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

```
<!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!");
            }
```


A. R. Shankar, *Pro HTML5 Games*, DOI 10.1007/978-1-4842-2910-1_1

```
        </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.

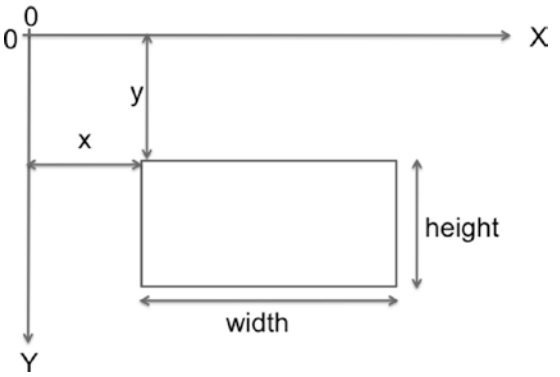

Figure 1-1. *Coordinate system for canvas*

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();
```

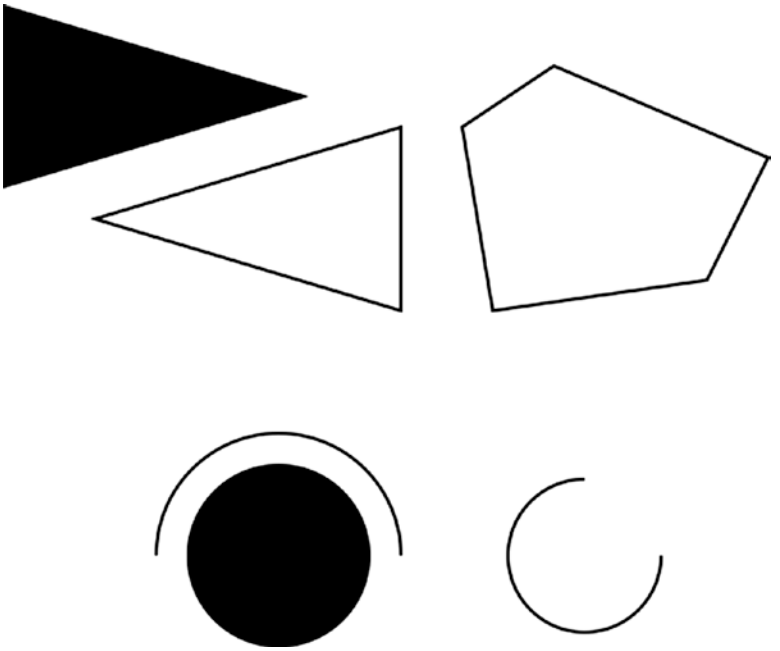

Figure 1-3. *Drawing complex shapes inside the canvas*

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.

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);
```