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 1 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 three 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. We will be using jQuery (`http://jquery.com/`) to help us with some of these manipulation tasks. 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/jquery.min.js" type="text/javascript" charset="utf-8"></script>
        <script src="js/game.js" type="text/javascript" charset="utf-8"></script>
        <link rel="stylesheet" href="styles.css" type="text/css" media="screen" charset="utf-8">
    </head>

    <body>
        <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">
                <img src="images/icons/prev.png">
                <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"><img src="images/icons/prev.png">Replay Current
Level</p>
                    <p id="playnextlevel"><img src="images/icons/next.png">Play Next Level </p>
                    <p id="showLevelScreen"><img src="images/icons/return.png">Return to Level
Screen</p>
                </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`.

In addition, we will also add CSS styles for these layers in an external file called `styles.css`. We will start by 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)

```
#gamecontainer {
    width:640px;
    height:480px;
    background: url(images/splashscreen.png);
    border: 1px solid black;
}

.gamelayer {
    width:640px;
    height:480px;
    position:absolute;
    display:none;
}

/* 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:

- Define our game container and all game layers with a size of 640px by 480px.
- Make 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 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.
- Add 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 in the Source Code/Download area of the Apress web site (`www.apress.com`). 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 surrounded by a black border, as shown in Figure 2-1.

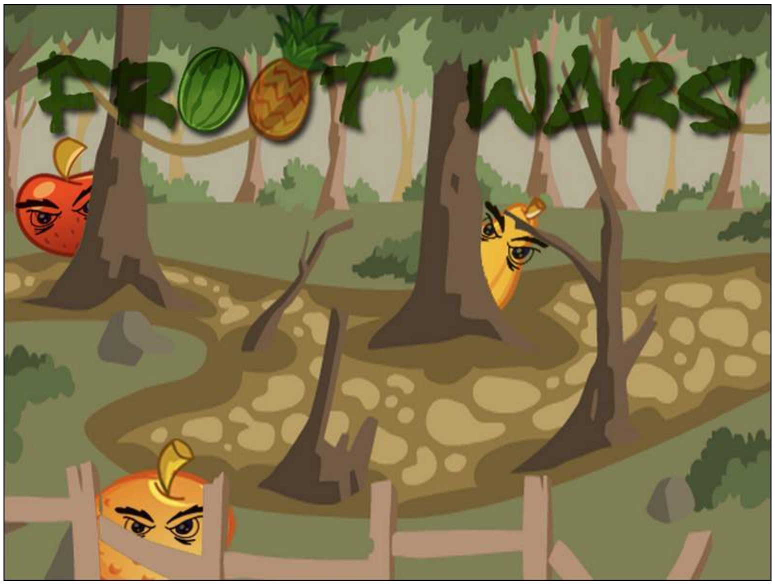

Figure 2-1. *The game splash screen*

We need to add some JavaScript code to start showing the main menu, the loading screen, and the game. To keep our code clean and easy to maintain, we will keep all our game-related JavaScript code in a separate 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(){

        // Hide all game layers and display the start screen
        $('.gamelayer').hide();
        $('#gamestartscreen').show();

        //Get handler for game canvas and context
        game.canvas = $('#gamecanvas')[0];
        game.context = game.canvas.getContext('2d');
    },
}
```

The code in Listing 2-3 defines a JavaScript object called game with an `init()` function. For now, this `init()` function just hides all game layers and shows the game start screen using the jQuery `hide()` and `show()` functions. It also saves pointers to the game `canvas` and `context` so we can refer to them more easily using `game.context` and `game.canvas`.

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 top of `game.js` (shown in Listing 2-4).

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

```
$(window).load(function() {
    game.init();
});
```

When we run our HTML code, 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.

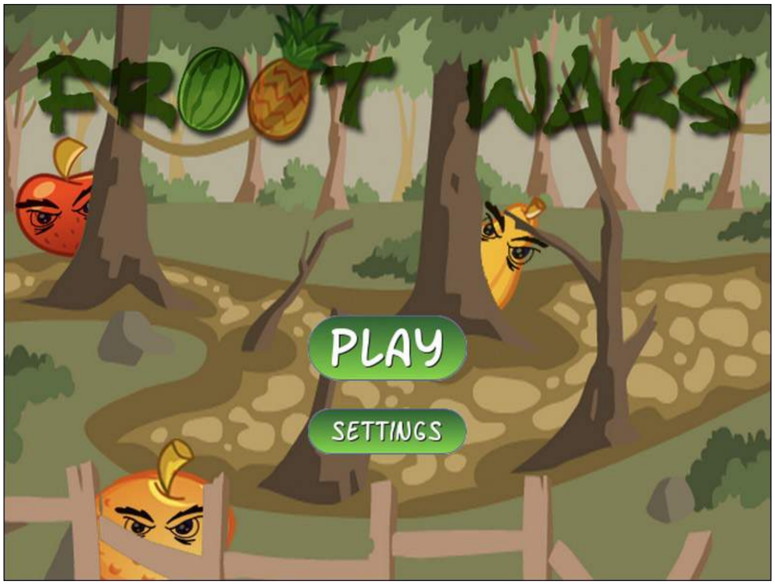

Figure 2-2. *The game start screen and menu options*

Level Selection

So far we have waited for the game HTML file to load completely and then show a main menu with two options. 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 html = "";
        for (var i = 0; i < levels.data.length; i++) {
            var level = levels.data[i];
            html += '<input type = "button" value = "' + (i + 1) + '">';
        };
        $('#levelselectscreen').html(html);

        // Set the button click event handlers to load level
        $('#levelselectscreen input').click(function(){
            levels.load(this.value-1);
            $('#levelselectscreen').hide();
        });
    },

    // 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 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. The level button click event handlers are set to call the `load()` method for each level and then hide the level selection screen.

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

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

```
init: function(){
    // Initialize objects
    levels.init();

    // Hide all game layers and display the start screen
    $('.gamelayer').hide();
    $('#gamestartscreen').show();
    //Get handler for game canvas and context
    game.canvas = $('#gamecanvas')[0];
    game.context = game.canvas.getContext('2d');
},
```

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;
}
```

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(){
    $('.gamelayer').hide();
    $('#levelselectscreen').show('slow');
},
```

This method first hides all the other game layers and then shows the levelselectscreen layer, using a slow animation.

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:

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

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

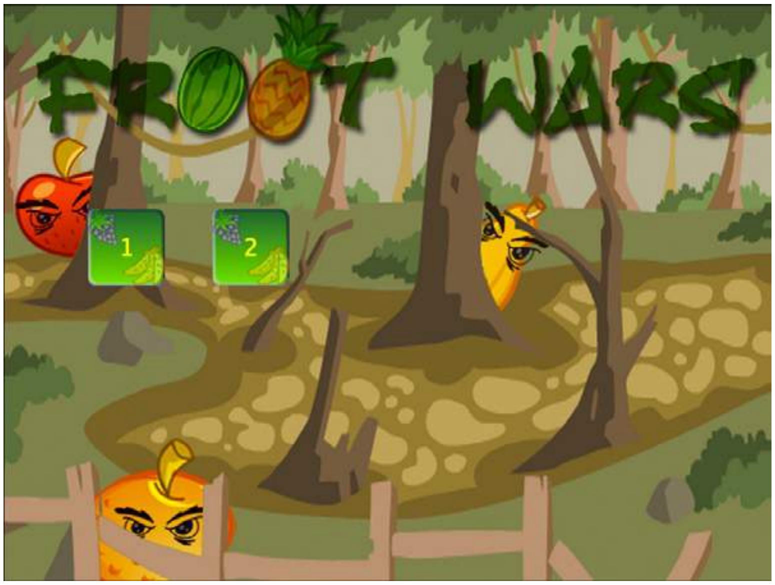

Figure 2-3. *The level selection screen*

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 call the `levels.load()` button 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

```
/* Loading Screen */
#loadingscreen {
    background:rgba(100,100,100,0.3);
}

#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.

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

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

    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.totalCount++;
        this.loaded = false;
        $('#loadingscreen').show();
        var image = new Image();
        image.src = url;
        image.onload = loader.itemLoaded;
        return image;
    },
    soundFileExtn:".ogg",
    loadSound:function(url){
        this.totalCount++;
        this.loaded = false;
        $('#loadingscreen').show();
        var audio = new Audio();
        audio.src = url + loader.soundFileExtn;
        audio.addEventListener("canplaythrough", loader.itemLoaded, false);
        return audio;
    },
    itemLoaded:function(){
        loader.loadedCount++;
        $('#loadingmessage').html('Loaded '+loader.loadedCount+' of '+loader.totalCount);
        if (loader.loadedCount === loader.totalCount){
            // Loader has loaded completely..
            loader.loaded=true;
            // Hide the loading screen
            $('#loadingscreen').hide();
            //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.
- 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 call 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(){
    // Initialize objects
    levels.init();
    loader.init();

    // Hide all game layers and display the start screen
    $('.gamelayer').hide();
    $('#gamestartscreen').show();

    //Get handler for game canvas and context
    game.canvas=$('#gamecanvas')[0];
    game.context=game.canvas.getContext('2d');
},
```

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 screen will display the loading screen shown in Figure 2-4 until all the images and sounds are loaded.

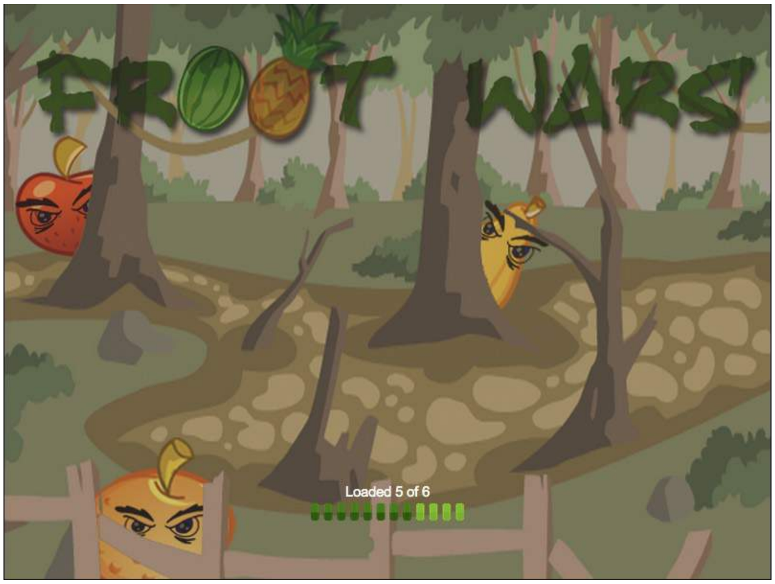

Figure 2-4. *The loading screen*

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

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,hero:[]};
      game.score=0;
      $('#score').html('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
      if(loader.loaded){
          game.start()
      } else {
          loader.onload = game.start;
      }
  }
```

The `load()` function creates a `currentLevel` object to store the loaded level data. So far we have only loaded three images. We will eventually 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 either calling it immediately or 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`. Before we can use `requestAnimationFrame`, we need to place the `requestAnimation` polyfill function from Chapter 1 at the top of `game.js` so that we can use it from our game code, as shown in Listing 2-13.

Listing 2-13. The requestAnimationFrame Polyfill

```
// Set up requestAnimationFrame and cancelAnimationFrame for use in the game code
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] ||
window[vendors[x]+'CancelRequestAnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime + timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };
    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
}());
```

We next 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 2-14.

Listing 2-14. The start() and animate() Functions Inside the game Object

```
// Game mode
mode:"intro",
// X & Y Coordinates of the slingshot
slingshotX:140,
slingshotY:280,
start:function(){
    $('.gamelayer').hide();
    // Display the game canvas and score
    $('#gamecanvas').show();
    $('#scorescreen').show();

    game.mode = "intro";
    game.offsetLeft = 0;
    game.ended = false;
    game.animationFrame = window.requestAnimationFrame(game.animate,game.canvas);
},
handlePanning:function(){
    game.offsetLeft++; // Temporary placeholder – keep panning to the right
},
animate:function(){
    // Animate the background
   game.handlePanning();

   // Animate the characters

    //  Draw the background with parallax scrolling

game.context.drawImage(game.currentLevel.backgroundImage,game.offsetLeft/4,0,640,480,0,0,640,480);

game.context.drawImage(game.currentLevel.foregroundImage,game.offsetLeft,0,640,480,0,0,640,480);

    // Draw the slingshot
    game.context.drawImage(game.slingshotImage,game.slingshotX-game.offsetLeft,game.slingshotY);

    game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,game.
slingshotY);

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

Again, the preceding code consists 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 that is a narrow bar on the top of the screen that contains.
- 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 temporary placeholders for animating the background and characters. We will be implementing these later. We then draw the background and foreground image using the `offsetLeft` variable to offset the x axis of the images. 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.

One thing to note is that the background image and foreground image are moved at different speeds relative to the scroll left: 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.

Finally, we draw the slingshot in the foreground.

■ **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 2-15.

Listing 2-15. CSS for Score Screen Panel

```
/* Score Screen */
#scorescreen  {
    height:60px;
    font: 32px Comic Sans MS;
    text-shadow: 0 0 2px #000;
    color:white;
}

#scorescreen img{
    opacity:0.6;
    top:10px;
    position:relative;
    padding-left:10px;
    cursor:pointer;
}
```

```
#scorescreen #score {
    position:absolute;
    top:5px;
    right:20px;
}
```

The scorescreen layer, unlike the other layers, is just a narrow band at the top of our game. We also add some transparency to ensure that the images (for stopping 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 Score bar in the top-right corner, as shown in Figure 2-5.

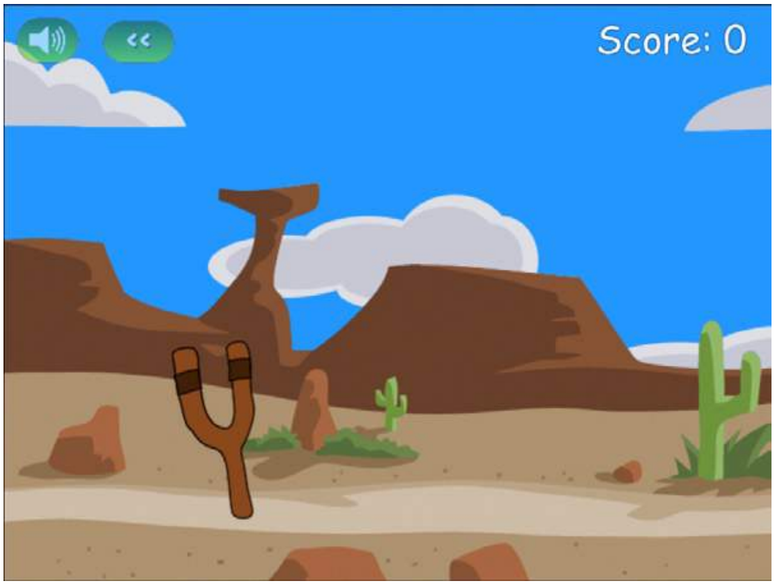

Figure 2-5. *A basic level with the score*

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. We could, potentially, add more layers and move them at different speeds to build more of an effect, but the two images illustrate this 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 use jQuery to create a separate `mouse` object inside `game.js` to handle all the mouse events, as shown in Listing 2-16.

Listing 2-16. Handling Mouse Events

```
var mouse={
    x:0,
    y:0,
    down:false,
    init:function(){
        $('#gamecanvas').mousemove(mouse.mousemovehandler);
        $('#gamecanvas').mousedown(mouse.mousedownhandler);
        $('#gamecanvas').mouseup(mouse.mouseuphandler);
        $('#gamecanvas').mouseout(mouse.mouseuphandler);
    },
    mousemovehandler:function(ev){
        var offset=$('#gamecanvas').offset();

        mouse.x=ev.pageX - offset.left;
        mouse.y=ev.pageY - offset.top;

        if (mouse.down) {
            mouse.dragging=true;
        }
    },
    mousedownhandler:function(ev){
        mouse.down=true;
        mouse.downX=mouse.x;
        mouse.downY=mouse.y;
        ev.originalEvent.preventDefault();
    },
    mouseuphandler:function(ev){
        mouse.down=false;
        mouse.dragging=false;
    }
}
```

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 jQuery's `offset()` method and the event object's `pageX` and `pageY` 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 `mouse.down` variable to `true` and stores the location where the mouse button was pressed. It additionally contains an extra line to prevent the default browser behavior of the click button.
- `mouseuphandler()`: Sets the `down` and `dragging` variables to `false`. If the mouse leaves the canvas area, we call this same method.

Now that we have these methods in place, we can always add more 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 2-17.

Listing 2-17. Initializing the Mouse from game.init()

```
init: function(){
    // Initialize objects
    levels.init();
    loader.init();
    mouse.init();

    // Hide all game layers and display the start screen
    $('.gamelayer').hide();
    $('#gamestartscreen').show();

    //Get handler for game canvas and context
    game.canvas = $('#gamecanvas')[0];
    game.context = game.canvas.getContext('2d');
},
```

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 mentioned 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, as shown in Listing 2-18. All of this code goes inside the game object after the `start()` method.

Listing 2-18. Implementing Panning Using the Game Modes

```
// Maximum panning speed per frame in pixels
maxSpeed:3,
// Minimum and Maximum panning offset
minOffset:0,
maxOffset:300,
// Current panning offset
offsetLeft:0,
// The game score
score:0,

//Pan the screen to center on newCenter
panTo:function(newCenter){
    if (Math.abs(newCenter-game.offsetLeft-game.canvas.width/4)>0
        && game.offsetLeft<= game.maxOffset && game.offsetLeft>= game.minOffset){

        var deltaX=Math.round((newCenter-game.offsetLeft-game.canvas.width/4)/2);
        if (deltaX && Math.abs(deltaX)>game.maxSpeed){
            deltaX=game.maxSpeed*Math.abs(deltaX)/(deltaX);
        }
        game.offsetLeft += deltaX;
    } else {

        return true;
    }
    if (game.offsetLeft<game.minOffset){
        game.offsetLeft=game.minOffset;
        return true;
    } else if (game.offsetLeft>game.maxOffset){
        game.offsetLeft=game.maxOffset;
        return true;
    }
    return false;
},
handlePanning: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"){
        // TODO:
        // 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"){
        game.panTo(game.slingshotX);
    }

    if (game.mode == "fired"){
        // TODO:
        // Pan to wherever the hero currently is
    }
},
```

We first create a method called `panTo()` that slowly pans the screen to a given x coordinate and returns `true` if the coordinate is near the center of the screen or if the screen has panned to the extreme left or right. It also caps the panning speed using `maxSpeed` so that the panning never becomes too fast. We have also improved the `handlePanning()` method so it implements a few of the game states we described earlier. We haven't implemented the `load-current-hero`, `firing`, and `fired` states yet.

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. We can also drag the mouse to the left or right of the screen to look around the level.

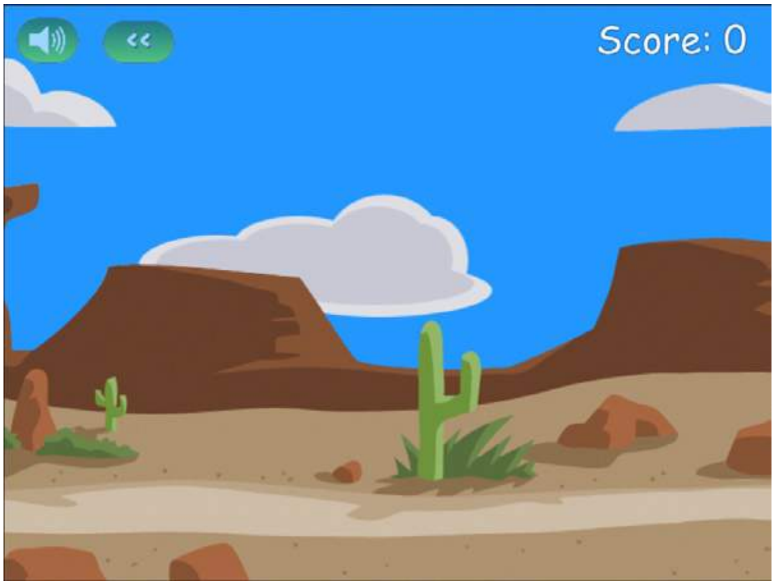

***Figure 2-6.** The final result: panning around the level*

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 user 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 be learning the basics of the Box2D physics engine and using it to model the physics for our game. We will learn how to animate our characters using data from the physics engine. We will then integrate this engine with our existing framework so that the game entities move realistically within our game world, after which we can actually start playing the game.