

Listing 1-1 provides a skeleton for a very basic HTML5 file that we will be using as a starting point for the rest of this chapter. Executing this code involves saving it as an HTML file and then opening the file in a web browser. If you do everything correctly, the browser should pop up the message "Hello World!"

Listing 1-1. Basic HTML5 File Skeleton

```html
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">
        <title>Sample HTML5 File</title>
        <script type="text/javascript">
            // This function will be called once the page loads completely
            function pageLoaded(){
                alert("Hello World!");
            }
```

```
        </script>
    </head>
    <body onload="pageLoaded();">

    </body>
</html>
```

■ **Note**   We use the body's `onload` event to call our `pageLoaded()` function so that we can be sure that our page has completely loaded before we start working with it. This will become important when we start manipulating elements like images and audio. Trying to access these elements before the browser has finished loading them will cause JavaScript errors or other unexpected behavior.

Before we start developing games, we need to go over some of the basic building blocks that we will be using to create our games. The most important ones that we need are

- The `canvas` element, to render shapes and images

- The `audio` element, to add sounds and background music

- The `image` element, to load our game artwork and display it on the canvas

- The browser timer functions, and game loops to handle animation

The canvas Element

The most important element for use in our games is the new `canvas` element. As per the HTML5 standard specification, "The `canvas` element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly." You can find the complete specification at https://html.spec.whatwg.org/multipage/scripting.html#the-canvas-element.

The canvas allows us to draw primitive shapes like lines, circles, and rectangles, as well as images and text, and has been optimized for fast drawing. Browsers have started enabling GPU-accelerated rendering of 2D canvas content, so that canvas-based games and animations run fast.

Using the `canvas` element is fairly simple. Place the `<canvas>` tag inside the body of the HTML5 file we created earlier, as shown in Listing 1-2.

Listing 1-2.   Creating a Canvas Element

```
<body onload="pageLoaded();">
    <canvas width="640" height="480" id="testcanvas" style="border: 1px solid black;">
        Your browser does not support HTML5 Canvas. Please shift to a newer browser.
    </canvas>
</body>
```

The code in Listing 1-2 creates a canvas that is 640 pixels wide and 480 pixels high. By itself, the canvas shows up as a blank area (with a black border that we specified in the style). We can now start drawing inside this rectangle using JavaScript.

■ **Note** Browsers that do not support canvas will ignore the <canvas> tag and render anything inside the <canvas> tag. You can use this feature to show users on older browsers alternative fallback content or a message directing them to a more modern browser.

We draw on the canvas using what is known as its primary rendering context. We can access this context with the getContext() method of the canvas object. The getContext() method takes one parameter: the type of context that we need. We will be using the 2d context for our games.

Listing 1-3 shows how we can access the canvas and its context once the page has loaded by modifying the pageLoaded() method.

Listing 1-3. Accessing the Canvas Context

```
<script type="text/javascript">
    function pageLoaded(){

        // Get a handle to the canvas object
        var canvas = document.getElementById("testcanvas");

        // Get the 2d context for this canvas
        var context = canvas.getContext("2d");

        // Our drawing code here...
    }
</script>
```

■ **Note** All browsers support the 2d context that we need for 2D graphics. Most browsers also implement other contexts with names such as webgl or experimental-webgl for 3D graphics.

This code doesn't seem to do anything yet. However, we now have access to a 2d context object. This context object provides us with a large number of methods that we can use to draw our game elements on the screen. This includes methods for the following:

- Drawing rectangles
- Drawing complex paths (lines, arcs, and so forth)
- Drawing text
- Customizing drawing styles (colors, alpha, textures, and so forth)
- Drawing images
- Transforming and rotating

We will look at each of these methods in more detail in the following sections.

Drawing Rectangles

Before you can start drawing on the canvas, you need to understand how to reference coordinates on it. The canvas uses a coordinate system with the origin (0, 0) at the top-left corner of the canvas, x increasing toward the right, and y increasing downward, as illustrated in Figure 1-1.
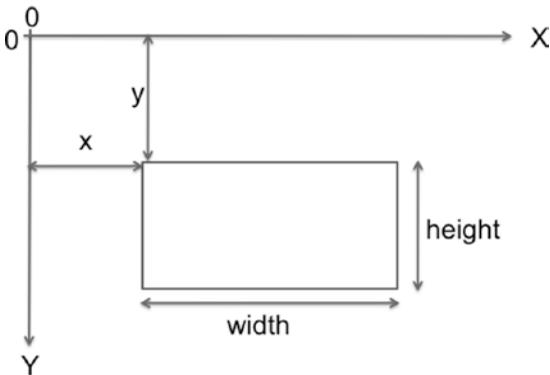
Figure 1-1. *Coordinate system for canvas*

We can draw a rectangle on the canvas using the context's rectangle methods:

- `fillRect(x, y, width, height)`: Draws a filled rectangle

- `strokeRect(x, y, width, height)`: Draws a rectangular outline

- `clearRect(x, y, width, height)`: Clears the specified rectangular area and makes it fully transparent

Listing 1-4. Drawing Rectangles Inside the Canvas

```
// FILLED RECTANGLES
// Draw a solid square with width and height of 100 pixels at (200,10)
context.fillRect(200, 10, 100, 100);
// Draw a solid square with width of 90 pixels and height of 30 pixels at (50,70)
context.fillRect(50, 70, 90, 30);

// STROKED RECTANGLES
// Draw a rectangular outline with width and height of 50 pixels at (110, 10)
context.strokeRect(110, 10, 50, 50);
// Draw a rectangular outline with width and height of 50 pixels at (30, 10)
context.strokeRect(30, 10, 50, 50);

// CLEARING RECTANGLES
// Clear a rectangle with width of 30 pixels and height of 20 pixels at (210, 20)
context.clearRect(210, 20, 30, 20);
// Clear a rectangle with width of 30 pixels and height of 20 pixels at (260, 20)
context.clearRect(260, 20, 30, 20);
```

The code in Listing 1-4 will draw multiple rectangles on the top-left corner of the canvas, as shown in Figure 1-2. Add the code to the bottom of the pageLoaded() method, save the file, and refresh the browser to see the result of these changes.
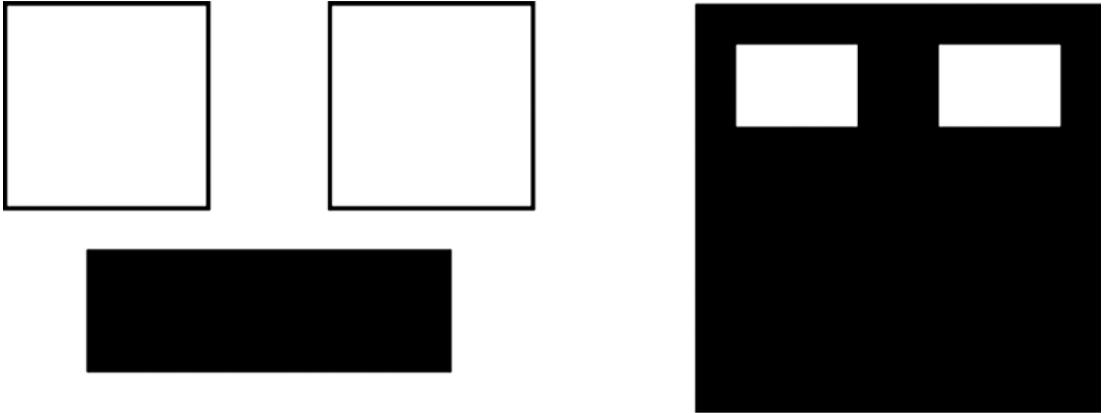
Figure 1-2. *Drawing rectangles inside the canvas*

Drawing Complex Paths

The context has several methods that allow us to draw complex shapes when simple boxes aren't enough:

- beginPath(): Starts recording a new shape
- closePath(): Closes the path by drawing a line from the current drawing point to the starting point
- fill(), stroke(): Fills or draws an outline of the recorded shape
- moveTo(x, y): Moves the drawing point to x, y
- lineTo(x, y): Draws a line from the current drawing point to x, y
- arc(x, y, radius, startAngle, endAngle, anticlockwise): Draws an arc at x, y with specified radius

Using these methods, drawing a complex path involves the following steps:

1. Use beginPath() to start recording the new shape.
2. Use moveTo(), lineTo(), and arc() to create the shape.
3. Optionally, close the shape using closePath().
4. Use either stroke() or fill() to draw an outline or filled shape. Using fill() automatically closes any open paths.

Listing 1-5 will create the triangles, arcs, and shapes shown in Figure 1-3.

Listing 1-5. Drawing Complex Shapes Inside the Canvas

```
// DRAWING COMPLEX SHAPES
// Draw a filled triangle
context.beginPath();
context.moveTo(10, 120);     // Start drawing at 10, 120
context.lineTo(10, 180);
context.lineTo(110, 150);
context.fill();              // Close the shape and fill it out

// Draw a stroked triangle
context.beginPath();
context.moveTo(140, 160); // Start drawing at 140, 160
context.lineTo(140, 220);
context.lineTo(40, 190);
context.closePath();
context.stroke();

// Draw a more complex set of lines
context.beginPath();
context.moveTo(160, 160); // Start drawing at 160, 160
context.lineTo(170, 220);
context.lineTo(240, 210);
context.lineTo(260, 170);
context.lineTo(190, 140);
context.closePath();
context.stroke();

// DRAWING ARCS & CIRCLES
// Draw a semicircle
context.beginPath();
// Draw an arc at (400, 50) with radius 40 from 0 to 180 degrees, anticlockwise
// PI radians = 180 degrees
context.arc(100, 300, 40, 0, Math.PI, true);
context.stroke();

// Draw a full circle
context.beginPath();
// Draw an arc at (500, 50) with radius 30 from 0 to 360 degrees, anticlockwise
// 2*PI radians = 360 degrees
context.arc(100, 300, 30, 0, 2 * Math.PI, true);
context.fill();

// Draw a three-quarter arc
context.beginPath();
// Draw an arc at (400, 100) with radius 25 from 0 to 270 degrees, clockwise
// (3/2*PI radians = 270 degrees)
context.arc(200, 300, 25, 0, 3 / 2 * Math.PI, false);
context.stroke();
```
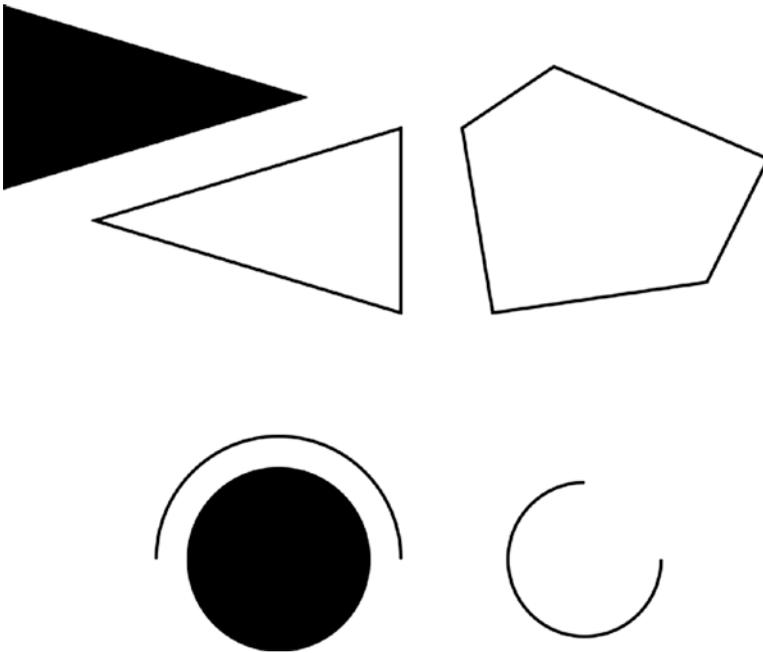
Figure 1-3. *Drawing complex shapes inside the canvas*

Drawing Text

The context also provides us with two methods for drawing text on the canvas:

- strokeText(text, x, y): Draws an outline of the text at (x, y)

- fillText(text, x, y): Fills out the text at (x, y)

Unlike text inside other HTML elements, text inside canvas does not have CSS layout options such as wrapping, padding, and margins. However, the text output can be modified by setting the context font, stroke, and fill style properties, as shown in Listing 1-6.

Listing 1-6. Drawing Text Inside the Canvas

```
// DRAWING TEXT
context.fillText("This is some text...", 330, 40);

// Modify the font
context.font = "10pt Arial";
context.fillText("This is in 10pt Arial...", 330, 60);

// Draw stroked text
context.font = "16pt Arial";
context.strokeText("This is stroked in 16pt Arial...", 330, 80);
```

The code in Listing 1-6 will draw the text shown in Figure 1-4.

This is some text...

This is in 10pt Arial...

This is stroked in 16pt Arial...

Figure 1-4. *Drawing text inside the canvas*

When setting the font property, you can use any valid CSS font property. As you can see from the previous example, while you may not have the same degree of flexibility in formatting that HTML and CSS provide, you can still do a lot with the canvas text methods. Of course, this would look a lot better if we could add some color.

Customizing Drawing Styles (Colors and Textures)

So far, everything we have drawn has been in black, but only because the canvas default drawing color is black. We have other options. We can style and customize the lines, shapes, and text on a canvas. We can draw using different colors, line styles, transparencies, and even fill textures inside the shapes.

If we want to apply colors to a shape, there are two important properties we can use:

- fillStyle: Sets the default color for all future fill operations

- strokeStyle: Sets the default color for all future stroke operations

Both properties can take valid CSS colors as values. This includes rgb() and rgba() values as well as color constant values. For example, context.fillStyle = "red"; will define the fill color as red for all future fill operations (fillRect, fillText, and fill).

In addition, the context object's createTexture() method creates a texture from an image, which can also be used as a fill style. Before we can use an image, we need to load the image into the browser. For now, we will just add an <img> tag after the <canvas> tag in our HTML file:

```
<img src="fire.png" id="fire">
```

The code in Listing 1-7 will draw colored and textured rectangles, as shown in Figure 1-5.

Listing 1-7. Drawing with Colors and Textures

```
// FILL STYLES AND COLORS
// Set fill color to red
context.fillStyle = "red";
// Draw a red filled rectangle
context.fillRect(310, 160, 100, 50);

// Set stroke color to green
context.strokeStyle = "green";
// Draw a green stroked rectangle
context.strokeRect(310, 240, 100, 50);
```

```
// Set fill color to yellow using rgb()
context.fillStyle = "rgb(255, 255, 0)";
// Draw a yellow filled rectangle
context.fillRect(420, 160, 100, 50);

// Set fill color to green with an alpha of 0.6
context.fillStyle = "rgba(0, 255, 0, 0.6)";
// Draw a semi-transparent green filled rectangle
context.fillRect(450, 180, 100, 50);

// TEXTURES
// Get a handle to the Image object
var fireImage = document.getElementById("fire");
var pattern = context.createPattern(fireImage, "repeat");

// Set fill style to newly created pattern
context.fillStyle = pattern;
// Draw a pattern filled rectangle
context.fillRect(420, 240, 130, 50);
```
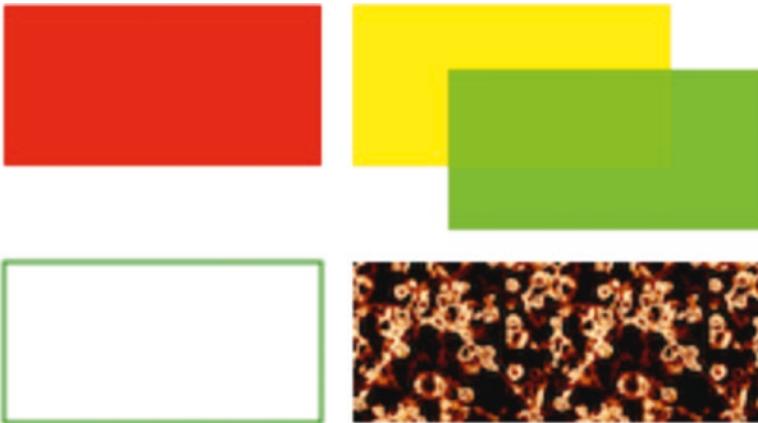

Figure 1-5. *Drawing with colors and textures*

In addition to these methods, the canvas also provides several methods to use color gradients, shadows, and patterns while drawing. I encourage you to take the time to explore the canvas and context API more thoroughly when you get the chance.

Drawing Images

Although we can achieve quite a lot using just the drawing methods we have covered so far, we still need to explore how to use images. Learning how to draw images will enable you to draw game backgrounds, character sprites, and effects like explosions that can make your games come alive.

We can draw images and sprites on the canvas using the `drawImage()` method. The context provides us with three different versions of this method:

- `drawImage(image, x, y)`: Draws the image on the canvas at (x, y)

- `drawImage(image, x, y, width, height)`: Scales the image to the specified width and height and then draws it at (x, y)

- `drawImage(image, sourceX, sourceY, sourceWidth, sourceHeight, x, y, width, height)`: Clips a rectangle from the image at (sourceX, sourceY) with dimensions (sourceWidth, sourceHeight), scales it to the specified width and height, and draws it on the canvas at (x, y)

Before we start drawing images, we need to load another image into the browser. We will add one more `<img>` tag after the `<canvas>` tag in our HTML file:

```
<img src="spaceship.png" id="spaceship">
```

Once the image has been loaded, we can draw it using the code shown in Listing 1-8.

Listing 1-8. Drawing Images

```
// DRAWING IMAGES
// Get a handle to the Image object
var image = document.getElementById("spaceship");

// Draw the image at (0, 350)
context.drawImage(image, 0, 350);

// Scale the image to half the original size
context.drawImage(image, 0, 400, 100, 25);

// Draw part of the image
context.drawImage(image, 0, 0, 60, 50, 0, 420, 60, 50);
```

The code in Listing 1-8 will draw the images shown in Figure 1-6. The last example in Listing 1-8, where we draw only a part of the image, will become especially useful when we start using sprite sheets to combine our game assets and store multiple sprites in a single large image.

Figure 1-6. *Drawing images*

Transforming and Rotating

The context object has several methods for transforming the coordinate system used for drawing elements. These methods are

- translate(x, y): Moves the canvas and its origin to a different point (x, y)

- rotate(angle): Rotates the canvas clockwise around the current origin by angle (radians)

- scale(x, y): Scales the objects drawn by a multiple of x and y along the respective axes

A common use of these methods is to rotate objects or sprites when drawing them. We can do this by

- Translating the canvas origin to the location of the object

- Rotating the canvas by the desired angle

- Drawing the object

- Restoring the canvas back to its original state

Let's look at rotating objects before drawing them, as shown in Listing 1-9.

Listing 1-9. Rotating Objects Before Drawing Them

```
// ROTATION AND TRANSLATION
//Translate origin to location of object
context.translate(250, 370);
//Rotate about the new origin by 60 degrees
context.rotate(Math.PI / 3);
context.drawImage(image, 0, 0, 60, 50, -30, -25, 60, 50);
//Restore to original state by rotating and translating back
context.rotate(-Math.PI / 3);
context.translate(-240, -370);
```

11

```
//Translate origin to location of object
context.translate(300, 370);
//Rotate about the new origin
context.rotate(3 * Math.PI / 4);
context.drawImage(image, 0, 0, 60, 50, -30, -25, 60, 50);
//Restore to original state by rotating and translating back
context.rotate(-3 * Math.PI / 4);
context.translate(-300, -370);
```

The code in Listing 1-9 will draw the two rotated ship images shown in Figure 1-7.
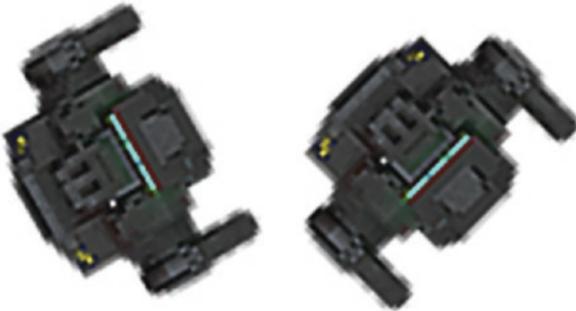
Figure 1-7. *Rotating images*

■ **Note**   Apart from rotating and translating back, you can also restore the canvas state by first using the save() method before starting the transformations and then calling the restore() method at the end of the transformations.

With this last example, we have covered all the essentials of the canvas that we will need to build our games. There is still a lot of the API that we have not covered here. You can read more about the canvas API at https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

The audio Element

Using the HTML5 audio element is the new standard way to embed an audio file into a web page. Until this element came along, most pages played audio files using embedded plug-ins (such as Flash).

The audio element can be created in HTML using the <audio> tag or in JavaScript using the Audio object. An example is shown in Listing 1-10.

Listing 1-10. The HTML5 <audio> Tag

```
<audio src="music.mp3" controls="controls">
    Your browser does not support HTML5 Audio. Please shift to a newer browser.
</audio>
```

■ **Note**   Browsers that do not support audio will ignore the `<audio>` tag and render anything inside the `<audio>` tag. You can use this feature to show users on older browsers alternative fallback content or a message directing them to a more modern browser.

The `controls` attribute included in Listing 1-10 makes the browser display a simple browser-specific interface for playing the audio file (such as a play/pause button and volume controls).

The `audio` element has several other attributes, such as the following:

- `preload`: Specifies whether or not the audio should be preloaded

- `autoplay`: Specifies whether or not to start playing the audio as soon as the object has loaded

- `loop`: Specifies whether to keep replaying the audio once it has finished

There are currently three popular file formats supported by browsers: MP3 (MPEG Audio Layer 3), WAV (Waveform Audio), and OGG (Ogg Vorbis). One thing to watch out for is that not all browsers support all audio formats. Firefox, for example, does not play MP3 files directly because of patent and licensing issues (and has to rely on operating system support), though it does play OGG and WAV files directly. Safari, on the other hand, supports MP3 but does not support OGG. Table 1-1 shows the formats supported by the latest version of popular browsers.

Table 1-1. *Audio Formats Supported by Current Browsers*

Browser	MP3	WAV	OGG
Internet Explorer	Yes	No	No
Edge	Yes	Yes	No
Firefox	Using OS support	Yes	Yes
Chrome	Yes	Yes	Yes
Safari	Yes	Yes	No
Opera	Yes	Yes	Yes

The way to work around this limitation is to provide the browser with alternative formats to play. The `audio` element allows multiple source elements within the `<audio>` tag, and the browser automatically uses the first recognized format (see Listing 1-11).

Listing 1-11. The <audio> Tag with Multiple Sources

```
<audio controls="controls">
    <source src="music.ogg" type="audio/ogg" />
    <source src="music.mp3" type="audio/mpeg" />
      Your browser does not support HTML5 Audio. Please shift to a newer browser.
</audio>
```

Audio can also be loaded dynamically by using the Audio object in JavaScript. The Audio object allows us to load, play, and pause sound files as needed, which is what will be used for games (see Listing 1-12).

Listing 1-12. Dynamically Loading an Audio File

```
<script>
    //Create a new Audio object
    var sound = new Audio();

    // Select the source of the sound
    sound.src = "music.ogg";
    // This will only work on browsers that support OGG

    // Play the sound
    // sound.play();
</script>
```

Unlike with the <audio> HTML tag, where we could easily specify multiple formats, when using JavaScript we need a way to detect the formats supported by the browser so we can load the appropriate format. The Audio object provides us with a method called canPlayType() that returns values of "", "maybe", or "probably" to indicate support for a specific codec. We can use this to create a simple check and load the appropriate audio format, as shown in Listing 1-13.

Listing 1-13. Testing for Audio Support

```
<script>
    var audio = document.createElement("audio");
    var mp3Support, oggSupport;

    if (audio.canPlayType) {
        // Currently canPlayType() returns: "", "maybe", or "probably"
        mp3Support = "" !== audio.canPlayType("audio/mpeg");
        oggSupport = "" !== audio.canPlayType("audio/ogg; codecs=\"vorbis\"");
    } else {
        // The audio tag is not supported
        mp3Support = false;
        oggSupport = false;
    }

    // Check for ogg, then mp3, and finally set soundFileExtn to undefined
    var soundFileExtn = oggSupport ? ".ogg" : mp3Support ? ".mp3" : undefined;

    if (soundFileExtn) {
        var sound = new Audio();
        // Load sound file with the detected extension

        sound.src = "music" + soundFileExtn;
        sound.play();
    }
</script>
```

Listing 1-13 uses canPlayType() to set a soundFileExtn property, which we can then use to load future audio files. We will use this idea when we build audio into our games in later chapters.

The Audio object triggers several different events to help us know when the sound has been loaded and is ready for playing. The loadedmetadata event is fired when the initial audio file metadata has been loaded by the browser. The canplay event is fired once enough of the audio file has been downloaded to start playing, and the canplaythrough event is fired when the browser can play the entire audio file without needing to pause and buffer the file. We can use the canplaythrough event to keep track of when the sound file has been loaded sufficiently for our purposes. Listing 1-14 shows an example of how the canplaythrough event can be used to play a sound once it has been loaded.

Listing 1-14. Waiting for an Audio File to Load

```
<script>
    // Play the sound after waiting for it to load
    if (soundFileExtn) {
        var sound = new Audio();

        sound.addEventListener("canplaythrough", function() {
            sound.play();
        });

        // Load sound file with the detected extension
        sound.src = "music" + soundFileExtn;
    }
</script>
```

Now that we have looked at how to check for supported audio formats, dynamically load audio, and detect when an audio file has loaded, we can combine these concepts to design an audio preloader that will dynamically load all the game audio resources before starting the game. We will look at this idea in more detail in the next few chapters when we build an asset loader for our games.

The image Element

The image element allows us to display images inside an HTML file. The simplest way to do this is by using the <image> tag and specifying an src attribute, as shown earlier and again here in Listing 1-15.

Listing 1-15. The <image> Tag

```
<img src="spaceship.png" id="spaceship">
```

You can also load an image dynamically using JavaScript by instantiating a new Image object and setting its src property, as shown in Listing 1-16.

Listing 1-16. Dynamically Loading an Image

```
var image = new Image();
image.src = "spaceship.png";
```

You can use either of these methods to get an image for drawing on a canvas.

Image Loading

Games are usually designed to wait for all the images to load completely before they start so as to avoid errors due to partly loaded images. While the images are being loaded, programmers commonly display a progress bar or status indicator that shows the percentage of images loaded.

The Image object provides us with an onload event that gets fired as soon as the browser finishes loading the image file. Using this event, we can keep track of when the image has loaded, as shown in the example in Listing 1-17.

Listing 1-17. Waiting for an Image to Load

```
image.onload = function() {
    alert("Image finished loading");
};
```

Using the onload event, we can create a simple image loader that tracks images loaded so far (see Listing 1-18).

Listing 1-18. Simple Image Loader

```
var imageLoader = {
    loaded: true,
    loadedImages: 0,
    totalImages: 0,
    load: function(url) {
        this.totalImages++;
        this.loaded = false;
        var image = new Image();
        image.src = url;
        image.onload = function() {
            imageLoader.loadedImages++;
            if (imageLoader.loadedImages === imageLoader.totalImages) {
                imageLoader.loaded = true;
            }
            image.onload = undefined;
        }
        return image;
    }
}
```

In this code, we create an imageLoader object with a load() method. This load() method takes an image URL, and increases the totalImages counter each time it is called. It then dynamically creates an Image object and sets the object's src property. Finally, it uses the object's onload event handler to increment the loadedImages counter, and once the counter reaches totalImages, it sets the loaded variable back to true.

This image loader can be invoked to load a large number of images (say in a loop). We can check to see if all the images are loaded by using imageLoader.loaded, and we can draw a percentage/progress bar by using loadedImages/totalImages.

Don't worry about actually using this loader yet. This is just a partial code snippet to help illustrate the basic idea for an image loader. We will be building a more complete version of an asset loader for our games in the coming chapters.

Sprite Sheets

Another concern when your game has a lot of images is how to optimize the way the server loads these images. Games can require anything from tens to hundreds of images. Even a simple real-time strategy (RTS) game will need images for different units, buildings, maps, backgrounds, and effects. In the case of units and buildings, you might need multiple versions of images to represent different directions and states, and in the case of animations, you might need an image for each frame of the animation.

In one of my earlier RTS game projects, I used individual images for each animation frame and state for every unit and building, ending up with over 1,000 images. Since most browsers make only a few simultaneous requests at a time, downloading all these images took a lot of time, with an overload of HTTP requests on the server. While this wasn't a problem when I was testing the code locally, it was a bit of a pain when the code went onto the server. Players ended up waiting 5 to 10 minutes (sometimes longer) for the game to load before they could actually start playing. All the concurrent requests also caused considerable load on my web server.

Luckily for us, there is a simple way to fix this problem of too many images and HTTP requests, and this is where sprite sheets come in. Sprite sheets store all the sprites (images) for a game entity in a single large image file. When displaying the images, we calculate the offset of the sprite we want to show and use the ability of the drawImage() method to draw only a part of an image. The spaceship.png image we have been using in this chapter is an example of a sprite sheet since it contains multiple spaceship sprites within the same file.

Looking at the code fragments in Listings 1-19 and 1-20, you can see examples of drawing an image loaded individually versus drawing an image loaded in a sprite sheet.

Listing 1-19. Drawing an Image Loaded Individually

```
// First: (Load individual images and store in a big array)

// Three arguments: the element, and destination (x, y) coordinates
var image = imageArray[imageNumber];
context.drawImage(image, x, y);
```

Listing 1-20. Drawing an Image Loaded in a Sprite Sheet

```
// First: (Load single sprite sheet image)

// Nine arguments: the element, source (x, y) coordinates,
// source width and height (for cropping),
// destination (x, y) coordinates, and
// destination width and height (resize)

context.drawImage (this.spriteImage, this.imageWidth*(imageNumber), O, this.imageWidth,
this.imageHeight, x, y, this.imageWidth, this.imageHeight);
```

In the first example, we store each individual sprite as a separate Image object in an array, and then draw a specific sprite by accessing the Image object. This method would require as many Image objects as sprites, and just as many HTTP requests to the server to fetch each image.

17

In the second example, we load a single large sprite sheet where all the sprites are placed side by side. Drawing the sprite involves calculating the x and y offset of the sprite within the image and then drawing just the appropriate portion of the image. This method involves only a single HTTP request and only a single Image object per sprite sheet, along with a little more complexity in computing the sprite location within the image. In terms of utilization of network resources, this is significantly better.

The following are some of the advantages of using a sprite sheet, which make using sprite sheets for any kind of complex game a no-brainer:

- *Fewer HTTP requests*: A unit that has 80 images (and so 80 requests) will now be downloaded in a single HTTP request.

- *Better compression*: Storing the images in a single file means that the header information doesn't repeat for each file and the single combined file is significantly smaller than all the individual files.

- *Faster load times*: With significantly lower HTTP requests and file sizes, the bandwidth usage and load times for the game drop as well, which means users won't have to wait for as long a time for the game to load.

Animation: Timer and Game Loops

The last thing you need to understand before you get started with actually building games is animation. Animating is just a matter of drawing an object, erasing it, and drawing it again at a new position, fast enough that the human eye only sees it as smooth movement.

The most common way to handle animation is by keeping a drawing function that gets called several times a second. Within this function, we iterate through all the game entities and draw them one by one.

Simpler games typically handle both animating or moving the entities and drawing them within the same drawing function. However, some games have a separate control/animation function that updates movement of the entities within the game, while the drawing function handles only the actual drawing of the entities on the screen. The animation function, since it is independent of the drawing function, can be called less often than the drawing function. Listing 1-21 contains skeleton code illustrating a typical animation and drawing routine.

Listing 1-21. Typical Animation and Drawing Loop

```
function animationLoop(){
    // Iterate through all the items in the game
    //And move them
}

function drawingLoop(){
    //1. Clear the canvas
    //2. Iterate through all the items
    //3. And draw each item
}
```

Assuming we built a working drawingLoop() method for our game, we need to figure out a way to call drawingLoop() repeatedly at regular intervals. The simplest way of achieving this is to use the two timer methods setInterval() and setTimeout(). setInterval(functionName, timeInterval) tells the browser to keep calling a given function repeatedly at fixed time intervals until the clearInterval() function is called. When we need to stop animating (when the game is paused, or has ended), we use clearInterval(). Listing 1-22 shows an example of how this would work.

Listing 1-22. Calling Drawing Loop with setInterval()

```
// Call drawingLoop() every 20 milliseconds
var gameLoop = setInterval(drawingLoop, 20);
// Stop calling drawingLoop() and clear the gameLoop variable
clearInterval(gameLoop);
```

setTimeout(functionName, timeInterval) tells the browser to call a given function one single time after a given time interval, as shown in the example in Listing 1-23.

Listing 1-23. Calling Drawing Loop with setTimeout()

```
function drawingLoop(){
    // 1. Call the drawingLoop() method once after 20 milliseconds
    var gameLoop = setTimeout(drawingLoop,20);

    // 2. Clear the canvas

    // 3. Iterate through all the items

    // 4. And draw them
}
```

Unlike with setInterval(), when using setTimeout() we need to make a new call each time since setTimeout() only calls the drawingLoop() method once. When we need to stop animating (when the game is paused, or has ended), we can use clearTimeout():

```
// Stop calling drawingLoop() and clear the gameLoop variable
clearTimeout(gameLoop);
```

Now, don't get too worried if some of this seems a little confusing or abstract at this point. This chapter is only meant to be a quick crash course, and I just want you to get a general overview of how this works. We will be looking at detailed working examples of all of these functions when we start building our games in later chapters, at which point everything should start making a lot more sense.

requestAnimationFrame

While using setInterval() or setTimeout() as a way to animate frames does work, browser vendors have come up with a new API specifically for handling animation. Some of the advantages of using this API instead of setInterval() are that the browser can do the following:

- Optimize the animation code into a single reflow-and-repaint cycle, resulting in smoother animation

- Pause the animation when the tab is not visible, leading to less CPU and GPU usage

- Automatically cap the frame rate on machines that do not support higher frame rates, or increase the frame rate on machines that are capable of processing them

Around the time that I was writing the first edition of this book, browser vendors had their own proprietary names for the methods in the API (such as Microsoft's msrequestAnimationFrame() method and Mozilla's mozRequestAnimationFrame() method). Since then, however, all browsers have standardized this API implementation and you can now use requestAnimationFrame() and cancelAnimationFrame() across all browsers that support HTML5.

■ **Note**   Now that we have no guarantee of frame rate (the browser decides the speed at which it will call our drawing loop), we need to ensure that animated objects move at the same speed on the screen independent of the actual frame rate. We do this either by animating objects in a separate `setTimeout()` or `setInterval()` loop, or by calculating the time since the previous drawing cycle and using it to interpolate the location of the object being animated.

The `requestAnimationFrame()` method can be called from within the `drawingLoop()` method similar to `setTimeout()`, as shown in Listing 1-24.

Listing 1-24. Calling Drawing Loop with requestAnimationFrame()

```
function drawingLoop(nowTime){
    // 1. Call the drawingLoop() method whenever the browser is ready to draw again
    var gameLoop = requestAnimationFrame(drawingLoop);

    // 2. Clear the canvas

    // 3. Iterate through all the items

    // 4. Optionally use nowTime and the last nowTime to interpolate frames

    // 5. And draw the items
}
```

When we need to stop animating (when the game is paused, or has ended), we can use `cancelAnimationFrame()`:

```
// Stop calling drawingLoop() and clear the gameLoop variable
cancelAnimationFrame(gameLoop);
```

This section has covered the primary ways to add animation to your games. We will be looking at actual implementations of these animation loops in the coming chapters.

Summary

In this chapter, we looked at the basic elements of HTML5 that are needed for building games. We covered how to use the `canvas` element to draw shapes, write text, and manipulate images. We examined how to use the `audio` element to load and play sounds across different browsers. We also briefly covered the basics of animation, preloading objects and using sprite sheets.

The topics we covered here are just a starting point and not exhaustive by any means. This chapter was meant to be a quick crash course or refresher on HTML5 and a handy reference for easily looking up syntax or code examples whenever needed. As I mentioned earlier, we will be going into these topics in more detail, along with complete implementations, as we build our games in the coming chapters.

If you had trouble keeping up and would like a more detailed explanation of the basics of JavaScript and HTML5, I would recommend reading introductory books on JavaScript and HTML5, such as *JavaScript for Absolute Beginners* by Terry McNavage and *The Essential Guide to HTML5* by Jeanine Meyer.

Now that we have the basics out of the way, let's get started building our first game.

CHAPTER 2

Creating a Basic Game World

The arrival of smartphones and handheld devices that support gaming has created a renewed interest in simple puzzle and physics-based games that can be played for short periods of time. Most of these games have a simple concept, small levels, and are easy to learn. One of the most popular and famous games in this genre is Angry Birds (by Rovio Entertainment), a puzzle/strategy game where players use a slingshot to shoot birds at enemy pigs. Despite a fairly simple premise, the game has been downloaded and installed on over two billion devices around the world. The game uses a physics engine to realistically model the slinging, collisions, and breaking of objects inside its game world.

Over the next four chapters, we are going to build our own physics-based puzzle game with complete playable levels. Our game, Froot Wars, will have fruits as protagonists, junk food as the enemy, and some breakable structures within the level.

We will be implementing all the essential components you will need in your own games—splash screens, loading screens and preloaders, menu screens, parallax scrolling, sound, realistic physics with the Box2D physics engine, and a scoreboard. Once you have this basic framework, you should be able to reuse these ideas in your own puzzle games.

So let's get started.

Basic HTML Layout

The first thing we need to do is to create the basic game layout. This will consist of several layers:

- *Splash screen*: Shown when the game page is loaded

- *Game start screen*: A menu that allows the player to start the game or modify settings

- *Loading/progress screen*: Shown whenever the game is loading assets (such as images and sound files)

- *Game canvas*: The actual game layer

- *Scoreboard*: An overlay above the game canvas to show a few buttons and the score

- *Ending screen*: A screen displayed at the end of each level

Each of these layers will be either a `div` element or a `canvas` element that we will display or hide as needed. The code will be laid out with separate folders for images and JavaScript code.

Creating the Splash Screen and Main Menu

We start with a skeleton HTML file, similar to the first chapter, and add the markup for our containers, as shown in Listing 2-1.

Listing 2-1. Basic Skeleton (index.html) with the Layers Added

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">
        <title>Froot Wars</title>
            <script src="js/game.js" type="text/javascript"></script>
        <link rel="stylesheet" href="styles.css" type="text/css" media="screen">
    </head>

    <body>
        <div id="wrapper">
            <div id="gamecontainer">

                <canvas id="gamecanvas" width="640" height="480" class="gamelayer">
                </canvas>

                <div id="scorescreen" class="gamelayer">
                    <img id="togglemusic" src="images/icons/sound.png" alt="Toggle Music">
                    <img src="images/icons/prev.png" alt="Restart Level ">
                    <span id="score">Score: 0</span>
                </div>

                <div id="gamestartscreen" class="gamelayer">
                    <img src="images/icons/play.png" alt="Play Game"><br>
                    <img src="images/icons/settings.png" alt="Settings">
                </div>

                <div id="levelselectscreen" class="gamelayer">
                </div>

                <div id="loadingscreen" class="gamelayer">
                    <div id="loadingmessage"></div>
                </div>

                <div id="endingscreen" class="gamelayer">
                    <div>
                        <p id="endingmessage">The Level Is Over Message</p>
                        <p id="playcurrentlevel" class="endingoption"><img src="images/
                        icons/prev.png" alt="Replay">Replay Current Level</p>
```

```
                    <p id="playnextlevel" class="endingoption"><img src="images/icons/
                    next.png" alt="Next">Play Next Level</p>
                    <p id="returntolevelscreen" class="endingoption"><img src="images/
                    icons/return.png" alt="Return">Return to Level Screen</p>
                </div>
            </div>

        </div>
    </div>
</body>
</html>
```

As you can see, we defined a main `gamecontainer div` element that contains each of the game layers: `gamestartscreen`, `levelselectscreen`, `loadingscreen`, `scorescreen`, `endingscreen`, and finally `gamecanvas`. All of these are placed inside a `wrapper div`, which we can later use for positioning and resizing the game around the page as needed.

We also link to two external files: `game.js` for JavaScript and `styles.css` for CSS. Keeping the JavaScript and CSS as separate files makes the code easier to maintain. In larger projects, it is common to break out the CSS and JavaScript into multiple files, and very large projects often use a dependency loading system to automatically load all the distinct JavaScript files. For this game, single files for JavaScript and CSS will suffice.

We will start by creating the `styles.css` file and adding styles for the game container and the starting menu screen, as shown in Listing 2-2.

Listing 2-2. CSS Styles for the Container and Start Screen (styles.css)

```css
body {
    background: #000900;

    /* Prevent the ugly blue highlighting from accidental selection of text */
    user-select: none;
}

#wrapper {
    position: absolute;
}

#gamecontainer {

    /* Set game container width, height, and background */
    width: 640px;
    height: 480px;
    background: url("images/splashscreen.png");
}

.gamelayer {
    width: 100%;
    height: 100%;
    position: absolute;
    display: none;
}
```

23

```
/* Game Starting Menu Screen */

#gamestartscreen {
    padding-top: 250px;
    text-align: center;
}

#gamestartscreen img {
    margin: 10px;
    cursor: pointer;
}
```

We have done the following in this CSS style sheet so far:

- Set the default page background color to almost black with a slight tinge of green and disabled highlighting of text or elements by dragging the mouse.

- Defined our game container with a size of 640px by 480px.

- Made sure all game layers are positioned using absolute positioning (they are placed on top of each other) so that we can show/hide and superimpose layers as needed. Each of these layers has the same size as the parent game container and is hidden by default.

- Set our game splash screen image as the main container background so it is the first thing a player sees when the page loads.

- Added some styling for our game start screen (the starting menu), which has options such as starting a new game and changing game settings.

■ **Note**   All the images and source code are available from this book's product page on the Apress website (`www.apress.com/9781484229095`) by clicking the Download Source Code button. If you would like to follow along, you can copy all the asset files into a fresh folder and build the game on your own.

If we open in a browser the HTML file we have created so far, we see the game splash screen, as shown in Figure 2-1.

Figure 2-1. *The game splash screen*

We need to add some JavaScript code to start showing the main menu, the loading screen, and the game. We will keep all our game-related JavaScript code in a single file (js/game.js).

We start by defining a game object that will contain most of our game code. The first thing we need is an init() function that will be called after the browser loads the HTML document.

Listing 2-3. A Basic game Object (js/game.js)

```
var game = {
    // Start initializing objects, preloading assets and display start screen
    init: function() {
        //Get handler for game canvas and context
        game.canvas = document.getElementById("gamecanvas");
        game.context = game.canvas.getContext("2d");

        // Hide all game layers and display the start screen
        game.hideScreens();
        game.showScreen("gamestartscreen");
    },

    //
    hideScreens: function() {
        var screens = document.getElementsByClassName("gamelayer");
```

```
        // Iterate through all the game layers and set their display to none
        for (let i = screens.length - 1; i >= 0; i--) {
            var screen = screens[i];

            screen.style.display = "none";
        }
    },

    hideScreen: function(id) {
        var screen = document.getElementById(id);

        screen.style.display = "none";
    },

    showScreen: function(id) {
        var screen = document.getElementById(id);

        screen.style.display = "block";
    },
};
```

The code in Listing 2-3 defines a JavaScript object called game with an init() function. This init() function first saves references to the game canvas and context so we can refer to them more easily using game.context and game.canvas. After that it hides all game layers and shows the game start screen using the hideScreens() and showScreen() methods. Next, we have three helper methods, hideScreens(), hideScreen(), and showScreen(), which modify the display CSS attribute to help us show or hide the menu screens that we created.

Trying to manipulate image and div elements before confirming that the page has loaded completely will result in unpredictable behavior (including JavaScript errors). We can safely call this game.init() method after the window has loaded by adding a small snippet of JavaScript code at the bottom of game.js (shown in Listing 2-4).

Listing 2-4. Calling game.init() Method Safely Using the load Event

```
// Initialize game once page has fully loaded
window.addEventListener("load", function() {
    game.init();
});
```

When we open our HTML code in the browser, the browser initially displays the splash screen and then displays the game start screen on top of the splash screen, as shown in Figure 2-2.
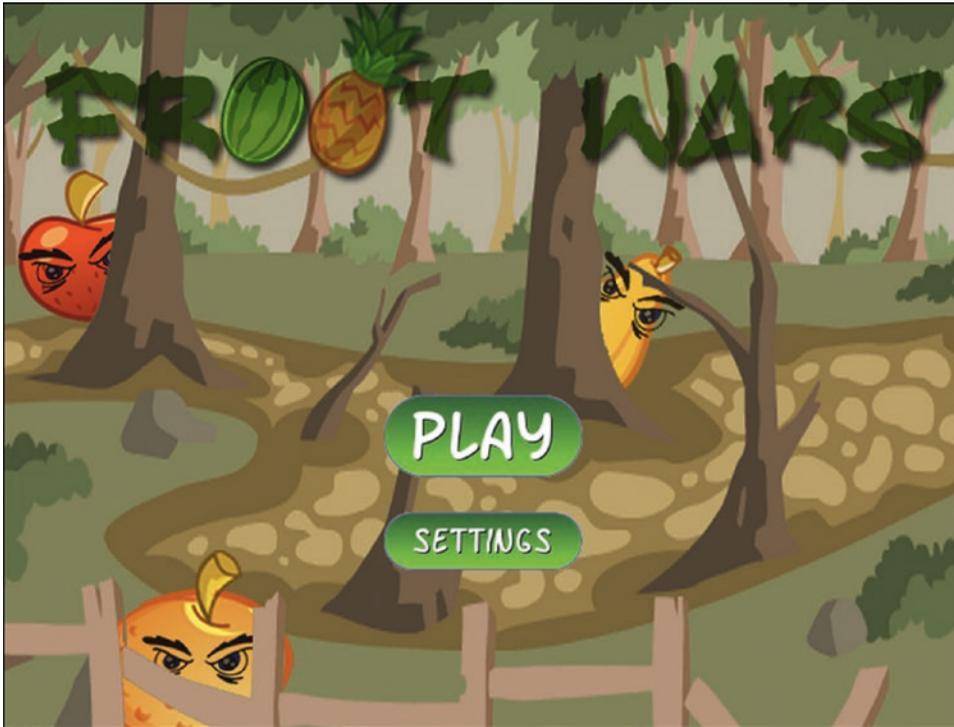
Figure 2-2. *The game start screen and menu options*

Level Selection

So far we have waited for the game HTML file to load completely and then displayed a main menu with two options, Play and Settings. When the user clicks the Play button, ideally we would like to display a level selection screen that shows a list of available levels.

Before we can do this, we need to create an object for handling levels. This object will contain both the level data and some simple functions for handling level initialization. We will create this levels object inside game.js and place it after the game object, as shown in Listing 2-5.

Listing 2-5. *Simple levels Object with Level Data and Functions*

```
var levels = {
    // Level data
    data: [{   // First level
        foreground: "desert-foreground",
        background: "clouds-background",
        entities: []
    }, {   // Second level
        foreground: "desert-foreground",
        background: "clouds-background",
        entities: []
    }],
```

```
    // Initialize level selection screen
    init: function() {
        var levelSelectScreen = document.getElementById("levelselectscreen");

        // An event handler to call
        var buttonClickHandler = function() {
            game.hideScreen("levelselectscreen");

            // Level label values are 1, 2. Levels are 0, 1
            levels.load(this.value - 1);
        };

        for (let i = 0; i < levels.data.length; i++) {
            var button = document.createElement("input");

            button.type = "button";
            button.value = (i + 1); // Level labels are 1, 2
            button.addEventListener("click", buttonClickHandler);

            levelSelectScreen.appendChild(button);
        }

    },

    // Load all data and images for a specific level
    load: function(number) {
    }
};
```

The levels object has a data array that contains information about each of the levels. For now, the only level information we store is a background image and foreground image. However, we will be adding information about the hero characters, the villains, and the destructible entities within each level. This will allow us to add new levels very quickly by just adding new items to the array.

The next thing the levels object contains is an init() function that goes through the level data and dynamically generates buttons for each of the levels. Each of the buttons is assigned a click event handler, which calls the load() method and then hides the level selection screen. Note that we use a level index starting from 0 internally since JavaScript arrays are zero-based, but when we display the level numbers to the player on the level selection screen, we start the numbering from 1.

Finally, the levels object has a placeholder load() method, which is currently empty.

We will call levels.init() from inside the game.init() method to generate the level selection screen buttons when the game is first initialized. The game.init() method now looks as shown in Listing 2-6.

Listing 2-6. Initializing Levels from game.init()

```
init: function() {
    //Get handler for game canvas and context
    game.canvas = document.getElementById("gamecanvas");
    game.context = game.canvas.getContext("2d");
```

```
    // Initialize objects
    levels.init();

    // Hide all game layers and display the start screen
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

We also need to add some CSS styling for the buttons inside styles.css, as shown in Listing 2-7.

Listing 2-7. CSS Styles for the Level Selection Screen

```
/* Level Selection Screen */

#levelselectscreen {
    padding-top: 150px;
    padding-left: 50px;
}

#levelselectscreen input {
    margin: 20px;
    cursor: pointer;

    background: url("images/icons/level.png") no-repeat;
    color: yellow;
    font-size: 20px;

    width: 64px;
    height: 64px;

    border: 0;

    /* Remove the default blue border when an input is selected */
    outline: 0;
}
```

This fairly simple CSS code adds some padding, margins, and styling to the buttons. It also sets a default background image for the buttons.

The next thing we need to do is create, inside the game object, a simple game.showLevelScreen() method that hides the main menu screen and displays the level selection screen, as shown in Listing 2-8.

Listing 2-8. showLevelScreen() Method Inside the game Object

```
showLevelScreen: function() {
    game.hideScreens();
    game.showScreen("levelselectscreen");
},
```

This method first hides all the other game layers and then shows the `levelselectscreen` layer.

The last thing we need to do is call the `game.showLevelScreen()` method when the user clicks the Play button. We do this by calling the method from the play image's `onclick` event in our HTML file:

```
<img src="images/icons/play.png" alt="Play Game"
    onclick="game.showLevelScreen()">
```

Now, when we start the game and click the Play button, the browser hides the main menu, and shows the level selection screen with buttons for each of the levels, as shown in Figure 2-3.

Figure 2-3. *The level selection screen*

Right now, we only have a couple of levels showing. However, as we add more levels, the code will automatically detect the levels and add the right number of buttons (formatted properly, thanks to the CSS). When the user clicks these buttons, the browser will hide the level selection screen and then call the `levels.load()` method that we have yet to implement.

Loading Images

Before we implement the levels themselves, we need to put in place the image loader and the loading screen. This will allow us to programmatically load the images for a level and start the game once all the assets have finished loading.

We are going to design a simple loading screen that contains an animated GIF with a progress bar image and some text above it showing the number of images loaded so far. First, we need to add the CSS in Listing 2-9 to styles.css.

Listing 2-9. CSS for the Loading Screen

```css
/* Loading Screen */

#loadingscreen {
    background: rgba(100, 100, 100, 0.5);
}

#loadingmessage {
    margin-top: 400px;
    text-align: center;
    height: 48px;
    color: white;
    background: url("images/loader.gif") no-repeat center;
    font: 12px Arial;
}
```

This CSS adds a dim gray color over the game background to let the user know that the game is currently processing something and is not ready to receive any user input. It also displays a loading message in white text. Finally, it places a progress bar image, which is an animated GIF file, in the background.

The next step is to create a JavaScript asset loader based on the code from Chapter 1. The loader will do the work of actually loading the assets and then updating the loadingscreen div element. We will define a loader object inside game.js, as shown in Listing 2-10.

Listing 2-10. The Image/Sound Asset Loader

```javascript
var loader = {
    loaded: true,
    loadedCount: 0, // Assets that have been loaded so far
    totalCount: 0, // Total number of assets that need loading

    init: function() {
        // Check for sound support
        var mp3Support, oggSupport;
        var audio = document.createElement("audio");

        if (audio.canPlayType) {
                // Currently canPlayType() returns:  "", "maybe" or "probably"
            mp3Support = "" !== audio.canPlayType("audio/mpeg");
            oggSupport = "" !== audio.canPlayType("audio/ogg; codecs=\"vorbis\"");
        } else {
            // The audio tag is not supported
            mp3Support = false;
            oggSupport = false;
        }

        // Check for ogg, then mp3, and finally set soundFileExtn to undefined
        loader.soundFileExtn = oggSupport ? ".ogg" : mp3Support ? ".mp3" : undefined;
    },
```

```
loadImage: function(url) {
    this.loaded = false;
    this.totalCount++;

    game.showScreen("loadingscreen");

    var image = new Image();

    image.addEventListener("load", loader.itemLoaded, false);
    image.src = url;

    return image;
},

soundFileExtn: ".ogg",

loadSound: function(url) {
    this.loaded = false;
    this.totalCount++;

    game.showScreen("loadingscreen");

    var audio = new Audio();

    audio.addEventListener("canplaythrough", loader.itemLoaded, false);
    audio.src = url + loader.soundFileExtn;

    return audio;
},

itemLoaded: function(ev) {
    // Stop listening for event type (load or canplaythrough) for this item
    now that it has been loaded
    ev.target.removeEventListener(ev.type, loader.itemLoaded, false);

    loader.loadedCount++;

    document.getElementById("loadingmessage").innerHTML = "Loaded " + loader.loadedCount
    + " of " + loader.totalCount;

    if (loader.loadedCount === loader.totalCount) {
        // Loader has loaded completely..
        // Reset and clear the loader
        loader.loaded = true;
        loader.loadedCount = 0;
        loader.totalCount = 0;

        // Hide the loading screen
        game.hideScreen("loadingscreen");
```

```
            // and call the loader.onload method if it exists
            if (loader.onload) {
                loader.onload();
                loader.onload = undefined;
            }
        }
    }
};
```

The asset loader in Listing 2-10 has the same elements we discussed in Chapter 1, but it is built in a more modular way. It has the following components:

- An init() method that detects the supported audio file format and saves it.

- Two methods for loading images and audio files: loadImage() and loadSound(). Both methods increment the totalCount variable and show the loading screen when invoked. The methods then dynamically create the asset, set the src attribute, and set the appropriate event listener (load for images and canplaythrough for audio) to call itemLoaded() once the asset is loaded.

- An itemLoaded() method that is invoked each time an asset finishes loading. This method updates the loaded count and the loading message. Once all the assets are loaded, the loading screen is hidden and an optional loader.onload() method is called (if defined). This lets us assign a callback function to be called once the images are loaded.

■ **Note**   Using a callback method makes it easy for us to wait while the images are loading and start the game once all the images have loaded.

Before the loader can be used, we need to call the loader.init() method from inside game.init() so that the loader is initialized when the game is getting initialized. The game.init() method now looks as shown in Listing 2-11.

Listing 2-11.   Initializing the Loader from game.init()

```
init: function() {
    //Get handler for game canvas and context
    game.canvas = document.getElementById("gamecanvas");
    game.context = game.canvas.getContext("2d");

    // Initialize objects
    levels.init();
    loader.init();

    // Hide all game layers and display the start screen
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

We will use the loader by calling one of the two load methods, loadImage() or loadSound(). When either of these load methods is called, the browser will display the loading screen shown in Figure 2-4 until all the images and sounds are loaded.

Figure 2-4. *The loading screen*

■ **Note**   You can optionally have different images for each of these screens by setting a different background property style for each div element.

Loading Levels

Now that we have an image loader in place, we can work on getting the levels loaded. For now, let's start with loading the game background, foreground, and slingshot images by defining a load() method inside the levels object, as shown in Listing 2-12.

Listing 2-12. Basic Skeleton for the load() Method Inside the levels Object

```
  // Load all data and images for a specific level
load: function(number) {

    // Declare a new currentLevel object
    game.currentLevel = { number: number };
    game.score = 0;

    document.getElementById("score").innerHTML = "Score: " + game.score;
    var level = levels.data[number];
```

```
    // Load the background, foreground, and slingshot images
    game.currentLevel.backgroundImage = loader.loadImage("images/backgrounds/" + level.
    background + ".png");
    game.currentLevel.foregroundImage = loader.loadImage("images/backgrounds/" + level.
    foreground + ".png");
    game.slingshotImage = loader.loadImage("images/slingshot.png");
    game.slingshotFrontImage = loader.loadImage("images/slingshot-front.png");

    // Call game.start() once the assets have loaded
    loader.onload = game.start;
}
```

The load() function creates a currentLevel object to store the loaded level data. So far we have only loaded a few images for the background, the foreground, and the front and back of the slingshot. We will eventually also use this method to load the heroes, villains, and blocks needed to build the game.

One last thing to note is that we call the game.start() method once the images are loaded by setting an onload callback. This start() method is where the actual game will be drawn.

Animating the Game

As discussed in Chapter 1, to animate our game, we will call our drawing and animation code multiple times a second using requestAnimationFrame.

We use the game.start() method to set up the animation loop, and then we draw the level inside the game.animate() method. The code is shown in Listing 2-13.

Listing 2-13. The start() and animate() Functions Inside the game Object

```
// Store current game state - intro, wait-for-firing, firing, fired, load-next-hero,
success, failure
mode: "intro",

// X & Y coordinates of the slingshot
slingshotX: 140,
slingshotY: 280,

// X & Y coordinate of point where band is attached to slingshot
slingshotBandX: 140 + 55,
slingshotBandY: 280 + 23,

// Flag to check if the game has ended
ended: false,

// The game score
score: 0,

// X axis offset for panning the screen from left to right
offsetLeft: 0,

start: function() {
    game.hideScreens();
```

```
    // Display the game canvas and score
    game.showScreen("gamecanvas");
    game.showScreen("scorescreen");

    game.mode = "intro";
    game.currentHero = undefined;

    game.offsetLeft = 0;
    game.ended = false;

    game.animationFrame = window.requestAnimationFrame(game.animate, game.canvas);

},

handleGameLogic: function() {
    // Temporary placeholder code. Keep panning the game towards the right
    game.offsetLeft++;
},

animate: function() {

    // Handle panning, game states, and control flow
    game.handleGameLogic();

    //  Draw the background with parallax scrolling
    // First draw the background image, offset by a fraction of the offsetLeft distance (1/4)
    // The bigger the fraction, the closer the background appears to be
    game.context.drawImage(game.currentLevel.backgroundImage, game.offsetLeft / 4, 0, game.
    canvas.width, game.canvas.height, 0, 0, game.canvas.width, game.canvas.height);
    // Then draw the foreground image, offset by the entire offsetLeft distance
    game.context.drawImage(game.currentLevel.foregroundImage, game.offsetLeft, 0, game.
    canvas.width, game.canvas.height, 0, 0, game.canvas.width, game.canvas.height);

    // Draw the base of the slingshot, offset by the entire offsetLeft distance
    game.context.drawImage(game.slingshotImage, game.slingshotX - game.offsetLeft, game.
    slingshotY);

    // Draw the front of the slingshot, offset by the entire offsetLeft distance
    game.context.drawImage(game.slingshotFrontImage, game.slingshotX - game.offsetLeft,
    game.slingshotY);

    if (!game.ended) {
        game.animationFrame = window.requestAnimationFrame(game.animate, game.canvas);
    }
},
```

The preceding code consists primarily of two methods, game.start() and game.animate(). The start() method does the following:

- Initializes a few variables that we need in the game: offsetLeft and mode. offsetLeft will be used for panning the game view around the entire level, and mode will be used to store the current state of the game (intro, wait for firing, firing, fired).

- Hides all other layers and displays the canvas layer and the score layer, which is a narrow bar on the top of the screen that contains the game score and a few game interface control elements.

- Sets the game animation interval to call the animate() function by using window. requestAnimationFrame.

The bigger method, animate(), will do all the animation and drawing within our game. The method starts with calling a temporary placeholder handleGameLogic() method, which we will use to handle panning as well as the game control flow using game modes. We will be implementing these later. For now, it contains a single line of code to keep increasing the offsetLeft property, which should pan the game screen toward the right.

We then draw the background and foreground images. For both the images, we first crop a canvas-sized portion of the image that is offset appropriately along the x-axis using the offsetLeft variable, and then draw it onto the canvas. One thing to note is that the background image and foreground image are moved at different speeds relative to the left offset: the background image is moved only one-fourth of the distance that the foreground image is moved. This difference in movement speed of the two layers will give us the illusion that the clouds are further away once we start panning around the level.

After the backgrounds, we draw the slingshot in the foreground, subtracting offsetLeft from its x-axis position so that the slingshot appears to stay in the same place while the game pans to the right.

Finally, we check if the game.ended flag has been set and, if not, use requestAnimationFrame to call animate() again. We can use the game.ended flag later to decide when to stop the animation loop.

■ **Note** Parallax scrolling is a technique used to create an illusion of depth by moving background images slower than foreground images. This technique exploits the fact that objects at a distance always appear to move slower than objects that are close by.

Before we can try out the code, we need to add a little more CSS styling inside styles.css to implement our score screen panel, as shown in Listing 2-14.

Listing 2-14. CSS for Score Screen Panel

```
/* Score Screen */

#scorescreen {
    height: 60px;
    font: 32px "Comic Sans MS";
    text-shadow: 0 0 2px black;
    color: white;
}
```

```
#scorescreen img {
    opacity: 0.6;
    top: 5%;
    left: 5%;
    position: relative;
    padding: 8px;
    cursor: pointer;
}

#score {
    position: absolute;
    top: 5%;
    right: 5%;
}
```

The scorescreen layer, unlike the other layers in our game, is just a narrow band at the very top of our game. Along with the usual positioning and styling, we set the opacity for the interface buttons to make them translucent. This ensures that the interface buttons (for toggling music and restarting the level) do not distract from the rest of the game.

When we run this code and try to start a level, we should see a basic level with the interface buttons and the score displayed at the top of the screen, as shown in Figure 2-5.
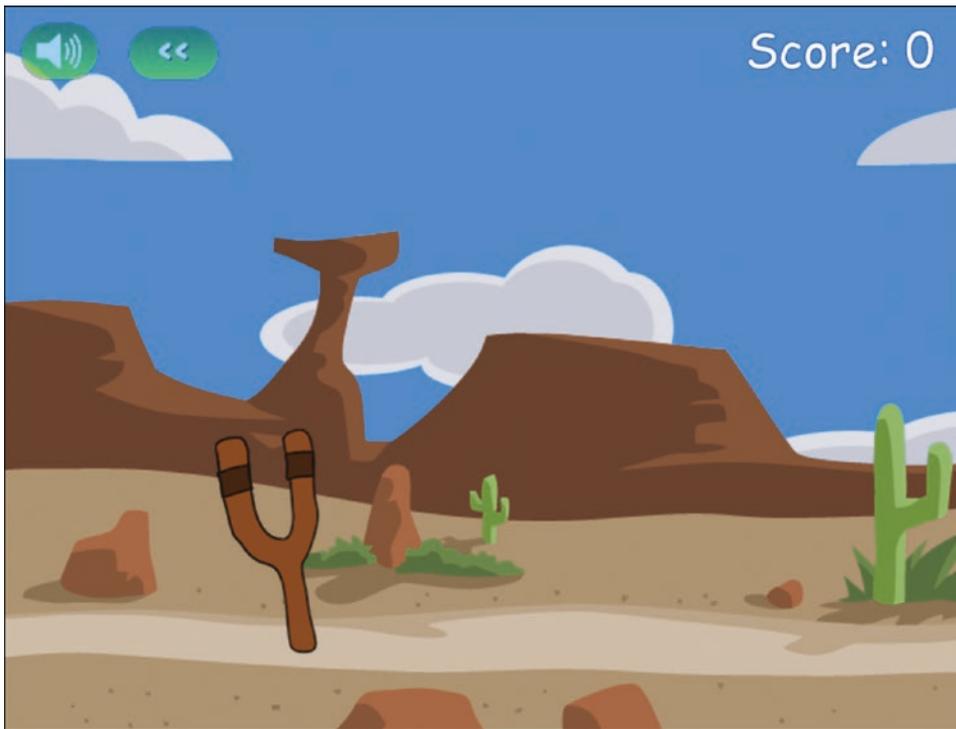
Figure 2-5. A basic level with the score

Our crude implementation of panning currently causes the screen to slowly pan toward the right until the image is no longer visible. Don't worry, we will be working on a better implementation soon.

As you can see, the clouds in the background move slower than the foreground because we move the background layer at a different speed, making it seem like the clouds are much farther than the mountains. We could potentially add more layers and move them at different speeds to build more of an effect. For example, the foreground with the cactus, the mountains, and the clouds in the background could form three distinct layers, moving at three different speeds. However, the two layers that we have right now are sufficient to illustrate the parallax effect fairly well.

Now that we have a basic level in place, we will add the ability to handle mouse input and implement panning around the level with game states.

Handling Mouse Input

JavaScript has several events that we can use to capture mouse input: `mousedown`, `mouseup`, and `mousemove`. To keep things simple we will create a separate `mouse` object inside `game.js` to handle all the mouse events, as shown in Listing 2-15.

Listing 2-15. Handling Mouse Events

```
var mouse = {
    x: 0,
    y: 0,
    down: false,
    dragging: false,

    init: function() {
        var canvas = document.getElementById("gamecanvas");

        canvas.addEventListener("mousemove",
            mouse.mousemovehandler, false);
        canvas.addEventListener("mousedown",
            mouse.mousedownhandler, false);
        canvas.addEventListener("mouseup",
            mouse.mouseuphandler, false);
        canvas.addEventListener("mouseout",
            mouse.mouseuphandler, false);
    },

    mousemovehandler: function(ev) {
        var offset = game.canvas.getBoundingClientRect();

        mouse.x = ev.clientX - offset.left;
        mouse.y = ev.clientY - offset.top;

        if (mouse.down) {
            mouse.dragging = true;
        }

        ev.preventDefault();
    },
```

```
    mousedownhandler: function(ev) {
        mouse.down = true;

        ev.preventDefault();
    },

    mouseuphandler: function(ev) {
        mouse.down = false;
        mouse.dragging = false;

        ev.preventDefault();
    }
};
```

This `mouse` object has an `init()` method that sets event handlers for when the mouse is moved, when a mouse button is pressed or released, and when the mouse leaves the canvas area. The following are the three handler methods that we use:

- `mousemovehandler()`: Uses the canvas's `getBoundingClientRect()` method and the event object's `clientX` and `clientY` properties to calculate the x and y coordinates of the mouse relative to the top-left corner of the canvas and stores them. It also checks whether the mouse button is pressed down while the mouse is being moved and, if so, sets the `dragging` variable to `true`.

- `mousedownhandler()`: Sets the `down` variable to `true`.

- `mouseuphandler()`: Sets the `down` and `dragging` variables to `false`. If the mouse leaves the canvas area, we call this same method.

All three methods additionally contain an extra line to prevent the default browser behavior for the mouse event.

Now that we have these methods in place, we can add code to interact with the game elements as needed. We also have access to the `mouse.x`, `mouse.y`, `mouse.dragging`, and `mouse.down` properties from anywhere within the game. As with all the previous `init()` methods, we call this method from `game.init()`, so it now looks as shown in Listing 2-16.

Listing 2-16. Initializing the Mouse from game.init()

```
init: function() {
    // Get handler for game canvas and context
    game.canvas = document.getElementById("gamecanvas");
    game.context = game.canvas.getContext("2d");

    // Initialize objects
    levels.init();
    loader.init();
    mouse.init();

    // Hide all game layers and display the start screen
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

With this bit of functionality in place, let's now implement some basic game states and panning.

Defining Our Game States

Remember the `game.mode` variable that we briefly looked at earlier when we were creating `game.start()`? Well, this is where it comes into the picture. We will be storing the current state of our game in this variable. Some of the modes or states that we expect our game to go through are as follows:

- `intro`: The level has just loaded and the game will pan around the level once to show the player everything in the level.

- `load-next-hero`: The game checks whether there is another hero to load onto the slingshot and, if so, loads the hero. If we run out of heroes or all the villains have been destroyed, the level ends.

- `wait-for-firing`: The game pans back to the slingshot area and waits for the user to fire the "hero." At this point, we are waiting for the user to click the hero. The user may also optionally drag the canvas screen with the mouse to pan around the level.

- `firing`: This happens after the user clicks the hero but before the user releases the mouse button. At this point, we are waiting for the user to drag the mouse around to decide the angle and height at which to fire the hero.

- `fired`: This happens after the user releases the mouse button. At this point, we launch the hero and let the physics engine handle everything while the user just watches. The game will pan so that the user can follow the path of the hero as far as possible.

We may implement more states as needed. One thing to note about these different states is that only one of them is possible at a time, and there are clear conditions for transitioning from one state to another, and what is possible during each state. This construct is popularly known as a *finite state machine* in computer science. We will be using these states to create some simple conditions for our panning code.

First we will build a `panTo()` method that will pan the screen to any specific location on the game, as shown in Listing 2-17. All of this code goes inside the `game` object after the `start()` method.

Listing 2-17. Implementing a panTo() Function

```
// Maximum panning speed per frame in pixels
maxSpeed: 3,

// Pan the screen so it centers at newCenter
// (or at least as close as possible)
panTo: function(newCenter) {

    // Minimum and Maximum panning offset
    var minOffset = 0;
    var maxOffset = game.currentLevel.backgroundImage.width - game.canvas.width;

    // The current center of the screen is half the screen width from the left offset
    var currentCenter = game.offsetLeft + game.canvas.width / 2;
```

```
    // If the distance between new center and current center is > 0 and we have not panned
    // to the min and max offset limits, keep panning
    if (Math.abs(newCenter - currentCenter) > 0 && game.offsetLeft <= maxOffset && game.
    offsetLeft >= minOffset) {
        // We will travel half the distance from the newCenter to currentCenter in each tick
        // This will allow easing
        var deltaX = (newCenter - currentCenter) / 2;

        // However if deltaX is really high, the screen will pan too fast, so if it is
        // greater than maxSpeed
        if (Math.abs(deltaX) > game.maxSpeed) {
            // Limit deltaX to game.maxSpeed (and keep the sign of deltaX)
            deltaX = game.maxSpeed * Math.sign(deltaX);
        }

        // And if we have almost reached the goal, just get to the ending in this turn
        if (Math.abs(deltaX) <= 1) {
            deltaX = (newCenter - currentCenter);
        }

        // Finally add the adjusted deltaX to offsetX so we move the screen by deltaX
        game.offsetLeft += deltaX;

        // And make sure we don't cross the minimum or maximum limits
        if (game.offsetLeft <= minOffset) {
            game.offsetLeft = minOffset;

            // Let calling function know that we have panned as close as possible to the
            // newCenter
            return true;
        } else if (game.offsetLeft >= maxOffset) {
            game.offsetLeft = maxOffset;

            // Let calling function know that we have panned as close as possible to the
            // newCenter
            return true;
        }

    } else {
        // Let calling function know that we have panned as close as possible to the
        // newCenter
        return true;
    }
},
```

The panTo() method slowly pans the screen to a given x coordinate (newCenter) and returns true either when the screen center reaches the coordinate or when the screen has panned to the extreme left or right.

The speed of panning varies based on the distance of the current center from newCenter, so the panning slows down as the screen pans closer to its destination. The code caps the panning speed using maxSpeed so that the panning never becomes too fast.

Each time panTo() is called, the screen center is shifted toward newCenter while there is still space to pan.

Eventually, once the screen either reaches its destination or reaches as close as possible (when offset reaches either minOffset or maxOffset), the method returns true. The maxOffset is calculated by comparing the width of the background image with that of the canvas, so the game will never pan past the end of the background image.

Now that we have an effective way to pan the screen, we will use it to implement panning within the handleGameLogic() method, as shown in Listing 2-18.

Listing 2-18. Implementing Panning in handleGameLogic()

```
handleGameLogic: function() {
    if (game.mode === "intro") {
        if (game.panTo(700)) {
            game.mode = "load-next-hero";
        }
    }

    if (game.mode === "wait-for-firing") {
        if (mouse.dragging) {
            game.panTo(mouse.x + game.offsetLeft);
        } else {
            game.panTo(game.slingshotX);
        }
    }

    if (game.mode === "load-next-hero") {
        // First count the heroes and villains and populate their respective arrays
        // Check if any villains are alive, if not, end the level (success)
        // Check if there are any more heroes left to load, if not end the level (failure)
        // Load the hero and set mode to wait-for-firing
        game.mode = "wait-for-firing";
    }

    if (game.mode === "firing") {
        // If the mouse button is down, allow the hero to be dragged around and aimed
        // If not, fire the hero into the air
    }

    if (game.mode === "fired") {
        // Pan to the location of the current hero as he flies
        // Wait till the hero stops moving or is out of bounds
    }

    if (game.mode === "level-success" || game.mode === "level-failure") {
        // First pan all the way back to the left
        // Then show the game as ended and show the ending screen
    }

},
```

We have now improved the handleGameLogic() method so it implements several of the game states we described earlier.

When the game is in the default intro mode, we pan the screen all the way to the right and, once there, switch the mode to load-next-hero. We haven't implemented the load-next-hero, firing, fired, level-success, or level-failure states yet. For now, the code just flips the load-next-hero mode on to wait-for-firing, which pans the screen back to the slingshot.

If we run the code we have so far, we will see that as the level starts, the screen pans toward the right until we reach the right extreme and panTo() returns true (see Figure 2-6). The game mode then changes from intro to wait-for-firing and the screen slowly pans back to the starting position and waits for user input.
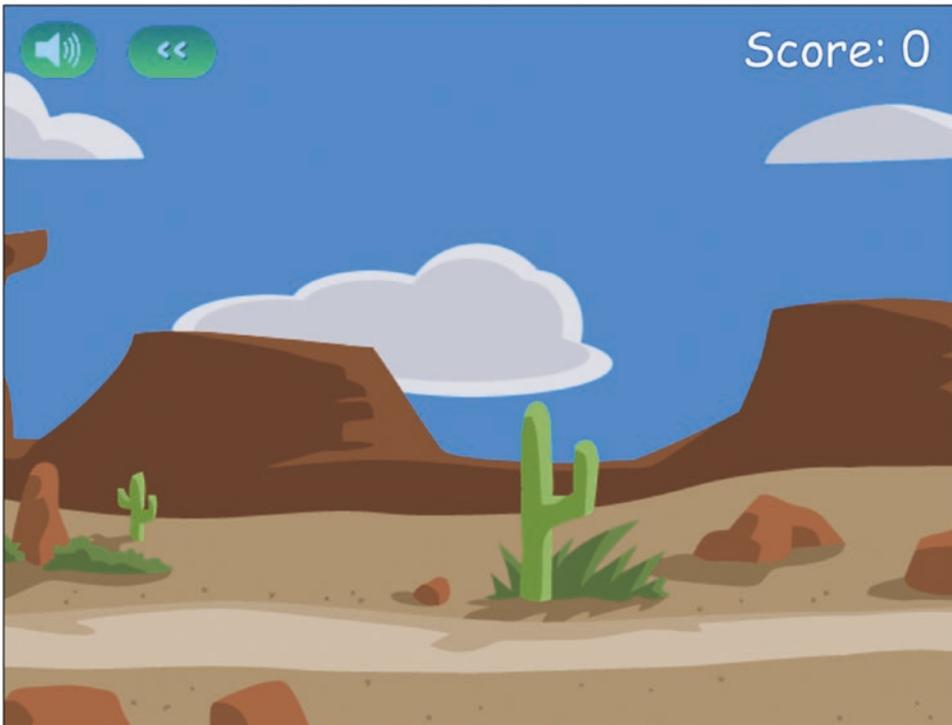
Figure 2-6. *The final result: panning around the level*

We can also use the mouse to interact with the level, by clicking and holding the mouse on the right side of the screen to make the screen pan right and then releasing the mouse button to pan back to the left.

Summary

In this chapter we set out to develop the basic framework for our game.

We started by defining and implementing a splash screen and game menu. We then created a simple level system and an asset loader to dynamically load the images used by each level. We set up the game canvas and animation loop and implemented parallax scrolling to give the illusion of depth. We used game states to simplify our game flow and move around our level in an interesting way. Finally, we captured and used mouse events to let the player pan around the level.

At this point we have a basic game world that we can interact with, so we are ready to add the various game entities and game physics.

In the next chapter we will take a break from this game code to briefly explore the basics of the Box2D physics engine and see how it can be used to model typical game physics. We will also look at how to animate characters using data from the physics engine.

Once we have done this, in Chapter 4, we will integrate the Box2D engine with our existing game framework so that the game entities move realistically within our game world, after which we can actually start playing the game.

■ ■ ■

Physics Engine Basics

A physics engine is a program that provides an approximate simulation of a game world by creating a mathematical model for all the object interactions and collisions within the game. It accounts for gravity, elasticity, friction, and conservation of momentum between colliding objects so that the objects move in a believable way. For our game, we are going to be using an existing and very popular physics engine called Box2D.

The Box2D engine is a free, open source physics engine that was originally written in C++ by Erin Catto. It has been used in a lot of popular physics-based games, including *Crayon Physics Deluxe*, *Rolando*, and *Angry Birds*. The engine has since been ported to several other languages, including Java, ActionScript, C#, and JavaScript. We will be using a JavaScript port of Box2D known as Box2dWeb. You can find the latest source code and documentation for Box2dWeb at `https://github.com/hecht-software/box2dweb`.

Before we start integrating the engine into our own game, let's go over some of the basic components of Box2D for creating and simulating worlds.

Box2D Fundamentals

Box2D uses a few basic objects to define and simulate the game world. The most important of these objects are as follows:

- *World*: The main Box2D object that contains all the world objects and simulates the game physics.

- *Body*: A rigid body that may consist of one or more shapes attached to the body via fixtures.

- *Shape*: A two-dimensional shape such as a circle or a polygon, which are the fundamental shapes used within Box2D.

- *Fixture*: Used to attach a shape to a body for collision detection. Fixtures hold additional, non-geometric data such as friction, collision, and filters.

- *Joint*: Used to constrain two bodies together in different ways. For example, a revolute joint constrains two bodies to share a common point while they are free to rotate about the point.

When using Box2D in our game, we first need to define the game world. We then add bodies and their corresponding shapes using fixtures. Once this is done, we step through the world and let Box2D move the bodies around. Finally, we draw the bodies after each step. Most of the heavy lifting is done by the Box2D world object.

Now let's look at these steps in more detail as we use Box2D to create a simple world.

Setting Up Box2D

We will start with a simple HTML file just like in the previous chapters (box2d-demo.html). The first thing we need to do is include a reference to the Box2dWeb library (Box2d.min.js) in the head section of the HTML file (see Listing 3-1).

Listing 3-1. Basic HTML5 File for Box2D (box2d-demo.html)

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">
        <title>Box2d Demo</title>
        <script src="Box2d.min.js" type="text/javascript"></script>
        <script src="box2d-demo.js" type="text/javascript"></script>
    </head>
    <body onload="init();">
        <canvas id="canvas" width="640" height="480" style="border: 1px solid black;">
            Your browser does not support HTML5 Canvas
        </canvas>
    </body>
</html>
```

As you see in Listing 3-1, the box2d.html file consists of only a single canvas element that we will be drawing on. We refer to two JavaScript files: the Box2dWeb library file and a second file that we will use to store all our JavaScript code (box2d-demo.js). Once the HTML file has loaded completely, it will call an init() function that we will use to initialize the Box2D world and start animating.

Referencing the Box2dWeb JavaScript file gives us access to the Box2D object in our JavaScript code. This object contains all the objects that we will need, including the world (Box2D.Dynamics.b2World) and the body (Box2D.Dynamics.b2Body).

It is convenient to define the commonly used objects as variables to save us some typing effort when we reference them. The first thing we will do in our JavaScript file (box2d-demo.js) is to declare these variables (see Listing 3-2).

Listing 3-2. Defining Commonly Used Objects as Variables

```
// Declare all the commonly used objects as variables for convenience
var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2BodyDef = Box2D.Dynamics.b2BodyDef;
var b2Body = Box2D.Dynamics.b2Body;
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
var b2World = Box2D.Dynamics.b2World;
var b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape;
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
var b2RevoluteJointDef = Box2D.Dynamics.Joints.b2RevoluteJointDef;
```

Once we define these variables as shortcuts, we can access the Box2D.Dynamics.b2World object by using the b2World variable. Now, let's start defining our world.

Defining the World

The `Box2D.Dynamics.b2World` object is the heart of Box2D. It contains methods for adding and removing objects, methods for simulating physics in incremental steps, and even an option for drawing the world on a canvas. Before we can start using Box2D, we need to create the `b2World` object. We do this in an `init()` function that we create inside our JavaScript file (`box2d-demo.js`), as shown in Listing 3-3.

Listing 3-3. Creating the b2World Object

```
var world;

//30 pixels on our canvas correspond to 1 meter in the box2d world
var scale = 30;

function init() {
    // Setup the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downwards
    // Allow objects that are at rest to fall asleep and be excluded from calculations
    var allowSleep = true;

    world = new b2World(gravity, allowSleep);
}
```

The `init()` function starts by defining `b2World` and passing the following two parameters to its constructor:

- `gravity`: Defined as a vector using a `b2Vec2` object, which takes two parameters, the x and y components. We set the world's gravity to be 9.8 meters per square second in the downward direction. The ability to set a custom gravity lets us simulate environments with different gravity fields, such as the moon or fantasy worlds with very low or very high gravity. We can also set `gravity` to 0 and use only the collision detection features of Box2D for games in which we don't need gravity (space-based games or top-down view games like racing games).

- `allowSleep`: Used by `b2World` to decide whether or not to include objects that are at rest during its simulation calculations. Allowing objects that are at rest to be excluded from calculations reduces the number of unnecessary calculations and thus helps improve performance. Even if an object is asleep, it will wake up if a moving body collides with it.

One other thing that we do within our code is define a `scale` variable that we will use to convert between Box2D units (meters) and our game units (pixels).

■ **Note**   Box2D uses the metric system for all its calculations. It works best with objects that are between 0.1 meter and 10 meters large. Since we use pixels when drawing on our canvas, we will need to convert between pixels and meters. A commonly used scale is 30 pixels to 1 meter.

Now that we have a basic world, we need to start adding bodies to it. The first body we will create is a static floor at the bottom of our world.

Adding Our First Body: The Floor

Creating a body in Box2D involves the following steps:

1. Declare a body definition in a b2BodyDef object. The b2BodyDef object contains details such as the position of the body (x and y coordinates) and the type of body (static or dynamic). Static bodies are not affected by gravity and collisions with other bodies and remain static, while dynamic bodies are affected by interactions with external forces and will fall, bounce, roll, and behave like typical objects in the real world.

2. Pass the body definition object to the createBody() method of the world and get back a body object.

3. Declare a fixture definition in a b2FixtureDef object. This is used to attach a shape to the body. A fixture definition also contains additional information such as density, friction coefficient, and the coefficient of restitution for the attached shape.

4. Set the shape of the fixture definition. The two types of shapes that are used in Box2D are polygons (b2PolygonShape) and circles (b2CircleShape). Pass the fixture definition to the createFixture() method of the body object and attach the shape to the body.

Now that we know these basic steps, we will create our first body inside the world: the floor. We will do this by creating a createFloor() method right below the init() function we created earlier. This is shown in Listing 3-4.

Listing 3-4. Creating the Floor

```
function createFloor() {
    // A body definition holds all the data needed to construct a rigid body
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_staticBody;
    bodyDef.position.x = 640 / 2 / scale;
    bodyDef.position.y = 450 / scale;

    // A fixture is used to attach a shape to a body for collision detection
    // A fixture definition is used to create a fixture
    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.2;

    fixtureDef.shape = new b2PolygonShape;
    fixtureDef.shape.SetAsBox(320 / scale, 10 / scale); // 640 pixels wide and 20 pixels tall

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

The first thing we do is define a bodyDef object. We set its type to be static (b2Body.b2_staticBody) since we want our floor to stay in the same place and not be affected by gravity or collisions with other bodies. We then set the position of the body near the bottom of our canvas (x = 320 pixels, y = 450 pixels) and use the scale variable to convert the pixels to meters for Box2D.

■ **Note**   Unlike the canvas, where the position of rectangles is based on the top-left corner, the Box2D body position is based on the origin of the object. In the case of boxes created using SetAsBox(), this origin is at the center of the box.

The next thing we do is define the fixture definition (fixtureDef). The fixture definition contains values like the density, the frictional coefficient, and the coefficient of restitution of its attached shape. The density is used to calculate the weight of the body, the frictional coefficient is used to make sure the body slides realistically, and the restitution is used to make the body bounce.

■ **Note**   The higher the coefficient of restitution, the more "bouncy" the object becomes. Values close to 0 mean that the body will not bounce and will lose most of its momentum in a collision (called an inelastic collision). Values close to 1 mean that the body retains most of its momentum and will bounce back as fast as it came (called an elastic collision).

We then set the shape for the fixture as a b2PolygonShape object. The b2PolygonShape object has a helper method called SetAsBox() that sets the polygon as a box which is centered on the origin of the parent body. The SetAsBox() method takes the half-width and half-height (the extents) of the box as parameters. Again, we use the scale variable to define a box that is 640 pixels wide and 20 pixels high.

Finally, we create the body by passing bodyDef to world.CreateBody() and create the fixture by passing the fixtureDef to body.CreateFixture().

One other thing we need to do is call this newly created method from inside the init() function we declared earlier so that this body is created when the init() function is called, as shown in Listing 3-5.

Listing 3-5. Calling createFloor() from init()

```
function init() {
    // Setup the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downwards

    // Allow objects that are at rest to fall asleep and be excluded from calculations
    var allowSleep = true;

    world = new b2World(gravity, allowSleep);

    createFloor();
}
```

Now that we have added our first body to the world, we need to learn how to draw the world so that we can see what we have created so far.

Drawing the World: Setting Up Debug Drawing

Box2D is primarily meant to be an engine that handles physics calculations, while we are expected to handle drawing all the objects in the world ourselves. However, the Box2D world object provides us with a simple DrawDebugData() method that we can use to draw the world on a given canvas for debugging and testing purposes.

The DrawDebugData() method draws a very simple representation of the bodies inside the world and is best used for helping us visualize the world while we are creating it.

Before we can use DrawDebugData(), we need to set up debug drawing by defining a b2DebugDraw() object and passing it to the world.SetDebugDraw() method. We do this in a setupDebugDraw() method that we will place below the createFloor() method inside box2d-demo.js (see Listing 3-6).

Listing 3-6. Setting Up Debug Drawing

```
var context;

function setupDebugDraw() {
    context = document.getElementById("canvas").getContext("2d");

    var debugDraw = new b2DebugDraw();

    // Use this canvas context for drawing the debugging screen
    debugDraw.SetSprite(context);
    // Set the scale
    debugDraw.SetDrawScale(scale);
    // Fill boxes with an alpha transparency of 0.3
    debugDraw.SetFillAlpha(0.3);
    // Draw lines with a thickness of 1
    debugDraw.SetLineThickness(1.0);
    // Display all shapes and joints
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);

    // Start using debug draw in our world
    world.SetDebugDraw(debugDraw);
}
```

We first define a handle to the canvas context using the getContext() method that you have previously seen.

We then create a new b2DebugDraw object and set a few attributes using its Set methods:

- SetSprite(): Used to provide a canvas context for the drawing.

- SetDrawScale(): Used to set the scale to convert between Box2D units and pixels.

- SetFillAlpha() and SetLineThickness(): Used to set drawing styles.

- SetFlags(): Used to choose which Box2D entities to draw. We have selected flags for drawing all shapes and joints, and we use logical OR operators to combine the two flags. Some of the other entities we can ask Box2D to draw are the center of mass (e_centerOfMassBit) and axis-aligned bounding boxes (e_aabbBit).

Finally, we pass the `debugDraw` object to the `world.SetDebugDraw()` method. After creating the function, we need to call it from inside the `init()` function as shown in Listing 3-7.

Listing 3-7. Calling setupDebugDraw() from init()

```
var allowSleep = true;

world = new b2World(gravity, allowSleep);

createFloor();

setupDebugDraw();
```

Now that debug drawing is set up, we can use the `world.DrawDebugData()` method to draw the current state of our Box2D world onto the canvas.

Animating the World

Animating a world using Box2D involves the following steps that we repeat within an animation loop:

1. Tell Box2D to run the simulation for a small time step (typically 1/60th of a second). We do this by using the `world.Step()` function.

2. Draw all the objects in their new positions using either `world.DrawDebugData()` or our own drawing functions.

3. Clear any forces that we have applied using `world.ClearForces()`.

We can implement these steps in our own `animate()` function that we create inside `box2d-demo.js` after `init()`, shown in Listing 3-8.

Listing 3-8. Setting Up a Box2D Animation Loop

```
var timeStep = 1 / 60;

//As per the Box2d manual, the suggested iteration count for Box2D is 8 for velocity and 3
for position
var velocityIterations = 8;
var positionIterations = 3;

function animate() {
    world.Step(timeStep, velocityIterations, positionIterations);
    world.ClearForces();

    world.DrawDebugData();

    setTimeout(animate, timeStep);
}
```

We first call `world.Step()` and pass it three parameters: time step, velocity iterations, and position iterations.

Box2D uses a computational algorithm called an *integrator*. Integrators simulate the physics equations at discrete points of time. The time step is the amount of time we want Box2D to simulate. We set this to a value of 1/60th of a second.

In addition to the integrator, Box2D also uses a larger bit of code called a *constraint solver*. The constraint solver solves all the constraints in the simulation, one at a time. To get a good solution, we need to iterate over all constraints a number of times. There are two phases in the constraint solver: a velocity phase and a position phase. Each phase has a separate iteration count, and we set these two values to 8 and 3, respectively.

■ **Note**   Generally, physics engines for games work well with a time step at least as fast as 60Hz or 1/60 second. As per Erin Catto's original C++ *Box2D v2.2.0 User Manual* (available at http://box2d.org/manual. pdf), it is preferable to keep the time step constant and not vary it with frame rate, as a variable time step produces variable results, which makes it difficult to debug.

Also as per the Box2d C++ manual, the suggested iteration count for Box2D is 8 for velocity and 3 for position. You can tune these numbers to your liking, but keep in mind that this has a trade-off between speed and accuracy. Using a lower iteration count increases performance but reduces accuracy. Likewise, using a higher iteration count decreases performance but improves the quality of your simulation.

After stepping through the simulation, we call `world.ClearForces()` to clear any forces that are applied to the bodies. We then call `world.DrawDebugData()` to draw the world on the canvas.

Finally, we use `setTimeout()` to call our animation loop again after the timeout for the next time step. We use `setTimeout()` for now because it is simpler for us to use the `Box2d.Step()` function with a constant frame rate. In the next chapter, we will look at how to use `requestAnimationFrame()` and a variable frame rate when integrating Box2D with our game.

Now that the animation loop is complete, we can see the world we have created so far by calling `animate()` from the `init()` function to start the animation loop, as shown in Listing 3-9.

Listing 3-9. Calling animate() from the init() Function

```
world = new b2World(gravity, allowSleep);

createFloor();

setupDebugDraw();

// Start the Box2D animation loop
animate();
```

When we open `box2d.html` in the browser, we should see our world with the floor drawn, as shown in Figure 3-1.
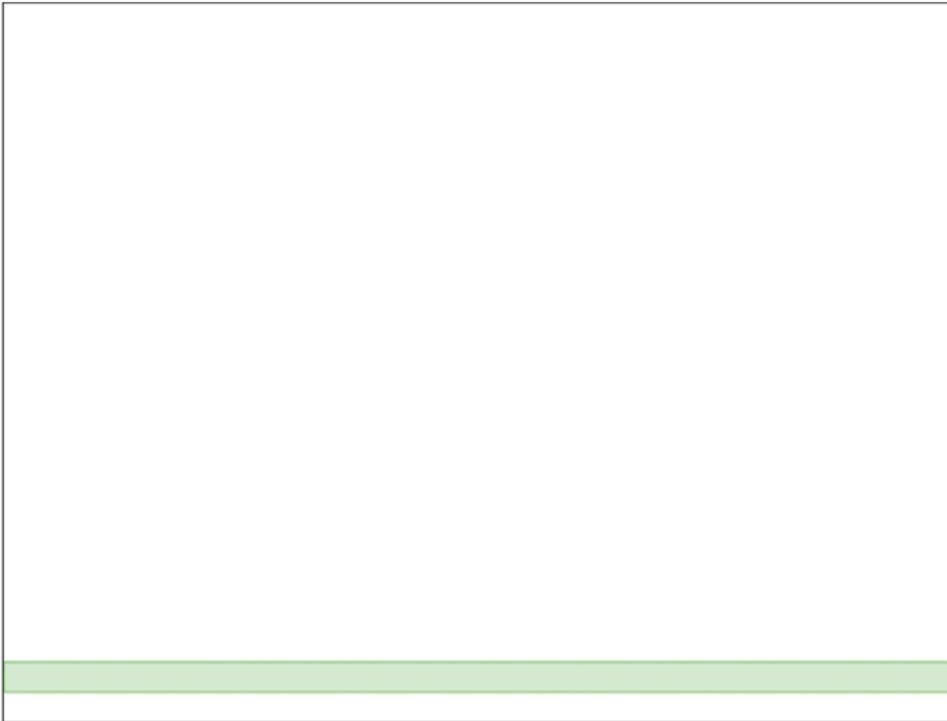
Figure 3-1. *Our first Box2D body: the floor*

This doesn't look like much yet. The floor is a static body that just stays floating at the bottom of the canvas. However, now that we have set up everything to create our basic world and display it on the screen, we can start adding some more Box2D elements to our world.

Adding More Box2D Elements

Box2D allows us to add different types of elements to our world, including the following:

- Simple bodies that are rectangular, circular, or polygon shaped

- Complex bodies that combine multiple shapes

- Joints such as revolute joints that connect multiple bodies

- Contact listeners that allow us to handle collision events

We will now look at each of these elements in turn in more detail.

Creating a Rectangular Body

We can create a rectangular body just like we created our floor—by defining a b2PolygonShape and using its SetAsBox() method. We will do this within a new method called createRectangularBody() that we will add to box2d-demo.js (see Listing 3-10).

Listing 3-10. Creating a Rectangular Body

```
function createRectangularBody() {
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 40 / scale;
    bodyDef.position.y = 100 / scale;

    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.3;

    fixtureDef.shape = new b2PolygonShape;
    fixtureDef.shape.SetAsBox(30 / scale, 50 / scale);

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

We create a body definition and place it near the top of the canvas at x = 40 pixels and y = 100 pixels. The one difference this time is that we define the body type as dynamic (b2Body.b2_dynamicBody). This means that the body will be affected by gravity and collisions. We then define the fixture with a polygon shape that is set as a box that is 60 pixels wide and 100 pixels tall. Again, note that we specify half-values, 30 and 50, in the SetAsBox() method. Finally, we add the body to our world.

We will need to add a call to createRectangularBody() inside the init() function so that it is called when the page loads. The init() function will now look like Listing 3-11.

Listing 3-11. Calling createRectangularBody() from init()

```
function init() {
    // Setup the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downwards

    // Allow objects that are at rest to fall asleep and be excluded from calculations
    var allowSleep = true;

    world = new b2World(gravity, allowSleep);

    createFloor();

    // Create some bodies with simple shapes
    createRectangularBody();

    setupDebugDraw();

    // Start the Box2D animation loop
    animate();
}
```

When we run the code in the browser, we should see the new body that we just created, as shown in Figure 3-2.
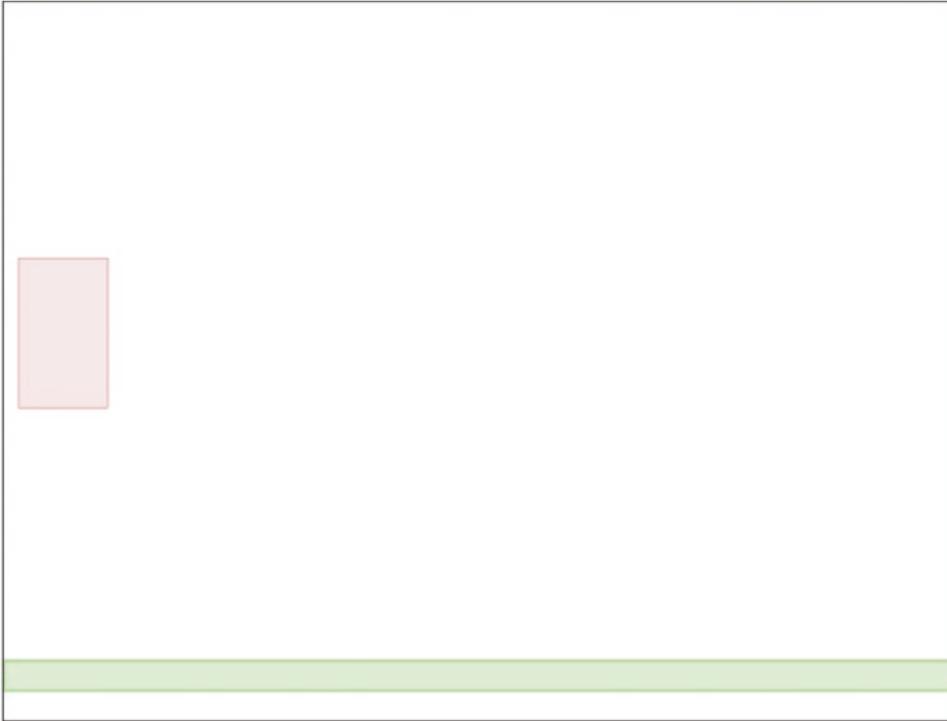
Figure 3-2. *Our first dynamic body: a bouncing rectangle*

Since this body is dynamic, it will fall downward because of gravity until it hits the floor, and then it will bounce off the floor. The body rises to a lower height after each bounce until it finally settles down on the floor. If we want, we can change the coefficient of restitution to decide how bouncy the object is.

■ **Note**   Once the body comes to rest, Box2D changes the color of the body and makes it darker. This is how Box2D tells us that the object is considered asleep. Box2D will wake up a body if another body collides with it.

Creating a Circular Body

The next body we will create is a simple circular body. We can define a circular shape by setting the shape property to a b2CircleShape object. We will do this within a new method called createCircularBody() that we will add to box2d-demo.js, as shown in Listing 3-12.

Listing 3-12. Creating a Circular Shape

```
function createCircularBody() {
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 130 / scale;
    bodyDef.position.y = 100 / scale;

    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.7;

    fixtureDef.shape = new b2CircleShape(30 / scale);

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

The b2CircleShape constructor takes one parameter, the radius of the circle. The rest of the code, defining a body, defining the fixture, and creating the body, remains very similar to the code for the rectangular body.

One change we have made is to increase the restitution value to 0.7, which is much higher than the value we used for our previous rectangular body. We need to call createCircularBody() from inside the init() function, right after createRectangularBody() as shown in Listing 3-13.

Listing 3-13. Calling createCircularBody() from init()

```
// Create some bodies with simple shapes
createRectangularBody();
createCircularBody();
```

Once we do this and run the code, we should see the new circular body that we just created (as shown in Figure 3-3).
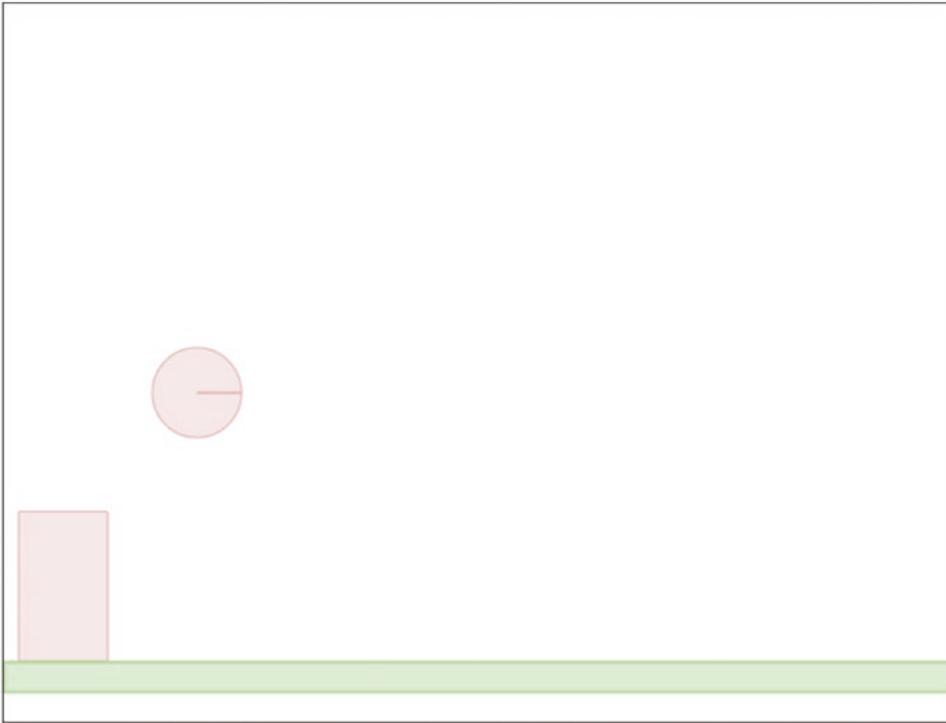
Figure 3-3. *A bouncier circular body*

You will notice that the circular body bounces much higher than the rectangular one, and takes a longer time to come to rest. This is because of the larger coefficient of restitution. If you set this value to 1, the ball will bounce back to the same height and never stop bouncing. If you choose a value greater than 1, the ball will go higher after each bounce, and eventually fly outside the screen.

Typically, a higher coefficient of restitution and lower gravity gives the game a spacey, sci-fi feel, while going in the opposite direction makes the game feel more realistic and grounded.

When creating your own game, you should play around with these values and tweak them until they feel right for your game.

Creating a Polygon-Shaped Body

The last simple shape we will create is the polygon. Box2D allows us to create any polygon we want by defining the coordinates of each of the points. The only restriction is that polygons need to be convex polygons (that is, no internal angle can be more than 180 degrees).

To create a polygon, we first need to create an array of b2Vec2 objects with the coordinates of each of its points, and then we need to pass the array to the shape.SetAsArray() method. We will do this within a new method called createSimplePolygonBody() that we will add to box2d-demo.js (see Listing 3-14).

Listing 3-14. Defining a Polygon Shape with Points

```
function createSimplePolygonBody() {
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 230 / scale;
    bodyDef.position.y = 50 / scale;

    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.6;

    fixtureDef.shape = new b2PolygonShape;
    // Create an array of b2Vec2 points in clockwise direction
    var points = [
        new b2Vec2(0, 0),
        new b2Vec2(40 / scale, 50 / scale),
        new b2Vec2(50 / scale, 100 / scale),
        new b2Vec2(-50 / scale, 100 / scale),
        new b2Vec2(-40 / scale, 50 / scale),

    ];

    // Use SetAsArray() to define the shape using the points array
    fixtureDef.shape.SetAsArray(points, points.length);

    var body = world.CreateBody(bodyDef);

    var fixture = body.CreateFixture(fixtureDef);
}
```

We defined a `points` array that contains the coordinates for each of the polygon points inside b2Vec2 objects. The following are a few things to note:

- All the coordinates are relative to the body origin. The first point (0,0) starts at the origin of the body and will be placed at the body position (230,50).

- We do not need to close out the polygon. Box2D will take care of this for us.

- All points must be defined in a clockwise direction.

■ **Tip**   If we define the coordinates in the counter-clockwise direction, Box2D will not be able to handle collisions correctly. If you find objects passing through each other, check to see whether you have defined points in the clockwise direction.

We then call the `SetAsArray()` method and pass it two parameters: the `points` array and the number of points. The rest of the code remains the same as it was for the previous shapes we covered.

Now we need to call `createSimplePolygonBody()` from the `init()` function as shown in Listing 3-15.

Listing 3-15. Calling createSimplePolygonBody() from init()

```
// Create some bodies with simple shapes
createRectangularBody();
createCircularBody();
createSimplePolygonBody();
```

If we run this code, we should see our new polygon-shaped body (see Figure 3-4).
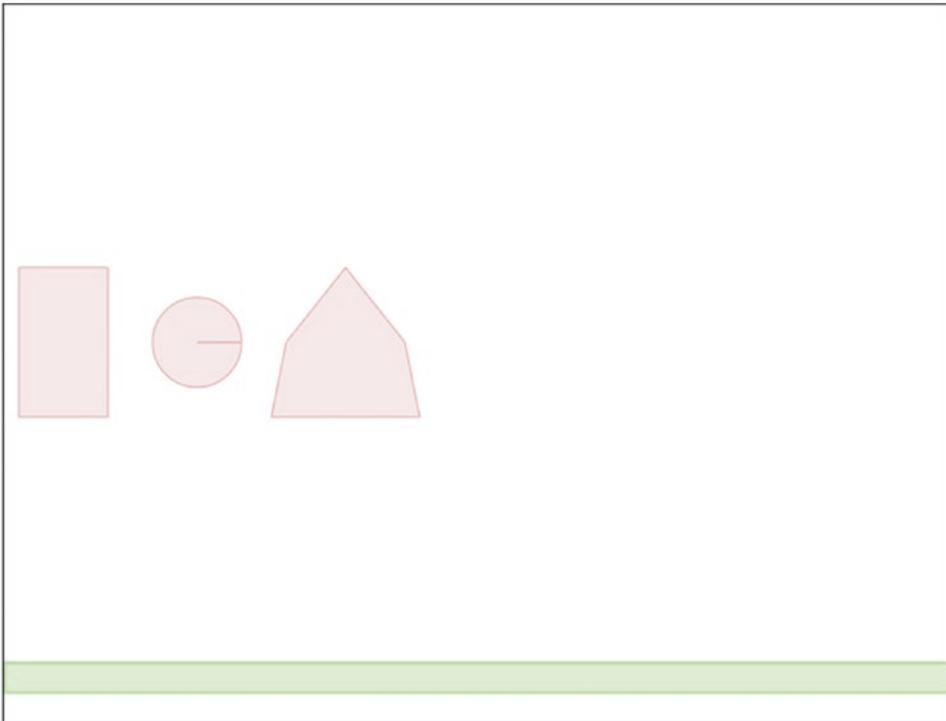
Figure 3-4. *A polygon-shaped body*

We now have created three simple bodies, with different shapes and properties. These simple shapes are usually enough to model a wide array of objects within our games (fruits, tires, crates, and so forth). Sometimes, however, these shapes are not enough. There are times when we need to create more complex objects that combine more than one shape.

Creating Complex Bodies with Multiple Shapes

So far we have been creating simple bodies with a single shape. However, as previously mentioned, Box2D lets us create bodies that contain multiple shapes.

To create a complex shape, all we need to do is attach multiple fixtures (each with its own shape) to the same body. Let's try to combine two of the shapes we just covered into a single body: a circle and a polygon. We will do this within a new method called `createComplexBody()` that we will add to `box2d-demo.js` (see Listing 3-16).

Listing 3-16. Creating a Body with Two Shapes

```
function createComplexBody() {
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 350 / scale;
    bodyDef.position.y = 50 / scale;
    var body = world.CreateBody(bodyDef);

    // Create first fixture and attach a circular shape to the body
    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.7;
    fixtureDef.shape = new b2CircleShape(40 / scale);
    body.CreateFixture(fixtureDef);

    // Create second fixture and attach a polygon shape to the body.
    fixtureDef.shape = new b2PolygonShape;
    var points = [
        new b2Vec2(0, 0),
        new b2Vec2(40 / scale, 50 / scale),
        new b2Vec2(50 / scale, 100 / scale),
        new b2Vec2(-50 / scale, 100 / scale),
        new b2Vec2(-40 / scale, 50 / scale),
    ];

    fixtureDef.shape.SetAsArray(points, points.length);
    body.CreateFixture(fixtureDef);
}
```

We first create a body, and then two different fixtures—the first for a circular shape and the second for a polygon shape. We then attach both these fixtures to the body using the `CreateFixture()` method. Box2D will automatically take care of creating a single rigid body that includes both these shapes.

One thing you might have noticed is that we reused the `fixtureDef` object for creating both the shape fixtures, and only changed its `shape` property. Reusing the object saves us the effort of setting properties like `density` and `restitution` again.

Now that we have created `createComplexBody()`, we need to call it from inside the `init()` function as shown in Listing 3-17.

Listing 3-17. Calling createComplexBody() from init()

```
// Create some bodies with simple shapes
createRectangularBody();
createCircularBody();
createSimplePolygonBody();

// Create a body combining two shapes
createComplexBody();
```

When we run this code, we should see our new complex body, as shown in Figure 3-5.
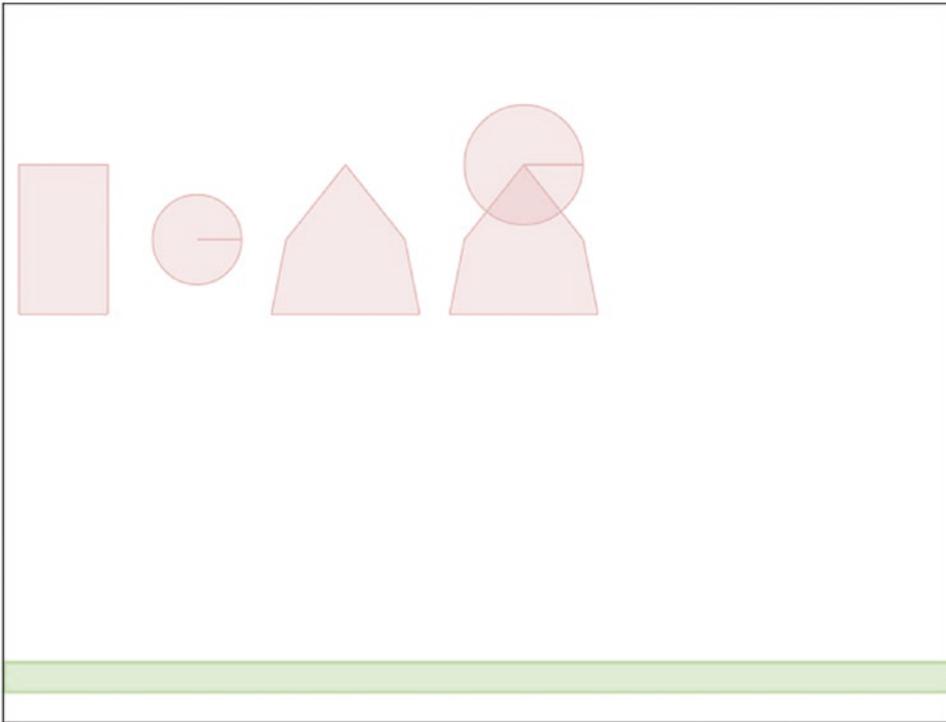
Figure 3-5. *A complex body with two shapes*

You will notice that the two shapes behave as one single unit. This is because Box2D treats these multiple shapes as a single rigid body. This ability to combine shapes allows us to emulate all kinds of object with complex shapes, such as trees and tables. It also allows us to get around the limitations on creating concave polygon shapes, since any concave polygon can be broken into multiple convex polygons.

Connecting Bodies with Joints

Now that we've explored how to make different types of bodies in Box2D, we will take a brief look at creating joints.

Joints are used to constrain bodies to the world or to each other. Box2D supports many different types of joints, including pulley, gear, distance, revolute, and weld joints.

Some of these joints restrict motion (for example, the distance joint and the weld joint), while others allow for interesting types of movement (for example, the pulley joint and the revolute joint). Some joints even provide motors that can be used to drive the joint at a specified speed. We will take a look at one of the simpler joints that Box2D offers: the revolute joint.

The revolute joint forces two bodies to share a common anchor point, often called a hinge point. What this means is that the bodies are attached to each other at this point, and can rotate about that point.

We can create a revolute joint by defining a b2RevoluteJointDef object and then passing it to the world.CreateJoint() method. This is illustrated in the createRevoluteJoint() method that we add to box2d-demo.js (see Listing 3-18).

Listing 3-18. Creating a Revolute Joint

```
function createRevoluteJoint() {
    // Define the first body
    var bodyDef1 = new b2BodyDef;

    bodyDef1.type = b2Body.b2_dynamicBody;
    bodyDef1.position.x = 480 / scale;
    bodyDef1.position.y = 50 / scale;
    var body1 = world.CreateBody(bodyDef1);

    // Create first fixture and attach a rectangular shape to the body
    var fixtureDef1 = new b2FixtureDef;

    fixtureDef1.density = 1.0;
    fixtureDef1.friction = 0.5;
    fixtureDef1.restitution = 0.5;
    fixtureDef1.shape = new b2PolygonShape;
    fixtureDef1.shape.SetAsBox(50 / scale, 10 / scale);

    body1.CreateFixture(fixtureDef1);

    // Define the second body
    var bodyDef2 = new b2BodyDef;

    bodyDef2.type = b2Body.b2_dynamicBody;
    bodyDef2.position.x = 470 / scale;
    bodyDef2.position.y = 50 / scale;
    var body2 = world.CreateBody(bodyDef2);

    // Create second fixture and attach a polygon shape to the body
    var fixtureDef2 = new b2FixtureDef;

    fixtureDef2.density = 1.0;
    fixtureDef2.friction = 0.5;
    fixtureDef2.restitution = 0.5;
    fixtureDef2.shape = new b2PolygonShape;
    var points = [
        new b2Vec2(0, 0),
        new b2Vec2(40 / scale, 50 / scale),
        new b2Vec2(50 / scale, 100 / scale),
        new b2Vec2(-50 / scale, 100 / scale),
        new b2Vec2(-40 / scale, 50 / scale),
    ];

    fixtureDef2.shape.SetAsArray(points, points.length);
    body2.CreateFixture(fixtureDef2);
```

```
    // Create a joint between body1 and body2
    var jointDef = new b2RevoluteJointDef;
    var jointCenter = new b2Vec2(470 / scale, 50 / scale);

    jointDef.Initialize(body1, body2, jointCenter);
    world.CreateJoint(jointDef);
}
```

In this code we first define two bodies, a rectangle (body1) and a polygon (body2), that are positioned on top of each other, and then add them to the world.

We then create a b2RevolutionJointDef object and initialize it by passing three parameters to the Initialize() method: the two bodies (body1 and body2), and the joint center, which is the point around which the joints rotate.

Note that the joint center is specified in Box2D world coordinates (the same coordinate system used to specify the location for the two bodies). Also note that the joint center is placed at a point that is located within both the bodies.

Finally, we call world.CreateJoint() to add the joint to the world.

We need to call createRevoluteJoint() from our init() function, as shown in Listing 3-19.

Listing 3-19. Calling createRevoluteJoint() from init()

```
// Create a body combining two shapes
createComplexBody();

// Join two bodies using a revolute joint
createRevoluteJoint();
```

When we run our code, we should see our revolute joint in action. You can see this in Figure 3-6.
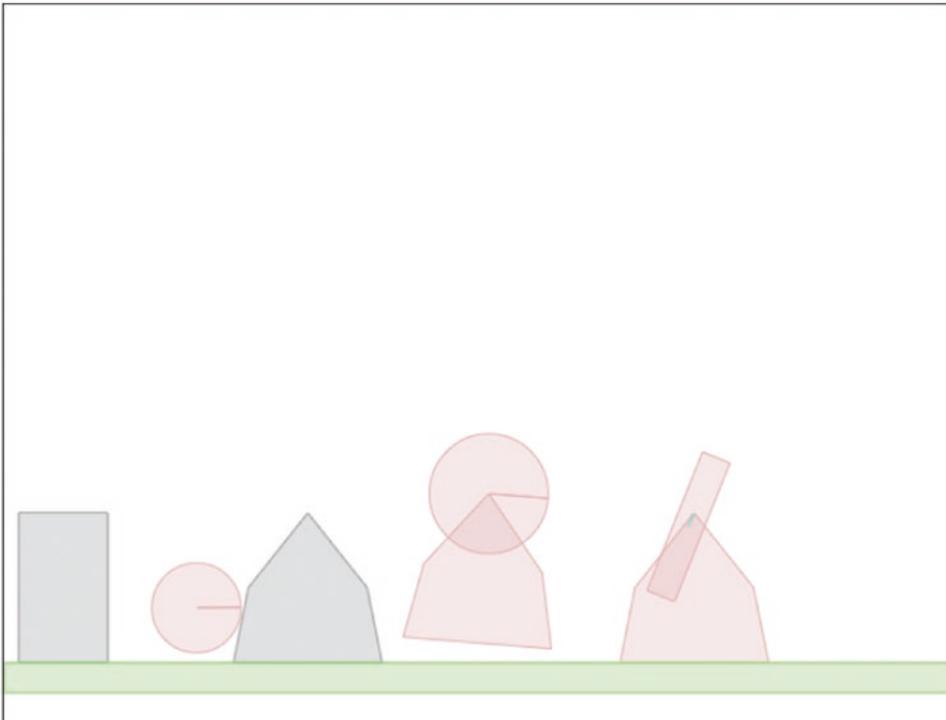
Figure 3-6. *A revolute joint in action*

As you can see, the rectangular body rotates about its anchor point, almost like a windmill blade. This is very different from the complex body we created earlier, where the shapes acted like a single body.

Each of the joints in Box2D can be combined in different ways to create interesting motions and effects, such as pulleys, ragdolls, and pendulums. You can read more about these other types of joints in the Box2D reference API, which you can find at `www.box2dflash.org/docs/2.1a/reference/`. Note that this documentation is for the Flash version of Box2D that our JavaScript version is based on. We can still refer to the method signatures and documentation in this Flash version when developing for the JavaScript version because the JavaScript version of Box2D was developed by directly converting the Flash version, and the method signatures remain the same across the two.

Tracking Collisions and Damage

One thing that you may have noticed in the previous few examples is that some of the bodies were colliding against each other and bouncing back and forth. It would be nice to be able to keep track of these collisions and the amount of impact they cause, and simulate a body getting damaged.

Before we can track the damage to an object, we need to be able to associate a life or health with it. Box2D provides us with methods that allow us to set custom properties for bodies, fixtures, or joints. We can assign any JavaScript object as a custom property for a body by calling its SetUserData() method, and retrieve the property later by calling its GetUserData() method.

Let's create another body that will have its own health unlike any of the previous bodies. We will do this inside a method called createSpecialBody() that we will add to box2d-demo.js (see Listing 3-20).

Listing 3-20. Creating a Special Body with Its Own Properties

```
var specialBody;

function createSpecialBody() {
    var bodyDef = new b2BodyDef;

    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 450 / scale;
    bodyDef.position.y = 0 / scale;

    specialBody = world.CreateBody(bodyDef);
    specialBody.SetUserData({ name: "special", life: 250 });

    // Create a fixture to attach a circular shape to the body
    var fixtureDef = new b2FixtureDef;

    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.5;

    fixtureDef.shape = new b2CircleShape(30 / scale);

    var fixture = specialBody.CreateFixture(fixtureDef);
}
```

The code for creating this body is similar to the code for the circular body that we looked at earlier. The only difference is that once we create the body, we call its SetUserData() method and pass it an object parameter with two custom properties, name and life.

We can add as many properties as we like to this object. Also, note that we saved a reference to the body in a variable called specialBody that we defined outside the function. This way, we can refer to this body even outside of the createSpecialBody() function.

If we call `createSpecialBody()` from the `init()` function, we won't see anything exceptional—just another bouncing circle. We still want to be able to track collisions happening to this body. This is where contact listeners come in.

Contact Listeners

Box2D provides us with objects called contact listeners that let us define event handlers for several contact-related events. To do this, we must first define a `b2ContactListener` object and override one or more of the events we want to monitor. The `b2ContactListener` has four events we can use based on what we need:

- `BeginContact()`: Called when two fixtures begin to touch.

- `EndContact()`: Called when two fixtures cease to touch.

- `PostSolve()`: Lets us inspect a contact after the solver is finished. This is useful for inspecting impulses.

- `PreSolve()`: Lets us inspect a contact before it goes to the solver.

Once we override the methods that we need, we need to pass the contact listener to the `world.SetContactListener()` method. Since we want to track the damage a collision causes, we will listen to the `PostSolve()` event, which provides us with the impulse transferred during a collision (see Listing 3-21).

Listing 3-21. Implementing a Contact Listener

```
function listenForContact() {
    var listener = new Box2D.Dynamics.b2ContactListener;

    listener.PostSolve = function(contact, impulse) {
        var body1 = contact.GetFixtureA().GetBody();
        var body2 = contact.GetFixtureB().GetBody();

        // If either of the bodies is the special body, reduce its life
        if (body1 == specialBody || body2 == specialBody) {
            var impulseAlongNormal = impulse.normalImpulses[0];

            specialBody.GetUserData().life -= impulseAlongNormal;
            console.log("The special body was in a collision with impulse", impulseAlongNormal,
            "and its life has now become ", specialBody.GetUserData().life);
        }
    };
    world.SetContactListener(listener);
}
```

As you can see, we create a `b2ContactListener` object and override its `PostSolve()` method with our own handler. The `PostSolve()` method provides us with two parameters: `contact`, which contains details of the fixtures that were involved in the collision, and `impulse`, which contains the normal and tangential impulse during the collision.

Within `PostSolve()`, we first extract the two bodies involved in the collision and check to see if our special body is one of them. If it is, we extract the impulse along the normal between the two bodies, and subtract life points from the body. We also log this event to the console so we can track each collision.

Obviously, this is a rather simplistic way of handling object damage, but it does what we need it to do. The greater the impulse in a collision, and the higher the number of collisions, the faster the body loses health.

■ **Note**   The `PostSolve()` method is called for every collision that takes place in the Box2D world, no matter how small. It is even called when an object is rolling on another. Be aware that this method will be called a lot.

Next we call both `createSimpleBody()` and `listenForContact()` from `init()` as shown in Listing 3-22.

Listing 3-22. Calling createSpecialBody() and listenForContact() from init()

```
// Join two bodies using a revolute joint
createRevoluteJoint();

// Create a body with special user data
createSpecialBody();

// Create contact listeners and track events
listenForContact();
```

If we run our code now, we should see the circle bouncing about, with a message in the browser console after each collision telling us how much the body's health has dropped, as shown in Figure 3-7.
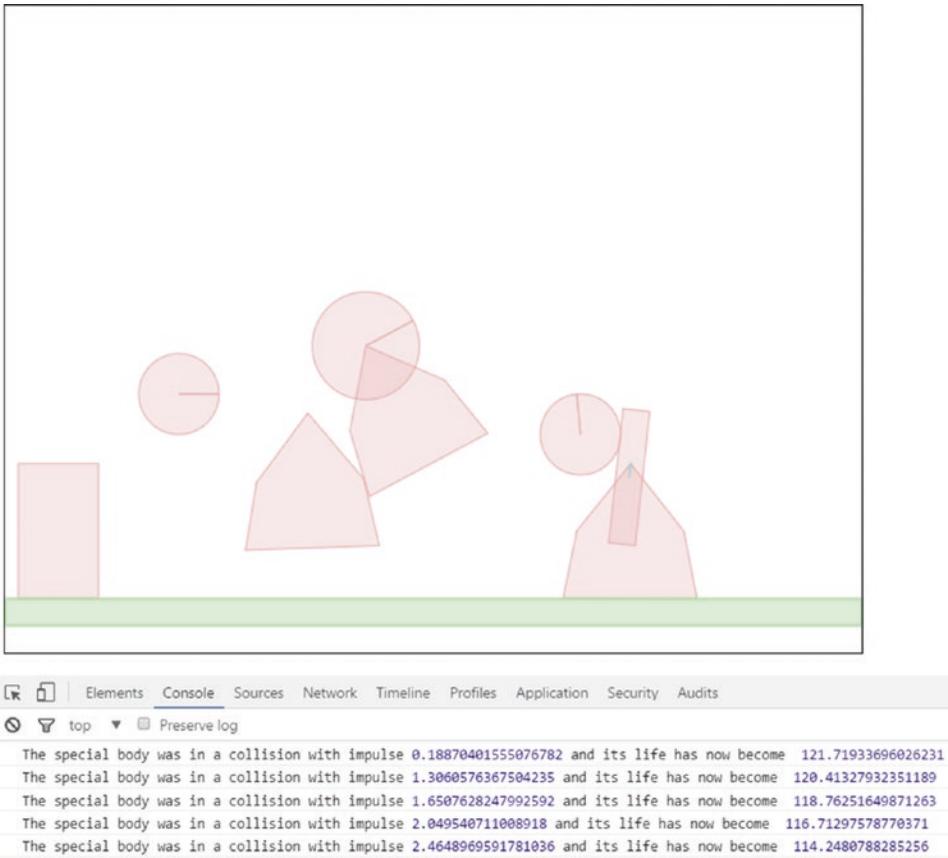
Figure 3-7. *Watching collisions with contact listeners*

It is nice to be able to track the life of our special body, but it would be nicer if we could do something when it runs out of life.

Now that we have access to specialBody and the life property, we can check after every iteration to see if the body life has reached 0 and, if so, remove it from the world using the world.DestroyBody() method. The easiest place to do this check is in the animate() method. The animate() function will now look like Listing 3-23.

Listing 3-23. Destroying the Body

```
function animate() {
    world.Step(timeStep, velocityIterations, positionIterations);
    world.ClearForces();

    world.DrawDebugData();

    // Kill Special Body if Dead
    if (specialBody && specialBody.GetUserData().life <= 0) {
        world.DestroyBody(specialBody);
        specialBody = undefined;
        console.log("The special body was destroyed");
    }

    setTimeout(animate, timeStep);
}
```

Once we finish calling world.Step() and drawing the world, we check to see whether specialBody is still defined and whether its life has reached 0. Once its life reaches 0, we remove the body from the world using DestroyBody() and then set specialBody to undefined.

This time when we run the code, the special body bounces around with its life dropping until it finally disappears. A message appears in the console telling us that the body was destroyed.

■ **Note**   We can track all the bodies and elements in a game using a similar principle by iterating through an array of objects. The point where we destroy a body is the perfect place for us to add explosion sounds or visual effects in a game and maybe update the score.

Drawing Our Own Characters

We have played with a lot of Box2D features so far. However, we have only been drawing using the default DrawDebugData() method. While this method is fine when testing code, we can't really write an amazing game looking like this. We need to know how to draw our own characters using all the drawing methods we covered in the first chapter.

Every b2Body object has two methods, GetPosition() and GetAngle(), that provide us with the coordinates and rotation of the body inside the Box2D world. Using the scale variable we defined in this chapter and the canvas translate() and rotate() methods we explored in Chapter 1, we can draw our characters or sprites on the canvas at the location that Box2D calculates for us.

To illustrate this, we can draw the special body that we have been playing with so far inside a drawSpecialBody() method that we will add to box2d-demo.js (see Listing 3-24).

Listing 3-24. Drawing Our Own Character

```
function drawSpecialBody() {
    // Get body position and angle
    var position = specialBody.GetPosition();
    var angle = specialBody.GetAngle();

    // Translate and rotate axis to body position and angle
    context.translate(position.x * scale, position.y * scale);
    context.rotate(angle);

    // Draw a filled circular face
    context.fillStyle = "rgb(200, 150, 250)";
    context.beginPath();
    context.arc(0, 0, 30, 0, 2 * Math.PI, false);
    context.fill();

    // Draw two rectangular eyes
    context.fillStyle = "rgb(255, 255, 255)";
    context.fillRect(-15, -15, 10, 5);
    context.fillRect(5, -15, 10, 5);

    // Draw an upward or downward arc for a smile depending on life
    context.strokeStyle = "rgb(255, 255, 255)";
    context.beginPath();
    if (specialBody.GetUserData().life > 100) {
        context.arc(0, 0, 10, Math.PI, 2 * Math.PI, true);
    } else {
        context.arc(0, 10, 10, Math.PI, 2 * Math.PI, false);
    }
    context.stroke();

    // Translate and rotate axis back to original position and angle
    context.rotate(-angle);
    context.translate(-position.x * scale, -position.y * scale);
}
```

We start by translating the canvas to the body's position and rotating the canvas to the body's angle. This is very similar to the code we looked at in Chapter 1.

We then draw a filled circle for the face, two rectangular eyes, and a smile using an arc. Just for fun, when the body life goes below 100, we change the smile to a sad face.

Finally, we undo the rotation and translation.

Before we can see this method in action, we will need to call it from inside animate(). The finished animate() method will now look like Listing 3-25.

Listing 3-25. The Finished animate() Method

```
function animate() {
    world.Step(timeStep, velocityIterations, positionIterations);
    world.ClearForces();

    world.DrawDebugData();
```

```
// Custom Drawing
if (specialBody) {
    drawSpecialBody();
}

// Kill Special Body if Dead
if (specialBody && specialBody.GetUserData().life <= 0) {
    world.DestroyBody(specialBody);
    specialBody = undefined;
    console.log("The special body was destroyed");
}

setTimeout(animate, timeStep);
}
```

What we have done here is check whether specialBody is still defined and call drawSpecialBody() if it is. Once the body dies, specialBody will become undefined and we will stop trying to draw it. You will notice that we draw after DrawDebugData() has completed, so we end up drawing on top of the debug drawing.

When we run this finished code, we see our new version of specialBody with a smiley face that becomes sad after a while before finally disappearing (see Figure 3-8).
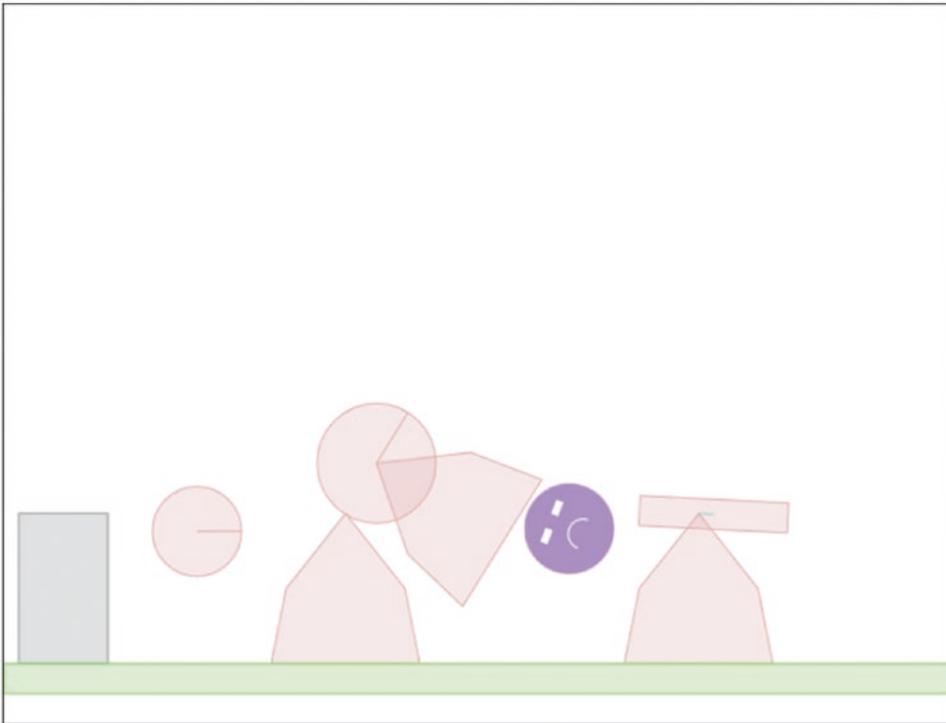
Figure 3-8. *Drawing our own character*

We have just animated our own character using the Box2D engine. This may not seem like much, but we now have all the building blocks that we need to build games using Box2D.

When you create your own game, you won't just be playing with boxes and circles. You will still use simple shapes that are similar in appearance to your game elements so that they seem to move realistically. However, you will be drawing all the characters yourself instead of using debug drawing, which means you can now use all of the methods that you learned about in Chapter 1, including drawings sprites, to create any desired effect for your characters.

Summary

In this chapter we took a crash course on the Box2D engine. We created a world in Box2D and drew different kinds of bodies within it. We made simple circular and rectangular shapes, polygons, and complex bodies that combined multiple shapes, and we used joints to combine shapes.

We animated the world realistically by letting Box2D handle the physics computations and drawing the world using `DrawDebugData()`. We used contact listeners to track collisions and slowly damage and destroy objects within the world. Finally, we drew our own character that was moved by Box2D.

We covered most of the elements of Box2D that we will be using in our game. If you would like to dive deeper into the Box2D API, you can look at the API reference available at `www.box2dflash.org/docs/`. You can also read the Box2D guide available at the same site.

In the next chapter, we will combine everything that we have learned so far to integrate Box2D into our game. We will create a framework to handle creation of our game entities inside Box2D. We will then use images and sprites to draw our characters over the parallax scrolling backgrounds that we built in Chapter 2. After that, we will spend some time polishing up our game by adding sound effects, and then wire everything together to create a finished, physics-based puzzle game.

CHAPTER 4

■ ■ ■

Integrating the Physics Engine

In Chapter 2, we developed the basic framework for our game, Froot Wars, and in Chapter 3, we looked at how to simulate a game world in Box2D. Now it is time to put together all the pieces to complete our game.

In this chapter, we will continue where we left off at the end of Chapter 2. We will add entities to our levels, use Box2D to simulate these entities, and then animate these entities within the game. We will use these entities to create a couple of working levels, and we will add mouse interactivity so that we can play the game. Once we have a working game, we will add sounds, background music, and a few other finishing touches to wrap up our game.

Now let's get started. We will be using the code from Chapter 2 as our starting point.

Defining Entities

So far, our game levels contain data for the background and foreground images and an empty array for entities. This entities array will eventually contain all the entities within our game: the heroes, the villains, the ground, and the blocks used to create the environment. We will then use this array to ask Box2D to create the corresponding Box2D shapes.

Typical entities will look like the examples shown in Listing 4-1.

Listing 4-1. Typical Entities

```
{ type: "ground", name: "dirt", x: 500, y: 440, width: 1000, height: 20, isStatic: true },
{ type: "block", name: "wood", x: 500, y: 375, angle: 90, width: 100, height: 25 },
{ type: "hero", name: "orange", x: 90, y: 410 },
{ type: "villain", name: "burger", x: 500, y: 200, calories: 590 },
```

The `type` property can contain values like `"hero"`, `"villain"`, `"ground"`, and `"block"`. We will use this property to decide how to handle an entity during creation and drawing operations.

The `x`, `y`, and `angle` properties are used to set the starting position and orientation of the entities.

The entity can also contain specific properties for its type, such as `calories`, which is the number of points scored when destroying a villain.

The `name` property tells us which sprite to use to draw the entity. All the images that we will use for the entities are stored in the `images/entities` folder.

The `name` property will also be used to refer to entity definitions. These definitions will include fixture data such as density and restitution, health data for destructible objects, and, in the case of heroes and villains, even details on the shape. Typical entity definitions will look like the examples shown in Listing 4-2.

Listing 4-2. Typical Entity Definitions

```
"wood": {
    fullHealth: 500,
    density: 0.7,
    friction: 0.4,
    restitution: 0.4,
},
"dirt": {
    density: 3.0,
    friction: 1.5,
    restitution: 0.2,
},
"burger": {
    shape: "circle",
    fullHealth: 40,
    radius: 25,
    density: 1,
    friction: 0.5,
    restitution: 0.4,
},
```

All the entity definitions contain the density, friction, and restitution necessary to model the game in Box2D. In the case of game characters, we store additional data for the shape and dimensions. Also note that the definitions contain a `fullHealth` property whenever we need a character or material to be destructible.

Now that we have decided how we will be storing the entities, we also need a way to create them. We will start by creating an `entities` object in `game.js` that will handle all entity-related operations in our game. This object will contain all the entity definitions as well as the methods for creating and drawing entities (see Listing 4-3).

Listing 4-3. The entities Object with Definitions for Entities

```
var entities = {
    definitions: {
        "glass": {
            fullHealth: 100,
            density: 2.4,
            friction: 0.4,
            restitution: 0.15
        },
        "wood": {
            fullHealth: 500,
            density: 0.7,
            friction: 0.4,
            restitution: 0.4
        },
        "dirt": {
            density: 3.0,
            friction: 1.5,
            restitution: 0.2
        },
```

```
    "burger": {
        shape: "circle",
        fullHealth: 40,
        radius: 25,
        density: 1,
        friction: 0.5,
        restitution: 0.4
    },
    "sodacan": {
        shape: "rectangle",
        fullHealth: 80,
        width: 40,
        height: 60,
        density: 1,
        friction: 0.5,
        restitution: 0.7
    },
    "fries": {
        shape: "rectangle",
        fullHealth: 50,
        width: 40,
        height: 50,
        density: 1,
        friction: 0.5,
        restitution: 0.6
    },
    "apple": {
        shape: "circle",
        radius: 25,
        density: 1.5,
        friction: 0.5,
        restitution: 0.4
    },
    "orange": {
        shape: "circle",
        radius: 25,
        density: 1.5,
        friction: 0.5,
        restitution: 0.4
    },
    "strawberry": {
        shape: "circle",
        radius: 15,
        density: 2.0,
        friction: 0.5,
        restitution: 0.4
    }
},
```

```
    // Take the entity, create a Box2D body, and add it to the world
    create: function(entity) {

    },

    // Take the entity, its position, and its angle and draw it on the game canvas
    draw: function(entity, position, angle) {

    }

};
```

The entities object contains an array with definitions for all the material types (glass, wood, and dirt) and definitions for all the heroes and villains that we will have in the game (orange, apple, and burger).

The values for some of these properties (such as size, restitution, and fullHealth) were decided based on feel, by constantly tweaking them in an effort to make the game as much fun as possible. The correct values for these properties will vary with each game you make.

We also have placeholders for the create() and draw() functions that we need to implement. However, before we can implement these, we need to add Box2D to our code.

Adding Box2D

The first thing we need to do is add a reference to Box2d.min.js in the <head> section of index.html before the reference to game.js. The <head> section of the file will now look like Listing 4-4.

Listing 4-4. Adding Box2D to the index.html <head> Section

```
 <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Froot Wars</title>
    <script src="js/Box2d.min.js" type="text/javascript"></script>
    <script src="js/game.js" type="text/javascript" charset="utf-8"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen">
</head>
```

One other thing that we will do is add references for all the commonly used Box2D objects to the beginning of game.js (see Listing 4-5).

Listing 4-5. Adding References to Commonly Used Box2D Objects

```
// Declare all the commonly used Box2D objects as variables for convenience
var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2BodyDef = Box2D.Dynamics.b2BodyDef;
var b2Body = Box2D.Dynamics.b2Body;
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
var b2World = Box2D.Dynamics.b2World;
var b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape;
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
var b2ContactListener = Box2D.Dynamics.b2ContactListener;
```

Now that we have the references set up, we can start using Box2D from within our game code. We will be creating a separate box2d object inside game.js to store all our Box2D-related methods (see Listing 4-6).

Listing 4-6. Creating a box2d Object

```
var box2d = {
    scale: 30,

    init: function() {
        // Set up the Box2D world that will do most of the physics calculation
        var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downward
        var allowSleep = true; // Allow objects that are at rest to fall asleep and be
        excluded from calculations

        box2d.world = new b2World(gravity, allowSleep);
    },

    createRectangle: function(entity, definition) {
        var bodyDef = new b2BodyDef();

        if (entity.isStatic) {
            bodyDef.type = b2Body.b2_staticBody;
        } else {
            bodyDef.type = b2Body.b2_dynamicBody;
        }

        bodyDef.position.x = entity.x / box2d.scale;
        bodyDef.position.y = entity.y / box2d.scale;
        if (entity.angle) {
            bodyDef.angle = Math.PI * entity.angle / 180;
        }

        var fixtureDef = new b2FixtureDef();

        fixtureDef.density = definition.density;
        fixtureDef.friction = definition.friction;
        fixtureDef.restitution = definition.restitution;

        fixtureDef.shape = new b2PolygonShape();
        fixtureDef.shape.SetAsBox(entity.width / 2 / box2d.scale, entity.height / 2 /
        box2d.scale);

        var body = box2d.world.CreateBody(bodyDef);

        body.SetUserData(entity);
        body.CreateFixture(fixtureDef);

        return body;
    },
```

```
    createCircle: function(entity, definition) {
        var bodyDef = new b2BodyDef();

        if (entity.isStatic) {
            bodyDef.type = b2Body.b2_staticBody;
        } else {
            bodyDef.type = b2Body.b2_dynamicBody;
        }

        bodyDef.position.x = entity.x / box2d.scale;
        bodyDef.position.y = entity.y / box2d.scale;

        if (entity.angle) {
            bodyDef.angle = Math.PI * entity.angle / 180;
        }
        var fixtureDef = new b2FixtureDef();

        fixtureDef.density = definition.density;
        fixtureDef.friction = definition.friction;
        fixtureDef.restitution = definition.restitution;

        fixtureDef.shape = new b2CircleShape(entity.radius / box2d.scale);

        var body = box2d.world.CreateBody(bodyDef);

        body.SetUserData(entity);
        body.CreateFixture(fixtureDef);

        return body;
    },

};
```

The box2d object contains an init() method where we initialize a new b2World object, just like we did in Chapter 3.

The object also contains two helper methods, createRectangle() and createCircle(). Both methods accept two parameters, the entity and definition objects that we described earlier. The entity object contains details about the entity we want to create, such as its position, angle, and whether or not the entity is static. The definition object contains details about the fixture, such as restitution and density.

Using these parameters, the methods create Box2D bodies and fixtures and add them to the Box2D world.

Finally, both the methods also attach the entity object to the body using the SetUserData() method. This enables us to retrieve any of the entity-related data for a Box2D body using its GetUserData() method.

One thing to note is that both these methods convert the position and size using box2d.scale and convert the angle from degrees to radians before they can be used by Box2D.

Creating Entities

Now that we have Box2D set up, we will implement the entities.create() method inside the entities object that we defined earlier. This method will take an entity object as a parameter and add it to the world (see Listing 4-7).

Listing 4-7. Defining the entities.create() Method

```
// Take the entity, create a Box2D body, and add it to the world
create: function(entity) {
    var definition = entities.definitions[entity.name];

    if (!definition) {
        console.log("Undefined entity name", entity.name);

        return;
    }

    switch(entity.type) {
        case "block": // simple rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.shape = "rectangle";
            entity.sprite = loader.loadImage("images/entities/" + entity.name + ".png");

            box2d.createRectangle(entity, definition);
            break;
        case "ground": // simple rectangles
                // No need for health. These are indestructible
            entity.shape = "rectangle";
                // No need for sprites. These won't be drawn at all
            box2d.createRectangle(entity, definition);
            break;
        case "hero":    // simple circles
        case "villain": // can be circles or rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.sprite = loader.loadImage("images/entities/" + entity.name + ".png");
            entity.shape = definition.shape;
            if (definition.shape === "circle") {
                entity.radius = definition.radius;
                box2d.createCircle(entity, definition);
            } else if (definition.shape === "rectangle") {
                entity.width = definition.width;
                entity.height = definition.height;
                box2d.createRectangle(entity, definition);
            }
            break;
        default:
            console.log("Undefined entity type", entity.type);
            break;
    }
},
```

In this method, we use the entity type to decide how to handle the `entity` object and its properties:

- *Block*: For `block` entities, we set the entity `health` and `fullHealth` properties based on the entity definition, and set the `shape` property to `"rectangle"`. We then load the sprite, and call the `box2d.createRectangle()` method.

- *Ground*: For ground entities, we set the `entity` object's `shape` property to `"rectangle"` and call the `box2d.createRectangle()` method. We do not load a sprite because we will be using the ground from the level foreground image and won't be drawing the ground separately.

- *Hero and villain*: For `hero` and `villain` entities, we set the entity `health`, `fullHealth`, and `shape` properties based on the entity definition. We then set either the `radius` or the `height` and `width` properties based on the shape of the entity. Finally, we call either `box2d.createRectangle()` or `box2d.createCircle()` based on the shape.

Now that we have a way to create entities, let's add some entities to our levels.

Adding Entities to Levels

The first thing we will do is add a few entities inside our `levels.data` array, as shown in Listing 4-8.

Listing 4-8. Adding Entities to the `levels.data` Array

```
// Level data
data: [{   // First level
    foreground: "desert-foreground",
    background: "clouds-background",
    entities: [
        // The ground
        { type: "ground", name: "dirt", x: 500, y: 440, width: 1000, height: 20,
        isStatic: true },
        // The slingshot wooden frame
        { type: "ground", name: "wood", x: 190, y: 390, width: 30, height: 80,
        isStatic: true },

        { type: "block", name: "wood", x: 500, y: 380, angle: 90, width: 100, height: 25 },
        { type: "block", name: "glass", x: 500, y: 280, angle: 90, width: 100, height: 25 },
        { type: "villain", name: "burger", x: 500, y: 205, calories: 590 },

        { type: "block", name: "wood", x: 800, y: 380, angle: 90, width: 100, height: 25 },
        { type: "block", name: "glass", x: 800, y: 280, angle: 90, width: 100, height: 25 },
        { type: "villain", name: "fries", x: 800, y: 205, calories: 420 },

        { type: "hero", name: "orange", x: 80, y: 405 },
        { type: "hero", name: "apple", x: 140, y: 405 }
    ]
}, {   // Second level
    foreground: "desert-foreground",
    background: "clouds-background",
```

```
    entities: [
        // The ground
        { type: "ground", name: "dirt", x: 500, y: 440, width: 1000, height: 20,
        isStatic: true },
        // The slingshot wooden frame
        { type: "ground", name: "wood", x: 190, y: 390, width: 30, height: 80,
        isStatic: true },

        { type: "block", name: "wood", x: 850, y: 380, angle: 90, width: 100, height: 25 },
        { type: "block", name: "wood", x: 700, y: 380, angle: 90, width: 100, height: 25 },
        { type: "block", name: "wood", x: 550, y: 380, angle: 90, width: 100, height: 25 },
        { type: "block", name: "glass", x: 625, y: 316, width: 150, height: 25 },
        { type: "block", name: "glass", x: 775, y: 316, width: 150, height: 25 },

        { type: "block", name: "glass", x: 625, y: 252, angle: 90, width: 100, height: 25 },
        { type: "block", name: "glass", x: 775, y: 252, angle: 90, width: 100, height: 25 },
        { type: "block", name: "wood", x: 700, y: 190, width: 150, height: 25 },

        { type: "villain", name: "burger", x: 700, y: 152, calories: 590 },
        { type: "villain", name: "fries", x: 625, y: 405, calories: 420 },
        { type: "villain", name: "sodacan", x: 775, y: 400, calories: 150 },

        { type: "hero", name: "strawberry", x: 30, y: 415 },
        { type: "hero", name: "orange", x: 80, y: 405 },
        { type: "hero", name: "apple", x: 140, y: 405 }
    ]
}],
```

The first level contains two ground entities—one for the floor and the other for the slingshot. These entities are meant to be static objects that are not drawn by us.

The level also contains four rectangular block entities (glass and wood). These are destructible elements that we have positioned using their angle, x, and y properties.

Finally, the level contains two hero entities (orange and apple) and two villain entities (burger and fries). Note that the villain entities have an extra property called calories, which we will be using to increase the player score whenever a villain has been destroyed.

The second level has a similar design, except with a few more entities.

Now that we have defined entities for each level, we need to load these entities when we load the level. To do this, we will modify the load() method of the levels object (see Listing 4-9).

Listing 4-9. Modifying levels.load() to Load the Entities

```
// Load all data and images for a specific level
load: function(number) {

    // Initialize Box2D world whenever a level is loaded
    box2d.init();

    // Declare a new currentLevel object
    game.currentLevel = { number: number, hero: [] };
    game.score = 0;
```

```
    document.getElementById("score").innerHTML = "Score: " + game.score;
    var level = levels.data[number];

    // Load the background, foreground, and slingshot images
    game.currentLevel.backgroundImage = loader.loadImage("images/backgrounds/" +
    level.background + ".png");
    game.currentLevel.foregroundImage = loader.loadImage("images/backgrounds/" +
    level.foreground + ".png");
    game.slingshotImage = loader.loadImage("images/slingshot.png");
    game.slingshotFrontImage = loader.loadImage("images/slingshot-front.png");

    // Load all the entities
    for (let i = level.entities.length - 1; i >= 0; i--) {
        var entity = level.entities[i];

        entities.create(entity);
    }

    // Call game.start() once the assets have loaded
    loader.onload = game.start;
}
```

The first change we have made is the addition of a call to `box2d.init()` at the very beginning of the method. This will create a new Box2D world for the level so that we can start adding our entities to it.

The other change is the addition of a `for` loop where we iterate through all the entities for a level and call `entities.create()` for each entity. Now when we load a level, Box2D will get initialized and all the entities will get loaded into the Box2D world.

We still can't see the bodies we have added. Let's use the Box2D debug drawing method introduced in Chapter 3 to see what we created.

Setting Up Box2D Debug Drawing

Our game doesn't strictly need debug drawing since we will be handling drawing the game world and entities ourselves. We will use debug drawing only temporarily, to help us design and test our levels. We can remove all traces of debug drawing once the game is complete. We will organize our debug drawing code so that it can easily be activated or deactivated within the code.

The first thing we will do is create a `box2d.setupDebugDraw()` method inside the `box2d` object for setting up debug drawing when we are initializing Box2D. We will then call this method from the `box2d.init()` method as shown in Listing 4-10.

Listing 4-10. Setting Up Debug Drawing

```
init: function() {
    // Set up the Box2D world that will do most of the physics calculation
    var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; // Allow objects that are at rest to fall asleep and be excluded
    from calculations
```

```
    box2d.world = new b2World(gravity, allowSleep);

    // Activate debug drawing. Comment the line below to disable it.
    this.setupDebugDraw();

},

debugCanvas: undefined,
setupDebugDraw: function() {
    // Dynamically create a canvas for the debug drawing
    if (!box2d.debugCanvas) {
        var canvas = document.createElement("canvas");

        canvas.width = 1024;
        canvas.height = 480;
        document.body.appendChild(canvas);
        canvas.style.top = "480px";
        canvas.style.position = "absolute";
        canvas.style.background = "white";
        box2d.debugCanvas = canvas;
    }

    // Set up debug draw
    var debugContext = box2d.debugCanvas.getContext("2d");
    var debugDraw = new b2DebugDraw();

    debugDraw.SetSprite(debugContext);
    debugDraw.SetDrawScale(box2d.scale);
    debugDraw.SetFillAlpha(0.3);
    debugDraw.SetLineThickness(1.0);
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);
    box2d.world.SetDebugDraw(debugDraw);
},
```

Within the method, we first use document.createElement() to create a new canvas object and append it to the document body. The canvas is sized to fit the entire level, and styled to be positioned below our game area, with a white background.

We then use the newly created debugCanvas property to set up the Box2D debug draw just like we did in Chapter 3.

Finally, we add a line in box2d.init() to call box2d.setupDebugDraw(). The advantage of doing everything in a single method like this is that we can remove all traces of debug draw just by commenting out this single line in box2d.init().

Before we can see the results of debug draw, we need to call the world object's DrawDebugData() method. We will do this in a new method called drawAllBodies() inside the game object, as shown in Listing 4-11. We will call this method from the animate() method of the game object.

Listing 4-11. Modifying animate() and Creating drawAllBodies()

```
animate: function() {

    // Handle panning, game states, and control flow
    game.handleGameLogic();

    //  Draw the background with parallax scrolling
    // First draw the background image, offset by a fraction of the offsetLeft distance (1/4)
    // The bigger the fraction, the closer the background appears to be
    game.context.drawImage(game.currentLevel.backgroundImage, game.offsetLeft / 4, 0,
    game.canvas.width, game.canvas.height, 0, 0, game.canvas.width, game.canvas.height);
    // Then draw the foreground image, offset by the entire offsetLeft distance
    game.context.drawImage(game.currentLevel.foregroundImage, game.offsetLeft, 0,
    game.canvas.width, game.canvas.height, 0, 0, game.canvas.width, game.canvas.height);

    // Draw the base of the slingshot, offset by the entire offsetLeft distance
    game.context.drawImage(game.slingshotImage, game.slingshotX - game.offsetLeft,
    game.slingshotY);

    // Draw all the bodies
    game.drawAllBodies();

    // Draw the front of the slingshot, offset by the entire offsetLeft distance
    game.context.drawImage(game.slingshotFrontImage, game.slingshotX - game.offsetLeft,
    game.slingshotY);

    if (!game.ended) {
        game.animationFrame = window.requestAnimationFrame(game.animate, game.canvas);
    }
},

drawAllBodies: function() {
    // Draw debug data if a debug canvas has been set up
    if (box2d.debugCanvas) {
        box2d.world.DrawDebugData();
    }
    // TODO: Iterate through all the bodies and draw them on the game canvas
},
```

For now, we have created a simple drawAllBodies() method that calls box2d.world.DrawDebugData() if the box2d.debugCanvas is present. We will eventually need to add code to iterate through all the bodies in the Box2D world and draw them on the game canvas.

We then call this new method from inside the game object's animate() method. One thing to note is that we draw the bodies after drawing the slingshot background but before drawing the slingshot foreground image. This way, the front of the slingshot is drawn on top of all the game entities.

If we run our code now and load the first level, we should see the debug canvas with all the entities, as shown in Figure 4-1.
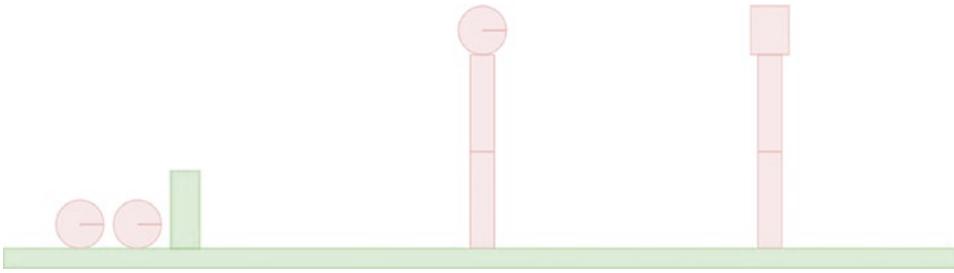
Figure 4-1. *First level drawn on the debug canvas*

The debug canvas view shows us all the game entities as circles and rectangles. We can also see the ground and slingshot blocks in a different color. We can use this view to quickly test our levels and make sure that all the entities are positioned correctly. Now that we can see that everything in the level looks alright, it's time to actually draw all the entities onto our game canvas.

Drawing the Entities

To draw an entity, we will define a method called draw() inside the entities object. This object will take the entity, its position, and its angle as parameters and draw it on the game canvas (see Listing 4-12).

Listing 4-12. The entities.draw() Method

```
// Take the entity, its position, and its angle and draw it on the game canvas
draw: function(entity, position, angle) {

    game.context.translate(position.x * box2d.scale - game.offsetLeft, position.y *
    box2d.scale);
    game.context.rotate(angle);
    var padding = 1;

    switch (entity.type) {
        case "block":
            game.context.drawImage(entity.sprite, 0, 0, entity.sprite.width,
            entity.sprite.height,
                        -entity.width / 2 - padding, -entity.height / 2 - padding,
                        entity.width + 2 * padding, entity.height + 2 * padding);
            break;
        case "villain":
        case "hero":
            if (entity.shape === "circle") {
                game.context.drawImage(entity.sprite, 0, 0, entity.sprite.width,
                entity.sprite.height,
                        -entity.radius - padding, -entity.radius - padding,
                        entity.radius * 2 + 2 * padding, entity.radius * 2 + 2 * padding);
            } else if (entity.shape === "rectangle") {
                game.context.drawImage(entity.sprite, 0, 0, entity.sprite.width,
                entity.sprite.height,
```

```
                          -entity.width / 2 - padding, -entity.height / 2 - padding,
                          entity.width + 2 * padding, entity.height + 2 * padding);
            }
            break;
        case "ground":
                // Do nothing... We draw objects like the ground & slingshot separately
            break;
    }

    game.context.rotate(-angle);
    game.context.translate(-position.x * box2d.scale + game.offsetLeft, -position.y *
    box2d.scale);
}
```

This method first translates and rotates the context to the position and angle of the entity. It then draws the object on the canvas based on the entity type and shape. Finally, it rotates and translates the context back to the original position.

One thing to note is that when using drawImage() the code stretches the image and makes it slightly larger than the original sprite by a padding size of one pixel in each direction. This is so that small gaps between Box2D objects get covered up.

■ **Note**   Box2D creates a "skin" around all polygons. The skin is used in stacking scenarios to keep polygons slightly separated. This allows continuous collision to work against the core polygon. When drawing Box2D objects, we need to compensate for this extra skin by drawing bodies slightly larger than their actual dimensions; otherwise, stacked objects will have unexplained gaps between them.

Now that we have defined an entities.draw() method, we need to call this method for every entity in our game world. We can iterate through every body in the game world by using the world object's GetBodyList() method. We will now modify the game object's drawAllBodies() method to do this, as shown in Listing 4-13.

Listing 4-13. Iterating Through All the Bodies and Drawing Them

```
drawAllBodies: function() {
    // Draw debug data if a debug canvas has been set up
    if (box2d.debugCanvas) {
        box2d.world.DrawDebugData();
    }

    // Iterate through all the bodies and draw them on the game canvas
    for (let body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if (entity) {
            entities.draw(entity, body.GetPosition(), body.GetAngle());
        }
    }

},
```

The for loop initializes body using world.GetBodyList(), which returns the first body in the world. The body object's GetNext() method returns the next body in the list until it reaches the end of the list and body becomes undefined, at which point we exit the for loop. Within the loop, we check to see if the body has an attached entity; if it does, we call entities.draw(), passing it the body's entity object, position, and angle.

If we run our game and load the first level now, we should see all the entities drawn on the canvas, as shown in Figure 4-2.
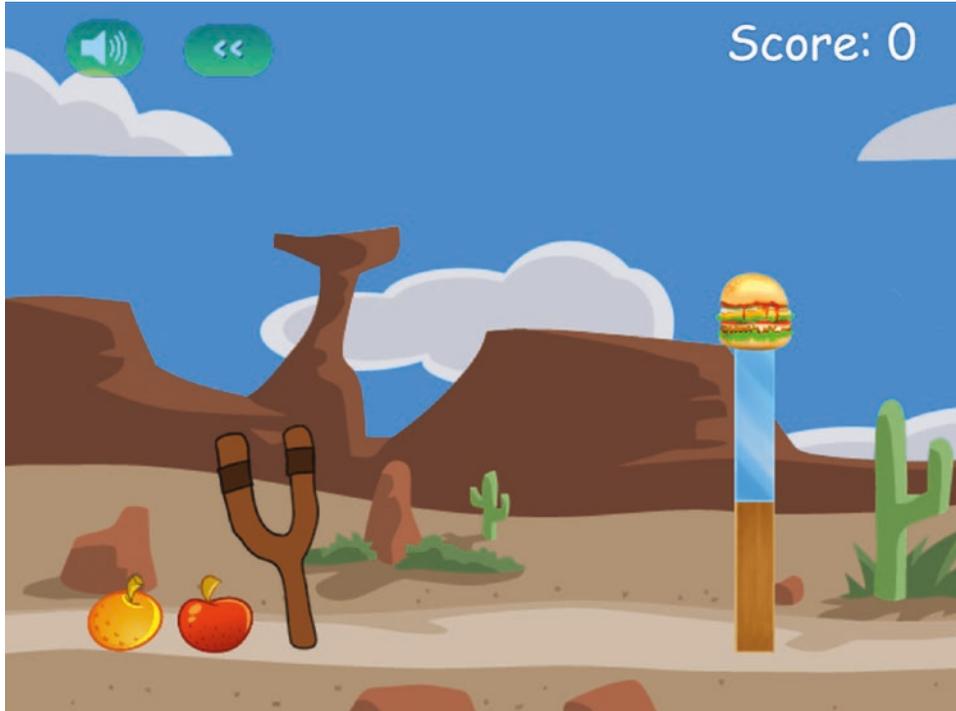
Figure 4-2. *Drawing the game entities on the canvas*

Once the level loads, the game pans to the right so that we can see the bad guys clearly, and then it pans back to the slingshot. We can see all the entities drawn properly at the same locations as on the debug canvas. The extra pixel we added in our draw() method ensures that all the stacked objects are positioned tightly next to each other. Note that the canvas preserves image transparencies when drawing images, which is why we can see the background through the glass block.

Now that we have drawn all the elements in the Box2D world, we need to animate the Box2D world.

Animating the Box2D World

As in the previous chapter, we can animate the Box2D world by calling the world object's Step() method and passing it the time step interval as a parameter. However, this is where things get a little tricky.

As per the Box2D manual recommendation, ideally, we should use a fixed time step for best results because variable time steps are hard to debug. Also as per the manual, Box2D works best with a time step of around 1/60th of a second, and you should use a time step no larger than 1/30th of a second. If the time step becomes very large, Box2D starts having problems with collisions, and bodies start passing through each other.

The requestAnimationFrame API can vary the frequency at which it calls the animate() method across browsers and machines. One way to get around this is to measure the time elapsed since the last call to animate() and pass this difference as a time step to Box2D.

However, if we switch tabs on the browser and then return to the game tab, the browser will call the animate() method less often, and this time step may become much larger than the upper limit of 1/30th of a second. To avoid problems due to a large time step, we will need to actively cap the time step if it becomes larger than 1/30th of a second.

Armed with this information, we will first define a step() method inside the box2d object. This method will take a time interval as a parameter and call the world object's Step() method (see Listing 4-14).

Listing 4-14. The box2d.step() Method

```
step: function(timeStep) {
    // As per Box2D docs, if the timeStep is larger than 1 / 30,
    // Box2D can start having problems with collision detection
    // So cap timeStep at 1 / 30

    if (timeStep > 1 / 30) {
        timeStep = 1 / 30;
    }

    // velocity iterations = 8
    // position iterations = 3

    box2d.world.Step(timeStep, 8, 3);
}
```

The step() method takes a time step in seconds and passes it to the world.Step() method. If timeStep is too large, we cap it at 1/30th of a second. We use the Box2D manual recommended values of 8 and 3 for velocity and position iterations. We will call this method from the game.animate() method after calculating the time step, as shown in Listing 4-15.

Listing 4-15. Calling box2d.step() from game.animate()

```
animate: function() {
    // Animate the characters
    var currentTime = new Date().getTime();

    if (game.lastUpdateTime) {
        var timeStep = (currentTime - game.lastUpdateTime) / 1000;

        box2d.step(timeStep);

    }

    game.lastUpdateTime = currentTime;

    // Handle panning, game states, and control flow
    game.handleGameLogic();
```

The first time `animate()` is called, `game.lastUpdateTime` will be undefined, so we will not calculate `timeStep` or call `box2d.step()`. However, in every subsequent animation cycle, we calculate the time that has passed since the last cycle and pass it to the `box2d.step()` method as `timeStep`. We then save the current time into the `game.lastUpdateTime` variable for the next animation cycle.

Loading the Hero

Now that the animation and engine are in place, it's time to implement some more game states (a.k.a. game modes). The first state that we will implement is the `load-next-hero` state. When in this state, the game needs to count the number of heroes and villains left in the game, check how many are left, and act accordingly as follows:

- If all the villains are gone, the game switches to the state `level-success`.

- If all the heroes are gone, the game switches to the state `level-failure`.

- If there are still heroes remaining, the game places the first hero on top of the slingshot and then switches to the state `wait-for-firing`.

We will do this by creating a method called `game.countHeroesAndVillains()` and modifying the `game.handleGameLogic()` method, as shown in Listing 4-16.

Listing 4-16. Handling the load-next-hero State

```
// Go through the heroes and villains still present in the Box2d world and store their
// Box2D bodies
heroes: undefined,
villains: undefined,
countHeroesAndVillains: function() {
    game.heroes = [];
    game.villains = [];
    for (let body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if (entity) {
            if (entity.type === "hero") {
                game.heroes.push(body);
            } else if (entity.type === "villain") {
                game.villains.push(body);
            }
        }
    }
},

handleGameLogic: function() {
    if (game.mode === "intro") {
        if (game.panTo(700)) {
            game.mode = "load-next-hero";
        }
    }
```

```
    if (game.mode === "wait-for-firing") {
        if (mouse.dragging) {
            game.panTo(mouse.x + game.offsetLeft);
        } else {
            game.panTo(game.slingshotX);
        }
    }

    if (game.mode === "load-next-hero") {

        // First count the heroes and villains and populate their respective arrays
        game.countHeroesAndVillains();

        // Check if any villains are alive, if not, end the level (success)
        if (game.villains.length === 0) {
            game.mode = "level-success";

            return;
        }

        // Check if there are any more heroes left to load, if not end the level (failure)
        if (game.heroes.length === 0) {
            game.mode = "level-failure";

            return;
        }

        // Load the hero and set mode to wait-for-firing
        if (!game.currentHero) {
            // Select the last hero in the heroes array
            game.currentHero = game.heroes[game.heroes.length - 1];

            // Starting position for loading the hero
            var heroStartX = 180;
            var heroStartY = 180;

            // And position it in mid-air, slightly above the slingshot
            game.currentHero.SetPosition({ x: heroStartX / box2d.scale, y: heroStartY /
            box2d.scale });
            game.currentHero.SetLinearVelocity({ x: 0, y: 0 });
            game.currentHero.SetAngularVelocity(0);

            // And since the hero had been sitting on the ground and is "asleep" in Box2D,
            "wake" it
            game.currentHero.SetAwake(true);
        } else {
            // Wait for hero to stop bouncing on top of the slingshot and fall asleep
            // and then switch to wait-for-firing
            game.panTo(game.slingshotX);
```

```
            if (!game.currentHero.IsAwake()) {
                game.mode = "wait-for-firing";
            }
        }
    }

    if (game.mode === "firing") {
        // If the mouse button is down, allow the hero to be dragged around and aimed
        // If not, fire the hero into the air
    }

    if (game.mode === "fired") {
        // Pan to the location of the current hero as it flies
        // Wait till the hero stops moving or is out of bounds
    }

    if (game.mode === "level-success" || game.mode === "level-failure") {
        // First pan all the way back to the left
        // Then show the game has ended and show the ending screen
    }
},
```

The countHeroesAndVillains() method iterates through all the bodies in the world and stores the heroes in the game.heroes array and the villains in the game.villains array.

Inside the handleGameLogic() method, when game.mode is load-next-hero, we first call countHeroesandVillains(). We then check to see if the villain or hero count is 0 and, if so, set game.mode to level-success or level-failure, respectively. If not, we save the last hero in the game.heroes array into the game.currentHero variable and set hero's position to a point in the air above the slingshot. We set its angular and linear velocity to 0. We also wake up the body in case it is asleep.

As soon as the body is woken up, it will be affected by gravity and start falling toward the slingshot. When the body drops on to the slingshot, it will bounce until it finally comes to rest and falls asleep again. We wait for this to occur, and once the body goes back to sleep we set game.mode to wait-for-firing.

If we run the game and start the first level, we will see the first hero bounce on the slingshot and come to rest, as shown in Figure 4-3.
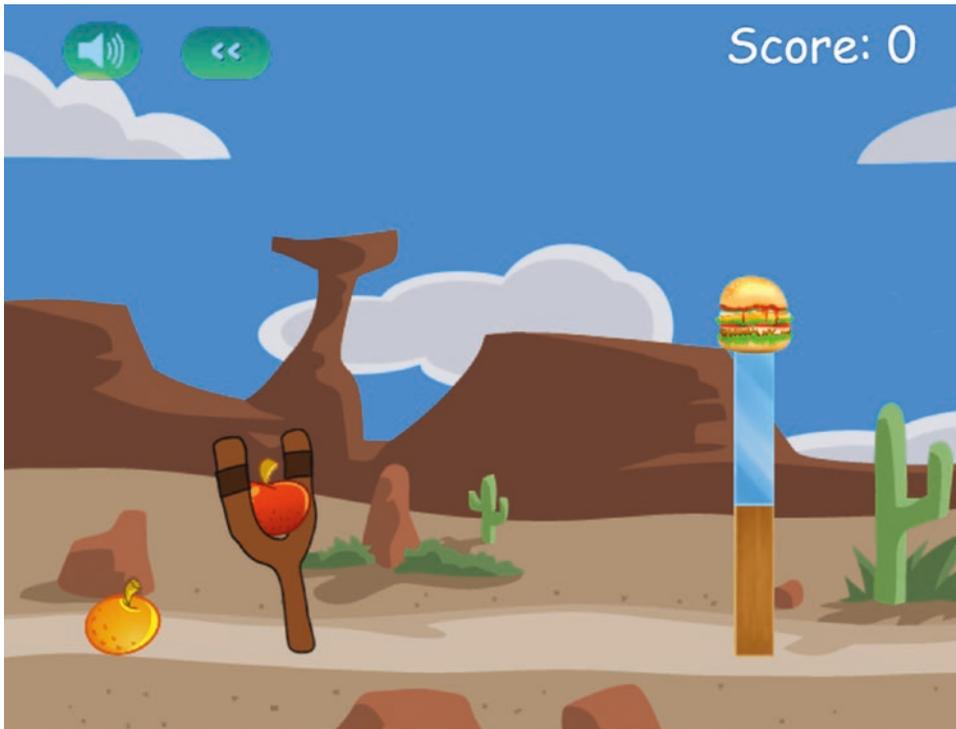
Figure 4-3. *First hero loaded on slingshot and waiting to be fired*

Now that we have the hero ready to be fired, we need to handle firing the hero from the slingshot.

Firing the Hero

We will implement firing the hero using three states:

- `wait-for-firing`: The game pans over the slingshot and waits for the mouse to be clicked and dragged while the pointer is above the hero. When this happens, it shifts to the `firing` state.

- `firing`: The game moves the hero with the mouse until the mouse button is released. When this happens, it pushes the hero with an impulse based on its distance from the slingshot and shifts to the `fired` state.

- `fired`: The game pans to follow the hero until it either comes to rest or goes outside the level bounds. The game then removes the hero from the game world and goes back to the `load-next-hero` state.

Before we can do that, we will need a way to detect when the user is attempting to move or fire the hero. To do so, we will first implement a method called `mouseOnCurrentHero()` inside the `game` object to test if the mouse pointer is positioned on the current hero (see Listing 4-17).

Listing 4-17. The game.mouseOnCurrentHero() Method

```
mouseOnCurrentHero: function() {
    if (!game.currentHero) {
        return false;
    }

    var position = game.currentHero.GetPosition();

    // Distance between center of the hero and the mouse cursor
    var distanceSquared = Math.pow(position.x * box2d.scale - mouse.x - game.offsetLeft, 2) +
    Math.pow(position.y * box2d.scale - mouse.y, 2);

    // Radius of the hero
    var radiusSquared = Math.pow(game.currentHero.GetUserData().radius, 2);

    // If the distance of mouse from the center is less than the radius, mouse is on the hero
    return (distanceSquared <= radiusSquared);
},
```

This method calculates the distance between the current hero center and the mouse location and compares it with the radius of the current hero to check if the mouse is positioned over the hero. If the distance is less than the radius, the mouse pointer is positioned on the hero.

We can get away with using this simple check since all our heroes are circular. If you want to implement heroes with different shapes, you might need a more complex method where you compare the mouse location with the bounds of the hero character.

We compare the squares of the values instead of calculating the square roots to save us an unnecessary calculation since comparing the squares will give us the same result.

Now that we have this method in place, we can implement the three states inside the handleGameLogic() method, as shown in Listing 4-18.

Listing 4-18. Handling the Firing States Inside the handleGameLogic() Method

```
if (game.mode === "wait-for-firing") {
    if (mouse.dragging) {
        if (game.mouseOnCurrentHero()) {
            game.mode = "firing";
        } else {
            game.panTo(mouse.x + game.offsetLeft);
        }
    } else {
        game.panTo(game.slingshotX);
    }
}

if (game.mode === "firing") {

    if (mouse.down) {
        game.panTo(game.slingshotX);
```

```
        // Limit dragging to maxDragDistance
        var distance = Math.pow(Math.pow(mouse.x - game.slingshotBandX + game.offsetLeft, 2) +
        Math.pow(mouse.y - game.slingshotBandY, 2), 0.5);
        var angle = Math.atan2(mouse.y - game.slingshotBandY, mouse.x - game.slingshotBandX);

        var minDragDistance = 10;
        var maxDragDistance = 120;
        var maxAngle = Math.PI * 145 / 180;

        if (angle > 0 && angle < maxAngle ) {
            angle = maxAngle;
        }
        if (angle < 0 && angle > -maxAngle ) {
            angle = -maxAngle;
        }
        // If hero has been dragged too far, limit movement
        if (distance > maxDragDistance) {
            distance = maxDragDistance;
        }

        // If the hero has been dragged in the wrong direction, limit movement
        if ((mouse.x + game.offsetLeft > game.slingshotBandX)) {
            distance = minDragDistance;
            angle = Math.PI;
        }

        // Position the hero based on the distance and angle calculated earlier
        game.currentHero.SetPosition({ x: (game.slingshotBandX + distance * Math.cos(angle) +
        game.offsetLeft) / box2d.scale,
            y: (game.slingshotBandY + distance * Math.sin(angle)) / box2d.scale });

    } else {
        game.mode = "fired";
        var impulseScaleFactor = 0.8;
        var heroPosition = game.currentHero.GetPosition();
        var heroPositionX = heroPosition.x * box2d.scale;
        var heroPositionY = heroPosition.y * box2d.scale;

        var impulse = new b2Vec2((game.slingshotBandX - heroPositionX) * impulseScaleFactor,
            (game.slingshotBandY - heroPositionY) * impulseScaleFactor);

        // Apply an impulse to the hero to fire it towards the target
        game.currentHero.ApplyImpulse(impulse, game.currentHero.GetWorldCenter());

        // Make sure the hero can't keep rolling indefinitely
        game.currentHero.SetAngularDamping(2);
    }
}
```

```
if (game.mode === "fired") {
    // Pan to the location of the current hero as it flies
    var heroX = game.currentHero.GetPosition().x * box2d.scale;

    game.panTo(heroX);

    // Wait till the hero stops moving or is out of bounds
    if (!game.currentHero.IsAwake() || heroX < 0 || heroX > game.currentLevel.
    foregroundImage.width) {
        // then remove the hero from the box2d world
        box2d.world.DestroyBody(game.currentHero);
        // clear the current hero
        game.currentHero = undefined;
        // and load next hero
        game.mode = "load-next-hero";
    }
}
```

The first state that we handle is wait-for-firing. When the state is wait-for-firing, if the mouse is being dragged and the mouse pointer is positioned on the hero, we change the state to firing; if the mouse pointer is not positioned on the hero, we pan the screen toward the cursor. If the mouse is not being dragged, we pan back toward the slingshot.

For the second state, firing, while the mouse button is down, we allow the player to move the hero around and set the position of the hero based on the current mouse position. We first calculate the distance and angle of the mouse from the top of the slingshot, and then, to prevent dragging the hero too far or to very large angles, we limit the angle and distance using minDragDistance, maxDragDistance, and maxAngle.

When the mouse button is released, we set the state to fired and apply an impulse to the hero using the b2Body object's ApplyImpulse() method. This method takes the impulse as a parameter in the form of a b2Vec2 object. We set the x and y values of the impulse vector as a multiple of the x and y distance of the hero from the top of the slingshot. The impulse scaling factor is a number that I came up with by experimenting with different values to find one that worked well for the game. We also set an angular damping on the hero that will cause it to slow down and come to a stop instead of rolling indefinitely.

Finally, when the state is fired, we pan the screen toward the hero and wait for the hero to either come to rest or fall outside of the game bounds. If it does either, we remove the hero from the world using the DestroyBody() method and change the state back to load-next-hero.

This cycle of states from load-next-hero to wait-for-firing to firing to fired will continue until we run out of either villains or heroes and move to the success or failure state.

If we run the code we have so far and load a level, we should be able to fire the hero at the blocks and knock them down, as shown in Figure 4-4.
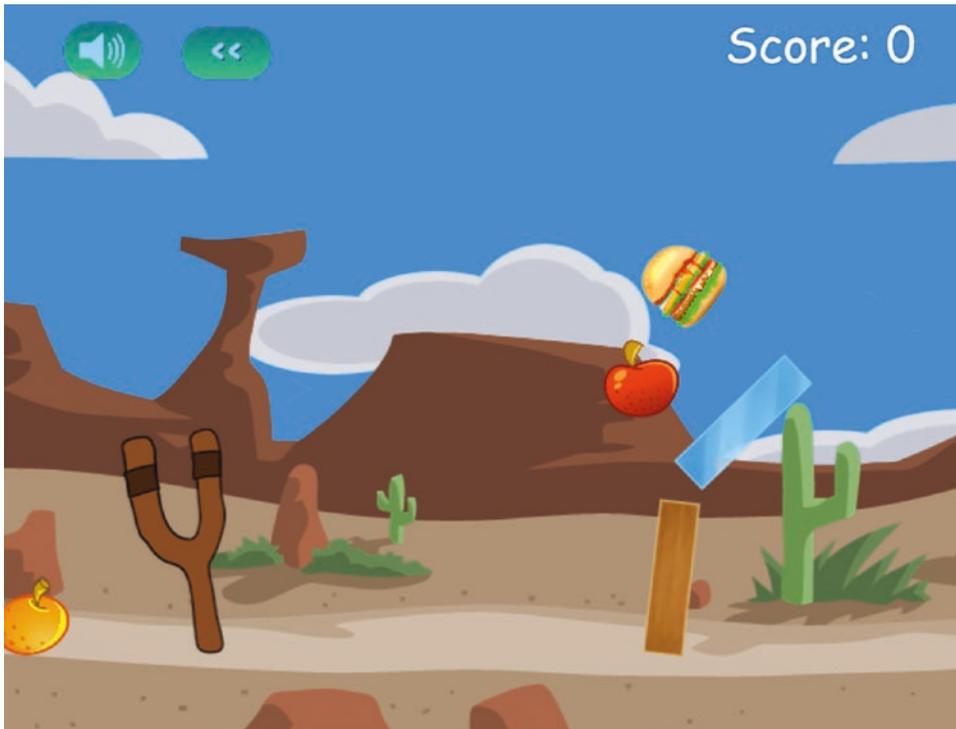
Figure 4-4. *Firing the hero at the blocks and knocking them over*

You will see that the game pans smoothly to follow the hero flying through the level. Once the hero either stops rolling or goes outside the bounds of the level, it is removed from the game and the next hero is loaded onto the slingshot. At this point, once all the heroes are gone, the game just stops and waits instead of ending the level. So, the next thing that we need to do is implement ending the level.

Ending the Level

Once a level ends, we will stop the game animation loop and display a level ending screen. This screen will give the user options to replay the current level, proceed to the next level, or return to the level selection screen.

The first thing we need to do is add the CSS for the endingscreen div element into styles.css, as shown in Listing 4-19.

Listing 4-19. CSS for the endingscreen div Element

```
/* Ending Screen */

#endingscreen {
    text-align: center;
    background: rgba(1, 1, 1, 0.3);
}
```

```css
#endingscreen div {

    /* Center the popup div within the screen */
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    transform-origin: center center;

    height: 250px;
    width: 330px;

    border: 1px solid gray;
    border-radius: 25px;
    background: rgba(1, 1, 1, 0.3);

    padding: 10px 30px 40px 50px;

    text-align: left;
}

.endingoption {
    font: 20px "Comic Sans MS";
    text-shadow: 0 0 2px black;
    color: white;
}

#endingscreen p img {
    top: 10px;
    position: relative;
    cursor: pointer;

    padding-right: 20px;
}

#endingmessage {
    font: 25px "Comic Sans MS";
    text-shadow: 0 0 2px black;
    color: yellow;
    text-align: center;
}
```

The CSS creates a dark background overlay above the game screen, centers the ending screen options, and then adds some general styling to the text.

Now that the ending screen is ready, we will implement a method called showEndingScreen() inside the game object to display the endingscreen div element (see Listing 4-20).

Listing 4-20. The game.showEndingScreen() Method

```
showEndingScreen: function() {
    var playNextLevel = document.getElementById("playnextlevel");
    var endingMessage = document.getElementById("endingmessage");

    if (game.mode === "level-success") {
        if (game.currentLevel.number < levels.data.length - 1) {
            endingMessage.innerHTML = "Level Complete. Well Done!!!";
            // More levels available. Show the play next level button
            playNextLevel.style.display = "block";
        } else {
            endingMessage.innerHTML = "All Levels Complete. Well Done!!!";
            // No more levels. Hide the play next level button
            playNextLevel.style.display = "none";
        }
    } else if (game.mode === "level-failure") {
        endingMessage.innerHTML = "Failed. Play Again?";
        // Failed level. Hide the play next level button
        playNextLevel.style.display = "none";
    }

    game.showScreen("endingscreen");
},
```

The showEndingScreen() method shows different ending messages based on the value of game.mode. The option to play the next level is shown if the player was successful and the current level was not the final level of the game. If the player was unsuccessful or the current level was the final level, the option is hidden.

We will now handle level-success and level-failure within the handleGameLogic() method of the game object as shown in Listing 4-21.

Listing 4-21. Implementing the Level Ending States in handleGameLogic()

```
if (game.mode === "level-success" || game.mode === "level-failure") {
    // First pan all the way back to the left
    if (game.panTo(0)) {
        // Then show the game has ended and show the ending screen
        game.ended = true;
        game.showEndingScreen();
    }
}
```

When game.mode is either level-success or level-failure, the game first pans back to the left, sets the game.ended property to true, then finally displays the ending screen shown in Figure 4-5.
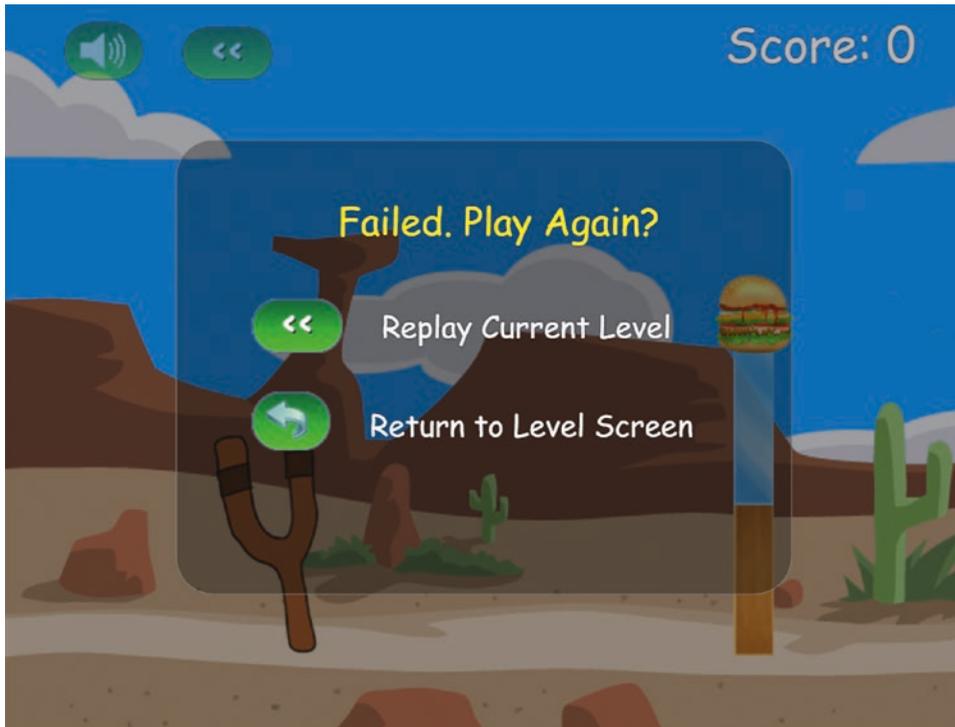
Figure 4-5. *The level ending screen*

Note that clicking the buttons won't do anything since we have not yet implemented any event handlers for the buttons.

Also, since we haven't implemented collision damage, the villains cannot die and we can never win the game. Therefore, the next thing we will implement is collision damage so we can destroy the bad guys and win the game.

Collision Damage

The first thing we need to do is track collisions by using a contact listener and overriding its PostSolve() method, just like we did in Chapter 3. We will create this listener in a handleCollisions() method that we will call immediately after creating the world in the init() method of the box2d object, as shown in Listing 4-22.

Listing 4-22. Handling Collisions Using a Contact Listener

```
init: function() {
    // Set up the Box2D world that will do most of the physics calculation
    var gravity = new b2Vec2(0, 9.8); // Declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; // Allow objects that are at rest to fall asleep and be excluded
    from calculations
```

```
    box2d.world = new b2World(gravity, allowSleep);

    // Activate debug drawing. Comment the line below to disable it.
    // this.setupDebugDraw();

    this.handleCollisions();
},

handleCollisions: function() {
    var listener = new b2ContactListener();

    listener.PostSolve = function(contact, impulse) {
        var body1 = contact.GetFixtureA().GetBody();
        var body2 = contact.GetFixtureB().GetBody();
        var entity1 = body1.GetUserData();
        var entity2 = body2.GetUserData();

        var impulseAlongNormal = Math.abs(impulse.normalImpulses[0]);

        // This listener is called a little too often. Filter out very tiny impulses.
        // After trying different values, 5 seems to work well as a threshold
        if (impulseAlongNormal > 5) {
            // If objects have a health, reduce health by the impulse value
            if (entity1.health) {
                entity1.health -= impulseAlongNormal;
            }

            if (entity2.health) {
                entity2.health -= impulseAlongNormal;
            }
        }
    };

    box2d.world.SetContactListener(listener);
},
```

Inside the `handleCollisions()` method, we declare a `b2ContactListener`, define its `PostSolve()` handler, and set the contact listener for the world, just as we did in Chapter 3.

Within the `PostSolve()` method, if either of the bodies involved in the collision has a `health` property, we reduce the health by the value of the impulse along the normal. Since the `PostSolve()` method is called for every little collision, we ignore any collision where `impulseAlongNormal` is less than a threshold value of 5.

The next thing we will do is add some code to check if a body's `health` property is less than zero or if the body has gone outside the level bounds. If either is true, we will remove the body from the world. We will do this by creating a method called `removeDeadBodies()` inside the game object as shown in Listing 4-23.

Listing 4-23. Removing Dead Bodies from the World

```
removeDeadBodies: function() {

    // Iterate through all the bodies
    for (let body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();
```

```
    if (entity) {
        var entityX = body.GetPosition().x * box2d.scale;

        // If the entity goes out of bounds or its health goes below 0
        if (entityX < 0 || entityX > game.currentLevel.foregroundImage.width ||
            (entity.health !== undefined && entity.health <= 0)) {

            // Remove the entity from the box2d world
            box2d.world.DestroyBody(body);

            // Update the score if a villain is killed
            if (entity.type === "villain") {
                game.score += entity.calories;
                document.getElementById("score").innerHTML = "Score: " + game.score;
            }
        }
    }
   }
 }
},
```

We iterate through all the bodies, just like we did when we were drawing them, and retrieve their entity data. If the code finds that the entity has gone outside the level bounds or the entity has lost all its health, we use the `world` object's `DestroyBody()` method to remove the body. Additionally, if the entity is a villain, we add the entity's `calorie` value to the game score and update the score on the screen.

We will call this method from the `animate()` method right after the call to `game.handleGameLogic()` as shown in Listing 4-24.

Listing 4-24. Calling removeDeadBodies() from animate()

```
// Handle panning, game states, and control flow
game.handleGameLogic();

// Remove any bodies that died during this animation cycle
game.removeDeadBodies();

// Draw the background with parallax scrolling
// First draw the background image, offset by a fraction of the offsetLeft distance (1/4)
// The bigger the fraction, the closer the background appears to be
```

If we run the game now, the villains should get destroyed and the score should increase, as shown in Figure 4-6.
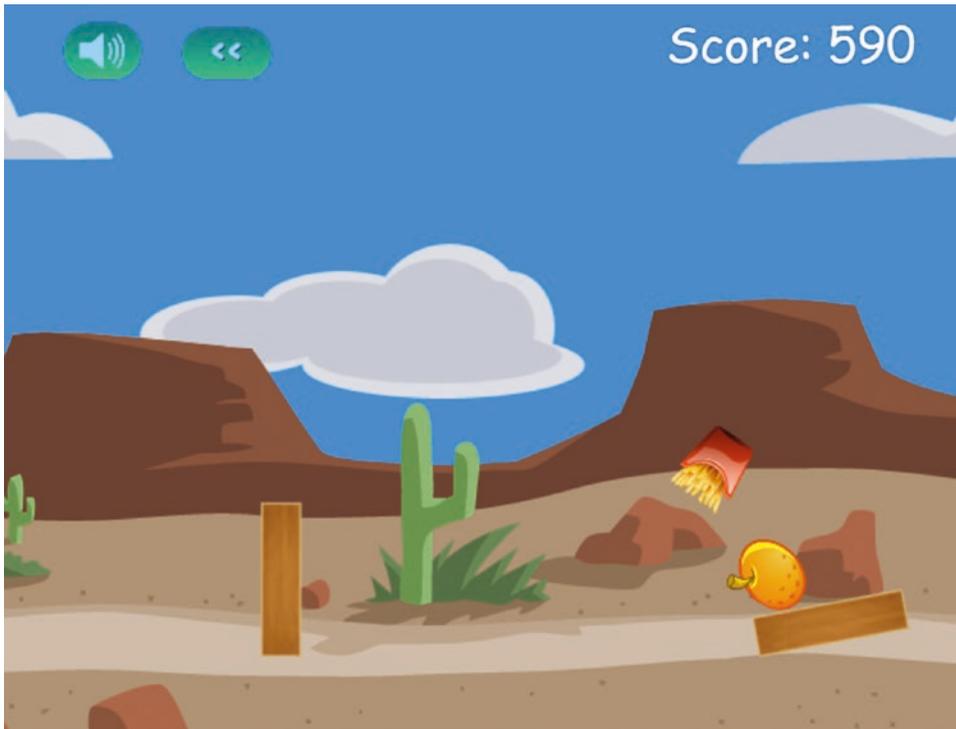
Figure 4-6. *The score increases after a bad guy gets destroyed*

Now that we have a working level, let's add a few finishing touches. The first thing we will do is draw a slingshot band when the hero is being fired.

Drawing the Slingshot Band

The slingshot band is going to be a thick brown line from the end of the slingshot to the extreme end of the hero. We will draw the band only when the game is in firing mode. We will do this in a drawSlingshotBand() method inside the game object, as shown in Listing 4-25.

Listing 4-25. Drawing the Slingshot Band

```
drawSlingshotBand: function() {
    game.context.strokeStyle = "rgb(68,31,11)"; // Dark brown color
    game.context.lineWidth = 7; // Draw a thick line

    // Use angle hero has been dragged and radius to calculate coordinates of edge of hero
    wrt. hero center
    var radius = game.currentHero.GetUserData().radius + 1; // 1px extra padding
    var heroX = game.currentHero.GetPosition().x * box2d.scale;
    var heroY = game.currentHero.GetPosition().y * box2d.scale;
    var angle = Math.atan2(game.slingshotBandY - heroY, game.slingshotBandX - heroX);
```

```
    // This is the X, Y position of the point where the band touches the hero
    var heroFarEdgeX = heroX - radius * Math.cos(angle);
    var heroFarEdgeY = heroY - radius * Math.sin(angle);

    game.context.beginPath();
    // Start line from top of slingshot (the back side)
    game.context.moveTo(game.slingshotBandX - game.offsetLeft, game.slingshotBandY);

    // Draw line to center of hero
    game.context.lineTo(heroX - game.offsetLeft, heroY);
    game.context.stroke();

    // Draw the hero on the back band
    entities.draw(game.currentHero.GetUserData(), game.currentHero.GetPosition(),
    game.currentHero.GetAngle());

    game.context.beginPath();
    // Move to edge of hero farthest from slingshot top
    game.context.moveTo(heroFarEdgeX - game.offsetLeft, heroFarEdgeY);

    // Draw line back to top of slingshot (the front side)
    game.context.lineTo(game.slingshotBandX - game.offsetLeft - 40, game.slingshotBandY + 15);
    game.context.stroke();
},
```

We start by setting the drawing color to a dark brown using the strokeStyle property. We next set the line drawing width to 7 pixels using the lineWidth property. We then draw a band from the back of the slingshot to the hero, draw the hero on top of the band, and, finally, draw a band from the front of the slingshot to the edge of the hero furthest from the slingshot.

We will call this method from the game.animate() method right after we draw all the other bodies, as shown in Listing 4-26.

Listing 4-26. Calling the drawSlingshotBand() Method from animate()

```
// Draw the base of the slingshot, offset by the entire offsetLeft distance
game.context.drawImage(game.slingshotImage, game.slingshotX - game.offsetLeft,
game.slingshotY);

// Draw all the bodies
game.drawAllBodies();

// Draw the band when we are firing a hero
if (game.mode === "firing") {
    game.drawSlingshotBand();
}

// Draw the front of the slingshot, offset by the entire offsetLeft distance
game.context.drawImage(game.slingshotFrontImage, game.slingshotX - game.offsetLeft,
game.slingshotY);
```

When we run this code, we should see a brown band around the hero, as shown in Figure 4-7.
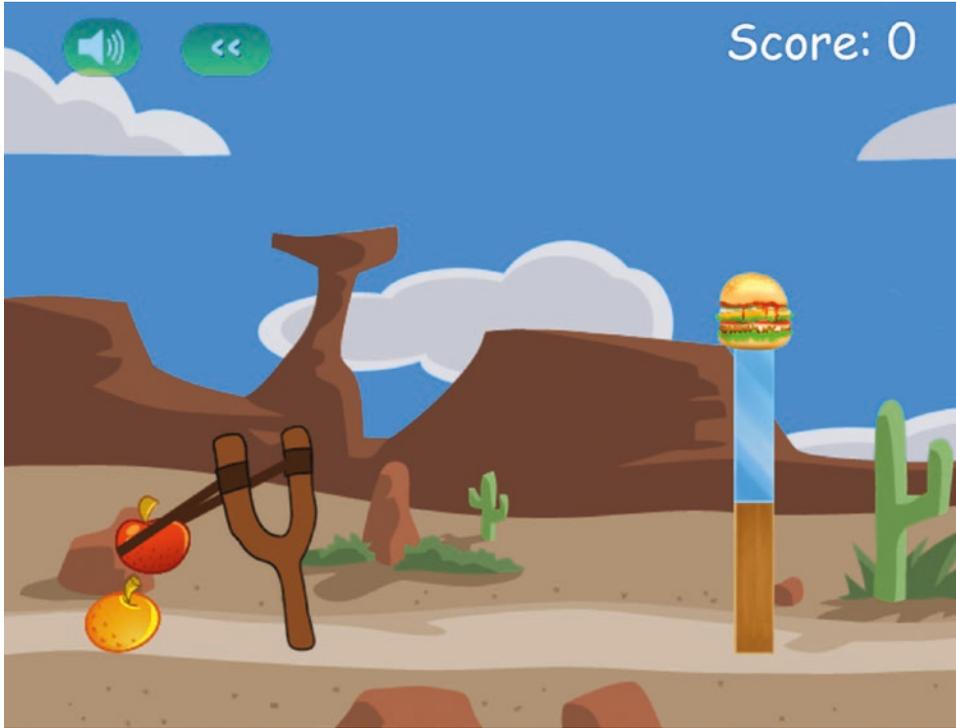
Figure 4-7. *Drawing the slingshot band*

This isn't a complete solution. The band might look a little unnatural at certain extreme angles. You might consider improving this method by superimposing some extra images on top of the band to cover up these edge effects. For now, this simple implementation will suffice.

Now that we have the artwork for the level wrapped up, let's implement the buttons for changing and restarting levels.

Changing Levels

We have already implemented one way to traverse levels, using the level selection screen. Now we will implement the buttons for restarting a level and proceeding to the next level.

We start by implementing the restartLevel() and startNextLevel() methods inside the game object, as shown in Listing 4-27.

Listing 4-27. Implementing restartLevel() and startNextLevel()

```
restartLevel: function() {
    window.cancelAnimationFrame(game.animationFrame);
    game.lastUpdateTime = undefined;
    levels.load(game.currentLevel.number);
},
```

```
startNextLevel: function() {
    window.cancelAnimationFrame(game.animationFrame);
    game.lastUpdateTime = undefined;
    levels.load(game.currentLevel.number + 1);
},
```

The methods are fairly simple. Both of them cancel any existing `animationFrame` loops, reset the `game.lastUpdateTime` variable, and finally call the `levels.load()` method with the appropriate level number.

We also need to call these level selection methods from the `onclick` event of the corresponding images in the `scorescreen` and `endingscreen` layers, as shown in Listing 4-28.

Listing 4-28. Setting the onclick Events for Changing Levels

```
<div id="scorescreen" class="gamelayer">
    <img id="togglemusic" src="images/icons/sound.png" alt="Toggle Music">
    <img src="images/icons/prev.png" alt="Restart Level" onclick="game.restartLevel()">
    <span id="score">Score: 0</span>
</div>

<div id="endingscreen" class="gamelayer">
    <div>
        <p id="endingmessage">The Level Is Over Message</p>
        <p id="playcurrentlevel" class="endingoption" onclick="game.restartLevel()">
        <img src="images/icons/prev.png" alt="Replay">Replay Current Level</p>
        <p id="playnextlevel" class="endingoption" onclick="game.startNextLevel()">
        <img src="images/icons/next.png" alt="Next">Play Next Level</p>
        <p id="returntolevelscreen" class="endingoption" onclick="game.
        showLevelScreen()"><img src="images/icons/return.png" alt="Return">Return to Level
        Screen</p>
    </div>
</div>
```

If we run the game, we should now be able to restart a level, proceed to the next level, or return to the level screen using the provided buttons.

We now have a working game with complete levels. We also have a simple way to build new levels. However, there is still one last element missing: sound.

Adding Sound

Adding sound makes a game much more immersive since it provides the player with an additional source of sensory stimulation and feedback, which makes the game feel a little more real.

We will start by adding a few sound effects for when the slingshot is released, for when a hero or villain bounces, and for when one of the blocks gets destroyed. We will also add some background music, along with the capability to turn it off if we want.

The sounds files for each of these effects are available in the audio folder (in both MP3 and OGG format).

We will start by loading these sound files in a `loadSounds()` method in the game object, which we will then call from the `init()` method, as shown in Listing 4-29.

Listing 4-29. Loading Sound and Background Music

```
init: function() {
    //Get handler for game canvas and context
    game.canvas = document.getElementById("gamecanvas");
    game.context = game.canvas.getContext("2d");

    // Initialize objects
    levels.init();
    loader.init();
    mouse.init();

    // Load All Sound Effects and Background Music
    game.loadSounds(function() {
        // Hide all game layers and display the start screen
        game.hideScreens();
        game.showScreen("gamestartscreen");
    });

},

loadSounds: function(onload) {
    game.backgroundMusic = loader.loadSound("audio/gurdonark-kindergarten");

    game.slingshotReleasedSound = loader.loadSound("audio/released");
    game.bounceSound = loader.loadSound("audio/bounce");
    game.breakSound = {
        "glass": loader.loadSound("audio/glassbreak"),
        "wood": loader.loadSound("audio/woodbreak")
    };

    loader.onload = onload;
},
```

The loadSounds() method loads the different sound files using the loader.loadSound() method and saves them for later reference. We store the break sounds in an associative array so that we can easily add sounds for more entities and reference them by name. The background music is an excellent Creative Commons–licensed tune called "Kindergarten" by Gurdonark. After initiating the loading of the sound files, we set the onload property of the loader.

Within the init() method, we first call loadSounds() and pass an onload function within which we display the game start screen. This way, the game will display a loading screen until the audio has loaded completely, and then finally display the game menu.

■ **Tip**   You can find some amazing free music for your own games at the ccMixter website, located at http://ccmixter.org.

Adding Break and Bounce Sounds

Now that we have loaded these sounds, we need to associate these sound effects with the entities and play them at the right time. We will modify the `entities.create()` method and set the break and bounce sounds in the entity definitions, as shown in Listing 4-30.

Listing 4-30. Assigning Sounds to Entities During Creation

```
create: function(entity) {
    var definition = entities.definitions[entity.name];

    if (!definition) {
        console.log("Undefined entity name", entity.name);

        return;
    }

    switch(entity.type) {
        case "block": // simple rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.shape = "rectangle";
            entity.sprite = loader.loadImage("images/entities/" + entity.name + ".png");

            entity.breakSound = game.breakSound[entity.name];

            box2d.createRectangle(entity, definition);
            break;
        case "ground": // simple rectangles
                // No need for health. These are indestructible
            entity.shape = "rectangle";
                // No need for sprites. These won't be drawn at all
            box2d.createRectangle(entity, definition);
            break;
        case "hero":    // simple circles
        case "villain": // can be circles or rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.sprite = loader.loadImage("images/entities/" + entity.name + ".png");
            entity.shape = definition.shape;

            entity.bounceSound = game.bounceSound;

            if (definition.shape === "circle") {
                entity.radius = definition.radius;
                box2d.createCircle(entity, definition);
            } else if (definition.shape === "rectangle") {
                entity.width = definition.width;
                entity.height = definition.height;
                box2d.createRectangle(entity, definition);
            }
            break;
```

```
        default:
            console.log("Undefined entity type", entity.type);
            break;
    }
},
```

The only change in the code is that we set a breakSound attribute for block entities and a bounceSound attribute for hero and villain entities. The advantage of attaching sounds to entities during creation like this is that every entity can have its own custom "break" sound and "bounce" sound if needed.

Now, all we need to do is play the sounds when the events actually occur. First, we play the bounce sound whenever we detect a collision, inside the handleCollisions method we defined earlier, as shown in Listing 4-31.

Listing 4-31. Playing the Bounce Sound During a Collision

```
handleCollisions: function() {
    var listener = new b2ContactListener();

    listener.PostSolve = function(contact, impulse) {
        var body1 = contact.GetFixtureA().GetBody();
        var body2 = contact.GetFixtureB().GetBody();
        var entity1 = body1.GetUserData();
        var entity2 = body2.GetUserData();

        var impulseAlongNormal = Math.abs(impulse.normalImpulses[0]);

        // This listener is called a little too often. Filter out very tiny impulses.
        // After trying different values, 5 seems to work well as a threshold
        if (impulseAlongNormal > 5) {
            // If objects have a health, reduce health by the impulse value
            if (entity1.health) {
                entity1.health -= impulseAlongNormal;
            }

            if (entity2.health) {
                entity2.health -= impulseAlongNormal;
            }

            // If entities have a bounce sound, play the sound
            if (entity1.bounceSound) {
                entity1.bounceSound.play();
            }

            if (entity2.bounceSound) {
                entity2.bounceSound.play();
            }

        }
    };
    box2d.world.SetContactListener(listener);
},
```

During a collision, we check if the entity has a bounceSound property defined and, if so, we play the sound. If we define bounce sounds for any entity, this code will automatically play it whenever the entity is in a significant collision.

Next, we play the break sound any time an object gets destroyed, inside the removeDeadBodies() method of the game object (see Listing 4-32).

Listing 4-32. Playing the Break Sound when an Object Is Destroyed

```
removeDeadBodies: function() {

    // Iterate through all the bodies
    for (let body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if (entity) {
            var entityX = body.GetPosition().x * box2d.scale;

            // If the entity goes out of bounds or its health goes below 0
            if (entityX < 0 || entityX > game.currentLevel.foregroundImage.width ||
                (entity.health !== undefined && entity.health <= 0)) {

                // Remove the entity from the box2d world
                box2d.world.DestroyBody(body);

                // Update the score if a villain is killed
                if (entity.type === "villain") {
                    game.score += entity.calories;
                    document.getElementById("score").innerHTML = "Score: " + game.score;
                }

                // If entity has a break sound, play the sound
                if (entity.breakSound) {
                    entity.breakSound.play();
                }
            }
        }
    }
},
```

Again, we check to see if the entity being destroyed has a breakSound property and, if so, we play the sound. So far we have defined break sounds for the glass and wood blocks, but we can easily extend the code to add sounds for the other entities.

Finally, we play the slingshotReleasedSound when game.mode changes from firing to fired inside the handleGameLogic() method (see Listing 4-33).

Listing 4-33. Playing the Slingshot Released Sound when the Hero Is Fired

```
} else {
    game.mode = "fired";
    var impulseScaleFactor = 0.8;
    var heroPosition = game.currentHero.GetPosition();
    var heroPositionX = heroPosition.x * box2d.scale;
    var heroPositionY = heroPosition.y * box2d.scale;
```

```
    var impulse = new b2Vec2((game.slingshotBandX - heroPositionX) * impulseScaleFactor,
        (game.slingshotBandY - heroPositionY) * impulseScaleFactor);

    // Apply an impulse to the hero to fire it towards the target
    game.currentHero.ApplyImpulse(impulse, game.currentHero.GetWorldCenter());

    // Make sure the hero can't keep rolling indefinitely
    game.currentHero.SetAngularDamping(2);

    // Play the slingshot released sound
    game.slingshotReleasedSound.play();
}
```

Now when you run the game, you should hear sound effects when the hero is fired, when it bumps against something, or when the blocks get destroyed. The last thing we will be adding is the background music.

Adding Background Music

We have already loaded the background music file along with the other sound files in the game.loadSounds() method. Now we need to create a few methods for starting, stopping, and toggling the background music. We will add these methods to the game object, as shown in Listing 4-34.

Listing 4-34. Methods for Controlling Background Music

```
startBackgroundMusic: function() {
    game.backgroundMusic.play();
    game.setBackgroundMusicButton();
},

stopBackgroundMusic: function() {
    game.backgroundMusic.pause();
    // Go to the beginning of the song
    game.backgroundMusic.currentTime = 0;

    game.setBackgroundMusicButton();
},

toggleBackgroundMusic: function() {
    if (game.backgroundMusic.paused) {
        game.backgroundMusic.play();
    } else {
        game.backgroundMusic.pause();
    }

    game.setBackgroundMusicButton();
},
```

```
setBackgroundMusicButton: function() {
    var toggleImage = document.getElementById("togglemusic");

    if (game.backgroundMusic.paused) {
        toggleImage.src = "images/icons/nosound.png";
    } else {
        toggleImage.src = "images/icons/sound.png";
    }
},
```

The startBackgroundMusic() method first calls the backgroundMusic object's play() method and then calls setBackgroundMusicButton() to set the toggle music button image appropriately.

The stopBackgroundMusic() method calls the backgroundMusic object's pause() method and sets the audio back to the beginning of the song by setting its currentTime property to 0. It then calls setBackgroundMusicButton() to change the music button image.

Finally, the toggleBackgroundMusic() method checks to see whether or not the music is currently paused, calls either the pause() or play() method, and then sets the toggle image appropriately.

The setBackgroundMusicButton() method simply sets the src property of the background music image based on whether or not the background music is currently playing.

Now that we have these methods in place, we need to call them. We will call the startBackgroundMusic() method when the game starts from inside the game.start() method, as shown in Listing 4-35.

Listing 4-35. Starting the Background Music

```
start: function() {
    game.hideScreens();

    // Display the game canvas and score
    game.showScreen("gamecanvas");
    game.showScreen("scorescreen");

    game.mode = "intro";
    game.currentHero = undefined;

    game.offsetLeft = 0;
    game.ended = false;

    game.animationFrame = window.requestAnimationFrame(game.animate, game.canvas);

    // Play the background music when the game starts
    game.startBackgroundMusic();
},
```

Next, we will call the stopBackgroundMusic() method whenever the level ends by adding it to the showEndingScreen() method, as shown in Listing 4-36.

Listing 4-36. Stopping the Background Music

```
showEndingScreen: function() {
    var playNextLevel = document.getElementById("playnextlevel");
    var endingMessage = document.getElementById("endingmessage");
```

```
    if (game.mode === "level-success") {
        if (game.currentLevel.number < levels.data.length - 1) {
            endingMessage.innerHTML = "Level Complete. Well Done!!!";
            // More levels available. Show the play next level button
            playNextLevel.style.display = "block";
        } else {
            endingMessage.innerHTML = "All Levels Complete. Well Done!!!";
            // No more levels. Hide the play next level button
            playNextLevel.style.display = "none";
        }
    } else if (game.mode === "level-failure") {
        endingMessage.innerHTML = "Failed. Play Again?";
        // Failed level. Hide the play next level button
        playNextLevel.style.display = "none";
    }

    game.showScreen("endingscreen");

    // Stop the background music when the game ends
    game.stopBackgroundMusic();
},
```

Finally, we will call the toggleBackgroundMusic() method from the onclick event of the toggle music button inside the scorescreen layer, as shown in Listing 4-37.

Listing 4-37. Toggling the Background Music

```
<div id="scorescreen" class="gamelayer">
    <img id="togglemusic" src="images/icons/sound.png" alt="Toggle Music"
    onclick="game.toggleBackgroundMusic()">
    <img src="images/icons/prev.png" alt="Restart Level" onclick="game.restartLevel()">
    <span id="score">Score: 0</span>
</div>
```

Now if we run the game, the background music starts playing as soon as the level starts. When we click the toggle button, the music pauses and the button changes to the no-sound icon, as shown in Figure 4-8.
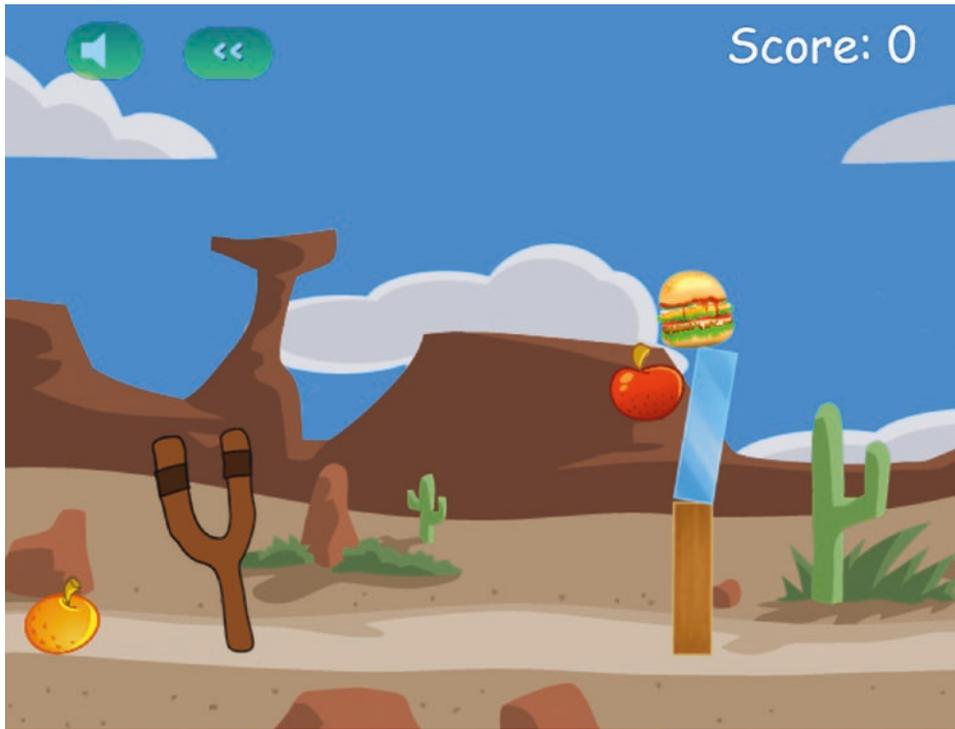
Figure 4-8. *The finished game with the background music switched off*

With this last change, we now have a complete, working game. We can select a level from the level selection screen, and play the game by slinging across the hero fruits to attack the evil junk food, while listening to sound effects and background music.

Of course, there is still a lot of room for us to expand the functionality of this game. Some of the obvious next steps would be to add animations for different entities, add more levels, tweak the game physics parameters, and add more heroes and villains with different characteristics.

However, the game has all the essential elements that people have come to expect from a good HTML5 game. You can use the code in this game as a starting point for any of your own physics engine–based games and take it wherever you would like.

Take some time to enjoy the game and come up with your own ideas for levels.

Summary

Over the past three chapters, we created our first physics engine–based HTML5 game. We started in Chapter 2 by creating a basic game framework with menus, a level system, and an asset loader and setting up game animation. We then covered the basics of Box2D in Chapter 3. Finally, in this chapter we integrated Box2D into our existing game framework and wrapped up our game by adding menu options, sounds effects, and music.

One limitation of this game we have created is that it will work only on desktop browsers, and not on mobile devices such as smartphones and tablets. However, now that HTML5 is fully supported by most mobile devices, it is possible to make this game work even on mobile devices with just a little additional work.

In the next chapter, we will look at some of the differences and challenges in building mobile device games using HTML5, as well as ways to handle them. We will then apply this knowledge to modify our existing game so it works on mobile devices as well.

■ ■ ■

Creating a Mobile Game

Over the last few years, the use of mobile devices and their mobile browsers has steadily increased, to the point where several popular sites have started reporting that the number of mobile device users visiting them exceeds the number of visits from desktop users. While we may not have necessarily reached a tipping point where mobile device usage has surpassed desktop usage everywhere, we have definitely reached a point where mobile device users comprise a significant market share and can no longer be ignored.

During this same time period, mobile browsers have been steadily improving their support for the HTML5 API, so essential HTML5 features can now be expected to work on most mobile devices without needing significant hacks or workarounds. What this means for us as game developers is that we can now start building HTML5 games that also work on mobile devices, something that wasn't easily possible a few years ago.

In this chapter we will look at the differences between the mobile device environment and the desktop environment and some of the considerations necessary when making games for mobile devices. We will then continue with our game Froot Wars, using the code from Chapter 4 as a starting point, and modify it to run on mobile devices.

So let's get started.

Challenges in Developing for Mobile Devices

Even though the HTML5 API and JavaScript features remain almost the same on mobile devices, developing for mobile devices is not without its challenges. Some of these challenges come from the fact that, compared to desktops, mobile devices have a smaller form factor with limited screen real estate, typically slower Internet access, different input methods and APIs, and significantly less computing power and memory at their disposal.

This necessitates being extremely careful about not wasting resources while working within the limitations of these devices. The most important considerations when developing for mobile devices are as follows:

- *Size and form factor*: The smaller form factor, limited screen space, and different device aspect ratios in various mobile devices mean games need to be designed to be responsive and to intelligently fit the available screen space. We no longer can assume the availability of space, and we need to adjust our game based on what space is actually available on the device. The ability to change screen orientation from portrait to landscape on mobile devices also means that our games need to be able to adjust to changes in the size of the content area dynamically.

- *Different input events*: Our game so far was designed to work with the mouse. However, mobile devices typically do not have access to a mouse and instead use touch-based gestures to emulate the mouse. While the emulation is sufficient for simpler uses such as clicking buttons, we need finer control when working with dragging and swiping within the game, which requires understanding and using the Touch API.

- *Browser limitations for audio*: In an attempt to improve the user experience and limit unnecessary bandwidth or resource usage, some mobile browsers such as Safari add additional safeguards such as not preloading audio and preventing audio from playing without user interaction. While this doesn't matter as much in typical HTML5 pages, it can significantly degrade the experience in an HTML5 game, and we need to find ways to make audio work smoothly.

- *Limited Internet bandwidth*: Typical mobile devices may be connecting to our game via slower EDGE, 2G, or 3G networks, so resources can take a long time to load. We need to ensure that the game takes this into consideration, by reducing the size of the resources to the best of our ability and then using a preloader to wait until the resources have loaded completely.

While there might be a few more small problems that pop up, taking care of these big issues while developing our games for mobile devices should be sufficient for a fairly decent mobile gameplay experience. Now that we know what the challenges are, we will start tackling them one at a time, starting with making the game responsive.

Making the Game Responsive

Before we start working on making the game responsive, let's take a look at the problem more closely. Luckily for us, most desktop browsers now allow us an easy way to emulate mobile devices, including different mobile aspect ratios and touch events, so we can do our initial development on the desktop before we actually test the game on mobile devices.

This emulation feature is called "Device Mode" on Chrome, and "Responsive Device Mode" on Firefox and Safari. The simplest way to activate this feature on any of these browsers is to open the developer console and click the button with the mobile phone–shaped icon. You can read detailed instructions for this at `https://developer.mozilla.org/en-US/docs/Tools/Responsive_Design_Mode` for Firefox, `https://developers.google.com/web/tools/chrome-devtools/device-mode/` for Chrome, and `https://support.apple.com/kb/PH26266` for Safari.

■ **Note**   While emulation will give you a close approximation of how your game will look on a mobile device, it cannot replicate exact mobile device conditions and thus is not a substitute for an actual mobile device. Testing your games on actual mobile devices is essential for making sure that your games work as intended.

If you open in your desktop browser the game we have developed to this point, you should be able to go to the developer console and activate mobile emulation as shown in Figure 5-1.
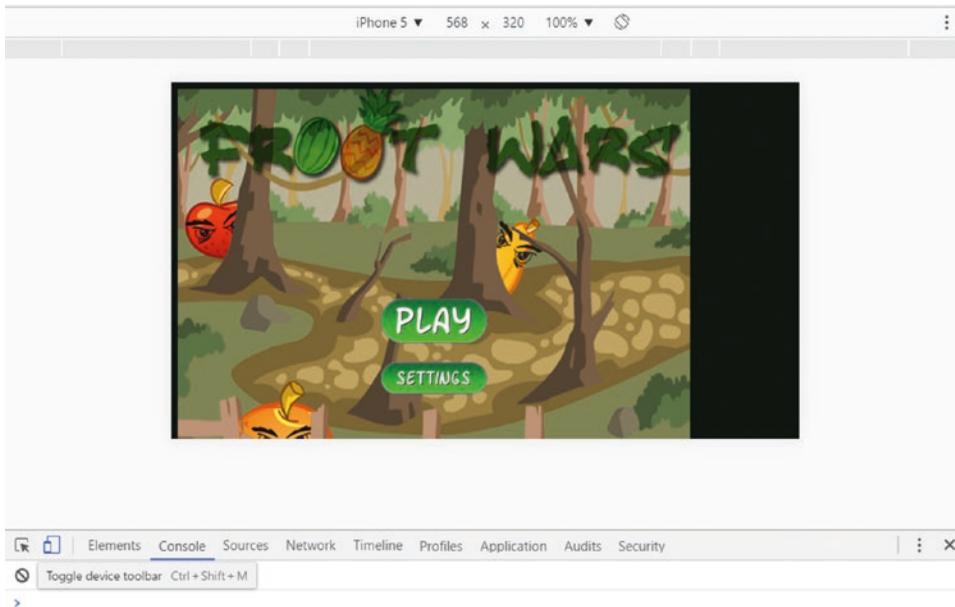
Figure 5-1. *Previewing the game in Responsive Device Mode*

This feature allows you to choose the device resolution of several popular mobile devices, including the iPhone and Nexus lines of phones. It also allows you to select the device orientation and easily switch between landscape and portrait mode at the click of a button. For now, let's just pick any one of the devices and set the orientation to landscape so we can see what our game looks like.

The first thing you will notice, as evident in Figure 5-1, is that the lack of proper responsive behavior is causing the game to be cropped at the bottom and aligned to the left with black space on the right. So, the first thing we need to add is automatic scaling and positioning.

Automatic Scaling and Resizing

In an effort to adjust desktop or non-mobile content for mobile devices, mobile browsers often attempt to automatically scale content in different ways. Since we will be handling scaling at our end, the first thing we need to do is let the browser know not to allow zooming and scaling of its own and try to use all available space. We will do this by adding a simple viewport meta tag to the head section of index.html as shown in Listing 5-1.

Listing 5-1. Preventing Automatic Scaling in the Browser

```
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
    minimum-scale=1, width=device-width">
    <title>Froot Wars</title>
    <script src="js/Box2d.min.js" type="text/javascript"></script>
    <script src="js/game.js" type="text/javascript"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen">
</head>
```

The newly added `viewport meta` tag first tells the browser not to allow the user to change the scale by using pinching gestures, since this would affect the game experience. It also tells the browser not try to scale and fit content and forces the scale to stay at its original value of 1. Finally, it tells the browser to use the entire available width of the device.

If you refresh the code in the browser, you should see that the browser no longer tries to scale or adjust the content, as shown in Figure 5-2.
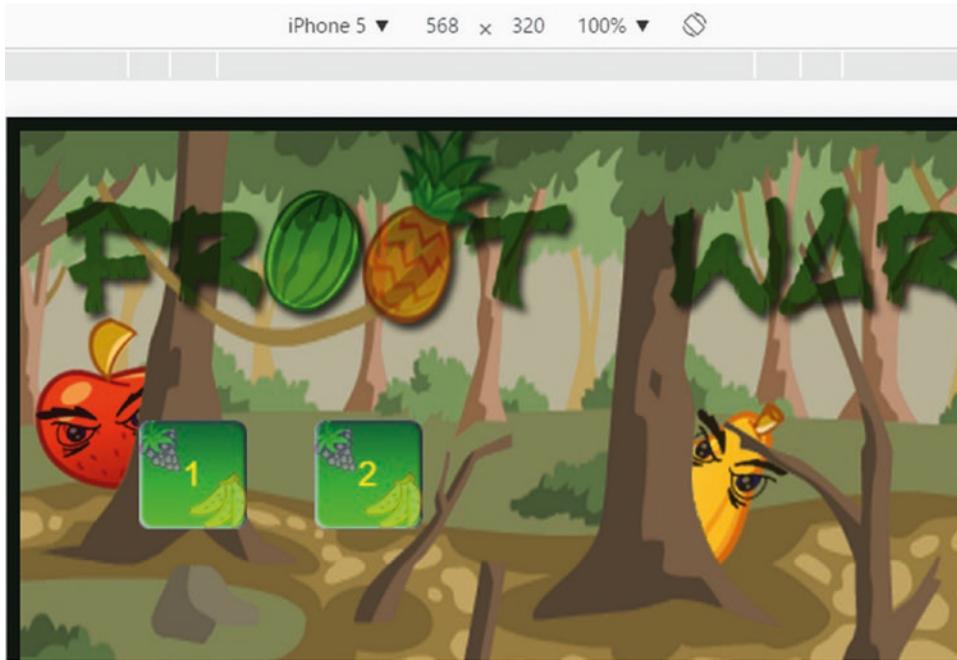
Figure 5-2. *The game with automatic and user scaling disabled*

Since the game is no longer scaled down, the dimensions of 640px by 480px are too large for the device, and the game is now cropped on the right side as well as the bottom. Now that we know the browser won't be scaling or modifying our content in unexpected ways, we can write our own code to make the game scale and fit the available space.

The first thing we will do is add some additional styles for the `body`, `wrapper`, and `gamecontainer` div elements in `styles.css` as shown in Listing 5-2.

Listing 5-2. Additional Styles for body, wrapper, and gamecontainer

```
body {
    background: #000900;

    /* Prevent the ugly blue highlighting from accidental selection of text */
    user-select: none;

    /* Disable long touch hold select */
    -webkit-touch-callout: none !important;

    overflow: hidden;
}
```

```
#wrapper {
    position: absolute;

    /* Wrapper covers entire window height and width */
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
}

#gamecontainer {

    /* Set game container width, height, and background */
    width: 640px;
    height: 480px;
    background: url("images/splashscreen.png");

    /* Center the game container relative to outer wrapper */
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    transform-origin: center center;
}
```

In this newly added CSS code, we ensure that the wrapper div covers the entire available window area, and that the gamecontainer div is centered within it. We also set the overflow style of the body element to hidden to prevent unnecessary scroll bars from showing up and prevent the context menu from showing up when the user touches the screen for a long time.

The game should now be centered on the device; however, the content is still too large for the screen and needs to be resized. To handle the resizing, we will create a new resize() method inside the game object as shown in Listing 5-3.

Listing 5-3. Adding a resize() Method to the game Object

```
scale: 1,
resize: function() {

    var maxWidth = window.innerWidth;
    var maxHeight = window.innerHeight;

    var scale = Math.min(maxWidth / 640, maxHeight / 480);

    var gameContainer = document.getElementById("gamecontainer");

    gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";

    game.scale = scale;
},
```

Within the resize() method, we first get the maximum available width and height using the innerWidth and innerHeight properties of the window object.

119

We then calculate the maximum amount we can scale up the game without having one of the dimensions become larger than the window size. To do this, we calculate the maximum that the game can be scaled along each axis and pick the lower of the two using `Math.min()`.

For example, if a device has dimensions of 1280px by 720px while our content is 640px by 480px, we could potentially scale the x-axis by 2 but the y-axis only by 1.5. Since scaling by anything more than 1.5 would cause the content to become too large along the y-axis, we pick the lower of the two values (1.5) as the scale to use.

Finally, we set the game container scale to this new scale value using a CSS `transform` attribute and save this value in `game.scale`.

Now that we have our `resize()` method, we need to call it when the game is first loaded and any time the browser is resized, as shown in Listing 5-4.

Listing 5-4. Calling the resize() Method on Loading and Resizing

```
// Initialize game once page has fully loaded
window.addEventListener("load", function() {
    game.resize();
    game.init();
});

window.addEventListener("resize", function() {
    game.resize();
});
```

We first modify the load event listener to call `game.resize()` as soon as the window has loaded, after which we continue to initialize the game. We also listen for the `resize` event and call `game.resize()` whenever the window is resized. This way, the game should automatically resize when it first loads, and adjust anytime the window is resized or the device is rotated.

If you now run this code, you should see the game perfectly centered and scaled as shown in Figure 5-3.
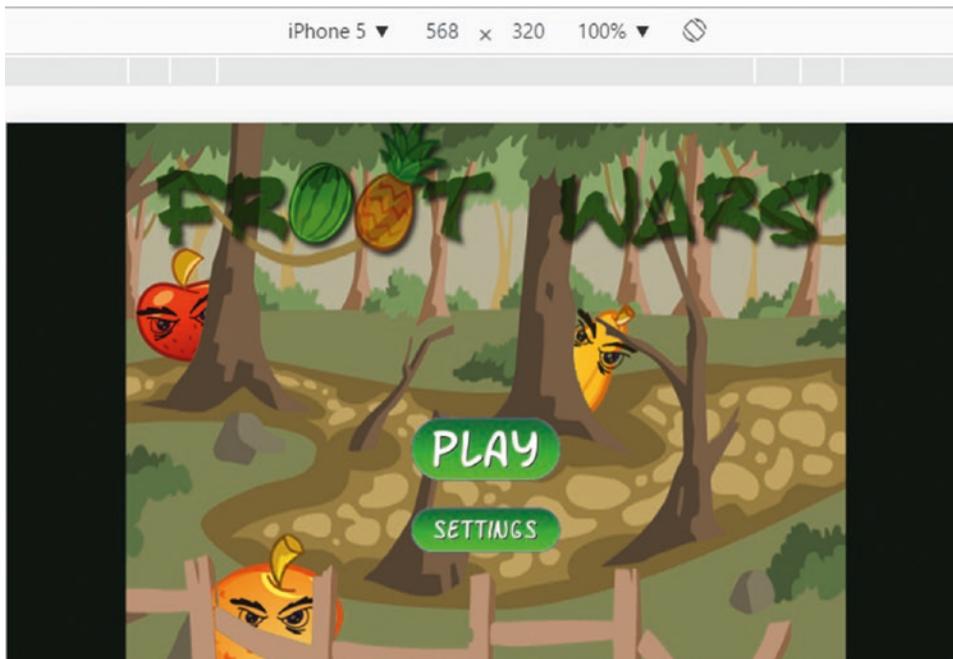
Figure 5-3. *Centered and scaled with the resize() method*

If you change the device orientation, the game should automatically adjust and fit into the new dimensions.

Even though the game is scaled reasonably well, you will notice that there is a considerable amount of black unused space on the sides. Most modern mobile devices tend to use wider aspect ratios, so we need a way to take this into account while still working well with devices that do not have a wide screen.

Handling Different Aspect Ratios

The problem we have in trying to make our game work for different aspect ratios is that our game is currently designed for a fixed 4:3 aspect ratio and our canvas and background image are sized to be exactly 640px by 480px.

We will start by modifying our CSS to use a wider version of the background image (1024px by 480px) as shown in Listing 5-5.

Listing 5-5. Using a Wider Background Image

```css
#gamecontainer {

    /* Set game container width, height, and background */
    width: 640px;
    height: 480px;

    /* Use a wider splash screen and center it within the container */
    background: url("images/splashscreenwide.png");
    background-position: center;
    background-repeat: no-repeat;

    /* Center the game container relative to outer wrapper */
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    transform-origin: center center;
}
```

We first change the background image to the wider version (splashscreenwide.png), and then ensure that it is centered and not repeated. This by itself won't make any apparent change to the game since the container is still sized at 640px by 480px.

Next we will modify the game.resize() method to also try to increase the aspect ratio where possible, as shown in Listing 5-6.

Listing 5-6. Changing the Aspect Ratio Inside game.resize()

```javascript
resize: function() {

    var maxWidth = window.innerWidth;
    var maxHeight = window.innerHeight;

    var scale = Math.min(maxWidth / 640, maxHeight / 480);

    var gameContainer = document.getElementById("gamecontainer");

    gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";
```

```
    // Find the maximum width we can set based on the current scale
    // and clamp the value between 640 and 1024
    var width = Math.max(640, Math.min(1024, maxWidth / scale ));

    // Apply this new width to game container and game canvas
    gameContainer.style.width = width + "px";

    var gameCanvas = document.getElementById("gamecanvas");

    gameCanvas.width = width;

    game.scale = scale;
},
```

In this newly added code, we first calculate the maximum width that we can set based on the newly computed scale. Since our artwork and level backgrounds have a width of 1024px, we also make sure that this new game width can never be greater than 1024px or less than the original 640px. We then set the container and canvas to this newly computed width.

If you run the code after these changes, you will find that the game automatically widens to try and use up the extra space on the sides as shown in Figure 5-4.
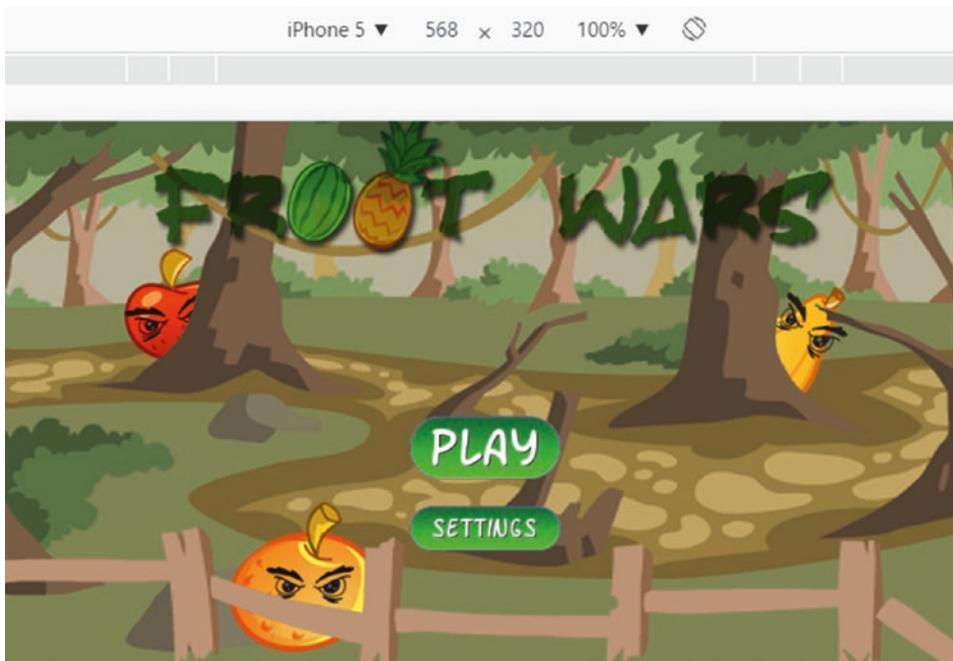
Figure 5-4. *Adjusting the aspect ratio with the resize() method*

You can use the device drop-down to switch to other devices with different aspect ratios, and the game automatically adjusts as much as possible to try and minimize any unused space.

If you click Play and start the game, you will see that the game canvas has also expanded to account for this wider space. Additionally, since our panning code takes canvas and level widths into account, even panning should automatically work with this new width.

We now have a responsive game that adjusts to different screen sizes and aspect ratios.

However, while the game looks fine visually, you will not be able to play it properly. We still need to make adjustments to the mouse object to handle touch events and adjust for the new game scale.

Fixing Mouse and Touch Event Handling

We need to resolve two problems in our input handling before the game can work properly:

- Our code to compute mouse.x and mouse.y does not adjust for the fact that the game is now scaled up by game.scale. This has a reasonably simple fix. We just need to scale down the computed x and y coordinates back by game.scale.

- Mobile devices that do not actually have a mouse do not generate all mouse events. They primarily generate touch events, and attempt to approximate mouse events where possible. This approximation works reasonably with events such as click, which is why our game buttons, which rely on the click event, continue to function. However, the browser will not generate a mousemove event and instead will generate only a touchmove event.

We will modify the mouse object to handle both sets of events (mouse and touch) and adjust for game scaling as shown in Listing 5-7.

Listing 5-7. Handling Touch Events and Scaling in the mouse Object

```
var mouse = {
    x: 0,
    y: 0,
    down: false,
    dragging: false,

    init: function() {
        var canvas = document.getElementById("gamecanvas");

        canvas.addEventListener("mousemove", mouse.mousemovehandler, false);
        canvas.addEventListener("mousedown", mouse.mousedownhandler, false);
        canvas.addEventListener("mouseup", mouse.mouseuphandler, false);
        canvas.addEventListener("mouseout", mouse.mouseuphandler, false);

        // Handle touchmove separately
        canvas.addEventListener("touchmove", mouse.touchmovehandler, false);

        // Reuse mouse handlers for touchstart, touchend, touchcancel
        canvas.addEventListener("touchstart", mouse.mousedownhandler, false);
        canvas.addEventListener("touchend", mouse.mouseuphandler, false);
        canvas.addEventListener("touchcancel", mouse.mouseuphandler, false);
    },
```

```
    mousemovehandler: function(ev) {
        var offset = game.canvas.getBoundingClientRect();

        mouse.x = (ev.clientX - offset.left) / game.scale;
        mouse.y = (ev.clientY - offset.top) / game.scale;

        if (mouse.down) {
            mouse.dragging = true;
        }

        ev.preventDefault();
    },

    touchmovehandler: function(ev) {
        var touch = ev.targetTouches[0];
        var offset = game.canvas.getBoundingClientRect();

        mouse.x = (touch.clientX - offset.left) / game.scale;
        mouse.y = (touch.clientY - offset.top) / game.scale;

        if (mouse.down) {
            mouse.dragging = true;
        }

        ev.preventDefault();
    },

    mousedownhandler: function(ev) {
        mouse.down = true;

        ev.preventDefault();
    },

    mouseuphandler: function(ev) {
        mouse.down = false;
        mouse.dragging = false;

        ev.preventDefault();
    }
};
```

We first modify the init() method to assign handlers for all the touch events: touchmove, touchstart, touchend, and touchcancel. We assign a new method called touchmovehandler() for the touchmove event, but reuse the mousedown() and mouseuphandler() methods for the remaining events. This is because touchmovehandler() will need to be slightly different from mousemovehandler(), and we cannot reuse the same method.

Next we modify the mousemovehandler() method to adjust for game.scale when calculating the x and y position.

Finally, we define the touchmovehander() method, which uses the event's targetTouches array to get the details of the first touch. We then use the touch object's clientX and clientY properties to calculate mouse.x and mouse.y like we did in the mousemovehandler() method.

One thing to keep in mind is that the Touch API is designed to be able to handle multiple simultaneous touches. We can access details of every one of these touches on the canvas using the `targetTouches` array, as well as uniquely identify each of them using the `identifier` property. You can read more about the Touch event API at https://developer.mozilla.org/en/docs/Web/API/Touch_events.

Our touch handling code assumes that the player will be using only one finger at a time, and uses only the first touch in the `targetTouches` array to emulate mouse-like behavior. This will result in slightly unexpected behavior if the user decides to use multiple touches simultaneously.

It is possible to develop a more robust solution that will ignore any additional touches apart from the first by using the `identifier` property within each touch object to uniquely identify the first touch. However, our simple implementation should suffice for now.

Another thing to note is that the call to `preventDefault()` at the bottom of all our handlers prevents the browser's typical behavior of firing the equivalent mouse events when a touch event is fired to emulate a mouse. Using `preventDefault()` will ensure that the browser does not call our event handlers twice—once as a touch event and then again as an emulated mouse event.

If you run the game on the device emulator, you should now be able to play the game normally. You should also be able to play the game on a normal browser without emulation and see that the game dynamically scales to fit the entire window as far as possible.

Now that the game works fairly well on our device emulator, it is time to load the game on an actual mobile device and see how it fares.

Loading the Game on a Mobile Device

The simplest way to load a game on a mobile device is to host the game on a web server and then access the URL from the mobile browser.

Numerous popular web servers are available for every operating system. However, to keep things simple, we will use the Node.js `http-server` package, which is a simple and bare-bones server that is ideal for quick development and testing work.

If you already have a web server on your machine and are comfortable with setting it up to serve the game code, you can probably skip the steps in this section.

The first thing you will need to do is install Node.js and its package manager, npm. You will find the necessary instructions for installing the latest version at https://docs.npmjs.com/getting-started/installing-node. The reason I recommend installing and using Node.js is that we will need Node.js in later chapters anyway, to build a JavaScript-based multiplayer game server.

Once you have installed Node.js and npm, it's time to install http-server. You can install http-server from the command line on your terminal using the command

```
npm install -g http-server
```

You can read more about http-server and its many configuration options and features at https://www.npmjs.com/package/http-server. Once you have http-server installed on your machine, you should be able to serve the game just by switching to the folder that contains index.html and using the command

```
http-server
```

As soon as the server starts, it should show you a status message letting you know the URLS by which you can access the game:

```
Starting up http-server, serving ./
Available on:
  http://192.168.0.100:8080
  http://127.0.0.1:8080
```

Note that the actual URLs displayed by http-server will vary depending on your network setup. Once the server has started, as long as your mobile device is on the same wireless network, you should be able to access the game on the mobile browser using one of the URLs provided. This server, while fairly simple, has several useful configuration options such as the ability to disable caching and to run on different ports. I'd encourage you to review the documentation and take a look at these options when you get a chance.

■ **Note** The URLs that http-server provides include the loopback IP address (127.0.0.1), which is only accessible from the machine where the server is running. To access the server from a different device, you need to use one of the non-loopback URLs, which should be accessible as long as your devices are on the same network. If you have trouble accessing the server, also make sure that your firewall isn't interfering.

Now that we can load the game on a mobile device, you will find that it seems to function as intended on Android devices. However, it does not even load properly on iOS devices and gets stuck on the loading screen, as shown in Figure 5-5.
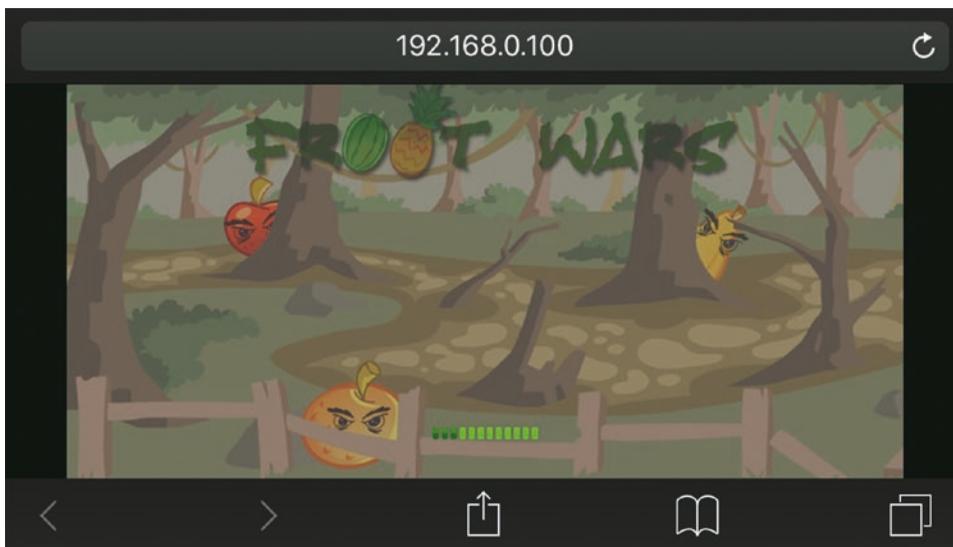
Figure 5-5. *Stuck at loading screen on an actual iOS device*

Unfortunately, this occurs because of an optimization that the Safari browser for mobile devices makes, wherein it does not load audio files until they need to be played to prevent unnecessary bandwidth usage. An unfortunate side effect of this is that the canplaythrough event does not fire for the audio objects that we try to preload, resulting in the game just hanging at the loading stage.

This is just one of many problems that we will face with audio on mobile devices. So the next thing we will focus on is fixing these audio problems.

Fixing Audio Problems on Mobile Browsers

In addition to the problem with preloading audio files in the Safari mobile browser, just discussed, there are several other limitations on mobile browsers. These include not being able to play multiple audio files simultaneously and some devices not allowing audio to play unless the sound has been triggered by a user interaction. Luckily for us, there is a simple way to work around most of these issues: the HTML5 Web Audio API.

The Web Audio API

The Web Audio API is an incredibly powerful and versatile system for controlling audio in the browser. It allows us to combine multiple audio sources, apply filters, and add all sorts of dynamics effects.

The Web Audio API uses an audio context and is designed to allow modular routing using a system of nodes and connections. You typical connect different nodes, starting with a source node (such as an Oscillator node or BufferSource node), which can be connected to different effect nodes (such as a Gain node), which are then finally connected to the destination node (AudioContext.destination). You can connect multiple sources either directly or via effect nodes to the destination node, allowing for dynamically creating multiple channels of sound.

You can play a simple sound using Web Audio with an Oscillator node as shown in Listing 5-8.

Listing 5-8. Using Web Audio with an OscillatorNode

```
<!DOCTYPE html>
<html>
    <head>
        <title>Webaudio Example 1</title>
    </head>
    <body>
        <script>
            // Some browsers support AudioContext, while others support webkitAudioContext
            var context = new (window.AudioContext || window.webkitAudioContext)();

            // An oscillator source node just plays a sound at a specific frequency
            var oscillatorNode = context.createOscillator();

            // Connect the oscillator directly to the destination
            oscillatorNode.connect(context.destination);

            // Start the oscillator now (at the current time)
            oscillatorNode.start(context.currentTime);

            // And stop it two seconds after the current time
            oscillatorNode.stop(context.currentTime + 2);
        </script>
    </body>
</html>
```

We start by first defining a new audio context using either AudioContext or webkitAudioContext, depending on which one is available to us on the browser. We then use the context.createOscillator() method to create an oscillator object, which we connect directly to the destination. Anything connected to this destination can be heard by us.

127

Next we tell the oscillator when to start and stop with respect to the context current time. In our case the start time is the current moment and the stop time is two seconds after this start time. If you run this code in your browser, you should hear a loud, high-pitched sound that lasts for exactly two seconds.

Now if we want to control the volume of the `Oscillator` node (or any other node), we can pass it through a `Gain` node as shown in Listing 5-9.

Listing 5-9. Passing Audio Through a Gain Node

```
<!DOCTYPE html>
<html>
    <head>
        <title>Webaudio Example 2</title>
    </head>
    <body>
        <script>
            // Initialize the audio context
            var context = new (window.AudioContext || window.webkitAudioContext)();

            // An oscillator source node just plays a sound at a specific frequency
            var oscillatorNode = context.createOscillator();

            // A gain node controls the volume
            var gainNode  = context.createGain();

            // Set the volume to 1/5th of the original volume
            gainNode.gain.value = 0.2;

            // Connect the oscillator to the gain node
            oscillatorNode.connect(gainNode);

            // Connect the gain node to the destination
            gainNode.connect(context.destination);

            // Start the oscillator now (at the current time)
            oscillatorNode.start(context.currentTime);

            // And stop it two seconds after the current time
            oscillatorNode.stop(context.currentTime + 2);
        </script>
    </body>
</html>
```

This time we connect the oscillator node to a gain node, set the gain value to a fraction of the original, and then connect it to the destination.

If you run this code, you should hear the same sound as before but at a much lower volume. This is how you typically chain nodes using the Web Audio API, to apply different kinds of effects to your sounds.

You can also use a `BufferSource` node to load audio files and play them. However, you will first need to load the audio file yourself using `XMLHttpRequest` as shown in Listing 5-10.

Listing 5-10. Playing Audio Files Using a BufferSource Node

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Webaudio Example 3</title>
    </head>
    <body>
        <script>
            // Initialize the audio context
            var context = new (window.AudioContext || window.webkitAudioContext)();

            // Load the audio file using an XMLHttpRequest
            var request = new XMLHttpRequest();
            request.open("GET", "audio/bounce.ogg", true);
            request.responseType = "arraybuffer";

            // Wait for the request to load the audio file
            request.onload = function() {
                 // Once the audio file has loaded, decode it
                var undecodedAudio = request.response;
                context.decodeAudioData(undecodedAudio, function (decodedAudioBuffer) {
                    // Once the audio has been decoded create a buffer source
                    var bufferSourceNode = context.createBufferSource();

                    // Tell the buffer source node to use the decoded audio buffer
                    bufferSourceNode.buffer = decodedAudioBuffer;

                    // Connect the buffer source node to the destination
                    bufferSourceNode.connect(context.destination);

                    // Start playing the buffer source node now
                    bufferSourceNode.start(context.currentTime);
                });
            };

            // Finally initiate the request
            request.send();

        </script>
    </body>
</html>
```

This time, we create an XMLHTTPRequest object that loads our audio file as an array buffer. Once the audio is loaded by the request, we use context.decodeAudio() to convert the audio data into an audio buffer that can be used by a BufferSource node. After the audio has been decoded, we create a BufferSource node, assign it the buffer, and then connect it to the destination and play it like any other source node.

An important thing to note is that because of a security restriction on the XMLHTTPRequest object, it cannot access local files by using the file:// protocol, and trying to do so will result in an error message. You have to be running this code on the web server for XMLHTTPRequest to work properly.

If you load this code via the web server URL, you should hear the audio file being played once the file loads. Depending on your browser, you might need to use the MP3 file instead of the OGG file.

Now that you understand how the Web Audio API works, it's time to integrate it into our game.

Integrating Web Audio

As you can imagine, modifying our game code to use Web Audio will take considerable effort and rewriting. We will need to modify the loader to use XMLHTTPRequest, decode the buffers, and store them after loading. We will also need to cache these requests so that the browser doesn't fire multiple requests in case a file is loaded multiple times. Each time we want to play an audio, we will need to load the appropriate buffer, create a BufferSource node, and play it. Pausing or stopping music will also need additional code.

Luckily for us, we have a much simpler way to migrate our code to use Web Audio. We are going to use a library called wAudio.js that I created exactly for this purpose. wAudio.js is a drop-in replacement for the HTML5 Audio object, which transparently uses the Web Audio API behind the scenes.

By including this library in our code, we can use wAudio() everywhere that we used Audio() before, and the game should work just like before, while using the methods of the Web Audio API.

To load wAudio.js, we need to first include the script in the head section of index.html as shown in Listing 5-11.

Listing 5-11. Loading wAudio.js in the head Section of index.html

```
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
    minimum-scale=1, width=device-width">
    <title>Froot Wars</title>
    <script src="js/Box2d.min.js" type="text/javascript"></script>
    <script src="js/wAudio.js" type="text/javascript"></script>
    <script src="js/game.js" type="text/javascript"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen">
</head>
```

Note that we need to load wAudio.js before we load our game. This will automatically create a wAudio object as long as the browser supports Web Audio. Now we need to modify the loadSound() method of the loader object to use wAudio if it is available, as shown in Listing 5-12.

Listing 5-12. Modifying loadSound() to Use wAudio.js

```
loadSound: function(url) {
    this.loaded = false;
    this.totalCount++;

    game.showScreen("loadingscreen");

    var audio = new (window.wAudio || Audio)();

    audio.addEventListener("canplaythrough", loader.itemLoaded, false);
    audio.src = url + loader.soundFileExtn;

    return audio;
},
```

As you can see, we made a very simple change to either use wAudio or fall back to using Audio in case wAudio is not present.

If you load the game on your mobile device now, you will notice that the game no longer gets stuck at the loading screen, and you can start the game without any problems even on iOS devices.

However, for some reason the game still doesn't play any sound on Safari. You will also find that if you try to mute and unmute the audio using the toggle music button in our game, the sound will miraculously start playing.

This odd behavior occurs because of another restriction in Safari, which is that audio playback needs to be initiated by a user input event. Once the first sound has been initiated by a user-generated event (such as a click or tap), audio starts playing normally.

The interesting thing about this restriction is that this first sound doesn't even need to be audible; we can play a sound through a gain filter set to a gain value of 0, and audio will still be restored. Since initializing audio like this is such a common requirement, the wAudio.js library includes a method called playMutedAudio(), which behind the scenes uses an oscillator node to play a short sound without any volume. To use this method, we will first create a playGame() method inside the game object as shown in Listing 5-13.

Listing 5-13. The playGame() Method Inside the game Object

```
// Called when the Play button is clicked
playGame: function() {

    // Initialize audio for mobile Safari
    if (window.wAudio) {
        window.wAudio.playMutedSound();
    }

    game.showLevelScreen();
},
```

The playGame() method is fairly simple. It first checks for the existence of wAudio and then calls wAudio.playMutedSound(). It then calls game.showLevelScreen() to display the level screen.

Next, we will call this method when the Play button in the start screen is clicked by modifying index.html as shown in Listing 5-14.

Listing 5-14. Calling playGame() when the Play Button Is Clicked

```
<div id="gamestartscreen" class="gamelayer">
    <img src="images/icons/play.png" alt="Play Game" onclick="game.playGame()"><br>
    <img src="images/icons/settings.png" alt="Settings">
</div>
```

Now when we start the game and click Play, wAudio will play the muted sound so that Safari's requirement for playing audio is satisfied. If we now run the game, the audio should work perfectly, even on iOS devices.

Thanks to the wAudio.js library, our migration to Web Audio was quick and painless. The latest version of the wAudio.js library will always be available at its GitHub URL-https://github.com/adityaravishankar/wAudio.js. This code is shared under an MIT license, so you can feel free to use it in any of your projects. If you prefer, you can also reuse just portions of the code that you find useful.

One thing to remember about this library is that the XMLHTTPRequest restriction on accessing file:// URLs also applies to wAudio, since wAudio uses XMLHTTPRequest behind the scenes. If we want to use wAudio in our game, we will need to access our game via a web server.

Now that our game works on mobile devices with audio, we will add a few finishing touches to our game so it feels less like a web page and more like a native application.

Adding Some Finishing Touches

There are still a few things that we can do to make the game look and feel better. These include preventing accidental scrolling include preventing accidental scrolling and removing the address bar by allowing web-app mode.

Preventing Accidental Scrolling

You might have already noticed that even though our game has been scaled to exactly match the window size, it is possible to accidentally scroll up or down by dragging your finger on the screen. Since we have no need to scroll within our game, we can disable this default behavior by listening to the document object's touchmove event as shown in Listing 5-15.

Listing 5-15. Disabling the Default Mobile Scroll

```
// Initialize game once page has fully loaded
window.addEventListener("load", function() {
    game.resize();
    game.init();
});

window.addEventListener("resize", function() {
    game.resize();
});

document.addEventListener("touchmove", function(ev) {
    ev.preventDefault();
});
```

All we do is add a listener for the touchmove event and it, call the event's preventDefault() method to prevent the browser's scroll behavior. If you play the game now, it will no longer scroll and the game area should stay in place.

The next problem we will need to tackle is the presence of the address bar at the top of the browser window, so the game can be played full screen like a native app.

Allowing Full Screen

Mobile browsers in general do not allow hiding the address bar at the top of the browser. Even desktop browsers that allow a full screen mode explicitly ask the user for permission before switching to full screen and hiding the address bar. This is largely due to security concerns to prevent malicious sites from rendering a fake address bar and spoofing other sites to try and trick the user into revealing sensitive information.

While there are a few scroll based hacks that sometimes work at temporarily hiding the navigation bar, these are somewhat unreliable and not recommended.

However, both Android and iOS devices now support meta tags that allow us to specify that a particular page is a web app designed to work in full screen mode. We can enable this by adding new meta tags into the head section of index.html as shown in Listing 5-16.

Listing 5-16. Adding meta Tags for Web App Mode

```
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
    minimum-scale=1, width=device-width">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="mobile-web-app-capable" content="yes">
    <title>Froot Wars</title>
    <script src="js/Box2d.min.js" type="text/javascript"></script>
    <script src="js/wAudio.js" type="text/javascript"></script>
    <script src="js/game.js" type="text/javascript"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen">
</head>
```

We add two new meta tags, one for iOS and another for Android, to let the browsers know that our game is web app capable.

Now this won't directly affect the browser experience. You need to open the game in the browser and use the option to save the web page to your home screen. Once you do this, you will find an icon for the game on your mobile device home screen, which you can click to start the game just like you would a typical mobile phone app.

When you start the game from the home screen, it should now run in full screen mode without the annoying address and status bars.

Now this isn't a perfect solution. Unless the player actually saves the game on their home screen, they will not have the ideal experience that we want to have.

An alternative that you might consider, so that you can distribute your HTML5 game as a native application via the mobile application stores, is using hybrid mobile application frameworks such as Apache Cordova.

Using Hybrid Mobile Application Frameworks

Apache Cordova is an open source mobile application framework, released by Adobe Systems. It enables software programmers to build applications for mobile devices using HTML, CSS, and JavaScript instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone.

In addition to wrapping the HTML code into a native application, Cordova also provides access to native device features in JavaScript via a system of plug-ins, allowing programmers to use device features such as the accelerometer and camera. The resulting applications are considered hybrid, meaning that they are neither truly native mobile applications nor purely Web based. You can read more about Cordova at https://cordova.apache.org.

Installing Cordova is as simple as running a single npm command:

npm install -g cordova

Cordova lets you develop applications for multiple devices in a single build process. You can create a Cordova project and set it up using the following command:

cordova create frootwars

Once you run this command, Cordova will create a folder with some automatically generated content. One of these folders is the www folder, which contains a sample web application, with the typical index.html, JavaScript, and CSS files. You can use this template as a starting point to create a Cordova web application.

Now that you have a template, you can add support for the device platforms you want to build the game for using the following commands:

```
cd frootwars
```

```
cordova platform add android
cordova platform add ios
cordova platform add browser
```

Cordova provides a platform called browser to allow you to test your application deployment in the browser. It also allows you to build native applications for iOS, Android, Windows, Blackberry, and several other platforms.

You can test the sample application on the browser platform just by running the following command:

```
cordova run browser
```

This should open the sample application inside a browser window. The HTML code for this application is the same code you saw within the www folder. To convert your game to work for Cordova, you will need to place your code into the www folder and modify it to use some of the recommended tags and Cordova commands in the sample application.

You can continue testing your changes by running the browser platform. However, before you can actually build the application for the other platforms, you will need to set up your development environment for each platform. This might mean installing XCode (https://developer.apple.com/xcode/) for iOS devices, Android Studio (https://developer.android.com/studio/) for Android devices, or Visual Studio (https://www.visualstudio.com/) for Windows devices.

You can also install plug-ins for any device feature that you would like to access for your game, such as in-app purchases to monetize your game, accelerometer and vibration to allow better interactivity, or geolocation and camera to build the next Pokémon Go killer.

Unfortunately, teaching you to use all these features to build a game with Cordova is beyond the scope of this chapter, since this subject matter could fill an entire book on its own.

Once you are comfortable with building web games in HTML5 and are ready to venture into the world of hybrid applications, I would recommend that you read the Cordova documentation (https://cordova.apache.org/docs/en/latest/) and explore further on your own.

Before we wrap up this chapter on mobile development, I'd like to cover one more thing: optimizing your game assets for better performance on mobile devices.

Optimizing Game Assets for Mobile

When working on games for mobile devices, it is important to keep in mind the fact that mobile device users might be accessing you game via a slow Internet connection. Anything we can do to make the game-loading experience as painless as possible, by reducing the bandwidth usage or the loading and waiting time, will make a significant difference to the user experience.

While there are many things that we can do to optimize the game experience on slow connections, the most important things that we can do are

- *Loading screens with progress updates*: A loading screen lets players know that the game is doing something in the background and gives them a way to track the progress so they have a general idea of how long they will need to wait. If you make players wait for a long period of time without letting them know what is happening or how much longer they need to wait, they are likely to get frustrated and impatient. In extreme cases users might just close the browser without even trying your game because it took too long to load. Our game already implements a game loader with a progress bar, which automatically shows up any time assets are being downloaded.

- *Lazy loading of assets*: If your game has 50 levels, but the player is only about to play the first level, there is no point in making the player wait while the data for all 50 levels has been loaded. It makes sense to load the game data "lazily," as and when it is needed. Our game already intelligently manages this by loading common assets up front and then loading level-specific assets only when a level is started. This way the player never has to wait too long for something to happen.

- *Reducing size of images*: We briefly discussed using techniques like sprite sheets, which reduce network load by minimizing server calls and reducing the total amount of data transferred. In addition, you should ensure that the resolution and size of your game images are not unnecessarily large, and should ideally be as close to the actual resolution that your game needs. Finally, you should add processes like PNG optimization and compression into your workflow and build processes so that all assets are automatically compressed in your final game. We will look into ways of doing this in later chapters.

- *Reducing size of audio*: Most of us are used to using high-quality audio with multiple channels and high bitrates when we listen to music or watch movies. However, mobile games don't necessarily need so much audio detail. Using programs like Audacity (`www.audacityteam.org/`), you can convert your game audio to mono instead of stereo, and reduce the bitrate of the audio with almost no perceptible difference in quality on the phone. As an example, the code folder contains a low-bitrate version of the background music used in this game. Making these small changes reduced the size of the OGG file from nearly 4 megabytes to 1 megabyte, which can be a phenomenal difference for someone using a slow 2G connection and paying for each megabyte.

- *Compressing the code*: Code minifiers such as `html-minifier` and `node-minify` take your development code and convert it to an extremely compressed version with shorter variable names and all comments and unnecessary spaces removed. For HTML, it is even possible to compress linked asset files and place them inline. While this version of the code is very hard for humans to read, it is significantly smaller than the original and can be downloaded by the browser much faster. Again, this is something you should ideally build into your workflow and build process so it is automated. We will look at ways of doing this in later chapters.

Now these are some of the most important ways that you can improve the experience for mobile device users. While it might not always be feasible to do all of these in every game, every little bit you can do will make the experience much better for the players trying your game.

Summary

In this chapter we looked at the challenges involved in building games for mobile devices. We saw how a desktop game can easily be converted into a mobile device game, by converting the game we created in the previous chapters.

We started by making the game responsive so it automatically scales to fit devices with different sizes and aspect ratios. We then modified the game input handling to use the Touch API. We also used the wAudio.js library to integrate the Web Audio API for better sound support on mobile devices. We then made the game behave like an app by disabling default scroll and adding web app tags. Finally, we explored the idea of using hybrid application frameworks and looked at ways to optimize game data for a better mobile experience.

At this point, you should have a strong understanding of the complexity and the typical steps involved when building a professional mobile device game. When you start developing your own games, even if they are not physics games, you should be able to use this game that we have built as a decent starting template, since it covers all the essentials that you will need—menus, asset loaders, level selection, canvas animation, sound and music, mouse and touch input, and support for mobile devices.

Now with this solid foundation, we are ready to take on a much bigger challenge. In the next few chapters we will be building a complete real-time strategy game with a single-player campaign as well as multiplayer mode. So let's keep going.

■ ■ ■

Creating the RTS Game World

Real-time strategy (RTS) games combine fast-paced tactical combat, resource management, and economy building within a defined game world.

A typical RTS game consists of a map of a world with different units, buildings, and terrain, as well as an interface to control and manipulate these elements. The player uses the interface to handle tasks such as gathering resources, constructing buildings, and creating an army, and then manages the army to achieve a set of goals defined for each level.

Although these games have an extensive history, the RTS genre was largely popularized by the games released by Westwood Studios and Blizzard Entertainment in the 1990s. Westwood's *Dune II* and *Command & Conquer* series are considered classics that helped define the genre. With its engaging story line and addictive multiplayer mode, Blizzard's *StarCraft* went on to elevate RTS gaming to an e-sport with professional competitive tournaments held around the world.

HTML5 now makes it possible to bring this genre to the browser in a way that wasn't possible earlier. In fact, one of my better-known game programming–related achievements a few years ago was single-handedly re-creating the original *Command & Conquer* entirely in HTML5. While generating a lot of buzz on the Web, this project proved beyond a doubt that HTML5 was now ready for the next generation of games.

Over the next few chapters, we will use what you learned in previous chapters and build upon it to create our own RTS game. We will define a game world with buildings, units, and an overarching story line to create an engaging single-player campaign. We will then use HTML5 WebSockets to add real-time multiplayer support to our game.

Most of the artwork for this game has been provided by Daniel Cook (`www.lostgarden.com`), who originally designed this art for an unreleased RTS title called *Hard Vacuum*. We will be reusing the artwork that he has graciously shared but will create our own game concept. Our game, *Last Colony*, will be about a small band of survivors on a planet that has just been attacked. We will explore the story and gameplay in more detail over the next few chapters.

While developing this game, we will keep the code as generic and customizable as possible so that you can later reuse this code to build your own ideas. If you would like to follow along with the book, you can find all the necessary starting assets, including the images and the audio, inside the assets folder of this chapter's code.

So, let's get started.

Basic HTML Layout

Like the previous game we developed, *Froot Wars*, our RTS game will consist of several layers. The following are the first few layers that we will define:

- *Splash screen and main menu*: Shown when the game loads and allows the player to select campaign or multiplayer mode

- *Loading screen*: Shown whenever the game is loading assets

- *Mission screen*: Shown before a mission starts, with instructions for the mission

- *Game interface screen*: The main game screen that includes the map area and a dashboard for controlling the game

We will define more screens as needed in later chapters. We will be organizing all of the artwork inside an images folder. Unlike the previous game, we will break the JavaScript code into several files (such as buildings.js, vehicles.js, levels.js, and common.js) inside the js folder so as to make the code easier to maintain.

Creating the Splash Screen and Main Menu

We will start by creating an HTML file and adding the markup for our containers, as shown in Listing 6-1.

Listing 6-1. Basic HTML Skeleton with Layers Added (index.html)

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">

        <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width">

        <title>Last Colony</title>

        <script src="js/common.js" type="text/javascript"></script>
        <script src="js/game.js" type="text/javascript"></script>
        <script src="js/mouse.js" type="text/javascript"></script>
        <script src="js/singleplayer.js" type="text/javascript"></script>
        <script src="js/levels.js" type="text/javascript"></script>

        <link rel="stylesheet" href="styles.css" type="text/css">
    </head>
    <body>
        <div id="wrapper">
            <div id="gamecontainer">
                <div id="gamestartscreen" class="gamelayer">
                    <span class="game-title">LAST<br>COLONY</span>
                    <span class="game-option" onclick = "singleplayer.start();">Campaign</span>
                    <span class="game-option" onclick = "multiplayer.start();">Multiplayer</span>
                </div>

                <div id="loadingscreen" class="gamelayer">
                    <div id="loadingmessage"></div>
                </div>
            </div>
        </div>
    </body>
</html>
```

The code first refers to the external JavaScript and CSS files we will be using. We will be creating and implementing all these JavaScript files over the course of this game. Within a main `wrapper` div, we also define a `gamecontainer` div that contains our first two game layers: `gamestartscreen` and `loadingscreen`.

The next thing we will do is define the initial style for the game container inside `styles.css`, as shown in Listing 6-2.

Listing 6-2. Initial Style Sheet (styles.css) for Game Container and Layer

```css
body {
    background: #090009;

    /* Disable scroll bars */
    overflow: hidden;

    /* Disable long touch hold select on mobile browsers */
    -webkit-touch-callout: none !important;
}

#wrapper {
    position: absolute;

    /* Wrapper covers entire window height and width */
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;

    /* Prevent the ugly blue highlighting from accidental selection of text */
    user-select: none;
}

#gamecontainer {
    /* Start with a default width that we can change later */
    width: 640px;
    height: 480px;

    /* Use a wider splash screen and center it within the container */
    background: url("images/screens/splashscreen.png");
    background-position: center;
    background-repeat: no-repeat;

    /* Center the game container relative to outer wrapper */
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    transform-origin: center center;
}
```

```
.gamelayer {
    width: 100%;
    height: 100%;
    position: absolute;
    display: none;
}
```

In this code, we first start with setting the body background color and disabling the scrollbar and long press context menus for mobile devices.

Next, we center the game container within the wrapper div and assign a background splash screen. We use a wide splash screen image so our game can dynamically adjust to different aspect ratios later, just like we did in our previous game, Froot Wars. For now, however, we assign the container an initial size of 640px by 480px.

Finally, we set the gamelayer class to position all the game layers on top of each other, assign them the same dimensions as the container, and hide them by default.

When we load index.html in the browser, we should now see our new splash screen, as shown in Figure 6-1.
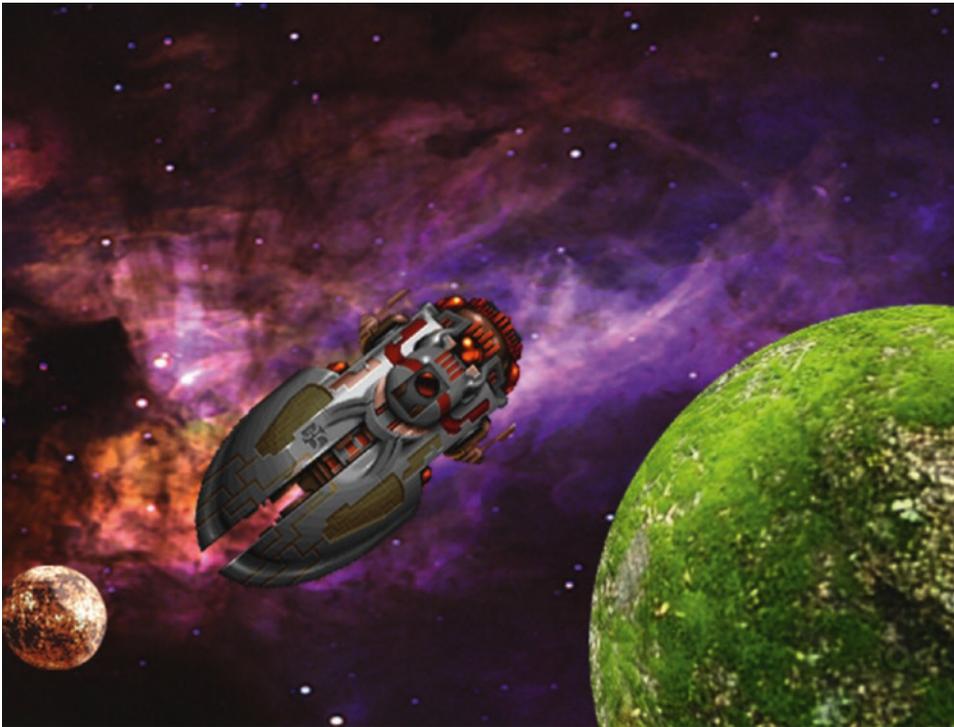
Figure 6-1. *The initial game splash screen*

Now that the splash screen is in place, we can implement the main menu screen and the game loading screen.

We will start by setting up the asset loader using the exact same code as we did in our previous game. We will place this code inside a separate file called common.js, as shown in Listing 6-3.

Listing 6-3. Setting Up the Asset Loader (common.js)

```
var loader = {
    loaded: true,
    loadedCount: 0, // Assets that have been loaded so far
    totalCount: 0, // Total number of assets that need loading

    init: function() {
        // Check for sound support
        var mp3Support, oggSupport;
        var audio = document.createElement("audio");

        if (audio.canPlayType) {
                // Currently canPlayType() returns:  "", "maybe", or "probably"
            mp3Support = "" !== audio.canPlayType("audio/mpeg");
            oggSupport = "" !== audio.canPlayType("audio/ogg; codecs=\"vorbis\"");
        } else {
            // The audio tag is not supported
            mp3Support = false;
            oggSupport = false;
        }

        // Check for ogg, then mp3, and finally set soundFileExtn to undefined
        loader.soundFileExtn = oggSupport ? ".ogg" : mp3Support ? ".mp3" : undefined;
    },

    loadImage: function(url) {
        this.loaded = false;
        this.totalCount++;

        game.showScreen("loadingscreen");

        var image = new Image();

        image.addEventListener("load", loader.itemLoaded, false);
        image.src = url;

        return image;
    },

    soundFileExtn: ".ogg",

    loadSound: function(url) {
        this.loaded = false;
        this.totalCount++;

        game.showScreen("loadingscreen");

        var audio = new Audio();
```

```
        audio.addEventListener("canplaythrough", loader.itemLoaded, false);
        audio.src = url + loader.soundFileExtn;

        return audio;
    },

    itemLoaded: function(ev) {
        // Stop listening for event type (load or canplaythrough) for this item now that it
        has been loaded
        ev.target.removeEventListener(ev.type, loader.itemLoaded, false);

        loader.loadedCount++;

        document.getElementById("loadingmessage").innerHTML = "Loaded " + loader.loadedCount +
        " of " + loader.totalCount;

        if (loader.loadedCount === loader.totalCount) {
            // Loader has loaded completely
            // Reset and clear the loader
            loader.loaded = true;
            loader.loadedCount = 0;
            loader.totalCount = 0;

            // Hide the loading screen
            game.hideScreen("loadingscreen");

            //and call the loader.onload method if it exists
            if (loader.onload) {
                loader.onload();
                loader.onload = undefined;
            }
        }
    }
};
```

Next, we will define our game object inside game.js, as shown in Listing 6-4.

Listing 6-4. Defining the game Object (game.js)

```
var game = {

    // Start initializing objects, preloading assets, and display start screen
    init: function() {
        // Initialize game objects
        loader.init();

        // Display the main game menu
        game.hideScreens();
        game.showScreen("gamestartscreen");
    },
```

```
    hideScreens: function() {
        var screens = document.getElementsByClassName("gamelayer");

        // Iterate through all the game layers and set their display to none
        for (let i = screens.length - 1; i >= 0; i--) {
            let screen = screens[i];

            screen.style.display = "none";
        }
    },

    hideScreen: function(id) {
        var screen = document.getElementById(id);

        screen.style.display = "none";
    },

    showScreen: function(id) {
        var screen = document.getElementById(id);

        screen.style.display = "block";
    },

    scale: 1,
    resize: function() {

        var maxWidth = window.innerWidth;
        var maxHeight = window.innerHeight;

        var scale = Math.min(maxWidth / 640, maxHeight / 480);

        var gameContainer = document.getElementById("gamecontainer");

        gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";

        game.scale = scale;

        // What is the maximum width we can set based on the current scale
        // Clamp the value between 640 and 1024
        var width = Math.max(640, Math.min(1024, maxWidth / scale ));

        // Apply this new width to game container and game canvas
        gameContainer.style.width = width + "px";
    },
};

/* Set up initial window event listeners */

// Initialize and resize the game once page has fully loaded
window.addEventListener("load", function() {
    game.resize();
    game.init();
}, false);
```

```
// Resize the game any time the window is resized
window.addEventListener("resize", function() {
    game.resize();
});
```

In this code, we create a `game` object with an `init()` method that first initializes our asset loader and then uses the `hideScreens()` and `showScreen()` methods to display the game start screen.

We also define a `resize()` method just like we did in our previous game. The method calculates the scale and maximum possible width for our game and sets the `gamecontainer` style accordingly.

Finally, we add event listeners to the `window` object to call these methods. We call `game.resize()` and `game.init()` once the window has loaded completely. We also call `game.resize()` whenever the window is resized.

Next, we need to append the CSS for the game starting screen and loading screen in `styles.css`, as shown in Listing 6-5.

Listing 6-5. Style for the Game Starting Screen and Loading Screen (styles.css)

```
/* Game Starting Menu Screen */

.game-title {
    position: absolute;

    top: 5%;
    right: 5%;
    text-align: right;
    width: 100%;

    font-family: "Courier New", Courier, monospace;
    font-size: 90px;
    line-height: 80px;

    color: white;
    text-shadow: -2px 0 purple, 0 2px purple, 2px 0 purple, 0 -2px purple;
}

.game-option {
    position: relative;
    top: 65%;
    left: 10%;
    display: block;

    font-family: "Courier New", Courier, monospace;
    font-size: 48px;
    color: white;
    text-shadow: -2px 0 purple, 0 2px purple, 2px 0 purple, 0 -2px purple;

    cursor: pointer;
}
```

```css
.game-option:hover {
    color: yellow;
}

/* Loading Screen */

#loadingscreen {
    background: rgba(100, 100, 100, 0.7);
}

#loadingmessage {
    position: relative;
    top: 400px;
    text-align: center;

    height: 48px;
    color: white;
    background: url("images/loader.gif") no-repeat center;
    font: 12px Arial;
}
```

When we open the game in the browser, we should see the starting screen with the main menu, as shown in Figure 6-2.

Figure 6-2. *The starting screen with the main menu*

You will notice that the game automatically scales and widens to fit the available screen area. The extra hidden portion of the splash screen image also automatically becomes visible as necessary.

The menu currently offers options for Campaign, which is our story-based single-player mode, and Multiplayer, which is our player-versus-player mode. You may have noticed in Listing 6-1 that the onclick handlers for these two options call the `singleplayer.start()` and `multiplayer.start()` methods, respectively. Right now, clicking the Campaign option won't do anything since we haven't yet implemented the `singleplayer.start()` method to start the single-player levels.

Before we can do so, however, we need to create our first single-player level.

Creating Our First Level

There are many viable approaches to defining maps or levels for our game.

One approach is to store all the information about the map terrain as metadata and then assemble all the necessary images for the terrain on the browser at runtime to draw the map. This approach, while slightly cumbersome, allows the use of sprite sheets for the map terrain, reducing the size of the map.

Another approach, which is slightly simpler, is to store the basic map as a large image with the terrain drawn out using our own level-designing tool. We then need to store only the location of the map image along with metadata such as game entities and mission objectives. This is the approach that we will be using for our game.

Map images can be designed very quickly by using general-purpose tile map–editing software such as Tiled (www.mapeditor.org). Tiled is an excellent free tool that is available for several operating systems including Windows, Mac, and Linux. Once you start the application, you can load the sprite sheet for the terrain as a tile set and then use it to draw the map as if you were using a painting application (see Figure 6-3).
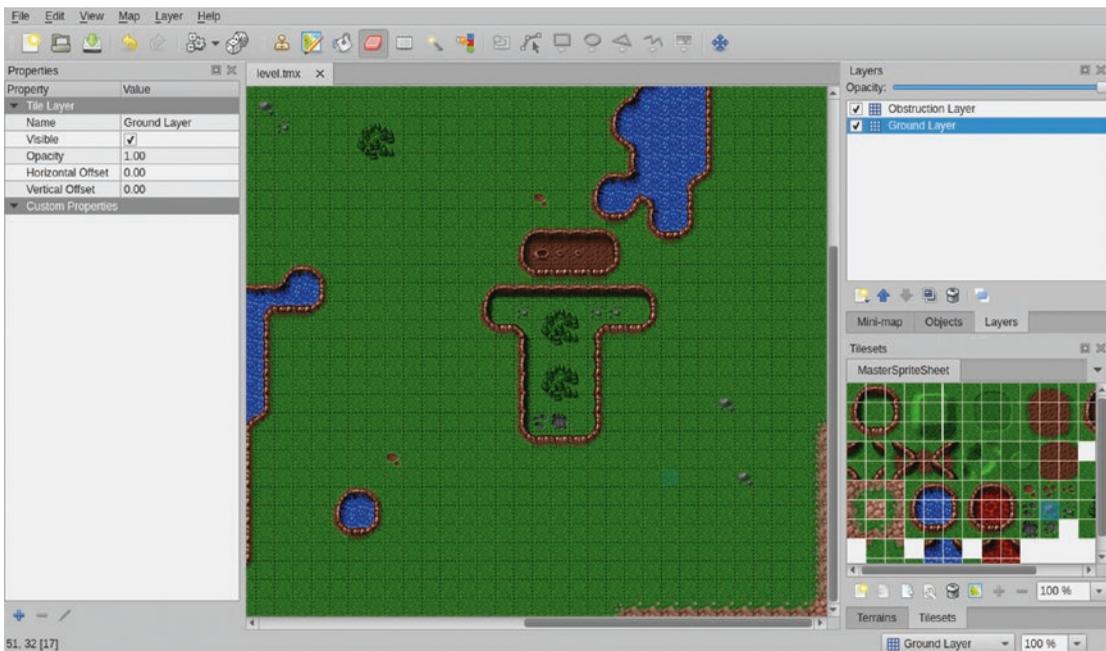
Figure 6-3. *Drawing a map using Tiled*

Note that we can use Tiled's layer feature to design the level in two layers. All the game terrain and obstructions are stored in a separate Obstruction layer. When the level JSON file is generated, we can use the metadata to identify areas of the map that are impassable or obstructed.

Once you draw the map, you can export it to several different file formats such as PNG images or JSON metadata.

You won't need to use this tool to follow along with the book since the maps we need for our game have already been generated. However, if you are considering developing your own game, I strongly recommend exploring Tiled's features.

All the files that you need, the exported level images, the JSON metadata, the master sprite sheet, and the Tiled project file are inside the level folder of this chapter's code. The exported images include a debug version of the map with all the grid lines between tiles drawn in.

The level folder also contains a convert-levels.js file, which is a Node.js script that takes the exported level.json file and creates a level-obstructed-terrain.json file for use within our game.

■ **Note**   The Tiled editor's JSON format contains references to the sprite sheet and offsets for all the tiles it uses. This means you can also use the JSON files to create maps that are assembled at runtime (instead of the preassembled ones we are creating).

Once we have our first map image designed, we will need to create the basic metadata describing the level. We will do this inside levels.js, as shown in Listing 6-6.

Listing 6-6. Defining the Basic Level Metadata (levels.js)

```
/* Details of the maps used by the levels */
var maps = {
    "plains": {
        "mapImage": "plains-debug.png",

        /* Terrain Data - Auto Generated By level/convert-levels.js */
        "mapGridWidth": 60,
        "mapGridHeight": 40,
        "mapObstructedTerrain": [[0, 0], [1, 0], [2, 0], [26, 0], [27, 0], /* Extremely huge
        array snipped for brevity */ [58, 39], [59, 39]],
    }
};

/* The actual levels played in the game */
var levels = {
    "singleplayer": [
        {
            "name": "Introduction",
            "briefing": "In this level you will learn how to pan across the map.\n\nDon't
            worry! We will be implementing more features soon.",

            /* Map Details */
            "mapName": "plains",
            "startX": 4,
            "startY": 4,
        }
    ],

    "multiplayer": [

    ]
};
```

We first define a maps object that contains details of the one map we have generated, named "plains." This includes the map image, and some terrain data that has been generated by convert-levels.js—the width and height of the map, as well as an extremely huge mapObstructedTerrain array, which contains the x and y coordinates of every grid square in the map that is impassable or obstructed.

I have snipped the array in Listing 6-6 because there is absolutely no point in showing you the entire array with several hundred numbers in it. You will find the complete mapObstructedTerrain array inside the level-obstructed-terrain.json file, as well as the finished game code. When you make your own maps using Tiled, you can use the convert-level.js script to generate the data for them.

The map image is broken down into a grid of squares 20 pixels wide by 20 pixels high (based on the size of the tiles we are using). For now, we are using a "debug" version of the map that has the grid drawn on top of the map. This will make it easier for us to position elements inside the level while we are building the game.

Next, we create a levels object that will contain all the levels within our game, with arrays for single-player and multiplayer.

The singleplayer array currently contains details for only one level. This array will eventually contain all our single-player campaign levels in chronological order. When the single-player campaign is started, the singleplayer object will load the first level in this array and then proceed down the list as the player completes each level.

The details that we store for the level include the level name and a mission briefing that we will display before we start the level.

We then refer to the plains map that we have defined earlier. By using this system of separating maps and levels, we can have multiple levels share the same map as needed, depending on the game's story line.

The starting map coordinates (startX and startY) let us decide where to position the screen on the map when we start the level using the grid coordinates.

Now that we have a simple map defined, we will set up the singleplayer object to display the mission briefing screen.

Loading the Mission Briefing Screen

The first thing we will do is add the HTML code for the mission briefing screen into the gamecontainer div within our HTML file. The gamecontainer div will now look like Listing 6-7.

Listing 6-7. Adding the Mission Briefing Screen (index.html)

```
<div id="gamecontainer">
    <div id="gamestartscreen" class="gamelayer">
        <span class="game-title">LAST<br>COLONY</span>
        <span class="game-option" onclick = "singleplayer.start();">Campaign</span>
        <span class="game-option" onclick = "multiplayer.start();">Multiplayer</span>
    </div>

    <div id="missionbriefingscreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-briefing.png" class="right-panel"
        draggable="false">
        <input type="button" id="entermission" onclick = "singleplayer.play();">
        <input type="button" id="exitmission" onclick = "singleplayer.exit();">
        <div id="missionbriefing"></div>
    </div>
```

```
    <div id="loadingscreen" class="gamelayer">
        <div id="loadingmessage"></div>
    </div>
</div>
```

The missionscreen div contains two buttons; they are for entering the mission screen and exiting the mission screen. It also contains a missionbriefing div that we will use to display the briefing message. Additionally it contains two images for the left and right side of the interface background area. We set the draggable attribute of these images to false to prevent them from being dragged around if the player accidentally clicks one of them.

Now that we have the HTML markup in place, we need to add the CSS styles for the mission screen into styles.css, as shown in Listing 6-8.

Listing 6-8. CSS Style for Mission Screen

```
/* Mission Briefing Screen */

#missionbriefingscreen {
    background: url("images/screens/interface-middle.png");
}

input[type="button"] {
    border-width: 0;
    outline: none;

    background-color: transparent;
    background-repeat: no-repeat;
    background-image: url("images/buttons.png");

    cursor: pointer;
}

.left-panel {
    left: 0;
    top: 0;
    position: absolute;
}

.right-panel {
    right: 0;
    top: 0;
    position: absolute;
}

#entermission {
    height: 52px;
    width: 188px;

    position: absolute;
    top: 82px;
    left: 3px;

    background-position: -4px -4px;
}
```

```
#entermission:disabled, #entermission:active {
    background-position: -196px -4px;
}

#exitmission {
    height: 52px;
    width: 72px;

    position: absolute;
    top: 82px;
    right: 164px;

    background-position: -4px -64px;
}

#exitmission:disabled, #exitmission:active {
    background-position: -84px -64px;
}

#missionbriefing {

    position: absolute;

    top: 170px;
    left: 40px;
    right: 220px;
    height: 270px;

    text-align: justify;

    color: rgb(130, 150, 162);
    text-shadow: -1px 1px black;

    font-size: 16px;
    font-family: "Courier New", Courier, monospace;
}
```

We define a new background for the mission briefing screen that fits into the center, behind the left and right images defined in the HTML. This center background automatically repeats itself to fit all available space. This way, the briefing screen can automatically adjust for different aspect ratios by keeping the left and right side of the interface the same size and expanding the center area as needed to adjust for different aspect ratios.

We then position the button and div elements to fit on top of the background. We keep different images for the enabled and disabled states of the buttons but store all of these sprites in a single sprite-sheet image file (buttons.png). Note that we specify left and right positions so that the buttons and briefing area automatically position and scale appropriately when the game container width changes.

Now that the mission briefing layer is in place, we will implement the singleplayer object inside singleplayer.js, as shown in Listing 6-9.

Listing 6-9. *Implementing the Basic singleplayer Object (singleplayer.js)*

```javascript
var singleplayer = {

    // Begin single-player campaign
    start: function() {
        // Hide the starting menu screen
        game.hideScreens();

        // Begin with the first level
        singleplayer.currentLevel = 0;

        // Start initializing the level
        singleplayer.initLevel();
    },

    currentLevel: 0,
    initLevel: function() {
        game.type = "singleplayer";
        game.team = "blue";

        // Don't allow player to enter mission until all assets for the level are loaded
        var enterMissionButton = document.getElementById("entermission");

        enterMissionButton.disabled = true;

        // Load all the items for the level
        var level = levels.singleplayer[singleplayer.currentLevel];

        game.loadLevelData(level);

        // Enable the Enter Mission button once all assets are loaded
        loader.onload = function() {
            enterMissionButton.disabled = false;
        };

        // Update the mission briefing text and show briefing screen
        this.showMissionBriefing(level.briefing);
    },

    showMissionBriefing: function(briefing) {
        var missionBriefingText = document.getElementById("missionbriefing");

        // Replace \n in briefing text with two <br> to create next paragraph
        missionBriefingText.innerHTML = briefing.replace(/\n/g, "<br><br>");

        // Display the mission briefing screen
        game.showScreen("missionbriefingscreen");
    },
```

```
    exit: function() {
        // Display the main game menu
        game.hideScreens();
        game.showScreen("gamestartscreen");
    },

};
```

We define a `singleplayer` object with four methods: `start()`, `initLevel()`, `showMissionBriefing()`, and `exit()`.

The `start()` method first hides all game layers and sets `singleplayer.currentLevel` to 0, which refers to the first level in the `maps.singleplayer` array that we defined earlier. Finally, it calls the `singleplayer.initLevel()` method that we will call every time we want to load a level.

The `initLevel()` method first sets the `game.type` and `game.team` variables to `singleplayer` and `blue`, respectively. We will use these values later once the game starts running. It then temporarily disables the Enter Mission button on the screen and starts loading the level assets. Once the assets are loaded, the Enter Mission button is enabled so that the player can click it and enter the game. Finally, it calls the `showMissionBriefing()` method, which puts the level briefing inside the `missionbriefing` div and displays the `missionbriefingscreen` div.

The `exit()` method hides all the game layers and takes us back to the main menu.

■ **Note**   We replace carriage returns with `<br>` tags so that they show up in the HTML. This way, we can easily break out the mission briefing into multiple paragraphs if we want.

Next we will define the `loadLevelData()` method inside the game object as shown in Listing 6-10.

Listing 6-10. Loading the Level (game.js)

```
loadLevelData: function(level) {
    game.currentLevel = level;
    game.currentMap = maps[level.mapName];

    // Load all the assets for the level starting with the map image
    game.currentMapImage = loader.loadImage("images/maps/" + maps[level.mapName].mapImage);
},
```

For now, we just store the level and map objects and load the current level's map image. This method will eventually load all the assets for a given level.

When we load the game in the browser and click the Campaign option, we should see the mission briefing screen for the first level, as shown in Figure 6-4.
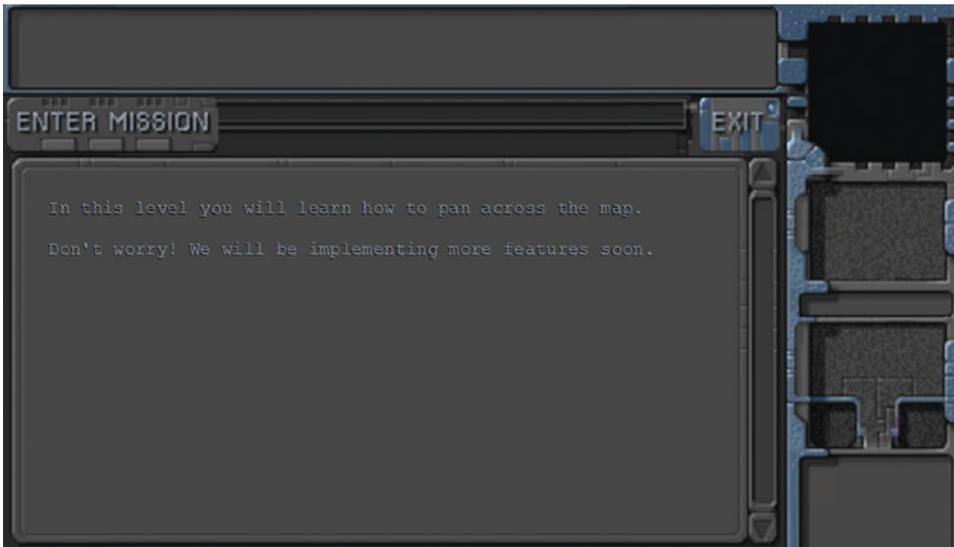
Figure 6-4. *The mission briefing screen for our first level*

The advantage of displaying the briefing screen while loading the assets in the background is that players can spend their time reading the mission briefing while waiting for all the assets to load. You will notice that the screen automatically adjusts to different aspect ratios and screen sizes by expanding the center region while keeping the left and right sides the same.

Clicking the Exit button should take us back to the main menu. Once the level data has loaded completely, the enter mission button will get enabled. We still can't enter the mission until we implement the actual game interface and the game animation and drawing loops, which is what we will be doing next.

Implementing the Game Interface

The first thing we will do is add the HTML markup for the game interface screen into the `gamecontainer div` in our HTML file. The `gamecontainer div` will now look like Listing 6-11.

Listing 6-11. Adding the Game Interface Layer (index.html)

```
<div id="gamecontainer">
    <div id="gamestartscreen" class="gamelayer">
        <span class="game-title">LAST<br>COLONY</span>
        <span class="game-option" onclick = "singleplayer.start();">Campaign</span>
        <span class="game-option" onclick = "multiplayer.start();">Multiplayer</span>
    </div>

    <div id="missionbriefingscreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-briefing.png" class="right-panel"
        draggable="false">
        <input type="button" id="entermission" onclick = "singleplayer.play();">
        <input type="button" id="exitmission" onclick = "singleplayer.exit();">
        <div id="missionbriefing"></div>
    </div>
```

```
    <div id="gameinterfacescreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-game.png" class="right-panel"
        draggable="false">
        <div id="gamemessages"></div>
        <div id="callerpicture"></div>
        <div id="cash"></div>
        <div id="sidebarbuttons">
        </div>
        <canvas id="gamebackgroundcanvas"></canvas>
        <canvas id="gameforegroundcanvas"></canvas>
    </div>

    <div id="loadingscreen" class="gamelayer">
        <div id="loadingmessage"></div>
    </div>
</div>
```

Our game interface layer consists of several different areas within it:

- *Game area*: This is where the player can see the map and interact with the buildings, units, and other entities within the game. This is implemented using two canvas elements: gamebackgroundcanvas for the map and gameforegroundcanvas for the entities inside the level (such as buildings and units).

- *Game messages*: This is where the player can see system notifications or story-driven messages.

- *Caller picture*: This is where the player will see profile pictures of the person sending story-driven messages.

- *Cash*: This is where players will see their cash reserves.

- *Sidebar buttons*: This is where players will see buttons they can use for creating units and buildings within the game.

We also use left and right background images just as we did in the mission briefing screen.

Now that the HTML is in place, we will add the CSS for the game interface screen to styles.css, as shown in Listing 6-12.

Listing 6-12. CSS for the Game Interface Screen

```
/* Game Interface Screen */

#gameinterfacescreen {
    background: url("images/screens/interface-middle.png");
}

#gameinterfacescreen #gamemessages {
    position: absolute;
    padding: 5px;

    top: 4px;
    left: 5px;
```

```
    right: 168px;
    height: 60px;

    color: rgb(130, 150, 162);

    overflow: hidden;
    font-size: 13px;
    font-family: "Courier New", Courier, monospace;
}

#gamemessages span {
    color: white;
}

#callerpicture {
    position: absolute;

    right: 20px;
    top: 155px;
    width: 114px;
    height: 72px;

    overflow: hidden;
}

#cash {
    width: 120px;
    height: 22px;
    position: absolute;
    right: 20px;
    top: 241px;

    color: rgb(130, 150, 162);
    overflow: hidden;
    font-size: 14px;
    font-family: "Courier New", Courier, monospace;
    text-align: right;
}

#gameinterfacescreen canvas {
    position: absolute;
    top: 79px;
    left: 0;
}
```

We start by defining a background for the center of the `gameinterfacescreen div`, just as we did for the `gamebriefingscreen div`, and then position the various other elements at the appropriate locations within the interface area. Both game canvas elements are positioned at the same location, with `foregroundcanvas` on top of `backgroundcanvas`.

Next we will modify the `init()` method of the game object to initialize the canvas elements when the game is initialized, as shown in Listing 6-13.

Listing 6-13. Initializing the Canvas Elements (game.js)

```
// Start initializing objects, preloading assets, and display start screen
init: function() {
    // Initialize objects
    loader.init();

    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},

canvasWidth: 480,
canvasHeight: 400,

initCanvases: function() {
    game.backgroundCanvas = document.getElementById("gamebackgroundcanvas");
    game.backgroundContext = game.backgroundCanvas.getContext("2d");

    game.foregroundCanvas = document.getElementById("gameforegroundcanvas");
    game.foregroundContext = game.foregroundCanvas.getContext("2d");

    game.foregroundCanvas.width = game.canvasWidth;
    game.backgroundCanvas.width = game.canvasWidth;

    game.foregroundCanvas.height = game.canvasHeight;
    game.backgroundCanvas.height = game.canvasHeight;
},
```

We add a call to the initCanvases() method, which stores the canvas and context objects and sets their initial width and height.

We also need to handle resizing the canvas elements whenever the window size changes. We will do this by modifying the game.resize() method as shown in Listing 6-14.

Listing 6-14. Resizing the Canvas Elements (game.js)

```
resize: function() {

    var maxWidth = window.innerWidth;
    var maxHeight = window.innerHeight;

    var scale = Math.min(maxWidth / 640, maxHeight / 480);

    var gameContainer = document.getElementById("gamecontainer");

    gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";

    game.scale = scale;
```

```
    // What is the maximum width we can set based on the current scale
    // Clamp the value between 640 and 1024
    var width = Math.max(640, Math.min(1024, maxWidth / scale ));

    // Apply this new width to game container and game canvas
    gameContainer.style.width = width + "px";

    // Subtract 160px for the sidebar
    var canvasWidth = width - 160;

    // Set a flag in case the canvas was resized
    if (game.canvasWidth !== canvasWidth) {
        game.canvasWidth = canvasWidth;
        game.canvasResized = true;
    }

},
```

We calculate the new width for the canvas based on the container width that we calculated earlier. If the value has changed, we also set a canvasResized flag to true. We will use this flag inside our drawing loop to decide whether we need to redraw parts of the game.

Now we will implement animation and drawing loops, as well as a game.start() method in our game, as shown in Listing 6-15.

Listing 6-15. Adding Animation and Drawing Loops and Starting the Game(game.js)

```
start: function() {
    // Display the game interface
    game.hideScreens();
    game.showScreen("gameinterfacescreen");

    game.running = true;
    game.refreshBackground = true;
    game.canvasResized = true;

    game.drawingLoop();
},

// A control loop that runs at a fixed period of time
animationTimeout: 100, // 100 milliseconds or 10 times a second

animationLoop: function() {
},

// The map is broken into square tiles of this size (20 pixels x 20 pixels)
gridSize: 20,
// X & Y panning offsets for the map
offsetX: 0,
offsetY: 0,
```

```
drawingLoop: function() {
    // Draw the background whenever necessary
    game.drawBackground();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},

drawBackground: function() {
    // Since drawing the background map is a fairly large operation,
    // we only redraw the background if it changes (due to panning or resizing)
    if (game.refreshBackground || game.canvasResized) {
        if (game.canvasResized) {
            game.backgroundCanvas.width = game.canvasWidth;
            game.foregroundCanvas.width = game.canvasWidth;

            // Ensure the resizing doesn't cause the map to pan out of bounds
            if (game.offsetX + game.canvasWidth > game.currentMapImage.width) {
                game.offsetX = game.currentMapImage.width - game.canvasWidth;
            }

            if (game.offsetY + game.canvasHeight > game.currentMapImage.height) {
                game.offsetY = game.currentMapImage.height - game.canvasHeight;
            }

            game.canvasResized = false;
        }

        game.backgroundContext.drawImage(game.currentMapImage, game.offsetX, game.offsetY,
        game.canvasWidth, game.canvasHeight, 0, 0, game.canvasWidth, game.canvasHeight);
        game.refreshBackground = false;
    }
},
```

We define a start() method that hides other layers and displays the game interface screen. It then sets the game.running, game.backgroundChanged, and game.canvasResized variables to true for later use. Finally, we call the drawingLoop() method for the first time.

We also define two different methods called animationLoop() and drawingLoop(). The animationLoop() method will handle all control-related and animation-related logic and needs to be run at a fixed interval (defined in animationTimeout). An animation timeout of 100 milliseconds is usually sufficient for a fairly smooth game. For now the animationLoop() method is empty. The drawingLoop() method handles the actual drawing of all the game elements onto the two game canvas objects. The method is called using requestAnimationFrame() and will run as many times a second as the browser allows.

We start by calling the game.drawBackground() method, which will draw the map on the background canvas whenever necessary.

We then call the drawingLoop() method again using requestAnimationFrame() if the game is still running. This way, once the drawingLoop() method has been called once, it will keep running and drawing the game until game.running becomes false.

In the drawBackground() method, the first thing that we do is check the canvasResized and refreshBackground flags to determine whether the background needs to be redrawn.

If the canvas was resized, we also check and adjust the panning offsets to ensure that the screen doesn't pan outside the map bounds. We then draw the map image (stored in currentMapImage when the map was loaded) using the panning offsets (offsetX, offsetY) and the canvas dimensions.

Finally, we reset both the flags to false. We use this optimization so that we don't need to redraw the entire background after each refresh, and only do so when something has actually changed.

The reason we break out the code into two different timer loops is because the animation code will contain logic such as pathfinding, processing commands, and changing the animation states of sprites, which will not need to be executed as often as the drawing code.

The animation code will also control the actual movement of units. By keeping this code independent of the drawing code, we ensure that units will move the same amount after each animation cycle. This will become very important when we handle multiplayer mode and need the game state to be synchronized across different machines. If we aren't careful, slight calculation differences between browsers and machines can cause unexpected results such as a bullet hitting an enemy unit in one browser but missing the enemy in the other browser.

Now that we have these loops in place, we will finally implement the singleplayer.play() method inside singleplayer.js, as shown in Listing 6-16.

Listing 6-16. The singleplayer.play() Method (singleplayer.js)

```
play: function() {
    // Run the animation loop once
    game.animationLoop();

    // Start the animation loop interval
    game.animationInterval = setInterval(game.animationLoop, game.animationTimeout);

    game.start();
},
```

This method is fairly simple. It calls the game.animationLoop() method for the first time and then uses the setInterval() method to call the method every 100 milliseconds (as set in game.animationTimeout). Finally, it calls the game.start() method that we defined earlier. The game.animationLoop() method is currently empty, but we will start using it when we add entities to our game in the next chapter.

If we run the game code we have so far, we should be able to click the Enter Mission button at the mission briefing screen and then see the game interface screen with the map loaded, as shown in Figure 6-5.

Figure 6-5. *The game interface screen with the first map loaded*

You can even resize the browser window and see that the game automatically shows more of the map as the window gets wider.

One thing you might notice is that the game starts off at the top-left corner of the map. To use the initial map offset settings that we provided in levels.js, we will need to load the offset values when we start the level. We will do this by modifying the initLevel() method in singleplayer.js, as shown in Listing 6-17.

Listing 6-17. Setting the Map Offset Inside initLevel() (singleplayer.js)

```
initLevel: function() {
    game.type = "singleplayer";
    game.team = "blue";

    // Don't allow player to enter mission until all assets for the level are loaded
    var enterMissionButton = document.getElementById("entermission");

    enterMissionButton.disabled = true;

    // Load all the items for the level
    var level = levels.singleplayer[singleplayer.currentLevel];

    game.loadLevelData(level);

    // Set player starting location
    game.offsetX = level.startX * game.gridSize;
    game.offsetY = level.startY * game.gridSize;

    // Enable the Enter Mission button once all assets are loaded
    loader.onload = function() {
        enterMissionButton.disabled = false;
    };
```

```
        // Update the mission briefing text and show briefing screen
        this.showMissionBriefing(level.briefing);
},
```

We added just two new lines to set `game.offsetX` and `game.offsetY` based on `level.startX` and `level.startY`. This time when we load the map, it loads at the offset we defined in the map.

Now that we have finished loading the map, we will implement panning around the map using the mouse.

Implementing Map Panning

The first thing we will do is set up mouse input by creating a `mouse` object inside `mouse.js` (see Listing 6-18).

Listing 6-18. Setting Up the mouse Object

```
var mouse = {
    init: function() {
        // Listen for mouse events on the game foreground canvas
        let canvas = document.getElementById("gameforegroundcanvas");

        canvas.addEventListener("mousemove", mouse.mousemovehandler, false);

        canvas.addEventListener("mouseenter", mouse.mouseenterhandler, false);
        canvas.addEventListener("mouseout", mouse.mouseouthandler, false);

        mouse.canvas = canvas;
    },

    // x,y coordinates of mouse relative to top-left corner of canvas
    x: 0,
    y: 0,

    // x,y coordinates of mouse relative to top-left corner of game map
    gameX: 0,
    gameY: 0,

    // game grid x,y coordinates of mouse
    gridX: 0,
    gridY: 0,

    calculateGameCoordinates: function() {
        mouse.gameX = mouse.x + game.offsetX ;
        mouse.gameY = mouse.y + game.offsetY;

        mouse.gridX = Math.floor((mouse.gameX) / game.gridSize);
        mouse.gridY = Math.floor((mouse.gameY) / game.gridSize);
    },
```

```
    setCoordinates: function(clientX, clientY) {
        let offset = mouse.canvas.getBoundingClientRect();

        mouse.x = (clientX - offset.left) / game.scale;
        mouse.y = (clientY - offset.top) / game.scale;

        mouse.calculateGameCoordinates();
    },

    // Is the mouse inside the canvas region
    insideCanvas: false,

    mousemovehandler: function(ev) {
        mouse.insideCanvas = true;
        mouse.setCoordinates(ev.clientX, ev.clientY);
    },

    mouseenterhandler: function() {
        mouse.insideCanvas = true;
    },

    mouseouthandler: function() {
        mouse.insideCanvas = false;
    },
};
```

We start by defining an `init()` method that assigns event listeners on the foreground canvas for a few mouse events: `mousemove`, `mouseenter`, and `mouseout`. We also save a reference to the canvas in `mouse.canvas`.

Next, we define variables to store the mouse coordinates relative to the canvas (`x,y`), relative to the map (`gameX,gameY`), and in terms of the map grid (`gridX,gridY`). We also define several variables to store the mouse state (`buttonPressed`, `dragSelect`, and `insideCanvas`). We also define a method called `calculateGameCoordinates()` that converts the mouse x and y coordinates to game coordinates.

Next, we define a helper method called `setCoordinates()`, which is called whenever the mouse is moved. We use the mouse event `clientX` and `clientY` properties, which are coordinates relative to the top of the window, and convert them to be relative to the game canvas, while adjusting for the game scale. We then call `calculateGameCoordinates()` so that the game-related coordinates are updated as well.

Finally, we define the actual event handler methods: `mousemovehandler`, `mouseenterhandler`, and `mouseouthandler`.

Whenever the mouse is moved, we set the `insideCanvas` flag to `true` since we know the mouse is somewhere inside the canvas. We then call `setCoordinates()` to update the stored mouse coordinates.

When the mouse enters the canvas, we set the `insideCanvas` flag to `true`, and when the mouse leaves the canvas, we set this flag to `false`. This way we can always know whether the mouse is inside or outside our game canvas area using `mouse.insideCanvas`, and can handle it as needed.

Now that we have set up our `mouse` object, we will modify our `game` object inside `game.js` to use the mouse. The first thing we need to do is call the `mouse.init()` method from inside the `game.init()` method. The updated `game.init()` method will look like Listing 6-19.

Listing 6-19. Calling mouse.init() from Inside game.init() (game.js)

```
init: function() {
    // Initialize objects
    loader.init();
    mouse.init();

    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

Next we will define a handlePanning() method inside the game object, as shown in Listing 6-20.

Listing 6-20. Defining the handlePanning() Method (game.js)

```
// Distance from edge of canvas at which panning starts
panningThreshold: 80,
// The maximum distance to pan in a single drawing loop
maximumPanDistance: 10,

handlePanning: function() {

    // Do not pan if mouse leaves the canvas
    if (!mouse.insideCanvas) {
        return;
    }

    if (mouse.x <= game.panningThreshold) {
        // Mouse is at the left edge of the game area. Pan to the left.
        let panDistance = game.offsetX;

        if (panDistance > 0) {
            game.offsetX -= Math.min(panDistance, game.maximumPanDistance);
            game.refreshBackground = true;
        }
    } else if (mouse.x >= game.canvasWidth - game.panningThreshold) {
        // Mouse is at the right edge of the game area. Pan to the right.
        let panDistance = game.currentMapImage.width - game.canvasWidth - game.offsetX;

        if (panDistance > 0) {
            game.offsetX += Math.min(panDistance, game.maximumPanDistance);
            game.refreshBackground = true;
        }
    }
```

```
    if (mouse.y <= game.panningThreshold) {
        // Mouse is at the top edge of the game area. Pan upwards.
        let panDistance = game.offsetY;

        if (panDistance > 0) {
            game.offsetY -= Math.min(panDistance, game.maximumPanDistance);
            game.refreshBackground = true;
        }
    } else if (mouse.y >= game.canvasHeight - game.panningThreshold) {
        // Mouse is at the bottom edge of the game area. Pan downwards.
        let panDistance = game.currentMapImage.height - game.offsetY - game.canvasHeight;

        if (panDistance > 0) {
            game.offsetY += Math.min(panDistance, game.maximumPanDistance);
            game.refreshBackground = true;
        }
    }

    if (game.refreshBackground) {
        // Update mouse game coordinates based on new game offsetX and offsetY
        mouse.calculateGameCoordinates();
    }
},
```

We start by defining two new variables, panningThreshold and maximumPanDistance, that store how close to the canvas edge the mouse cursor needs to be for panning to occur and how fast the panning can be.

The handlePanning() method itself checks to see whether the mouse is inside the canvas, near any of the edges of the canvas, and that there is still some map left to pan. If all the conditions are met, we adjust the offsets in the appropriate direction by the panning distance, and set the refreshBackground flag to true. This will let the drawBackground() method know that the background needs to be redrawn.

Finally, if the map was panned, we also refresh the mouse game coordinates since they will change any time the map pans.

The last change that we will make to the game object is calling the handlePanning() method from inside game.drawingLoop(). The final drawingLoop() method will look like Listing 6-21.

Listing 6-21. Calling game.handlePanning()(game.js)

```
drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();

    // Draw the background whenever necessary
    game.drawBackground();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

At this point, if we run the game, we should be able to pan around the map by moving the mouse near the edges of the canvas so that we can explore the entire map, as shown in Figure 6-6.

Figure 6-6. *Panning around the map*

Summary

In this chapter, we set out to develop the basic framework for our RTS game.

Just like in Chapter 2, we implemented a splash screen and a starting menu. We also used ideas from Chapter 5 to make the game responsive and automatically adjust to different window sizes and aspect ratios.

We looked at creating a map using the Tiled editor and exporting it as an image. We then created our first level by combining the map image with some basic level metadata.

We implemented a `singleplayer` object that loads map data and displays a mission briefing screen. We then created the game interface screen and set up the animation and drawing loops for the game so we could load and see the initial map on the canvas. Finally, we captured and used mouse events to let the user pan around the level.

While we have a lot of the essential elements of our game world in place, we are still missing the actual entities to interact with, such as buildings and vehicles.

In the next chapter, we will start adding these different entities to our level. We will draw them on the screen using sprite sheets and animation states. We will then set up a framework for selecting these entities so we can interact with them.

CHAPTER 7

■ ■ ■

Adding Entities to Our World

In the previous chapter, we put together the basic framework for our RTS game. We loaded a level and panned around using the mouse.

In this chapter, we will build upon that by adding entities to our game world. We will build a general framework that will allow us to easily add entities such as buildings and units to a level. Finally, we will add the ability for the player to select these entities using the mouse.

Let's get started. We will use the code from Chapter 6 as our starting point.

Defining Entities

These are the game entities we will be adding to our game:

- *Buildings*: Our game will have four types of buildings.

    - *Base*: Primary structure used to construct other buildings

    - *Starport*: Used to teleport in both ground vehicles and aircraft

    - *Harvester*: Used to extract resources from oil fields

    - *Ground turret*: Defensive structure used to guard against ground vehicles

- *Vehicles*: Our game will have four types of vehicles.

    - *Transport*: An unarmed vehicle used to transport supplies and people

    - *Harvester*: A mobile unit that deploys into the harvester building at an oil field

    - *Scout tank*: A light, fast-moving tank used for scouting

    - *Heavy tank*: A slower tank with heavier armor and weaponry

- *Aircraft*: Our game will have two types of aircraft.

    - *Chopper*: A slow-moving craft that can attack both land and air

    - *Wraith*: A fast-moving jet aircraft that can attack only in the air

- *Terrain*: Apart from the terrain already integrated in our map, we will define two additional types of terrain.

    - *Oil field*: Source of mineral resources that can be extracted for cash by deploying a harvester

    - *Rocks*: Interesting rock formations

We will store our entity types in separate JavaScript files to make the code easier to maintain. The first thing we will do is add references to the new JavaScript files inside the head section of our HTML file as shown in Listing 7-1.

Listing 7-1. Adding References to Entities (index.html)

```
<script src="js/buildings.js" type="text/javascript"></script>
<script src="js/vehicles.js" type="text/javascript"></script>
<script src="js/aircraft.js" type="text/javascript"></script>
<script src="js/terrain.js" type="text/javascript"></script>
```

With this code in place, we are now ready to start defining our first set of entities, the buildings, starting with the main base.

Defining Our First Entity: The Main Base

The first building we will define is the main base. Unlike other buildings in the game that can be constructed by the player, the main base will always be preconstructed before the level starts. The base allows the player to teleport in other buildings as long as the player has sufficient resources.

The base will consist of a single sprite sheet image that contains different animation states for the base (see Figure 7-1).

Figure 7-1. Sprite sheet for the base

As you can see, the sheet consists of two different rows of frames for the blue and green teams. The sprites in this case consist of a default animation (four frames), a damaged base (one frame), and finally an animation for when the base is constructing a building (three frames). We will be using similar sprite sheets and a common loading and drawing mechanism for all the entities within our game.

The first thing we will do is define a buildings object inside buildings.js, as shown in Listing 7-2.

Listing 7-2. Defining the buildings Object(buildings.js)

```
var buildings = {
    list: {
        "base": {
            name: "base",
            // Properties for drawing the object
```

```
            // Dimensions of the individual sprite
            pixelWidth: 60,
            pixelHeight: 60,

            // Dimensions of the base area
            baseWidth: 40,
            baseHeight: 40,

            // Offset of the base area from the top-left corner of the sprite
            pixelOffsetX: 0,
            pixelOffsetY: 20,

            // Grid squares necessary for constructing the building
            buildableGrid: [
                [1, 1],
                [1, 1]
            ],

            // Grid squares that are passable or obstructed for pathfinding
            passableGrid: [
                [1, 1],
                [1, 1]
            ],

            // How far the building can "see" through fog of war
            sight: 3,

            // Maximum possible life
            hitPoints: 500,

            cost: 5000,

            spriteImages: [
                { name: "healthy", count: 4 },
                { name: "damaged", count: 1 },
                { name: "constructing", count: 3 }
            ],
        },
    },

    defaults: {
        type: "buildings",
    },

    load: loadItem,
    add: addItem,

};
```

The `buildings` object uses a specific design pattern that we will be following for all our game entities.

First, we store the definition for all the different types of buildings inside a `list` array. Each of these definition objects contain properties specific to the type of building. These include properties for drawing the object (such as `pixelWidth`), properties for pathfinding (`passableGrid`), general properties such as `hitPoints` and `cost`, and finally the list of sprite images with names and sprite counts for each animation. So far, we have defined only our base entity inside the `list` array.

Next, we store properties and methods common to all the buildings inside the `defaults` object. This will include properties such as `type` as well as methods such as `processActions()` and `drawSprite()`, which we will be defining later.

Note that we can always override these default methods, by defining a method with the same name in the entity definition within the `list` array.

Finally, we have the methods `load()` and `add()`, which are necessary for the creation of our entities.

The `load()` method will load the sprite sheet and definitions for a given entity, while the `add()` method will create a new instance of a given entity to be added to the game. We currently point these toward methods called `loadItem()` and `addItem()` that we have not yet defined.

Now that we have a basic building definition in place, we will define the `loadItem()` and `addItem()` methods inside `common.js` so that they can be used by all the entities (see Listing 7-3).

Listing 7-3. Defining the loadItem() and addItem() Methods (common.js)

```
// The default load() method used by all our game entities
function loadItem(name) {
    var item = this.list[name];

    // If the item sprite array has already been loaded, then no need to do it again
    if (item.spriteArray) {
        return;
    }

    item.spriteSheet = loader.loadImage("images/" + this.defaults.type + "/" + name + ".png");
    item.spriteArray = [];
    item.spriteCount = 0;

    item.spriteImages.forEach(function(spriteImage) {

        let constructImageCount = spriteImage.count;
        let constructDirectionCount = spriteImage.directions;

        if (constructDirectionCount) {
            // If the spriteImage has directions defined, store sprites for each direction
            in spriteArray
            for (let i = 0; i < constructDirectionCount; i++) {
                let constructImageName = spriteImage.name + "-" + i;

                item.spriteArray[constructImageName] = {
                    name: constructImageName,
                    count: constructImageCount,
                    offset: item.spriteCount
                };
                item.spriteCount += constructImageCount;
            }
        } else {
```

```
            // If the spriteImage has no directions, store just the name and image count in
            spriteArray
            let constructImageName = spriteImage.name;

            item.spriteArray[constructImageName] = {
                name: constructImageName,
                count: constructImageCount,
                offset: item.spriteCount
            };

            item.spriteCount += constructImageCount;
        }
    });
}


// Polyfill for a few browsers that still do not support Object.assign
if (typeof Object.assign !== "function") {
    Object.assign = function(target, varArgs) { // .length of function is 2
        "use strict";
        if (target === null) { // TypeError if undefined or null
            throw new TypeError("Cannot convert undefined or null to object");
        }

        var to = Object(target);

        for (var index = 1; index < arguments.length; index++) {
            var nextSource = arguments[index];

            if (nextSource != null) { // Skip over if undefined or null
                for (var nextKey in nextSource) {
                    // Avoid bugs when hasOwnProperty is shadowed
                    if (Object.prototype.hasOwnProperty.call(nextSource, nextKey)) {
                        to[nextKey] = nextSource[nextKey];
                    }
                }
            }
        }

        return to;
    };
}

// The default add() method used by all our game entities
function addItem(details) {
    var name = details.name;

    // Initialize the item with any default properties the item should have
    var item = Object.assign({}, baseItem);

    // Assign the item all the default properties for its category type
    Object.assign(item, this.defaults);
```

```
    // Assign item properties based on the item name
    Object.assign(item, this.list[name]);

    // By default, set the item's life to its maximum hit points
    item.life = item.hitPoints;

    // Override item defaults based on details
    Object.assign(item, details);

    return item;
}

// Default properties that every item should have
var baseItem = {
    animationIndex: 0,
    direction: 0,

    selected: false,
    selectable: true,

    orders: { type: "stand" },
    action: "stand",
};
```

The `loadItem()` method uses the image loader to load the sprite sheet image into the `spriteSheet` property. It then goes through the `spriteImages` definition and creates a `spriteArray` object that stores the starting offsets for each of the sprite animations.

You will notice that the code checks for the existence of `count` and `directions` properties when creating the array. This allows us to define multidirectional sprites, which we will need for drawing entities like turrets and vehicles.

The `addItem()` method starts with a copy of the `baseItem` object and then, using `Object.assign()`, first applies the defaults for the entity type (for example, buildings), extends it with properties for the specific entity name (for example, base), sets the life for the item, and finally applies any additional properties passed into the `details` parameter. Currently, `baseItem` contains only a few properties like `animationIndex` and `direction`.

This interesting way of creating objects gives us our own implementation of multiple inheritance, allowing us to define and override properties at four different levels: `baseItem` properties, item type properties, item-specific properties, and additional details passed as parameters to the method (such as the item position and team color).

In addition to these two functions, we define a polyfill for older browsers that might not support `Object.assign()`. This code will be ignored in newer browsers that already support `Object.assign()` for cloning an object with all its properties.

Now that we have defined our first entity, we need a simple way of adding entities to a level.

Adding Entities to the Level

The first thing we will do is modify our level definition to include a list of entity types required to be loaded and a list of items to add to the level before it starts. We will modify the first level that we created in `levels.js`, as shown in Listing 7-4.

Listing 7-4. Loading and Adding Entities Inside the Level (levels.js)

```
var levels = {
    "singleplayer": [
        {
            "name": "Introduction",
            "briefing": "In this level you will learn how to pan across the map.\nDon't
            worry! We will be implementing more features soon.",

            /* Map Details */
            "mapName": "plains",

            /* Starting location for player */
            "startX": 4,
            "startY": 4,

            /* Entities to be loaded */
            "requirements": {
                "buildings": ["base"],
                "vehicles": [],
                "aircraft": [],
                "terrain": []
            },

            /* Entities to be added */
            "items": [
                { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
                { "type": "buildings", "name": "base", "x": 12, "y": 16, "team": "green" },
                { "type": "buildings", "name": "base", "x": 15, "y": 15, "team": "green",
                "life": 50 },
            ]
        }
    ],

    "multiplayer": [

    ]
};
```

We have added two new sections to the first single-player level: `requirements` and `items`.

The `requirements` property contains the buildings, vehicles, aircraft, and terrain to preload for this level. For now, we load only buildings of type `base`.

The `items` array contains details of the entities we want to add to the level. The details we provide include the item type and name, the x and y grid coordinates, and the color of the team. These are the bare-minimum properties that we need in order to uniquely define an entity.

We have added three base buildings with random positions and teams. The last building in the `items` array also contains an additional property: `life`. Because of the way we defined the `addItem()` method earlier, this `life` property will override the default value of `life` for the base. This way, we will also have an example of a damaged building.

Next we will modify the `loadLevelData()` method in `game.js` to load and add these entities when the game starts (see Listing 7-5).

Listing 7-5. Loading and Adding Entities Inside loadLevelData()(game.js)

```
loadLevelData: function(level) {
    game.currentLevel = level;
    game.currentMap = maps[level.mapName];

    // Load all the assets for the level starting with the map image
    game.currentMapImage = loader.loadImage("images/maps/" + maps[level.mapName].mapImage);

    // Initialize all the arrays for the game
    game.resetArrays();

    // Load all the assets for every entity defined in the level requirements array
    for (let type in level.requirements) {
        let requirementArray = level.requirements[type];

        requirementArray.forEach(function(name) {
            if (window[type] && typeof window[type].load === "function") {
                window[type].load(name);
            } else {
                console.log("Could not load type :", type);
            }
        });
    }

    // Add all the items defined in the level items array to the game
    level.items.forEach(function(itemDetails) {
        game.add(itemDetails);
    });
},
```

We do three things in the newly added code. We first initialize the game arrays by calling a
game.resetArrays() method.

We then iterate through the requirements object and call the appropriate load() method for each
entity. The load() methods in turn will call the asset loader to asynchronously load all the images for the
entity in the background and enable the entermission button once all the images have been loaded.

Finally, we iterate through the items array and pass the item details to a game.add() method.

Next we will add resetArrays(), add(), and remove() methods to the game object inside game.js
(see Listing 7-6).

Listing 7-6. Adding resetArrays(), add(), and remove() (game.js)

```
resetArrays: function() {
    // Count items added in game, to assign them a unique id
    game.counter = 0;

    // Track all the items currently in the game
    game.items = [];
    game.buildings = [];
    game.vehicles = [];
    game.aircraft = [];
    game.terrain = [];
```

```
    // Track items that have been selected by the player
    game.selectedItems = [];
},

add: function(itemDetails) {
    // Set a unique id for the item
    if (!itemDetails.uid) {
        itemDetails.uid = ++game.counter;
    }

    var item = window[itemDetails.type].add(itemDetails);

    // Add the item to the items array
    game.items.push(item);

    // Add the item to the type-specific array
    game[item.type].push(item);

    return item;
},

remove: function(item) {
    // Unselect item if it is selected
    item.selected = false;
    for (let i = game.selectedItems.length - 1; i >= 0; i--) {
        if (game.selectedItems[i].uid === item.uid) {
            game.selectedItems.splice(i, 1);
            break;
        }
    }

    // Remove item from the items array
    for (let i = game.items.length - 1; i >= 0; i--) {
        if (game.items[i].uid === item.uid) {
            game.items.splice(i, 1);
            break;
        }
    }

    // Remove items from the type-specific array
    for (let i = game[item.type].length - 1; i >= 0; i--) {
        if (game[item.type][i].uid === item.uid) {
            game[item.type].splice(i, 1);
            break;
        }
    }
},
```

The resetArrays() method merely initializes all the game-specific arrays and the game items counter.

The add() method generates a unique identifier (UID) for an item using the counter in case the item doesn't already have one, invokes the appropriate entity's add() method, and finally saves the item in the appropriate game arrays. For the base building, this method would first call buildings.add() and then add the new building to the game.items and game.buildings arrays.

175

The remove() method removes a specified item from the selectedItems, items, and entity-specific arrays. This way, any time an item is removed from the game (for example, when it is destroyed) it is automatically removed from the selection and the items array.

Now that we have set up the code for both defining the entity and adding entities to the level, we are ready to start drawing them on the screen.

Drawing the Entities

To draw the entities, we need to implement the animate() and draw() methods inside the entity object and then call these methods from the game animationLoop() and drawingLoop() methods.

We start by implementing the default animate() and draw() methods for all items inside the baseItem object in common.js. The baseItem object will now look like Listing 7-7.

Listing 7-7. Implementing the Default draw() and animate() Methods (common.js)

```
// Default properties that every item should have
var baseItem = {
    animationIndex: 0,
    direction: 0,

    selected: false,
    selectable: true,

    orders: { type: "stand" },
    action: "stand",

    // Default method for animating an item
    animate: function() {

        // Check the health of the item
        if (this.life > this.hitPoints * 0.4) {
            // Consider item healthy if it has more than 40% life
            this.lifeCode = "healthy";
        } else if (this.life > 0) {
            // Consider item damaged if it has less than 40% life
            this.lifeCode = "damaged";
        } else {
            // Remove item from the game if it has died (life is 0 or negative)
            this.lifeCode = "dead";
            game.remove(this);

            return;
        }

        // Process the current action
        this.processActions();
    },

    // Default method for drawing an item
    draw: function() {
```

```
    // Compute pixel coordinates on canvas for drawing item
    this.drawingX = (this.x * game.gridSize) - game.offsetX - this.pixelOffsetX;
    this.drawingY = (this.y * game.gridSize) - game.offsetY - this.pixelOffsetY;

    this.drawSprite();
},};
```

In the animate() method, we first set the lifeCode property of the item based on its health and hitPoints. If an item's health drops below zero, we set lifeCode to dead and remove it from the game. We then invoke the item's processAction() method.

In the draw() method, we first compute the coordinates on the canvas where we need to draw the item by converting the grid x and y coordinates and save them in the drawingX and drawingY properties. We then invoke the item's drawSprite() method.

Next we will implement processAction() and drawSprite() for buildings inside the defaults section of the buildings object, as shown in Listing 7-8.

Listing 7-8. Implementing processAction() and drawSprite() Methods (building.js)

```
defaults: {
    type: "buildings",

    processActions: function() {
        switch (this.action) {
            case "stand":
                this.imageList = this.spriteArray[this.lifeCode];
                this.imageOffset = this.imageList.offset + this.animationIndex;
                this.animationIndex++;

                if (this.animationIndex >= this.imageList.count) {
                    this.animationIndex = 0;
                }

                break;

            case "construct":
                this.imageList = this.spriteArray["constructing"];
                this.imageOffset = this.imageList.offset + this.animationIndex;
                this.animationIndex++;

                // Once constructing is complete go back to standing
                if (this.animationIndex >= this.imageList.count) {
                    this.animationIndex = 0;
                    this.action = "stand";
                }

                break;
        }
    },

    // Default function for drawing a building
    drawSprite: function() {
        let x = this.drawingX;
        let y = this.drawingY;
```

```
        // All sprite sheets will have blue in the first row and green in the second row
        let colorIndex = (this.team === "blue") ? 0 : 1;
        let colorOffset = colorIndex * this.pixelHeight;

        // Draw the sprite at x, y
        game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
        pixelWidth, colorOffset, this.pixelWidth, this.pixelHeight, x, y, this.pixelWidth,
        this.pixelHeight);
    }
},
```

The processAction() method updates the imageList and animationIndex based on the item's action property. For now, we implement only the stand and construct actions.

For the stand action, we choose either the healthy or damaged sprite animation and increment the animationIndex property. In case the animationIndex exceeds the number of frames in the sprite, we roll the value back to zero. This way, the animation rotates through every frame in the sprite in a repeated loop.

For the construct action, we display the constructing sprites and roll over into the stand action once it has completed.

The drawSprite() method is relatively simpler. We first calculate the image offset for the image color row (based on team). We then use the foreground context's drawImage() method to draw the appropriate part of the sprite sheet image on the foreground canvas at the drawing coordinates we computed earlier.

Now that the draw() and animate() methods are in place, we need to call them from the game object. We will modify the game.animationLoop() and game.drawingLoop() methods inside game.js, as shown in Listing 7-9.

Listing 7-9. Calling draw() and animate() from the Game Loops (game.js)

```
animationLoop: function() {
    // Animate each of the elements within the game
    game.items.forEach(function(item) {
        item.animate();
    });

    // Sort game items into a sortedItems array based on their x,y coordinates
game.sortedItems = Object.assign([], game.items);
    game.sortedItems.sort(function(a, b) {
        return a.y - b.y + ((a.y === b.y) ? (b.x - a.x) : 0);
    });
},

// The map is broken into square tiles of this size (20 pixels x 20 pixels)
gridSize: 20,
// X & Y panning offsets for the map
offsetX: 0,
offsetY: 0,

drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();
```

```
    // Draw the background whenever necessary
    game.drawBackground();

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0, 0, game.canvasWidth, game.canvasHeight);

    // Start drawing the foreground elements
    game.sortedItems.forEach(function(item) {
        item.draw();
    });

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

Within the animationLoop() method, we first iterate through all the game items and call their animate() methods. We then create a game.sortedItems array, which contains all the items sorted by y values and then x values after adjusting for pixelOffset. This way, items that have a lower y value are earlier in the array.

The new code inside the drawingLoop() method merely iterates through the sortedItems array and calls the draw() method of each item. We use the sortedItems array so that items are drawn in order from back to front based on their y coordinates. This is a simple implementation of depth sorting, which ensures that items closer to the player obscure items behind them, thus giving the illusion of depth.

With this last change, we are now ready to see our first game entity drawn on the screen. If we open the game in the browser and load the first level, we should see the three base buildings we defined in the map drawn next to each other (see Figure 7-2).
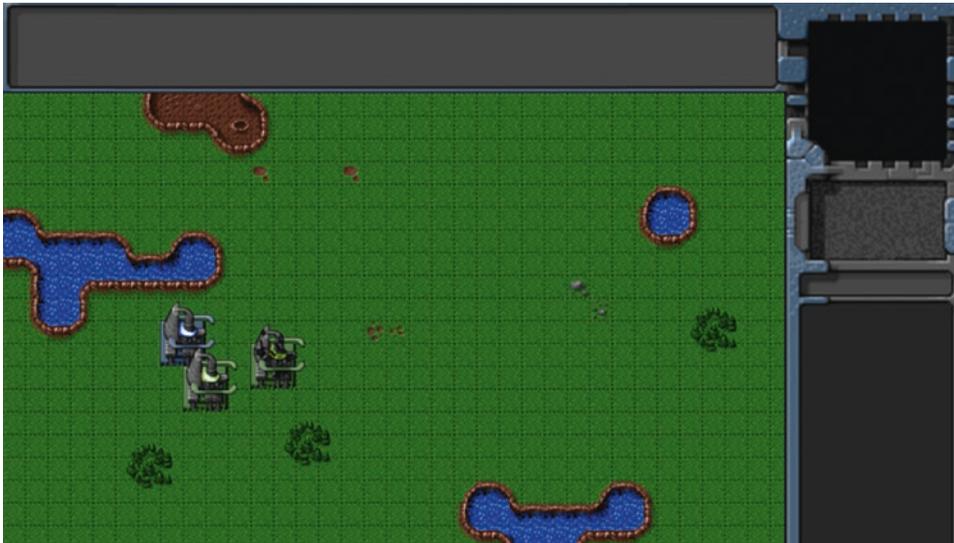
Figure 7-2. *The three base buildings*

179

As you can see, the first blue team base is shown with a flashing blue light using the "healthy" animation.

The second green team base is drawn on top of the first one and partially obscures it. This is a result of our depth-sorting step and lets the player clearly see that the second base is in front of the first one.

Finally, the third base with a lower value of life looks damaged. This is because we automatically use the "damaged" animation whenever the life of the building is less than 40 percent of its maximum hit points. If you recall, we specified this lower life value in the item details inside levels.js.

Now that we have the framework for showing buildings within the game, let's add the remaining buildings, starting with the starport.

Adding the Starport

The starport is used to purchase both land and air units. The starport sprite sheet has a few interesting animations that the base did not have: a teleporting animation sequence that we will use when the building is first created, and an opening and closing animation sequence that we will use when we transport in new units.

The first thing we will do is add the starport definition to the buildings list just below the base definition inside buildings.js (see Listing 7-10).

Listing 7-10. Definition for Starport (buildings.js)

```
"starport": {
    name: "starport",
    pixelWidth: 40,
    pixelHeight: 60,
    baseWidth: 40,
    baseHeight: 55,
    pixelOffsetX: 1,
    pixelOffsetY: 5,
    buildableGrid: [
        [1, 1],
        [1, 1],
        [1, 1]
    ],
    passableGrid: [
        [1, 1],
        [0, 0],
        [0, 0]
    ],
    sight: 3,
    cost: 2000,
    canConstruct: true,
    hitPoints: 300,
    spriteImages: [
        { name: "teleport", count: 9 },
        { name: "closing", count: 18 },
        { name: "healthy", count: 4 },
        { name: "damaged", count: 1 }
    ],
},
```

The starport definition is very similar to the base definition, apart from the additional sprite images, `teleport` and `closing`.

Next, we will need to account for animating the opening, closing, and teleporting animation states. We will do this by modifying the existing cases as well as adding a few more cases inside the `processActions()` method for the buildings inside `buildings.js`, as shown in Listing 7-11.

Listing 7-11. Handling Teleporting, Opening, and Closing

```
case "teleport":
    this.imageList = this.spriteArray["teleport"];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    // Once teleporting is complete, move to stand mode
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "stand";
    }

    break;

case "close":
    this.imageList = this.spriteArray["closing"];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    // Once closing is complete, go back to standing
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "stand";
    }

    break;

case "open":
    this.imageList = this.spriteArray["closing"];
    // Opening is just the closing sprites running backward
    this.imageOffset = this.imageList.offset + this.imageList.count - this.animationIndex;
    this.animationIndex++;

    // Once opening is complete, go back to close
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "close";
    }

    break;
```

Like the `construct` animation state, the `teleport`, `close`, and `open` animation states do not keep repeating once they end. The `teleport` animation rolls over into the `stand` animation state. The `open` animation (which is merely the `close` animation state running backward) rolls over into the `close` animation state, which then rolls over into the `stand` animation state.

This way, we can initialize the starport with a `teleport` or `open` animation state, knowing that it will eventually move back to the `stand` animation state once the current animation completes.

Now, we can add a few starports to the level by modifying the requirements and items inside `levels.js`, as shown in Listing 7-12.

Listing 7-12. Adding Starports to the Level

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport"],
    "vehicles": [],
    "aircraft": [],
    "terrain": []
},

/* Entities to be added */
"items": [
    { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "base", "x": 12, "y": 16, "team": "green" },
    { "type": "buildings", "name": "base", "x": 15, "y": 15, "team": "green", "life": 50 },

    { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 10, "team": "blue", "action":
    "teleport" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 6, "team": "green", "action":
    "open" },
]
```

When we open the game in the browser and start the level, we should see three new starport buildings, as shown in Figure 7-3.
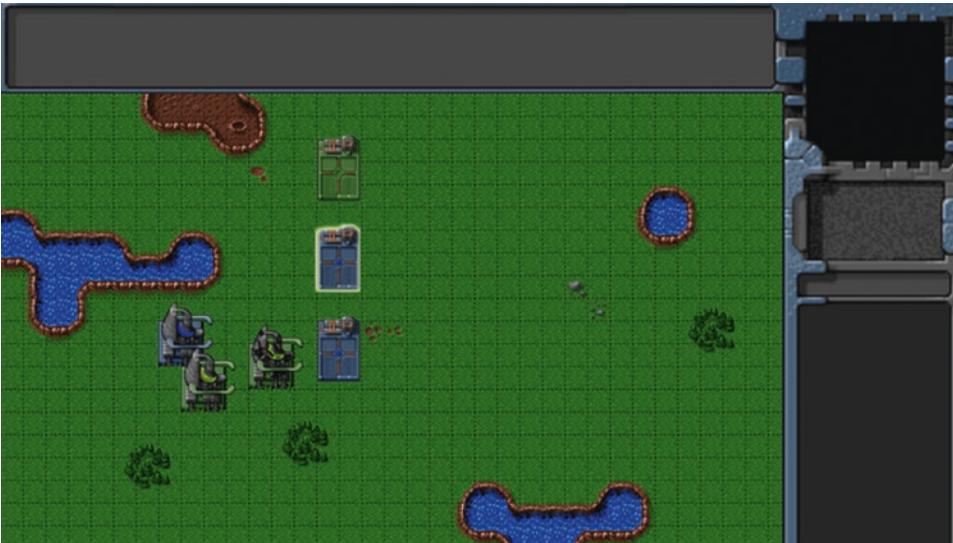
Figure 7-3. *The three starport buildings*

The first green team starport opens and then closes. The second blue team starport first glows and comes into existence and then switches to stand mode, while the last blue team starport merely waits in stand mode.

Now that the starport has been added, the next building we will look at is the harvester.

Adding the Harvester

The harvester is a unique entity in the sense that it is both a building and a vehicle. Unlike the other buildings in the game, the harvester is created by deploying a harvester vehicle at an oil field, where it turns into the building (see Figure 7-4).
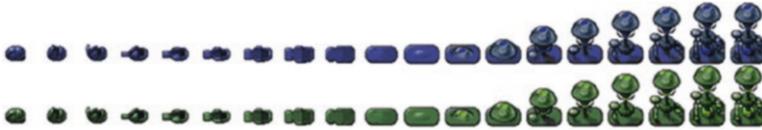
Figure 7-4. *Harvester deploying into building form*

The first thing we will do is add the harvester definition to the buildings list just below the starport definition inside buildings.js (see Listing 7-13).

Listing 7-13. Definition for Harvester Building (buildings.js)

```
"harvester": {
    name: "harvester",
    pixelWidth: 40,
    pixelHeight: 60,
    baseWidth: 40,
    baseHeight: 20,
    pixelOffsetX: -2,
    pixelOffsetY: 40,
    buildableGrid: [
        [1, 1]
    ],
    passableGrid: [
        [1, 1]
    ],
    sight: 3,
    cost: 5000,
    hitPoints: 300,
    spriteImages: [
        { name: "deploy", count: 17 },
        { name: "healthy", count: 3 },
        { name: "damaged", count: 1 }
    ],
},
```

Next, we will need to account for the deploying animation state. We will do this by adding the deploy case to the processActions() method inside buildings.js, as shown in Listing 7-14.

Listing 7-14. Handling the deploy Animation State (buildings.js)

```
case "deploy":
    this.imageList = this.spriteArray["deploy"];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    // Once deploying is complete, go to stand
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "stand";
    }

    break;
```

The deploy state, like the teleport state we defined earlier, automatically rolls into the stand animation state once it completes.

Now, we can add the harvester to the level by modifying the requirements and items inside levels.js, as shown in Listing 7-15.

Listing 7-15. Adding the Harvester to the Level

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport", "harvester"],
    "vehicles": [],
    "aircraft": [],
    "terrain": []
},

/* Entities to be added */
"items": [
    { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "base", "x": 12, "y": 16, "team": "green" },
    { "type": "buildings", "name": "base", "x": 15, "y": 15, "team": "green", "life": 50 },

    { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 10, "team": "blue",
    "action": "teleport" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 6, "team": "green",
    "action": "open" },

    { "type": "buildings", "name": "harvester", "x": 20, "y": 10, "team": "blue" },
    { "type": "buildings", "name": "harvester", "x": 22, "y": 12, "team": "green",
    "action": "deploy" },
]
```

When we open the game in the browser and start the level, we should see two new harvester buildings, as shown in Figure 7-5.

Figure 7-5. *The two harvester buildings*

The blue harvester is in the default stand mode, while the green harvester, which is in deploy mode, transforms into a building and then switches to stand mode.

Now that the harvester has been added, the last building we will look at is the ground turret.

Adding the Ground Turret

The ground turret is a defensive structure that attacks only ground-based threats.

It is the only building that uses direction-based sprites so, unlike with the other buildings, we need to take the turret's direction into account during animation and drawing.

The direction property can take values ranging from 0 to 7 increasing in the clockwise direction, with 0 pointing toward the north and 7 pointing in the northwest direction, as shown in Figure 7-6.

Figure 7-6. *Direction sprites for the ground turret ranging from 0 to 7*

The first thing we will do is add the ground turret definition to the buildings list just below the harvester definition inside buildings.js (see Listing 7-16).

Listing 7-16. Definition for Ground Turret (buildings.js)

```
"ground-turret": {
    name: "ground-turret",
    canAttack: true,
    canAttackLand: true,
    canAttackAir: false,
    weaponType: "cannon-ball",
    action: "stand",
    direction: 0, // Face upward (0) by default
    directions: 8, // Total of 8 turret directions allowed (0-7)
    orders: { type: "guard" },
    pixelWidth: 38,
    pixelHeight: 32,
    baseWidth: 20,
    baseHeight: 18,
    cost: 1500,
    canConstruct: true,
    pixelOffsetX: 9,
    pixelOffsetY: 12,
    buildableGrid: [
        [1]
    ],
    passableGrid: [
        [1]
    ],
    sight: 5,
    hitPoints: 200,
    spriteImages: [
        { name: "teleport", count: 9 },
        { name: "healthy", count: 1, directions: 8 },
        { name: "damaged", count: 1 }
    ],
}
```

The gun turret has a few additional properties that indicate whether it can be used to attack the enemy, the direction the turret is pointing, and the type of weapon it uses. We will use these properties later when we implement combat in our game.

The `healthy` sprites have an additional `directions` property that is used by the `itemLoad()` method to generate sprites for each direction.

Next, we will modify the `stand` case in the `processActions()` method to handle directions, as shown in Listing 7-17.

Listing 7-17. Handling Directions in the stand Animation State (buildings.js)

```
case "stand":
    if (this.name === "ground-turret" && this.lifeCode === "healthy") {
        // For a healthy turret, use direction to choose image list
        let direction = Math.round(this.direction) % this.directions;

        this.imageList = this.spriteArray[this.lifeCode + "-" + direction];
    } else {
```

```
    // In all other cases, use lifeCode
    this.imageList = this.spriteArray[this.lifeCode];
}

this.imageOffset = this.imageList.offset + this.animationIndex;
this.animationIndex++;

if (this.animationIndex >= this.imageList.count) {
    this.animationIndex = 0;
}

break;
```

We modify the code to use the turret direction to pick the appropriate image list if it is healthy. We round the value of direction and ensure that it lies between 0 and 7 so that we can use fractional direction values if we need, but the sprite will still be picked correctly. If the turret is damaged, it does not have directions, so we fall back to the default behavior of using lifeCode to pick the image list.

Now, we can add the turret to the level by modifying the requirements and items inside levels.js, as shown in Listing 7-18.

Listing 7-18. Adding the Ground Turret to the Level

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport", "harvester", "ground-turret"],
    "vehicles": [],
    "aircraft": [],
    "terrain": []
},

/* Entities to be added */
"items": [
    { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "base", "x": 12, "y": 16, "team": "green" },
    { "type": "buildings", "name": "base", "x": 15, "y": 15, "team": "green", "life": 50 },

    { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 10, "team": "blue",
    "action": "teleport" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 6, "team": "green",
    "action": "open" },

    { "type": "buildings", "name": "harvester", "x": 20, "y": 10, "team": "blue" },
    { "type": "buildings", "name": "harvester", "x": 22, "y": 12, "team": "green",
    "action": "deploy" },

    { "type": "buildings", "name": "ground-turret", "x": 14, "y": 9, "team":
    "blue", "direction": 3 },
    { "type": "buildings", "name": "ground-turret", "x": 14, "y": 12, "team":
    "green", "direction": 1 },
    { "type": "buildings", "name": "ground-turret", "x": 16, "y": 10, "team":
    "blue", "action": "teleport" },

]
```

We specify a starting `direction` property for the first two turrets and set the `action` property to `teleport` for the third. When we open the game in the browser and start the level, we should see three new turrets, as shown in Figure 7-7.
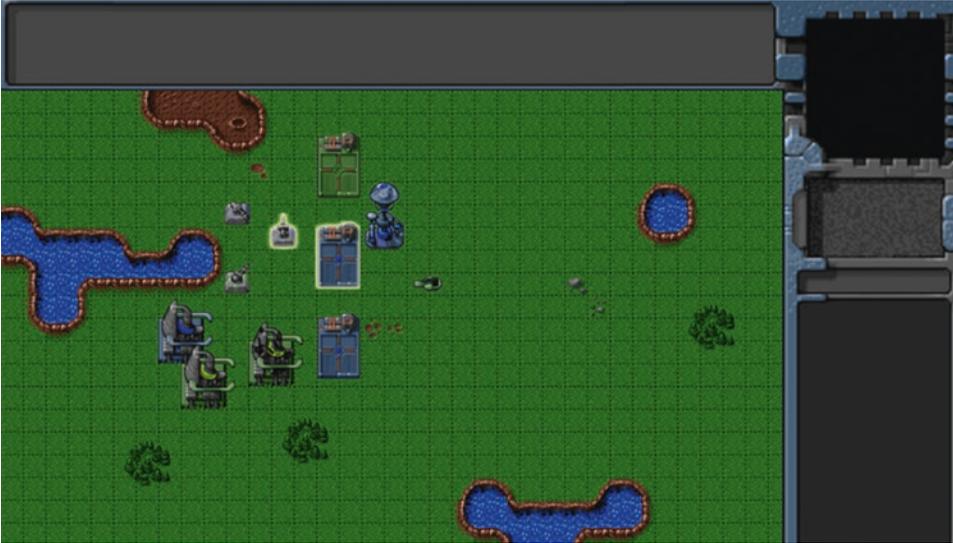
Figure 7-7. *The three ground turret buildings*

The first two turrets are in `stand` mode and face two different directions, while the third one teleports in facing the default direction and switches to `stand` mode after teleporting in.

At this point, we have implemented all the buildings that we need. Now it's time to start adding a few vehicles to our game.

Adding the Vehicles

All the vehicles in our game, including the transport, will have a simple sprite sheet with the vehicle pointing in eight directions similar to the ground turret, as shown in Figure 7-8.
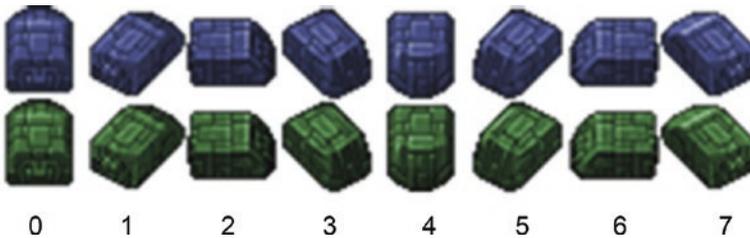
Figure 7-8. *The transport sprite sheet*

We will set up the code for our vehicles by defining a new vehicles object inside vehicles.js, as shown in Listing 7-19.

Listing 7-19. Defining the vehicles Object (vehicles.js)

```
var vehicles = {
    list: {
        "transport": {
            name: "transport",
            pixelWidth: 31,
            pixelHeight: 30,
            pixelOffsetX: 15,
            pixelOffsetY: 15,
            radius: 15,
            speed: 15,
            sight: 3,
            cost: 400,
            hitPoints: 100,
            turnSpeed: 3,
            spriteImages: [
                { name: "stand", count: 1, directions: 8 }
            ],
        },
        "harvester": {
            name: "harvester",
            pixelWidth: 21,
            pixelHeight: 20,
            pixelOffsetX: 10,
            pixelOffsetY: 10,
            radius: 10,
            speed: 10,
            sight: 3,
            cost: 1600,
            canConstruct: true,
            hitPoints: 50,
            turnSpeed: 3,
            spriteImages: [
                { name: "stand", count: 1, directions: 8 }
            ],
        },
        "scout-tank": {
            name: "scout-tank",
            canAttack: true,
            canAttackLand: true,
            canAttackAir: false,
            weaponType: "bullet",
            pixelWidth: 21,
            pixelHeight: 21,
            pixelOffsetX: 10,
            pixelOffsetY: 10,
            radius: 11,
            speed: 20,
```

```
            sight: 4,
            cost: 500,
            canConstruct: true,
            hitPoints: 50,
            turnSpeed: 5,
            spriteImages: [
                { name: "stand", count: 1, directions: 8 }
            ],
        },
        "heavy-tank": {
            name: "heavy-tank",
            canAttack: true,
            canAttackLand: true,
            canAttackAir: false,
            weaponType: "cannon-ball",
            pixelWidth: 30,
            pixelHeight: 30,
            pixelOffsetX: 15,
            pixelOffsetY: 15,
            radius: 13,
            speed: 15,
            sight: 5,
            cost: 1200,
            canConstruct: true,
            hitPoints: 50,
            turnSpeed: 4,
            spriteImages: [
                { name: "stand", count: 1, directions: 8 }
            ],
        }
    },

    defaults: {
        type: "vehicles",
        directions: 8,
        canMove: true,

        processActions: function() {
            let direction = Math.round(this.direction) % this.directions;

            switch (this.action) {
                case "stand":

                    this.imageList = this.spriteArray["stand-" + direction];
                    this.imageOffset = this.imageList.offset + this.animationIndex;
                    this.animationIndex++;

                    if (this.animationIndex >= this.imageList.count) {
                        this.animationIndex = 0;
                    }
```

```
                    break;
                }
            },

            // Default function for drawing a vehicle
            drawSprite: function() {
                let x = this.drawingX;
                let y = this.drawingY;

                let colorIndex = (this.team === "blue") ? 0 : 1;
                let colorOffset = colorIndex * this.pixelHeight;

                game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
                pixelWidth, colorOffset, this.pixelWidth, this.pixelHeight, x, y, this.
                pixelWidth, this.pixelHeight);
            },
        },

    load: loadItem,
    add: addItem,
};
```

The structure of our `vehicles` object is very similar to the `buildings` object. We have a `list` property where we define the four vehicle types: the transport, the harvester, the scout tank, and the heavy tank.

All of the vehicle sprites also have a `directions` property. The default `stand` animation implementation inside `processActions()` uses the vehicle's direction to select the sprite to draw, just like we did for the ground turret. We use `animationIndex` to handle multiple images within a sprite so that we can add vehicles with animation if needed.

The vehicles also have properties such as `speed`, `sight`, and `cost`. The transport and harvester do not have any weapons, while the two tanks have weapon-based properties (`canAttack`, `canAttackLand`, `weaponType`) similar to the ground turret building we defined earlier. We will use all of these properties in later chapters to implement movement and combat.

Now, we can add these vehicles to the levels by first modifying the `requirements` property inside `levels.js`, as shown in Listing 7-20.

Listing 7-20. Adding the Vehicles to the Level Requirements (levels.js)

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport", "harvester", "ground-turret"],
    "vehicles": ["transport", "harvester", "scout-tank", "heavy-tank"],
    "aircraft": [],
    "terrain": []
},
```

Next, we will add a few new vehicles to the items section of the level as shown in Listing 7-21.

Listing 7-21. Adding the Vehicles to the Level Items (levels.js)

```
{ "type": "vehicles", "name": "transport", "x": 26, "y": 10, "team": "blue", "direction": 2 },
{ "type": "vehicles", "name": "harvester", "x": 26, "y": 12, "team": "blue", "direction": 3 },
{ "type": "vehicles", "name": "scout-tank", "x": 26, "y": 14, "team": "blue", "direction": 4 },
{ "type": "vehicles", "name": "heavy-tank", "x": 26, "y": 16, "team": "blue", "direction": 5 },
```

```
{ "type": "vehicles", "name": "transport", "x": 28, "y": 10, "team": "green", "direction": 7 },
{ "type": "vehicles", "name": "harvester", "x": 28, "y": 12, "team": "green", "direction": 6 },
{ "type": "vehicles", "name": "scout-tank", "x": 28, "y": 14, "team": "green", "direction": 1 },
{ "type": "vehicles", "name": "heavy-tank", "x": 28, "y": 16, "team": "green", "direction": 0 },
```

When we open the game in the browser and start the level, we should see the vehicles, as shown in Figure 7-9.
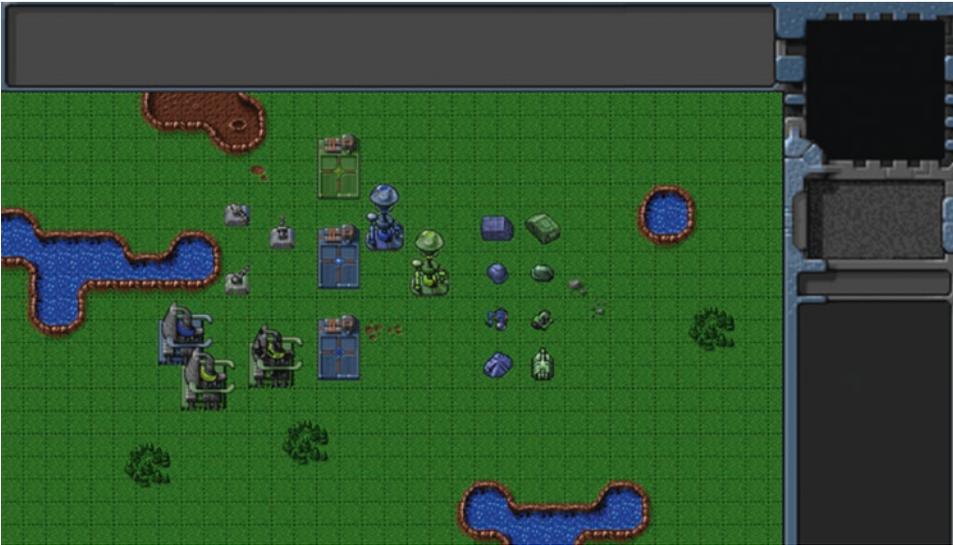
Figure 7-9. *Adding vehicles to the level*

The vehicles point in different directions based on the properties we set when adding them to the `items` list. With the vehicles implemented, it is time to add the aircraft to our game.

Adding the Aircraft

The aircraft in our game have a sprite sheet similar to vehicles except for one difference: shadows. An aircraft sprite sheet has a third row with shadows in it. Also, the chopper sprite sheet has multiple images for each direction, as shown in Figure 7-10.
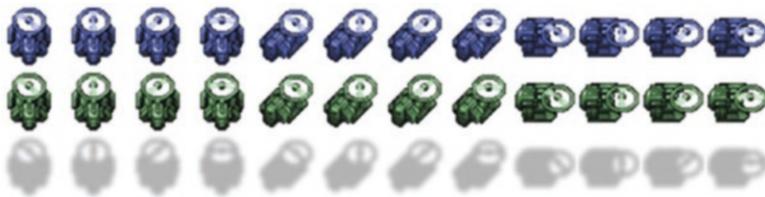
Figure 7-10. *The chopper sprite sheet with shadows*

We will set up the code for our aircraft by defining a new `aircraft` object inside `aircraft.js`, as shown in Listing 7-22.

Listing 7-22. Defining the aircraft Object (aircraft.js)

```
var aircraft = {
    list: {
        "chopper": {
            name: "chopper",
            cost: 900,
            canConstruct: true,
            pixelWidth: 40,
            pixelHeight: 40,
            pixelOffsetX: 20,
            pixelOffsetY: 20,
            weaponType: "heatseeker",
            radius: 18,
            sight: 6,
            canAttack: true,
            canAttackLand: true,
            canAttackAir: true,
            hitPoints: 50,
            speed: 25,
            turnSpeed: 4,
            pixelShadowHeight: 40,
            spriteImages: [
                { name: "stand", count: 4, directions: 8 }
            ],
        },
        "wraith": {
            name: "wraith",
            cost: 600,
            canConstruct: true,
            pixelWidth: 30,
            pixelHeight: 30,
            canAttack: true,
            canAttackLand: false,
            canAttackAir: true,
            weaponType: "fireball",
            pixelOffsetX: 15,
            pixelOffsetY: 15,
            radius: 15,
            sight: 8,
            speed: 40,
            turnSpeed: 4,
            hitPoints: 50,
            pixelShadowHeight: 40,
            spriteImages: [
                { name: "stand", count: 1, directions: 8 }
            ],
        }
    },
```

```
    defaults: {
        type: "aircraft",
        directions: 8,
        canMove: true,

        processActions: function() {
            let direction = Math.round(this.direction) % this.directions;

            switch (this.action) {
                case "stand":

                    this.imageList = this.spriteArray["stand-" + direction];
                    this.imageOffset = this.imageList.offset + this.animationIndex;
                    this.animationIndex++;

                    if (this.animationIndex >= this.imageList.count) {
                        this.animationIndex = 0;
                    }

                    break;

            }
        },

        drawSprite: function() {
            let x = this.drawingX;
            let y = this.drawingY;

            let colorIndex = (this.team === "blue") ? 0 : 1;
            let colorOffset = colorIndex * this.pixelHeight;
            // The aircraft shadow is on the third row of the sprite sheet
            let shadowOffset = this.pixelHeight * 2;

            // Draw the aircraft pixelShadowHeight pixels above its position
            game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
            pixelWidth, colorOffset, this.pixelWidth, this.pixelHeight, x, y - this.
            pixelShadowHeight, this.pixelWidth, this.pixelHeight);

            // Draw the shadow at aircraft position
            game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
            pixelWidth, shadowOffset, this.pixelWidth, this.pixelHeight, x, y, this.
            pixelWidth, this.pixelHeight);

        },
    },

    load: loadItem,
    add: addItem,
};
```

The structure of our `aircraft` object is similar to the `vehicles` object. We have a `list` property where we define the two aircraft types: the chopper and the wraith.

All of the aircraft sprites have the `directions` property. The default `stand` animation implementation inside `processActions()` uses the aircraft's direction to select the sprite to draw. In the case of the chopper, we also use `animationIndex` to handle multiple images for each direction.

The one big difference is in the way the `drawSprite()` method is implemented. We draw a shadow at the location of the aircraft and draw the actual aircraft `pixelShadowHeight` pixels above the location of the aircraft. This way, the aircraft looks like it is floating above the ground and the shadow is on the ground below it.

Now, we can add these aircraft to the map by first modifying the `requirements` inside `levels.js`, as shown in Listing 7-23.

Listing 7-23. Adding the Aircraft to the Level Requirements (levels.js)

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport", "harvester", "ground-turret"],
    "vehicles": ["transport", "harvester", "scout-tank", "heavy-tank"],
    "aircraft": ["chopper", "wraith"],
    "terrain": []
},
```

Next, we will add a few aircraft to the items section of the level as shown in Listing 7-24.

Listing 7-24. Adding the Aircraft to the Level Items (levels.js)

```
{ "type": "aircraft", "name": "chopper", "x": 20, "y": 22, "team": "blue", "direction": 2 },
{ "type": "aircraft", "name": "wraith", "x": 23, "y": 22, "team": "green", "direction": 3 },
```

When we open the game in the browser and start the level, we should see the aircraft hovering above the ground, as shown in Figure 7-11.


Figure 7-11. *The aircraft floating above the ground*

The shadows help create the illusion that the aircraft are floating above the ground while also serving to mark their exact position on the ground. The chopper blades and their shadow on the ground seem to rotate because of the animation that we use to draw the chopper.

With the aircraft implemented, we will now add the terrain to our game.

Adding the Terrain

With the exception of the oil field, the terrain entities in our game are static bodies intended only for cosmetic use. The oil field is a special entity above which the harvester vehicle can deploy into the harvester building. The oil field sprite sheet includes two versions of the oil field: a default version and a "hint" version that shows a blurry harvester placed above the oil field as a hint for the player.

We will set up the code for our terrain by defining a new terrain object inside terrain.js, as shown in Listing 7-25.

Listing 7-25. Defining the terrain Object (terrain.js)

```
var terrain = {
    list: {
        "oilfield": {
            name: "oilfield",
            pixelWidth: 40,
            pixelHeight: 60,
            baseWidth: 40,
            baseHeight: 20,
            pixelOffsetX: 0,
            pixelOffsetY: 40,
            buildableGrid: [
                [1, 1]
            ],
            passableGrid: [
                [0, 0]
            ],
            spriteImages: [
                { name: "hint", count: 1 },
                { name: "stand", count: 1 }
            ],
        },
        "bigrocks": {
            name: "bigrocks",
            pixelWidth: 40,
            pixelHeight: 70,
            baseWidth: 40,
            baseHeight: 40,
            pixelOffsetX: 0,
            pixelOffsetY: 30,
            buildableGrid: [
                [1, 1],
                [0, 1]
            ],
```

```
            passableGrid: [
                [1, 1],
                [0, 1]
            ],
            spriteImages: [
                { name: "stand", count: 1 }
            ],
        },
        "smallrocks": {
            name: "smallrocks",
            pixelWidth: 20,
            pixelHeight: 35,
            baseWidth: 20,
            baseHeight: 20,
            pixelOffsetX: 0,
            pixelOffsetY: 15,
            buildableGrid: [
                [1]
            ],
            passableGrid: [
                [1]
            ],
            spriteImages: [
                { name: "stand", count: 1 }
            ],
        },
    },

    defaults: {
        type: "terrain",
        selectable: false,

        animate: function() {
            // No need to do a health check for terrain. Just call processActions
            this.processActions();
        },

        processActions: function() {
            // Since there is no animation or special handling, just set imageList based on
            action
            this.imageList = this.spriteArray[this.action];
            this.imageOffset = this.imageList.offset;
        },

        drawSprite: function() {
            let x = this.drawingX;
            let y = this.drawingY;

            var colorOffset = 0; // No team based colors for terrain
```

```
        game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
        pixelWidth, colorOffset, this.pixelWidth, this.pixelHeight, x, y, this.
        pixelWidth, this.pixelHeight);
    }
},

load: loadItem,
add: addItem,
};
```

The structure of our `terrain` object is similar to the `buildings` object. We have a `list` property where we define the terrain types: the oil field, the big rocks, and the small rocks. However, we override the `animate()` method to remove the health check that is typical for other items. We implement a simpler `processActions()` method since our terrain does not use animations or directions. We also implement a simpler `drawSprite()` method that does not use team-based colors.

Now, we can add the terrain to the level by first modifying the requirements inside `levels.js`, as shown in Listing 7-26.

Listing 7-26. Adding the Terrain to the Level Requirements (levels.js)

```
/* Entities to be loaded */
"requirements": {
    "buildings": ["base", "starport", "harvester", "ground-turret"],
    "vehicles": ["transport", "harvester", "scout-tank", "heavy-tank"],
    "aircraft": ["chopper", "wraith"],
    "terrain": ["oilfield", "bigrocks", "smallrocks"]
},
```

Next, we will add some terrain to the items section of the level as shown in Listing 7-27.

Listing 7-27. Adding the Terrain to the Level Items (levels.js)

```
{ "type": "terrain", "name": "oilfield", "x": 5, "y": 7 },
{ "type": "terrain", "name": "oilfield", "x": 8, "y": 7, "action": "hint" },

{ "type": "terrain", "name": "bigrocks", "x": 5, "y": 3 },
{ "type": "terrain", "name": "smallrocks", "x": 8, "y": 3 },
```

We add two oil fields, one of which has the `action` property set to `hint`. When we open the game in the browser and start the level, we should see the rocks and the oil fields, as shown in Figure 7-12.
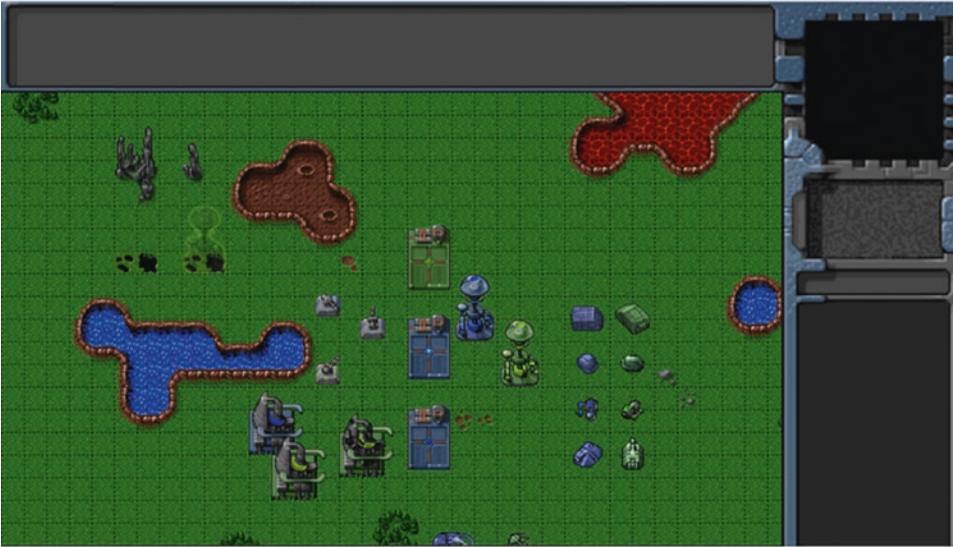
Figure 7-12. *Adding the rocks and the oil fields*

The oil field on the right with the hint has a subtle glowing image of a harvester to let the player know that a harvester can be deployed there. This hint version of the oil field can be used in the earlier levels of the game to introduce the player to the idea of harvesting from oil fields.

With this, we have implemented all the important entities in the game. Of course, at this point all we can do is scroll around the map and look at these entities. The next thing we will work on is adding the ability to interact with them by selecting them.

Selecting Game Entities

We will allow players to select entities either by clicking them or by dragging a selection box across them.

We will start by enabling click selection, by adding two new methods to the mouse object inside mouse.js, as shown in Listing 7-28.

Listing 7-28. Enabling Selection by Clicking (mouse.js)

```
// Called whenever player completes a left-click on the game canvas
leftClick: function(shiftPressed) {
    let clickedItem = mouse.itemUnderMouse();

    if (clickedItem) {
        // Pressing Shift adds to existing selection. If Shift is not pressed, clear
        existing selection
        if (!shiftPressed) {
            game.clearSelection();
        }

        game.selectItem(clickedItem, shiftPressed);
    }
},
```

```
// Return the first item detected under the mouse
itemUnderMouse: function() {
    for (let i = game.items.length - 1; i >= 0; i--) {
        let item = game.items[i];

        // Dead items will not be detected
        if (item.lifeCode === "dead") {
            continue;
        }

        let x = item.x * game.gridSize;
        let y = item.y * game.gridSize;

        if (item.type === "buildings" || item.type === "terrain") {
            // If mouse coordinates are within rectangular area of building or terrain
            if (x <= mouse.gameX && x >= (mouse.gameX - item.baseWidth) && y <= mouse.gameY
            && y >= (mouse.gameY - item.baseHeight)) {
                return item;
            }
        } else if (item.type === "aircraft") {
            // If mouse coordinates are within radius of aircraft (adjusted for
            pixelShadowHeight)
            if (Math.pow(x - mouse.gameX, 2) + Math.pow(y - mouse.gameY - item.
            pixelShadowHeight, 2) < Math.pow(item.radius, 2)) {
                return item;
            }
        } else if (item.type === "vehicles") {
            // If mouse coordinates are within radius of item
            if (Math.pow(x - mouse.gameX, 2) + Math.pow(y - mouse.gameY, 2) < Math.pow
            (item.radius, 2)) {
                return item;
            }
        }
    }
},
```

The mouse.leftClick() method first checks whether there is an item under the mouse during the click using the itemUnderMouse() method. If an item is under the mouse, we call the game.selectItem() method. The game.clearSelection() method is called before selecting the new item unless the Shift key is pressed during the click. This way, users can select multiple items by holding down the Shift key while selecting, or clear the previous selection and select a new item by clicking it without the Shift key.

The itemUnderMouse() method iterates through all the items in the list and returns the first item that is under the mouse gameX and gameY coordinates using different criteria for different item types:

- In the case of buildings and terrain, we check whether the base of the item is under the mouse. This way, the player can click the base of a building to select it but won't have problems selecting vehicles behind the building.

- In the case of vehicles, we check whether the mouse is within a radius from the vehicle center.

- In the case of aircraft, we check whether the mouse is within a radius from the aircraft center and not the shadow by using the pixelShadowHeight property to adjust the y coordinate.

Next, we will track when the left mouse button has been pressed by creating a mousedown event handler (see Listing 7-29).

Listing 7-29. Implementing the mousedown Event Handler (mouse.js)

```
// Is the left mouse button currently pressed
buttonPressed: false,

mousedownhandler: function(ev) {
    mouse.insideCanvas = true;
    mouse.setCoordinates(ev.clientX, ev.clientY);

    if (ev.button === 0) { // Left mouse button was pressed
        mouse.buttonPressed = true;

        mouse.dragX = mouse.gameX;
        mouse.dragY = mouse.gameY;
    }
},
```

Inside the mousedownhandler() method, we set the insideCanvas flag to true and update the coordinates just as we do in the mousemovehandler() method. We then check whether the left button was pressed using the event's button property, and if so, we update the buttonPressed property and save the current mouse coordinates for later use.

Next, we will update the mousemovehandler() method to track when dragging has started (see Listing 7-30).

Listing 7-30. Track Drag Selection in the mousemove Event Handler (mouse.js)

```
mousemovehandler: function(ev) {
    mouse.insideCanvas = true;

    mouse.setCoordinates(ev.clientX, ev.clientY);
    mouse.checkIfDragging();
},

// Is the player dragging and selecting with the left mouse button pressed
dragSelect: false,

// If the mouse is dragged more than this, assume the player is trying to select something
dragSelectThreshold: 5,

checkIfDragging: function() {
    if (mouse.buttonPressed) {
        // If the mouse has been dragged more than threshold, treat it as a drag
        if ((Math.abs(mouse.dragX - mouse.gameX) > mouse.dragSelectThreshold && Math.
        abs(mouse.dragY - mouse.gameY) > mouse.dragSelectThreshold)) {
            mouse.dragSelect = true;
        }
    } else {
        mouse.dragSelect = false;
    }
},
```

Inside the `mousemovehandler()` method we add an additional call to the `checkIfDragging()` method. The `checkIfDragging()` method sets the `dragSelect` property to `true` if the mouse has been moved by more than `dragSelectThreshold` pixels from the point where the mouse button was first pressed, along both x and y coordinates. This little threshold check ensures that every click and mouse interaction is not treated as a drag selection attempt. If the mouse button is not pressed, `dragSelect` is set to `false`.

Next, we will track when the left mouse button has been released by creating a `mouseup` event handler (see Listing 7-31).

Listing 7-31. Implementing the mouseup Event Handler (mouse.js)

```
mouseuphandler: function(ev) {
    mouse.setCoordinates(ev.clientX, ev.clientY);

    let shiftPressed = ev.shiftKey;

    if (ev.button === 0) { // Left mouse button was released
        if (mouse.dragSelect) {
            // If currently drag-selecting, attempt to select items with the selection
            rectangle
            mouse.finishDragSelection(shiftPressed);
        } else {
            // If not dragging, treat this as a normal click once the mouse is released
            mouse.leftClick(shiftPressed);
        }

        mouse.buttonPressed = false;
    }
},

finishDragSelection: function(shiftPressed) {
    if (!shiftPressed) {
        // If shift key is not pressed, clear any previously selected items
        game.clearSelection();
    }

    // Calculate the bounds of the selection rectangle
    let x1 = Math.min(mouse.gameX, mouse.dragX);
    let y1 = Math.min(mouse.gameY, mouse.dragY);
    let x2 = Math.max(mouse.gameX, mouse.dragX);
    let y2 = Math.max(mouse.gameY, mouse.dragY);

    game.items.forEach(function(item) {
        // Unselectable items, dead items, opponent team items, and buildings are not
        drag-selectable
        if (!item.selectable || item.lifeCode === "dead" || item.team !== game.team || item.
        type === "buildings") {
            return;
        }
```

```
        let x = item.x * game.gridSize;
        let y = item.y * game.gridSize;

        if (x1 <= x && x2 >= x) {
            if ((item.type === "vehicles" && y1 <= y && y2 >= y)
                // In case of aircraft, adjust for pixelShadowHeight
                || (item.type === "aircraft" && (y1 <= y - item.pixelShadowHeight) &&
                (y2 >= y - item.pixelShadowHeight))) {

                game.selectItem(item, shiftPressed);
            }
        }
    });

    mouse.dragSelect = false;
},
```

If the left mouse button is released, we check whether the mouse was being dragged. If there was no dragging happening, we treat the event as a normal left-click and call the leftClick() method that we defined earlier.

If, however, the mouse was being dragged before the button was released, we call the finishDragSelection() method where we iterate through every game item and check whether it lies within the bounds of the dragged rectangle. We then call game.selectItem() to select any items that are within the rectangle.

Most importantly, we only allow drag selection for our own vehicles and aircraft and not for enemy entities or our own buildings. This is because drag selection is typically used to select groups of units to move them or attack with them quickly, and selecting enemy units or our own buildings does not really help the player.

We will also need to modify the mouse.init() method to listen for the mousedown and mouseup events and call the appropriate event handlers, as shown in Listing 7-32.

Listing 7-32. Adding Event Listeners in the init() Method (mouse.js)

```
init: function() {
    // Listen for mouse events on the game foreground canvas
    let canvas = document.getElementById("gameforegroundcanvas");

    canvas.addEventListener("mousemove", mouse.mousemovehandler, false);

    canvas.addEventListener("mouseenter", mouse.mouseenterhandler, false);
    canvas.addEventListener("mouseout", mouse.mouseouthandler, false);

    canvas.addEventListener("mousedown", mouse.mousedownhandler, false);
    canvas.addEventListener("mouseup", mouse.mouseuphandler, false);

    mouse.canvas = canvas;
},
```

Next, we will implement a mouse.draw() method that will draw a rectangle to mark the selection area when we are drag-selecting, as shown in Listing 7-33.

Listing 7-33. Drawing the Drag Selection Area (mouse.js)

```
draw: function() {
    // If the player is dragging and selecting, draw a white box to mark the selection area
    if (this.dragSelect) {
        let x = Math.min(this.gameX, this.dragX);
        let y = Math.min(this.gameY, this.dragY);

        let width = Math.abs(this.gameX - this.dragX);
        let height = Math.abs(this.gameY - this.dragY);

        game.foregroundContext.strokeStyle = "white";
        game.foregroundContext.strokeRect(x - game.offsetX, y - game.offsetY, width,
        height);
    }
},
```

We will also need to modify the `game.animationLoop()` method to call `mouse.draw()` as shown in Listing 7-34.

Listing 7-34. Calling mouse.draw() from game.animationLoop (game.js)

```
drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();

    // Draw the background whenever necessary
    game.drawBackground();

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0, 0, game.canvasWidth, game.canvasHeight);

    // Start drawing the foreground elements
    game.sortedItems.forEach(function(item) {
        item.draw();
    });

    // Draw the mouse
    mouse.draw();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

Finally, we will add two selection-related methods to the game object, as shown in Listing 7-35.

Listing 7-35. Adding Selection-Related Methods to game Object (game.js)

```
clearSelection: function() {
    while (game.selectedItems.length > 0) {
        game.selectedItems.pop().selected = false;
    }
},
```

```
selectItem: function(item, shiftPressed) {
    // Pressing shift and clicking on a selected item will deselect it
    if (shiftPressed && item.selected) {
        // Deselect item
        item.selected = false;

        for (let i = game.selectedItems.length - 1; i >= 0; i--) {
            if (game.selectedItems[i].uid === item.uid) {
                game.selectedItems.splice(i, 1);
                break;
            }
        }

        return;
    }

    if (item.selectable && !item.selected) {
        item.selected = true;
        game.selectedItems.push(item);
    }
},
```

The `clearSelection()` method iterates through the `game.selectedItems` array, clears the selected flag from each item, and removes the item from the array.

The `selectItem()` method either adds a selectable item to the `selectedItems` array or removes it from the array depending on whether the Shift key is pressed. This way, players can unselect a selected item by clicking it with the Shift key pressed.

At this point, we have all the code we need to select items inside the game. However, we still need a way to highlight selected items so we can identify them visually. This is what we will implement next.

Highlighting Selected Entities

When the player selects an item, we will detect it using the item's `selected` property and draw an enclosing selection boundary around the item. We will also add an indicator to show us how much life the item has.

We will do this by defining two default methods, `drawSelection()` and `drawLifeBar()`, for each of the entities and modify the `draw()` method to call them.

We will start by modifying the `draw()` method inside the `baseItem` object in `common.js` to draw selected items as shown in Listing 7-36.

Listing 7-36. Modifying draw() to Draw Selected Items (common.js)

```
// Default method for drawing an item
draw: function() {
    // Compute pixel coordinates on canvas for drawing item
    this.drawingX = (this.x * game.gridSize) - game.offsetX - this.pixelOffsetX;
    this.drawingY = (this.y * game.gridSize) - game.offsetY - this.pixelOffsetY;

    if (this.selected) {
        this.drawSelection();
        this.drawLifeBar();
    }
```

```
        this.drawSprite();
},

/* Selection related properties */
selectionBorderColor: "rgba(255,255,0,0.5)",
selectionFillColor: "rgba(255,215,0,0.2)",
lifeBarBorderColor: "rgba(0,0,0,0.8)",
lifeBarHealthyFillColor: "rgba(0,255,0,0.5)",
lifeBarDamagedFillColor: "rgba(255,0,0,0.5)",

lifeBarHeight: 5,
```

Within the draw() method, we check if the item is selected, and if so, call its drawSelection() and drawLifeBar() methods. We also add a few selection and life bar–related properties to baseItem.

Next, we will implement these methods in the buildings object (see Listing 7-37).

Listing 7-37. Implementing drawSelection() and drawLifeBar() for Buildings (buildings.js)

```
drawLifeBar: function() {
    let x = this.drawingX + this.pixelOffsetX;
    let y = this.drawingY - 2 * this.lifeBarHeight;

    game.foregroundContext.fillStyle = (this.lifeCode === "healthy") ? this.lifeBarHealthy
    FillColor : this.lifeBarDamagedFillColor;

    game.foregroundContext.fillRect(x, y, this.baseWidth * this.life / this.hitPoints, this.
    lifeBarHeight);

    game.foregroundContext.strokeStyle = this.lifeBarBorderColor;
    game.foregroundContext.lineWidth = 1;

    game.foregroundContext.strokeRect(x, y, this.baseWidth, this.lifeBarHeight);
},

drawSelection: function() {
    let x = this.drawingX + this.pixelOffsetX;
    let y = this.drawingY + this.pixelOffsetY;

    game.foregroundContext.strokeStyle = this.selectionBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.fillStyle = this.selectionFillColor;

    // Draw a filled rectangle around the building
    game.foregroundContext.fillRect(x - 1, y - 1, this.baseWidth + 2, this.baseHeight + 2);
    game.foregroundContext.strokeRect(x - 1, y - 1, this.baseWidth + 2, this.baseHeight + 2);
},
```

The drawLifeBar() method merely draws a bar slightly above the building with a green or red color depending on the life of the building. The length of the bar is proportional to the life of the building. The drawSelection() method draws a yellow rectangle around the base of the building.

Next, we will implement these methods for the vehicles object (see Listing 7-38).

Listing 7-38. Implementing drawSelection() and drawLifeBar() for Vehicles (vehicles.js)

```
drawLifeBar: function() {
    let x = this.drawingX;
    let y = this.drawingY - 2 * this.lifeBarHeight;

    game.foregroundContext.fillStyle = (this.lifeCode === "healthy") ? this.lifeBarHealthy
    FillColor : this.lifeBarDamagedFillColor;

    game.foregroundContext.fillRect(x, y, this.pixelWidth * this.life / this.hitPoints,
    this.lifeBarHeight);

    game.foregroundContext.strokeStyle = this.lifeBarBorderColor;
    game.foregroundContext.lineWidth = 1;

    game.foregroundContext.strokeRect(x, y, this.pixelWidth, this.lifeBarHeight);
},

drawSelection: function() {
    let x = this.drawingX + this.pixelOffsetX;
    let y = this.drawingY + this.pixelOffsetY;

    game.foregroundContext.strokeStyle = this.selectionBorderColor;
    game.foregroundContext.lineWidth = 1;

    // Draw a filled circle around the vehicle
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x, y, this.radius, 0, Math.PI * 2, false);
    game.foregroundContext.fillStyle = this.selectionFillColor;
    game.foregroundContext.fill();
    game.foregroundContext.stroke();
},
```

This time, the drawSelection() method draws a yellow, lightly filled circle under the selected vehicle. Like before, the drawLifeBar() method draws a life bar above the vehicle.

Lastly, we will implement these methods for the aircraft object (see Listing 7-39).

Listing 7-39. Implementing drawSelection() and drawLifeBar() for Aircraft (aircraft.js)

```
drawLifeBar: function() {
    let x = this.drawingX;
    let y = this.drawingY - 2 * this.lifeBarHeight - this.pixelShadowHeight;

    game.foregroundContext.fillStyle = (this.lifeCode === "healthy") ? this.lifeBarHealthy
    FillColor : this.lifeBarDamagedFillColor;
    game.foregroundContext.fillRect(x, y, this.pixelWidth * this.life / this.hitPoints,
    this.lifeBarHeight);
    game.foregroundContext.strokeStyle = this.lifeBarBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.strokeRect(x, y, this.pixelWidth, this.lifeBarHeight);
},
```

207

```
drawSelection: function() {
    let x = this.drawingX + this.pixelOffsetX;
    let y = this.drawingY + this.pixelOffsetY - this.pixelShadowHeight;

    game.foregroundContext.strokeStyle = this.selectionBorderColor;
    game.foregroundContext.fillStyle = this.selectionFillColor;
    game.foregroundContext.lineWidth = 2;

    // Draw a filled circle around the aircraft
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x, y, this.radius, 0, Math.PI * 2, false);
    game.foregroundContext.stroke();
    game.foregroundContext.fill();

    // Draw a circle around the aircraft shadow
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x, y + this.pixelShadowHeight, 4, 0, Math.PI * 2, false);
    game.foregroundContext.stroke();

    // Join the center of the two circles with a line
    game.foregroundContext.beginPath();
    game.foregroundContext.moveTo(x, y);
    game.foregroundContext.lineTo(x, y + this.pixelShadowHeight);
    game.foregroundContext.stroke();
}
```

This time, the `drawLifeBar()` method adjusts for the shadow height when drawing the life bar. The `drawSelection()` method draws a yellow circle around the aircraft, a straight line from the aircraft to the shadow, and finally a small circle at the center of the shadow.

With this last change, we have implemented drawing selections for all the entities. We don't need to implement selections for terrain entities since they cannot be selected within the game.

If we run the game in our browser, we should now be able to select items by either clicking them or dragging the mouse over multiple units. These selected items should then show up highlighted, as shown in Figure 7-13.

Figure 7-13. *Selected items show up highlighted*

Notice that the life bar above the damaged building clearly shows us how badly damaged it is. You can add or subtract items from the selection by clicking them with the Shift key pressed. We have now completely implemented entity selection in our game.

Summary

We covered a lot of ground in this chapter. Starting with an empty level from the previous chapter, we developed a general framework for animating and drawing items within the game by implementing `draw()` and `animate()` methods for these entities.

We handled depth sorting before drawing the items so that items closer to the screen obscured items that were farther away. Using this framework, we then added buildings, vehicles, aircraft, and terrain to our game.

Finally, we implemented the ability to select these entities using the mouse and highlight these selected entities.

In the next chapter, we will implement sending commands to these entities starting with the most important one: movement. We will also look at using pathfinding and steering algorithms so that units navigate intelligently around buildings and other obstacles.

So, let's keep going.

■ ■ ■

Intelligent Unit Movement

In the previous chapter, we built a framework for animating and drawing entities within our game and then added different types of buildings, vehicles, aircraft, and terrain to it. Finally, we added the ability to select these entities.

In this chapter, we will add a framework to give selected units commands and to get the entities to follow orders. We will then implement the most basic of these orders: unit movement by using a combination of pathfinding and steering algorithms to move our units intelligently.

Now let's get started. We will use the code from Chapter 7 as a starting point.

Commanding Units

We will command units using a convention that has now become standard within most modern RTS games. We will select units using left-clicks and command them by using right-clicks.

Right-clicking a navigable spot on the map will command selected units to move to the spot. Right-clicking an enemy unit or building will command all selected units that can attack to attack the enemy. Right-clicking a friendly unit will tell all selected units to follow it around and protect it. And finally, right-clicking an oil field with a harvester vehicle selected will tell the harvester to move to the oil field and deploy on it.

The first thing we need to do is modify the mouse object to handle right-click events, as shown in Listing 8-1.

Listing 8-1. Handle Commands on Right-Click (mouse.js)

```
mouserightclickhandler: function(ev) {
    mouse.rightClick(ev, true);

    // Prevent the browser from showing the context menu
    ev.preventDefault(true);
},

// Called whenever player completes a right-click on the game canvas
rightClick: function() {
    let clickedItem = mouse.itemUnderMouse();

    // Handle actions like attacking and movement of selected units
    if (clickedItem) { // Player right-clicked on something
        if (clickedItem.type !== "terrain") {
            if (clickedItem.team !== game.team) { // Player right-clicked on an enemy item
                let uids = [];
```

```
            // Identify selected units from player's team that can attack
            game.selectedItems.forEach(function(item) {
                if (item.team === game.team && item.canAttack) {
                    uids.push(item.uid);
                }
            }, this);

            // Command units to attack the clicked item
            if (uids.length > 0) {
                game.sendCommand(uids, { type: "attack", toUid: clickedItem.uid });
            }
        } else { // Player right-clicked a friendly item
            let uids = [];

            // Identify selected units from player's team that can move
            game.selectedItems.forEach(function(item) {
                if (item.team === game.team && item.canAttack && item.canMove) {
                    uids.push(item.uid);
                }
            }, this);

            // Command units to guard the clicked item
            if (uids.length > 0) {
                game.sendCommand(uids, { type: "guard", toUid: clickedItem.uid });
            }

        }
    } else if (clickedItem.name === "oilfield") { // Player right-clicked on an oilfield
        let uids = [];

        // Identify the first selected harvester (since only one can deploy at a time)
        for (let i = game.selectedItems.length - 1; i >= 0; i--) {
            let item = game.selectedItems[i];

            if (item.team === game.team && item.type === "vehicles" && item.name ===
            "harvester") {
                uids.push(item.uid);
                break;
            }
        }

        // Command it to deploy on the oil field
        if (uids.length > 0) {
            game.sendCommand(uids, { type: "deploy", toUid: clickedItem.uid });
        }
    }
} else { // Player right-clicked the ground
    let uids = [];

    // Identify selected units from player's team that can move
    game.selectedItems.forEach(function(item) {
```

```
            if (item.team === game.team && item.canMove) {
                uids.push(item.uid);
            }
        }, this);

        // Command units to move to the clicked location
        if (uids.length > 0) {
            game.sendCommand(uids, { type: "move", to: { x: mouse.gameX / game.gridSize,
            y: mouse.gameY / game.gridSize } });
        }
    }
},
```

We start by defining a mouserightclickhandler() method, which calls the rightClick() method and prevents the default browser behavior of showing a context menu on a right-click.

Inside the rightClick() method, when the player right-clicks inside the game map, we first check to see whether the mouse is above an object.

If the player has not clicked an object, we call the game.sendCommand() method to send a move order to all friendly vehicles and aircraft that are selected.

If the player has clicked an object, we similarly send either an attack, guard, or deploy command to the appropriate units. We also pass the UID of the clicked item as a parameter called toUid within the order.

We also need to modify the init() method to add an event listener for the contextmenu event as shown in Listing 8-2.

Listing 8-2. Listening for the contextmenu Event (mouse.js)

```
init: function() {
    // Listen for mouse events on the game foreground canvas
    let canvas = document.getElementById("gameforegroundcanvas");

    canvas.addEventListener("mousemove", mouse.mousemovehandler, false);

    canvas.addEventListener("mouseenter", mouse.mouseenterhandler, false);
    canvas.addEventListener("mouseout", mouse.mouseouthandler, false);

    canvas.addEventListener("mousedown", mouse.mousedownhandler, false);
    canvas.addEventListener("mouseup", mouse.mouseuphandler, false);

    canvas.addEventListener("contextmenu", mouse.mouserightclickhandler, false);

    mouse.canvas = canvas;
},
```

With the right-click logic in place, we now have to implement methods for sending and receiving game commands.

Sending and Receiving Commands

We could have implemented sending commands by modifying the orders property of selected items inside the rightClick() method that we modified earlier. However, we are going to use a slightly more complex implementation.

Any clicking action that generates a command will call the game.sendCommand() method. The sendCommand() method will pass the call to either the singleplayer or multiplayer object. These objects will then send the command details back to the game.processCommand() method. Within the game.processCommand() method, we will update the orders for all the appropriate objects. We will start by adding these methods to the game object inside game.js, as shown in Listing 8-3.

Listing 8-3. Implementing sendCommand() and processCommand() (game.js)

```
// Send command to either singleplayer or multiplayer object
sendCommand: function(uids, details) {
    if (game.type === "singleplayer") {
        singleplayer.sendCommand(uids, details);
    } else {
        multiplayer.sendCommand(uids, details);
    }
},

getItemByUid: function(uid) {
    for (let i = game.items.length - 1; i >= 0; i--) {
        if (game.items[i].uid === uid) {
            return game.items[i];
        }
    }
},

// Receive command from singleplayer or multiplayer object and send it to units
processCommand: function(uids, details) {
    // In case the target "to" object is in terms of uid, fetch the target object
    var toObject;

    if (details.toUid) {
        toObject = game.getItemByUid(details.toUid);
        if (!toObject || toObject.lifeCode === "dead") {
            // The to object no longer exists. Invalid command
            return;
        }
    }

    uids.forEach(function(uid) {
        let item = game.getItemByUid(uid);

        // If uid is for a valid item, set the order for the item
        if (item) {
            item.orders = Object.assign({}, details);
            if (toObject) {
                item.orders.to = toObject;
            }
        }
    });
},
```

The sendCommand() method passes the call to either the singleplayer or multiplayer object's sendCommand() method based on the game type. Using this layer of abstraction allows us to use the same method for both single-player and multiplayer while handling the commands differently.

While the single-player version of sendCommand() will just call processCommand() back immediately, the multiplayer version will send the command to the server, which will then forward the command to all the players at the same time.

We also implement the getItemByUid() method that looks up item UIDs and returns entity objects.

We pass UIDs instead of actual game objects to the sendCommand() method to reduce the size of the sent data, which will become necessary for the multiplayer version of the game. A typical item object contains a lot of details for animating and drawing the object such as methods, sprite sheet images, and all the item properties. While needed for drawing the item, transmitting this extra data to the server and getting it back is a waste of bandwidth and quite unnecessary, especially since the entire object can be replaced by a single integer (the UID), which can be used to look up the object with the getItemByUid() method.

The processCommand() method first looks up any toUid property and gets the resulting item. If no item with the UID exists, it assumes the command is invalid and ignores the command. The method then looks up each of the items passed in the uids array and sets their orders object to a copy of the order details provided in the parameters.

The next thing we will do is implement the singleplayer object's sendCommand() method inside singleplayer.js, as shown in Listing 8-4.

Listing 8-4. Implementing the Single-Player sendCommand() Method (singleplayer.js)

```
sendCommand: function(uids, details) {
    game.processCommand(uids, details);
},
```

As you can see, the implementation of sendCommand() is fairly simple. We merely forward the call to game.processCommand().

Now that we have set up a mechanism for commanding units and setting their orders, we need to set up a way for the units to process these orders and execute them.

Processing Orders

Our implementation for processing orders will be fairly simple. We will implement a method called processOrders() for every entity that needs it and call the processOrders() method for all game items from inside the game animation loop.

We will start by modifying the game object's animationLoop() method inside game.js, as shown in Listing 8-5.

Listing 8-5. Calling processOrders() from Inside the Animation Loop (game.js)

```
animationLoop: function() {
    // Process orders for any item that handles orders
    game.items.forEach(function(item) {
        if (item.processOrders) {
            item.processOrders();
        }
    });

    // Animate each of the elements within the game
    game.items.forEach(function(item) {
        item.animate();
    });
```

```
    // Sort game items into a sortedItems array based on their x,y coordinates
    game.sortedItems = Object.assign([], game.items);

    game.sortedItems.forEach(function(item) {
        item.sortY = item.y + item.pixelOffsetY;
        item.sortX = item.x + item.pixelOffsetX;
    });

    game.sortedItems.sort(function(a, b) {
        // Compare item centers
        return a.sortY - b.sortY + ((a.sortY === b.sortY) ? (b.sortX - a.sortX) : 0);
    });
},
```

The new code iterates through every game item and calls the item's processOrders() method if it exists. Now, we can implement the processOrders() method for the game entities one by one and watch as these entities start obeying our commands.

Let's start by implementing movement for aircraft.

Implementing Aircraft Movement

Unlike moving land vehicles, moving aircraft is fairly simple since aircraft are not affected by terrain, buildings, or other vehicles. When an aircraft is given a move order, it will just turn toward the destination and then move forward in a straight line. Once the aircraft nears its destination, it will go back to its stand state.

We will implement this as a default processOrders() method for aircraft inside aircraft.js, as shown in Listing 8-6.

Listing 8-6. Movement in the Aircraft Object's Default processOrders() Method (aircraft.js)

```
processOrders: function() {
    this.lastMovementX = 0;
    this.lastMovementY = 0;

    if (this.orders.to) {
        var distanceFromDestination = Math.pow(Math.pow(this.orders.to.x - this.x, 2) +
        Math.pow(this.orders.to.y - this.y, 2), 0.5);
        var radius = this.radius / game.gridSize;
    }

    switch (this.orders.type) {
        case "move":
            // Move toward destination until distance from destination is less than aircraft
            radius
            if (distanceFromDestination < radius) {
                this.orders = { type: "stand" };
            } else {
                this.moveTo(this.orders.to, distanceFromDestination);
            }

            break;
    }
},
```

We start by initializing the movementX and movementY variables for later use. We then check the order type inside a case statement.

In case the order type is move, we call the moveTo() method until the aircraft's distance from the destination (stored in the to parameter) is less than the aircraft's radius. Once the aircraft has reached its destination, we change the order back to stand.

Right now, we have implemented only one order. Any time the aircraft gets an order it doesn't know how to handle, it will continue floating at its current location. We will be implementing more orders as we go along.

The next thing we will do is implement a default moveTo() method that will be used by both the aircraft (see Listing 8-7).

Listing 8-7. The Aircraft Object's Default moveTo() Method (aircraft.js)

```
// How slow should unit move while turning
speedAdjustmentWhileTurningFactor: 0.4,

moveTo: function(destination, distanceFromDestination) {
    // Find out where we need to turn to get to destination
    let newDirection = this.findAngle(destination);

    // Turn toward new direction if necessary
    this.turnTo(newDirection);

    // Calculate maximum distance that aircraft can move per animation cycle
    let maximumMovement = this.speed * this.speedAdjustmentFactor * (this.turning ? this.
    speedAdjustmentWhileTurningFactor : 1);
    let movement = Math.min(maximumMovement, distanceFromDestination);

    // Calculate x and y components of the movement
    let angleRadians = -(this.direction / this.directions) * 2 * Math.PI;

    this.lastMovementX = -(movement * Math.sin(angleRadians));
    this.lastMovementY = -(movement * Math.cos(angleRadians));

    this.x = this.x + this.lastMovementX;
    this.y = this.y + this.lastMovementY;
},
```

We first use a findAngle() method to determine the direction toward the destination, and then call the turnTo() method to turn toward the destination if necessary. Next, we calculate the maximum amount that the aircraft can move.

Note that we adjust the speed when the aircraft is turning by using the speedAdjustmentWhile TurningFactor variable. Setting this value to 0 would mean the aircraft would turn in place before moving, while keeping it closer to 1 would mean the aircraft would turn over a huge turning radius.

We then calculate the x and y components of the movement and add it to the aircraft's x and y coordinates, saving these values in lastMovementX and lastMovementY.

Next we will add some common movement- and turning-related code to baseItem in common.js, as shown in Listing 8-8.

Listing 8-8. Movement- and Turning-Related Code in baseItem (common.js)

```
/* Movement-related properties */
speedAdjustmentFactor: 1 / 64,
turnSpeedAdjustmentFactor: 1 / 8,

// Finds the angle toward a destination in terms of a direction (0 <= angle < directions)
findAngle: function (destination) {
    var dy = destination.y - this.y;
    var dx = destination.x - this.x;

    // Convert Arctan to value between (0 - directions)
    var angle = this.directions / 2 - (Math.atan2(dx, dy) * this.directions / (2 * Math.PI));

    angle = (angle + this.directions) % this.directions;

    return angle;
},

// Return the smallest difference (between -directions/2 and +directions/2) toward
newDirection
angleDiff: function(newDirection) {
    let currentDirection = this.direction;
    let directions = this.directions;

    // Make both directions between -directions/2 and +directions/2
    if (currentDirection >= directions / 2) {
        currentDirection -= directions;
    }

    if (newDirection >= directions / 2) {
        newDirection -= directions;
    }

    var difference = newDirection - currentDirection;

    // Ensure difference is also between -directions/2 and +directions/2
    if (difference < -directions / 2) {
        difference += directions;
    }

    if (difference > directions / 2) {
        difference -= directions;
    }

    return difference;
},

turnTo: function(newDirection) {
    // Calculate difference between new direction and current direction
    let difference = this.angleDiff(newDirection);
```

```
    // Calculate maximum amount that aircraft can turn per animation cycle
    let turnAmount = this.turnSpeed * this.turnSpeedAdjustmentFactor;

    if (Math.abs(difference) > turnAmount) {
        // Change direction by turn amount
        this.direction += turnAmount * Math.abs(difference) / difference;

        // Ensure direction doesn't go below 0 or above this.directions
        this.direction = (this.direction + this.directions) % this.directions;

        this.turning = true;
    } else {
        this.direction = newDirection;
        this.turning = false;
    }
},
```

We start by defining two movement-related properties, `speedAdjustmentFactor` and `turnSpeedAdjustmentFactor`. These two factors are used to convert an entity's `speed` and `turnSpeed` values into in-game units for movement and turning.

We then define two methods, `findAngle()` and `angleDiff()`, which are used by our `moveTo()` and `turnTo()` methods. The `findAngle()` method computes the angle toward a destination in terms of our in-game direction, which will be a value between 0 and 8.

The `angleDiff()` method returns the smallest angle that the item needs to turn to point toward a new direction. It returns a value ranging from -4 to 4, with the sign indicating direction, so if the aircraft needs to turn anti-clockwise by 1, it will return a value of -1.

Finally we implement the `turnTo()` method, which uses `angleDiff()` to compute how much the unit needs to turn, and modifies the unit's `direction` property appropriately. We also set the `turning` property to true if the unit is still turning.

We are now ready to start moving our aircraft within the game, but before we do that, let's simplify our level by removing all the unnecessary items from the map. The new `singleplayer` section inside `levels.js` will look like Listing 8-9.

Listing 8-9. Removing Unnecessary Items from the Level (levels.js)

```
"singleplayer": [
    {
        "name": "Movement",
        "briefing": "In this level you will start commanding units and moving them around
        the map.",

        /* Map Details */
        "mapName": "plains",

        /* Starting location for player */
        "startX": 0,
        "startY": 0,

        /* Entities to be loaded */
        "requirements": {
            "buildings": ["base", "starport", "harvester", "ground-turret"],
            "vehicles": ["transport", "harvester", "scout-tank", "heavy-tank"],
            "aircraft": ["chopper", "wraith"],
            "terrain": ["oilfield", "bigrocks", "smallrocks"]
        },
```

219

```
        /* Entities to be added */
        "items": [
            { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
            { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },
            { "type": "buildings", "name": "harvester", "x": 20, "y": 10, "team": "blue" },
            { "type": "buildings", "name": "ground-turret", "x": 24, "y": 7, "team": "blue",
            "direction": 3 },

            { "type": "vehicles", "name": "transport", "x": 24, "y": 10, "team": "blue",
            "direction": 2 },
            { "type": "vehicles", "name": "harvester", "x": 16, "y": 12, "team": "blue",
            "direction": 3 },
            { "type": "vehicles", "name": "scout-tank", "x": 24, "y": 14, "team": "blue",
            "direction": 4 },
            { "type": "vehicles", "name": "heavy-tank", "x": 24, "y": 16, "team": "blue",
            "direction": 5 },

            { "type": "aircraft", "name": "chopper", "x": 7, "y": 9, "team": "blue",
            "direction": 2 },
            { "type": "aircraft", "name": "wraith", "x": 11, "y": 9, "team": "blue",
            "direction": 3 },

            { "type": "terrain", "name": "oilfield", "x": 3, "y": 5, "action": "hint" },
            { "type": "terrain", "name": "bigrocks", "x": 19, "y": 6 },
            { "type": "terrain", "name": "smallrocks", "x": 8, "y": 3 }
        ]
    }
],
```

When you run the game in the browser, you should be able to select the two aircraft and move them around on the new map shown in Figure 8-1.

Figure 8-1. *Moving aircraft around the new map*

When you select an aircraft and right-click the map somewhere, the aircraft should turn and move toward the destination. You will notice that the wraith aircraft moves faster than the chopper because we specified a higher value for speed in the wraith entity's properties.

You may also notice that right-clicking a building or a friendly unit doesn't do anything. This is because right-clicking a friendly item generates the guard order, which we have not yet implemented.

Implementing movement for our aircraft was fairly simple because we took the creative liberty of assuming that aircraft could avoid buildings, vehicles, and other aircraft by virtue of adjusting their height.

However, when it comes to vehicles, we can no longer do that. We need to worry about finding the shortest path between a vehicle and its destination while driving around obstacles such as buildings and terrain. This is where pathfinding comes in.

Pathfinding

Pathfinding, or pathing, is the process of finding the shortest path between two points. Typically it involves the use of various algorithms to traverse a graph of nodes starting at one vertex and exploring adjacent nodes until the destination node is reached.

Two of the most commonly used algorithms for graph-based pathfinding are Dijkstra's algorithm and its variant called the A* (pronounced "A star") algorithm.

A* uses an additional distance heuristic that helps it find paths faster than Dijkstra. Because of its performance and accuracy, it is widely used in games. You can read more about the algorithm at http://en.wikipedia.org/wiki/A*. We will also be using A* for the vehicle pathing in our game.

We will use an excellent MIT-licensed JavaScript implementation of A* by Andrea Giammarchi. The code has been optimized for JavaScript, and its performance even on large graphs is fairly good. This library has been used in game projects such as Mozilla's BrowserQuest demo (http://browserquest.mozilla.org/). You can read about the library at http://webreflection.blogspot.com/2006/10/javascript-path-finder-with-star.html. We will add a reference to the A* implementation (stored in astar.js) to the head section of index.html, as shown in Listing 8-10.

Listing 8-10. Adding Reference to the A* Implementation (index.html)

```
<!-- A* Implementation by Andrea Giammarchi -->
<script src="js/astar.js" type="text/javascript"></script>
```

The implementation, while fairly complex, is relatively easy to use. The code gives us access to an Astar() method that accepts four parameters: the map graph to use, the starting coordinates, the ending coordinates, and optionally a name for the heuristic to use.

The method returns either an array with all the intermediate steps of the shortest path or an empty array in case there is no possible path.

Now that we have our A* algorithm in place, we need to provide it with a graph or grid for pathfinding.

Defining Our Pathfinding Grid

We have already broken our map into a grid of squares with the dimensions 20 pixels by 20 pixels. We will store the pathfinding grid as a two-dimensional array with values of 0 and 1 for passable and impassable squares, respectively. We have already defined a list of all the impassable squares in the mapObstructedTerrain array of our map. This array contains the x and y coordinates for every grid inside our map that is impassable. This includes areas with trees, mountains, water, craters, and lava.

We will use this array to create a terrain grid by defining a createTerrainGrid() method inside the game object as shown in Listing 8-11.

Listing 8-11. Creating the Terrain Grid (game.js)

```
// Create a grid that stores all obstructed tiles as 1 and unobstructed as 0
createTerrainGrid: function() {

    let map = game.currentMap;

    // Initialize Terrain Grid to 2d array of zeroes
    game.currentMapTerrainGrid = new Array(map.gridMapHeight);

    var row = new Array(map.gridMapWidth);

    for (let x = 0; x < map.mapGridWidth; x++) {
        row[x] = 0;
    }

    for (let y = 0; y < map.mapGridHeight; y++) {
        game.currentMapTerrainGrid[y] = row.slice(0);
    }

    // Take all the obstructed terrain coordinates and mark them on the terrain grid as
    unpassable
    map.mapObstructedTerrain.forEach(function(obstruction) {
        game.currentMapTerrainGrid[obstruction[1]][obstruction[0]] = 1;
    }, this);

    // Reset the passable grid
    game.currentMapPassableGrid = undefined;

},
```

We initialize an array called `currentMapTerrainGrid` inside the game object and set it to the dimensions of our map using `mapGridWidth` and `mapGridHeight`, with all of its cells set to 0. We then set all the obstructed tiles to 1 and leave the unobstructed tiles as 0. Finally, we reset a `currentMapPassableGrid` property, which we will be using later.

We will generate the terrain grid when the level starts by calling it from within the `singleplayer` object's `initLevel()` method (see Listing 8-12).

Listing 8-12. Creating the Terrain Grid When Starting Level (singleplayer.js)

```
initLevel: function() {
    game.type = "singleplayer";
    game.team = "blue";

    // Don't allow player to enter mission until all assets for the level are loaded
    var enterMissionButton = document.getElementById("entermission");

    enterMissionButton.disabled = true;

    // Load all the items for the level
    var level = levels.singleplayer[singleplayer.currentLevel];

    game.loadLevelData(level);
```

```
    // Set player starting location
    game.offsetX = level.startX * game.gridSize;
    game.offsetY = level.startY * game.gridSize;

    game.createTerrainGrid();

    // Enable the Enter Mission button once all assets are loaded
    loader.onload = function() {
        enterMissionButton.disabled = false;
    };

    // Update the mission briefing text and show briefing screen
    this.showMissionBriefing(level.briefing);
},
```

If we were to highlight the obstructed squares in currentMapTerrainGrid on our map, it would look like Figure 8-2.

Figure 8-2. *Obstructed grid squares defined in currentMapTerrainGrid*

While currentMapTerrainGrid marks out all the obstacles in the map terrain, it still does not include the buildings and terrain entities on the map.

We will keep another array inside the game object called currentMapPassableGrid that will combine the building and terrain entities and the currentMapTerrainGrid array we defined earlier. This array will need to be re-created every time buildings or terrain get added to or removed from the game. We will do this in a rebuildPassableGrid() method within the game object (see Listing 8-13).

Listing 8-13. Creating the Passable Grid (game.js)

```javascript
// Make a copy of a 2-dimensional array
makeArrayCopy: function(originalArray) {
    var length = originalArray.length;
    var copy = new Array(length);

    for (let i = 0; i < length; i++) {
        copy[i] = originalArray[i].slice(0);
    }

    return copy;
},

rebuildPassableGrid: function() {

    // Initialize Passable Grid with the value of Terrain Grid
    game.currentMapPassableGrid = game.makeArrayCopy(game.currentMapTerrainGrid);

    // Also mark all building and terrain as unpassable items
    for (let i = game.items.length - 1; i >= 0; i--) {
        var item = game.items[i];

        if (item.type === "buildings" || item.type === "terrain") {
            for (let y = item.passableGrid.length - 1; y >= 0; y--) {
                for (let x = item.passableGrid[y].length - 1; x >= 0; x--) {
                    if (item.passableGrid[y][x]) {
                        game.currentMapPassableGrid[item.y + y][item.x + x] = 1;
                    }
                }
            }
        }
    }
},
```

We first copy the `currentMapTerrainGrid` array into `currentMapPassableGrid` using a `makeArrayCopy()` method. We then iterate through all the game items and use the `passableGrid` property, which we defined for all buildings and terrain, to mark out grid squares that are not passable. If we were to highlight the obstructed squares in our map based on `currentMapPassableGrid`, it would look like Figure 8-3.

Figure 8-3. *Obstructed grid squares defined in currentMapPassableGrid*

Because of the way we define `passableGrid` for each building, it is possible to allow portions of buildings to be passable (for example, the lower portion of the starport).

We will need to ensure that `game.currentMapPassableGrid` is reset anytime a building is added or removed from the game. We do this by adding an extra condition inside the `add()` and `remove()` methods of the game object, as shown in Listing 8-14.

Listing 8-14. Clearing currentMapPassableGrid Inside add() and remove() (game.js)

```
add: function(itemDetails) {
    // Set a unique id for the item
    if (!itemDetails.uid) {
        itemDetails.uid = ++game.counter;
    }

    var item = window[itemDetails.type].add(itemDetails);

    // Add the item to the items array
    game.items.push(item);

    // Add the item to the type-specific array
    game[item.type].push(item);

    // Reset currentMapPassableGrid whenever the map changes
    if (item.type === "buildings" || item.type === "terrain") {
        game.currentMapPassableGrid = undefined;
    }

    return item;
},
```

```
remove: function(item) {
    // Unselect item if it is selected
    item.selected = false;
    for (let i = game.selectedItems.length - 1; i >= 0; i--) {
        if (game.selectedItems[i].uid === item.uid) {
            game.selectedItems.splice(i, 1);
            break;
        }
    }

    // Remove item from the items array
    for (let i = game.items.length - 1; i >= 0; i--) {
        if (game.items[i].uid === item.uid) {
            game.items.splice(i, 1);
            break;
        }
    }

    // Remove items from the type-specific array
    for (let i = game[item.type].length - 1; i >= 0; i--) {
        if (game[item.type][i].uid === item.uid) {
            game[item.type].splice(i, 1);
            break;
        }
    }

    // Reset currentMapPassableGrid whenever the map changes
    if (item.type === "buildings" || item.type === "terrain") {
        game.currentMapPassableGrid = undefined;
    }
},
```

Within both methods, we check whether the item being added or removed is a building or terrain type and, if so, reset the currentMapPassableGrid variable.

Now that we have defined the movement grid for our A* algorithm, we are ready to implement vehicle movement.

Implementing Vehicle Movement

We will start by adding a default processOrders() method for the vehicles object inside vehicles.js, as shown in Listing 8-15.

Listing 8-15. The Default processOrders() Method for Vehicles (vehicles.js)

```
processOrders: function() {
    this.lastMovementX = 0;
    this.lastMovementY = 0;

    if (this.orders.to) {
        var distanceFromDestination = Math.pow(Math.pow(this.orders.to.x - this.x, 2) +
        Math.pow(this.orders.to.y - this.y, 2), 0.5);
```

```
        var radius = this.radius / game.gridSize;
    }

    switch (this.orders.type) {
        case "move":
            // Move toward destination until distance from destination is less than vehicle
            radius
            if (distanceFromDestination < radius) {
                this.orders = { type: "stand" };
            } else {
                let moving = this.moveTo(this.orders.to, distanceFromDestination);

                // Pathfinding couldn't find a path so stop
                if (!moving) {
                    this.orders = { type: "stand" };
                    break;
                }
            }

            break;
    }
```

The method is fairly similar to the processOrders() method that we defined for aircraft. The one subtle difference is that we check whether the moveTo() method returns a value of true indicating it is able to move toward the destination and reset the order to stand in case it does not. We do this because it is possible for the pathfinding algorithm to not find a valid path, and moveTo() will return a value indicating this.

Next, we will implement the default moveTo() method for vehicles, as shown in Listing 8-16.

Listing 8-16. The Default moveTo() Method for Vehicles (vehicles.js)

```
// How slow should unit move while turning
speedAdjustmentWhileTurningFactor: 0.5,

moveTo: function(destination, distanceFromDestination) {

    let start = [Math.floor(this.x), Math.floor(this.y)];
    let end = [Math.floor(destination.x), Math.floor(destination.y)];

    // Direction that we will need to turn to reach destination
    let newDirection;

    let vehicleOutsideMapBounds = (start[1] < 0 || start[1] > game.currentMap.mapGridHeight
    - 1 || start[0] < 0 || start[0] > game.currentMap.mapGridWidth);
    let vehicleReachedDestinationTile = (start[0] === end[0] && start[1] === end[1]);

    // Rebuild the passable grid if needed
    if (!game.currentMapPassableGrid) {
        game.rebuildPassableGrid();
    }

    if (vehicleOutsideMapBounds || vehicleReachedDestinationTile) {
        // Don't use A*. Just turn toward destination.
        newDirection = this.findAngle(destination);
```

```
            this.orders.path = [[this.x, this.y], [destination.x, destination.y]];
        } else {
            // Use A* to try and find a path to the destination
            let grid;

            if (destination.type === "buildings" || destination.type === "terrain") {
                // In case of buildings or terrain, modify the grid slightly so algorithm can
                find a path
                // First copy the passable grid
                grid = game.makeArrayCopy(game.currentMapPassableGrid);
                // Then modify the destination to be "passable"
                grid[Math.floor(destination.y)][Math.floor(destination.x)] = 0;
            } else {
                // In all other cases just use the passable grid
                grid = game.currentMapPassableGrid;
            }

            this.orders.path = AStar(grid, start, end, "Euclidean");

            if (this.orders.path.length > 1) {
                // The next step is the center of the next path item
                let nextStep = { x: this.orders.path[1][0] + 0.5, y:
                this.orders.path[1][1] + 0.5 };

                newDirection = this.findAngle(nextStep);
            } else {
                // Let the calling function know that there is no path
                return false;
            }
        }
    }

    // Turn toward new direction if necessary
    this.turnTo(newDirection);

    // Calculate maximum distance that vehicle can move per animation cycle
    let maximumMovement = this.speed * this.speedAdjustmentFactor * (this.turning ?
    this.speedAdjustmentWhileTurningFactor : 1);
    let movement = Math.min(maximumMovement, distanceFromDestination);

    // Calculate x and y components of the movement
    let angleRadians = -(this.direction / this.directions) * 2 * Math.PI;

    this.lastMovementX = -(movement * Math.sin(angleRadians));
    this.lastMovementY = -(movement * Math.cos(angleRadians));

    this.x = this.x + this.lastMovementX;
    this.y = this.y + this.lastMovementY;

    // Let the calling function know that we were able to move
    return true;
},
```

We start by defining the start and end values for our path by truncating the vehicle location and destination. We then check whether game.currentMapPassableGrid is defined and call game. rebuildPassableGrid() if it is not.

Next, we check whether the unit has reached the destination tile, or is outside the game bounds, in which case we ignore pathfinding and just set the new direction using findAngle(). We do this because the AStar() method would fail if we passed it starting coordinates that were outside the grid, and would have nothing to do if the start and end values were the same.

If we do need pathfinding, we first check whether the destination is a building or terrain. If so, we copy the game.currentMapPassableGrid into a grid variable and define the destination grid square as passable. This hack lets the A* algorithm find a path to a building even though the destination is impassable.

The next step is calculating the path and new direction. If the vehicle is within the map bounds, we call the AStar() method while passing it values of start, end, and the grid. We specify a heuristic method of Euclidean, which allows diagonal movement and seems to work well for our game.

If the AStar() method returns a path with at least two elements, we calculate newDirection by finding the angle from the vehicle to the middle of next grid. If the path does not contain at least two elements, we assume this is because AStar() could not find a path and return false to indicate that there was no movement.

After calculating the direction to move in, we use the turnTo() method to turn toward newDirection, and then, just like we did with aircraft, we move along the current vehicle direction and save the distance travelled in the movementX and movementY variables. Finally, we pass back a value of true to indicate that the vehicle was able to move.

If you run the game now, you should be able to select vehicles and move them around the map by right-clicking a spot on the map. The vehicles will move along a path that avoids all the terrain and building obstacles. Figure 8-4 illustrates typical paths returned by the pathfinding algorithm.

Figure 8-4. *Typical movement paths using pathfinding algorithm*

One thing that you will notice is that while the vehicles avoid unpassable terrain, they still drive over other vehicles.

A simple way to fix this is to just mark all squares occupied by any vehicle as unpassable. However, this simplistic approach can end up blocking very large portions of the map since vehicles often move across multiple grid squares. A significant disadvantage of this method is that if we try to move a bunch of vehicles through a narrow passage, the first vehicle will block the passage, causing the pathfinding for the vehicles behind to try to find a longer alternate route or, worse, assume there is no possible path and give up.

A better alternative is to implement a steering step that checks for collisions with other objects and modifies the vehicle's direction while still trying to maintain the original path as far as possible.

Collision Detection and Steering

Steering, like pathfinding, is a fairly vast AI subject. The idea of applying steering behavior in games has been around for a long time, but it was popularized by the work of Craig Reynolds in the mid to late 1980s. His paper "Steering Behaviors for Autonomous Characters" and his Java demos are still considered the basic starting point for developing steering mechanisms in games. You can read more about his research and look at demos of various steering mechanisms at http://www.red3d.com/cwr/steer/.

We will use a fairly simple implementation for our game. We will first check whether moving a vehicle along its present direction will result in collisions with any object.

If there are colliding objects, we will create repulsive forces from any colliding objects to our vehicle and a mild attractive force toward the next grid square in the pathfinding path. We will then combine all of these forces as vectors to see which direction the vehicle will need to move to get away from the collisions. We will steer the vehicle toward this direction until the vehicle is no longer colliding with any object, at which point the vehicle will return to the basic pathfinding mode.

We will also distinguish between hard and soft collisions based on the distance from colliding objects. A vehicle that is about to have a soft collision may still move while turning; however, a vehicle about to have a hard collision will not move forward at all and will only turn away from the collisions.

We will start by implementing two methods for handling collisions inside the default section of the `vehicle` object inside `vehicles.js`, as shown in Listing 8-17.

Listing 8-17. Methods for Handling Collisions (vehicles.js)

```
// Make a list of collisions that the vehicle will have if it goes along present path
checkForCollisions: function(grid) {
    // Calculate new position on present path at maximum speed
    let movement = this.speed * this.speedAdjustmentFactor;
    let angleRadians = -(this.direction / this.directions) * 2 * Math.PI;
    let newX = this.x - (movement * Math.sin(angleRadians));
    let newY = this.y - (movement * Math.cos(angleRadians));

    this.colliding = false;
    this.hardCollision = false;

    // List of objects that will collide after next movement step
    let collisionObjects = [];

    // Test for collision with grid up to 3 squares away from this vehicle
    let x1 = Math.max(0, Math.floor(newX) - 3);
    let x2 = Math.min(game.currentMap.mapGridWidth - 1, Math.floor(newX) + 3);
    let y1 = Math.max(0, Math.floor(newY) - 3);
    let y2 = Math.min(game.currentMap.mapGridHeight - 1, Math.floor(newY) + 3);
```

```
let gridHardCollisionThreshold = Math.pow(this.radius * 0.9 / game.gridSize, 2);
let gridSoftCollisionThreshold = Math.pow(this.radius * 1.1 / game.gridSize, 2);

for (let j = x1; j <= x2;j++) {
    for (let i = y1; i <= y2 ;i++) {
        if (grid[i][j] === 1) { // Grid square is obstructed

            let distanceSquared = Math.pow(j + 0.5 - newX, 2) + Math.pow(i + 0.5 - newY, 2);

            if (distanceSquared < gridHardCollisionThreshold) {
                // Distance of obstructed grid from vehicle is less than hard collision
                threshold
                collisionObjects.push({ collisionType: "hard", with: { type: "wall", x:
                j + 0.5, y: i + 0.5 } });

                this.colliding = true;
                this.hardCollision = true;

            } else if (distanceSquared < gridSoftCollisionThreshold) {
                // Distance of obstructed grid from vehicle is less than soft collision
                threshold
                collisionObjects.push({ collisionType: "soft", with: { type: "wall", x:
                j + 0.5, y: i + 0.5 } });

                this.colliding = true;
            }
        }
    }
}

for (let i = game.vehicles.length - 1; i >= 0; i--) {
    let vehicle = game.vehicles[i];

    // Test vehicles that are less than 3 squares away for collisions
    if (vehicle !== this && Math.abs(vehicle.x - this.x) < 3 && Math.abs
    (vehicle.y - this.y) < 3) {
        if (Math.pow(vehicle.x - newX, 2) + Math.pow(vehicle.y - newY, 2) < Math.
        pow((this.radius + vehicle.radius) / game.gridSize, 2)) {
            // Distance between vehicles is less than hard collision threshold
            (sum of vehicle radii)
            collisionObjects.push({ collisionType: "hard", with: vehicle });

            this.colliding = true;
            this.hardCollision = true;

        } else if (Math.pow(vehicle.x - newX, 2) + Math.pow(vehicle.y - newY, 2)
        < Math.pow((this.radius * 1.5 + vehicle.radius) / game.gridSize, 2)) {
            // Distance between vehicles is less than soft collision threshold
            (1.5 times vehicle radius + colliding vehicle radius)
            collisionObjects.push({ collisionType: "soft", with: vehicle });
```

```
                this.colliding = true;
            }
        }
    }

    return collisionObjects;
},

// Find a direction that steers away from the collision objects
steerAwayFromCollisions: function(collisionObjects) {
    // Create a force vector object that adds up repulsion from all colliding objects
    let forceVector = { x: 0, y: 0 };

    // By default, the next step has a mild attraction force
    collisionObjects.push({ collisionType: "attraction", with: { x: this.orders.path[1][0] +
    0.5, y: this.orders.path[1][1] + 0.5 } });

    for (let i = collisionObjects.length - 1; i >= 0; i--) {
        let collObject = collisionObjects[i];
        let objectAngle = this.findAngle(collObject.with);
        let objectAngleRadians = -(objectAngle / this.directions) * 2 * Math.PI;
        let forceMagnitude;

        switch (collObject.collisionType) {
            case "hard":
                forceMagnitude = 2;
                break;
            case "soft":
                forceMagnitude = 1;
                break;
            case "attraction":
                forceMagnitude = -0.25;
                break;
        }

        forceVector.x += (forceMagnitude * Math.sin(objectAngleRadians));
        forceVector.y += (forceMagnitude * Math.cos(objectAngleRadians));
    }

    // Find a new direction based on the force vector
    let newDirection = this.directions / 2 - (Math.atan2(forceVector.x, forceVector.y) *
    this.directions / (2 * Math.PI));

    newDirection = (newDirection + this.directions) % this.directions;

    return newDirection;
},
```

The first method, checkForCollisions(), makes a list of objects that the vehicle will collide with. We first calculate the new position of the vehicle if it moves along its current direction. We then check whether there are any impassable grid squares nearby that might collide with the vehicle in this new position by

comparing the distances between their centers with certain threshold values based on the vehicle radius. We mark collisions as "hard" if they are colliding or "soft" if they are almost ready to collide. All collisions are then added to the collisionObjects array.

We then repeat this process with the vehicles array by testing all vehicles that are close by for possible collisions using the sum of their radii as a threshold distance.

The second method, steerAwayFromCollisions(), iterates through a list of collision objects and defines a repulsive force for each with a magnitude of either 1 or 2 based on whether the collision is "soft" or "hard." We also define an attractive force toward the next pathfinding grid square. Finally, we add up all these forces into a forceVector object, use it to calculate the direction that would take the vehicle the farthest away from all the forces, and assign it to the newDirection variable, which we then return.

Now that we have these methods in place, we will modify the default moveTo() method we defined earlier to handle collisions (see Listing 8-18).

Listing 8-18. Handling Collisions Inside the default moveTo() Method (vehicles.js)

```
moveTo: function(destination, distanceFromDestination) {

    let start = [Math.floor(this.x), Math.floor(this.y)];
    let end = [Math.floor(destination.x), Math.floor(destination.y)];

    // Direction that we will need to turn to reach destination
    let newDirection;

    let vehicleOutsideMapBounds = (start[1] < 0 || start[1] > game.currentMap.map
    GridHeight - 1 || start[0] < 0 || start[0] > game.currentMap.mapGridWidth);
    let vehicleReachedDestinationTile = (start[0] === end[0] && start[1] === end[1]);

    // Rebuild the passable grid if needed
    if (!game.currentMapPassableGrid) {
        game.rebuildPassableGrid();
    }

    if (vehicleOutsideMapBounds || vehicleReachedDestinationTile) {
        // Don't use A*. Just turn toward destination.
        newDirection = this.findAngle(destination);

        this.orders.path = [[this.x, this.y], [destination.x, destination.y]];
    } else {
        // Use A* to try and find a path to the destination
        let grid;

        if (destination.type === "buildings" || destination.type === "terrain") {
            // In case of buildings or terrain, modify the grid slightly so algorithm can
            find a path
            // First copy the passable grid
            grid = game.makeArrayCopy(game.currentMapPassableGrid);
            // Then modify the destination to be "passable"
            grid[Math.floor(destination.y)][Math.floor(destination.x)] = 0;
        } else {
            // In all other cases just use the passable grid
            grid = game.currentMapPassableGrid;
        }
```

```
        this.orders.path = AStar(grid, start, end, "Euclidean");

        if (this.orders.path.length > 1) {
            // The next step is the center of the next path item
            let nextStep = { x: this.orders.path[1][0] + 0.5, y:
            this.orders.path[1][1] + 0.5 };

            newDirection = this.findAngle(nextStep);
        } else {
            // Let the calling function know that there is no path
            return false;
        }
    }

    // Collision handling and steering
    let collisionObjects = this.checkForCollisions(game.currentMapPassableGrid);

    // Moving along the present path will cause a collision
    if (this.colliding) {
        newDirection = this.steerAwayFromCollisions(collisionObjects);
    }

    // Turn toward new direction if necessary
    this.turnTo(newDirection);

    // Calculate maximum distance that vehicle can move per animation cycle
    let maximumMovement = this.speed * this.speedAdjustmentFactor * (this.turning ? this.
    speedAdjustmentWhileTurningFactor : 1);
    let movement = Math.min(maximumMovement, distanceFromDestination);

    // Don't move if we are in a hard collision
    if (this.hardCollision) {
        movement = 0;
    }

    // Calculate x and y components of the movement
    let angleRadians = -(this.direction / this.directions) * 2 * Math.PI;

    this.lastMovementX = -(movement * Math.sin(angleRadians));
    this.lastMovementY = -(movement * Math.cos(angleRadians));

    this.x = this.x + this.lastMovementX;
    this.y = this.y + this.lastMovementY;

    // Let the calling function know that we were able to move
    return true;
},
```

After the initial pathfinding step, we call the checkCollisionObjects() method and get a list of objects that the vehicle will collide with. If the vehicle is colliding, we then use the steerAwayFromCollisions() method to find a new direction to move toward.

What this means is as long as there are no colliding objects, the vehicle will head toward the next grid square defined in its path. The moment the vehicle senses a collision, its primary motivation will be to avoid the collision by taking evasive action. Once the collision threat has been averted, the vehicle will return to its original path-following behavior.

We also add an extra check to prevent the vehicle from moving forward if movement will result in a hard collision. As a result, the vehicle will stop completely rather than actually collide with another object.

If you run the game now and try to move a vehicle, you will find that it steers around other vehicles to avoid colliding with them.

One problem that you might notice is that if you try to move multiple vehicles to the same spot, the first one stops at the correct location, while the others keep circling around trying in vain to reach the occupied spot. We will need to fix this by adding some intelligence to the way a vehicle handles trying to move to a blocked spot.

The ideal behavior would be to stop at a little distance from the destination if the destination is blocked and to stop even farther away if the vehicle has been colliding for a long time without reaching its destination.

We will implement this by modifying the default processOrders() method, as shown in Listing 8-19.

Listing 8-19. Modifying processOrders() to Handle Stopping (vehicles.js)

```
processOrders: function() {
    this.lastMovementX = 0;
    this.lastMovementY = 0;

    if (this.orders.to) {
        var distanceFromDestination = Math.pow(Math.pow(this.orders.to.x - this.x, 2) +
        Math.pow(this.orders.to.y - this.y, 2), 0.5);
        var radius = this.radius / game.gridSize;
    }

    switch (this.orders.type) {
        case "move":
            // Move toward destination until distance from destination is less than vehicle
            radius
            if (distanceFromDestination < radius) {
                // Stop when within on vehicle radius of destination
                this.orders = { type: "stand" };
            } else if (this.colliding && distanceFromDestination < 3 * radius) {
                // Stop when within 3 radius of the destination if colliding with something
                this.orders = { type: "stand" };
                break;
            } else {
                if (this.colliding && distanceFromDestination < 5 * radius) {
                    // Count collisions within 5 radius distance of goal
                    if (!this.orders.collisionCount) {
                        this.orders.collisionCount = 1;
                    } else {
                        this.orders.collisionCount ++;
                    }

                    // Stop if more than 30 collisions occur
                    if (this.orders.collisionCount > 30) {
                        this.orders = { type: "stand" };
```

```
                        break;
                }
        }

        let moving = this.moveTo(this.orders.to, distanceFromDestination);

        // Pathfinding couldn't find a path so stop
        if (!moving) {
            this.orders = { type: "stand" };
            break;
        }
    }
}

    break;
}
},
```

We first try to stop at the destination if the vehicle is within 1 radius of the destination. We also stop if the vehicle is colliding and within a 3-radius distance of the destination. Finally, we stop if the vehicle has been colliding more than 30 times while being within a 5-radius distance of the destination. This last condition handles situations where the vehicle has been bumping around a crowded area for a while without finding a way to reach its destination.

If you run the game now and try to move multiple vehicles together, you will see that they intelligently stop near their destination even in crowded areas.

At this point, we have a reasonably good pathfinding and steering solution for intelligent unit movement. This system can be developed further to improve performance and add other intelligent behavior such as queuing, flocking, and leader following, depending on your game requirements. You should definitely research this topic further as you implement unit movement within your own games, starting with the work by Craig Reynolds (`www.red3d.com/cwr/steer/`).

Now that we have vehicle movement working, let's take the time to implement one more movement-related order: deploying the harvester.

Deploying the Harvester

We designed the harvester as a deployable vehicle that opens up into a harvester building when deployed on an oil field. We already set up the code to pass the deploy order to a harvester when the player right-clicks an oil field. Now we will implement the deploy case within the vehicle object's processOrders() method inside vehicles.js, as shown in Listing 8-20.

Listing 8-20. Implementation of the deploy Case Inside processOrders() (vehicles.js)

```
case "deploy":
    // If oil field has been used already, then cancel order
    if (this.orders.to.lifeCode === "dead") {
        this.orders = { type: "stand" };

        return;
    }
```

```
if (distanceFromDestination < radius + 1) {
    // After reaching within 1 square of oil field, turn harvester to point toward left
    (direction 6)
    this.turnTo(6);

    if (!this.turning) {
        // If oil field has been used already, then cancel order
        if (this.orders.to.lifeCode === "dead") {
            this.orders = { type: "stand" };

            return;
        }

        // Once it is pointing to the left, remove the harvester and oil field and
        deploy a harvester building
        game.remove(this.orders.to);
        this.orders.to.lifeCode = "dead";

        game.remove(this);
        this.lifeCode = "dead";

        game.add({ type: "buildings", name: "harvester", x: this.orders.to.x, y: this.
        orders.to.y, action: "deploy", team: this.team });
    }
} else {
    let moving = this.moveTo(this.orders.to, distanceFromDestination);

    // Pathfinding couldn't find a path so stop
    if (!moving) {
        this.orders = { type: "stand" };
    }
}

break;
```

We start by using the moveTo() method to move the harvester to the oil field. Once the harvester reaches the oil field, we use the turnTo() method and turn the harvester toward the left (direction 6). Finally, we remove the harvester vehicle and oil field items from the game and add a harvester building at the oil field's location with the action set to deploy. While moving toward the oil field and just before deploying, we check whether the oil field has already been used, and if so, cancel the order and go back to stand mode.

If we run our game, select the harvester vehicle, and then right-click the oil field, we should see the harvester move to the oil field and deploy into a building, as shown in Figure 8-5.
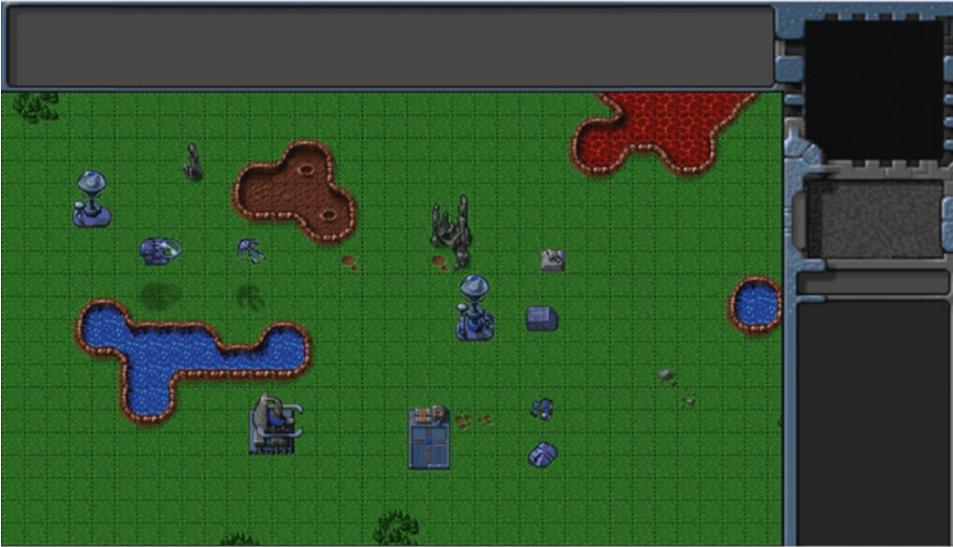
Figure 8-5. *Harvester vehicle deploying into a harvester building*

The harvester moves to the oil field, turns into position, and then seems to expand into a harvester building. As you can see, with the movement framework in place, handling different orders is very easy.

Before we wrap up unit movement, we will address one last thing. You may have noticed that the unit movement, especially for fast units such as the wraith, seems a little choppy. We will try to smoothen this unit movement.

Smoother Unit Movement

Our game animation loop currently runs at a steady 10 frames per second. Even though our drawing loop runs faster (typically 30 to 60 frames per second), it has no new information to draw during these extra loops, so effectively it too draws at 10 frames per second. This results in the choppy-looking movement that we see.

One simple way to make the animation look much smoother is to interpolate the vehicle movement between animation frames. We can calculate the time since the last animation loop and use it to create an interpolation factor that is used to position the units during intermediate drawing loops. This little adjustment will make the units seem to move at a much higher frame rate, even though they are actually being moved only at 10 frames per second.

We will start by modifying the game object's `animationLoop()` method to save the last animation time and the `drawingLoop()` method to calculate an interpolation factor based on the current drawing time and the last animation time. The final version of `animationLoop()` and `drawingLoop()` will look like Listing 8-21.

Listing 8-21. Calculating a Movement Interpolation Factor (game.js)

```
animationLoop: function() {
    // Process orders for any item that handles orders
    game.items.forEach(function(item) {
        if (item.processOrders) {
            item.processOrders();
        }
    });
```

```
    // Animate each of the elements within the game
    game.items.forEach(function(item) {
        item.animate();
    });

    // Sort game items into a sortedItems array based on their x,y coordinates
    game.sortedItems = Object.assign([], game.items);
    game.sortedItems.sort(function(a, b) {
        return a.y - b.y + ((a.y === b.y) ? (b.x - a.x) : 0);
    });

    // Save the time that the last animation loop completed
    game.lastAnimationTime = Date.now();
},

drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();

    // Check the time since the game was animated and calculate a linear interpolation
    factor (-1 to 0)
    game.lastDrawTime = Date.now();
    if (game.lastAnimationTime) {
        game.drawingInterpolationFactor = (game.lastDrawTime - game.lastAnimationTime)
        / game.animationTimeout - 1;

        // No point interpolating beyond the next animation loop...
        if (game.drawingInterpolationFactor > 0) {
            game.drawingInterpolationFactor = 0;
        }
    } else {
        game.drawingInterpolationFactor = -1;
    }

    // Draw the background whenever necessary
    game.drawBackground();

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0, 0, game.canvasWidth, game.canvasHeight);

    // Start drawing the foreground elements
    game.sortedItems.forEach(function(item) {
        item.draw();
    });

    // Draw the mouse
    mouse.draw();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

We save the current time into game.lastAnimationTime at the end of the animationLoop() method. We then use this variable and the current time to calculate the game.drawingInterpolationFactor variable that is a number between -1 and 0. A value of -1 indicates that we draw the unit at its previous location, while a value of 0 means that we draw the unit at its present location. Any value between -1 and 0 means that we draw the unit at an intermediate location between the two points. We cap the value at 0 to prevent any extrapolation from happening (i.e., drawing the unit beyond the point that it has been animated).

Now that we have calculated the interpolation factor, we will use it along with the unit lastMovementX and lastMovementY values to position the element while drawing. First, we will modify the default draw() method for the baseItem object inside common.js, as shown in Listing 8-22.

Listing 8-22. Interpolating Movement While Drawing (common.js)

```
// Default method for drawing an item
draw: function() {
    // Compute pixel coordinates on canvas for drawing item
    this.drawingX = (this.x * game.gridSize) - game.offsetX - this.pixelOffsetX;
    this.drawingY = (this.y * game.gridSize) - game.offsetY - this.pixelOffsetY;

    // Adjust based on interpolation factor
    if (this.canMove) {
        this.drawingX += this.lastMovementX * game.drawingInterpolationFactor * game.
        gridSize;
        this.drawingY += this.lastMovementY * game.drawingInterpolationFactor * game.
        gridSize;
    }

    if (this.selected) {
        this.drawSelection();
        this.drawLifeBar();
    }

    this.drawSprite();
},
```

The only change that we made is adding the extrapolation-related term to the x and y coordinate calculations if the item has a canMove flag, which we have set only for vehicles and aircraft. If we run the game and move the units around, the movement should now be visibly smoother than before.

Note that while the movement itself is now smooth, you might notice that turning still seems somewhat jerky. This is due to the fact that we use a limited number of sprites with only 8 directions. When making your own games, generating sprites with more directions (16 or 32) should make turning look significantly smoother.

With this last change, we can now consider unit movement wrapped up.

Summary

In this chapter, we implemented intelligent unit movement for our game.

We started by developing a framework to give selected units commands and for the entities to then follow orders.

We implemented the `move` order for aircraft by moving them straight toward their destination and for vehicles by using a combination of A* for pathfinding and repulsive forces for steering. We then implemented the `deploy` order for harvesters using the movement code we developed.

Finally, we smoothened the unit movement by integrating an interpolation step within our drawing code.

In the next chapter, we will implement more of our game rules: creating and placing buildings, teleporting vehicles and aircraft from the starport, and harvesting for money. So, let's keep going.

Adding More Game Elements

In the previous chapter, we developed a framework for unit movement that combines pathfinding and steering. We used this framework to implement `move` and `deploy` orders for the vehicles. Finally, we made our unit movement look smoother by interpolating the movement steps during intermediate drawing cycles.

We now have a game where the player can select units and command them to move around the map.

In this chapter, we will build upon this code by adding some more game elements. We will start by implementing an economy where the player can earn money by harvesting and then spend the money on creating buildings and units.

We will then build a framework to create scripted events within a game level, which we can use to control the game story line. We will also add the ability to display messages and notifications to the user. We will then use these elements to handle the completion of a mission within a level.

Let's get started. We will use the code from Chapter 8 as a starting point.

Implementing the Basic Economy

Our game will have a fairly simple economic system. Players will start each mission with an initial amount of money. They can then earn more by deploying a harvester at an oil field. Players will be able to see their cash balance in the sidebar. Once players have sufficient money, they can use it to purchase buildings and units using the sidebar functionality.

The first thing we will do is modify the game to provide money to the player when the level starts.

Setting the Starting Money

We will start by removing some of the extra items in the `items` array and specifying the starting cash for both players in the first map inside `levels.js`, as shown in Listing 9-1.

Listing 9-1. Setting the Starting Cash Amount for the Level (levels.js)

```
/* Entities to be added */
"items": [
    { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },

    { "type": "vehicles", "name": "harvester", "x": 16, "y": 12, "team": "blue",
    "direction": 3, "uid": -1 },
    { "type": "terrain", "name": "oilfield", "x": 3, "y": 5, "action": "hint" },
```

```
    { "type": "terrain", "name": "bigrocks", "x": 19, "y": 6 },
    { "type": "terrain", "name": "smallrocks", "x": 8, "y": 3 }
],

/* Economy Related*/
"cash": {
    "blue": 5000,
    "green": 1000
},
```

We removed all the unnecessary items from the items list. We also added a cash object that sets the starting cash for the blue team to 5000 and for the green team to 1000.

You may have noticed that we have specified a UID for the harvester vehicle. We will use this later in the chapter when we handle triggers and scripted events.

■ **Note**   We use negative values when we specify UIDs for an item so we can be sure that the UIDs will never clash with autogenerated UIDs, which are always positive.

Next, we will need to load these cash values inside the game object's loadLevelData() method, as shown in Listing 9-2.

Listing 9-2. Loading Cash Amount When Loading Level (game.js)

```
loadLevelData: function(level) {
    game.currentLevel = level;
    game.currentMap = maps[level.mapName];

    // Load all the assets for the level starting with the map image
    game.currentMapImage = loader.loadImage("images/maps/" + maps[level.mapName].mapImage);

    // Initialize all the arrays for the game
    game.resetArrays();

    // Load all the assets for every entity defined in the level requirements array
    for (let type in level.requirements) {
        let requirementArray = level.requirements[type];

        requirementArray.forEach(function(name) {
            if (window[type] && typeof window[type].load === "function") {
                window[type].load(name);
            } else {
                console.log("Could not load type :", type);
            }
        });
    }

    // Add all the items defined in the level items array to the game
    level.items.forEach(function(itemDetails) {
        game.add(itemDetails);
    });
```

```
    // Load starting cash for the level
    game.cash = Object.assign({}, level.cash);
},
```

We make a copy of the level.cash object using the Object.assign() method. This is so that changes to game.cash do not affect the level data.

At this point, the game should load the starting cash amount for both players when the level is loaded. However, before we can see the cash value, we need to implement the sidebar.

Implementing the Sidebar

We will implement the sidebar functionality by defining a few methods within a sidebar object inside sidebar.js, as shown in Listing 9-3.

Listing 9-3. Creating the sidebar Object (sidebar.js)

```
var sidebar = {

    init: function() {
        this.cash = document.getElementById("cash");
    },

    animate: function() {
        // Display the current cash balance value
        this.updateCash(game.cash[game.team]);
    },

    // Cache the value to avoid unnecessary DOM updates
    _cash: undefined,
    updateCash: function(cash) {
        // Only update the DOM value if it is different from cached value
        if (this._cash !== cash) {
            this._cash = cash;
            // Display the cash amount with commas
            this.cash.innerHTML = cash.toLocaleString();
        }
    },
};
```

The init() method saves a reference to the cash div element. The animate() method calls updateCash() with the player's cash value. The updateCash() method updates the cash div if the value has changed.

We will call the sidebar.animate() method from within the game object's animationLoop() method, as shown in Listing 9-4.

Listing 9-4. Calling sidebar.animate() from game.animationLoop() (game.js)

```
animationLoop: function() {

    // Animate the sidebar
    sidebar.animate();
```

```
    // Process orders for any item that handles orders
    game.items.forEach(function(item) {
        if (item.processOrders) {
            item.processOrders();
        }
    });

    // Animate each of the elements within the game
    game.items.forEach(function(item) {
        item.animate();
    });

    // Sort game items into a sortedItems array based on their x,y coordinates
    game.sortedItems = Object.assign([], game.items);
    game.sortedItems.sort(function(a, b) {
        return a.y - b.y + ((a.y === b.y) ? (b.x - a.x) : 0);
    });

    // Save the time that the last animation loop completed
    game.lastAnimationTime = Date.now();
},
```

Next, we will call the sidebar.init() method from inside the game.init() method when the game is initialized, as shown in Listing 9-5.

Listing 9-5. Initializing the Sidebar from Inside game.init() (game.js)

```
// Start initializing objects, preloading assets, and display start screen
init: function() {
    // Initialize objects
    loader.init();
    mouse.init();
    sidebar.init();

    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

Finally, we will add a reference to sidebar.js inside the head section of index.html, as shown in Listing 9-6.

Listing 9-6. Adding a Reference to sidebar.js (index.html)

```
<script src="js/sidebar.js" type="text/javascript"></script>
```

If we run the code so far, we should see the player's cash balance in the sidebar area, as shown in Figure 9-1.

Figure 9-1. *Cash balance shown on sidebar*

Now that we have a basic sidebar with a cash balance, we will implement a way for the player to generate more money by harvesting.

Generating Money

We already implemented the ability to deploy a harvester in the previous chapter. To start earning money when harvesting, we will modify the deploy animation state and implement a new harvest animation state in the default processActions() method inside buildings.js, as shown in Listing 9-7.

Listing 9-7. Implementing harvest State Inside processActions() (buildings.js)

```
case "deploy":
    this.imageList = this.spriteArray["deploy"];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    // Once deploying is complete, go to harvest
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "harvest";
    }

    break;

case "harvest":
    this.imageList = this.spriteArray[this.lifeCode];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;
```

```
if (this.animationIndex >= this.imageList.count) {
    this.animationIndex = 0;
    if (this.lifeCode === "healthy") {
        // Harvesters mine 2 credits of cash per animation cycle
        game.cash[this.team] += 2;
    }
}

break;
```

The harvest case is similar to the stand case. However, every time the animation runs through one complete cycle, we add two credits to the player's cash balance. We do this only if the harvester building is not damaged.

We also modify the deploy state to roll over into the harvest state instead of the stand state. This way, once the harvester is deployed, it will automatically start earning money.

If we start the game and deploy the harvester into the oil field, we should see the cash balance slowly increasing, as shown in Figure 9-2.
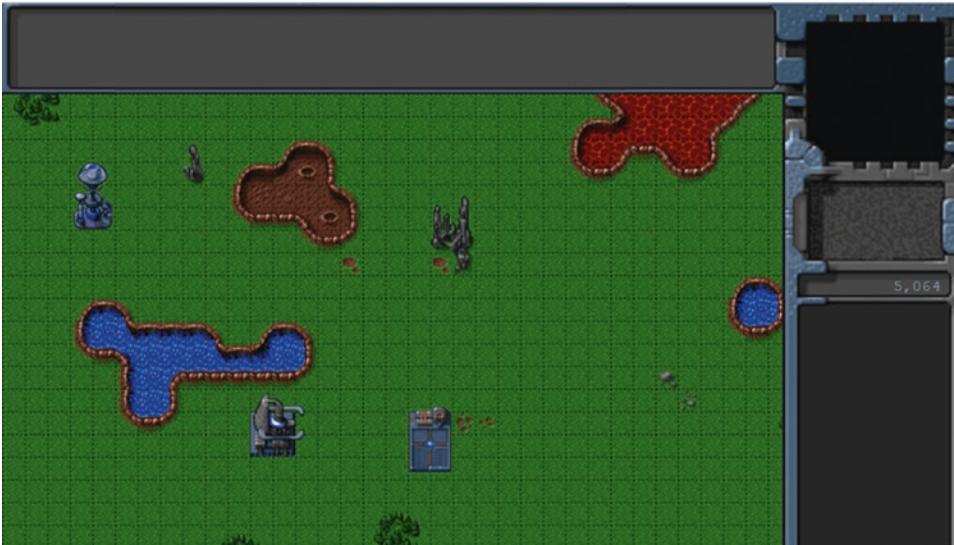
Figure 9-2. *Deployed harvester slowly earning money*

We now have a basic game economy set up. We are ready to implement the purchase of buildings and units.

Purchasing Buildings and Units

In our game, the base building is used to construct buildings, and the starport is used to construct vehicles and aircraft. Players will purchase items by selecting the building they want to construct from and then clicking the appropriate purchase button on the sidebar.

We will start by adding these purchase buttons to our sidebar.

Adding Sidebar Buttons

We will first add the HTML markup for the buttons to the gameinterfacescreen div inside index.html, as shown in Listing 9-8.

Listing 9-8. Adding the Sidebar Purchase Buttons (index.html)

```
<div id="gameinterfacescreen" class="gamelayer">
    <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
    <img src="images/screens/interface-right-game.png" class="right-panel" draggable="false">
    <div id="gamemessages"></div>
    <div id="callerpicture"></div>
    <div id="cash"></div>
    <div id="sidebarbuttons">
        <input type="button" id="starport" title = "Starport">
        <input type="button" id="ground-turret" title = "Turret">
        <input type="button" id="harvester" title = "Harvester">
        <input type="button" id="scout-tank" title = "Scout Tank">
        <input type="button" id="heavy-tank" title = "Heavy Tank">
        <input type="button" id="chopper" title = "Copter">
        <input type="button" id="wraith" title = "Wraith">
    </div>
    <canvas id="gamebackgroundcanvas"></canvas>
    <canvas id="gameforegroundcanvas"></canvas>
</div>
```

Next we will add the appropriate CSS styles for these buttons to styles.css, as shown in Listing 9-9.

Listing 9-9. CSS Styles for Sidebar Buttons (styles.css)

```
/* Sidebar Buttons */

#sidebarbuttons {
    position: absolute;

    right: 12px;
    top: 265px;

    width: 130px;
    height: 214px;

    padding: 0;
    margin: 0;

    overflow: hidden;
}
```

```css
#sidebarbuttons input[type="button"] {
    width: 64px;
    height: 54px;
    padding: 0;
    margin: 0;
    background-color: transparent;
    position: absolute;
}

/* Starport Button */

#starport {

    background-position: -4px -247px;
}

#starport:active, #starport:disabled {
    background-position: -208px -247px;
}

/* Turret Button */

#ground-turret {
    right: 0;
    /*background-position: -140px -247px;*/
    background-position: -74px -247px;
}

#ground-turret:active, #ground-turret:disabled {
    /*background-position: -344px -247px;*/
    background-position: -278px -247px;
}

/* Scout Tank */

#scout-tank {
    top: 107px;
    background-position: -4px -306px;
}

#scout-tank:active, #scout-tank:disabled {
    background-position: -208px -306px;
}

/* Heavy Tank */

#heavy-tank {
    right: 0;
    top: 107px;
    background-position: -74px -306px;
}
```

250

```css
#heavy-tank:active, #heavy-tank:disabled {
    background-position: -278px -306px;
}

/* Harvester */

#harvester {
    top: 55px;
    background-position: -140px -364px;
}

#harvester:active, #harvester:disabled {
    background-position: -344px -364px;
}

/* Chopper */

#chopper {
    top: 159px;
    background-position: -4px -364px;
}

#chopper:active, #chopper:disabled {
    background-position: -208px -364px;
}

/* Wraith */

#wraith {
    background-position: -74px -364px;
    top: 159px;
    right: 0;
}

#wraith:active, #wraith:disabled {
    background-position: -278px -364px;
}
```

The HTML markup adds the buttons to the sidebar, while the CSS styles define images for these buttons using the buttons.png file.

If we run the game in the browser, we should see the purchase buttons in the sidebar, as shown in Figure 9-3.

Figure 9-3. *Purchase buttons in the sidebar*

At this point, all of the buttons look enabled and active; however, clicking the buttons does not do anything. The buttons need to be enabled or disabled depending on whether the player is allowed to construct the items.

Enabling and Disabling Sidebar Buttons

The next thing we will do is to ensure that sidebar buttons are enabled only if the item has been added to the level, the appropriate building for constructing the item is selected, and the player has enough money to construct the item. We will do this by adding two new methods to the sidebar object as shown in Listing 9-10.

Listing 9-10. Enabling and Disabling Sidebar Buttons (sidebar.js)

```
constructables: undefined,
initRequirementsForLevel: function() {
    this.constructables = {};
    let constructableTypes = ["buildings", "vehicles", "aircraft"];

    constructableTypes.forEach(function(type) {
        for (let name in window[type].list) {
            let details = window[type].list[name];
            let isInRequirements = game.currentLevel.requirements[type].indexOf(name) > -1;

            if (details.canConstruct) {
                sidebar.constructables[name] = {
                    name: name,
                    type: type,
                    permitted: isInRequirements,
```

```
                    cost: details.cost,
                    constructedIn: (type === "buildings") ? "base" : "starport"
                };
            }
        }
    });
},

enableSidebarButtons: function() {
    let cashBalance = game.cash[game.team];

    // Check if player has a base or starport selected
    let baseSelected = false;
    let starportSelected = false;

    game.selectedItems.forEach(function(item) {
        if (item.team === game.team && item.lifeCode === "healthy" && item.action === "stand") {
            if (item.name === "base") {
                baseSelected = true;
            } else if (item.name === "starport") {
                starportSelected = true;
            }
        }
    });

    for (let name in this.constructables) {
        let item = this.constructables[name];
        let button = document.getElementById(name);

        // Does player have sufficient money to buy item
        let sufficientMoney = cashBalance >= item.cost;
        // Does the player have the appropriate building selected?
        let correctBuilding = (baseSelected && item.constructedIn === "base")
            || (starportSelected && item.constructedIn === "starport");

        button.disabled = !(item.permitted && sufficientMoney && correctBuilding);

    }
}
```

The first method, initRequirementsForLevel(), iterates through the list array of the buildings, vehicles, and aircraft objects. For each item in the list, it checks whether the item's canConstruct flag is true, and if so, adds some item details into the constructables object, including whether the item has been added to the level requirements.

Within the enableSidebarButton() method, we first check whether a valid base or starport has been selected. A valid base or starport belongs to the player, is healthy, and is currently in stand mode, which means it is not currently constructing anything else.

We enable the button for a building if the base has been selected, the building type has been loaded in the level requirements, and the player has enough cash to buy the building. We do the same thing for vehicles and aircraft if a valid starport has been selected.

We will call the `initRequirementsForLevel()` method when the level is loaded in the `game.loadLevel Data()` method, as shown in Listing 9-11.

Listing 9-11. Calling initLevelRequirements() During Level Load (game.js)

```
loadLevelData: function(level) {
    game.currentLevel = level;
    game.currentMap = maps[level.mapName];

    // Load all the assets for the level starting with the map image
    game.currentMapImage = loader.loadImage("images/maps/" + maps[level.mapName].mapImage);

    // Initialize all the arrays for the game
    game.resetArrays();

    // Load all the assets for every entity defined in the level requirements array
    for (let type in level.requirements) {
        let requirementArray = level.requirements[type];

        requirementArray.forEach(function(name) {
            if (window[type] && typeof window[type].load === "function") {
                window[type].load(name);
            } else {
                console.log("Could not load type :", type);
            }
        });
    }

    // Add all the items defined in the level items array to the game
    level.items.forEach(function(itemDetails) {
        game.add(itemDetails);
    });

    // Load starting cash for the level
    game.cash = Object.assign({}, level.cash);

    sidebar.initRequirementsForLevel();

},
```

We will call the `enableSidebarButtons()` method from the `sidebar.animate()` method, as shown in Listing 9-12.

Listing 9-12. Calling enableSidebarButtons() from animate() (sidebar.js)

```
animate: function() {
    // Display the current cash balance value
    this.updateCash(game.cash[game.team]);

    // Enable buttons if player has sufficient cash and has the correct building selected
    this.enableSidebarButtons();
},
```

254

If we run the game now, the sidebar buttons will get enabled once we select a base or starport, as shown in Figure 9-4.
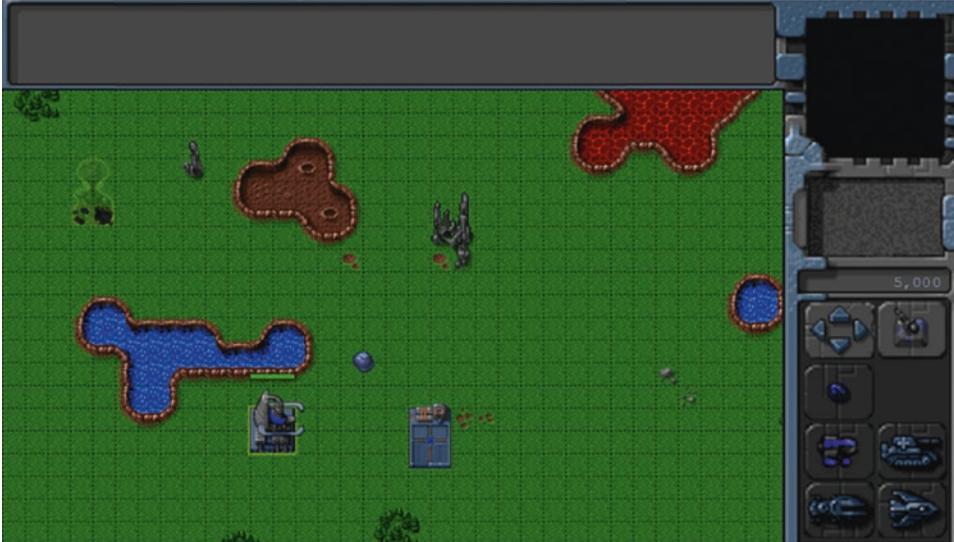
Figure 9-4. *Sidebar building construction buttons enabled by selecting base*

As you can see in the figure, the building buttons have been enabled while the vehicle and aircraft buttons are disabled because the base has been selected. We can similarly activate the vehicle and aircraft construction buttons by selecting the starport.

Now it's time to implement constructing vehicles and aircraft at the starport.

Constructing Vehicles and Aircraft at the Starport

The first thing we will do is modify the sidebar object's init() method to handle the click event for the buttons, as shown in Listing 9-13.

Listing 9-13. Setting click Event for Sidebar Buttons (sidebar.js)

```
init: function() {
    this.cash = document.getElementById("cash");

    let buttons = document.getElementById("sidebarbuttons").getElementsByTagName("input");

    Array.prototype.forEach.call(buttons, function(button) {
        button.addEventListener("click", function() {
            // The input button id is the name of the object that needs to be constructed
            let name = this.id;
            let details = sidebar.constructables[name];

            if (details.type === "buildings") {
                sidebar.constructBuilding(details);
```

```
        } else {
            sidebar.constructInStarport(details);
        }
    });
});
},
```

In the newly added code, we iterate through all the buttons in the sidebar, and assign each button the same click handler. Within the click handler, we look up the details for the item using the button's id and call either the `constructBuilding()` method or `constructInStarport()` method depending on whether the item is a building or unit.

Next, we will define the `constructInStarport()` method as shown in Listing 9-14.

Listing 9-14. Defining the constructInStarport() Method (sidebar.js)

```
constructInStarport: function(details) {

    // Search for a selected starport which can construct the unit
    let starport;

    for (let i = game.selectedItems.length - 1; i >= 0; i--) {
        let item = game.selectedItems[i];

        if (item.name === "starport" && item.team === game.team
            && item.lifeCode === "healthy" && item.action === "stand") {

            starport = item;
            break;
        }
    }

    // If an eligible starport is found, tell it to make the unit
    if (starport) {
        game.sendCommand([starport.uid], { type: "construct-unit", details: details });
    }
},
```

Within the method, we get the first eligible starport among the selected items. We then use the `game.sendCommand()` method to send the starport a `construct-unit` command with details of the unit to construct.

Next, we will create a `processOrder()` method for the starport building that implements the `construct-unit` order. We will add this method inside the starport definition, as shown in Listing 9-15.

Listing 9-15. Implementing processOrder() for the Starport (buildings.js)

```
"starport": {
    name: "starport",
    pixelWidth: 40,
    pixelHeight: 60,
    baseWidth: 40,
    baseHeight: 55,
```

```
pixelOffsetX: 1,
pixelOffsetY: 5,
buildableGrid: [
    [1, 1],
    [1, 1],
    [1, 1]
],
passableGrid: [
    [1, 1],
    [0, 0],
    [0, 0]
],
sight: 3,
cost: 2000,
canConstruct: true,
hitPoints: 300,
spriteImages: [
    { name: "teleport", count: 9 },
    { name: "closing", count: 18 },
    { name: "healthy", count: 4 },
    { name: "damaged", count: 1 }
],

isUnitOnTop: function() {
    let unitOnTop = false;

    for (let i = game.items.length - 1; i >= 0; i--) {
        let item = game.items[i];

        if (item.type === "vehicles" || item.type === "aircraft") {
            if (item.x > this.x && item.x < this.x + 2 && item.y > this.y && item.y
            < this.y + 3) {
                unitOnTop = true;
                break;
            }
        }
    }

    return unitOnTop;
},

processOrders: function() {
    switch (this.orders.type) {
        case "construct-unit":
            if (this.lifeCode !== "healthy") {
                // If the building isn't healthy, ignore the order
                this.orders = { type: "stand" };
                break;
            }
```

```
                var unitOnTop = this.isUnitOnTop();
                var cost = window[this.orders.details.type].list[this.orders.details.name].cost;
                var cash = game.cash[this.team];

                if (unitOnTop) {
                    // Check whether there is a unit standing on top of the building
                    if (this.team === game.team) {
                        game.showMessage("system", "Warning! Cannot teleport unit while
                        landing bay is occupied.");
                    }

                } else if (cash < cost) {
                    // Check whether player has insufficient cash
                    if (this.team === game.team) {
                        game.showMessage("system", "Warning! Insufficient Funds. Need " +
                        cost + " credits.");
                    }
                } else {
                    this.action = "open";
                    this.animationIndex = 0;

                    let itemDetails = Object.assign({}, this.orders.details);

                    // Position new unit above center of starport
                    itemDetails.x = this.x + 0.5 * this.pixelWidth / game.gridSize;
                    itemDetails.y = this.y + 0.5 * this.pixelHeight / game.gridSize;

                    // Subtract the cost from player cash
                    game.cash[this.team] -= cost;

                    // Set unit to be teleported in once it is constructed
                    itemDetails.action = "teleport";
                    itemDetails.team = this.team;
                    this.constructUnit = itemDetails;
                }

                this.orders = { type: "stand" };
                break;
            }
        }
},
```

We start by checking whether the starport is healthy, and if not, we clear the order. Next we check if any unit is already positioned above the starport, and if so, we use the game.showMessage() method to notify the player that a unit cannot be teleported while the landing bay is occupied. Next we check whether we have sufficient funds and, if not, notify the user.

Note that when showing messages using game.showMessage(), we confirm that the game.team is the same as the building's team. This ensures that a message is only shown to the building owner, which will be important to us when we implement multiplayer. If we do not add this check, every player in the game would be shown the message.

Finally, we implement the actual purchase of the unit. We first set the animation action of the building to open. We then set the `position`, `action`, and `team` properties for the item. We save the details of the new unit in the `constructUnit` variable and finally subtract the cost of the item from the player's cash balance.

You may have noticed that we set a `teleport` action for the newly constructed unit. We will need to implement this for both vehicles and aircraft.

Next we will modify the open animation state inside the `buildings` object's `processActions()` method to add the unit to the game, as shown in Listing 9-16.

Listing 9-16. Adding the Unit Once the Starport Opens (buildings.js)

```
case "open":
    this.imageList = this.spriteArray["closing"];
    // Opening is just the closing sprites running backwards
    this.imageOffset = this.imageList.offset + this.imageList.count - this.animationIndex;
    this.animationIndex++;

    // Once opening is complete, go back to close
    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
        this.action = "close";

        // If constructUnit has been set, add the new unit to the game
        if (this.constructUnit) {
            game.add(this.constructUnit);
            this.constructUnit = undefined;
        }
    }

    break;
```

Once the open animation is complete, we check whether the `constructUnit` property has been set, and if it has, we add the unit to the game before unsetting the variable.

Next we will implement a `showMessage()` method inside the game object, as shown in Listing 9-17.

Listing 9-17. The game Object's showMessage() Method

```
// Profile pictures for game characters
characters: {
    "system": {
        "name": "System Control",
        "image": "system.png"
    },
},

showMessage: function(from, message) {
    let callerpicture = document.getElementById("callerpicture");
    let gamemessages = document.getElementById("gamemessages");

    // If the message is from a defined game character, show profile picture
    let character = game.characters[from];

    if (character) {
```

```
        // Use the character's defined name
        from = character.name;

        if (character.image) {
            // Display the character image in the caller picture area
            callerpicture.innerHTML = "<img src=\"images/characters/" + character.image + "\"/>";

            // Remove the caller picture after six seconds
            setTimeout(function() {
                callerpicture.innerHTML = "";
            }, 6000);
        }
    }

    // Append message to messages pane and scroll to the bottom

    let messageHTML = "<span>" + from + ": </span>" + message + "<br>";

    gamemessages.innerHTML += messageHTML;
    gamemessages.scrollTop = gamemessages.scrollHeight;

},
```

We first define a `characters` object that contains the name and the image for the system character. Within the `showMessage()` method, we check whether we have a character image for the `from` parameter and, if so, display the image for six seconds. Next, we append the message to the `gamemessages div` and scroll to the bottom of the div.

Whenever the `showMessage()` method is called, it will display the message in the messages window and the picture in the sidebar, as shown in Figure 9-5.
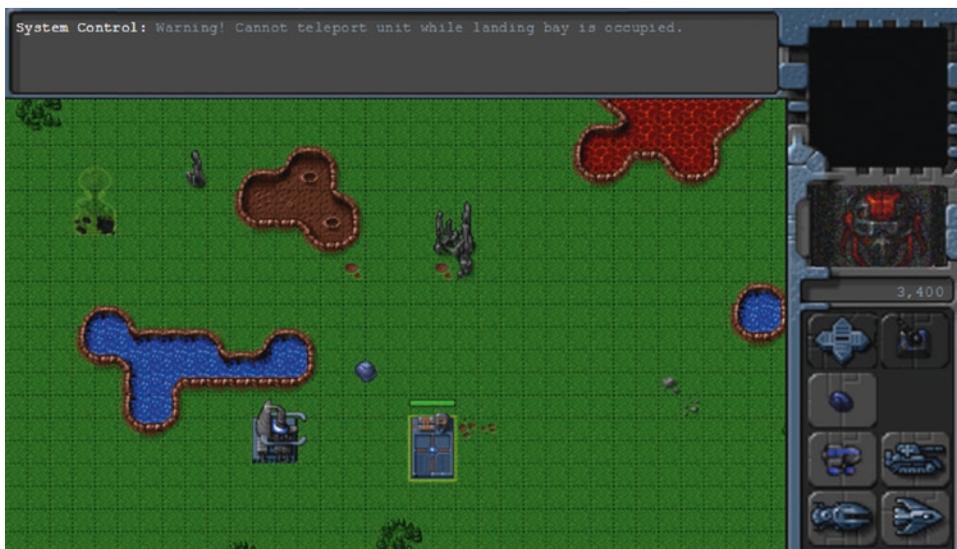
Figure 9-5. Displaying a system warning using showMessage()

We can use this mechanism to show the player dialogue from various game characters as we move the game story line forward. This will allow the single-player campaign to be more plot-driven and make the game much more engaging.

Finally, we will modify the vehicles and aircraft objects to implement the new teleport action.

We will start by adding a case for the teleport action right below the stand action inside the vehicles object's processActions() method, as shown in Listing 9-18.

Listing 9-18. Adding a Case for the Teleport Action Inside processActions() (vehicles.js)

```
case "teleport":

    this.imageList = this.spriteArray["stand-" + direction];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
    }

    // Initialize the brightness variable when unit is first teleported
    if (this.brightness === undefined) {
        this.brightness = 0.6;
    }

    this.brightness -= 0.05;

    // Once brightness gets to zero, clear brightness and just stand normally
    if (this.brightness <= 0) {
        this.brightness = undefined;
        this.action = "stand";
    }

    break;
```

We first set the imageOffset and the animationIndex just like we did for the default stand action. We then set a brightness variable to 0.6 and gradually reduce it to 0, at which point we switch the action state back to stand.

Next we will add a case for the teleport action right below the stand action inside the aircraft object's processActions() method, as shown in Listing 9-19.

Listing 9-19. Adding a Case for the Teleport Action Inside processActions() (aircraft.js)

```
case "teleport":

    this.imageList = this.spriteArray["stand-" + direction];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;

    if (this.animationIndex >= this.imageList.count) {
        this.animationIndex = 0;
    }
```

261

```
    // Initialize the brightness variable when unit is first teleported
    if (this.brightness === undefined) {
        this.brightness = 0.6;
    }

    this.brightness -= 0.05;

    // Once brightness gets to zero, clear brightness and just stand normally
    if (this.brightness <= 0) {
        this.brightness = undefined;
        this.action = "stand";
    }

    break;
```

Similar to what we did for vehicles, we set a brightness property, gradually drop it down to 0, and then set the action state to stand.

Finally, we will modify the baseItem object's default draw() method to use the brightness property, as shown in Listing 9-20.

Listing 9-20. Modifying the draw() Method to Handle Teleport Brightness (common.js)

```
// Default method for drawing an item
draw: function() {
    // Compute pixel coordinates on canvas for drawing item
    this.drawingX = (this.x * game.gridSize) - game.offsetX - this.pixelOffsetX;
    this.drawingY = (this.y * game.gridSize) - game.offsetY - this.pixelOffsetY;

    // Adjust based on interpolation factor
    if (this.canMove) {
        this.drawingX += this.lastMovementX * game.drawingInterpolationFactor *
        game.gridSize;
        this.drawingY += this.lastMovementY * game.drawingInterpolationFactor *
        game.gridSize;
    }

    if (this.selected) {
        this.drawSelection();
        this.drawLifeBar();
    }

    this.drawSprite();

    // Draw a glow around unit while teleporting in
    if (this.brightness) {
        let x = this.drawingX + this.pixelOffsetX;
        let y = this.drawingY + this.pixelOffsetY - (this.pixelShadowHeight ?
        this.pixelShadowHeight : 0);
```

```
        game.foregroundContext.beginPath();
        game.foregroundContext.arc(x, y, this.radius, 0, Math.PI * 2, false);
        game.foregroundContext.fillStyle = "rgba(255,255,255," + this.brightness + ")";
        game.foregroundContext.fill();
    }
},
```

Within the newly added code, we check whether the unit has a `brightness` property set, and if so, we draw a filled white circle on top of the unit with a fill alpha value based on the brightness. The position of the circle is automatically offset by `pixelShadowHeight` for aircraft. Since the `brightness` property's value drops from 0.6 to 0, the circle will gradually shift from being bright white to completely transparent.

If you run the game in the browser, you should now be able to select the starport and construct a vehicle or aircraft, as shown in Figure 9-6.

Figure 9-6. *Aircraft teleporting in at the starport*

The aircraft teleports right above the starport, inside a white glowing circle. You will notice that the sidebar buttons get disabled while the aircraft is being teleported in. Also, the cash balance decreases by the cost of the aircraft. When the player can no longer afford a unit, its button will automatically get disabled. Trying to construct a unit while the starport has another unit hovering over it will result in the system warning shown in Figure 9-5.

Now that we have implemented constructing vehicles and aircraft, it's time to implement constructing buildings at the base.

Constructing Buildings at the Base

We will start by implementing the sidebar object's constructBuilding() method, as shown in Listing 9-21.

Listing 9-21. Implementing the constructBuilding() Method (sidebar.js)

```
constructBuilding: function(details) {
    sidebar.deployBuilding = details;
},
```

Within the method, we set the sidebar.deployBuilding property to the details of the building to be constructed.

Next, we will modify the sidebar animate() method to handle deploying a building, as shown in Listing 9-22.

Listing 9-22. Modifying animate() to Handle Building Deployment (sidebar.js)

```
animate: function() {
    // Display the current cash balance value
    this.updateCash(game.cash[game.team]);

    // Enable buttons if player has sufficient cash and has the correct building selected
    this.enableSidebarButtons();

    // If sidebar is in deployBuilding mode, check whether building can be placed
    if (this.deployBuilding) {
        this.checkBuildingPlacement();
    }
},
```

If the deployBuilding variable has been set, we call a checkBuildingPlacement() method, which we will use to verify that a building can be placed at the current mouse location. Next we will add the checkBuildingPlacement() method to the sidebar object, as shown in Listing 9-23.

Listing 9-23. Adding the checkBuildingPlacement() Method (sidebar.js)

```
checkBuildingPlacement: function() {

    let name = sidebar.deployBuilding.name;
    let details = buildings.list[name];

    // Create a buildable grid to identify where building can be placed
    game.rebuildBuildableGrid();

    // Use buildableGrid to identify whether we can place the building
    let canDeployBuilding = true;
    let placementGrid = game.makeArrayCopy(details.buildableGrid);

    for (let y = placementGrid.length - 1; y >= 0; y--) {
        for (let x = placementGrid[y].length - 1; x >= 0; x--) {
```

```
        // If a tile needs to be buildable for the building
        if (placementGrid[y][x] === 1) {
            // Check whether the tile is inside the map and buildable
            if (mouse.gridY + y >= game.currentMap.mapGridHeight
            || mouse.gridX + x >= game.currentMap.mapGridWidth
            || game.currentMapBuildableGrid[mouse.gridY + y][mouse.gridX + x]) {
                // Otherwise mark tile as unbuildable
                canDeployBuilding = false;
                placementGrid[y][x] = 2;
            }
        }
    }
}

sidebar.placementGrid = placementGrid;
sidebar.canDeployBuilding = canDeployBuilding;

},
```

Within the method, we first call the game.rebuildBuildableGrid() method to create the game. currentMapBuildableGrid array.

Next, we set the sidebar.placementGrid variable using the buildableGrid property of the building being deployed. We then iterate through the placement grid to check whether it is possible to deploy the building at the current mouse location. If any of the squares on which the building will be placed are outside the map bounds or marked as unbuildable in the currentMapBuildableGrid array, we mark the corresponding square on the placementGrid array as unbuildable (with a value of 2) and set the canDeployBuilding flag to false.

Next we will implement the rebuildBuildableGrid() method inside the game object, as shown in Listing 9-24.

Listing 9-24. Creating the buildableGrid in the rebuildBuildableGrid() Method (game.js)

```
rebuildBuildableGrid: function() {
    game.currentMapBuildableGrid = game.makeArrayCopy(game.currentMapTerrainGrid);

    game.items.forEach(function(item) {

        if (item.type === "buildings" || item.type === "terrain") {
            // Mark all squares that the building uses as unbuildable
            for (let y = item.buildableGrid.length - 1; y >= 0; y--) {
                for (let x = item.buildableGrid[y].length - 1; x >= 0; x--) {
                    if (item.buildableGrid[y][x]) {
                        game.currentMapBuildableGrid[item.y + y][item.x + x] = 1;
                    }
                }
            }
        } else if (item.type === "vehicles") {
            // Mark all squares under or near the vehicle as unbuildable
            let radius = item.radius / game.gridSize;
            let x1 = Math.max(Math.floor(item.x - radius), 0);
            let x2 = Math.min(Math.floor(item.x + radius), game.currentMap.mapGridWidth - 1);
```

```
            let y1 = Math.max(Math.floor(item.y - radius), 0);
            let y2 = Math.min(Math.floor(item.y + radius), game.currentMap.mapGridHeight - 1);

            for (let x = x1; x <= x2; x++) {
                for (let y = y1; y <= y2; y++) {
                    game.currentMapBuildableGrid[y][x] = 1;
                }
            }
        }

    });
},
```

Similar to the currentMapPassableGrid array, the currentMapBuildableGrid array represents every square on the current map where it is possible for the player to build something. We will exclude any square on the map that has obstructed terrain, has been already built upon, or currently has vehicles on it.

We start by initializing the currentMapBuildableGrid array to the currentMapTerrainGrid array. We then mark out all squares under a building or terrain entity as unbuildable, just as we did when creating the passable array. Finally, we mark all grid squares next to a vehicle as unbuildable.

Next we will modify the draw() method of the mouse object to mark the grid location where the building will be deployed, as shown in Listing 9-25.

Listing 9-25. Drawing the Building Deploy Grid Under the Mouse Cursor (mouse.js)

```
buildableColor: "rgba(0,0,255,0.3)",
unbuildableColor: "rgba(255,0,0,0.3)",
draw: function() {
    // If the player is dragging and selecting, draw a white box to mark the selection area
    if (this.dragSelect) {
        let x = Math.min(this.gameX, this.dragX);
        let y = Math.min(this.gameY, this.dragY);

        let width = Math.abs(this.gameX - this.dragX);
        let height = Math.abs(this.gameY - this.dragY);

        game.foregroundContext.strokeStyle = "white";
        game.foregroundContext.strokeRect(x - game.offsetX, y - game.offsetY, width, height);
    }

    if (mouse.insideCanvas && sidebar.deployBuilding && sidebar.placementGrid) {
        let x = (this.gridX * game.gridSize) - game.offsetX;
        let y = (this.gridY * game.gridSize) - game.offsetY;

        for (let i = sidebar.placementGrid.length - 1; i >= 0; i--) {
            for (let j = sidebar.placementGrid[i].length - 1; j >= 0; j--) {
                let tile = sidebar.placementGrid[i][j];

                if (tile) {
                    game.foregroundContext.fillStyle = (tile === 1) ? this.buildableColor :
                    this.unbuildableColor;
                    game.foregroundContext.fillRect(x + j * game.gridSize, y + i *
                    game.gridSize, game.gridSize, game.gridSize);
```

```
                }
            }
        }
    }
},
```

We first check whether the `deployBuilding` and `placementGrid` variables have been set. We then iterate through all the squares of the placement grid, and draw either blue or red squares depending on whether we can place the building at that grid location or not. Based on our current convention, a value of 0 indicates the tile isn't needed to construct the building, a value of 1 means it is necessary and available and will be colored blue, and a value of 2 means it is necessary but currently unbuildable and will be colored red.

Finally, we will modify the `checkIfDragging()` method of the `mouse` object to ensure that drag selection isn't possible while a building is being deployed, as shown in Listing 9-26.

Listing 9-26. Disabling Drag Selection While Deploying a Building (mouse.js)

```
checkIfDragging: function() {
    if (mouse.buttonPressed && !sidebar.deployBuilding) {
        // If the mouse has been dragged more than threshold treat it as a drag
        if ((Math.abs(mouse.dragX - mouse.gameX) > mouse.dragSelectThreshold &&
        Math.abs(mouse.dragY - mouse.gameY) > mouse.dragSelectThreshold)) {
            mouse.dragSelect = true;
        }
    } else {
        mouse.dragSelect = false;
    }
},
```

If you run the game now, select the main base, and try to create a building, you should see the building deploy grid at the mouse location, as shown in Figure 9-7.

Figure 9-7. *Building deploy grid with red marking unbuildable squares*

Notice how the game tiles are either blue or red based on whether or not the building can be placed on them. Any time even one of the squares on the placement grid is red, the sidebar.canDeployBuilding flag will also be set to false, letting us know that we cannot place a building at the current mouse location.

Now that we can initiate building deploy mode, we will implement placing the building by left-clicking the mouse or canceling the mode by right-clicking the mouse. We will start by modifying the leftClick() and rightClick() methods of the mouse object, as shown in Listing 9-27.

Listing 9-27. Completing or Canceling deploy Mode Based on Click Type (mouse.js)

```
// Called whenever player completes a left-click on the game canvas
leftClick: function(shiftPressed) {
    if (sidebar.deployBuilding) {
        if (sidebar.canDeployBuilding) {
            sidebar.finishDeployingBuilding();
        } else {
            game.showMessage("system", "Warning! Cannot deploy building here.");
        }

        return;
    }

    let clickedItem = mouse.itemUnderMouse();

    if (clickedItem) {
        // Pressing Shift adds to existing selection. If shift is not pressed,
        clear existing selection
        if (!shiftPressed) {
            game.clearSelection();
        }

        game.selectItem(clickedItem, shiftPressed);
    }
},

// Called whenever player completes a right-click on the game canvas
rightClick: function() {
    // If the game is in deployBuilding mode, right-clicking will cancel deployBuilding mode
    if (sidebar.deployBuilding) {
        sidebar.cancelDeployingBuilding();

        return;
    }

    let clickedItem = mouse.itemUnderMouse();

    // Handle actions like attacking and movement of selected units
    if (clickedItem) { // Player right-clicked on something
        if (clickedItem.type !== "terrain") {
            if (clickedItem.team !== game.team) { // Player right-clicked on an enemy item
                let uids = [];
```

```
                    // Identify selected units from player's team that can attack
                    game.selectedItems.forEach(function(item) {
                        if (item.team === game.team && item.canAttack) {
                            uids.push(item.uid);
                        }
                    }, this);

                    // Command units to attack the clicked item
                    if (uids.length > 0) {
                        game.sendCommand(uids, { type: "attack", toUid: clickedItem.uid });
                    }
                } else { // Player right-clicked on a friendly item
                    let uids = [];

                    // Identify selected units from player's team that can move
                    game.selectedItems.forEach(function(item) {
                        if (item.team === game.team && item.canAttack && item.canMove) {
                            uids.push(item.uid);
                        }
                    }, this);

                    // Command units to guard the clicked item
                    if (uids.length > 0) {
                        game.sendCommand(uids, { type: "guard", toUid: clickedItem.uid });
                    }

                }
            } else if (clickedItem.name === "oilfield") { // Player right-clicked on an oilfield
                let uids = [];

                // Identify the first selected harvester (since only one can deploy at a time)
                for (let i = game.selectedItems.length - 1; i >= 0; i--) {
                    let item = game.selectedItems[i];

                    if (item.team === game.team && item.type === "vehicles" && item.name ===
                    "harvester") {
                        uids.push(item.uid);
                        break;
                    }
                }

                // Command it to deploy on the oil field
                if (uids.length > 0) {
                    game.sendCommand(uids, { type: "deploy", toUid: clickedItem.uid });
                }
            }
        } else { // Player right-clicked on the ground
            let uids = [];

            // Identify selected units from player's team that can move
            game.selectedItems.forEach(function(item) {
```

```
                if (item.team === game.team && item.canMove) {
                    uids.push(item.uid);
                }
            }, this);

            // Command units to move to the clicked location
            if (uids.length > 0) {
                game.sendCommand(uids, { type: "move", to: { x: mouse.gameX / game.gridSize,
                y: mouse.gameY / game.gridSize } });
            }
        }
    }
},
```

If the player left-clicks when in deploy mode, we check the canDeployBuilding variable and call sidebar.finishDeployingBuilding() if we can deploy the building, and we show a warning message using game.showMessage() if we cannot.

If the player right-clicks when in deploy mode, we call the sidebar.cancelDeployingBuilding() method.

Next we will implement these two new methods, finishDeployingBuilding() and cancelDeployingBuilding(), inside the sidebar object, as shown in Listing 9-28.

Listing 9-28. finishDeployingBuilding() and cancelDeployingBuilding() (sidebar.js)

```
cancelDeployingBuilding: function() {
    sidebar.deployBuilding = undefined;
    sidebar.placementGrid = undefined;
    sidebar.canDeployBuilding = false;
},

finishDeployingBuilding: function() {
    // Search for a selected base which can construct the unit
    let base;

    for (let i = game.selectedItems.length - 1; i >= 0; i--) {
        let item = game.selectedItems[i];

        if (item.name === "base" && item.team === game.team
            && item.lifeCode === "healthy" && item.action === "stand") {

            base = item;
            break;
        }
    }

    // If an eligible base is found, tell it to make the unit
    if (base) {
        let name = sidebar.deployBuilding.name;
        let details = {
            name: name,
            type: "buildings",
            x: mouse.gridX,
            y: mouse.gridY
        };
```

```
        game.sendCommand([base.uid], { type: "construct-building", details: details });
    }

    // Clear deploy building variables
    sidebar.cancelDeployingBuilding();

},
```

The `cancelDeployingBuilding()` method merely clears out all the building deployment–related variables. The `finishDeployingBuilding()` method selects the first available base and then uses the `game.sendCommand()` method to send it the `construct-building` order. This is very similar to how the `constructInStarport()` method sends a command to a starport.

Next, we will create a `processOrder()` method for the base building that implements the `construct-building` order. We will add this method inside the base definition, as shown in Listing 9-29.

Listing 9-29. Implementing processOrder() Inside the Base Definition (buildings.js)

```
processOrders: function() {
    switch (this.orders.type) {
        case "construct-building":
            this.action = "construct";
            this.animationIndex = 0;

            // Teleport in building and subtract the cost from player cash
            var itemDetails = this.orders.details;

            itemDetails.team = this.team;
            itemDetails.action = "teleport";

            var item = game.add(itemDetails);

            game.cash[this.team] -= item.cost;

            this.orders = { type: "stand" };

            break;
    }
}
```

We first set the base entity's action state to `construct`. Next, we add the building to the game with an action state of `teleport`. Finally, we subtract the cost of the building from the cash balance and set the base entity's `orders` property back to `stand`.

If you run the game now and try to deploy the building by left-clicking at a valid location on the map, the building should get teleported in at that location, as shown in Figure 9-8.
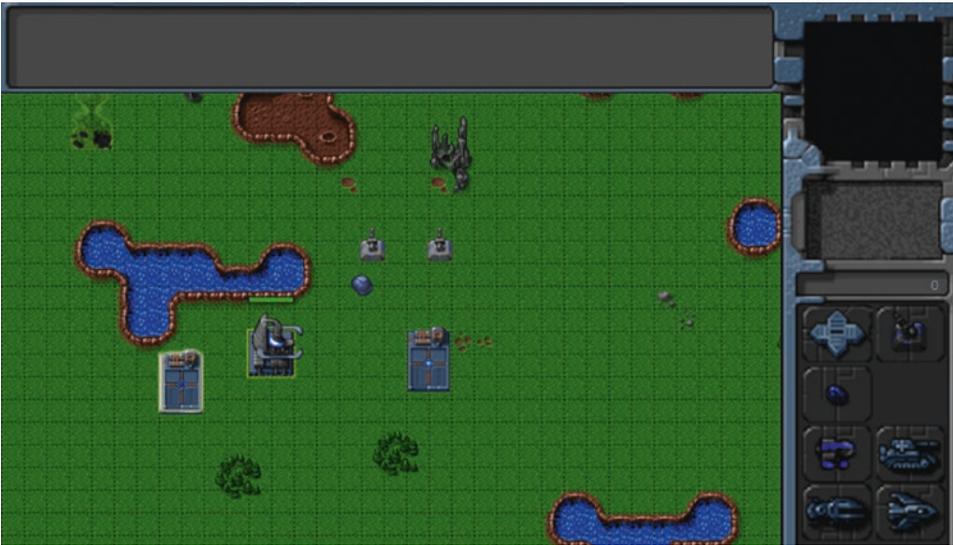
Figure 9-8. *Deployed building gets teleported in*

You will notice that the cash balance decreases by the cost of the building. When the player can no longer afford a building, its button will automatically get disabled. Also, if you try to deploy the building at an invalid location, you will see a system warning message telling you that the building cannot be deployed at that location.

We can now construct both units and buildings in our game. The last thing we will implement in this chapter is ending levels based on triggered events.

Ending a Level

Whenever players complete the objectives for a level successfully, we will show them a message box notifying them and then load the next level. If a player fails a mission, we will give the player the option of replaying the current level or leaving the single-player campaign.

We will check for the success and failure criteria by implementing a system of triggered events within our game. We will use this same event system to script story-based events in later chapters.

The first thing we will do is implement a message dialog box.

Implementing the Message Dialog Box

The message box will be a modal dialog box with either only an Okay button or both Okay and Cancel buttons.

We will start by adding the HTML markup for the message box screen to the game container inside index.html, as shown in Listing 9-30.

Listing 9-30. Adding HTML Markup for Message Box (index.html)

```
<div id="gamecontainer">
    <div id="gamestartscreen" class="gamelayer">
        <span class="game-title">LAST<br>COLONY</span>
```

```
        <span class="game-option" onclick = "singleplayer.start();">Campaign</span>
        <span class="game-option" onclick = "multiplayer.start();">Multiplayer</span>
    </div>

    <div id="missionbriefingscreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-briefing.png" class="right-panel"
         draggable="false">
        <input type="button" id="entermission" onclick = "singleplayer.play();">
        <input type="button" id="exitmission" onclick = "singleplayer.exit();">
        <div id="missionbriefing"></div>
    </div>

    <div id="gameinterfacescreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-game.png" class="right-panel"
         draggable="false">
        <div id="gamemessages"></div>
        <div id="callerpicture"></div>
        <div id="cash"></div>
        <div id="sidebarbuttons">
            <input type="button" id="starport" title = "Starport">
            <input type="button" id="ground-turret" title = "Turret">
            <input type="button" id="harvester" title = "Harvester">
            <input type="button" id="scout-tank" title = "Scout Tank">
            <input type="button" id="heavy-tank" title = "Heavy Tank">
            <input type="button" id="chopper" title = "Copter">
            <input type="button" id="wraith" title = "Wraith">
        </div>
        <canvas id="gamebackgroundcanvas"></canvas>
        <canvas id="gameforegroundcanvas"></canvas>
    </div>

    <div id="messageboxscreen" class="gamelayer">
        <div id="messagebox">
            <span id="messageboxtext"></span>
            <input type="button" id="messageboxok" onclick="game.messageBoxOK();">
            <input type="button" id="messageboxcancel" onclick="game.messageBoxCancel();">
        </div>
    </div>

    <div id="loadingscreen" class="gamelayer">
        <div id="loadingmessage"></div>
    </div>
</div>
```

Next, we will add the styles for the message box to `styles.css`, as shown in Listing 9-31.

Listing 9-31. Styles for Message Box (styles.css)

```css
/* Message Box Screen */

#messageboxscreen {
    background: rgba(0, 0, 0, 0.7);
    z-index: 10;
}

#messagebox {

    /* Center the message box within screen */
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    transform-origin: center center;

    width: 296px;
    height: 178px;

    background: url("images/screens/messagebox.png") no-repeat center;
    color: rgb(130, 150, 162);
    overflow: hidden;

    font-size: 13px;
    font-family: "Courier New", Courier, monospace;
}

#messagebox span {
    position: absolute;

    top: 30px;
    left: 50px;
    width: 200px;
    height: 100px;
}

#messageboxok {
    position: absolute;

    top: 128px;
    left: 11px;
    width: 74px;
    height: 26px;

    background-position: -4px -122px;
}
```

```css
#messageboxok:active, #messageboxok:disabled {
    background-position: -82px -122px;
}

#messageboxcancel {
    position: absolute;

    left: 197px;
    top: 130px;
    width: 73px;
    height: 24px;

    background-position: -4px -154px;
}

#messageboxcancel:active, #messageboxcancel:disabled {
    background-position: -82px -154px;
}
```

Finally, we will add some methods for implementing the message box to the game object, as shown in Listing 9-32.

Listing 9-32. Adding Message Box Methods to the game Object (game.js)

```javascript
/* Message Box related code*/

messageBoxOkCallback: undefined,
messageBoxCancelCallback: undefined,
showMessageBox: function(message, onOK, onCancel) {
    // Set message box text
    let messageBoxText = document.getElementById("messageboxtext");

    messageBoxText.innerHTML = message.replace(/\n/g, "<br><br>");

    // Set message box onOK handler
    if (typeof onOK === "function") {
        game.messageBoxOkCallback = onOK;
    } else {
        game.messageBoxOkCallback = undefined;
    }

    // Set onCancel handler if defined and show Cancel button
    let cancelButton = document.getElementById("messageboxcancel");

    if (typeof onCancel === "function") {
        game.messageBoxCancelCallback = onCancel;
        // Show the Cancel button
        cancelButton.style.display = "";
    } else {
        game.messageBoxCancelCallback = undefined;
        // Hide the Cancel button
        cancelButton.style.display = "none";
    }
```

275

```
    // Display the message box and wait for user to click a button
    game.showScreen("messageboxscreen");
},

messageBoxOK: function() {
    game.hideScreen("messageboxscreen");
    if (typeof game.messageBoxOkCallback === "function") {
        game.messageBoxOkCallback();
    }
},

messageBoxCancel: function() {
    game.hideScreen("messageboxscreen");
    if (typeof game.messageBoxCancelCallback === "function") {
        game.messageBoxCancelCallback();
    }
},
```

The showMessageBox() method first sets the message inside the messageboxtext element. Next it saves the onOK and onCancel callback method parameters into the messageBoxOkCallback and messageBoxCancelCallback variables. It shows or hides the Cancel button based on whether a cancel callback method parameter was passed. Finally, it shows the messageboxscreen layer.

The messageBoxOK() and messageBoxCancel() methods hide the messageboxscreen layer and then call their respective callback methods if they have been set.

When the showMessageBox() method is called without specifying any callback methods, it will display the message box on a darkened screen with only an Okay button, as shown in Figure 9-9.

Figure 9-9. *A sample message shown in the message box*

Now that the code for the message box is in place, we will implement our game triggers.

Implementing Triggers

Our game will use two types of triggers:

- Timed triggers will execute an action after a specified time. They may also keep repeating at regular intervals.

- Conditional triggers will execute an action when a specified condition comes true.

We will start by adding a triggers array within our level inside the levels object, as shown in Listing 9-33.

Listing 9-33. Adding Triggers into the Level (levels.js)

```
/* Conditional and Timed Trigger Events */
"triggers": [
    /* Timed Events*/
    {
        "type": "timed",
        "time": 1000,
        "action": function() {
            game.showMessage("system", "You have 20 seconds left.\nGet the harvester near
            the oil field.");
        }
    },
    {
        "type": "timed",
        "time": 21000,
        "action": function() {
            singleplayer.endLevel(false);
        }
    },
    /* Conditional Event */
    {
        "type": "conditional",
        "condition": function() {
            let transport = game.getItemByUid(-1);

            // True if transport has reached top-left quadrant near oil field
            return (transport.x < 7 && transport.y < 7);
        },
        "action": function() {
            singleplayer.endLevel(true);
        }
    }
],
```

All the triggers have a type property and an action method. We have defined three triggers within the array.

The first trigger is a timed trigger with a time set to 1 second (or 1000 milliseconds). In its action parameter, we call game.showMessage() and tell the player that he has 20 seconds to move the harvester near the oil field.

The second trigger, which is timed for 20 seconds later, calls the singleplayer.endLevel() method with a parameter of false, indicating the mission failed.

The final trigger is a conditional trigger. The condition method returns `true` when the transport is within the top-left quadrant of the map with x and y coordinates less than 7. When this condition is triggered, the `action` method calls the `singleplayer.endLevel()` method with a parameter of `true` indicating the mission was successfully completed.

Next we will implement the `endLevel()` method inside the `singleplayer` object, as shown in Listing 9-34.

Listing 9-34. Implementing the singleplayer endLevel() Method (singleplayer.js)

```
endLevel: function(success) {
    clearInterval(game.animationInterval);
    game.end();

    if (success) {
        let moreLevels = (singleplayer.currentLevel < levels.singleplayer.length - 1);

        if (moreLevels) {
            game.showMessageBox("Mission Accomplished.", function() {
                game.hideScreens();
                // Start the next level
                singleplayer.currentLevel++;
                singleplayer.initLevel();
            });
        } else {
            game.showMessageBox("Mission Accomplished.\nThis was the last mission in the
            campaign.\nThank You for playing.", function() {
                game.hideScreens();
                // Return to the main menu
                game.showScreen("gamestartscreen");
            });
        }
    } else {
        game.showMessageBox("Mission Failed.\nTry again?", function() {
            game.hideScreens();
            // Restart the current level
            singleplayer.initLevel();
        }, function() {
            game.hideScreens();
            // Return to the main menu
            game.showScreen("gamestartscreen");
        });
    }
}
```

We first clear the `game.animationInterval` timer that calls the `game.animationLoop()` method. Next we call the `game.end()` method.

If the level was completed successfully, we check whether there are more levels in the map. If so, we notify the player that the mission was successful in a message box and then start the next level when the player clicks the Okay button. If there are no more levels, we notify the player but go back to the game starting menu when the player clicks Okay.

If the level was not completed successfully, we ask the player if he wants to try again. If the player clicks Okay, we restart the current level. If instead the player clicks Cancel, we return to the game starting menu.

278

Next, we will add a few trigger-related methods to the game object, as shown in Listing 9-35.

Listing 9-35. Adding Trigger-Related Methods to the game Object (game.js)

```
/* Methods for handling triggered events within the game */

initTrigger: function(trigger) {
    if (trigger.type === "timed") {
        trigger.timeout = setTimeout(function() {
            game.runTrigger(trigger);
        }, trigger.time);
    } else if (trigger.type === "conditional") {
        trigger.interval = setInterval(function() {
            game.runTrigger(trigger);
        }, 1000);
    }
},

runTrigger: function(trigger) {
    if (trigger.type === "timed") {
        // Reinitialize the trigger based on repeat settings
        if (trigger.repeat) {
            game.initTrigger(trigger);
        }
        // Call the trigger action
        trigger.action(trigger);
    } else if (trigger.type === "conditional") {
        //Check if the condition has been satisfied
        if (trigger.condition()) {
            // Clear the trigger
            game.clearTrigger(trigger);
            // Call the trigger action
            trigger.action(trigger);
        }
    }
},

clearTrigger: function(trigger) {
    if (trigger.timeout !== undefined) {
        clearTimeout(trigger.timeout);
        trigger.timeout = undefined;
    }

    if (trigger.interval !== undefined) {
        clearInterval(trigger.interval);
        trigger.interval = undefined;
    }
},

end: function() {
    // Clear any game triggers
    if (game.currentLevel.triggers) {
```

```
        for (var i = game.currentLevel.triggers.length - 1; i >= 0; i--) {
            game.clearTrigger(game.currentLevel.triggers[i]);
        }
    }

    game.running = false;
},
```

The first method we implement is initTrigger(), which takes a trigger as a parameter, and then attempts to initialize it. We check whether the trigger is timed or conditional. For timed triggers, we call the runTrigger() method after the timeout specified in the time parameter. For conditional triggers, we call the runTrigger() method every second. In both cases we save the timeout or interval within the trigger object so we can access it later.

In the runTrigger() method, we check whether the trigger is timed or conditional. For timed triggers with the repeat parameter specified, we call initTrigger() again. We then execute the trigger action. For conditional triggers, we check whether the condition is true. If so, we clear the trigger using the clearTrigger() method and then execute the action.

The clearTrigger() method just clears the timeout or interval for the trigger.

Finally, the end() method clears all the triggers for a level and sets the game.running variable to false.

The last change we will make is to the game object's start() method, as shown in Listing 9-36.

Listing 9-36. Initializing the Triggers Inside the start() Method (game.js)

```
 start: function() {
    // Display the game interface
    game.hideScreens();
    game.showScreen("gameinterfacescreen");

    game.running = true;
    game.refreshBackground = true;
    game.canvasResized = true;

    game.drawingLoop();

    // Clear the game messages area
    let gamemessages = document.getElementById("gamemessages");

    gamemessages.innerHTML = "";

    // Initialize all game triggers
    game.currentLevel.triggers.forEach(function(trigger) {
        game.initTrigger(trigger);
    });
},
```

In the newly added code, we first clear the gamemessages container when we start the level. This will ensure messages from earlier attempts at the level are cleared when we restart a level.

Next we iterate through the current level's triggers array and call initTrigger() for each trigger. With this final change, everything that we need to use our triggered events is in place.

If we run the game now, we should get a message asking us to take the harvester near the oil field within 20 seconds. If we do not do so in time, we will see a message box indicating that the mission failed, as shown in Figure 9-10.
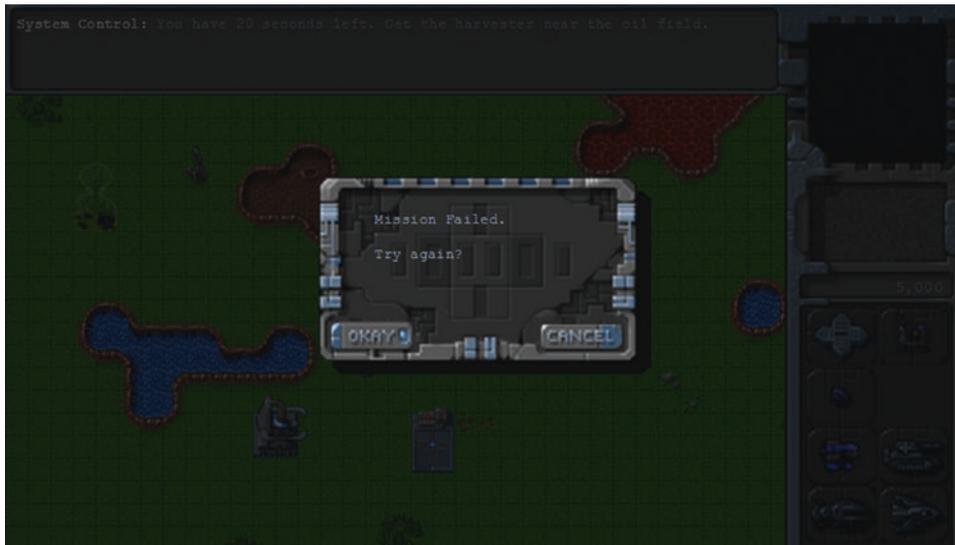
Figure 9-10. Message shown when the mission fails

If we click the Okay button, the level will restart, and we will be returned to the mission briefing screen. If we click the Cancel button instead, we will be taken back to the main menu.

If we move the harvester toward the oil field and reach there before the 20 seconds are up, we will see a message box indicating that the mission was accomplished, as shown in Figure 9-11.
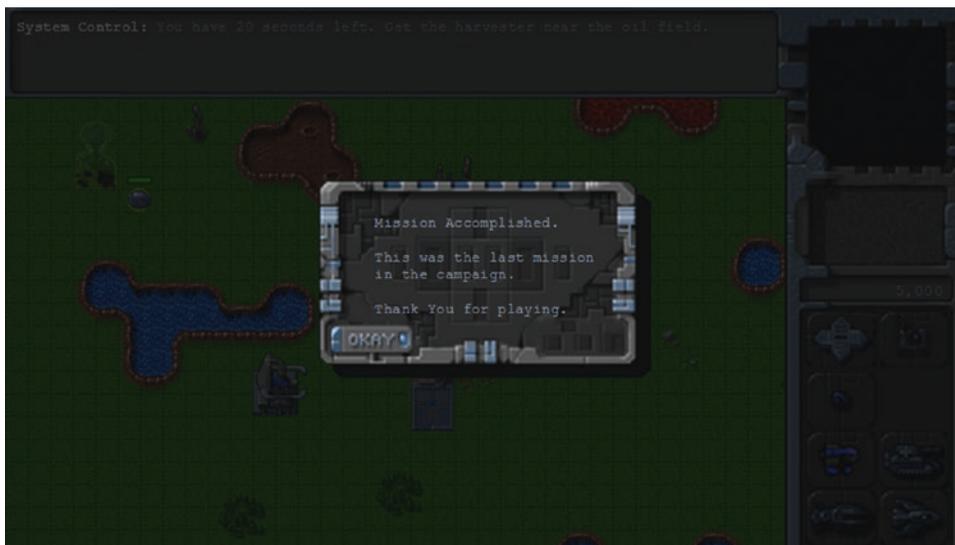
Figure 9-11. Message shown when the mission is accomplished

Since this is the only mission in our campaign, we will see the campaign ending message box. When we click Okay, we will be taken back to the main menu.

Summary

We accomplished a lot in this chapter. We started by creating a basic economy where we could earn cash by harvesting. We then implemented the ability to purchase units at the starport and buildings at the base using the buttons on the sidebar.

We developed a messaging system and a message dialog box to communicate with the player. We then built a system for trigger-based actions that handled both timed and conditional triggers. Finally, we used these triggers to create a simple mission objective and criteria for succeeding or failing the mission. Even though it is a fairly simple mission, we now have the infrastructure in place to build much more complex levels.

In the next chapter, we will handle another important component of our game: combat. We will implement different attack-based order states for both units and turrets. We will use a combination of triggers and order states to make the units behave intelligently during combat. Finally, we will look at implementing a fog of war so that units cannot see or attack unexplored territory.

■ ■ ■

Adding Weapons and Combat

Over the past few chapters, we built the basic framework for our game; added entities such as vehicles, aircraft, and buildings; implemented unit movement; and created a simple economy using the sidebar. We now have a game where we can start the level, earn money, purchase buildings and units, and move these units around to achieve simple goals.

In this chapter, we will implement weapons for vehicles, aircraft, and turrets. We will add the ability to process combat-based orders such as attacking, guarding, patrolling, and hunting to allow the units to fight in an intelligent way. Finally, we will implement a fog of war that limits visibility on the map, allowing for interesting strategies such as sneak attacks and ambushes.

Let's get started. We will use the code from Chapter 9 as a starting point.

Implementing the Combat System

Our game will have a fairly simple combat system. All units and turrets will have their own weapon and bullet type defined. When attacking an enemy, units will first get within range, turn toward the target, and then fire a bullet at them. Once the unit fires a bullet, it will wait until its weapon has reloaded before it fires again.

The bullet itself will be a separate game entity with its own animation logic. When fired, the bullet will fly toward its target and explode once it reaches its destination.

The first thing we will do is add bullets to our game.

Adding Bullets

We will start by defining a new `bullets` object inside `bullets.js`, as shown in Listing 10-11.

Listing 10-1. Defining the bullets Object (bullets.js)

```
var bullets = {
    list: {
        "fireball": {
            name: "fireball",
            speed: 60,
            reloadTime: 30,
            range: 8,
            damage: 10,
            spriteImages: [
                { name: "fly", count: 1, directions: 8 },
                { name: "explode", count: 7 }
            ],
        },
```

```
    "heatseeker": {
        name: "heatseeker",
        reloadTime: 40,
        speed: 25,
        range: 9,
        damage: 20,
        turnSpeed: 2,
        spriteImages: [
            { name: "fly", count: 1, directions: 8 },
            { name: "explode", count: 7 }
        ],
    },
    "cannon-ball": {
        name: "cannon-ball",
        reloadTime: 40,
        speed: 25,
        damage: 10,
        range: 6,
        spriteImages: [
            { name: "fly", count: 1, directions: 8 },
            { name: "explode", count: 7 }
        ],
    },
    "bullet": {
        name: "bullet",
        damage: 5,
        speed: 50,
        range: 5,
        reloadTime: 20,
        spriteImages: [
            { name: "fly", count: 1, directions: 8 },
            { name: "explode", count: 3 }
        ],
    }
},

defaults: {
    type: "bullets",
    canMove: true,

    distanceTravelled: 0,
    directions: 8,

    pixelWidth: 10,
    pixelHeight: 11,
    pixelOffsetX: 5,
    pixelOffsetY: 5,

    radius: 6,
```

```
    action: "fly",
    selected: false,
    selectable: false,

    orders: { type: "fire" },

    // How slow should bullet move while turning
    speedAdjustmentWhileTurningFactor: 1,

    moveTo: function(destination) {
        // Weapons like the heatseeker can turn slowly toward target while moving
        if (this.turnSpeed) {
            // Find out where we need to turn to get to destination
            var newDirection = this.findAngleForFiring(destination);

            // Turn toward new direction if necessary
            this.turnTo(newDirection);
        }

        // Calculate maximum distance that bullet can move per animation cycle
        let maximumMovement = this.speed * this.speedAdjustmentFactor;
        let movement = maximumMovement;

        // Calculate x and y components of the movement
        let angleRadians = -(this.direction / this.directions) * 2 * Math.PI;

        this.lastMovementX = -(movement * Math.sin(angleRadians));
        this.lastMovementY = -(movement * Math.cos(angleRadians));

        this.x = this.x + this.lastMovementX;
        this.y = this.y + this.lastMovementY;

        // Track distance travelled by bullet
        this.distanceTravelled += movement;
    },

    reachedTarget: function() {
        var item = this.target;

        if (item.type === "buildings") {
            return (item.x <= this.x && item.x >= this.x - item.baseWidth / game.gridSize
            && item.y <= this.y && item.y >= this.y - item.baseHeight / game.gridSize);
        } else if (item.type === "aircraft") {
            return (Math.pow(item.x - this.x, 2) + Math.pow(item.y - (this.y + item.
            pixelShadowHeight / game.gridSize), 2) < Math.pow((item.radius) / game.
            gridSize, 2));
        } else {
            return (Math.pow(item.x - this.x, 2) + Math.pow(item.y - this.y, 2) < Math.
            pow((item.radius) / game.gridSize, 2));
        }
    },
```

```
        processOrders: function() {
            this.lastMovementX = 0;
            this.lastMovementY = 0;
            switch (this.orders.type) {
                case "fire":
                    // Move toward destination and stop when close by or if travelled past range
                    var reachedTarget = false;

                    if (this.distanceTravelled > this.range
                        || (reachedTarget = this.reachedTarget())) {
                        if (reachedTarget) {
                            // Bullet damages target and then explodes
                            this.target.life -= this.damage;

                            this.orders = { type: "explode" };
                            this.action = "explode";
                            this.animationIndex = 0;
                        } else {
                            // Bullet fizzles out without hitting target
                            game.remove(this);
                        }
                    } else {
                        this.moveTo(this.target);
                    }
                    break;
            }
        },

        animate: function() {
            // No need to do a health check for terrain. Just call processActions
            this.processActions();
        },

        processActions: function() {
            let direction = Math.round(this.direction) % this.directions;

            switch (this.action) {
                case "fly":
                    this.imageList = this.spriteArray["fly-" + direction];
                    this.imageOffset = this.imageList.offset;
                    break;

                case "explode":
                    this.imageList = this.spriteArray["explode"];
                    this.imageOffset = this.imageList.offset + this.animationIndex;
                    this.animationIndex++;

                    if (this.animationIndex >= this.imageList.count) {
                        // Bullet explodes completely and then disappears
                        game.remove(this);
                    }
```

```
                    break;
            }
        },

        drawSprite: function() {
            let x = this.drawingX;
            let y = this.drawingY;

            let colorOffset = 0;  // No team-based colors for bullets

            game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset * this.
            pixelWidth, colorOffset, this.pixelWidth, this.pixelHeight, x, y, this.
            pixelWidth, this.pixelHeight);
        },
    },

    load: loadItem,
    add: addItem,
};
```

The bullets object follows the same pattern as all the other game entities. We start by defining a list of four bullet types: fireball, heatseeker, cannon-ball, and bullet. Each of the bullets has a common set of properties:

- speed: The speed at which the bullet travels

- reloadTime: The number of animation cycles after firing before the bullet can be fired again

- damage: The amount of damage to the target when the bullet explodes

- range: The maximum range that a bullet will fly before it loses momentum

The bullets also have two animation sequences defined: fly and explode. The fly state has eight directions similar to vehicles and aircraft. The explode state has only one direction but has multiple frames.

We then define a default moveTo() method, which is similar to the aircraft moveTo() method. Within this method we first check whether the bullet can turn and, if so, gently turn the bullet toward its destination using the findAngleForFiring() method to calculate the angle toward the center of the target. Next, we move the bullet forward along its current direction and update the bullet's distanceTravelled property.

Next we define a reachedTarget() method that checks whether the bullet has reached its target. We check whether the bullet's coordinates are inside the base area for buildings and within the item radius for vehicles and aircraft. If so, we return a value of true.

Within the processOrders() method, we implement the fire order. We check whether the bullet has either reached its target or traveled beyond its range. If not, we continue to move the bullet toward the target.

If the bullet travels beyond its range without hitting the target, we remove it from the game. If the bullet reaches its target, we first set the bullet's order and animation state to explode and reduce the life of its target by the damage amount.

Since bullets do not require a health check, we use a simpler animate() method that just forwards the call to processActions(), just as we did for the terrain object. Within the processActions() method, we remove the bullet once the explode animation sequence completes.

Now that we have defined the bullets object, we will add a reference to bullets.js inside the head section of index.html, as shown in Listing 10-2.

Listing 10-2. *Adding a Reference to the bullets Object (index.html)*

```
<script src="js/bullets.js" type="text/javascript"></script>
```

We will also add the `findAngleForFiring()` method to the `defaultItem` object inside `common.js`, as shown in Listing 10-3.

Listing 10-3. *Defining the findAngleForFiring() Method (common.js)*

```
// Finds the angle from center of source to a target in terms of a direction
(0 <= angle < directions)
findAngleForFiring: function(target) {
    var dy = target.y - this.y;
    var dx = target.x - this.x;

    // Adjust dx and dy to point toward center of target
    if (target.type === "buildings") {
        dy += target.baseWidth / 2 / game.gridSize;
        dx += target.baseHeight / 2 / game.gridSize;
    } else if (target.type === "aircraft") {
        dy -= target.pixelShadowHeight / game.gridSize;
    }

    // Adjust dx and dy to start from center of source
    if (this.type === "buildings") {
        dy -= this.baseWidth / 2 / game.gridSize;
        dx -= this.baseHeight / 2 / game.gridSize;
    } else if (this.type === "aircraft") {
        dy += this.pixelShadowHeight / game.gridSize;
    }

    // Convert arctan to value between (0 - directions)
    var angle = this.directions / 2 - (Math.atan2(dx, dy) * this.directions / (2 * Math.PI));

    angle = (angle + this.directions) % this.directions;

    return angle;
}
```

The `findAngleForFiring()` method is similar to the `findAngle()` method except we adjust the values of the dy and dx variables to point to the center of the source and target. For buildings, we adjust dx and dy using the `baseWidth` and `baseHeight` properties, and for aircraft we adjust dy by the `pixelShadowHeight` property. This way, bullets can be aimed at the center of the target.

We will also modify the `loadItem()` method inside `common.js` to load the bullet for an item when the item loads, as shown in Listing 10-4.

Listing 10-4. Loading the Bullets When Loading the Item (common.js)

```javascript
// The default load() method used by all our game entities
function loadItem(name) {
    var item = this.list[name];

    // If the item sprite array has already been loaded then no need to do it again
    if (item.spriteArray) {
        return;
    }

    item.spriteSheet = loader.loadImage("images/" + this.defaults.type + "/" + name + ".png");
    item.spriteArray = [];
    item.spriteCount = 0;

    item.spriteImages.forEach(function(spriteImage) {

        let constructImageCount = spriteImage.count;
        let constructDirectionCount = spriteImage.directions;

        if (constructDirectionCount) {
            // If the spriteImage has directions defined, store sprites for each direction
            in spriteArray
            for (let i = 0; i < constructDirectionCount; i++) {
                let constructImageName = spriteImage.name + "-" + i;

                item.spriteArray[constructImageName] = {
                    name: constructImageName,
                    count: constructImageCount,
                    offset: item.spriteCount
                };
                item.spriteCount += constructImageCount;
            }
        } else {
            // If the spriteImage has no directions, store just the name and image count in
            spriteArray
            let constructImageName = spriteImage.name;

            item.spriteArray[constructImageName] = {
                name: constructImageName,
                count: constructImageCount,
                offset: item.spriteCount
            };

            item.spriteCount += constructImageCount;
        }
    });

    // Load the weapon if item has one
    if (item.weaponType) {
        bullets.load(item.weaponType);
    }
}
```

When loading an item, we check whether it has a `weaponType` property defined and, if so, load the bullet for the weapon using the `bullets.load()` method. All entities that are capable of attacking will have a `weaponType` property.

The next change we will make is to modify the game object's `drawingLoop()` method to draw exploding bullets on top of all other items in the game. The updated `drawingLoop()` method will look like Listing 10-5.

Listing 10-5. Modifying drawingLoop() to Draw Bullets Above Other Items (game.js)

```
drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();

    // Check the time since the game was animated and calculate a linear interpolation
    factor (-1 to 0)
    game.lastDrawTime = Date.now();
    if (game.lastAnimationTime) {
        game.drawingInterpolationFactor = (game.lastDrawTime - game.lastAnimationTime) /
        game.animationTimeout - 1;

        // No point interpolating beyond the next animation loop...
        if (game.drawingInterpolationFactor > 0) {
            game.drawingInterpolationFactor = 0;
        }
    } else {
        game.drawingInterpolationFactor = -1;
    }

    // Draw the background whenever necessary
    game.drawBackground();

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0, 0, game.canvasWidth, game.canvasHeight);

    // Start drawing the foreground elements
    game.sortedItems.forEach(function(item) {
        item.draw();
    });

    // Draw exploding bullets on top of everything else
    game.bullets.forEach(function(bullet) {
        if (bullet.action === "explode") {
            bullet.draw();
        }
    });

    // Draw the mouse
    mouse.draw();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

When drawing the items, we first draw all the items including the bullets. We then iterate through all the bullets and draw the bullets that have an action of explode. This way, explosions will be drawn on top of all other items, and will always be clearly visible in the game.

Finally, we will modify the game object's resetArrays() method to also reset the game.bullets[] array, as shown in Listing 10-6.

Listing 10-6. Resetting the bullets Array Inside resetArrays() (game.js)

```
resetArrays: function() {
    // Count items added in game, to assign them a unique id
    game.counter = 0;

    // Track all the items currently in the game
    game.items = [];
    game.buildings = [];
    game.vehicles = [];
    game.aircraft = [];
    game.terrain = [];

    // Track items that have been selected by the player
    game.selectedItems = [];

    game.bullets = [];
},
```

Now that we have implemented the bullets object, it's time to implement combat-based orders for the turrets, vehicles, and aircraft.

Combat-Based Orders for Turrets

Ground turrets can fire cannonballs at any ground-based threat. When in guard or attack mode, they will search for a valid target that is in sight, aim the turret toward the target, and fire bullets until the target is either destroyed or out of range.

We will start by implementing the processOrders() method for the ground-turret object inside buildings.js, as shown in Listing 10-7.

Listing 10-7. Modifying ground-turret Object to Implement Attack (buildings.js)

```
turnSpeed: 1,
processOrders: function() {
    if (this.reloadTimeLeft) {
        this.reloadTimeLeft--;
    }

    // Damaged turret cannot do anything
    if (this.lifeCode !== "healthy") {
        return;
    }

    var targets;
```

```
    switch (this.orders.type) {
        case "guard":
            targets = this.findTargetsInSight();

            if (targets.length > 0) {
                this.orders = { type: "attack", to: targets[0] };
            }

            break;

        case "attack":
            // If the current target is no longer valid, go back to guarding
            if (!this.isValidTarget(this.orders.to) || !this.isTargetInSight(this.orders.to)) {
                this.orders = { type: "guard" };
                break;
            }

            var targetDirection = this.findAngleForFiring(this.orders.to);

            // Turn toward target direction if necessary
            this.turnTo(targetDirection);

            // Check if turret has finished turning
            if (!this.turning) {
                // Check if weapon has finished reloading
                if (!this.reloadTimeLeft) {

                    // Calculate the starting position of the bullet
                    let angleRadians = -(targetDirection / this.directions) * 2 * Math.PI ;
                    let bulletX = this.x + 0.5 - (1 * Math.sin(angleRadians));
                    let bulletY = this.y + 0.5 - (1 * Math.cos(angleRadians));

                    // Fire the bullet
                    game.add({ name: this.weaponType, type: "bullets", x: bulletX, y:
                    bulletY, direction: targetDirection, target: this.orders.to });

                    // Set the weapon reloading cooldown
                    this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                }
            }

            break;
    }
}
```

Within the processOrders() method, we decrease the value of the reloadTimeLeft property if the property is defined and is greater than 0. If the turret lifeCode is not healthy (it is damaged or dead), we do nothing and exit.

Next, we define the behavior for both the guard and attack orders. In guard mode, we use the findTargetsInSight() method to find any visible targets and, if we find any, attack the first one in the list.

In `attack` mode, if the current target of the turret is undefined, dead, or out of sight, we go back to `guard` mode.

If the turret does have a valid target, we turn the turret toward the target. Once the turret is facing the target and `reloadTimeLeft` is 0, we fire a bullet by adding it to the game using the `game.add()` method and reset the turret's `reloadTimeLeft` property to the bullet's reload time.

Next, we will implement the combat-related methods, `isValidTarget()` and `findTargetsInSight()`, inside the `defaultItem` object in `common.js`, as shown in Listing 10-8.

Listing 10-8. Implementing isValidTarget() and findTargetsInSight() (common.js)

```
isValidTarget: function(item) {
    // Cannot target units that are dead or from the same team
    if (!item || item.lifeCode === "dead" || item.team === this.team) {
        return false;
    }

    if (item.type === "buildings" || item.type === "vehicles") {
        return this.canAttackLand;
    } else if (item.type === "aircraft") {
        return this.canAttackAir;
    }

},

isTargetInSight: function(item, sightBonus = 0) {
    return Math.pow(item.x - this.x, 2) + Math.pow(item.y - this.y, 2)
        < Math.pow(this.sight + sightBonus, 2);
},

findTargetsInSight: function(sightBonus = 0) {
    var targets = [];

    game.items.forEach(function(item) {
        if (this.isValidTarget(item) && this.isTargetInSight(item, sightBonus)) {
            targets.push(item);
        }
    }, this);

    // Sort targets based on distance from attacker
    var attacker = this;

    targets.sort(function(a, b) {
        return (Math.pow(a.x - attacker.x, 2) + Math.pow(a.y - attacker.y, 2)) -
        (Math.pow(b.x - attacker.x, 2) + Math.pow(b.y - attacker.y, 2));
    });

    return targets;
}
```

The `isValidTarget()` method returns `true` if the target item is alive, from the opposite team, and it can be attacked by the attacking item.

The findTargetsInSight() method checks all the items in the game.items() array to see whether they are valid targets and within sight range, and if so, adds them to the targets array. It then sorts the targets array by distance of each target from the attacker. The method also accepts an optional sightBonus parameter, which allows us to find targets beyond the range of the item. By default, this parameter is set to 0.

The isTargetInSight() method checks whether an item is within the item's sight range, after applying the sightBonus to it.

Before we see the results of our code, we will update our level, as shown in Listing 10-9.

Listing 10-9. Updating the Level for Combat (levels.js)

```
{
    "name": "Combat",
    "briefing": "In this level you will start using weapons for combat.",

    /* Map Details */
    "mapName": "plains",

    /* Starting location for player */
    "startX": 0,
    "startY": 10,

    /* Entities to be loaded */
    "requirements": {
        "buildings": ["base", "starport", "harvester", "ground-turret"],
        "vehicles": ["transport", "harvester", "scout-tank", "heavy-tank"],
        "aircraft": ["chopper", "wraith"],
        "terrain": ["oilfield", "bigrocks", "smallrocks"]
    },

    /* Entities to be added */
    "items": [
        { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
        { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },

        { "type": "vehicles", "name": "harvester", "x": 16, "y": 12, "team": "blue",
        "direction": 3 },
        { "type": "terrain", "name": "oilfield", "x": 3, "y": 5, "action": "hint" },

        { "type": "terrain", "name": "bigrocks", "x": 19, "y": 6 },
        { "type": "terrain", "name": "smallrocks", "x": 8, "y": 3 },

        { "type": "vehicles", "name": "scout-tank", "x": 26, "y": 14, "team": "blue",
        "direction": 4 },
        { "type": "vehicles", "name": "heavy-tank", "x": 26, "y": 16, "team": "blue",
        "direction": 5 },
        { "type": "aircraft", "name": "chopper", "x": 20, "y": 12, "team": "blue", "direction": 2 },
        { "type": "aircraft", "name": "wraith", "x": 23, "y": 12, "team": "blue",
        "direction": 3 },

        { "type": "buildings", "name": "ground-turret", "x": 15, "y": 23, "team": "green" },
        { "type": "buildings", "name": "ground-turret", "x": 20, "y": 23, "team": "green" },
```

```
        { "type": "vehicles", "name": "scout-tank", "x": 16, "y": 26, "team": "green",
        "direction": 4 },
        { "type": "vehicles", "name": "heavy-tank", "x": 18, "y": 26, "team": "green", "direction": 6 },
        { "type": "aircraft", "name": "chopper", "x": 20, "y": 27, "team": "green", "direction": 2 },
        { "type": "aircraft", "name": "wraith", "x": 22, "y": 28, "team": "green", "direction": 3 },

        { "type": "buildings", "name": "base", "x": 19, "y": 28, "team": "green" },
        { "type": "buildings", "name": "starport", "x": 15, "y": 28, "team": "green", "uid":
        -1 },
    ],

    cash: {
        blue: 5000,
        green: 5000
    },

    /* Conditional and Timed Trigger Events */
    "triggers": [
    ],
}
```

We removed the triggers that we defined in the previous chapter so the level doesn't end after 30 seconds. We also added a few enemy items for us to fight with. Now, if we run the game in the browser and move a vehicle close to the enemy turrets, the turrets should start attacking the vehicle, as shown in Figure 10-1.
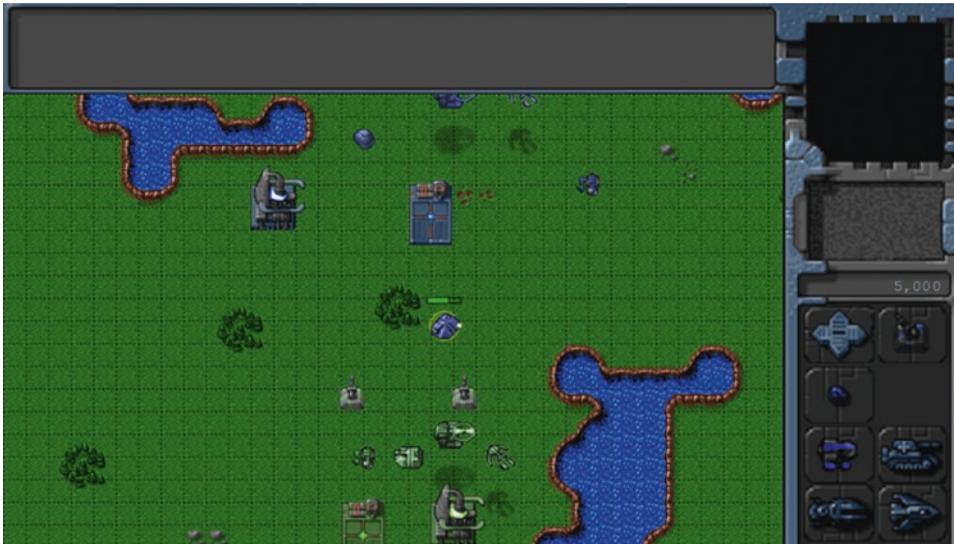
Figure 10-1. *Turret firing at a vehicle within range*

The bullets explode when they hit the vehicle and decrease the vehicle's life. Once the vehicle loses all its life, it disappears from the game. The turret stops shooting at a target if the target goes out of range, and moves on to the next target.

Next, we will implement combat-based orders for aircraft.

Combat-Based Orders for Aircraft

We will define several basic combat-based order states for aircraft:

- attack: Move within range of a target and shoot at it.

- stand: Stay in one place and attack any enemy that comes close.

- guard: Follow a friendly unit and shoot at any enemy that comes close.

- hunt: Actively seek out enemies anywhere on the map and attack them.

- patrol: Move between two points and shoot at any enemy that comes within range.

- sentry: Stay in one place and attack enemies slightly more aggressively than in stand mode.

We will implement these states by modifying the default processOrders() method inside the aircraft object, as shown in Listing 10-10.

Listing 10-10. Implementing Combat Orders for Aircraft (aircraft.js)

```
processOrders: function() {
    this.lastMovementX = 0;
    this.lastMovementY = 0;

    if (this.orders.to) {
        var distanceFromDestination = Math.pow(Math.pow(this.orders.to.x - this.x, 2) +
        Math.pow(this.orders.to.y - this.y, 2), 0.5);
        var radius = this.radius / game.gridSize;
    }

    if (this.reloadTimeLeft) {
        this.reloadTimeLeft--;
    }

    var targets;

    switch (this.orders.type) {
        case "move":
            // Move toward destination until distance from destination is less than aircraft
            radius
            if (distanceFromDestination < radius) {
                this.orders = { type: "stand" };
            } else {
                this.moveTo(this.orders.to, distanceFromDestination);
            }

            break;
```

```
case "stand":
    // Look for targets that are within sight range
    targets = this.findTargetsInSight();

    if (targets.length > 0) {
        this.orders = { type: "attack", to: targets[0] };
    }

    break;

case "sentry":
    // Look for targets up to 2 squares beyond sight range
    targets = this.findTargetsInSight(2);

    if (targets.length > 0) {
        this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
    }

    break;

case "hunt":
    // Look for targets anywhere on the map
    targets = this.findTargetsInSight(100);

    if (targets.length > 0) {
        this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
    }

    break;

case "attack":
    // If the target is no longer valid, cancel the current order
    if (!this.isValidTarget(this.orders.to)) {
        this.cancelCurrentOrder();
        break;
    }

    // Check if aircraft is within sight range of target
    if (this.isTargetInSight(this.orders.to)) {
        // Turn toward target and then start attacking when within range of the target
        var targetDirection = this.findAngleForFiring(this.orders.to);

        // Turn toward target direction if necessary
        this.turnTo(targetDirection);

        // Check if aircraft has finished turning
        if (!this.turning) {
            // If reloading has completed, fire bullet
            if (!this.reloadTimeLeft) {
                this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                var angleRadians = -(targetDirection / this.directions) * 2 * Math.PI ;
```

```
                    var bulletX = this.x - (this.radius * Math.sin(angleRadians) / game.
                    gridSize);
                    var bulletY = this.y - (this.radius * Math.cos(angleRadians) / game.
                    gridSize) - this.pixelShadowHeight / game.gridSize;

                    game.add({ name: this.weaponType, type: "bullets", x: bulletX, y:
                    bulletY, direction: targetDirection, target: this.orders.to });
                }
            }

        } else {
            // Move toward the target
            this.moveTo(this.orders.to, distanceFromDestination);
        }

        break;

    case "patrol":
        targets = this.findTargetsInSight(1);

        if (targets.length > 0) {
            // Attack the target, but save the patrol order as previousOrder
            this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
            break;
        }

        // Move toward destination until it is inside of sight range
        if (distanceFromDestination < this.sight) {
            // Swap to and from locations
            var to = this.orders.to;

            this.orders.to = this.orders.from;
            this.orders.from = to;

        } else {
            // Move toward the next destination
            this.moveTo(this.orders.to, distanceFromDestination);
        }

        break;

    case "guard":
        // If the item being guarded is dead, cancel the current order
        if (this.orders.to.lifeCode === "dead") {
            this.cancelCurrentOrder();
            break;
        }

        // If the target is inside of sight range
        if (distanceFromDestination < this.sight) {
            // Find any enemies near
```

```
            targets = this.findTargetsInSight(1);
            if (targets.length > 0) {
                // Attack the nearest target, but save the guard order as previousOrder
              this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
                break;
            }
        } else {
            // Move toward the target
            this.moveTo(this.orders.to, distanceFromDestination);
        }

        break;
    }
},
```

Within the processOrders() method, we decrease the value of the reloadTimeLeft property just like we did for turrets. We then define cases for each of the new combat-based order states.

If the order type is stand, we use findTargetsInSight() to check whether any target is nearby and, if so, attack the nearest one. We do the same thing when the order type is sentry, except we pass a range increment parameter of 2 so that the aircraft attacks units even when they are slightly beyond its typical range.

The hunt case is very similar except the range increment parameter is 100, which should ideally cover the entire map. This means the aircraft will attack any enemy unit or building on the map starting with the nearest one.

For the attack case, we first check whether the target is still alive. If not, we cancel the order using the cancelCurrentOrder() method.

Next we check whether the target is within range, and if not, we move closer to the target. We then point the aircraft toward the target, wait until the reloadTimeLeft variable is 0, and then shoot a bullet at the target, just as we did with turrets.

The patrol case is a combination of the move and sentry cases. We move the aircraft to the location defined in the to property and, once it reaches near the location, turn around and move toward the from location. In case a target comes within range, we set the order to attack with the previousOrder property set to the current order. This way, if the aircraft sees an enemy while patrolling, it will first attack the enemy and then go back to patrolling once the enemy has been destroyed.

Finally, in the case of guard mode, we move the aircraft within sight of the unit the aircraft is guarding and attack any enemy that comes close.

Whenever we initiate a new attack order, we save the current order in the previousOrder variable so that it can be restored once the attack has been completed, using the cancelCurrentOrder() method.

Next, we will implement the cancelCurrentOrder() method in the baseItem object in common.js as shown in Listing 10-11.

Listing 10-11. Implementing cancelCurrentOrder() in baseItem (common.js)

```
// Get back to the previous order if any, otherwise just stand
cancelCurrentOrder: function() {
    if (this.orders.previousOrder) {
        this.orders = this.orders.previousOrder;
    } else {
        this.orders = { type: "stand" };
    }
},
```

Within the method, we set the unit's orders to the saved `previousOrder` property if it has been set; otherwise, we go back to the default `stand` mode.

If you run the code we have so far, you should be able to see the different aircraft attacking each other, as shown in Figure 10-2.

Figure 10-2. *Aircraft attacking each other*

You can command an aircraft to attack an enemy or guard a friend by right-clicking after selecting the aircraft. The chopper can attack both land and air units, while the wraith can attack only air units.

We will typically use the `sentry`, `hunt`, and `patrol` orders to give the computer AI a slight advantage and make the game more challenging for the player. The player will not have access to these orders.

■ **Tip**   We can easily implement patrol for the player by modifying the `rightClick` method to send a patrol command if a modifier key (such as Ctrl or Shift) is pressed when the player right-clicks the ground.

Next, we will implement combat-based orders for vehicles.

Combat-Based Orders for Vehicles

The combat-based order states for vehicles will be very similar to the order states for aircraft:

- `attack`: Move within range of a target and shoot at it.

- `stand`: Stay in one place and attack any enemy that comes close.

- `guard`: Follow a friendly unit and shoot at any enemy that comes close.

- `hunt`: Actively seek out enemies anywhere on the map and attack them.

- patrol: Move between two points and shoot at any enemy that comes within range.

- sentry: Stay in one place and attack enemies slightly more aggressively than in stand mode.

We will implement these states by modifying the default processOrders() method inside the vehicles object, as shown in Listing 10-12.

Listing 10-12. Implementing Combat Orders for Vehicles (vehicles.js)

```
processOrders: function() {
    this.lastMovementX = 0;
    this.lastMovementY = 0;

    if (this.orders.to) {
        var distanceFromDestination = Math.pow(Math.pow(this.orders.to.x - this.x, 2) +
        Math.pow(this.orders.to.y - this.y, 2), 0.5);
        var radius = this.radius / game.gridSize;
    }

    if (this.reloadTimeLeft) {
        this.reloadTimeLeft--;
    }

    var targets;

    switch (this.orders.type) {
        case "move":
            // Move toward destination until distance from destination is less than vehicle
            radius
            if (distanceFromDestination < radius) {
                // Stop when within on vehicle radius of destination
                this.orders = { type: "stand" };
            } else if (this.colliding && distanceFromDestination < 3 * radius) {
                // Stop when within 3 radius of the destination if colliding with something
                this.orders = { type: "stand" };
                break;
            } else {
                if (this.colliding && distanceFromDestination < 5 * radius) {
                    // Count collisions within 5 radius distance of goal
                    if (!this.orders.collisionCount) {
                        this.orders.collisionCount = 1;
                    } else {
                        this.orders.collisionCount ++;
                    }

                    // Stop if more than 30 collisions occur
                    if (this.orders.collisionCount > 30) {
                        this.orders = { type: "stand" };
                        break;
                    }
                }
```

```
        let moving = this.moveTo(this.orders.to, distanceFromDestination);

        // Pathfinding couldn't find a path so stop
        if (!moving) {
            this.orders = { type: "stand" };
            break;
        }
    }

    break;

case "deploy":
    // If oil field has been used already, then cancel order
    if (this.orders.to.lifeCode === "dead") {
        this.orders = { type: "stand" };

        return;
    }

    if (distanceFromDestination < radius + 1) {
        // After reaching within 1 square of oil field, turn harvester to point
        toward left (direction 6)
        this.turnTo(6);

        if (!this.turning) {
            // If oil field has been used already, then cancel order
            if (this.orders.to.lifeCode === "dead") {
                this.orders = { type: "stand" };

                return;
            }

            // Once it is pointing to the left, remove the harvester and oil field
            and deploy a harvester building
            game.remove(this.orders.to);
            this.orders.to.lifeCode = "dead";

            game.remove(this);
            this.lifeCode = "dead";

            game.add({ type: "buildings", name: "harvester", x: this.orders.to.x, y:
            this.orders.to.y, action: "deploy", team: this.team });
        }
    } else {
        let moving = this.moveTo(this.orders.to, distanceFromDestination);

        // Pathfinding couldn't find a path so stop
        if (!moving) {
            this.orders = { type: "stand" };
        }
    }
```

```
        break;

    case "stand":
        // Look for targets that are within sight range
        targets = this.findTargetsInSight();

        if (targets.length > 0) {
            this.orders = { type: "attack", to: targets[0] };
        }

        break;

    case "sentry":
        // Look for targets up to 2 squares beyond sight range
        targets = this.findTargetsInSight(2);

        if (targets.length > 0) {
            this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
        }

        break;

    case "hunt":
        // Look for targets anywhere on the map
        targets = this.findTargetsInSight(100);

        if (targets.length > 0) {
            this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
        }

        break;

    case "attack":
        // If the target is no longer valid, cancel the current order
        if (!this.isValidTarget(this.orders.to)) {
            this.cancelCurrentOrder();
            break;
        }

        // Check if vehicle is within sight range of target
        if (this.isTargetInSight(this.orders.to)) {
            // Turn toward target and then start attacking when within range of the target
            var targetDirection = this.findAngleForFiring(this.orders.to);

            // Turn toward target direction if necessary
            this.turnTo(targetDirection);

            // Check if vehicle has finished turning
            if (!this.turning) {
                // If reloading has completed, fire bullet
```

```
                if (!this.reloadTimeLeft) {
                    this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                  var angleRadians = -(targetDirection / this.directions) * 2 * Math.PI ;
                    var bulletX = this.x - (this.radius * Math.sin(angleRadians) / game.
                    gridSize);
                    var bulletY = this.y - (this.radius * Math.cos(angleRadians) / game.
                    gridSize);

                    game.add({ name: this.weaponType, type: "bullets", x: bulletX, y:
                    bulletY, direction: targetDirection, target: this.orders.to });
                }
            }

        } else {
            // Move toward the target
            this.moveTo(this.orders.to, distanceFromDestination);
        }

        break;

    case "patrol":
        targets = this.findTargetsInSight(1);

        if (targets.length > 0) {
            // Attack the target, but save the patrol order as previousOrder
            this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
            break;
        }

        // Move toward destination until it is inside of sight range
        if (distanceFromDestination < this.sight) {
            // Swap to and from locations
            var to = this.orders.to;

            this.orders.to = this.orders.from;
            this.orders.from = to;

        } else {
            // Move toward the next destination
            this.moveTo(this.orders.to, distanceFromDestination);
        }

        break;

    case "guard":
        // If the item being guarded is dead, cancel the current order
        if (this.orders.to.lifeCode === "dead") {
            this.cancelCurrentOrder();
            break;
        }
```

```
            // If the target is inside of sight range
            if (distanceFromDestination < this.sight) {
                // Find any enemies near
                targets = this.findTargetsInSight(1);
                if (targets.length > 0) {
                    // Attack the nearest target, but save the guard order as previousOrder
                    this.orders = { type: "attack", to: targets[0], previousOrder: this.orders };
                    break;
                }
            } else {
                // Move toward the target
                this.moveTo(this.orders.to, distanceFromDestination);
            }

            break;
    }
},
```

The implementation of the states is almost the same as for aircraft. If we run the game now, we should be able to attack with the vehicles, as shown in Figure 10-3.
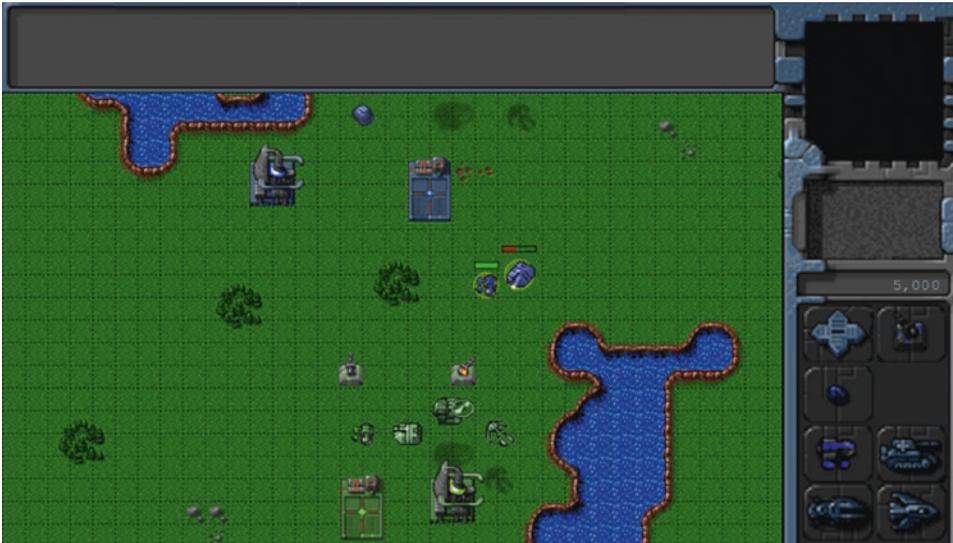
Figure 10-3. *Attacking with the vehicles*

We can now attack with vehicles, aircraft, or turrets.

You may have noticed that while the opposing team's units attack when you come close, they are still very easily defeated. Now that the combat system is in place, we will explore ways to make the enemy more intelligent and the game more challenging.

Building Intelligent Enemy

The primary goal in building an intelligent enemy AI is to make sure that the person playing the game finds it reasonably challenging and has a fun experience completing the level. An important thing to realize about RTS games, especially the single-player campaign, is that the enemy AI doesn't need to be a grandmaster-level chess player. The fact is, we can provide the player with a very compelling experience using only a combination of combat order states and conditional scripted events.

Typically, the "intelligent" way to behave for the AI will vary with each level.

In a simple level where there are no production facilities and only ground units, the only possible behavior is to drive up to the enemy units and attack them. A combination of `patrol` and `sentry` orders is usually more than enough to achieve this. We could also make the level interesting by attacking the player at a specific time or when a certain event occurs (for example, when the player arrives at a certain location or constructs a particular building).

In a more complex level, we might make the enemy challenging by constructing and sending in waves of enemies at specific intervals using timed triggers and the `hunt` order.

We can see some of these ideas at work by adding a few more items and triggers to the level, as shown in Listing 10-13.

Listing 10-13. Adding Triggers and Items to Make the Level Challenging (levels.js)

```
/* Entities to be added */
"items": [
    { "type": "buildings", "name": "base", "x": 11, "y": 14, "team": "blue" },
    { "type": "buildings", "name": "starport", "x": 18, "y": 14, "team": "blue" },

    { "type": "vehicles", "name": "harvester", "x": 16, "y": 12, "team": "blue", "direction": 3 },
    { "type": "terrain", "name": "oilfield", "x": 3, "y": 5, "action": "hint" },

    { "type": "terrain", "name": "bigrocks", "x": 19, "y": 6 },
    { "type": "terrain", "name": "smallrocks", "x": 8, "y": 3 },

    { "type": "vehicles", "name": "scout-tank", "x": 26, "y": 14, "team": "blue", "direction": 4 },
    { "type": "vehicles", "name": "heavy-tank", "x": 26, "y": 16, "team": "blue", "direction": 5 },
    { "type": "aircraft", "name": "chopper", "x": 20, "y": 12, "team": "blue", "direction": 2 },
    { "type": "aircraft", "name": "wraith", "x": 23, "y": 12, "team": "blue", "direction": 3 },

    { "type": "buildings", "name": "ground-turret", "x": 15, "y": 23, "team": "green" },
    { "type": "buildings", "name": "ground-turret", "x": 20, "y": 23, "team": "green" },

    { "type": "vehicles", "name": "scout-tank", "x": 16, "y": 26, "team": "green",
    "direction": 4, "orders": { "type": "sentry" } },
    { "type": "vehicles", "name": "heavy-tank", "x": 18, "y": 26, "team": "green",
    "direction": 6, "orders": { "type": "sentry" } },
    { "type": "aircraft", "name": "chopper", "x": 20, "y": 27, "team": "green", "direction": 2,
    "orders": { "type": "hunt" } },
    { "type": "aircraft", "name": "wraith", "x": 22, "y": 28, "team": "green", "direction": 3,
    "orders": { "type": "hunt" } },

    { "type": "buildings", "name": "base", "x": 19, "y": 28, "team": "green" },
    { "type": "buildings", "name": "starport", "x": 15, "y": 28, "team": "green", "uid": -1 },
],
```

```
/* Economy Related*/
"cash": {
    "blue": 5000,
    "green": 5000
},

/* Conditional and Timed Trigger Events */
"triggers": [
    /* Timed Events*/
    {
        "type": "timed",
        "time": 1000,
        "action": function() {
            game.sendCommand([-1], { type: "construct-unit", details: { type: "aircraft",
            name: "wraith", orders: { "type": "patrol", "from": { "x": 22, "y": 30 }, "to":
            { "x": 15, "y": 21 } } } });
        }
    },
    {
        "type": "timed",
        "time": 5000,
        "action": function() {
            game.sendCommand([-1], { type: "construct-unit", details: { type: "aircraft",
            name: "chopper", orders: { "type": "patrol", "from": { "x": 15, "y": 30 }, "to":
            { "x": 22, "y": 21 } } } });
        }
    },
    {
        "type": "timed",
        "time": 10000,
        "action": function() {
            game.sendCommand([-1], { type: "construct-unit", details: { type: "vehicles",
            name: "heavy-tank", orders: { "type": "patrol", "from": { "x": 15, "y": 30 },
            "to": { "x": 22, "y": 21 } } } });
        }
    },
    {
        "type": "timed",
        "time": 15000,
        "action": function() {
            game.sendCommand([-1], { type: "construct-unit", details: { type: "vehicles",
            name: "scout-tank", orders: { "type": "patrol", "from": { "x": 22, "y": 30 },
            "to": { "x": 15, "y": 21 } } } });
        }
    },
    {
        "type": "timed",
        "time": 60000,
        "action": function() {
            game.showMessage("AI", "Now every enemy unit is going to attack you in a wave.");
            var units = [];
```

```
            for (var i = 0; i < game.items.length; i++) {
                var item = game.items[i];

                if (item.team === "green" && (item.type === "vehicles" || item.type ===
                "aircraft")) {
                    units.push(item.uid);
                }
            }
            game.sendCommand(units, { type: "hunt" });
        }
    },
],
```

The first thing we do is order an enemy chopper and a wraith to hunt as soon as the game starts. Next, we assign a UID of -1 to the enemy starport and set a few timed triggers to build different types of patrolling units every few seconds.

Finally, after 60 seconds, we command all enemy units to hunt and notify the player using the showMessage() method.

If we run the code now, we can expect the AI to defend itself fairly well and attack very aggressively at the end of 60 seconds, as shown in Figure 10-4.
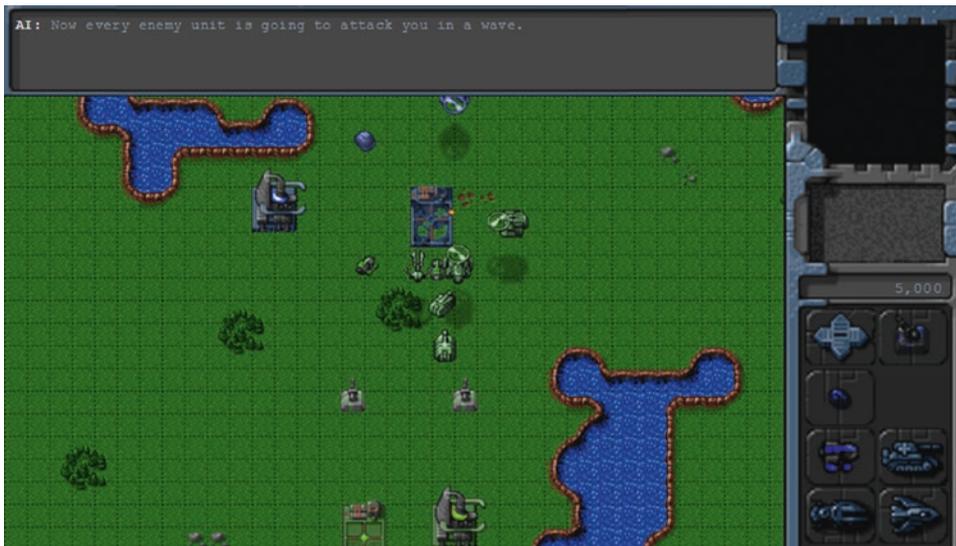
Figure 10-4. *Computer AI aggressively attacking player*

Obviously, this is a fairly contrived example. No one will want to play a game where they get attacked this brutally within the first minute of playing. However, as this example illustrates, we can make the game as easy or as challenging as we need just by adjusting these triggers and orders.

■ **Tip**   You can implement separate sets of triggers and starting items depending on a difficulty setting so that the player can play easy or challenging versions of the same campaign based on the setting selected.

Now that we have implemented the combat system and explored ways to make the game AI challenging, the last thing we will look at in this chapter is adding a fog of war.

Adding a Fog of War

The fog of war is typically a dark, colored shroud that covers all unexplored terrain within the map. As player units move around the map, the fog is cleared anywhere that the unit can see.

This introduces elements of exploration and intrigue to the game. The ability to hide under the fog allows the use of strategies such as hidden bases, ambushes, and sneak attacks.

Some RTS games permanently remove the fog once an area is explored, while others clear the fog only in areas within sight of a player unit and bring back the fog once the unit leaves the area. For our game, we will be using the second implementation.

Defining the Fog Object

We will start by defining a new fog object inside fog.js, as shown in Listing 10-14.

Listing 10-14. Implementing the fog Object (fog.js)

```
var fog = {
    grid: [],
    canvas: document.createElement("canvas"),
    initLevel: function() {
        // Set fog canvas to the size of the map
        this.canvas.width = game.currentMap.mapGridWidth * game.gridSize;
        this.canvas.height = game.currentMap.mapGridHeight * game.gridSize;

        this.context = this.canvas.getContext("2d");

        // Set the fog grid for the player to 2d array with all values set to 1
        this.defaultFogGrid = [];

        let row = new Array(game.currentMap.gridMapWidth);

        for (let x = 0; x < game.currentMap.mapGridWidth; x++) {
            row[x] = 1;
        }

        for (let y = 0; y < game.currentMap.mapGridHeight; y++) {
            this.defaultFogGrid[y] = row.slice(0);
        }
    },

    isPointOverFog: function(x, y) {
        // If the point is outside the map bounds consider it fogged
```

```
    if (y < 0 || y / game.gridSize >= game.currentMap.mapGridHeight || x < 0 || x /
    game.gridSize >= game.currentMap.mapGridWidth ) {
        return true;
    }

    // If not, return value based on the player's fog grid
    return this.grid[game.team][Math.floor(y / game.gridSize)][Math.floor(x / game.
    gridSize)] === 1;
},

animate: function() {
    // Fill fog with semi solid black color over the map
    this.context.drawImage(game.currentMapImage, 0, 0);
    this.context.fillStyle = "rgba(0,0,0,0.8)";
    this.context.fillRect(0, 0, this.canvas.width, this.canvas.height);

    // Initialize the players fog grid
    this.grid[game.team] = game.makeArrayCopy(this.defaultFogGrid);

    // Clear all areas of the fog where a player item has vision
    fog.context.globalCompositeOperation = "destination-out";
    game.items.forEach(function(item) {
        var team = game.team;

        if (item.team === team && !item.keepFogged) {
            var x = Math.floor(item.x);
            var y = Math.floor(item.y);
            var x0 = Math.max(0, x - item.sight + 1);
            var y0 = Math.max(0, y - item.sight + 1);
            var x1 = Math.min(game.currentMap.mapGridWidth - 1, x + item.sight - 1 +
            (item.type === "buildings" ? item.baseWidth / game.gridSize : 0));
            var y1 = Math.min(game.currentMap.mapGridHeight - 1, y + item.sight - 1 +
            (item.type === "buildings" ? item.baseHeight / game.gridSize : 0));

            for (var j = x0; j <= x1; j++) {
                for (var k = y0; k <= y1; k++) {
                    if ((j > x0 && j < x1) || (k > y0 && k < y1)) {
                        if (this.grid[team][k][j]) {
                            this.context.fillStyle = "rgba(100,0,0,0.9)";
                            this.context.beginPath();
                            this.context.arc(j * game.gridSize + 12, k * game.gridSize +
                            12, 16, 0, 2 * Math.PI, false);
                            this.context.fill();
                            this.context.fillStyle = "rgba(100,0,0,0.7)";
                            this.context.beginPath();
                            this.context.arc(j * game.gridSize + 12, k * game.gridSize +
                            12, 18, 0, 2 * Math.PI, false);
                            this.context.fill();
```

```
                            this.context.fillStyle = "rgba(100,0,0,0.5)";
                            this.context.beginPath();
                            this.context.arc(j * game.gridSize + 12, k * game.gridSize +
                            12, 24, 0, 2 * Math.PI, false);
                            this.context.fill();

                        }
                        this.grid[team][k][j] = 0;
                    }
                }
            }
        }
    }, this);
    fog.context.globalCompositeOperation = "source-over";
},

draw: function() {
    game.foregroundContext.drawImage(this.canvas, game.offsetX, game.offsetY,
    game.canvasWidth, game.canvasHeight, 0, 0, game.canvasWidth, game.canvasHeight);
}
};
```

We start by defining a canvas inside the fog object. The initLevel() method resizes the canvas object to the size of the current map and defines a defaultFogGrid array that has the same dimensions as the map with all its elements set to 1.

The isPointOverFog() method returns true if a given x and y coordinate is either outside the map bounds or lies within a fogged cell.

Within the animate() method, we first initialize the fog canvas to the map background with a semi-transparent black layer over it. This way, fogged areas of the map show up as darkened background terrain.

We then iterate through each of the items in the game and clear the fog array and the fog canvas around the player's items based on their sight property. We do not clear the fog for items that are the opposing player's or that have a keepFogged attribute set to true.

Finally, the draw() method draws the fog canvas onto the game.foregroundContext context using the same offsets that we used when drawing the map onto the game.backgroundContext context.

Drawing the Fog

Now that we have defined the fog object, we will start by adding a reference to fog.js inside the head section of index.html, as shown in Listing 10-15.

Listing 10-15. Adding a Reference to the fog Object (index.html)

```
<script src="js/fog.js" type="text/javascript"></script>
```

Next, we need to initialize the fog once the level is loaded. We will do this by calling the fog.initLevel() method inside the singleplayer object's initLevel() method, as shown in Listing 10-16.

Listing 10-16. Initializing the fog Object for the Level (singleplayer.js)

```
initLevel: function() {
    game.type = "singleplayer";
    game.team = "blue";

    // Don't allow player to enter mission until all assets for the level are loaded
    var enterMissionButton = document.getElementById("entermission");

    enterMissionButton.disabled = true;

    // Load all the items for the level
    var level = levels.singleplayer[singleplayer.currentLevel];

    game.loadLevelData(level);

    fog.initLevel();

    // Set player starting location
    game.offsetX = level.startX * game.gridSize;
    game.offsetY = level.startY * game.gridSize;

    game.createTerrainGrid();

    // Enable the Enter Mission button once all assets are loaded
    loader.onload = function() {
        enterMissionButton.disabled = false;
    };

    // Update the mission briefing text and show briefing screen
    this.showMissionBriefing(level.briefing);
},
```

Next we need to modify the game object's animationLoop() and drawingLoop() methods to call fog.animate() and fog.draw() respectively, as shown in Listing 10-17.

Listing 10-17. Calling fog.animate() and fog.draw() (game.js)

```
animationLoop: function() {

    // Animate the sidebar
    sidebar.animate();

    // Process orders for any item that handles orders
    game.items.forEach(function(item) {
        if (item.processOrders) {
            item.processOrders();
        }
    });
```

```
    // Animate each of the elements within the game
    game.items.forEach(function(item) {
        item.animate();
    });

    // Sort game items into a sortedItems array based on their x,y coordinates
    game.sortedItems = Object.assign([], game.items);
    game.sortedItems.sort(function(a, b) {
        return a.y - b.y + ((a.y === b.y) ? (b.x - a.x) : 0);
    });

    fog.animate();

    // Save the time that the last animation loop completed
    game.lastAnimationTime = Date.now();
},

// The map is broken into square tiles of this size (20 pixels x 20 pixels)
gridSize: 20,
// X & Y panning offsets for the map
offsetX: 0,
offsetY: 0,

drawingLoop: function() {
    // Pan the map if the cursor is near the edge of the canvas
    game.handlePanning();

    // Check the time since the game was animated and calculate a linear interpolation
    factor (-1 to 0)
    game.lastDrawTime = Date.now();
    if (game.lastAnimationTime) {
        game.drawingInterpolationFactor = (game.lastDrawTime - game.lastAnimationTime) /
        game.animationTimeout - 1;

        // No point interpolating beyond the next animation loop...
        if (game.drawingInterpolationFactor > 0) {
            game.drawingInterpolationFactor = 0;
        }
    } else {
        game.drawingInterpolationFactor = -1;
    }

    // Draw the background whenever necessary
    game.drawBackground();

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0, 0, game.canvasWidth, game.canvasHeight);

    // Start drawing the foreground elements
    game.sortedItems.forEach(function(item) {
        item.draw();
    });
```

313

```
    // Draw exploding bullets on top of everything else
    game.bullets.forEach(function(bullet) {
        if (bullet.action === "explode") {
            bullet.draw();
        }
    });

    fog.draw();

    // Draw the mouse
    mouse.draw();

    // Call the drawing loop for the next frame using request animation frame
    if (game.running) {
        requestAnimationFrame(game.drawingLoop);
    }
},
```

If we run the code now, we should see the entire map shrouded in a fog of war, as shown in Figure 10-5.
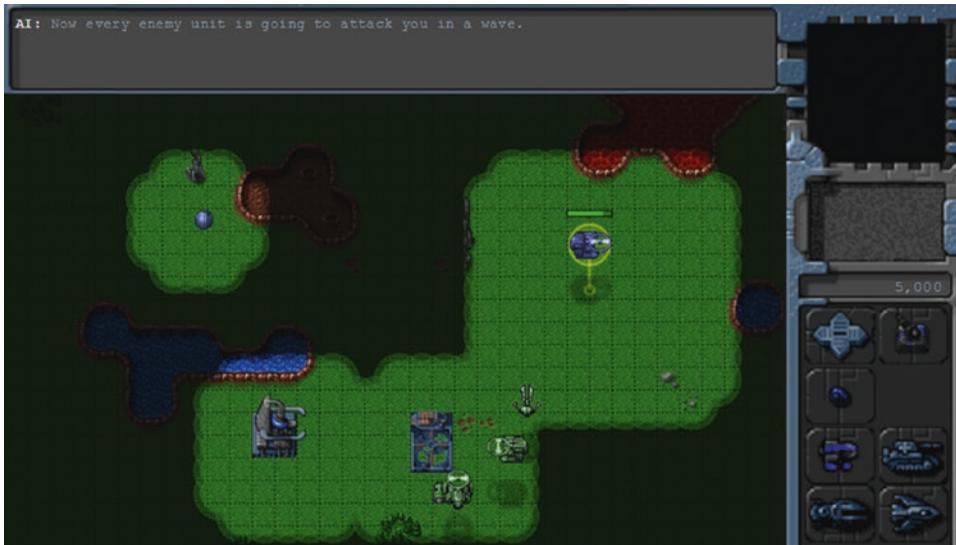
Figure 10-5. *Map shrouded in fog of war*

You will see that the fog is uncovered around friendly units and buildings. Also, the fogged area shows the original terrain but does not show any units under it.

The fog effectively hides all enemy movement, allowing them to show up seemingly out of nowhere. The same enemy attack feels much scarier when we have no idea about the size or the location of the opposing army.

Before we wrap up the chapter, we will add a few finishing touches to the fog of war.

Adding Finishing Touches

The first change we will make is to prevent the deploying of buildings on fogged areas by making fogged areas unbuildable. We will modify the sidebar object's checkBuildingPlacement() method, as shown in Listing 10-18.

Listing 10-18. Making Fogged Areas Unbuildable (sidebar.js)

```
checkBuildingPlacement: function() {

    let name = sidebar.deployBuilding.name;
    let details = buildings.list[name];

    // Create a buildable grid to identify where building can be placed
    game.rebuildBuildableGrid();

    // Use buildableGrid to identify whether we can place the building
    let canDeployBuilding = true;
    let placementGrid = game.makeArrayCopy(details.buildableGrid);

    for (let y = placementGrid.length - 1; y >= 0; y--) {
        for (let x = placementGrid[y].length - 1; x >= 0; x--) {

            // If a tile needs to be buildable for the building
            if (placementGrid[y][x] === 1) {
                // Check whether the tile is inside the map and buildable
                if (mouse.gridY + y >= game.currentMap.mapGridHeight
                || mouse.gridX + x >= game.currentMap.mapGridWidth
                || fog.grid[game.team][mouse.gridY + y][mouse.gridX + x]
                || game.currentMapBuildableGrid[mouse.gridY + y][mouse.gridX + x]) {
                    // Otherwise mark tile as unbuildable
                    canDeployBuilding = false;
                    placementGrid[y][x] = 2;
                }
            }
        }
    }

    sidebar.placementGrid = placementGrid;
    sidebar.canDeployBuilding = canDeployBuilding;

},
```

We add an extra condition for testing the fog grid when creating the placementGrid array so that a fogged grid square is no longer buildable. If we run the game and try to build on a fogged area, we should see a warning, as shown in Figure 10-6.
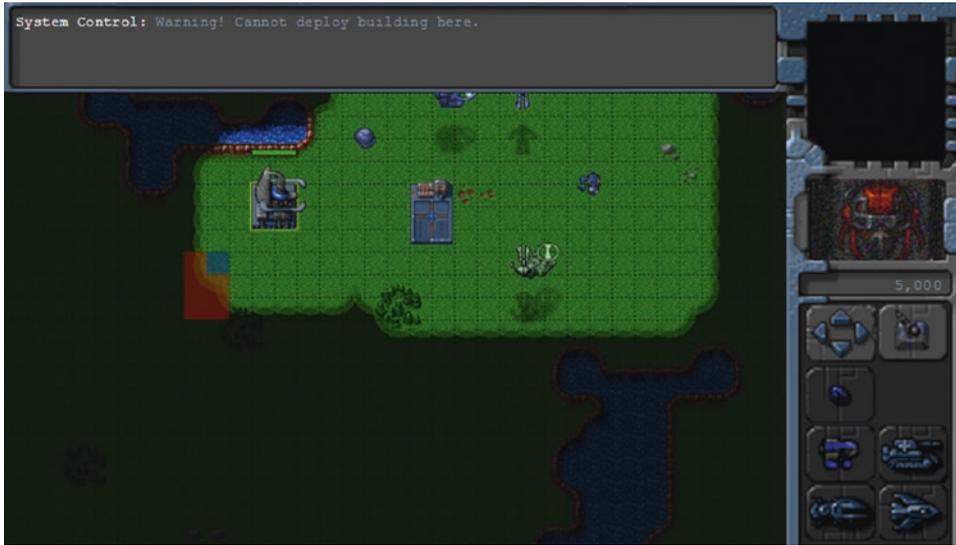
Figure 10-6. *Cannot deploy buildings on fogged areas*

As you can see, the building deploy grid turns red on fogged areas to indicate that the player cannot build there. If you still try to click a fogged area, you will get a system warning.

Next, we will make sure that the player cannot select or detect a building or unit that is under the fog. We do this by modifying the mouse object's itemUnderMouse() method, as shown in Listing 10-19.

Listing 10-19. Hiding Objects Under the Fog (mouse.js)

```
// Return the first item detected under the mouse.
itemUnderMouse: function() {
    // If the mouse is over fog, don't detect any items
    if (fog.isPointOverFog(mouse.gameX, mouse.gameY)) {
        return;
    }

    for (let i = game.items.length - 1; i >= 0; i--) {
        let item = game.items[i];

        // Dead items will not be detected
        if (item.lifeCode === "dead") {
            continue;
        }

        let x = item.x * game.gridSize;
        let y = item.y * game.gridSize;

        if (item.type === "buildings" || item.type === "terrain") {
            // If mouse coordinates are within rectangular area of building or terrain
            if (x <= mouse.gameX && x >= (mouse.gameX - item.baseWidth) && y <= mouse.gameY
            && y >= (mouse.gameY - item.baseHeight)) {
                return item;
            }
```

```
    } else if (item.type === "aircraft") {
            // If mouse coordinates are within radius of aircraft (adjusted for
            pixelShadowHeight)
        if (Math.pow(x - mouse.gameX, 2) + Math.pow(y - mouse.gameY - item.
        pixelShadowHeight, 2) < Math.pow(item.radius, 2)) {
            return item;
        }
    } else if (item.type === "vehicles") {
            // If mouse coordinates are within radius of item
        if (Math.pow(x - mouse.gameX, 2) + Math.pow(y - mouse.gameY, 2) < Math.pow(item.
        radius, 2)) {
            return item;
        }
    }
  }
},
```

We return nothing if the point under the mouse is fogged. This way, enemy units under the fog are undetectable and cannot be clicked by the player.

With this last change, we now have a completely working fog of war in our game.

Summary

In this chapter, we implemented a combat system for our game. We started by defining a `bullets` object with different types of bullets. We then added several combat-based order states to our turrets, aircraft, and vehicles. We used these orders along with the triggers system we defined in the previous chapter to create a fairly challenging enemy. Finally, we implemented a fog of war.

Our game now has most of the essential elements of an RTS. In the next chapter, we will polish our game framework by adding sound and mobile support. We will then use this framework to build a few interesting levels and wrap up our single-player campaign.

■ ■ ■

Wrapping Up the Single-Player Campaign

Our game framework now has almost everything we need to build a very nice single-player campaign: a level system, various units and buildings, intelligent movement using pathfinding, an economy, and finally combat.

Now it's time to add the finishing touches and wrap up our single-player campaign. We will first add sound effects such as explosions and voices to our game. We will then build several levels by combining and using the various elements that we developed over the past few chapters. You will see how these building blocks fall into place to create a complete game.

Let's get started. We will continue where we left off at the end of Chapter 10.

Adding Sound

RTS games have a lot more happening at the same time than games in other genres such as the physics game we developed in the first few chapters. If we are not careful, there is a possibility of overwhelming a player with so much audio input that it becomes a distraction and takes away from their immersion. For our game, we will focus on sounds that will make the player aware of essential events within the game.

- *Acknowledging commands*: Any time the player selects a unit and gives it a command, we will have the unit acknowledge that it received the command.

- *Messages*: Whenever the player receives either a system warning or a story line–driven notification, we will alert the player with a sound.

- *Combat*: We will add sounds during combat so that players instantly know that they are under attack somewhere on the map.

Setting Up Sounds

We will start by creating a sounds object inside sounds.js, as shown in Listing 11-1.

Listing 11-1. Creating a sounds Object (sounds.js)

```
var sounds = {
    list: {
        "bullet": ["bullet1", "bullet2"],
        "heatseeker": ["heatseeker1", "heatseeker2"],
```

```
        "fireball": ["laser1", "laser2"],
        "cannon-ball": ["cannon1", "cannon2"],
        "message-received": ["message"],
        "acknowledge-attacking": ["engaging"],
        "acknowledge-moving": ["yup", "roger1", "roger2"],
    },

    loaded: {},
    init: function() {
        // Iterate through the sound names in the list, and load audio files for each
        for (let soundName in this.list) {
            let sound = {
                // Store a counter to keep track of which sound is played next
                counter: 0
            };

            sound.audioObjects = [];
            this.list[soundName].forEach(function(fileName) {
                sound.audioObjects.push(loader.loadSound("audio/" + fileName));
            }, this);

            this.loaded [soundName] = sound;
        }
    },

    play: function(soundName) {
        let sound = sounds.loaded[soundName];

        if (sound) {
            // Play audio for sound name based on counter location
            let audioObject = sound.audioObjects[sound.counter];

            audioObject.play();

            // Move to the next audio next time
            sound.counter++;
            if (sound.counter >= sound.audioObjects.length) {
                sound.counter = 0;
            }
        }
    }
};
```

Within the sound object, we start by declaring a list, which maps a sound name to one or more sound files. For example, the bullet sound maps to two files: bullet1 and bullet2. You will notice that we don't specify the file extension (.ogg or .mp3). We let the loader object handle selecting the appropriate audio file extension for the browser.

Next we declare an init() method that iterates through the list of sounds, uses the loader.loadSound() method to load each audio file, and then creates an audioObjects array for each sound name. We then add this sound object to the loaded object.

Finally, we declare a `play()` method that looks up the appropriate `sound` object from the `loaded` array and then plays the audio object using its `play()` method. You will notice that we use a counter for each `sound` object to ensure that we iterate through the sounds for a given sound name so that a different sound is played each time `play()` is called. This allows us to play different versions of sounds for an event instead of hearing the same monotonous sound each time.

Next, we will add a reference to `sounds.js` inside the `head` section of `index.html`, as shown in Listing 11-2.

Listing 11-2. Referring to sounds.js (index.html)

```
<script src="js/sounds.js" type="text/javascript"></script>
```

Finally, we will load all these sounds when the game is initialized by calling the `init()` method from inside the game object's `init()` method, as shown in Listing 11-3.

Listing 11-3. Initializing the sounds Object Inside the game.init() Method (game.js)

```
// Start initializing objects, preloading assets and display start screen
init: function() {
    // Initialize objects
    loader.init();
    mouse.init();
    sidebar.init();
    sounds.init();

    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

Now that the `sounds` object is in place, we can start adding sounds for each event, starting with acknowledging commands.

Acknowledging Commands

We allow the player to give units several types of commands: `attack`, `move`, `deploy`, and `guard`. Any time a unit is sent an `attack` command, we will play the `acknowledge-attacking` sound. When the unit is sent any other command such as `move` or `guard`, we will play the `acknowledge-moving` sound.

We will do this by calling `sounds.play()` from inside the `rightClick()` method of the `mouse` object, as shown in Listing 11-4.

Listing 11-4. Acknowledging Commands Inside the rightClick() Method (mouse.js)

```
// Called whenever player completes a right-click on the game canvas
rightClick: function() {
    // If the game is in deployBuilding mode, right-clicking will cancel deployBuilding mode
    if (sidebar.deployBuilding) {
        sidebar.cancelDeployingBuilding();

        return;
    }
```

```
    let clickedItem = mouse.itemUnderMouse();

    // Handle actions like attacking and movement of selected units
    if (clickedItem) { // Player right-clicked on something
        if (clickedItem.type !== "terrain") {
            if (clickedItem.team !== game.team) { // Player right-clicked on an enemy item
                let uids = [];

                // Identify selected units from player's team that can attack
                game.selectedItems.forEach(function(item) {
                    if (item.team === game.team && item.canAttack) {
                        uids.push(item.uid);
                    }
                }, this);

                // Command units to attack the clicked item
                if (uids.length > 0) {
                    game.sendCommand(uids, { type: "attack", toUid: clickedItem.uid });

                    sounds.play("acknowledge-attacking");
                }

            } else { // Player right-clicked on a friendly item
                let uids = [];

                // Identify selected units from player's team that can move
                game.selectedItems.forEach(function(item) {
                    if (item.team === game.team && item.canAttack && item.canMove) {
                        uids.push(item.uid);
                    }
                }, this);

                // Command units to guard the clicked item
                if (uids.length > 0) {
                    game.sendCommand(uids, { type: "guard", toUid: clickedItem.uid });

                    sounds.play("acknowledge-moving");
                }

            }
        } else if (clickedItem.name === "oilfield") { // Player right-clicked on an oilfield
            let uids = [];

            // Identify the first selected harvester (since only one can deploy at a time)
            for (let i = game.selectedItems.length - 1; i >= 0; i--) {
                let item = game.selectedItems[i];

                if (item.team === game.team && item.type === "vehicles" && item.name ===
                "harvester") {
                    uids.push(item.uid);
                    break;
                }
            }
```

```
        // Command it to deploy on the oil field
        if (uids.length > 0) {
            game.sendCommand(uids, { type: "deploy", toUid: clickedItem.uid });

            sounds.play("acknowledge-moving");
        }
    }
} else { // Player right-clicked on the ground
    let uids = [];

    // Identify selected units from player's team that can move
    game.selectedItems.forEach(function(item) {
        if (item.team === game.team && item.canMove) {
            uids.push(item.uid);
        }
    }, this);

    // Command units to move to the clicked location
    if (uids.length > 0) {
        game.sendCommand(uids, { type: "move", to: { x: mouse.gameX / game.gridSize,
        y: mouse.gameY / game.gridSize } });

        sounds.play("acknowledge-moving");
    }
}
},
```

We call the sounds.play() method with the appropriate sound name whenever we send a game command.

One interesting thing to point out is that we play the sound when the command is sent out, not when it is received and processed. While this makes very little difference during the single-player campaign, it becomes important during multiplayer.

Usually, network latency and other issues can cause a lag of up to a few hundred milliseconds between the sending of a command and it actually being received by all the players. By playing the sound as soon as the mouse is clicked, we give the player the illusion that the command has been executed immediately and make the effect of lag less noticeable.

■ **Note**   In an attempt to hide game lag, some games use animation sequences in addition to sounds to indicate to the player that the unit is processing the command even before the server has acknowledged the command. Games such as first-person shooters often attempt to predict the unit movement and start moving the unit before receiving the server acknowledgment. RTS games typically do not need client-side prediction techniques.

If you open and run the game now, you should hear the units acknowledge the command before they start moving or attacking. Next, let's add the message sound.

Messages

We will play a short beeping sound to notify players whenever they are shown a message. We will do this by playing the `message-received` sound from inside the game object's `showMessage()` method, as shown in Listing 11-5.

Listing 11-5. Message Notification Sound Inside the showMessage() Method (game.js)

```
showMessage: function(from, message) {

    sounds.play("message-received");

    let callerpicture = document.getElementById("callerpicture");
    let gamemessages = document.getElementById("gamemessages");

    // If the message is from a defined game character, show profile picture
    let character = game.characters[from];

    if (character) {

        // Use the character's defined name
        from = character.name;

        if (character.image) {
            // Display the character image in the caller picture area
            callerpicture.innerHTML = "<img src=\"images/characters/" + character.image + "\"/>";

            // Remove the caller picture after six seconds
            setTimeout(function() {
                callerpicture.innerHTML = "";
            }, 6000);
        }
    }

    // Append message to messages pane and scroll to the bottom

    let messageHTML = "<span>" + from + ": </span>" + message + "<br>";

    gamemessages.innerHTML += messageHTML;
    gamemessages.scrollTop = gamemessages.scrollHeight;

},
```

If you play the game now, you should hear beeping whenever a new message is displayed.
The last set of sounds we will implement is for combat.

Combat

You may have noticed that we declared four different sound types within our sounds list: `bullet`, `heatseeker`, `cannon-ball`, and `fireball`. These four sounds correspond to the four bullet types that we declared in the previous chapter. Any time we fire a bullet, we will play the sound for the appropriate bullet.

We can easily do this by modifying the add() method inside game.js to play the appropriate sound whenever a bullet is added, as shown in Listing 11-6.

Listing 11-6. Playing Sound When a Bullet Is Added (game.js)

```
add: function(itemDetails) {
    // Set a unique id for the item
    if (!itemDetails.uid) {
        itemDetails.uid = ++game.counter;
    }

    var item = window[itemDetails.type].add(itemDetails);

    // Add the item to the items array
    game.items.push(item);

    // Add the item to the type-specific array
    game[item.type].push(item);

    // Reset currentMapPassableGrid whenever the map changes
    if (item.type === "buildings" || item.type === "terrain") {
        game.currentMapPassableGrid = undefined;
    }

    // Play bullet firing sound when a bullet is created
    if (item.type === "bullets") {
        sounds.play(item.name);
    }

    return item;
},
```

If you play the game now, you should hear the distinct sounds of the different weapons as they are being fired.

We could keep adding more sounds to our game if we wanted, such as explosions, construction noises, conversation, and even background music. The process would remain the same. However, the sounds we have implemented so far are sufficient for now.

Now that we have sound in our game, let's take a look at adding support for mobile and touch devices.

Supporting Mobile Devices

We have already designed our game to be responsive and work with different resolutions and aspect ratios. This means that if we were to open the game on any mobile device, the game would automatically scale and fit correctly. So to support mobile devices, we will need to enable touch support, add mobile meta tags, and enable web audio, just like we did in Chapter 5.

Enabling Touch Support

The first thing we will do is add event listeners for the touch events inside the mouse.init() method as shown in Listing 11-7.

Listing 11-7. Listening for Touch Events (mouse.js)

```
init: function() {
    // Listen for mouse events on the game foreground canvas
    let canvas = document.getElementById("gameforegroundcanvas");

    canvas.addEventListener("mousemove", mouse.mousemovehandler, false);

    canvas.addEventListener("mouseenter", mouse.mouseenterhandler, false);
    canvas.addEventListener("mouseout", mouse.mouseouthandler, false);

    canvas.addEventListener("mousedown", mouse.mousedownhandler, false);
    canvas.addEventListener("mouseup", mouse.mouseuphandler, false);

    canvas.addEventListener("contextmenu", mouse.mouserightclickhandler, false);

    canvas.addEventListener("touchstart", mouse.touchstarthandler, { passive: false });
    canvas.addEventListener("touchend", mouse.touchendhandler, { passive: false });
    canvas.addEventListener("touchmove", mouse.touchmovehandler, { passive: false });

    mouse.canvas = canvas;
},
```

We listen for all three touch events, touchstart, touchend, and touchmove, and assign handler methods for each. We set an additional passive property to false, which lets the browser know that we might be preventing the default behavior of these events from inside the handlers.

Next, we need to implement the actual handlers. We will start by implementing the touchstarthandler() and touchmovehandler() methods to enable drag selection using mobile as shown in Listing 11-8.

Listing 11-8. The touchstarthandler() and touchmovehandler() Methods (mouse.js)

```
touchstarthandler: function(ev) {
    mouse.insideCanvas = true;
    let touch = event.targetTouches[0];

    mouse.setCoordinates(touch.clientX, touch.clientY);

    mouse.buttonPressed = true;

    mouse.dragX = mouse.gameX;
    mouse.dragY = mouse.gameY;

    ev.preventDefault();
},
```

```
touchmovehandler: function(ev) {
    mouse.insideCanvas = true;

    let touch = ev.targetTouches[0];

    mouse.setCoordinates(touch.clientX, touch.clientY);
    mouse.checkIfDragging();

    ev.preventDefault();
},
```

The touchstarthandler() method is very similar to the mousedownhandler() method. We set the insideCanvas and buttonPressed properties to true, set the mouse coordinates using setCoordinates(), and save the mouse position in dragX and dragY. The only significant difference is that we do not distinguish between left-clicks and right-clicks, and we use the first touch in the event's targetTouches array to get clientX and clientY for determining the touch coordinates.

The touchmovehandler() method also sets the insideCanvas attribute to true, calls setCoordinates(), and finally calls the checkIfDragging() method, which will set the dragSelect attribute to true if the touch is moved by more than dragSelectThreshold pixels.

If you try to run the game now with touch device emulation, you will find that you can initiate drag selection, but still cannot end it. You also cannot give any of the units any instructions. For this, we still need to implement the touchendhandler() method.

Unlike our previous game, Froot Wars, which only used left-click, this game uses both left-click and right-click. We need a way to emulate right-click on touch devices. While some devices emulate right-click using a long touch hold, this does not lend itself well to fast RTS gameplay. Instead, we will use a double-tap to act as a right-click. To do this, we will keep track of when the touch is ended, and wait for a second tap for a short time period. If the second tap does not come, we will treat it as a left-click. If however a second tap does come, we will treat it as a right-click. The touchendhandler() method will look like Listing 11-9.

Listing 11-9. The touchendhandler() Method (mouse.js)

```
doubleTapTimeoutThreshold: 300,
doubleTapTimeout: undefined,

touchendhandler: function(ev) {

    // While a typical touch device isn't likely to have a keyboard, leave this just in case
    let shiftPressed = ev.shiftKey;

    if (mouse.dragSelect) {
        // If currently drag-selecting, attempt to select items with the selection rectangle
        mouse.finishDragSelection(shiftPressed);

    } else {
        // If not dragging, wait for threshold before treating it as a left-click
        if (!mouse.doubleTapTimeout) {
            mouse.doubleTapTimeout = setTimeout(function() {
                mouse.doubleTapTimeout = undefined;
                mouse.leftClick();

            }, mouse.doubleTapTimeoutThreshold);
        } else {
```

```
            // If a second tap occurs before timeout, treat it as a double-tap
            (our approximation of a right-click)
            clearTimeout(mouse.doubleTapTimeout);
            mouse.doubleTapTimeout = undefined;
            mouse.rightClick();
        }

    }

    mouse.buttonPressed = false;

    // When a touch event ends, act as if the mouse has left the canvas
    mouse.insideCanvas = false;

    ev.preventDefault();
},
```

We start by defining a `doubleTapTimeoutThreshold` property, which is the number of milliseconds that we will wait before assuming that a tap is a left-click.

Inside the `touchendhandler()` method, we first check if the touch was part of a drag selection, and if so assume that the player just wants to finish selecting and call the `finishDragSelection()` method.

If there is no drag selection happening, and we are not waiting for a double-tap, we set a `doubleTapTimeout` timeout for 300 milliseconds, at the end of which we call the `leftClick()` method. If a touch ends while we are waiting for the double-tap, we clear `doubleTapTimeout` and call the `rightClick()` method instead.

If you run the game on a mobile device, or in your browser with mobile emulation on, you will see that you can use touch to play the game. Unit selection with dragging or tapping will work as expected, as will commanding with double-tap. Because of the way we implemented panning, you can even pan around the map by just touching the edges of the map. The sidebar buttons should work because of the mobile device click emulation, which will convert tap events to click events. Our game now supports touch events.

Finally, we will add two `meta` tags to mark the game as mobile web-app capable in the `head` section of index.html, as shown in Listing 11-10.

Listing 11-10. Adding meta Tags for Mobile (index.html)

```
<meta http-equiv="Content-type" content="text/html; charset=utf-8">

<meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
minimum-scale=1, width=device-width">
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="mobile-web-app-capable" content="yes">
```

The two newly added tags allow the game to be saved on the home screen as full-screen applications on most mobile devices. When started from the home screen, the game experience is very similar to apps installed via the app stores.

If you try the game on a mobile browser and save the game page to the home screen, you should be able to play the game full-screen, without the URL bar or the browser frame, just like an installed application. Now it's time to fix audio by enabling WebAudio.

Enabling WebAudio Support

In an attempt to prevent excessive battery usage, mobile browsers put a lot of restrictions on audio playback. The simplest way to get around most of these restrictions is to use the WebAudio API and to play a sound on the first user interaction to unlock audio playback on mobile.

To make this conversion to WebAudio effortless, we will use the same wAudio.js library that we used in Chapter 5. We will start by adding a reference to wAudio.js inside the head section of index.html as shown in Listing 11-11.

Listing 11-11. Referring to wAudio.js (index.html)

```
<script src="js/wAudio.js" type="text/javascript"></script>
```

Next, we will modify the loadSound() method inside the loader object to use wAudio if available, as shown in Listing 11-12.

Listing 11-12. Using the wAudio Object to Load Sounds (common.js)

```
loadSound: function(url) {
    this.loaded = false;
    this.totalCount++;

    game.showScreen("loadingscreen");

    var audio = new (window.wAudio || Audio)();

    audio.addEventListener("canplaythrough", loader.itemLoaded, false);
    audio.src = url + loader.soundFileExtn;

    return audio;
},
```

The loadSound() method should now use the wAudio object if available, or fall back to the default HTML5 audio if not. Our game audio should now work on most mobile devices, but we still need to unlock the audio on a user interaction before it can work properly on iOS devices. Since this is such a common requirement, wAudio.js has two ways to do so: the wAudio.playMutedSound() method, which we can call from any tap or click event, and the wAudio.mobileAutoEnable property, which will automatically unlock the audio on the first user interaction. We will set the wAudio.mobileAutoEnable property to true inside game.init(), as shown in Listing 11-13.

Listing 11-13. Setting mobileAutoEnable in game.init() (common.js)

```
// Start initializing objects, preloading assets, and display start screen
init: function() {
    // Initialize objects
    loader.init();
    mouse.init();
    sidebar.init();
    sounds.init();
```

```
    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // If wAudio has been added, automatically enable mobile audio on the first user touch
    event
    if (window.wAudio) {
        wAudio.mobileAutoEnable = true;
    }

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

Again, we check for the presence of the wAudio object, and if present, we set its mobileAutoEnable property to true. Behind the scenes, the wAudio object will wait for the first touch event anywhere on the document, and then play a muted sound to unlock audio playback on the device automatically. If you run the game on a mobile device, you should now see that the game audio works as expected on all mobile devices.

Keep in mind that since wAudio uses XMLHttpRequest, the game will now have to be hosted on a web server to work around the browser's security feature that prevents access to local files. If you find hosting on a web server inconvenient during development, you can comment out the wAudio.js script tag inside index.html, and the code should fall back to using HTML5 audio as before. When you are finished with game development and ready to release the game, you can uncomment the line to re-enable the wAudio.js library.

If you find this library useful, you can read more about it, as well as find the latest code, at https://github.com/adityaravishankar/wAudio.js. This library is MIT licensed, so you can feel free to use it any of your own game projects.

Now that our game works on mobile devices, it's time to start building the actual levels for our single-player campaign.

Building the Single-Player Campaign

We will build three levels in our game campaign. Each of the levels will get progressively harder, while building upon the story from the previous levels. These levels will illustrate the typical types of levels you would find in an RTS game.

Before we begin, we will first modify our game map to use the non-debug version of the map image as shown in Listing 11-14.

Listing 11-14. Removing the Debug Grid (levels.js)

```
/* Details of the maps used by the levels */
var maps = {
    "plains": {
        "mapImage": "plains.png",

        /* Terrain Data - Auto Generated By level/convert-levels.js */
        "mapGridWidth": 60,
        "mapGridHeight": 40,
        "mapObstructedTerrain": [[0, 0], [1, 0], [2, 0], [26, 0], [27, 0], /* Extremely huge
        array snipped for brevity */ [58, 39], [59, 39]],
    }
};
```

If you run the game now, you should no longer see the grid on top of the map. Now it's time to create the first level of our campaign: Rescue.

The Rescue

The introductory level in our game will be a relatively easy mission so that the player can get comfortable with moving units around the map and attacking enemy units.

The player will need to navigate across a map populated with easily defeated enemies and then escort a convoy of transport vehicles back to that player's starting location. After the mission briefing, we will move the story line forward using character dialogue that is triggered by timed and conditional triggers.

We will start with a completely fresh level object inside levels.js, as shown in Listing 11-15.

Listing 11-15. Creating the First Level (levels.js)

```
{
    "name": "Rescue",
    "briefing": "In the months since the great war, mankind has fallen into chaos. Billions
    are dead with cities in ruins.\nSmall groups of survivors band together to try and
    survive as best as they can.\nWe are trying to reach out to all the survivors in this
    sector before we join back with the main colony.",

    /* Map Details */
    "mapName": "plains",
    "startX": 36,
    "startY": 0,

    /* Entities to be loaded */
    "requirements": {
        "buildings": ["base"],
        "vehicles": ["transport", "scout-tank", "heavy-tank"],
        "aircraft": [],
        "terrain": []
    },

    /* Entities to be added */
    "items": [
        /* Slightly damaged base */
        { "type": "buildings", "name": "base", "x": 55, "y": 6, "team": "blue", "life": 100 },

        /* Our hero tank */
        { "type": "vehicles", "name": "heavy-tank", "uid": -1, "x": 57, "y": 12,
        "direction": 4, "team": "blue" },

        /* Two transport vehicles waiting just to be rescued just outside the visible map */
        { "type": "vehicles", "name": "transport", "uid": -3, "selectable": false, "x": -3,
        "y": 2, "direction": 2, "team": "blue" },
        { "type": "vehicles", "name": "transport", "uid": -4, "selectable": false, "x": -3,
        "y": 4, "direction": 2, "team": "blue" },
```

```
        /* Two damaged enemy scout-tanks patroling the area*/
        { "type": "vehicles", "name": "scout-tank", "uid": -2, "x": 40, "y": 20,
        "direction": 4, "team": "green", "life": 20, "orders": { "type": "patrol", "from": {
        "x": 34, "y": 20 }, "to": { "x": 42, "y": 25 } } },
        { "type": "vehicles", "name": "scout-tank", "uid": -5, "x": 14, "y": 0, "direction": 4,
        "team": "green", "life": 20, "orders": { "type": "patrol", "from": { "x": 14, "y": 0 },
        "to": { "x": 14, "y": 14 } } },

    ],

    "cash": {
        "blue": 0,
        "green": 0
    },

    /* Conditional and Timed Trigger Events */
    "triggers": [
        {
            "type": "timed", "time": 3000,
            // Tell the player to search for the convoy
            "action": function() {
                game.showMessage("op", "Commander!! We haven't heard from the last convoy in
                over two hours. They should have arrived by now.");
            }
        },
        {
            "type": "timed", "time": 10000,
            // Give player hint to help find the convoy
            "action": function() {
                game.showMessage("op", "They were last seen in the North West Sector. Could
                you investigate?");
            }
        },
        {
            "type": "conditional",
            // Check if either the hero tank or the two convoy vehicles are dead
            "condition": function() {
                return (game.isItemDead(-1) || game.isItemDead(-3) || game.isItemDead(-4));
            },
            // End the mission as failure
            "action": function() {
                singleplayer.endLevel(false);
            }
        },
    ],
},
```

The first portion of the level consists of the same basic metadata that we saw in earlier levels. We start with the mission briefing, which gives the player a little background on the level. We also set the starting position to the top-right corner of the map. Next we load a few essential items in the requirements array and set the starting cash balance for both players to 0.

Within the level's items array, we add a damaged base, a heavy tank that the player will control, two enemy scout tanks that patrol the area, and two transports. We set UIDs for each of these units so that we can refer to them from the triggers.

Since this is the first level, we reduce the life of the enemy scout tanks so the player will find it easy to destroy them. The transports are positioned slightly outside the bounds of the top-left corner of the map so that they do not become visible to the player until the right time.

Within the triggers array, we define our first few triggers. The first two timed triggers show the player a message from the operator, asking them to find the missing transport. The third is a conditional trigger that will end the mission as a failure if either the transports or the heavy tank get destroyed by using the isItemDead() method.

Next, we will add a few new characters to the characters object inside the game object, as shown in Listing 11-16.

Listing 11-16. Adding New Characters (game.js)

```
// Profile pictures for game characters
characters: {
    "system": {
        "name": "System Control",
        "image": "system.png"
    },
    "op": {
        "name": "Operator",
        "image": "girl1.png"
    },
    "pilot": {
        "name": "Pilot",
        "image": "girl2.png"
    },
    "driver": {
        "name": "Driver",
        "image": "man1.png"
    }
},
```

■ **Note**　These new character images are Creative Commons–licensed artwork found at
http://opengameart.org.

We will also define the isItemDead() method inside game.js, as shown in Listing 11-17.

Listing 11-17. The isItemDead() Method (game.js)

```
isItemDead: function(uid) {
    let item = game.getItemByUid(uid);

    return !item || item.lifeCode === "dead";
}
```

We consider an item dead if we can no longer find it in the game.items array or if its lifeCode property is set to dead.

If you run the game so far, you should see the operator giving you your first mission task by asking you to investigate the situation, as shown in Figure 11-1.
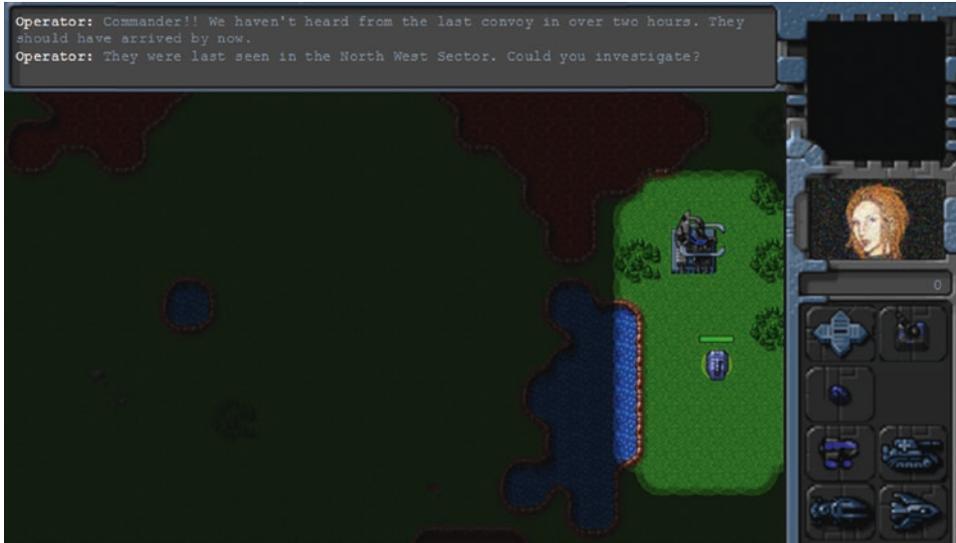
Figure 11-1. *The first mission task*

You should be able to select the tank and move it around, with the fog of war slowly clearing up as you explore the map.

Now, we will introduce the enemy and the convoy by adding a few more triggers to the first level, as shown in Listing 11-18.

Listing 11-18. Introducing the Enemy and the Convoy (levels.js)

```
{
    "type": "conditional",
    // Check if first enemy is dead
    "condition": function() {
        return game.isItemDead(-2);
    },
    // Make a comment about the rebel aggression
    "action": function() {
        game.showMessage("op", "The rebels have been getting very aggressive lately. I hope
        the convoy is safe. Find them and escort them back to the base.");
    }
},
{
    "type": "conditional",
    // Check if hero has reached the top-left quadrant of the map
    "condition": function() {
        let hero = game.getItemByUid(-1);

        return (hero && hero.x < 30 && hero.y < 30);
    },
```

```
    // Display distress call from the driver
    "action": function() {
        game.showMessage("driver", "Can anyone hear us? Our convoy has been pinned down by
        rebel tanks. We need help.");
    }
},
```

In the first conditional trigger, we show a message from the operator discussing the rebels once the first enemy scout tank is destroyed. In the second conditional trigger, we show a message from the convoy driver once we enter the top-left corner of the map.

If we run the game now, we should see the operator urging us to hurry after the first fight with the rebels and the convoy driver calling for help when we approach the convoy location, as shown in Figure 11-2.
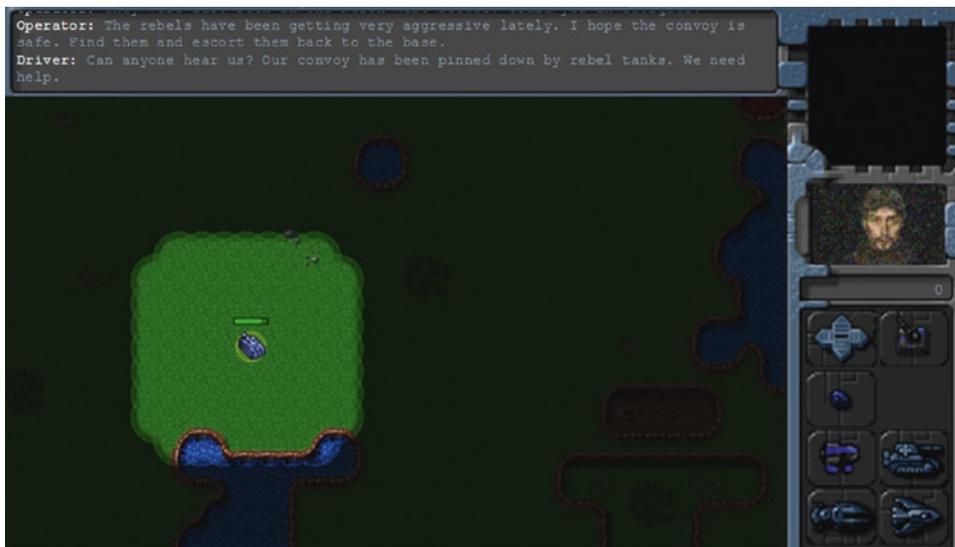
Figure 11-2. *Convoy driver asking for help*

Finally, we will add a few triggers to implement rescuing the convoy and completing the mission, as shown in Listing 11-19.

Listing 11-19. Rescuing the Convoy and Completing the Mission (levels.js)

```
{
    "type": "conditional",
    // Check if player is near convoy location
    "condition": function() {
        let hero = game.getItemByUid(-1);

        return (hero && hero.x < 10 && hero.y < 10);
    },
```

335

```
    // Show thank you message from driver and tell convoy to follow hero
    "action": function() {
        var hero = game.getItemByUid(-1);

        game.showMessage("driver", "Thank you. We thought we would never get out of here
        alive.");
        game.sendCommand([-3, -4], { type: "guard", to: hero });
    }
},
{
    "type": "conditional",
    // Check if convoy vehicles are near the base
    "condition": function() {
        var transport1 = game.getItemByUid(-3);
        var transport2 = game.getItemByUid(-4);

        return (transport1 && transport2 && transport1.x > 52 && transport2.x > 52 &&
        transport2.y < 18 && transport1.y < 18);
    },
    // End the mission as success
    "action": function() {
        singleplayer.endLevel(true);
    }
},
```

In the first conditional trigger, we show another message from the driver when the hero tank reaches the top-left corner of the map. We then command both the transports to guard the tank, which means they will follow the tank wherever it goes.

In the second conditional trigger, we end the game once both the transports reach the top-right corner of the map where the base is located.

If you run the game now, you should see the convoy driver thank you for saving the convoy and then follow you back to the base, as shown in Figure 11-3.
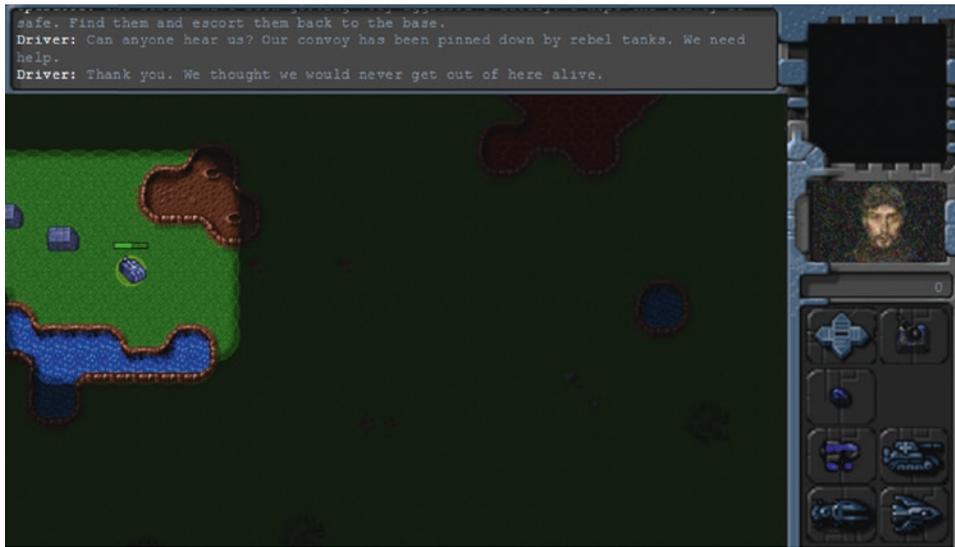
Figure 11-3. Rescuing the convoy

The return journey should be uneventful since all the enemies in this level are dead. Once the two transports reach the base, the mission will end. You have just completed your first mission in the single-player campaign.

Now it's time to make the next level, Assault.

Assault

The second level in our game will be a little more challenging than the first one. This time we will introduce the player to the idea of micromanaging units to attack the enemy, without having to worry about managing resources or the production of units.

Players will be provided a steady stream of reinforcements that they will need to use to locate and capture a small enemy base across the map. The enemy base will keep sending out steady waves of attacking units to make the mission more challenging.

We will create a new level object inside the singleplayer array of the levels object, as shown in Listing 11-20. This new level will automatically be loaded once the first mission has been completed.

Listing 11-20. Creating the Second Level (levels.js)

```
{
    "name": "Assault",
    "briefing": "Thanks to the supplies from the convoy, we now have the base up and running.
    \n The rebels nearby are proving to be a problem. We need to take them out.
    \n First set up the base defenses. Then find and destroy all rebels in the area.
    \n The colony will be sending us reinforcements to help us out.",

    /* Map Details */
    "mapName": "plains",
    "startX": 36,
    "startY": 0,
```

337

```
    /* Entities to be loaded */
    "requirements": {
        "buildings": ["base", "ground-turret", "starport", "harvester"],
        "vehicles": ["transport", "scout-tank", "heavy-tank"],
        "aircraft": ["chopper"],
        "terrain": []
    },

    /* Economy Related*/
    "cash": {
        "blue": 0,
        "green": 0
    },

    /* Entities to be added */
    "items": [
        { "type": "buildings", "name": "base", "uid": -1, "x": 55, "y": 6, "team": "blue" },

        { "type": "buildings", "name": "ground-turret", "x": 53, "y": 17, "team": "blue" },
        { "type": "vehicles", "name": "heavy-tank", "uid": -2, "x": 55, "y": 16,
        "direction": 4, "team": "blue", "orders": { "type": "sentry" } },

        /* The first wave of attacks*/
        { "type": "vehicles", "name": "scout-tank", "x": 55, "y": 36, "direction": 4,
        "team": "green", "orders": { "type": "hunt" } },
        { "type": "vehicles", "name": "scout-tank", "x": 53, "y": 36, "direction": 4,
        "team": "green", "orders": { "type": "hunt" } },

        /* Enemies patroling the area */
        { "type": "vehicles", "name": "scout-tank", "x": 5, "y": 5, "direction": 4,
        "team": "green", "orders": { "type": "patrol", "from": { "x": 5, "y": 5 },
        "to": { "x": 20, "y": 20 } } },
        { "type": "vehicles", "name": "scout-tank", "x": 5, "y": 15, "direction": 4,
        "team": "green", "orders": { "type": "patrol", "from": { "x": 5, "y": 15 },
        "to": { "x": 20, "y": 30 } } },
        { "type": "vehicles", "name": "scout-tank", "x": 25, "y": 5, "direction": 4,
        "team": "green", "orders": { "type": "patrol", "from": { "x": 25, "y": 5 },
        "to": { "x": 25, "y": 20 } } },
        { "type": "vehicles", "name": "scout-tank", "x": 35, "y": 5, "direction": 4,
        "team": "green", "orders": { "type": "patrol", "from": { "x": 35, "y": 5 },
        "to": { "x": 35, "y": 30 } } },

        /* The Evil Rebel Base*/
        { "type": "buildings", "name": "base", "uid": -11, "x": 5, "y": 36, "team": "green" },
        { "type": "buildings", "name": "starport", "uid": -12, "x": 1, "y": 30,
        "team": "green" },
        { "type": "buildings", "name": "starport", "uid": -13, "x": 4, "y": 32,
        "team": "green" },

        { "type": "buildings", "name": "harvester", "x": 1, "y": 38, "team": "green",
        "action": "deploy" },
```

```
        { "type": "buildings", "name": "ground-turret", "x": 5, "y": 28, "team": "green" },
        { "type": "buildings", "name": "ground-turret", "x": 7, "y": 33, "team": "green" },
        { "type": "buildings", "name": "ground-turret", "x": 8, "y": 37, "team": "green" },
    ],

    /* Conditional and Timed Trigger Events */
    "triggers": [
        {
            "type": "timed", "time": 8000,
            // Send in reinforcements to guard the hero tank from the first enemy wave
            "action": function() {
                game.showMessage("op", "Commander!! Reinforcements have arrived from the
                colony.");
                let hero = game.getItemByUid(-2);

                game.add({
                    "type": "vehicles",
                    "name": "scout-tank",
                    "team": "blue",
                    "x": 61, "y": 22,
                    "orders": { "type": "guard", "to": hero }
                });
                game.add({
                    "type": "vehicles",
                    "name": "scout-tank",
                    "team": "blue",
                    "x": 61, "y": 21,
                    "orders": { "type": "guard", "to": hero }
                });
            }
        },
        {
            "type": "timed", "time": 25000,
            // Supply extra cash
            "action": function() {
                game.cash["blue"] = 1500;
                game.showMessage("op", "Commander!! We have enough resources for another
                ground turret. Set up the turret to keep the base safe from any more
                attacks.");
            }
        },
    ]
},
```

The first portion of the level has nearly the same metadata as the previous level. We are reusing the map from level 1. The only thing that changes is the mission briefing.

Next, we load all the essential items in the requirements array, and set the starting cash balance for both players to 0.

This time, we add a lot more items in the level's items array. We start with the base, a heavy tank that the player will control, and a ground turret to protect the base.

Next, we add two enemy scout tanks that are set to hunt mode so they will attack our base as soon as the game starts. We then add several more enemy scout tanks that are set to patrol around the map.

Finally, we add an enemy base that has two starports and a refinery. It is also well defended with several ground turrets and scout tanks patrolling nearby.

The `triggers` array contains two timed triggers that are set off within the first few seconds of the game. Within the first trigger, we add two friendly scout tanks to the game and notify the player that reinforcements have arrived.

In the second trigger, we give players 1,500 credits and tell them they have enough resources to build one ground turret. Placing this turret will be the only sidebar-related task players will perform in this game.

If you run the game, you will find that the second level starts in a much more exciting way than the first level. You will see that the base is under attack within the first few seconds and reinforcements arrive in the nick of time to save you, as shown in Figure 11-4.

Figure 11-4. *Saved by reinforcements*

Once the attack has been stopped, the operator will notify you that you have enough resources to build one more turret.

Now we will add a few more triggers to add waves of attacking units and reinforcements, as shown in Listing 11-21.

Listing 11-21. Adding Enemy Waves and Reinforcements (levels.js)

```
{
    "type": "timed", "time": 60000, "repeat": true,
    // Construct a couple of bad guys to hunt the player every time enemy has enough money
    "action": function() {
        if (game.cash["green"] > 1000) {
            game.sendCommand([-12, -13], {
                type: "construct-unit",
                details: { type: "vehicles", name: "scout-tank", orders: { "type": "hunt" } }
            });
        }
    }
},
```

```
{
    "type": "timed", "time": 180000, "repeat": true,
    // Send in more reinforcements every three minutes
    "action": function() {
        game.showMessage("op", "Commander!! More reinforcements have arrived.");
        game.add({
            "type": "vehicles",
            "name": "scout-tank",
            "team": "blue",
            "x": 61, "y": 22,
            "orders": { "type": "move", "to": { "x": 55, "y": 21 } }
        });
        game.add({
            "type": "vehicles",
            "name": "heavy-tank",
            "team": "blue",
            "x": 61, "y": 23,
            "orders": { "type": "move", "to": { "x": 56, "y": 23 } }
        });
    }
},
```

In the first timed trigger, we check whether the green player has enough money every 60 seconds, and whenever the green player does, we construct a couple of scout tanks in hunt mode. In the second timed trigger, we send the hero two reinforcing units every 180 seconds.

Unlike the first level, the enemy has a lot more units and turret defenses. A direct frontal assault on the enemy base will not work because the player is likely to lose all the units. The player also cannot wait too long since the enemy will keep sending out waves of enemies every few minutes.

A player's best strategy will be to make small attacks, chipping away at the opposition and then falling back to the base for reinforcements until ready to make the final attack.

Finally, we will add triggers to provide the player with air support and to complete the mission, as shown in Listing 11-22.

Listing 11-22. Adding Air Support and Completing the Mission (levels.js)

```
{
    "type": "timed", "time": 600000,
    // Send in air support if the mission hasn't finished after 10 minutes
    "action": function() {
        game.showMessage("pilot", "Close Air Support en route. Will try to do whatever I can
        to help.");
        game.add({
            "type": "aircraft",
            "name": "chopper",
            "team": "blue",
            "selectable": false,
            "x": 61, "y": 22,
            "orders": { "type": "hunt" }
        });
    }
},
```

```
{
    "type": "conditional",
    // Check if the player's base has been destroyed
    "condition": function() {
        return game.isItemDead(-1);
    },
    // End level as failure
    "action": function() {
        singleplayer.endLevel(false);
    }
},

{
    "type": "conditional",
    // Check if the enemy base is at least half destroyed
    "condition": function() {
        let enemyBase = game.getItemByUid(-11);

        return (!enemyBase || (enemyBase.life <= enemyBase.hitPoints / 2));
    },
    // End level as success
    "action": function() {
        singleplayer.endLevel(true);
    }
},
```

In the first timed trigger, we release a friendly chopper in hunt mode, ten minutes after the game starts. The next two conditional triggers set the conditions for successfully completing or failing the mission.

If we run the game now, we should see the chopper pilot coming in to give us a helping hand after some time, as shown in Figure 11-5.

Figure 11-5. *Pilot flying in chopper for air support*

If you have trouble completing the mission, the extra air support should help. Again, we introduce a new character, the pilot, who will stay with us for the next mission. With the assistance of the chopper, we can capture the enemy base and complete the mission so we can go on to the final mission.

Now it's time to build our final mission: Under Siege.

Under Siege

The final level in our game will be the most challenging. The player will need to constantly build units to fight off several waves of enemy units.

This time, the player will take over the enemy base captured after the last mission. The player will be provided some initial supplies to help get started. After that, the player will need to fend off several waves of units and protect the transport vehicles filled with refugees until the colony reinforcements arrive to help them.

We will create a new level object inside the singleplayer array of the levels object, as shown in Listing 11-23. This new level will automatically be loaded once the second mission has been completed.

Listing 11-23. Creating the Third Level (levels.js)

```
{
    "name": "Under Siege",
    "briefing": "Thanks to the attack led by you, we now have control of the rebel base. We
    can expect the rebels to try to retaliate.\n The colony is sending in aircraft to help
    us evacuate back to the main camp. All we need to do is hang tight until the choppers
    get here. \n Luckily, we have some supplies and ammunition to defend ourselves with
    until they get here. \n Protect the transports at all costs.",

    /* Map Details */
    "mapName": "plains",
    "startX": 0,
    "startY": 20,

    /* Entities to be loaded */
    "requirements": {
        "buildings": ["base", "ground-turret", "starport", "harvester"],
        "vehicles": ["transport", "scout-tank", "heavy-tank"],
        "aircraft": ["chopper", "wraith"],
        "terrain": []
    },

    /* Economy Related*/
    "cash": {
        "blue": 200,
        "green": 0
    },

    /* Entities to be added */
    "items": [
        /* The Rebel Base, which is now in our hands */
        { "type": "buildings", "name": "base", "uid": -11, "x": 5, "y": 36, "team": "blue" },
        { "type": "buildings", "name": "starport", "uid": -12, "x": 1, "y": 28,
        "team": "blue" },
```

```
      { "type": "buildings", "name": "starport", "uid": -13, "x": 4, "y": 32,
      "team": "blue" },
      { "type": "buildings", "name": "harvester", "x": 1, "y": 38, "team": "blue",
      "action": "deploy" },
      { "type": "buildings", "name": "ground-turret", "x": 7, "y": 28, "team": "blue" },
      { "type": "buildings", "name": "ground-turret", "x": 8, "y": 32, "team": "blue" },
      { "type": "buildings", "name": "ground-turret", "x": 11, "y": 37, "team": "blue" },

      /* The transports that need to be protected*/
      { "type": "vehicles", "name": "transport", "uid": -1, "x": 2, "y": 33, "team":
      "blue", "direction": 2, "selectable": false },
      { "type": "vehicles", "name": "transport", "uid": -2, "x": 1, "y": 34, "team":
      "blue", "direction": 2, "selectable": false },
      { "type": "vehicles", "name": "transport", "uid": -3, "x": 2, "y": 35, "team":
      "blue", "direction": 2, "selectable": false },
      { "type": "vehicles", "name": "transport", "uid": -4, "x": 1, "y": 36, "team":
      "blue", "direction": 2, "selectable": false },

      /* The chopper pilot from the last mission */
      {
          "type": "aircraft",
          "name": "chopper",
          "x": 15, "y": 40,
          "team": "blue",
          "selectable": false,
          "uid": -5,
          "orders": { "type": "patrol", "from": { "x": 15, "y": 40 }, "to": { "x": 0,
          "y": 25 } }
      },

      /* The first wave of attacks*/
      { "type": "vehicles", "name": "scout-tank", "x": 15, "y": 16, "direction": 4,
      "team": "green", "orders": { "type": "hunt" } },
      { "type": "vehicles", "name": "scout-tank", "x": 17, "y": 16, "direction": 4,
      "team": "green", "orders": { "type": "hunt" } },

      /* Secret Rebel bases*/
      { "type": "buildings", "name": "starport", "uid": -23, "x": 35, "y": 37,
      "team": "green" },
      { "type": "buildings", "name": "starport", "uid": -24, "x": 33, "y": 37,
      "team": "green" },
      { "type": "buildings", "name": "harvester", "x": 28, "y": 39, "team": "green",
      "action": "deploy" },
      { "type": "buildings", "name": "harvester", "x": 30, "y": 39, "team": "green",
      "action": "deploy" },

      { "type": "buildings", "name": "starport", "uid": -21, "x": 3, "y": 0,
      "team": "green" },
      { "type": "buildings", "name": "starport", "uid": -22, "x": 6, "y": 0,
      "team": "green" },
```

```
            { "type": "buildings", "name": "harvester", "x": 0, "y": 2, "team": "green",
            "action": "deploy" },
            { "type": "buildings", "name": "harvester", "x": 0, "y": 4, "team": "green",
            "action": "deploy" },

        ],

        /* Conditional and Timed Trigger Events */
        "triggers": [
            {
                // Check if any of the transports is dead
                "condition": function() {
                    return game.isItemDead(-1) || game.isItemDead(-2) || game.isItemDead(-3) ||
                    game.isItemDead(-4);
                },
                // End the level as failure
                "action": function() {
                    singleplayer.endLevel(false);
                }
            },
            {
                "type": "timed", "time": 5000,
                // Display warning message about attacks
                "action": function() {
                    game.showMessage("op", "Commander!! The rebels have started attacking. We
                    need to defend the base and protect the transports at all costs.");
                }
            },
        ],
}
```

Again, we are reusing the map from level 1. The first portion of the level has nearly the same metadata as the previous levels. The only thing that changes is the mission briefing.

Next we load all the essential items in the `requirements` array and set the starting cash balance for both players.

This time, we add a lot more items to the map. First, we re-create the entire enemy base, but for the player team. Next, we add several transport vehicles that we will be protecting in this mission and add the chopper from the last mission on patrol mode to protect the base. Next, we add a few enemy units for a first wave of attacks. Finally, we define several starports and refineries for two secret enemy bases that will be attacking the player.

Within the triggers, we define one conditional trigger that ends the mission in case even one of the transports dies. The second timed trigger just displays a message from the operator.

If we run the game and start the third level, we should see the rebels attacking the base and the patrolling chopper fending them off, as shown in Figure 11-6.

Figure 11-6. *Patrolling chopper defending the base*

Now that the first wave of attacks has been fended off, we will build a little drama with a small cinematic story line within the mission by adding a few creative triggers to the map, as shown in Listing 11-24.

Listing 11-24. Adding a Little Drama to the Level (levels.js)

```
{
    "type": "timed", "time": 20000,
    // Add a new transport to the top right of the map
    "action": function() {
        game.add({ "type": "vehicles", "name": "transport", "x": 57, "y": 3, "team": "blue",
        "direction": 4, "selectable": false, "uid": -6 });
        game.showMessage("driver", "Commander!! The colony has sent some extra supplies. We
        are coming in from the North East sector through rebel territory. We could use a
        little protection.");
    }
},
{
    "type": "timed", "time": 24000,
    // Make the pilot guard the new transport
    "action": function() {
        game.sendCommand([-5], { "type": "guard", "toUid": -6 });
        game.showMessage("pilot", "Hang tight. I'm on my way.");
    }
},

{
    "type": "timed", "time": 28000,
    // Add some villains to make it interesting
    "action": function() {
```

```
            game.add({ "type": "vehicles", "name": "scout-tank", "x": 57, "y": 28,
            "team": "green", "orders": { "type": "hunt" } });
            game.add({ "type": "aircraft", "name": "wraith", "x": 55, "y": 33, "team": "green",
            "orders": { "type": "sentry" } });
            game.add({ "type": "aircraft", "name": "wraith", "x": 53, "y": 33, "team": "green",
            "orders": { "type": "sentry" } });
            game.add({ "type": "vehicles", "name": "scout-tank", "x": 35, "y": 25, "life": 20,
            "direction": 4, "team": "green", "orders": { "type": "patrol", "from": { "x": 35,
            "y": 25 }, "to": { "x": 35, "y": 30 } } });
        }
    },
    {
        "type": "timed", "time": 48000,
        // Start moving the transport toward the base
        "action": function() {
            game.showMessage("driver", "Thanks! Appreciate the backup. All right. Off we go.");
            game.sendCommand([-6], { "type": "move", "to": { "x": 0, "y": 32 } });
        }
    },
    {
        "type": "conditional",
        // Check if pilot has been hurt
        "condition": function() {
            let pilot = game.getItemByUid(-5);

            return pilot.life < pilot.hitPoints;
        },
        // Have pilot ask for help
        "action": function() {
            game.showMessage("pilot", "We are under attack! Need assistance. This doesn't look
            good.");
        }
    },
    { "type": "conditional",
        // Check if new transport has reached base with supplies
        "condition": function() {
            let driver = game.getItemByUid(-6);

            return driver && driver.x < 2 && driver.y > 30;
        },
        // Give player extra cash "supplies"
        "action": function() {
            game.showMessage("driver", "The rebels came out of nowhere. There was nothing we
            could do. She saved our lives. Hope these supplies were worth it.");
            game.cash["blue"] += 1200;
        }
    },
```

In the first timed trigger, the driver from the first mission asks for assistance while standing at the top-right corner of the map.

In the second timed trigger, the pilot announces she is on her way. We then command the pilot to guard the transport.

In the third timed trigger, we also add several enemy units to the map.

In the fourth timed trigger, which will be set off around the time the pilot arrives at the transport's location, we command the transport to start moving toward the base.

In the fifth conditional trigger—which is set off if the pilot's chopper gets attacked—the pilot messages asking for help.

In the final conditional trigger, which is set off once the transport reaches its destination, the driver talks about his experience, and the player's cash resources are increased.

If you run the game now, you should see a fairly interesting scene play out. You will see the pilot going out to help the driver when he calls for help. The pilot then protects the transport before getting ambushed by several enemy aircraft. The transport continues to drive toward the base while under enemy fire. Once the driver reaches the base, he provides the player with some supplies and describes the experience, as shown in Figure 11-7.
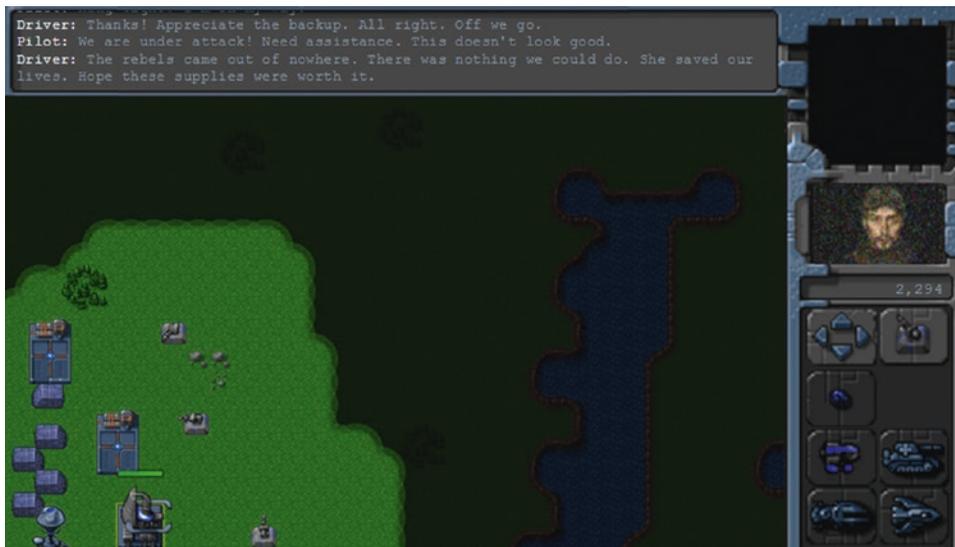
Figure 11-7. *Driver describes the ordeal after getting back*

At the end of the whole experience, the player now has some extra cash to start building units.

Even though the story in our game is a little rushed, as you can see, this trigger mechanism can be used to tell a fairly interesting story. The game framework can also be extended to use a combination of video, audio, or animated GIFs to make the experience even more immersive.

Now let's add a trigger to set up the waves of enemy units, as shown in Listing 11-25.

Listing 11-25. Adding Enemy Waves (levels.js)

```
{
    "type": "timed", "time": 150000, "repeat": true,
    // Send in waves of enemies every 150 seconds
    "action": function() {
        // Count aircraft and tanks already available to bad guys
        let wraithCount = 0;
```

```
let chopperCount = 0;
let scoutTankCount = 0;
let heavyTankCount = 0;

game.items.forEach(function(item) {
    if (item.team === "green") {
        switch (item.name) {
            case "chopper":
                chopperCount++;
                break;
            case "wraith":
                wraithCount++;
                break;
            case "scout-tank":
                scoutTankCount++;
                break;
            case "heavy-tank":
                heavyTankCount++;
                break;
        }
    }
}, this);

// Make sure enemy has at least two wraiths and two heavy tanks, and use the
remaining starports to build choppers and scouts
if (wraithCount === 0) {
    // No wraiths alive. Ask both starports to make wraiths
    game.sendCommand([-23, -24], { type: "construct-unit", details: { type:
    "aircraft", name: "wraith", "orders": { "type": "hunt" } } });
} else if (wraithCount === 1) {
    // One wraith alive. Ask starports to make one wraith and one chopper
    game.sendCommand([-23], { type: "construct-unit", details: { type: "aircraft",
    name: "wraith", "orders": { "type": "hunt" } } });
    game.sendCommand([-24], { type: "construct-unit", details: { type: "aircraft",
    name: "chopper", "orders": { "type": "hunt" } } });
} else {
    // Two wraiths alive. Ask both starports to make choppers
    game.sendCommand([-23, -24], { type: "construct-unit", details: { type:
    "aircraft", name: "chopper", "orders": { "type": "hunt" } } });
}

if (heavyTankCount === 0) {
    // No heavy-tanks alive. Ask both starports to make heavy-tanks
    game.sendCommand([-21, -22], { type: "construct-unit", details: { type:
    "vehicles", name: "heavy-tank", "orders": { "type": "hunt" } } });
} else if (heavyTankCount === 1) {
// One heavy-tank alive. Ask starports to make one heavy-tank and one scout-tank
    game.sendCommand([-21], { type: "construct-unit", details: { type: "vehicles",
    name: "heavy-tank", "orders": { "type": "hunt" } } });
    game.sendCommand([-22], { type: "construct-unit", details: { type: "vehicles",
    name: "scout-tank", "orders": { "type": "hunt" } } });
```

```
        } else {
            // Two heavy-tanks alive. Ask both starports to make scout-tanks
            game.sendCommand([-21, -22], { type: "construct-unit", details: { type:
            "vehicles", name: "scout-tank", "orders": { "type": "hunt" } } });
        }
        // Ask any enemy units on the field to attack
        let uids = [];

        game.items.forEach(function(item) {
            if (item.team === "green" && item.canAttack) {
                uids.push(item.uid);
            }
        }, this);

        game.sendCommand(uids, { "type": "hunt" });
    }
},
```

Unlike the previous level, the enemy waves trigger is a little more intelligent. The timed trigger runs every 150 seconds. It first counts the number of enemy units of each type. It then decides which units to build based on what units are available. In this simple example, we first make sure that the green team has at least two wraiths to control the sky and two heavy tanks to control the ground, and if not, we build them at the starports. We build choppers and scout tanks on any remaining starports. Finally, we command all green team units that can attack to go in hunt mode.

If you run the game now, the enemy will send out waves every few minutes. This time, the composition of the enemy units will vary with each attack. If you don't plan your defense properly, you can expect to get overwhelmed by the enemy.

This AI can be improved further by tweaking the enemy composition based on the player's composition or selecting the targets to attack intelligently. However, as you can see, even this simple set of instructions provides us with a fairly challenging enemy.

Now that we have a challenging enemy in the level, we will implement triggers for ending the mission, as shown in Listing 11-26.

Listing 11-26. Implementing the Ending (levels.js)

```
{
    "type": "timed", "time": 480000,
    // After 8 minutes, start preparing for the end
    "action": function() {
        game.showMessage("op", "Commander!! The colony air fleet is just a few minutes away.");
    }
},
{
    "type": "timed", "time": 600000,
    //After 10 minutes send in reinforcements
    "action": function() {
        game.showMessage("op", "Commander!! The colony air fleet is approaching");
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 28, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 29, "team": "blue",
        "orders": { "type": "hunt" } });
```

```javascript
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 30, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 31, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 32, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 33, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 34, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 35, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 36, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 37, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "wraith", "x": -1, "y": 38, "team": "blue",
        "orders": { "type": "hunt" } });
        game.add({ "type": "aircraft", "name": "chopper", "x": -1, "y": 39, "team": "blue",
        "orders": { "type": "hunt" } });
    }
},
{
    "type": "timed", "time": 660000,
    // And a minute after the reinforcements arrive, end the level
    "action": function() {
        singleplayer.endLevel(true);
    }
},
```

The first trigger, eight minutes into the game, is the operator announcing that the colony air fleet has almost arrived. In the second trigger, two minutes later, we add an entire fleet of friendly aircraft in hunt mode. Finally, one minute after the fleet arrival we end the level.

The only goal of this mission is for the player to survive and protect the transport until the fleet arrives. If you play the mission now and survive long enough, you should see the large fleet flying in and destroying the enemy, as shown in Figure 11-8.

Figure 11-8. *The colony air fleet flying in to save the day*

Once the fleet flies in, we have an extra minute to enjoy watching them attack and destroy the enemy before completing the last level in our single-player campaign.

Summary

In this chapter, we completed the entire single-player campaign of our RTS game. We started by adding sound to the game. We then added support for mobile devices by listening for touch events and used the wAudio.js library to switch to Web Audio.

Finally, we used the framework that we built over the past few chapters to develop several levels for the campaign. We looked at ways to make the levels challenging and interesting by creatively using triggered events. We also saw how we could weave a complete story into the game using triggered events combined with the game's messaging framework.

At this point, you have a complete, working, single-player RTS game that you can either extend or use for your own ideas. A good way to go forward from here is to try developing your own interesting levels for this campaign.

After that, if you are ready for something more challenging, you should try building your own game. You can use this code to prototype new game ideas fairly quickly by just modifying the artwork and adjusting the settings. If the feedback on your prototype is encouraging, you can then invest more time and effort to build a complete game.

Of course, while playing against the computer is fun, it can be a lot more fun to challenge your friends. In the next two chapters, we will look at the HTML5 WebSocket API and how we can use it to build a multiplayer game so you can play against your friends over the network. So, once you have spent some time enjoying the single-player game, proceed on to the next chapter so that we can get started with multiplayer.

■ ■ ■ ■

Multiplayer with WebSockets

No matter how challenging we make a single-player game, it will always lack the challenge of competing against another human being. Multiplayer games allow players to either compete against each other or work cooperatively toward a common goal.

Now that we have a working single-player campaign, we will look at how we can add multiplayer support to our RTS game by using the HTML5 WebSocket API.

Before we start adding multiplayer to our game, let's first take a look at some networking basics using the WebSocket API with Node.js.

Using the WebSocket API with Node.js

The heart of our multiplayer game is the HTML5 WebSocket API. Before WebSockets came along, the only way browsers could interact with a server was by polling and long-polling the server with a steady stream of requests. These methods, while they worked, had a very high network latency as well as high-bandwidth usage, making them unsuitable for real-time multiplayer games.

All of this changed with the arrival of the WebSocket API. The API defines a bidirectional, full-duplex communications channel over a single TCP socket, providing us with an efficient, low-latency connection between browser and server.

In simple terms, we can now create a single, persistent connection between a browser and server and send data back and forth much faster than before. You can read more about the benefits of WebSockets at www.websocket.org/. Let's take a look at a simple example of communication between the browser and the server using WebSockets.

WebSockets on the Browser

Using WebSockets to communicate with a server involves the following steps:

1. Instantiating a `WebSocket` object by providing the server URL

2. Implementing the `onopen`, `onclose`, and `onerror` event handlers as needed

3. Implementing the `onmessage` event handler to handle actions when a message is received from the server

4. Sending messages to the server by using the `send()` method

5. Closing the connection to the server by using the `close()` method

We can create a simple WebSocket client in a new HTML file, as shown in Listing 12-1. We will place this new file inside the `websocketdemo` folder to keep it separate from our game code.

Listing 12-1. A Simple WebSocket Client (client.html)

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebSocket Client</title>
        <meta charset="UTF-8">
        <script type="text/javascript">
            var websocket;
            var serverUrl = "ws://localhost:8080";

            function displayMessage(message) {
                document.getElementById("display").value += message + "\n";
            }

            function initWebSocket() {

                // Make sure the browser supports WebSockets
                if (!window.WebSocket) {
                    displayMessage("Your browser does not support WebSockets");
                    return;
                }

                // Instantiate a new websocket object with the server URL
                websocket = new WebSocket(serverUrl);

                // Set handler for when the socket connection is opened
                websocket.addEventListener("open", function() {
                    displayMessage("WebSocket connection opened");
                    document.getElementById("sendmessage").disabled = false;
                });

                // Set handler for when the socket connection is closed
                websocket.addEventListener("close", function() {
                    displayMessage("WebSocket connection closed");
                    document.getElementById("sendmessage").disabled = true;
                });

                // Set handler for when a socket connection error occurs
                websocket.addEventListener("error", function() {
                    displayMessage("WebSocket connection error");
                    document.getElementById("sendmessage").disabled = true;
                });

                // Set handler for when the socket receives a message
                websocket.addEventListener("message", function(message) {
                    displayMessage("Received Message: \"" + message.data + "\"");
                });
            }
```

```
        function sendMessage() {

            if (websocket.readyState === WebSocket.OPEN) {
                var message = document.getElementById("message").value;

                displayMessage("Sending Message: \"" + message + "\"");
                websocket.send(message);
            } else {
                displayMessage("Cannot send message. The WebSocket connection is not open.");
                document.getElementById("sendmessage").disabled = true;
            }
        }


    </script>
    </head>
    <body onload="initWebSocket()">
        <p>
            <input type="text" placeholder="Enter Message Here" size="40" id="message">
            <input type="button" value="Send" id="sendmessage" onclick="sendMessage()"
            disabled="true">
        </p>
        <p>
            <textarea rows="10" cols="80" id="display"></textarea>
        </p>
    </body>
</html>
```

The body tag of the HTML file contains a few basic elements: an input box for a message, a button for sending messages, and a textarea to display all messages.

Within the script tag, we start by declaring a server URL that points to the WebSocket server using the WebSocket protocol (ws://).

We then declare a simple displayMessage() method that appends a given message to the display textarea element.

Next we declare the initWebSocket() method that initializes the WebSocket connection and sets up the event handlers.

Within this method, we first check for the existence of the WebSocket object to verify that the browser supports WebSockets and, if not, display an appropriate message and exit.

We then initialize the WebSocket object by passing the server URL to the constructor and saving it to the websocket variable.

Finally, we define handlers for the onopen, onclose, onerror, and onmessage event handlers, where we display appropriate messages to the user. We also enable the sendmessage button when the connection is opened and disable it when the connection is closed.

In the sendMessage() method, we check that the connection is open using the readyState property and then use the send() method to send the contents of the message input box to the server.

Our browser client needs a server that it can communicate with using the WebSocket protocol. There are already several WebSocket server implementations available for most popular languages, such as jWebSocket (http://jwebsocket.org/) and Jetty (www.eclipse.org/jetty/) for Java, Socket.io (https://github.com/socketio/socket.io) and WebSocket-Node (https://github.com/theturtle32/WebSocket-Node) for Node.js, and WebSocket++ (https://github.com/zaphoyd/websocketpp) for C++.

In this book, we will be using WebSocket-Node for Node.js. We will start by setting up Node.js and creating an HTTP server and then add WebSocket support to it.

Creating an HTTP Server in Node.js

Node.js (http://nodejs.org/) is a server-side platform consisting of several libraries built on top of Google's JavaScript V8 engine. Originally created by Ryan Dahl starting in 2009, Node.js was designed for easily building fast, scalable network applications. Programs are written in JavaScript using an event-driven, nonblocking I/O model that is lightweight and efficient. Node.js has gained a lot of popularity in a relatively short time and is used by a large number of companies, including LinkedIn, Microsoft, and Yahoo.

Before you can start writing Node.js code, you will need to install Node.js on your computer. Implementations of Node.js are available for most operating systems, such as Windows, Mac OS X, Linux, and SunOS. For Windows and Mac OS X, the simplest installation method is to run the ready-made installer files downloadable at http://nodejs.org/download/. Detailed instructions for setting up Node.js via package manager on your specific operating system are also available at https://nodejs.org/en/download/package-manager/.

Once Node.js has been set up correctly, you will be able to run Node.js programs from the command line by calling the node executable and passing the program name as a parameter.

When building a Node.js application, it is considered a good practice to create a package.json file, which contains important details for the project such as the name, version, and most importantly any external project dependencies that need to be downloaded to run the project. If Node.js and its package manager npm have been set up correctly, you should be able to set up this file by running the following command from the command line:

```
npm init
```

The npm utility will then walk you through the process of creating a package.json file by asking you a few questions about the project, while providing sensible defaults. As you add dependencies to your project, they can be saved to this file as well, making it easier for other users to build and run your projects. An example of a package.json file is shown in Listing 12-2.

Listing 12-2. A Sample package.json File

```
{
  "name": "websocket-demo",
  "version": "1.0.0",
  "description": "A simple WebSocket server demo",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "author": "Aditya Ravi Shankar",
  "license": "MIT"
}
```

After setting up the package.json file, we can create a simple HTTP web server inside a new JavaScript file, as shown in Listing 12-3. We will place this file inside the websocketdemo folder.

Listing 12-3. A Simple HTTP Web Server in Node.js (server.js)

```
// Create an HTTP Server
var http = require("http");

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request, response) {
    console.log("Received HTTP request for URL", request.url);

    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("This is a simple node.js HTTP server.");
});

// Listen on port 8080
server.listen(8080, function() {
    console.log("Server has started listening on port 8080");
});
```

The code for building a simple web server in Node.js is surprisingly small using the Node.js HTTP library. You can find detailed documentation on this library at https://nodejs.org/api/http.html.

We first refer to the HTTP library using the require() method and save it to the http variable. We then create an HTTP server by calling the createServer() method and passing it a method that will handle all HTTP requests. In our case, we send back the same text response for any HTTP request to the server. Finally, we tell the server to start listening on port 8080, and ask it to show a message on the console once it has started listening.

If you run the code in server.js from the command line and try to access the web server's URL (http://localhost:8080) from the browser, you should see the output shown in Figure 12-1.
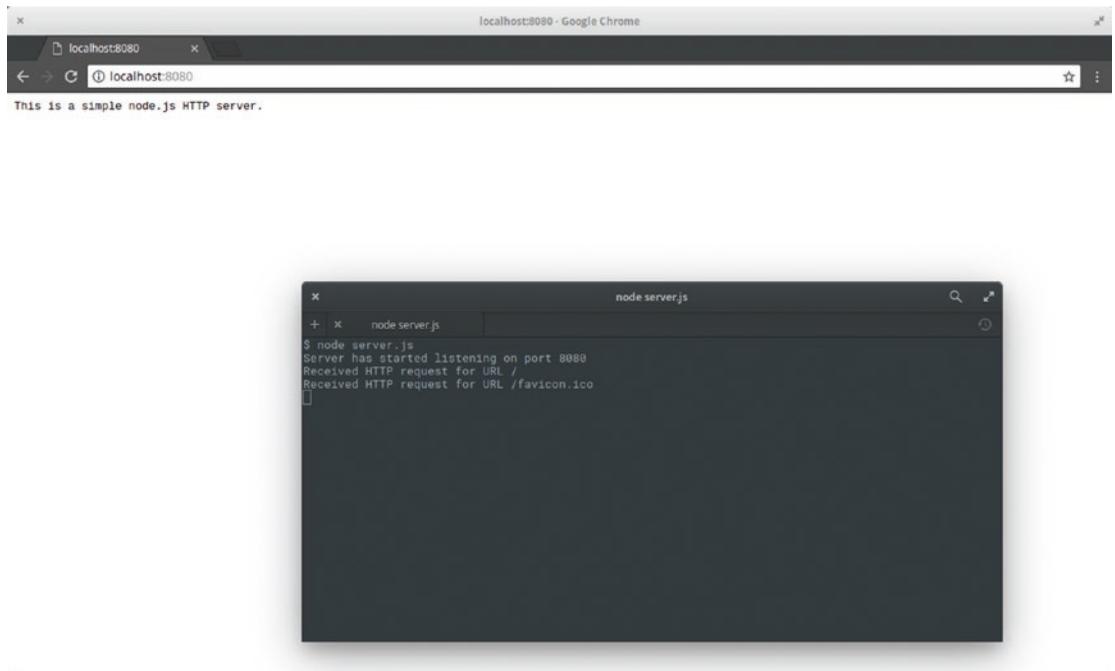
Figure 12-1. *A simple HTTP server in Node.js*

357

We have our HTTP server up and running. This server will return the same page no matter what path we pass after the server name in the URL. This server also does not yet support WebSockets.

Next, we will add WebSocket support to this server by using the WebSocket-Node package.

Creating a WebSocket Server

The first thing you will need to do is install the WebSocket-Node package using the npm command. Detailed instructions for installation along with sample code is available at https://github.com/theturtle32/WebSocket-Node.

If Node.js is set up correctly, you should be able to set up WebSocket by running the following command from the command line:

```
npm install websocket --save
```

■ **Tip**   In case you have previously installed Node.js and WebSocket-Node, you should ensure that you are using the latest version by running the npm update command.

The extra save parameter tells npm to add this dependency to the package.json file. As long as you include package.json with your source code, others can install all the dependencies for the project using the npm install command.

Once the WebSocket package has been installed, we will add WebSocket support by modifying server.js, as shown in Listing 12-4.

Listing 12-4. Implementing a Simple WebSocket Server (server.js)

```
// Create an HTTP Server
var http = require("http");

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request, response) {
    console.log("Received HTTP request for URL", request.url);

    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("This is a simple node.js HTTP server.");
});

// Listen on port 8080
server.listen(8080, function() {
    console.log("Server has started listening on port 8080");
});

// Attach WebSocket server to HTTP server
var WebSocketServer = require("websocket").server;
var wsServer = new WebSocketServer({
    httpServer: server
});
```

```
// Logic to determine whether a specified connection is allowed
function connectionIsAllowed(request) {
    // Check criteria such as request.origin, request.remoteAddress
    // Return false to prevent connection, true to allow connection
    return true;
}

// Handle WebSocket Connection Requests
wsServer.on("request", function(request) {
    // Reject requests based on certain criteria
    if (!connectionIsAllowed(request)) {
        request.reject();
        console.log("WebSocket Connection from " + request.remoteAddress + " rejected.");

        return;
    }

    // Accept Connection
    var websocket = request.accept();

    console.log("WebSocket Connection from " + request.remoteAddress + " accepted.");
    websocket.send("Hi there. You are now connected to the WebSocket Server");

    websocket.on("message", function(message) {
        if (message.type === "utf8") {
            console.log("Received Message: " + message.utf8Data);
            websocket.send("Server received your message: " + message.utf8Data);
        }
    });

    websocket.on("close", function(reasonCode, description) {
        console.log("WebSocket Connection from " + request.remoteAddress + " closed.");
    });
});
```

The first part of the code where we create the HTTP server remains the same. In the newly added code, we start by using the require() method to save a reference to the WebSocket server. We then create a new WebSocketServer object by passing the HTTP server that we created earlier as a configuration option. You can read about the different WebSocketServer configuration options as well as details on the WebSocketServer API at https://github.com/theturtle32/WebSocket-Node/blob/master/docs/index.md.

Next we implement the handler for the request event of the server. We first check whether the connection request should be rejected and, if so, call the reject() method of the request.

We use a method called connectionIsAllowed() to filter connections that need to be rejected. Right now we approve all connections; however, this method can use information such as the connection request's IP address and origin to intelligently filter requests.

If the connection is allowed, we accept the request using the accept() method and save the resulting WebSocket connection to the websocket variable. This websocket variable is the server-side equivalent of the websocket variable that we created in the client HTML file earlier.

Once we create the connection, we use the websocket object's send() method to send the client a welcome message notifying it that the connection has been made.

Next we implement the handler for the message event of the websocket object. Every time a message arrives, we send back a message to the client saying the server just received the message and then log the message to the console.

■ **Note**   The WebSocket API allows for multiple message data types such as UTF8 text, binary, and blob data. Unlike on the browser, the message object on the server side stores the message data using different properties (such as utf8Data, binaryData) based on the data type.

Finally, we implement the handler for the close event, where we just log the fact that the connection was closed.

If you run the server.js code from the command line using the command node server.js, and open client.html in the browser, you should see the interaction between the client and the server, as shown in Figure 12-2.
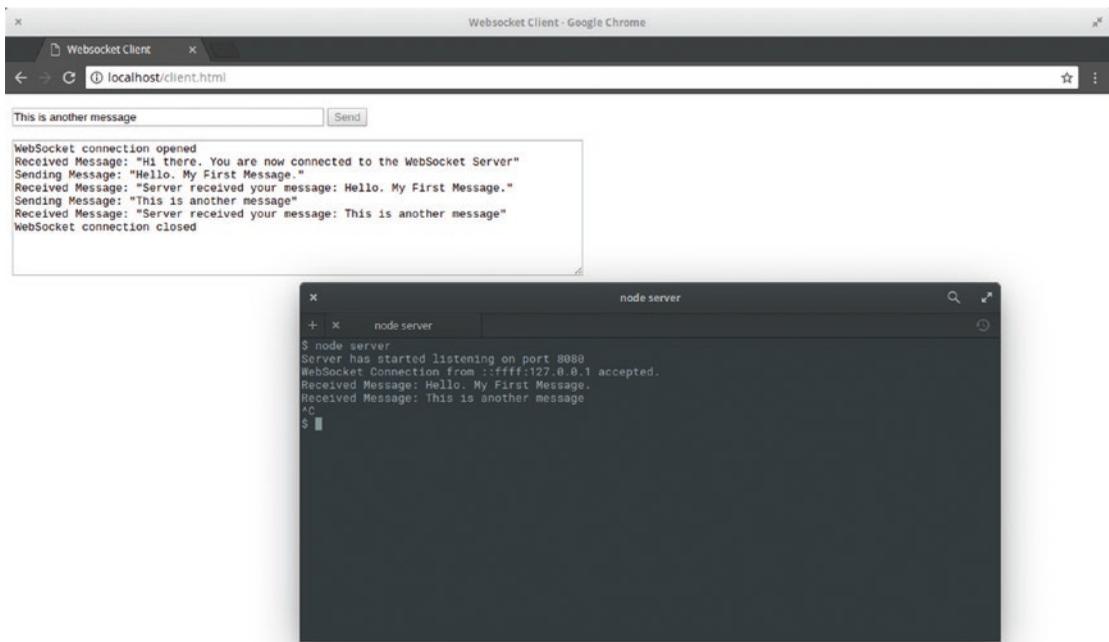
Figure 12-2. *Interaction between client and server*

As soon as the WebSocket connection is established, the browser receives a welcome message from the server. The Send button also gets enabled on the client. If you type a message and click Send, the server displays the message in the console and sends back a response to the client, which displays the response. Finally, if you shut down the server using the Ctrl+C key combination, the client displays a message that the connection has been closed.

We now have a working example of transmitting plain-text messages back and forth between the client and the server.

■ **Note** It is possible to reduce the message size and optimize bandwidth usage by using binary data instead of plain text. However, we will continue to use UTF8 text even in our game implementation to keep the code simple.

Now that we have looked at the basics of WebSocket communication, it is time to add multiplayer to our game. We will continue from where we left off at the end of Chapter 11.

The first thing we will build is a multiplayer game lobby.

Building the Multiplayer Game Lobby

Our game lobby will display a list of game rooms. Players can join or leave these rooms from the game lobby screen. Once two players join a room, the multiplayer game will start, and the two players can compete with each other.

Defining the Multiplayer Lobby Screen

We will start by adding the HTML code for the multiplayer lobby screen into the gamecontainer div in index.html, as shown in Listing 12-5.

Listing 12-5. HTML Code for the Multiplayer Lobby Screen (index.html)

```
<div id="multiplayerlobbyscreen" class="gamelayer">
    <table id="multiplayergameslist">
    </table>
    <input type="button" id="multiplayerjoin" onclick="multiplayer.join();">
    <input type="button" id="multiplayercancel" onclick="multiplayer.cancel();">
</div>
```

The layer contains a table element to display the list of game rooms along with two buttons. We will also add the CSS code for the lobby screen to styles.css, as shown in Listing 12-6.

Listing 12-6. CSS Code for the Multiplayer Lobby Screen (styles.css)

```
/* Multiplayer Lobby Screen */

#multiplayerlobbyscreen {
    background: url("images/screens/multiplayerlobby.png") no-repeat center;
    text-align: center;
}

#multiplayergameslist {

    /* Center on the screen */
    position: absolute;
    left: 50%;
    top: 50%;
```

```css
    transform: translate(-50%, -50%);
    transform-origin: center center;

    width: 394px;
    height: 274px;

    /* Hide scrollbar and border effects */
    overflow: hidden;
    border: none;
    outline: none;

    /* Translucent gray background */
    background: rgba(0, 0, 0, 0.5);

    /* Format Text */
    color: lightblue;
    font-size: 14px;
    font-family: "Courier New", Courier, monospace;
    padding: 20px;

    text-align: left;
}

/* Styles for Join and Cancel buttons */

#multiplayerlobbyscreen input[type="button"] {
    position: relative;
    top: 410px;
    width: 74px;
    height: 26px;
}

#multiplayerjoin {
    left: -225px;
    background-position: -4px -186px;
}

#multiplayerjoin:active, #multiplayerjoin:disabled {
    background-position: -82px -186px;
}

#multiplayercancel {
    background-position: -4px -154px;
    left: 225px;
}

#multiplayercancel:active, #multiplayercancel:disabled {
    background-position: -82px -154px;
}
```

```css
/* Styles for different game room statuses */

tr {
    cursor: pointer;
}

.selected {
    background-color: #274466;
}

.running, .starting {
    background-color: #444444;
    color: #666666;
    cursor: not-allowed;
}

.waiting {
    color: lightgreen;
}

.empty {
    color: lightblue;
}
```

Now that the lobby screen is in place, we will build the code to connect the browser to the server and populate the games list.

Populating the Games List

We will start by defining a new `multiplayer` object inside `multiplayer.js`, as shown in Listing 12-7.

Listing 12-7. Defining the multiplayer Object (multiplayer.js)

```javascript
var multiplayer = {

    // Open multiplayer game lobby
    websocket: undefined,
    start: function() {
        if (!window.WebSocket) {
            game.showMessageBox("Your browser does not support WebSocket. Multiplayer will
            not work.");

            return;
        }

        const websocketUrl = "ws://" + (window.location.hostname || "localhost") + ":8080";

        this.websocket = new WebSocket(websocketUrl);

        this.websocket.addEventListener("open", multiplayer.handleWebSocketOpen);
        this.websocket.addEventListener("message", multiplayer.handleWebSocketMessage);
```

```
        this.websocket.addEventListener("close", multiplayer.handleWebSocketConnectionError);
        this.websocket.addEventListener("error", multiplayer.handleWebSocketConnectionError);
    },

    // Display multiplayer lobby screen after connection is opened
    handleWebSocketOpen: function() {
        game.hideScreens();
        game.showScreen("multiplayerlobbyscreen");
    },

    handleWebSocketMessage: function(message) {
        var messageObject = JSON.parse(message.data);

        switch (messageObject.type) {
            case "room-list":
                multiplayer.updateRoomStatus(messageObject.roomList);
                break;
        }
    },

    statusMessages: {
        "starting": "Game Starting",
        "running": "Game in Progress",
        "waiting": "Waiting for second player",
        "empty": "Open"
    },

    selectRow: function(index) {
        var list = document.getElementById("multiplayergameslist");

        // Remove any existing selected rows
        for (let i = list.rows.length - 1; i >= 0; i--) {
            let row = list.rows[i];

            row.classList.remove("selected");
        }

        list.selectedIndex = index;
        let row = list.rows[index];

        list.value = row.cells[0].value;
        row.classList.add("selected");
    },

    updateRoomStatus: function(roomList) {
        var list = document.getElementById("multiplayergameslist");

        // Clear all the old options
        for (let i = list.rows.length - 1; i >= 0; i--) {
            list.deleteRow(i);
        }
```

```
        roomList.forEach(function(status, index) {
            let statusMessage = multiplayer.statusMessages[status];
            let roomId = index + 1;
            let label = "Game " + roomId + ". " + statusMessage;

            // Create a new option for the room
            let row = document.createElement("tr");
            let cell = document.createElement("td");

            cell.innerHTML = label;
            cell.value = roomId;

            row.appendChild(cell);

            row.addEventListener("click", function() {
                if (!list.disabled && !row.disabled) {
                    multiplayer.selectRow(index);
                }
            });

            row.className = status;

            list.appendChild(row);

            // Disable rooms that are running or starting
            if (status === "running" || status === "starting") {
                row.disabled = true;
            }

            // In case multiplayer.roomId is set, select the room with that roomId and
            // unselect others
            if (multiplayer.roomId === roomId) {
                this.selectRow(index);
            }

        }, this);

    },

};
```

Inside the `multiplayer` object, we first define a `start()` method that tries to initialize a WebSocket connection to the server and saves the connection object in the `websocket` variable. We then set the `websocket` object's `onmessage` event handler to call the `handleWebSocketMessage()` method and use the `onopen` event handler to call the `handleWebSocketOpen()` method, which will display the lobby screen once the connection is opened.

An interesting thing to note is how we decide the URL for the WebSocket. If the game is accessed using a `file://` URL, we use `localhost` for the WebSocket server. If, however, the game is hosted on a web server and accessed via an `http://` URL, we get `hostname` from the `window` object. In both cases, we assume that the server is running on port 8080.

Next we define the handleWebSocketMessage() method to handle the message data. Instead of passing strings like we did in our WebSocket example earlier, we are going to be passing complete objects between the server and the browser by using JSON.parse() and JSON.stringify() to convert between objects and strings. We start by parsing the message data into a messageObject variable and then use its type property to decide how to handle the message.

If the type property is set to room-list, we call the updateRoomStatus() method and pass it the roomList property.

Finally, we define an updateRoomStatus() method that takes an array of status messages and populates the multiplayergameslist table element. We disable any options that have a status of starting or running and also set the CSS class of the option to the status. We define a statusMessages object that we use to map the status codes passed to us by the server into more descriptive messages.

We also define a selectRow() method that allows us to select a specific row within the table, highlight it, and access the index of the selected row using the selectedIndex attribute, and the room number using the value attribute. In essence, the table now behaves like a nice-looking HTML select element.

Next, we will add a reference to multiplayer.js inside the head section of index.html, as shown in Listing 12-8.

Listing 12-8. Adding Reference to multiplayer.js (index.html)

```
<script src="js/multiplayer.js" type="text/javascript"></script>
```

Now that the multiplayer client is in place, we will build our multiplayer server. We will keep our code inside a server folder to separate it from our client code.

We will start by building a package.json file using npm init and installing the WebSocket package using npm install websocket --save, just as we did in our WebSocket example.

We will then define our multiplayer WebSocket server inside a new file called server.js, as shown in Listing 12-9.

Listing 12-9. Defining the Multiplayer Server (server.js)

```
// Create an HTTP Server
var http = require("http");

// Create a simple web server that returns the same response for any request
var server = http.createServer(function(request, response) {
    console.log("Received HTTP request for URL", request.url);

    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("This is a simple node.js HTTP server.");
});

// Listen on port 8080
server.listen(8080, function() {
    console.log("Server has started listening on port 8080");
});

// Attach WebSocket server to HTTP server
var WebSocketServer = require("websocket").server;
var wsServer = new WebSocketServer({
    httpServer: server,
    autoAcceptConnections: false
});
```

```javascript
// Initialize a set of 10 rooms
var gameRooms = [];

for (var i = 0; i < 10; i++) {
    gameRooms.push({ status: "empty", players: [], roomId: i + 1 });
}

// Store all the players currently connected to the server
var players = [];

wsServer.on("request", function(request) {

    var connection = request.accept();

    console.log("Connection from " + request.remoteAddress + " accepted.");

    // Add the player to the players array
    var player = {
        connection: connection,
        latencyTrips: []
    };

    players.push(player);

    // Send a fresh game room status list the first time player connects
    sendRoomList(connection);

    // Handle receiving of messages
    connection.on("message", function(message) {
        if (message.type === "utf8") {
            var clientMessage = JSON.parse(message.utf8Data);

            // Handle message based on message type
            switch (clientMessage.type) {

            }
        }
    });

    // Handle closing of connection
    connection.on("close", function() {
        console.log("Connection from " + request.remoteAddress + " disconnected.");

        // Remove the player from the players array
        var index = players.indexOf(player);

        if (index > -1) {
            players.splice(index, 1);
        }

    });
});
```

```
function getRoomListMessageString() {
    var roomList = [];

    for (var i = 0; i < gameRooms.length; i++) {
        roomList.push(gameRooms[i].status);
    }

    var message = { type: "room-list", roomList: roomList };
    var messageString = JSON.stringify(message);

    return messageString;
}

function sendRoomList(connection) {
    var messageString = getRoomListMessageString();

    connection.send(messageString);
}
```

We start by defining the HTTP server and the WebSocketServer like we did in our earlier websocketdemo example.

Next, we define a rooms array and fill it with ten room objects that have a status property set to empty.

Finally, we implement the connection request event handler. We start by creating a player object for the connection and adding it to the players array. We then call the sendRoomList() method for the connection.

Next, we implement the message event handler for the connection to parse the message data and respond based on the type property just like we did on the client. We aren't processing any message types yet.

Next, we implement the close event handler where we remove the player from the players array once the connection closes.

Finally, we create a sendRoomList() method that sends a status array inside a message object of type room-list. This is the same message that we will be parsing on the client side.

If we run the newly created server.js file and then open our game in the browser, we should be able to click the Multiplayer menu option and arrive at the multiplayer game lobby screen, as shown in Figure 12-3.

Figure 12-3. *The multiplayer game lobby screen*

Behind the scenes, the client is creating a socket connection to the server, and the server is sending back a room-list message to the client, which is then used to populate the list.

You should be able to select any of the game rooms but cannot join or leave these rooms. We will now implement joining and leaving a game room.

Joining and Leaving a Game Room

We will start by implementing join() and cancel() methods inside the multiplayer object, as shown in Listing 12-10.

Listing 12-10. Implementing join() and cancel() (multiplayer.js)

```
join: function() {
    var selectedRoom = document.getElementById("multiplayergameslist").value;

    if (selectedRoom) {
        // If a room has been selected, try to join the room
        multiplayer.sendWebSocketMessage({ type: "join-room", roomId: selectedRoom });

        // Disable room list and join button
        document.getElementById("multiplayergameslist").disabled = true;
        document.getElementById("multiplayerjoin").disabled = true;
    } else {
        // Otherwise ask player to select a room first
        game.showMessageBox("Please select a game room to join.");
    }
},
```

```
cancel: function() {
    if (multiplayer.roomId) {
        // If the player is in a room, Cancel will just leave the room
        multiplayer.sendWebSocketMessage({ type: "leave-room", roomId: multiplayer.roomId });
        document.getElementById("multiplayergameslist").disabled = false;
        document.getElementById("multiplayerjoin").disabled = false;

        // Clear roomId and color
        delete multiplayer.roomId;
        delete multiplayer.color;
    } else {
        // If the player is not in a room, leave the multiplayer screen itself
        multiplayer.closeAndExit();
    }
},

closeAndExit: function() {
    // Clear all handlers and close connection
    multiplayer.websocket.removeEventListener("open", multiplayer.handleWebSocketOpen);
    multiplayer.websocket.removeEventListener("message", multiplayer.handleWebSocketMessage);

    multiplayer.websocket.close();

    // Enable room list and Join button
    document.getElementById("multiplayergameslist").disabled = false;
    document.getElementById("multiplayerjoin").disabled = false;

    // Show the starting menu layer
    game.hideScreens();
    game.showScreen("gamestartscreen");
},

sendWebSocketMessage: function(messageObject) {
    var messageString = JSON.stringify(messageObject);

    this.websocket.send(messageString);
},
```

In the join() method, we check whether a room has been selected and, if it has, send a join-room WebSocket message to the server with the roomId property, using the sendWebSocketMessage() method. We then disable the Join button and the games list. If no room is selected, we ask the player to select a room first.

The Cancel button serves two purposes: leaving a joined room or leaving the multiplayer lobby entirely. Within the cancel() method, we first check whether the player is in a room using the multiplayer.roomId property. If so, we send a leave-room WebSocket message to the server, delete the roomId and color properties, and enable the Join button and the games list element. If not, we close the socket connection and return to the game start screen using the closeAndExit() method.

In the closeAndExit() method, we first clear the websocket object's event handlers and close the connection. We then enable the Join button and games list and return to the game start screen.

Finally, we define a `sendWebSocketMessage()` method that converts the `messageObject` into a string and sends it to the server.

Next, we will modify the server to handle the `join-room` and `leave-room` message types by modifying the `message` event handler in `server.js`, as shown in Listing 12-11.

Listing 12-11. Handling join-room and leave-room Messages (server.js)

```
// Handle receiving of messages
connection.on("message", function(message) {
    if (message.type === "utf8") {
        var clientMessage = JSON.parse(message.utf8Data);

        // Handle message based on message type
        switch (clientMessage.type) {
            case "join-room":
                joinRoom(player, clientMessage.roomId);
                sendRoomListToEveryone();
                break;

            case "leave-room":
                leaveRoom(player, clientMessage.roomId);
                sendRoomListToEveryone();
                break;
        }
    }
});
```

When a `join-room` message comes in, we first call the `joinRoom()` method and then send the room list to all players using the `sendRoomListToEveryone()` method. Similarly, when a `leave-room` message comes in, we first call the `leaveRoom()` method and then call the `sendRoomListToEveryone()` method.

Next, we will define these three new methods inside `server.js`, as shown in Listing 12-12.

Listing 12-12. The joinRoom(), leaveRoom(), and sendRoomListToEveryone() Methods (server.js)

```
function sendRoomListToEveryone() {
    var messageString = getRoomListMessageString();

    // Notify all connected players of the room status changes
    players.forEach(function(player) {
        player.connection.send(messageString);
    });
}

function joinRoom(player, roomId) {
    var room = gameRooms[roomId - 1];

    console.log("Adding player to room", roomId);
    // Add the player to the room
    room.players.push(player);
    player.room = room;
```

```
    // Update room status and choose player color (blue for first player, green for the second)
    if (room.players.length === 1) {
        room.status = "waiting";
        player.color = "blue";
    } else if (room.players.length === 2) {
        room.status = "starting";
        player.color = "green";
    }

    // Confirm to player that he was added
    var confirmationMessage = { type: "joined-room", roomId: roomId, color: player.color };
    var confirmationMessageString = JSON.stringify(confirmationMessage);

    player.connection.send(confirmationMessageString);

    return room;
}

function leaveRoom(player, roomId) {
    var room = gameRooms[roomId - 1];

    console.log("Removing player from room", roomId);

    // Remove the player from the players array
    var index = room.players.indexOf(player);

    if (index > -1) {
        room.players.splice(index, 1);
    }

    delete player.room;

    // Update room status
    if (room.players.length === 0) {
        room.status = "empty";
    } else if (room.players.length === 1) {
        room.status = "waiting";
    }
}
```

In the sendRoomListToEveryone() method, we iterate through all the players in the players array and send them a room-list message with the list of rooms.

In the joinRoom() method, we first get the room object using roomId and add the player to the room object's players array. We then set the room object's status to waiting or starting depending on how many players are in the room. We also set the player's color to blue or green based on whether the player is the first or second player to join the room. Finally, we send a joined-room message back to the player, with details of the room ID and player color.

In the leaveRoom() method, we first get the room object using roomId and remove the player from the room object's players array. We then set the room object's status to empty or waiting depending on how many players are left in the room.

The next change we will make is to handle the joined-room confirmation message inside multiplayer.js, as shown in Listing 12-13.

Listing 12-13. Handling the joined-room Message (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

    }
},
```

When a joined-room message comes in, we save the message's roomId and color properties inside the multiplayer object.

Finally, we will ensure that a player is removed from a game room if the player is disconnected by modifying the close event handler on the server, as shown in Listing 12-14.

Listing 12-14. Handling Player Disconnects (server.js)

```
// Handle closing of connection
connection.on("close", function() {
    console.log("Connection from " + request.remoteAddress + " disconnected.");

    // Remove the player from the players array
    var index = players.indexOf(player);

    if (index > -1) {
        players.splice(index, 1);
    }

    var room = player.room;

    if (room) {
        // If the player was in a room, remove the player from the room
        leaveRoom(player, room.roomId);

        // Notify everyone about the changes
        sendRoomListToEveryone();
    }
});
```

In the newly added code, we check whether the disconnected player is in a room, and if so, we remove the player from the room using the leaveRoom() method and then notify everyone using the sendRoomListToEveryone() method.

If we restart the server and run the game in more than one browser window, we should be able to join a room in one window and see the status change in both the windows, as shown in Figure 12-4.
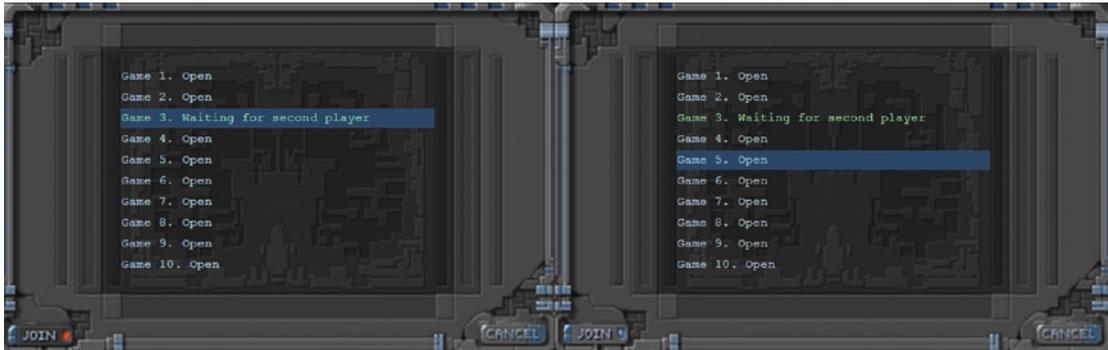
Figure 12-4. *Room status updated on both browsers when joining a room*

You will notice that the Join button and the list get disabled once you join a room and the room has a new status of waiting. If you join the same room on both browsers, the room status changes to starting and no one else can join the room.

If you click Cancel, you will leave the room, and the Join button will be reenabled. If you click Cancel again, you will be taken back to the main menu. If you disconnect from the server by closing the browser window, you will be removed from the room.

We now have a working game lobby where players can join and leave game rooms. Next we will start the multiplayer game once the players join the game room.

Starting the Multiplayer Game

Our multiplayer game will start once two players join a game room. We will need to tell both the clients to load the same level. Once the level has loaded on both browsers, we will then start the game. The first thing we need to do is define a new multiplayer level.

Defining the Multiplayer Level

The multiplayer level, while being similar to the single-player levels, will contain some extra information such as the starting location for each player and the starting items for each team. We will start by defining a new level inside the multiplayer array in the levels object, as shown in Listing 12-15.

Listing 12-15. A Multiplayer Level Inside the multiplayer Array (levels.js)

```
"multiplayer": [
    {
        /* Map Details */
        "mapName": "plains",

        /* Entities to be loaded */
        "requirements": {
            "buildings": ["base", "harvester", "starport", "ground-turret"],
            "vehicles": ["transport", "scout-tank", "heavy-tank", "harvester"],
            "aircraft": ["wraith", "chopper"],
            "terrain": ["oilfield"]
        },

        /* Starting Cash */
        "cash": {
            "blue": 3000,
            "green": 3000
        },

        /* Entities to be added */
        "items": [
            { "type": "terrain", "name": "oilfield", "x": 16, "y": 4, "action": "hint" },
            { "type": "terrain", "name": "oilfield", "x": 34, "y": 12, "action": "hint" },
            { "type": "terrain", "name": "oilfield", "x": 1, "y": 30, "action": "hint" },
            { "type": "terrain", "name": "oilfield", "x": 38, "y": 38, "action": "hint" },
        ],

        /* Entities for each starting team */
        "teamStartingItems": [
            { "type": "buildings", "name": "base", "x": 0, "y": 0 },
            { "type": "vehicles", "name": "harvester", "x": 4, "y": 0 },
            { "type": "vehicles", "name": "heavy-tank", "x": 4, "y": 2 },
            { "type": "vehicles", "name": "scout-tank", "x": 6, "y": 0 },
            { "type": "vehicles", "name": "scout-tank", "x": 6, "y": 2 },
        ],

        /* Possible starting spawn locations for the players */
        "spawnLocations": [
            { "x": 48, "y": 36, "startX": 36, "startY": 20 },
            { "x": 3, "y": 36, "startX": 0, "startY": 20 },
            { "x": 36, "y": 3, "startX": 32, "startY": 0 },
            { "x": 3, "y": 3, "startX": 0, "startY": 0 },
        ],

        /* Conditional and Timed Trigger Events */
        "triggers": [
        ]

    }

]
```

The two new elements that we have introduced in the multiplayer level are the `teamStartingItems` and `spawnLocations` arrays.

The `teamStartingItems` array contains a list of items that each team will have at the beginning of the level. The x and y coordinates will be relative to the location where the team is spawned.

The `spawnLocations` array contains a few spots on the map where each player team can start. Each object contains the x and y coordinates of the location, as well as the starting panning offset for the location.

Now that we have defined the multiplayer level, we need to load the level once the two players join a game room.

Loading the Multiplayer Level

When two players join a room, we will tell them both to initialize the level and wait for both to confirm that the level has been initialized. Once this happens, we will tell them both to start the game.

We will start by adding a few new methods to `server.js` to handle initializing and starting the game, as shown in Listing 12-16.

Listing 12-16. Initializing and Starting the Game (server.js)

```
function initializeGame(room) {
    console.log("Both players Joined. Initializing game for Room " + room.roomId);

    // Number of players who have loaded the level
    room.playersReady = 0;

    // Load the first multiplayer level for both players
    // This logic can change later to let the players pick a level
    var currentLevel = 0;

    // Randomly select two spawn locations between 0 and 3 for both players
    var spawns = [0, 1, 2, 3];
    var spawnLocations = { "blue": spawns.splice(Math.floor(Math.random() * spawns.length),
    1), "green": spawns.splice(Math.floor(Math.random() * spawns.length), 1) };

    sendRoomWebSocketMessage(room, { type: "initialize-level", spawnLocations:
    spawnLocations, currentLevel: currentLevel });
}

function startGame(room) {
    console.log("Both players are ready. Starting game in room", room.roomId);

    room.status = "running";
    sendRoomListToEveryone();
    // Notify players to start the game
    sendRoomWebSocketMessage(room, { type: "play-game" });

}

function sendRoomWebSocketMessage(room, messageObject) {
    var messageString = JSON.stringify(messageObject);
```

```
    room.players.forEach(function(player) {
        player.connection.send(messageString);
    });
}
```

In the `initializeGame()` method, we initialize the `playersReady` variable to 0, select two random spawn locations for both the players, and send the location to both the players inside an `initialize-level` message using the `sendRoomWebSocketMessage()` method.

In the `startGame()` method, we set the room status to `running`, update every player's room list, and finally send both players the `play-game` message using the `sendRoomWebSocketMessage()` method.

Finally, in the `sendRoomWebSocketMessage()` method we iterate through all the players in a room and send each of them a given message.

Next, we will modify the message event handler in `server.js` to initialize and start the game, as shown in Listing 12-17.

Listing 12-17. Modifying the Message Event Handler (server.js)

```
// Handle receiving of messages
connection.on("message", function(message) {
    if (message.type === "utf8") {
        var clientMessage = JSON.parse(message.utf8Data);

        // Handle Message based on message type
        switch (clientMessage.type) {
            case "join-room":
                joinRoom(player, clientMessage.roomId);

                sendRoomListToEveryone();

                if (player.room.players.length === 2) {
                    // Two players have joined. Initialize the game
                    initializeGame(player.room);
                }

                break;

            case "leave-room":
                leaveRoom(player, clientMessage.roomId);
                sendRoomListToEveryone();
                break;

            case "initialized-level":
                player.room.playersReady++;

                if (player.room.playersReady === 2) {
                    // Both players are ready, Start the game
                    startGame(player.room);
                }

                break;
        }
    }
```

We first modify the `join-room` case to call the `initializeGame()` method once the second player has joined. Next, when we receive an `initialized-level` message from a player, we increment the `playersReady` count. Once this count reaches two, we call the `startGame()` method.

This way, the server waits for two players to join a room, after which it tells both the players to initialize the level. Once both players confirm that they have initialized the level, the server starts the game.

Next we will add two new methods to the `multiplayer` object to initialize the multiplayer level and start the game, as shown in Listing 12-18.

Listing 12-18. Initializing and Starting the Multiplayer Game (multiplayer.js)

```
currentLevel: 0,
initLevel: function(spawnLocations) {
    game.type = "multiplayer";
    game.team = multiplayer.color;

    // Load all the items for the level
    var level = levels.multiplayer[multiplayer.currentLevel];

    game.loadLevelData(level);

    fog.initLevel();

    // Initialize multiplayer-related variables
    multiplayer.commands = [[]];
    multiplayer.lastReceivedTick = 0;
    multiplayer.currentTick = 0;

    // Add starting items for both teams at their respective spawn locations
    for (let team in spawnLocations) {
        let spawnIndex = spawnLocations[team];

        for (let i = 0; i < level.teamStartingItems.length; i++) {
            let itemDetails = Object.assign({}, level.teamStartingItems[i]);

            // Position item at spawn location
            itemDetails.x += level.spawnLocations[spawnIndex].x;
            itemDetails.y += level.spawnLocations[spawnIndex].y;
            itemDetails.team = team;

            game.add(itemDetails);
        }
    }

    // Position current player at correct spawn offset location
    let spawnIndex = spawnLocations[game.team];

    game.offsetX = level.spawnLocations[spawnIndex].startX * game.gridSize;
    game.offsetY = level.spawnLocations[spawnIndex].startY * game.gridSize;

    game.createTerrainGrid();
```

```
    // Notify the server once all assets have been loaded
    loader.onload = function() {
        multiplayer.sendWebSocketMessage({ type: "initialized-level" });
    };

},

play: function() {
    // Run the animation loop once
    game.animationLoop();

    game.start();
},
```

You will notice that the multiplayer `initLevel()` method is very similar to its single-player counterpart. Within the method, we start by initializing the `game.team` and `game.type` variables. We then load the level data like we did for single-player. We then initialize a few multiplayer-related variables that we will use later.

Next, we place all the starting items for both the players at their respective spawn locations and set the offset location for each player based on their spawn locations.

We then load the terrain grid like we did for single-player, and finally we send the `initialized-level` message to the server once the level has loaded completely to let the server know that we are ready to start the game.

In the `play()` method, we call `animationLoop()` once, and then call `game.start()`. Unlike the single-player version, we do not set an interval to call `animationLoop()` repeatedly. We will be using a slightly more complicated method for invoking `animationLoop()` to ensure that both the players stay in sync during the game.

Next, we will modify the `handleWebSocketMessage()` method in `multiplayer.js` to call these newly created methods, as shown in Listing 12-19.

Listing 12-19. Modifying Message Handler to Initialize and Start Game (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

        case "initialize-level":
            multiplayer.currentLevel = messageObject.currentLevel;
            multiplayer.initLevel(messageObject.spawnLocations);
            break;
```

```
        case "play-game":
            multiplayer.play();
            break;

    }
},
```

We merely call the `initLevel()` method when we receive an `initialize-level` message and call the `playGame()` method when we receive a `play-game` message.

If we restart the server and run the game in two browser windows, we should be able to join the same room from both browsers and see the game load in both, as shown in Figure 12-5.
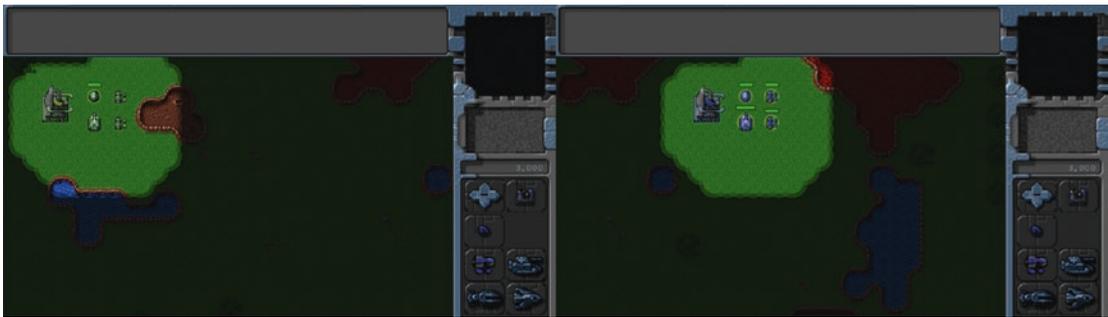
Figure 12-5. *Multiplayer game loading in both browser windows*

Once the two players join the room, the server automatically assigns both the players different colors and spawn locations. When the game loads, both players are placed at their respective spawn locations with the same starting team: two scout tanks, a heavy tank, and a harvester.

We can scroll around the map and even select units; however, we still can't play the game by giving these units commands. This is what we will implement in the next chapter.

Summary

In this chapter, we looked at using the WebSocket API with Node.js for a simple client-server architecture. First we installed Node.js and the WebSocket-Node package and used it to build a simple WebSocket server. Then we built a simple WebSocket-based browser client and sent messages back and forth between the browser and the server.

We used this same architecture to implement a multiplayer game lobby with rooms that players could join and leave. We designed a multiplayer level with spawn locations and starting teams. Finally, we loaded and started the same level on two different browsers, while placing the two players at different spawn locations.

In the next chapter, we will implement the actual multiplayer gameplay by passing commands between the browsers and server. We will use triggers to implement winning and losing a game. Finally, we will add some finishing touches and wrap up the multiplayer section of our game.

CHAPTER 13

■ ■ ■

Multiplayer Gameplay

In the previous chapter, we saw how the WebSocket API could be used with a Node.js server to implement a simple client-server networking architecture. We used this to build a simple game lobby, so two players can now join a game room on a server and start a multiplayer game against each other.

In this chapter, we will continue where we left off at the end of Chapter 12 and implement a framework for the actual multiplayer gameplay using the lock-step networking model. We will look at ways to handle typical game networking problems such as latency and game synchronization. We will then use the sendCommand() architecture that we designed in earlier chapters to ensure that the players' commands are executed on both the browsers so that the games stay in sync. We will then implement winning and losing in the game by using triggers like we did in Chapter 11. Finally, we will implement a chat system for our game.

Let's get started.

The Lock-Step Networking Model

So far, we used the Node.js server to communicate simple messages such as the game lobby status and joining or leaving a room. These messages were independent of each other, and one player's messages did not affect another player. However, when it comes to the gameplay, this communication is going to get a little more complex.

One of the most important challenges in building a multiplayer game is to ensure that all the players are in sync. This means every time a change occurs in any of the games (for example, a player issues a move or attack command), the change is communicated to the other players so that they too can make the same change.

What is even more important is that the action or change occurs at the same moment in both the players' machines. If there is a delay in executing these changes, subtle differences in unit positions will eventually build up, resulting in noticeable divergences between the game states.

For example, a unit that is just half a second late in arriving at an enemy location might avoid an enemy attack and survive in one browser, while the same unit may have been destroyed on the other player's browser. The moment something like this happens, the two players are now playing two completely different games instead of the same one.

To ensure that both players are completely in sync, we will implement an architecture known as the *lock-step* networking model. Both players will start with the same game state. When the player gives a unit a command, we will send the command to the server instead of executing it immediately. The server will then send the same command to the connected players with instructions on when to execute the command. Once the players receive the command, they will execute it at the same time, ensuring that the games stay synchronized.

The server will achieve this behavior by running its own game timer, at 10 clock ticks per second. When a player sends the server a command, the server will record the clock tick when it received the command. The server will then send the command to the players, while specifying the game tick to execute the command. The players in turn will keep track of the current game tick and execute the command at the right tick.

One thing to remember is that since the server needs to execute the commands for all the players at the same time, it will need to wait for the commands from all the players to arrive before stepping ahead to the next game tick, which is why it's called *lock-step*.

This process is further complicated by the fact that network latency can cause communication delays, with messages sometimes taking several hundred milliseconds to travel between client and server. Our networking model will need to measure and take this latency into account to ensure smooth gameplay.

We will start by modifying our game code to measure the network latency for each player when the player first connects to the server.

Measuring Network Latency

For our purposes, we will define the latency as the time taken by a message to travel from the server to the client. We will measure this latency by sending several messages back and forth between the server and the client and then taking an average of the time used for each trip.

We will start by defining two new methods for measuring the latency inside `server.js`, as shown in Listing 13-1.

Listing 13-1. Methods for Measuring Network Latency (server.js)

```
function measureLatencyStart(player) {
    var measurement = { start: Date.now() };

    player.latencyTrips.push(measurement);

    var clientMessage = { type: "latency-ping" };

    player.connection.send(JSON.stringify(clientMessage));
}

// The game clock will run at 1 tick every 100 milliseconds
var gameTimeout = 100;

function measureLatencyEnd(player) {
    // Complete the calculations for the current measurement
    var currentMeasurement = player.latencyTrips[player.latencyTrips.length - 1];

    currentMeasurement.end = Date.now();
    currentMeasurement.roundTrip = currentMeasurement.end - currentMeasurement.start;

    // Calculate the average round trip for all the trips so far
    var totalTime = 0;

    player.latencyTrips.forEach(function (measurement) {
        totalTime += measurement.roundTrip;
    });

    player.averageRoundTrip = totalTime / player.latencyTrips.length;
```

```
    // By default game commands are run one tick after they are received by the server
    player.tickLag = 1;

    // If averageRoundTrip is greater than gameTimeout, increase tickLag to adjust for latency
    player.tickLag += Math.round(player.averageRoundTrip / gameTimeout);

    console.log("Measuring Latency for player. Attempt", player.latencyTrips.length, "-
    Average Round Trip:", player.averageRoundTrip + "ms", "Tick Lag:", player.tickLag);
}
```

In the measureLatencyStart() method, we first create a new measurement object with a start property set to the current time and add the object to the player.latencyTrips array. We then send a message of type latency-ping to the player. The player will respond to this message by sending back a message of type latency-pong.

In the measureLatencyEnd() method, we take the last measurement from the player.latencyTrips array and set its end property to the current time and its roundTrip property to the difference between the end and start times.

We then calculate the average round trip for the player by adding up all the roundTrip values and then dividing the sum by the number of trips.

Finally, we use averageRoundTrip to calculate a tickLag property for the player. This is the number of ticks after sending a command that the player can be safely expected to have received the command. By default, tickLag is set to 1, which means we ask all the players to execute the command one tick after we receive it.

We then adjust tickLag for network latency by estimating the number of extra game ticks the player's round-trip time is likely to cause. We use a very simple heuristic for this—dividing the average round-trip time by the time each tick takes and then rounding this value.

You can play around with this heuristic and fine-tune it for accuracy if you like; however, for the purposes of smooth gameplay, it is safer to have a high value. If the value is too low, one of the players with high latency might not receive instructions in time and every other player will have their game freeze until the slower player catches up. If the value is too high, players will have a higher delay between the time that they give the command and the time it actually executes, but the game will never freeze.

It has been found that players are able to get used to network lag and automatically adjust for it as long as the delay is consistent. Any time the lag varies too much or the game freezes unexpectedly, players tend to get frustrated by it. In my experience, the lag compensation from this simple heuristic is sufficient for a very decent game experience.

We also use a simple trick to mask this slight game lag. Units when given commands will immediately acknowledge it with audio feedback, giving the player the illusion that their command was executed immediately. The actual execution still takes place a few hundred milliseconds later, after it has gone to a server, been acknowledged, and sent back to the clients; however this delay isn't as noticeable. Some games use additional animations to go along with the sound, such as a reloading or charging weapon animation to let the player know that the command has started execution.

Next we will modify the multiplayer object's handleWebSocketMessage() method to respond to the server's latency-ping message, as shown in Listing 13-2.

Listing 13-2. Responding to latency-ping with a latency-pong (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

        case "initialize-level":
            multiplayer.currentLevel = messageObject.currentLevel;
            multiplayer.initLevel(messageObject.spawnLocations);
            break;

        case "play-game":
            multiplayer.play();
            break;

        case "latency-ping":
            multiplayer.sendWebSocketMessage({ type: "latency-pong" });
            break;

    }
```

When the browser receives a latency-ping message from the server, it immediately sends back a latency-pong message to the server.

Finally, we will modify the request event handler for the websocket object on the server to start measuring latency when a player connects and to finish measuring latency when the player sends back a latency-pong response, as shown in Listing 13-3.

Listing 13-3. Starting and Finishing Latency Measurement (server.js)

```
wsServer.on("request", function(request) {

    var connection = request.accept();

    console.log("Connection from " + request.remoteAddress + " accepted.");

    // Add the player to the players array
    var player = {
        connection: connection,
        latencyTrips: []
    };

    players.push(player);
```

```
// Send a fresh game room status list the first time player connects
sendRoomList(connection);

// Measure latency for player
measureLatencyStart(player);

// Handle receiving of messages
connection.on("message", function(message) {
    if (message.type === "utf8") {
        var clientMessage = JSON.parse(message.utf8Data);

        // Handle message based on message type
        switch (clientMessage.type) {
            case "join-room":
                joinRoom(player, clientMessage.roomId);
                sendRoomListToEveryone();

                if (player.room.players.length === 2) {
                    // Two players have joined. Initialize the game
                    initializeGame(player.room);
                }

                break;

            case "leave-room":
                leaveRoom(player, clientMessage.roomId);
                sendRoomListToEveryone();
                break;

            case "initialized-level":
                player.room.playersReady++;

                if (player.room.playersReady === 2) {
                    // Both players are ready, Start the game
                    startGame(player.room);
                }

                break;

            case "latency-pong":
                measureLatencyEnd(player);

                // Measure latency at least thrice
                if (player.latencyTrips.length < 3) {
                    measureLatencyStart(player);
                }
                break;
        }
    }
});
```

```
    // Handle closing of connection
    connection.on("close", function() {
        console.log("Connection from " + request.remoteAddress + " disconnected.");

        // Remove the player from the players array
        var index = players.indexOf(player);

        if (index > -1) {
            players.splice(index, 1);
        }

    });

    // Handle closing of connection
    connection.on("close", function() {
        console.log("Connection from " + request.remoteAddress + " disconnected.");

        // Remove the player from the players array
        var index = players.indexOf(player);

        if (index > -1) {
            players.splice(index, 1);
        }

        var room = player.room;

        if (room) {
            // If the player was in a room, remove the player from the room
            leaveRoom(player, room.roomId);

            // Notify everyone about the changes
            sendRoomListToEveryone();
        }
    });
});
```

We start by adding a `latencyTrips` array to the `player` object and calling `measureLatencyStart()` once the player has connected.

We then modify the message handler to handle messages of type `latency-pong`. When the player responds to a `latency-ping` message with a `latency-pong` message, we call the `measureLatencyEnd()` method that we defined earlier. We then check whether we have at least three latency measurements and, if not, call the `measureLatencyStart()` method again.

Now, if you start the server and run the game, the server will make three attempts to measure the latency. You can see the WebSocket communication using the browser's developer console, as shown in Figure 13-1.
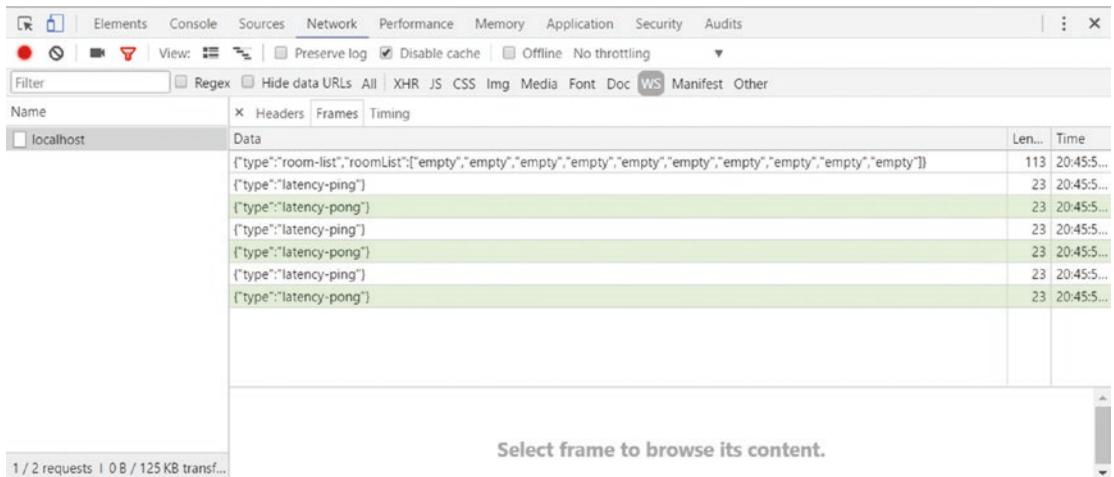
Figure 13-1. *Observing the WebSocket communication in the developer console*

You will notice that the incoming and outgoing messages have different background colors, making them easier to distinguish. The client first receives a message of type room-list, which it uses to update the multiplayer game lobby. It then receives three latency-ping messages and responds by sending three latency-pong messages. All of this happens within the first few seconds after a connection is established.

Now that we have measured latency for the players, it's time to implement sending commands.

Sending Commands

Once the game starts, we will maintain a game clock with a game tick number on both the server and the clients. When a player sends a command to the server, we will send the command back to the clients with instructions to execute the command at a later tick calculated by using tickLag.

We will start by modifying the multiplayer object's handleWebSocketMessage() method to receive commands within a game-tick message, as shown in Listing 13-4.

Listing 13-4. Receiving Commands in game-tick Message (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

        case "initialize-level":
            multiplayer.currentLevel = messageObject.currentLevel;
            multiplayer.initLevel(messageObject.spawnLocations);
            break;
```

387

```
        case "play-game":
            multiplayer.play();
            break;

        case "latency-ping":
            multiplayer.sendWebSocketMessage({ type: "latency-pong" });
            break;

        case "game-tick":
            multiplayer.lastReceivedTick = messageObject.tick;
            multiplayer.commands[messageObject.tick] = messageObject.commands;
            break;
    }
```

When we receive a `game-tick` message from the server containing a list of commands and the tick number on which the commands need to be executed, we save the commands in the `multiplayer.commands` array and then update the `lastReceivedTick` variable.

Next we will implement the game loop and handle sending and processing commands, as shown in Listing 13-5.

Listing 13-5. Handling Commands in the Client (multiplayer.js)

```
play: function() {
    // Run the animation loop once
    game.animationLoop();

    // Instead of animationLoop, use tickLoop, which will coordinate with the server to call
    // animationLoop
    multiplayer.animationInterval = setInterval(multiplayer.tickLoop, game.animationTimeout);

    game.start();
},

sendCommand: function(uids, details) {
    multiplayer.sentCommandForTick = true;
    multiplayer.sendWebSocketMessage({ type: "command", uids: uids, details: details,
    currentTick: multiplayer.currentTick });
},

tickLoop: function() {
    // If the commands for that tick have been received
    // execute the commands and move on to the next tick
    // otherwise wait for server to catch up
    if (multiplayer.currentTick <= multiplayer.lastReceivedTick) {
        var commands = multiplayer.commands[multiplayer.currentTick];

        if (commands) {
            for (var i = 0; i < commands.length; i++) {
                game.processCommand(commands[i].uids, commands[i].details);
            }
        }
```

```
        game.animationLoop();

        // In case no command was sent for this current tick, send an empty command to the
        server
        // So that the server knows that everything is working smoothly
        if (!multiplayer.sentCommandForTick) {
            multiplayer.sendCommand();
        }

        // Move on to the next tick
        multiplayer.currentTick++;
        multiplayer.sentCommandForTick = false;
    }
},
```

First, in the play() method, we set an interval to call the tickLoop() method every 100 milliseconds when the game starts.

Next, in the sendCommand() method, we send a message of type command to the server with the details of the command as well as the UIDs for the command.

The command message also contains the current game tick. This way, the command message can let the server know what game tick the client is currently on. We also set the sentCommandForTick flag to true.

In the tickLoop() method, we check to see whether we have received commands for the current tick. In case we have not, we will wait for the commands to arrive from the server. This is how we ensure that all of the clients act in sync.

If we have received the commands for the tick, we process all the received commands using the game.processCommand() method. We then call the game.animationLoop() method.

In case we have not sent out any commands so far, we also send an empty command to the server. This acts as a heartbeat message to let the server know which game tick was completed even when the player doesn't issue any commands during the game tick.

Finally, we increment the game tick number and clear the sentCommandForTick flag.

With this in place, the client will coordinate with the server by sending commands with sendCommand() and receiving them back via game-tick messages. It will then call animationLoop() for each game tick as long as it has received commands from the server for the game tick. Ideally, if there aren't any network delays, tickLoop() will call animationLoop() every 100 milliseconds and never skip a tick.

Now that the client has been modified to send and receive commands, we will modify the server to handle these commands as well.

We will start by modifying the message event handler on the server to handle messages of type command, as shown in Listing 13-6.

Listing 13-6. Handing Messages of Type command (server.js)

```
// Handle message based on message type
switch (clientMessage.type) {
    case "join-room":
        joinRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();

        if (player.room.players.length === 2) {
            // Two players have joined. Initialize the game
            initializeGame(player.room);
        }

        break;
```

```
    case "leave-room":
        leaveRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();
        break;

    case "initialized-level":
        player.room.playersReady++;

        if (player.room.playersReady === 2) {
            // Both players are ready, Start the game
            startGame(player.room);
        }

        break;

    case "latency-pong":
        measureLatencyEnd(player);

        // Measure latency at least thrice
        if (player.latencyTrips.length < 3) {
            measureLatencyStart(player);
        }
        break;

    case "command":
        if (player.room && player.room.status === "running") {
            if (clientMessage.uids) {
                player.room.commands.push({ uids: clientMessage.uids, details:
                clientMessage.details });
            }

            player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
            player.tickLag;
        }
        break;
}
```

When the server receives a message of type command, we check whether the message has UIDs. If so, we store the commands in the room's commands array. If not, the message is just a heartbeat message with no command that needs saving. We then update the lastTickConfirmed property for the player.

Finally, we will modify the startGame() method in server.js to start the game loop and run the game, as shown in Listing 13-7.

Listing 13-7. Modifying the startGame() Method (server.js)

```
function startGame(room) {
    console.log("Both players are ready. Starting game in room", room.roomId);

    room.status = "running";
    sendRoomListToEveryone();
    // Notify players to start the game
    sendRoomWebSocketMessage(room, { type: "play-game" });
```

```
    room.commands = [];
    room.lastTickConfirmed = { "blue": 0, "green": 0 };
    room.currentTick = 0;

        // Calculate tick lag for room as the max of both players' tick lags
    var roomTickLag = Math.max(room.players[0].tickLag, room.players[1].tickLag);

    room.interval = setInterval(function() {
        // Confirm that both players have sent in commands for up to present tick
        if (room.lastTickConfirmed["blue"] >= room.currentTick && room.
        lastTickConfirmed["green"] >= room.currentTick) {
            // Commands should be executed after the tick lag
            sendRoomWebSocketMessage(room, { type: "game-tick", tick: room.currentTick +
            roomTickLag, commands: room.commands });
            room.currentTick++;
            room.commands = [];
        } else {
            // One of the players is causing the game to lag. Handle appropriately
            if (room.lastTickConfirmed["blue"] < room.currentTick) {
                console.log("Room", room.roomId, "Blue is lagging on Tick:", room.
                currentTick, "by", room.currentTick - room.lastTickConfirmed["blue"]);
            }

            if (room.lastTickConfirmed["green"] < room.currentTick) {
                console.log("Room", room.roomId, "Green is lagging on Tick:", room.
                currentTick, "by", room.currentTick - room.lastTickConfirmed["green"]);
            }
        }
    }, gameTimeout);
}
```

When the game starts, we initialize the commands array, currentTick, and the lastTickConfirmed object for the room. We then calculate the tick lag for the room as the maximum of the tick lag for the two players and save it in the roomTickLag variable.

Next, we start the timer loop for the game using setInterval(). Within this loop, we first check that both players have caught up with the server by sending commands for the present game tick.

If so, we send out a game-tick message to the players with a list of commands and ask them to execute the commands roomTickLag ticks after the current tick. This way, both the players will execute the command at the same time, even if the message takes a little time to reach the players.

We then clear the commands array on the server and increase the currentTick variable for the room.

If the server hasn't received confirmation for the current tick from both the clients, we log a message to the console and do not increment the tick. You can modify this code to check whether the server has been waiting for a long time and, if so, send the players a notification that the server is experiencing lag because of a specific player.

If you start the server and run the game on two different browsers, you should be able to command the units and have your first multiplayer battle, as shown in Figure 13-2.
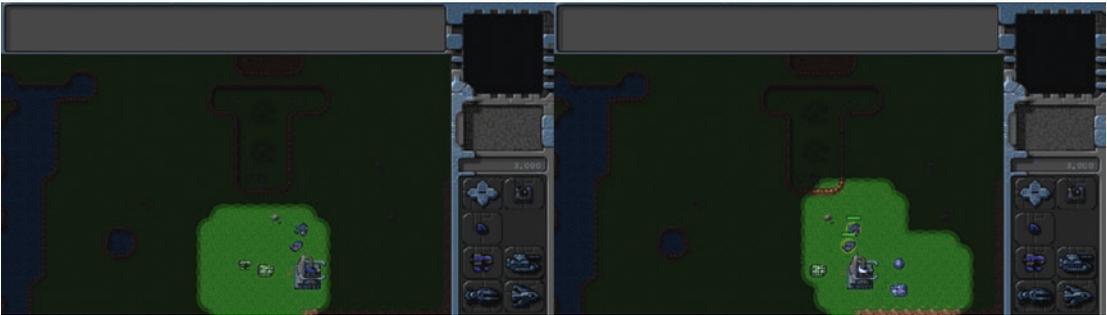
Figure 13-2. *Commanding units in a multiplayer battle*

The multiplayer portion of our game is now working. Right now both the browsers are on the same machine. You can host the code on a web server to access them from different machines. You can also move the server code onto a separate Node.js machine and modify the multiplayer object to point to this new server instead of localhost. If you want to move to a public server, you can find several hosting providers that provide Node.js support, such as EvenNode (https://www.evennode.com) and Nodejitsu (https://www.nodejitsu.com/).

Now that we have implemented sending commands, we will implement ending the multiplayer game.

Ending the Multiplayer Game

The multiplayer game can be ended in two ways. The first is if one of the players defeats the other by satisfying the requirements for the level. The other is if a player either closes the browser or gets disconnected from the server.

Ending the Game When a Player Is Defeated

We will implement ending the game using triggered events just like we did in Chapter 11. This gives us the flexibility to design different types of multiplayer levels such as capture the flag or death match. We are limited only by our imagination.

For now, we will make the level end when one side is completely destroyed. We will start by creating a simple triggered event in the multiplayer level, as shown in Listing 13-8.

Listing 13-8. Trigger for Ending the Multiplayer Level (levels.js)

```
/* Conditional and Timed Trigger Events */
"triggers": [
    {
        "type": "conditional",
        // Check if the player has lost all units and buildings
        "condition": function() {
            for (let i = 0; i < game.items.length; i++) {
                let item = game.items[i];

                if (item.team === game.team) {
                    // Player still has one item in the game
                    return false;
                }
            }
```

```
            return true;
        },
        // Player has lost the game
        "action": function() {
            multiplayer.loseGame();
        }
    },
]
```

In the conditional trigger, we check whether the game.items array contains at least one item belonging to the player. If the player has no items left, we call the loseGame() method.

Next we will add the loseGame() and endGame() methods to the multiplayer object, as shown in Listing 13-9.

Listing 13-9. Adding loseGame() and endgame() Methods (multiplayer.js)

```
loseGame: function() {
    multiplayer.sendWebSocketMessage({ type: "lose-game" });
},

endGame: function(message) {
    game.running = false;
    clearInterval(multiplayer.animationInterval);

    // Show reason for game ending, and on Okay, exit multiplayer screen
    game.showMessageBox(message, multiplayer.closeAndExit);
},
```

In the loseGame() method, we send a message of type lose-game to the server to let it know that the player has lost the game.

In the endGame() method, we clear the game.running flag and the multiplayer.animationInterval interval. We then show a message box with the reason for ending the game and finally call the multiplayer.closeAndExit() method once the Okay button on the message box is clicked.

Next, we will modify the server to handle game ending. This will involve handling the lose-game message from either of the clients, notifying all the clients that the game has ended, and then cleaning up the game-related variables on the server.

We will start by defining a new endGame() method in server.js, as shown in Listing 13-10.

Listing 13-10. The Server endGame() Method (server.js)

```
function endGame(room, message) {
    // Stop the game loop on the server
    clearInterval(room.interval);

    // Tell both players to end game
    sendRoomWebSocketMessage(room, { type: "end-game", message: message });

    // Empty the room
    room.players.forEach(function(player) {
        leaveRoom(player, room.roomId);
    });
    room.status = "empty";

    sendRoomListToEveryone();
}
```

This method handles clearing up all the game-related properties and notifying the clients that the game has ended. We start by clearing the interval for the game loop. We then send the `end-game` message to all the players in the room with the reason for the game ending provided as a `message` parameter. We then remove all the players from the room using the `leaveRoom()` method and set the room status to `empty`. Finally, we send the updated room list to all connected players.

Next, we will modify the message event handler on the server to handle messages of type `lose-game`, as shown in Listing 13-11.

Listing 13-11. Handing Messages of Type lose-game (server.js)

```
// Handle message based on message type
switch (clientMessage.type) {
    case "join-room":
        joinRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();

        if (player.room.players.length === 2) {
            // Two players have joined. Initialize the game
            initializeGame(player.room);
        }

        break;

    case "leave-room":
        leaveRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();
        break;

    case "initialized-level":
        player.room.playersReady++;

        if (player.room.playersReady === 2) {
            // Both players are ready, Start the game
            startGame(player.room);
        }

        break;

    case "latency-pong":
        measureLatencyEnd(player);

        // Measure latency at least thrice
        if (player.latencyTrips.length < 3) {
            measureLatencyStart(player);
        }
        break;

    case "command":
        if (player.room && player.room.status === "running") {
```

```
        if (clientMessage.uids) {
            player.room.commands.push({ uids: clientMessage.uids, details:
            clientMessage.details });
        }

        player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
        player.tickLag;
    }
    break;

case "lose-game":
    if (player.room && player.room.status === "running") {
        endGame(player.room, "The " + player.color + " team has been defeated.");
    }
    break;
}
```

When we receive a lose-game message from one of the players, we call the endGame() method with the reason for ending the game.

Finally, we will modify the multiplayer object's handleWebSocketMessage() method to receive messages of type end-game, as shown in Listing 13-12.

Listing 13-12. Receiving Messages of Type end-game (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

        case "initialize-level":
            multiplayer.currentLevel = messageObject.currentLevel;
            multiplayer.initLevel(messageObject.spawnLocations);
            break;

        case "play-game":
            multiplayer.play();
            break;

        case "latency-ping":
            multiplayer.sendWebSocketMessage({ type: "latency-pong" });
            break;
```

```
    case "game-tick":
        multiplayer.lastReceivedTick = messageObject.tick;
        multiplayer.commands[messageObject.tick] = messageObject.commands;
        break;

    case "end-game":
        multiplayer.endGame(messageObject.message);
        break;
    }
},
```

When the client receives an end-game message, we call multiplayer.endGame() with the reason provided in the message.

If you start the server and run the game, you should see a message box when one of the players destroys all of the other player's units and buildings, as shown in Figure 13-3.

Figure 13-3. *Game ends when one player defeats the other*

If you click the Okay button, you should be taken back to the main game menu. You will notice that when a game ends, the lobby automatically shows the room as empty so that the next set of players can join the room.

We will also need to end the game whenever one of the players closes the browser or is disconnected from the server, since the game cannot continue with just one player.

Ending the Game When a Player Is Disconnected

Whenever a player disconnects from the server while playing a game, it will trigger a WebSocket close event on the server. We will handle this disconnect by modifying the close event handler on the server, as shown in Listing 13-13.

Listing 13-13. Handling Player Disconnects (server.js)

```
// Handle closing of connection
connection.on("close", function() {
    console.log("Connection from " + request.remoteAddress + " disconnected.");

    // Remove the player from the players array
    var index = players.indexOf(player);

    if (index > -1) {
        players.splice(index, 1);
    }

    var room = player.room;

    if (room) {
        var status = room.status;

        // If the player was in a room, remove the player from the room
        leaveRoom(player, room.roomId);

        // If the game had started or was already running, end the game and notify other player
        if (status === "running" || status === "starting") {
            var message = "The " + player.color + " player has been disconnected.";

            endGame(room, message);
        }

        // Notify everyone about the changes
        sendRoomListToEveryone();
    }

});
```

If the player is in a room, we remove the player from the room and send the updated room list to everyone. If the game was running, we also call the endgame() method with the reason that the player has disconnected.

If you start the server and begin a multiplayer game, you should see a disconnect message when either of the players gets disconnected, as shown in Figure 13-4.
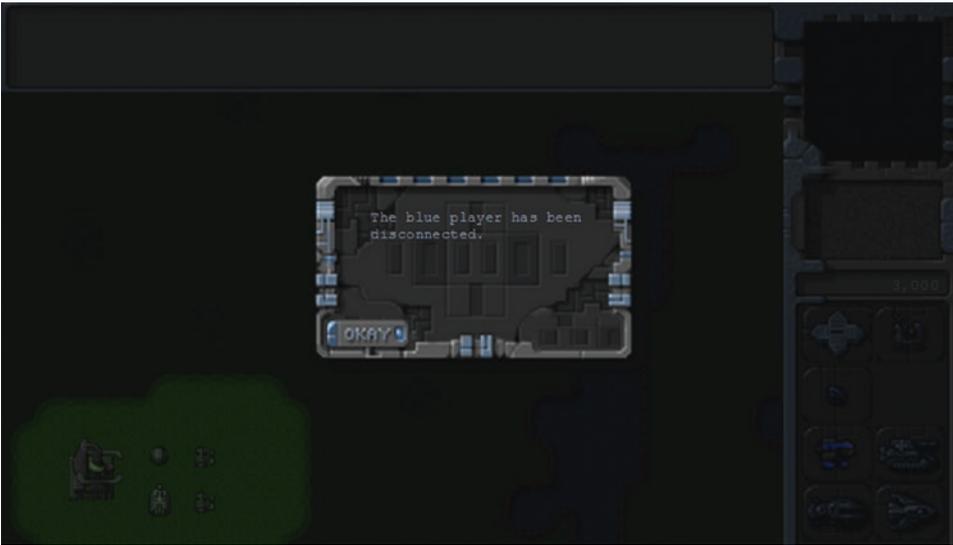
Figure 13-4. *Message shown when a player gets disconnected*

Clicking the Okay button will take you back to the main menu screen. Again, the lobby automatically shows the room as empty so that the next set of players can join the room.

The last thing we will handle is ending the game if a connection error occurs and the connection is lost.

Ending the Game When a Connection Is Lost

Whenever the client gets disconnected from the server or an error occurs, it will trigger either an error event or a close event on the client. We will handle this by implementing these event handlers within the start() method of the multiplayer object, as shown in Listing 13-14.

Listing 13-14. Handling Connection Errors (multiplayer.js)

```
start: function() {
    if (!window.WebSocket) {
        game.showMessageBox("Your browser does not support WebSocket. Multiplayer will not
        work.");

        return;
    }

    const websocketUrl = "ws://" + (window.location.hostname || "localhost") + ":8080";

    this.websocket = new WebSocket(websocketUrl);

    this.websocket.addEventListener("open", multiplayer.handleWebSocketOpen);
    this.websocket.addEventListener("message", multiplayer.handleWebSocketMessage);

    this.websocket.addEventListener("close", multiplayer.handleWebSocketConnectionError);
    this.websocket.addEventListener("error", multiplayer.handleWebSocketConnectionError);
},
```

```
// Display an error message and end game in case of a connection error
handleWebSocketConnectionError: function() {
    multiplayer.endGame("Error connecting to multiplayer server.");
},
```

For both the events, we call the endGame() method with an error message. We will also remove these handlers when we exit multiplayer by modifying the closeAndExit() method, as shown in Listing 13-15.

Listing 13-15. Clearing Error Handlers (multiplayer.js)

```
closeAndExit: function() {
    // Clear all handlers and close connection
    multiplayer.websocket.removeEventListener("open", multiplayer.handleWebSocketOpen);
    multiplayer.websocket.removeEventListener("message", multiplayer.handleWebSocketMessage);

    multiplayer.websocket.removeEventListener("close", multiplayer.
    handleWebSocketConnectionError);
    multiplayer.websocket.removeEventListener("error", multiplayer.
    handleWebSocketConnectionError);

    multiplayer.websocket.close();

    // Enable room list and join button
    document.getElementById("multiplayergameslist").disabled = false;
    document.getElementById("multiplayerjoin").disabled = false;

    // Show the starting menu layer
    game.hideScreens();
    game.showScreen("gamestartscreen");
},
```

If you run the game now and shut down the server to cause a server disconnect, you should see an error message, as shown in Figure 13-5.

Figure 13-5. *Message shown in case of a connection error*

If there is a problem with the connection while the player is either in the lobby or playing a game, the browser will now display this error message and then return to the main game screen.

A more robust implementation would include trying to reconnect to the server within a timeout period and then resuming the game. We can achieve this by passing a reconnect message with a unique player ID to the server and handling the message appropriately on the server side. However, we will stick with this simpler implementation for our game.

Before we wrap up the multiplayer portion of our game, we will implement one last feature in our game: player chat.

Implementing Player Chat

The first thing that we need to do to enable chat is create an input area to enter these chat messages, and style it appropriately.

We will start by defining an input box for chat messages inside the gameinterfacescreen layer in index.html, as shown in Listing 13-16.

Listing 13-16. Adding Input Box for Chat Message (index.html)

```
<div id="gameinterfacescreen" class="gamelayer">
    <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
    <img src="images/screens/interface-right-game.png" class="right-panel" draggable="false">
    <div id="gamemessages"></div>
    <div id="callerpicture"></div>
    <div id="cash"></div>
    <div id="sidebarbuttons">
        <input type="button" id="starport" title = "Starport">
        <input type="button" id="ground-turret" title = "Turret">
        <input type="button" id="harvester" title = "Harvester">
        <input type="button" id="scout-tank" title = "Scout Tank">
```

```
        <input type="button" id="heavy-tank" title = "Heavy Tank">
        <input type="button" id="chopper" title = "Copter">
        <input type="button" id="wraith" title = "Wraith">
    </div>
    <canvas id="gamebackgroundcanvas"></canvas>
    <canvas id="gameforegroundcanvas"></canvas>
    <input type="text" id="chatmessage">
</div>
```

Next, we will add some extra styles for the chat message input to `styles.css`, as shown in Listing 13-17.

Listing 13-17. Styles for Chat Message Input Box (styles.css)

```css
#chatmessage {
    position: absolute;
    top: 460px;

    background: rgba(0, 255, 0, 0.1);

    color: green;

    border: 1px solid green;

    /* Hide chat input by default */
    display: none;
}

#chatmessage:focus {
    outline: none;
}
```

Finally, we will modify the `resize()` method of the `game` object to resize the chat input whenever the window is resized, as shown in Listing 13-18.

Listing 13-18. Resizing the Input for Chat Messages (game.js)

```javascript
resize: function() {

    var maxWidth = window.innerWidth;
    var maxHeight = window.innerHeight;

    var scale = Math.min(maxWidth / 640, maxHeight / 480);

    var gameContainer = document.getElementById("gamecontainer");

    gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";

    // What is the maximum width we can set based on the current scale
    // Clamp the value between 640 and 1024
    var width = Math.max(640, Math.min(1024, maxWidth / scale ));
```

```
    // Apply this new width to game container and game canvas
    gameContainer.style.width = width + "px";

    // Subtract 160px for the sidebar
    var canvasWidth = width - 160;

    // Set a flag in case the canvas was resized
    if (game.canvasWidth !== canvasWidth) {
        game.canvasWidth = canvasWidth;
        game.canvasResized = true;
    }

    // Set the chatmessage input to the same value
    document.getElementById("chatmessage").style.width = canvasWidth + "px";

    game.scale = scale;
},
```

Now that we have set up the chat input box, we need to allow the player to enter chat messages and send them to the server. We will start by adding an event handler to listen for keydown events so we can handle keyboard input for chat messages, as shown in Listing 13-19.

Listing 13-19. Handling Keyboard Input for Chat Messages (game.js)

```
handleKeyboardInput: function(ev) {
    // Chatting only allowed in multiplayer when game is running
    if (game.type === "multiplayer" && game.running) {

        let chatMessage = document.getElementById("chatmessage");
        // Invisible elements have a null offsetParent
        let chatInputVisible = chatMessage.offsetParent !== null;

        if (ev.key === "Enter") {
            if (chatInputVisible) {
                // Send any text in the message input
                let message = chatMessage.value.trim();

                if (message) {
                    multiplayer.sendChatMessage(message);
                }

                // Clear the input and hide it
                chatMessage.value = "";
                chatMessage.style.display = "none";
            } else {
                // Show the input and set focus on it
                chatMessage.style.display = "inline";
                chatMessage.focus();
            }
        } else if (ev.key === "Escape") {
```

```
        if (chatInputVisible) {
            // Clear the input and hide it
            chatMessage.value = "";
            chatMessage.style.display = "none";
        }
    }
}
},
```

Whenever a key is pressed, we first confirm that a multiplayer game is running and exit if it is not. If the Enter key is pressed, we check whether the chatmessage input box is visible. We then either display the input box if it is not already visible, or send the contents of the message box to the server using the sendChatMessage() method if it is already visible. We then clear the contents of the input box and hide it. If the Escape key is pressed while the input box is visible, we clear the contents of the input box and hide it.

This way, the player can press Enter to start chatting, type a message, and press Enter again to send the message. If the player starts typing a message and then decides not to send it, pressing Escape will clear the message.

We will need to add an event listener to listen for the keydown event and call the handleKeyboardInput() method, as shown in Listing 13-20.

Listing 13-20. Listening for keydown Events (game.js)

```
// Initialize game once page has fully loaded
window.addEventListener("load", function() {
    game.resize();
    game.init();
}, false);

window.addEventListener("resize", function() {
    game.resize();
});
```

window.addEventListener("keydown", game.handleKeyboardInput);

Finally, we will define a sendChatMessage() method inside the multiplayer object as shown in Listing 13-21.

Listing 13-21. Handing Messages of Type chat (multiplayer.js)

```
sendChatMessage: function(message) {
    multiplayer.sendWebSocketMessage({ type: "chat", message: message });
}
```

The sendChatMessage() method sends the player's message to the server with type set to chat.

Now that the client can send chat messages to the server, we will modify the message event handler on the server to handle messages of type chat, as shown in Listing 13-22.

Listing 13-22. Handing Messages of Type chat (server.js)

```
// Handle message based on message type
switch (clientMessage.type) {
    case "join-room":
        joinRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();

        if (player.room.players.length === 2) {
            // Two players have joined. Initialize the game
            initializeGame(player.room);
        }

        break;

    case "leave-room":
        leaveRoom(player, clientMessage.roomId);
        sendRoomListToEveryone();
        break;

    case "initialized-level":
        player.room.playersReady++;

        if (player.room.playersReady === 2) {
            // Both players are ready, Start the game
            startGame(player.room);
        }

        break;

    case "latency-pong":
        measureLatencyEnd(player);

        // Measure latency at least thrice
        if (player.latencyTrips.length < 3) {
            measureLatencyStart(player);
        }
        break;

    case "command":
        if (player.room && player.room.status === "running") {
            if (clientMessage.uids) {
                player.room.commands.push({ uids: clientMessage.uids, details:
                clientMessage.details });
            }

            player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
            player.tickLag;
        }
        break;
```

```
    case "lose-game":
        if (player.room && player.room.status === "running") {
            endGame(player.room, "The " + player.color + " team has been defeated.");
        }
        break;

    case "chat":
        if (player.room && player.room.status === "running") {
            // Sanitize the message to remove any HTML tags
            var cleanedMessage = clientMessage.message.replace(/[<>]/g, "");

            sendRoomWebSocketMessage(player.room, { type: "chat", from: player.color,
                message: cleanedMessage });
        }
        break;
}
```

When we receive a message of type chat from a player, we send back a message of type chat to all the players in the room, with a from property set to the player's color and a message property set to the message we just received.

Ideally, you should validate the chat message so that a player cannot send malicious HTML and script tags inside chat messages. For now, we use a simple regular expression to strip out any HTML tags from the message before sending it back.

Finally, we will modify the multiplayer object's handleWebSocketMessage() method to receive messages of type chat, as shown in Listing 13-23.

Listing 13-23. Receiving Messages of Type chat (multiplayer.js)

```
handleWebSocketMessage: function(message) {
    var messageObject = JSON.parse(message.data);

    switch (messageObject.type) {
        case "room-list":
            multiplayer.updateRoomStatus(messageObject.roomList);
            break;

        case "joined-room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;

        case "initialize-level":
            multiplayer.currentLevel = messageObject.currentLevel;
            multiplayer.initLevel(messageObject.spawnLocations);
            break;

        case "play-game":
            multiplayer.play();
            break;
```

405

```
        case "latency-ping":
            multiplayer.sendWebSocketMessage({ type: "latency-pong" });
            break;

        case "game-tick":
            multiplayer.lastReceivedTick = messageObject.tick;
            multiplayer.commands[messageObject.tick] = messageObject.commands;
            break;

        case "end-game":
            multiplayer.endGame(messageObject.message);
            break;

        case "chat":
            game.showMessage(messageObject.from, messageObject.message);
            break;
    }
},
```

If you start the server and play a multiplayer game now, you should be able to send chat messages from one player to the other, as shown in Figure 13-6.

Figure 13-6. *Chat between players during multiplayer*

We now have a working player chat for multiplayer. With this last change, we can consider our multiplayer game wrapped up.

Summary

We have come a long way over the course of this book. We started by looking at the basic elements of HTML5 needed to build games, such as drawing and animating on the canvas, playing audio, and using sprite sheets.

We then used these basics to build a Box2D physics engine–based game called *Froot Wars*. In the process, we looked at creating splash screens, asset loaders, and customizable levels. We then examined the building blocks of the Box2D engine and integrated Box2D with the game to create realistic-looking physics. We then added sound effects and background music to create a very polished game. Finally, we looked at converting a game for mobile devices by making it responsive to different resolutions and aspect ratios, handle touch events, and use WebAudio.

After that, we built a complete real-time strategy game called Last Colony. Building upon ideas from the previous chapters, we first created a single-player game world with large pannable levels and different types of entities. We added intelligent movement using pathfinding and steering, combat using states and triggers, and even a game economy. We then saw how this framework could be used to tell a compelling story over several levels of a single-player campaign.

Finally, over the last two chapters, we used Node.js and WebSockets to add multiplayer support to our game. We started by looking at the basics of WebSocket communication and using it to create a multiplayer game lobby.

We then implemented a framework for multiplayer gameplay using the lock-step networking model, which also compensated for network latency while maintaining game synchronization. We handled connection errors as well as game completion using triggered events. Finally, we built a chat system to send messages between the players.

In the final chapter of this book, I'd like to share some of the resources, tools, and processes that I have used over the years to build a fairly large variety of games. Incorporating some of these ideas into your own game development should help in making your future game development faster and streamlined, so you can feel comfortable taking on larger and more ambitious projects.

CHAPTER 14

Essential Game Developer Toolkit

Over the past few years, I have worked on a large number and variety of game projects for clients, including endless runner games, racing games, base-defense games, arcade games, puzzle games, educational games, and different types of multiplayer games.

During this time, I discovered two very important things about game development. The first is that most games consist of large portions of similar, repetitive code, and a lot of games can be developed very quickly by reusing common components. The second thing that I realized is that the more I automated and streamlined my workflow and eliminated repetitive and boring tasks, the easier it became to take on larger and much more challenging game projects.

I specifically chose the two games that I did for this book because together they cover a set of components and techniques that can be reused in a very large variety of games. With some tweaks, you should be able to create most typical smaller mobile games such as puzzle, physics, and tower defense games as well as larger projects such as role-playing games.

While the actual gameplay may change, the basic infrastructure that you need for common tasks such as loading assets, selecting and loading levels, animating and commanding entities, playing sound, and displaying scores should remain the same across most games that you develop. Even the multiplayer server design should remain similar for a large variety of multiplayer games.

You should be able to reuse large portions of code from these sample games to get you started on your projects, so you can focus on the creation and design aspects instead of getting bogged down in writing the basic infrastructure each time you make a new game. You will also find that as you keep making games, creating newer games will become much more easy and effortless, as you start developing an intuition for which building blocks to combine, and learn to reuse these blocks instead of trying to create everything from scratch each time.

However, there are a few simple things that you can do to make your game development even faster, as discussed in this chapter:

- Use a code editor specifically customized for web and game development

- Write clean and modular code for easier debugging, testing, and reuse across projects

- Set up a development workflow to automate any repetitive tasks with a decent set of tools

Customizing Your Code Editor

As a game developer, your code editor is where you will be spending a lot of your time. Choosing the right editor and customizing it for your needs is essential if you don't want to waste a significant portion of this time struggling with your editor to do common repetitive tasks.

Over the years, I have used a variety of code editors, and at any given time you will find at least a dozen extremely popular code editors with comparable features, for any given language.

My current favorite editor is Visual Studio Code (`https://code.visualstudio.com/`), a free and open source code editor developed by Microsoft for Windows, macOS, and Linux. Even though VS Code has been around only since 2015, it has already developed a very large fan following because of its speed, customizability, plug-in support, and steadily growing list of features.

The editor is completely open source and has been written largely in JavaScript, making it easy to modify and extend. As a result, it also has a large community that continues to develop new extensions and features for it.

Some of my favorite features, and the reason that I recommend Visual Studio Code, are

- Syntax highlighting and code completion

- Custom extensions

- Git integration

- Integrated debugging

Syntax Highlighting and Code Completion

Syntax highlighting is one of those features that everyone now expects in any decent code editor. It improves the readability of the code by allowing you to easily distinguish between objects, methods, comments, and quoted text using color, making it easier to read, understand, and navigate the code. In fact, I consider this feature so essential that I would recommend dropping your current editor if it doesn't support syntax highlighting.

Another feature that VS Code has is IntelliSense, the ability to provide intelligent code completion, parameter information, and member lists. This feature lets you easily see the member functions of an object as you type, making it very easy to find the method or property you need, see any parameters for the methods, and reduce unnecessary typing (see Figure 14-1).
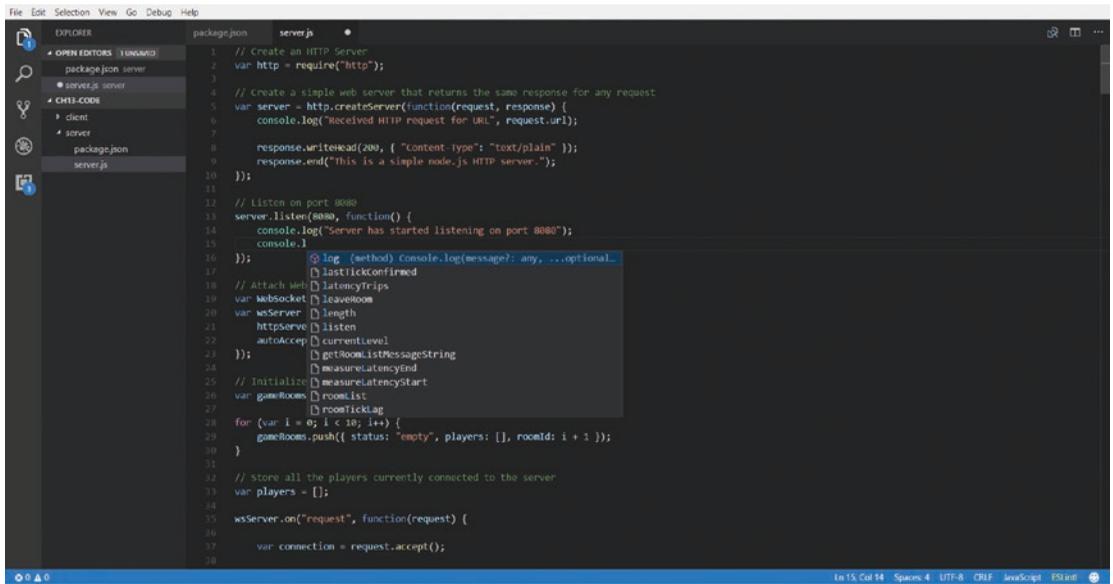
Figure 14-1. *Syntax highlighting and code completion in action*

One advantage of working with an editor that supports code completion is that you can start using descriptive names for all your methods and variables, since you no longer need to worry about typing the variable again and again. This makes your code much more readable, and easier to maintain. Another advantage is that you get hints when you don't remember the exact name of a variable or parameter, so you don't have to look up the API each and every time.

Custom Extensions

Visual Studio Code has a complete API for writing your own custom extensions in JavaScript. You can read more about the extensions API at https://code.visualstudio.com/docs/extensions/overview.

These extensions allow you to add all kinds of features to the editor. There are already extensions for verifying that your code is well written and error free, for automatically formatting your code, and even one to automatically add browser vendor prefixes to your CSS code. All of these extensions, and countless more, can be installed using the Extensions sidebar inside the editor (see Figure 14-2).
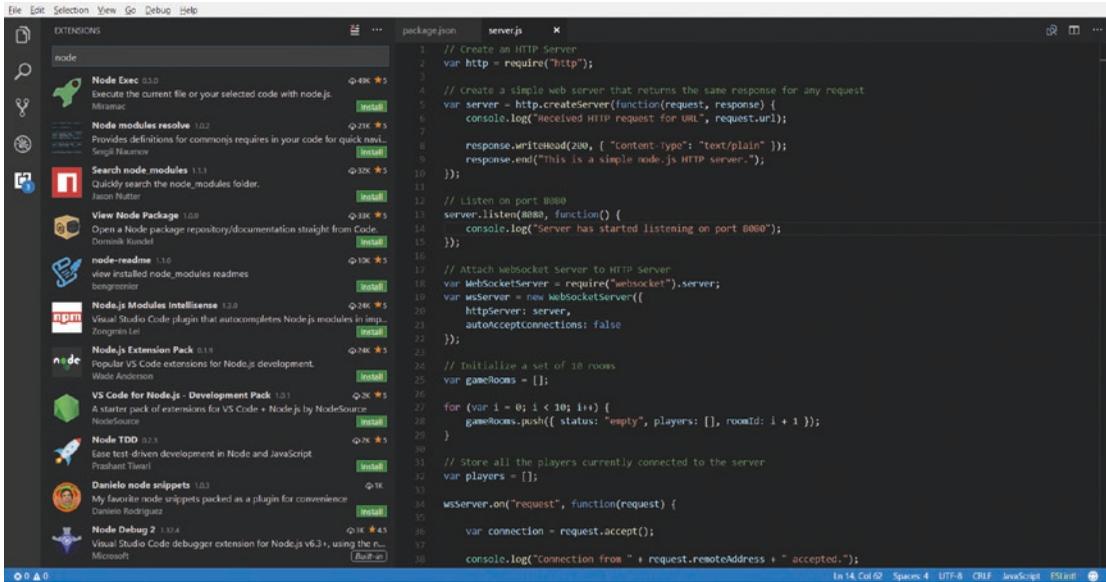
Figure 14-2. *Finding and installing extensions*

While you can explore the extensions library and install extensions as needed, the extensions that I strongly recommend installing are for linting and code snippets.

Linting

Linting is the process of running a program to analyze your code for potential errors. This includes logical and syntactical errors, as well as formatting and stylistic errors such as nonadherence to coding standards.

VS Code has extensions that will allow you to monitor your HTML, CSS, and JavaScript code for lint. Installing the lint plug-ins—ESLint (http://eslint.org/) for JavaScript, HTMLHint (http://htmlhint.com/) for HTML, and Stylelint (https://stylelint.io/) for CSS—allows the editor to easily prompt you to fix your errors, almost like a spell checker in a typical word processor (see Figure 14-3).
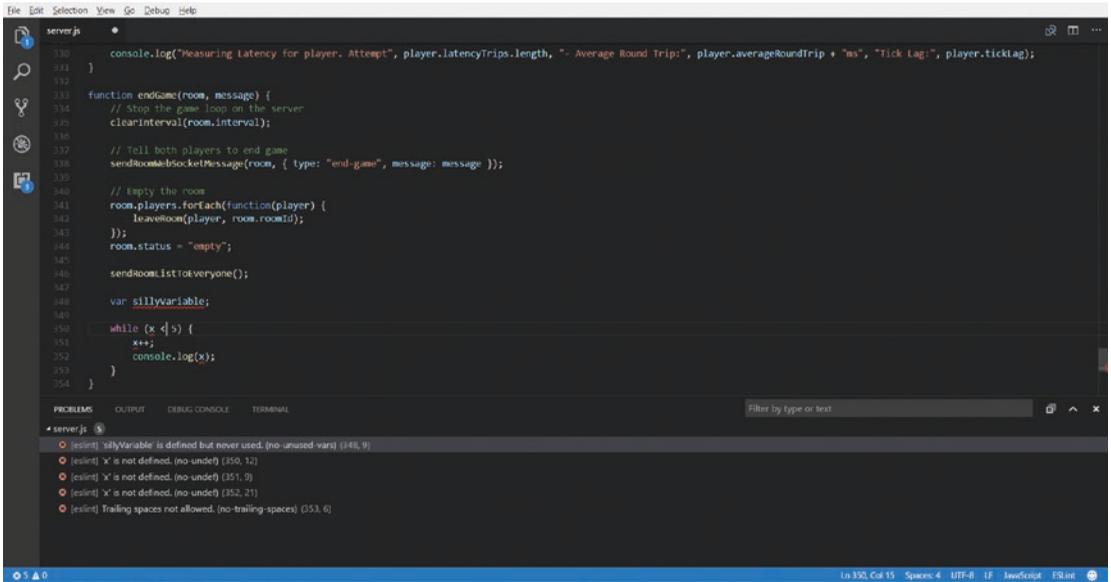
Figure 14-3. *Typical errors reported by a linting tool*

As you can see, having the editor track possible errors in your code can be extremely helpful. ESLint also has the ability to automatically fix any auto-fixable errors such as spacing or indentation, which is very convenient.

You can customize each of these linting tools by specifying the rules that you want to apply for your code inside their respective configuration files. A typical ESLint configuration file is shown in Listing 14-1.

Listing 14-1. A Typical ESLint Configuration File

```
{
    "env": {
        "browser": true,
        "jquery": true
    },
    "globals": {
        "Box2D": true
    },
    "extends": "eslint:recommended",
    "rules": {
        "no-const-assign": "error",
        "no-this-before-super": "error",
        "no-undef": "error",
        "no-unreachable": "error",
        "no-unused-vars": "error",
        "constructor-super": "error",
        "valid-typeof": "error",
        "no-console": "off",
        "indent": ["error", 4, { "SwitchCase": 1 }],
        "quotes": ["error", "double"],
```

```
        "semi": ["error","always"],
        "no-multi-spaces": "error",
        "no-trailing-spaces": "error",
        "space-infix-ops": "error",
        "space-unary-ops": "error",
        "func-call-spacing": ["error", "never"],
        "array-bracket-spacing": ["error", "never"],
        "block-spacing": "error",
        "brace-style": "error",
        "comma-spacing": ["error", { "before": false, "after": true }],
        "key-spacing": "error",
        "newline-after-var": ["error", "always"],
        "newline-before-return": "error",
        "no-multiple-empty-lines": "error",
        "no-tabs": "error",
        "space-before-blocks": "error",
        "object-curly-spacing": ["error", "always"],
        "eqeqeq": "error",
        "curly": "error",
        "keyword-spacing": "error"
    }
}
```

The configuration rules allow you to be as strict as you want with your coding standards. You can specify whether you want to enforce spaces before curly braces, or the number of spaces used when you press the Tab key.

The complete list of rules for each of these tools is available at their respective websites. For example, the list of rules for ESLint is available at https://eslint.org/docs/rules/.

Obviously, you don't need to use all of these rules. You are free to pick and choose the ones that you would like to enforce in your project.

In fact, all the code in this book was checked and formatted using these linting tools with a fairly strict set of rules, which is why everything from the indentation style to the spaces around brackets is consistent all through the book.

While in the short term these linting tools will help you to quickly fix your code and maintain a consistent coding convention, in the long term they will also train you to naturally write clean and error-free code that follows the latest best practices.

Code Snippets

Another useful feature is the ability to define code snippets and assign shortcut letters to access them. A typical example in JavaScript would be to map the letters cl as a shortcut for the statement console. log(""). You would then just type the shortcut and press the Tab key to let the editor automatically replace the shortcut with the entire statement.

In addition to defining code snippets that you often reuse (such as a basic HTML5 file template or the asset loader object template), you can also install entire packs of commonly used snippets as extensions.

While the use of code snippets might not seem like much, being able to generate multiple lines of code by just typing a couple of letters will give you an incredible boost in productivity and development speed.

In addition to these two categories of extensions, I also recommend keeping an eye out for new extensions that you might find useful.

Git Integration

Git (https://git-scm.com/) is a free and open source version-control system designed to keep track of changes in source code while working with multiple contributors.

Git can be used via either the command line or any one of a large number of graphical user interfaces such as GitKraken (https://www.gitkraken.com/) or SourceTree (https://www.sourcetreeapp.com/).

Visual Studio Code also provides an integrated Git interface, shown in Figure 14-4, which allows you to handle all common Git tasks without leaving your editor.
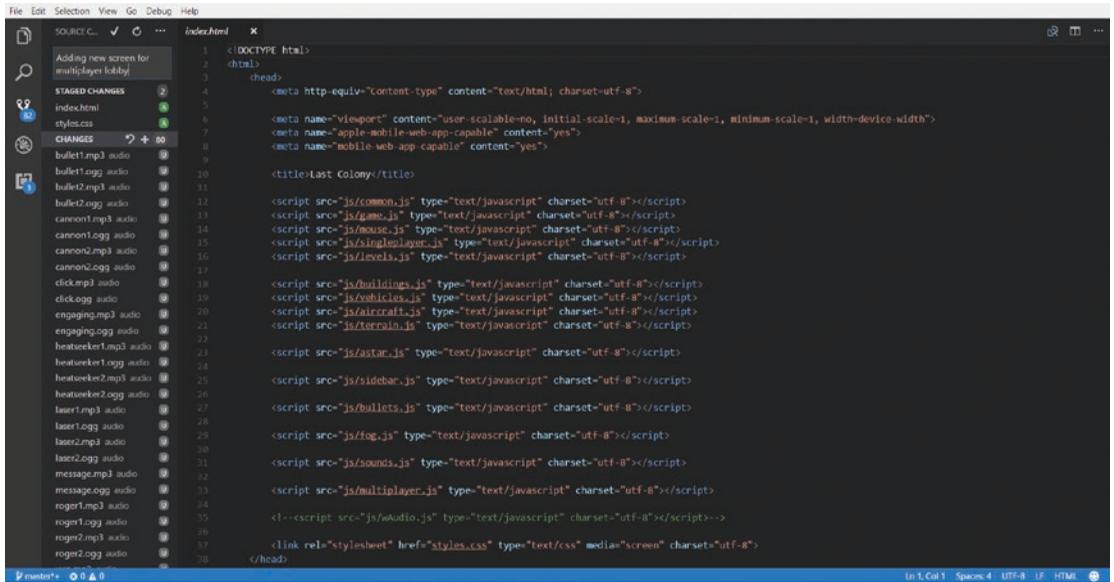
Figure 14-4. *Using Git from inside VS Code*

If you plan to collaborate with other developers, intend to share your code on an online repository like GitHub, or would like to keep track of file changes in your code, then learning to use Git is a must.

If you have never used Git before and would like to learn to use Git, I'd recommend reading the book *Pro Git, Second Edition*, by Scott Chacon and Ben Straub. A free online version of this book is available at https://git-scm.com/book/en/v2.

Integrated Debugging

Another useful feature in Visual Studio Code is the ability to debug your code. You can easily add breakpoints to your code, step through lines of code, and watch the values of variables from inside the editor window (see Figure 14-5).
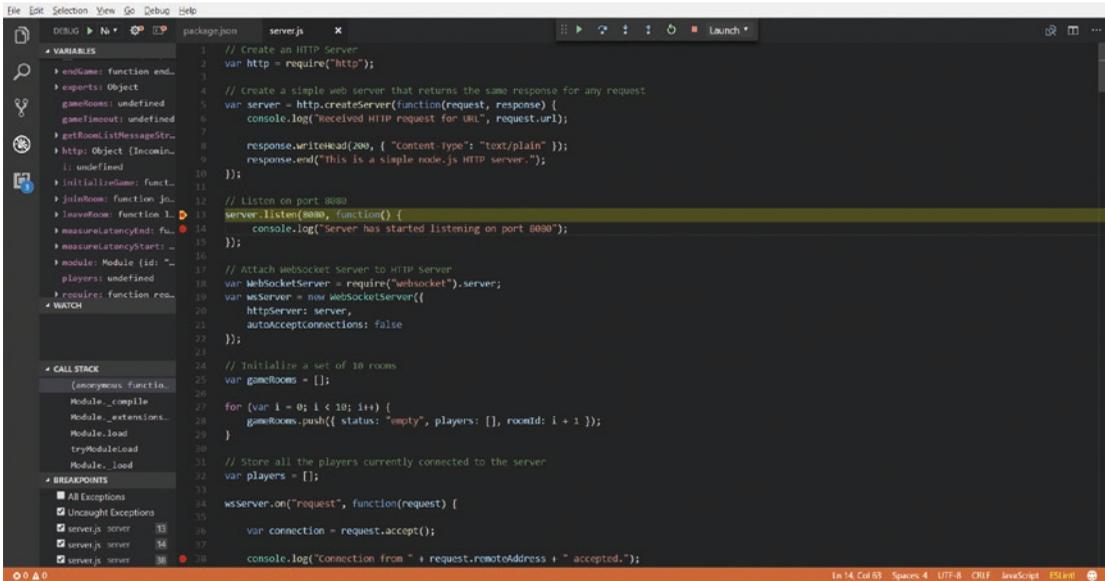
Figure 14-5. *Debugging code from inside VS Code*

There are also extensions to integrate VS Code with various browser debuggers, so you can debug not just your Node.js JavaScript code, but also your browser JavaScript code. You can read more about the debugging feature of VS Code here: https://code.visualstudio.com/docs/editor/debugging.

These are just a few of the features that make Visual Studio Code my current favorite editor. It has tons of other features that I use regularly, such as the ability to use multiple cursors, to easily find and jump to a function definition, to find and replace text using regular expressions, or to open any file just by typing a few letters from the file's name.

Keep in mind that there are several other editors that can be customized to provide similar functionality. Feel free to explore and try different editors until you find one that is perfect for you. However, no matter which editor you eventually decide to use, make sure that it contains at least some of these essential features, so you can be as productive as possible.

Writing Modular Code

Another thing that will help you massively as a game developer is learning to write clean, modular, and reusable code. A few of the more important rules for this, which you have already seen used in this book, are

- *Minimize code repetition*: Whenever you notice code that will be needed in more than one place, immediately move it into a separate method so it can be called from wherever it is needed. Having common code in a single place makes it easier to modify and maintain, and putting it inside a method with a clearly understandable name makes your code easier to read. For example, in Last Colony, we moved common methods like `addItem()` and `loadItem()` to `common.js` so they could be used by all the game item types without needing to repeat the code in the individual item files.

- *Convert useful code into reusable modules*: If some code looks like it can be used in multiple games, abstract it out into a separate module for easy reuse. For example, in our games, we moved all the asset-loading code into a separate `loader` object that we used for both Froot Wars and Last Colony. The loader has been designed to be easily usable for any game.

- *Make your code easily readable*: The closer your code looks to regular English, the easier it becomes to read, debug, and maintain. This means using descriptive names for all your methods and variables so if someone else were to see these methods, they would easily be able to tell what was going on. For example, in Last Colony we used method names like `showMissionBriefing()`, `updateRoomStatus()`, and `createTerrainGrid()`. In addition to this, any time you have a section of code that isn't self-explanatory, start with a comment briefly explaining what the code is supposed to do. This will help you when reviewing or debugging the code, as well as help other developers in understanding the code.

Following just these basic principles should go a long way in helping you write readable, reusable, and maintainable code. You will find that clean code tends to have fewer bugs, and is much easier to modify or extend. Learning to write clean code is one of the essential foundations for collaborating with other developers to work on large projects.

If you would like to read more on the topic, you can look at books such as *The Art of Readable Code* by Dustin Boswell and Trevor Foucher. Another good resource is the book *Clean Code* by Robert C. Martin. While the book is written with Java in mind, the ideas from the book have been adapted for JavaScript in a guide for producing readable, reusable, and refactorable JavaScript, which you can read at https://github.com/ryanmcdermott/clean-code-javascript.

Automating Your Development Workflow

Automating your tasks can give you huge benefits. The most obvious one is that it will save your time and effort in completing difficult tasks. But more importantly, by eliminating human intervention you also reduce human error. There is no longer a chance of an accidental space or a misspelling causing your game to break and then needing a long time to debug—something all too common when we have to do long and boring tasks manually. Lastly, being able to do tasks fast and accurately allows you to scale up the scope of your project.

A very simple example of this automation in action is our using a Node.js script to convert the Last Colony Tiled editor output into the map format for our game. Done manually, this could have taken a lot of effort, with a real chance of a copy-paste error. However, with the metadata generation automated, it is possible to delegate the task of map design to an artist or level designer, who just uses the map editor to draw the maps using a drawing interface.

417

You could then go on to generate the output for all of the maps in a single script that iterates through all the map files in the folder and generates the finished game files in just one quick run. Now, instead of having to limit yourself to five or ten slowly created levels, you could look at the possibility of hundreds of auto-generated levels designed by multiple non-technical level designers, safe in the knowledge that every level can be automatically converted and added to your game.

I personally tend to automate or at least semi-automate any task that I find annoying or distracting. For example, when writing this book I had to take lots of screenshots in each chapter and then crop them very precisely for use within the book. Knowing that all the game images needed to be cropped at the same offset and with the same dimensions, I wrote a simple script that used the ImageMagick toolkit to crop the images for me. This meant I did not need to interrupt my writing workflow just to open an image editor and crop an image, saving me time and effort and allowing me to focus on the task at hand.

In my larger game projects, I tend to automate everything from generating audio and image sprite sheets and minifying and compressing game code to building the final-release game folder for deployment on the server.

Essential Tools for a Streamlined Workflow

You can easily start automating the tasks within your own projects by using a few useful command-line utilities and tools. Most of the tools in my regular toolkit can be broken down into five categories: image handling, audio/video handling, code linting and compression, servers, and build automation.

Image Handling

In my experience, creating game art and assets is probably one of the most time-consuming parts of game development. This includes everything from drawing the initial artwork and preparing the art for game use to creating sprite sheets and compressed image files for the final game.

There are three image-handling tools that I consider essential for game development:

- *Image editor*: One thing that you definitely need is a decent image editor for creating and modifying your artwork. Both Adobe Photoshop (www.adobe.com/products/ photoshop.html) and the GNU Image Manipulation Program, or GIMP (https:// www.gimp.org/), are great options for professional work. Both have a very large community for support and instructional resources, and allow for scripting and automation of repetitive tasks via macros.

- *Command-line image editing*: ImageMagick (https://www.imagemagick.org/) is my favorite command-line tool for image manipulation. As mentioned on its website, you can use ImageMagick to resize, flip, mirror, rotate, distort, shear, and transform images, adjust image colors, apply various special effects, and draw text, lines, polygons, ellipses, and Bézier curves. You can even use it to combine images into sprite sheets, crop and resize images, and convert between different image formats. All of this can be done via the command line or even JavaScript using the node-imagemagick module (https://github.com/yourdeveloper/node-imagemagick), making it easy to automate common image-manipulation tasks.

- *Sprite sheet creation tool*: While ImageMagick can be used for creating simpler sprite sheets, another favorite tool for creating more complex sprite sheets with different image sizes and types is TexturePacker (https://www.codeandweb.com/ texturepacker). It can combine your game artwork into sprite sheets, intelligently position the images within the sprite sheet so they occupy the least possible space, and generate metadata about the sprite position. It also comes with a command-line tool to allow automation of sprite sheet generation.

Code Linting and Compression

We have already discussed code linting and writing clean code briefly when talking about code editors. Most editor linting plug-ins just serve as front ends for these command-line tools. The linting tools I currently use are HTMLHint (`http://htmlhint.com/`) for HTML, Stylelint (`https://stylelint.io/`) for CSS, and ESLint (`http://eslint.org/`) for JavaScript.

Another important set of tools is for code compression or minification. While spaces, indenting, and descriptive variable names are essential for code readability, they are of no use to browsers when running the code, and only serve to use up bandwidth.

Code compression tools (or minifiers) remove unnecessary white space from the code, replace longer variable names with short single-letter variables, and apply various language-specific optimizations to make the code files as small as possible.

The minifiers that I currently use are HTMLMinifier (`https://github.com/kangax/html-minifier`) for HTML, Clean-CSS (`https://github.com/jakubpawlowicz/clean-css`) for CSS, and UglifyJS (`https://github.com/mishoo/UglifyJS`) for JavaScript.

The most common way to use these tools is to integrate them into a build process so that the source files are combined and minified into a release folder when you are ready to release a new version of the game. The release folder, which contains only compressed versions of the code files, is then deployed onto a web server so they can be accessed and run by browsers.

Servers

There are two server tools that I find invaluable when developing HTML5- and Node.js-based games:

- *http-server*: http-server is a simple, zero-configuration command-line HTTP server. We looked at its use briefly when we started developing mobile game code back in Chapter 5. The program allows you to serve the contents of any folder just by navigating to the folder from the command line and calling it, which is extremely convenient for local development and testing. It allows you to specify options such as the port you want the server to run on and how long you want it to cache content. You can read more about http-server at `https://www.npmjs.com/package/http-server`.

- *Process Manager 2*: Process Manager 2 (PM2) is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime, and to facilitate common system admin tasks. This is very useful when you want to run one or more Node.js applications, such as the Last Colony multiplayer server. PM2 lets you easily keep track of all the server processes, view their logs and memory usage, and track whether an application crashed and needed to be restarted. It can also monitor your code files and automatically restart the application whenever the code changes. You can read more about PM2 at `https://www.npmjs.com/package/pm2`.

Build Automation

Even though all these other tools will make your development faster and easier, you can go still further by automating your build process. A good build automation tool will allow you to automate all the painful and time-consuming tasks without taking too much effort to configure and set up these tasks. While there are several popular build toolkits out there, my personal favorite is Gulp.js.

Gulp.js is a task runner, built on Node.js and npm, that allows you to define tasks in JavaScript as well as access most npm modules from within your build scripts. This reduces the typical learning curve associated with using the build tool, while still giving you the power and flexibility to do anything that you would do in a typical Node.js script.

419

Instructions for setting up Gulp.js as well as detailed documentation are available at the Gulp.js site (https://gulpjs.com/).

Gulp.js also has wrapper plug-ins for most of the tools discussed earlier, which allows you to easily invoke them from within your build scripts. For example, Clean-CSS has the `gulp-clean-css` plug-in (https://www.npmjs.com/package/gulp-clean-css) and UglifyJS has the `gulp-uglify` plug-in (https://www.npmjs.com/package/gulp-uglify). You will find examples of usage for the plug-in included with the documentation for most plug-ins.

A typical example of a build script that minifies all your HTML code is shown in Listing 14-2.

Listing 14-2. A Typical Gulp.js Build Script

```
var gulp = require("gulp");
var htmlmin = require("gulp-htmlmin");

gulp.task("minify", function() {
    return gulp.src("src/*.html")
        .pipe(htmlmin({ collapseWhitespace: true }))
        .pipe(gulp.dest("dist"));
});
```

This example script uses the `gulp-htmlmin` plug-in to take all the HTML files within the `src` folder and minify them before saving the compressed files in the `dist` folder. You can read more about the `gulp-htmlmin` plug-in for HTMLMinifier at https://github.com/jonschlinkert/gulp-htmlmin.

You should be able to find a preexisting gulp plug-in for most common build tasks, and easily write your own plug-in in JavaScript for any tasks that don't have one.

Once you set up your plug-ins and write your tasks, you can potentially have a script run all the tasks that you need with a single invocation of the `gulp` command. In time, as you get comfortable with using (and reusing) build scripts, you should start automating as much as possible, so you can focus on new development and game creation.

Summary

If you have been following along since the beginning of this book, you should now have the knowledge, resources, and confidence to build your own amazing games in HTML5, and I want to thank you for taking this journey with me.

My goal in writing this book was to demystify the process of building complex games in HTML5 and provide you with everything that you would need to build such games on your own. I sincerely hope that I was successful in this goal.

A lot of the changes and improvements in this second edition, such as more-detailed explanations, newer game features, and additional content on mobile development, were based on the questions and feedback from readers, and I would appreciate hearing your feedback as well. I would also love to hear about how you used this book as a starting point for your own game projects.

If you have any questions, comments, or feedback, you can reach with me via my website at www.adityaravishankar.com.

While this book should have given you a great start, in my experience most of the learning in game programming comes from setting out on your own journey—trying your own experiments, making your own mistakes, and constantly growing and improving with each experience. So keep going, keep creating, and keep learning.

I wish you all the best in your game programming journey.

Index

■ **T, U, V**

■ **W, X, Y, Z**