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.

The HTML5 specification is currently a work in progress, and browsers are still implementing some of its newer features. However, the elements that we need for building some very amazing games are already supported by most modern browsers (Google Chrome, Mozilla Firefox, Internet Explorer 9+, Safari, and Opera).

All you need to get started on developing your games in HTML5 are a good text editor to write your code (I use TextMate for the Mac—http://macromates.com/) and a modern, HTML5-compatible browser (I use Google Chrome—http://www.google.com/chrome).

The structure of an HTML5 file is very similar to that of files in previous versions of HTML except that it has a much simpler DOCTYPE tag at the beginning of the file. 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, this file should pop up the message "Hello World!"

Listing 1-1. Basic HTML5 File Skeleton

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv = "Content-type" content = "text/html; charset = utf-8">
        <title>Sample HTML5 File</title>
        <script type = "text/javascript" charset = "utf-8">
            // 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 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 canvas and image. Trying to access these elements before the browser has finished loading them will cause JavaScript errors.

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, or other visual images on the fly." You can find the complete specification at www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html.

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

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

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 its primary rendering context. We can access this context with the `getContext()` method in 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.

Listing 1-3. Accessing the Canvas Context

```
<script type = "text/javascript" charset = "utf-8">
    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. Browsers also implement other contexts with their own proprietary names, such as `experimental-webgl` for 3D graphics.

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

The canvas uses a coordinate system with the origin (0,0) at the top-left corner, x increasing toward the right, and y increasing downward, as illustrated in Figure 1-1.

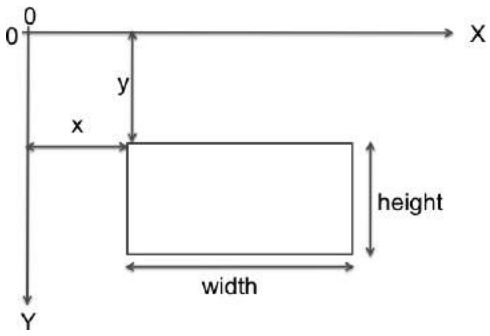

Figure 1-1. *Coordinate system for canvas*

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

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

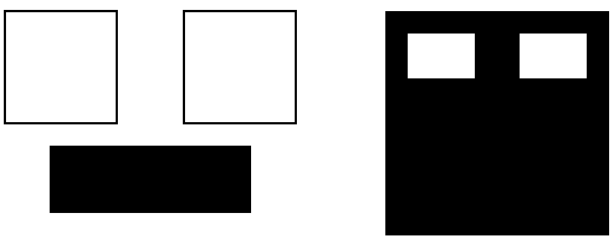

Figure 1-2. *Drawing rectangles inside the canvas*

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
// 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

// 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();

// 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

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

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

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

The code in Listing 1-4 will create the triangles, arcs and shapes shown in Figure 1-3.

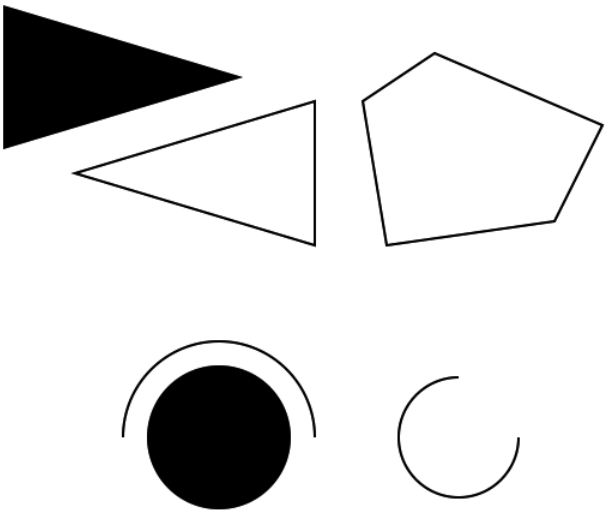

Figure 1-3. *Drawing complex shapes inside the canvas*

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. The text output, however, can be modified by setting the context font property as well as the stroke and fill styles, as shown in Listing 1-6. When setting the font property, you can use any valid CSS font property.

Listing 1-6. Drawing Text Inside the Canvas

```
// Drawing text
context.fillText('This is some text...',330,40);
```

```
// Modifying the font
context.font = '10 pt Arial';
context.fillText('This is in 10 pt Arial...',330,60);

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

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

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

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`).

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

Listing 1-7. Drawing with Colors and Transparency

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

```
// Set fill color to green with an alpha of 0.5
context.fillStyle = "rgba(0,255,0,0.6)";
// Draw a semi transparent green filled rectangle
context.fillRect (450,180,100,50);
```

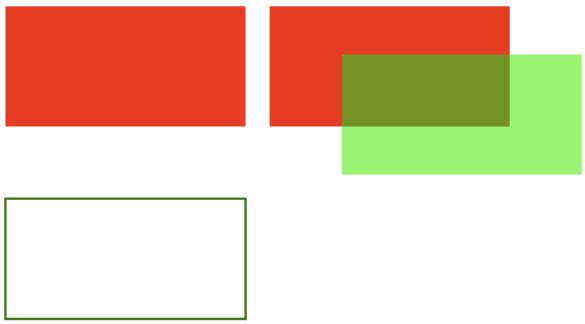

Figure 1-5. *Drawing with colors and transparency*

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 (sourceX, sourceY, 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 an image into the browser. For now, we will just add an `<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

```
// Get a handle to the image object
var image = document.getElementById('spaceship');

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

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

// Drawing 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.

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

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

```
//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);

//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.

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 because of licensing issues, but it works with OGG files. Safari, on the other hand, supports MP3 but does not support OGG. Table 1-1 shows the formats supported by the most popular browsers.

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

Browser	MP3	WAV	OGG
Internet Explorer 9+	Yes	No	No
Firefox 3.6+	No	Yes	Yes
Chrome 3+	Yes	No	Yes
Safari 4+	Yes	Yes	No
Opera 9.5+	No	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";

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

Again, as with the `<audio>` HTML tag, we need a way to detect which format the browser supports and 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 = "" ! = myAudio.canPlayType('audio/mpeg');
          oggSupport = "" ! = myAudio.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 = "bounce"+soundFileExtn;
        sound.play();
    }
</script>
```

The `Audio` object triggers an event called `canplaythrough` when the file is ready to be played. We can use this event to keep track of when the sound file has been loaded. Listing 1-14 shows an example.

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

```
<script>
    if(soundFileExtn) {
        var sound = new Audio();
        sound .addEventListener('canplaythrough', function(){
```

```
            alert('loaded');
            sound.play();
        });
        // Load sound file with the detected extension
        sound.src = "bounce"+soundFileExtn;
    }
</script>
```

We can use this to design an audio preloader that will load all the game resources before starting the game. We will look at this idea in more detail in the next few chapters.

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 it's 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 programmed to wait for all the images to load before they start. A common thing for programmers to do is to 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;
            }
        }
        return image;
    }
}
```

This image loader can be invoked to load a large number of images (say in a loop). Checking to see if all the images are loaded can be done using `imageLoader.loaded`, and a percentage/progress bar can be drawn using `loadedImages/totalImages`.

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.

On 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. People ended up waiting 5 to 10 minutes (sometimes longer) for the game to load before they could actually start playing. This is where sprite sheets come in.

Sprite sheets store all the sprites (images) for an object 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.

Looking at 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), 0, this.imageWidth,
this.imageHeight, x, y, this.imageWidth, this.imageHeight);
```

The following are some of the advantages of using a sprite sheet:

- *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 and the combined file size is significantly smaller than the sum of 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 a long time for the game to load.

Animation: Timer and Game Loops

Animating is just a matter of drawing an object, erasing it, and drawing it again at a new position. The most common way to handle this is by keeping a drawing function that gets called several times a second. In some games, there is also a separate control/animation function that updates movement of the entities within the game and is called less often than the drawing routine. Listing 1-21 shows a typical example.

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
}
```

Now 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.

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 once 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
}
```

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);
```

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

Different browser vendors have their own proprietary names for the methods in the API (such as Microsoft's `msrequestAnimationFrame` and Mozilla's `mozRequestAnimationFrame`). However, there is a simple piece of code (see Listing 1-24) that acts as a cross-browser polyfill providing you with the two methods that you use: `requestAnimationFrame()` and `cancelAnimationFrame()`.

Listing 1-24. A Simple requestAnimationFrame Polyfill

```
(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);
        };
}());
```

■ **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 by calculating the time since the previous drawing cycle and using this calculation to interpolate the location of the object being animated.

Once this polyfill is in place, the `requestAnimationFrame()` method can be called from within the drawingLoop() method similar to `setTimeout()` (see Listing 1-25).

Listing 1-25. 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 them
}
```

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 is meant to be a quick refresher on HTML5. We will be going into these topics in more detail, with complete implementations, as we build our games in the coming chapters.

If you have 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.