Native Interactivity and Animation for the Web

2nd Edition

HTML5 Canvas

O'REILLY®

Steve Fulton & Jeff Fulton

SECOND EDITION

HTML5 Canvas

Steve Fulton and Jeff Fulton

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

HTML5 Canvas, Second Edition

by Steve Fulton and Jeff Fulton

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Kara Ebrahim
Copyeditor: nSight, Inc.
Proofreader: nSight, Inc.
Indexer: Lucie Haskins
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

April 2013: Second Edition

Revision History for the Second Edition:

2013-04-10: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449334987* for release details.

ISBN: 978-1-449-33498-7

[LSI]

Table of Contents

CHAPTER 1
Introduction to HTML5 Canvas

HTML5 is the current iteration of HTML, the *HyperText Markup Language*. HTML was first standardized in 1993, and it was the fuel that ignited the World Wide Web. HTML is a way to define the contents of a web page using tags that appear within pointy brackets (< >).

HTML5 Canvas is an *immediate mode* bitmapped area of the screen that can be manipulated with JavaScript. Immediate mode refers to the way the canvas renders pixels on the screen. HTML5 Canvas completely redraws the bitmapped screen on every frame by using Canvas API calls from JavaScript. As a programmer, your job is to set up the screen display before each frame is rendered so that the correct pixels will be shown.

This makes HTML5 Canvas very different from Flash, Silverlight, or SVG, which operate in *retained mode*. In this mode, a display list of objects is kept by the graphics renderer, and objects are displayed on the screen according to attributes set in code (that is, the *x* position, *y* position, and alpha transparency of an object). This keeps the programmer away from low-level operations but gives her less control over the final rendering of the bitmapped screen.

The basic HTML5 Canvas API includes a 2D context that allows a programmer to draw various shapes, render text, and display images directly onto a defined area of the browser window. You can apply colors; rotations; gradient fills; alpha transparencies; pixel manipulations; and various types of lines, curves, boxes, and fills to augment the shapes, text, and images you place onto the canvas.

In itself, the HTML5 Canvas 2D context is a display API used to render graphics on a bitmapped area, but there is very little in that context to create applications using the technology. By adding cross-browser-compatible JavaScript functionality for keyboard and mouse inputs, timer intervals, events, objects, classes, sound, math functions, and so on, you can learn to take HTML5 Canvas and create stunning animations, applications, and games.

Here's where this book comes in. We are going to break down the Canvas API into digestible parts and then put it back together, demonstrating how to use it to create applications. Many of the techniques you will learn in this book have been tried and used successfully on other platforms, and now we are applying them to this exciting new technology.

Browser Support for HTML5 Canvas

With the exception of Internet Explorer 8, HTML5 Canvas is supported in some way by most modern web browsers, with specific feature support growing on an almost daily basis. The best support seems to be from Google Chrome, followed closely by Safari, Internet Explorer 10, Firefox, and Opera. We will utilize a JavaScript library named *modernizr.js* that will help us figure out which browsers support which Canvas features.

What Is HTML5?

Recently the definition of HTML5 has undergone a transition. When we wrote the first edition of this book in 2010, the W3C HTML5 specification was a distinct unit that covered a finite set of functionality. This included things like new HTML mark-up, `<video>`, `<audio>`, and `<canvas>` tags. However, in the past year, that definition has changed.

So, what *is* HTML5 now? The W3C HTML5 FAQ says this about HTML5:

> HTML5 is an open platform developed under royalty free licensing terms. People use the term HTML5 in two ways:
>
> - to refer to a set of technologies that together form the future Open Web Platform. These technologies include HTML5 specification, CSS3, SVG, MathML, Geolocation, XmlHttpRequest, Context 2D, Web Fonts (WOFF) and others. The boundary of this set of technologies is informal and changes over time.
> - to refer to the HTML5 specification, which is, of course, also part of the Open Web Platform.

What we have learned through conversations and project work in the past few months is that, to the common person who does not follow this closely (or more likely, the common customer who needs something done right away), *it's all HTML5*, and therefore when someone says "HTML5," they are actually referring to the "Open Web Platform."

The one thing we are certain about regarding this "Open Web Platform" is that the one technology that was definitely left off the invite list was Adobe Flash.

So what is HTML5? In a nutshell, it is "not Flash" (and other like technologies), and HTML5 Canvas is the technology that has the best capability of replacing Flash functionality on the web and mobile web. This book will teach you how to get started.

The Basic HTML5 Page

Before we get to Canvas, we need to talk a bit about the HTML5 standards that we will be using to create our web pages.

HTML is the standard language used to construct pages on the World Wide Web. We will not spend much time on HTML, but it does form the basis of `<canvas>`, so we cannot skip it entirely.

A basic HTML page is divided into sections, commonly `<head>` and `<body>`. The new HTML5 specification adds a few new sections, such as `<nav>`, `<article>`, `<header>`, and `<footer>`.

The `<head>` tag usually contains information that will be used by the HTML `<body>` tags to create the HTML page. It is a standard convention to put JavaScript functions in the `<head>`, as you will see later when we discuss the `<canvas>` tag. There might be reasons to put some JavaScript in the `<body>`, but we will make every attempt to keep things simple by having all JavaScript in the `<head>`.

Basic HTML for a page might look like Example 1-1.

Example 1-1. A basic HTML page

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>CH1EX1: Basic Hello World HTML Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

<!doctype html>

This tag informs the web browser to render the page in standards mode. According to the HTML5 spec from W3C, this is required for HTML5 documents. This tag simplified a long history of oddities when it came to rendering HTML in different browsers. This should always be the first line of HTML in a document.

<html lang="en">

This is the `<html>` tag with the language referenced: for example, `"en"` = English. Some of the more common language values are:

Chinese: `lang = "zh"`
French: `lang = "fr"`
German: `lang = "de"`
Italian: `lang = "it"`
Japanese: `lang = "ja"`
Korean: `lang = "ko"`
Polish: `lang = "pl"`
Russian: `lang = "ru"`
Spanish (Castilian): `lang = "es"`

<meta charset="UTF-8">

This tag tells the web browser which character-encoding method to use for the page. Unless you know what you're doing, there is no need to change it. This is a required element for HTML5 pages.

<title>...</title>

This is the title that will be displayed in the browser window for the HTML page. This is a very important tag, because it is one of the main pieces of information a search engine uses to catalog the content on the HTML page.

A Simple HTML5 Page

Now let's look at this page in a web browser. (This would be a great time to get your tools together to start developing code.) Open your chosen text editor, and get ready to use your preferred web browser: Safari, Firefox, Opera, Chrome, or IE.

1. In your text editor, type in the code from Example 1-1.
2. Save the code as *CH1EX1.html* in a directory of your choosing.
3. Under the File menu in Chrome, Safari, or Firefox, you should find the option Open File. Click that selection. You should then see a box to open a file. (On Windows using Chrome, you might need to press Ctrl+O to open a file.)
4. Locate the *CH1EX1.html* that you just created.
5. Click Open.

You should see something similar to Figure 1-1.

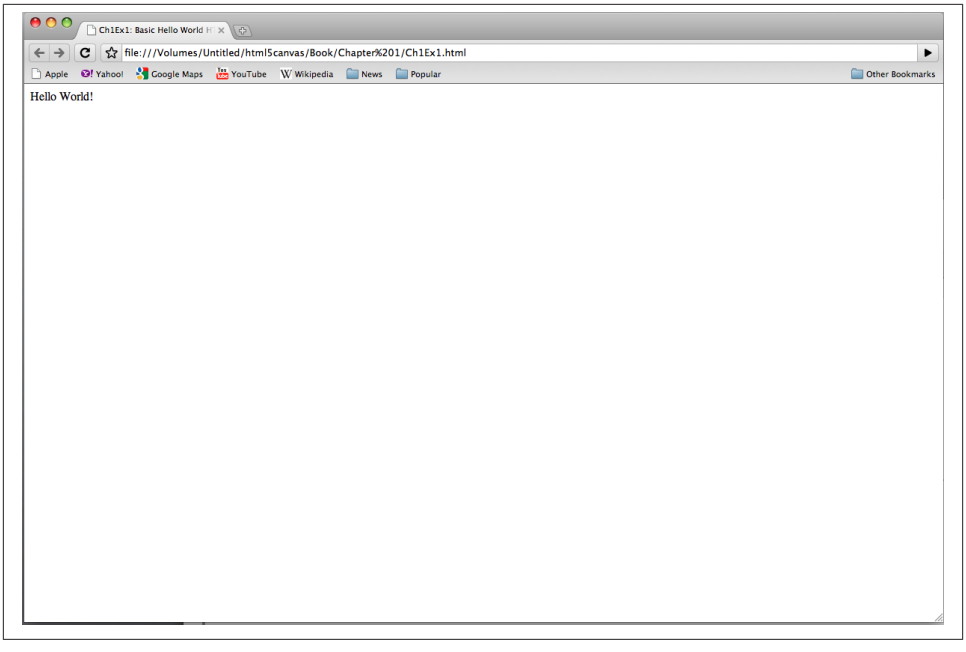


Figure 1-1. HTML Hello World!

Basic HTML We Will Use in This Book

Many HTML tags can be used to create an HTML page. In past versions of HTML, tags that specifically instructed the web browser on how to render the HTML page (for example, `<font>` and `<center>`) were very popular. However, as browser standards have become more restrictive in the past decade, those types of tags have been pushed aside, and the use of CSS (Cascading Style Sheets) has been adopted as the primary way to style HTML content. Because this book is not about creating HTML pages (that is, pages that don't have Canvas in them), we are not going to discuss the inner workings of CSS.

We will focus on only two of the most basic HTML tags: `<div>` and `<canvas>`.

This is the main HTML tag that we will use in this book. We will use it to position `<canvas>` on the HTML page.

Example 1-2 uses a `<div>` tag to position the words "Hello World!" on the screen, as shown in Figure 1-2.

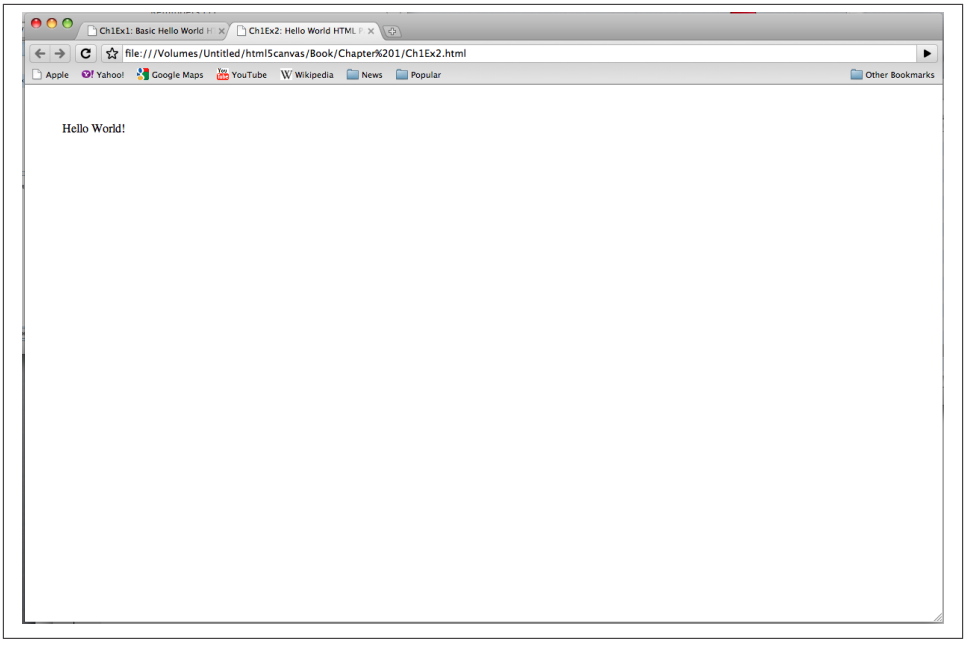
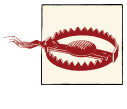


*Figure 1-2. HTML5 Hello World! with a *

Example 1-2. HTML5 Hello World!

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>CH1EX2: Hello World HTML Page With A DIV </title>
</head>
<body>
<div style="position: absolute; top: 50px; left: 50px;">
Hello World!
</div>
</body>
</html>
```

The `style="position: absolute; top: 50px; left: 50px;"` code is an example of inline CSS in an HTML page. It tells the browser to render the content at the absolute position of `50` pixels from the top of the page and `50` pixels from the left of the page.



This `<div>` might position the Canvas in the web browser, but it will not help us when we try to capture mouse clicks on the Canvas. In Chapter 5, we will discuss a way to both position the Canvas and capture mouse clicks in the correct locations.

<canvas>

Our work with <canvas> will benefit from using the absolute positioning method with . We will place our <canvas> inside the tag, and it will help us retrieve information, such as the relative position of the mouse pointer when it appears over a canvas.

The Document Object Model (DOM) and Canvas

The *Document Object Model* represents all the objects on an HTML page. It is language-neutral and platform-neutral, allowing the content and style of the page to be updated after it is rendered in the web browser. The DOM is accessible through JavaScript and has been a staple of JavaScript, DHTML, and CSS development since the late 1990s.

The canvas element itself is accessible through the DOM in a web browser via the Canvas 2D context, but the individual graphical elements created on Canvas are not accessible to the DOM. As we stated earlier, this is because Canvas works in immediate mode and does not have its own objects, only instructions on what to draw on any single frame.

Our first example will use the DOM to locate the <canvas> tag on the HTML5 page so that we can manipulate it with JavaScript. There are two specific DOM objects we will need to understand when we start using <canvas>: window and document.

The window object is the top level of the DOM. We will need to test this object to make sure all the assets and code have loaded before we can start our Canvas applications.

The document object contains all the HTML tags that are on the HTML page. We will need to look at this object to find the instance of <canvas> that manipulates with JavaScript.

JavaScript and Canvas

JavaScript, the programming language we will use to create Canvas applications, can be run inside nearly any web browser in existence. If you need a refresher on the topic, read Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly), which is a very popular and well-written reference on the subject.

Where Does JavaScript Go and Why?

Because we will create the programming logic for the Canvas in JavaScript, a question arises: where does that JavaScript go in the pages we have already created?

It's a good idea to place your JavaScript in the <head> of your HTML page because it makes it easy to find. However, placing JavaScript there means that the entire HTML

page needs to load before your JavaScript can work with the HTML. This also means that the JavaScript code will start to execute before the entire page loads. As a result, you will need to test to see whether the HTML page has loaded before you run your JavaScript program.

There has been a recent move to put JavaScript right before the `</body>` at the end of an HTML document to make sure that the whole page loads before the JavaScript runs. However, because we are going to test to see whether the page has loaded in JavaScript before we run our `<canvas>` program, we will put our JavaScript in the traditional `<head>` location. If you are not comfortable with this, you can adapt the style of the code to your liking.

No matter where you put the code, you can place it inline in the HTML page or load an *external .js* file. The code for loading an external JavaScript file might look like this:

```
<script type="text/javascript" src="canvasapp.js"></script>
```

To make things simple, we will code our JavaScript inline in the HTML page. However, if you know what you are doing, saving an external file and loading it will work just as well.

In HTML5, you no longer have to specify the script type.

HTML5 Canvas "Hello World!"

As we just mentioned, one of the first things we need to do when putting Canvas on an HTML5 page is test to see whether the entire page has loaded and all HTML elements are present before we start performing any operations. This will become essential when we start working with images and sounds in Canvas.

To do this, you need to work with *events* in JavaScript. Events are dispatched by objects when a defined event occurs. Other objects listen for events so that they can do something based on the event. Some common events that an object in JavaScript might listen for are keystrokes, mouse movements, and when something has finished loading.

The first event we need to listen for is a `window` object's `load` event, which occurs when the HTML page has finished loading.

To add a *listener* for an event, use the `addEventListener()` method that belongs to objects that are part of the DOM. Because `window` represents the HTML page, it is the top level of the DOM.

The `addEventListener()` function accepts three arguments:

Event: `load`
: This is the named event for which we are adding a listener. Events for existing objects like `window` are already defined.

Event handler function: `eventWindowLoaded()`
: Call this function when the event occurs. In our code, we will then call the `canvasApp()` function, which will start our main application execution.

`useCapture`*:* `true` *or* `false`
: This sets the function to capture this type of event before it propagates lower in the DOM tree of objects. We will always set this to `false`.

The final code we will use to test to see whether the `window` has loaded is as follows:

```
window.addEventListener("load", eventWindowLoaded, false);
function eventWindowLoaded () {
   canvasApp();
}
```

Alternatively, you can set up an event listener for the `load` event in a number of other ways:

```
window.onload = function()
   {
      canvasApp();
   }
```

or:

```
window.onload = canvasApp;
```

We will use the first method throughout this book.

Encapsulating Your JavaScript Code for Canvas

Now that we have created a way to test to see whether the HTML page has loaded, we can start creating our JavaScript application. Because JavaScript runs in an HTML page, it could be running with other JavaScript applications and code simultaneously. Usually, this does not cause any problems. However, there is a chance that your code might have variables or functions that conflict with other JavaScript code on the HTML page.

Canvas applications are a bit different from other apps that run in the web browser. Because Canvas executes its display in a defined region of the screen, its functionality is most likely self-contained, so it should not interfere with the rest of the page, and vice versa. You might also want to put multiple Canvas apps on the same page, so there must be some kind of separation of JavaScript when defining the code.

To avoid this issue, you can encapsulate your variables and functions by placing them inside another function. Functions in JavaScript are objects themselves, and objects in

JavaScript can have both properties and methods. By placing a function inside another function, you are making the second function local in scope to the first function.

In our example, we are going to have the `canvasApp()` function that is called from the `window load` event contain our entire Canvas application. This "Hello World!" example will have one function named `drawScreen()`. As soon as `canvasApp()` is called, we will call `drawScreen()` immediately to draw our "Hello World!" text.

The `drawScreen()` function is now local to `canvasApp()`. Any variables or functions we create in `canvasApp()` will be local to `drawScreen()` but not to the rest of the HTML page or other JavaScript applications that might be running.

Here is the sample code for how we will encapsulate functions and code for our Canvas applications:

```
function canvasApp() {
   drawScreen();

   ...

   function drawScreen() {

        ...

   }

}
```

Adding Canvas to the HTML Page

In the `<body>` section of the HTML page, add a `<canvas>` tag using code such as the following:

```
<canvas id="canvasOne" width="500" height="300">
 Your browser does not support HTML5 Canvas.
</canvas>
```

Now, let's break this down to understand what we are doing. The `<canvas>` tag has three main *attributes*. In HTML, attributes are set within pointy brackets of an HTML tag. The three attributes we need to set are:

`id`
: The `id` is the name we will use to reference this `<canvas>` tag in our JavaScript code. `canvasOne` is the name we will use.

`width`
: The width, in pixels, of the canvas. The `width` will be `500` pixels.

`height`
: The height, in pixels, of the canvas. The `height` will be `300` pixels.

HTML5 elements, including `canvas`, have many more attributes: `tabindex`, `title`, `class`, `accesskey`, `dir`, `draggable`, `hidden`, and so on.

Between the opening `<canvas>` and closing `</canvas>` tags, you can put text that will be displayed if the browser executing the HTML page does not support Canvas. For our Canvas applications, we will use the text "Your browser does not support HTML5 Canvas." However, you can adjust this text to say anything.

Using the document Object to Reference the Canvas Element in JavaScript

We will now make use of the DOM to reference the `<canvas>` we defined in HTML. Recall that the `document` object represents every element of an HTML page after it has loaded.

We need a reference to the `Canvas` object so that we will know where to display the Canvas API calls we will make from JavaScript.

First, we will define a new variable named `theCanvas` that will hold the reference to the `Canvas` object.

Next, we retrieve a reference to `canvasOne` by calling the `getElementById()` function of `document`, and passing the name `canvasOne`, which we defined as the `id` of the `<canvas>` tag we created in the HTML page:

```
var theCanvas = document.getElementById("canvasOne");
```

Testing to See Whether the Browser Supports Canvas

Now that we have a reference to the `canvas` element on the HTML page, we need to test to see whether it contains a *context*. The Canvas context refers to the drawing surface defined by a web browser to support Canvas. Simply put, if the context does not exist, neither does the Canvas. There are several ways to test this. This first test looks to see whether the `getContext` method exists before we call it using Canvas, as we have already defined it in the HTML page:

```
if (!theCanvas || !theCanvas.getContext) {
   return;
}
```

Actually, this tests two things. First, it tests to see whether `theCanvas` does not contain `false` (the value returned by `document.getElementById()` if the named `id` does not exist). Then, it tests whether the `getContext()` function exists.

The `return` statement breaks out and stops execution if the test fails.

Another method—popularized by Mark Pilgrim on his HTML5 website—uses a function with a test of a dummy canvas created for the sole purpose of seeing whether browser support exists:

```
function canvasSupport () {
    return !!document.createElement('canvas').getContext;
}
function canvasApp() {
   if (!canvasSupport) {
      return;
  }

}
```

Our favorite method is to use the *modernizr.js* library. Modernizr—an easy-to-use, lightweight library for testing support for various web-based technologies—creates a set of static Booleans that you can test against to see whether Canvas is supported.

To include *modernizr.js* in your HTML page, download the code from *http://www.modernizr.com/* and then include the external *.js* file in your HTML page:

```
<script src="modernizr.js"></script>
```

To test for Canvas, change the `canvasSupport()` function to look like this:

```
function canvasSupport () {
   return Modernizr.canvas;
}
```

We are going to use the *modernizr.js* method because we think it offers the best approach for testing whether Canvas is supported in web browsers.

Retrieving the 2D Context

Finally, we need to get a reference to the 2D context so that we can manipulate it. HTML5 Canvas is designed to work with multiple contexts, including a proposed 3D context. However, for the purposes of this book, we need to get only the 2D context:

```
var context = theCanvas.getContext("2d");
```

The drawScreen() Function

It's time to create actual Canvas API code. Every operation we perform on Canvas will be through the `context` object, because it references the object on the HTML page.

We will delve into writing text, graphics, and images to HTML5 Canvas in later chapters, so for now, we will spend only a short time on the code of the `drawScreen()` function.

The "screen" here is really the defined drawing area of the canvas, not the whole browser window. We refer to it as such because within the context of the games and applications

you will write, it is effectively the "window" or "screen" into the canvas display that you will be manipulating.

The first thing we want to do is clear the drawing area. The following two lines of code draw a yellow box on the screen that is the same size as the canvas. `fillStyle()` sets the color, and `fillRect()` creates a rectangle and puts it on the screen:

```
context.fillStyle = "#ffffaa";
context.fillRect(0, 0, 500, 300);
```

Notice that we are calling functions of the `context`. There are no screen objects, color objects, or anything else. This is an example of the immediate mode we described earlier.

Again, we will discuss the text functions of Canvas in the next chapter, but here is a short preview of the code we will use to put the text "Hello World!" on the screen.

First, we set the color of the text in the same way that we set the color of the rectangle:

```
context.fillStyle  = "#000000";
```

Then we set the font size and weight:

```
context.font = "20px Sans-Serif";
```

Next, we set the vertical alignment of the font:

```
context.textBaseline = "top";
```

Finally, we print our text on the screen by calling the `fillText()` method of the `context` object. The three parameters of this method are text string, `x` position, and `y` position:

```
context.fillText  ("Hello World!", 195, 80);
```

Let's add some graphics to our "Hello World!" text. First, let's load in an image and display it. We will dive into images and image manipulation in Chapter 4, but for now, let's just get an image on the screen. To display an image on the canvas, you need to create an instance of the `Image()` object, and set the `Image.src` property to the name of the image to load.

You can also use another canvas or a video as the image to display. We will discuss these topics in Chapter 4 and Chapter 6.

Before you display it, you need to wait for the image to load. Create an anonymous callback function for the `Image load` event by setting the `onload` function of the `Image`

object. The anonymous callback function will be executed when the `onload` event occurs. When the image has loaded, you then call `context.drawImage()`, passing three parameters to put it on the canvas: `Image` object, `x` position, and `y` position:

```
var helloWorldImage = new Image();
helloWorldImage.onload = function () {
   context.drawImage(helloWorldImage, 160, 130);
}
helloWorldImage.src = "helloworld.gif";
```

Finally, let's draw a box around the text and the image. To draw a box with no fill, use the `context.strokeStyle` property to set a color for the stroke (the border of the box), and then call the `context.strokeRect()` method to draw the rectangle border. The four parameters for the `strokeRect()` method are the upper left `x` and `y` coordinates, the width, and the height:

```
context.strokeStyle = "#000000";
context.strokeRect(5,  5, 490, 290);
```

The full code for the HTML5 "Hello World!" application is shown in Example 1-3, and its results are illustrated in Figure 1-3.

Example 1-3. HTML5 Canvas Hello World!

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>CH1EX3: Your First Canvas Application </title>

<script src="modernizr.js"></script>
<script type="text/javascript">
window.addEventListener("load", eventWindowLoaded, false);

var Debugger = function () { };
Debugger.log = function (message) {
   try {
      console.log(message);
   } catch (exception) {
      return;
   }
}

function eventWindowLoaded () {
   canvasApp();
}

function canvasSupport () {
    return Modernizr.canvas;
}

function canvasApp () {
```

```
        if (!canvasSupport()) {
          return;
        }

      var theCanvas = document.getElementById("canvasOne");
      var context = theCanvas.getContext("2d");

      Debugger.log("Drawing Canvas");

        function drawScreen() {
         //background
         context.fillStyle = "#ffffaa";
         context.fillRect(0, 0, 500, 300);

         //text
         context.fillStyle  = "#000000";
         context.font = "20px Sans-Serif";
         context.textBaseline = "top";
         context.fillText  ("Hello World!", 195, 80 );

         //image
         var helloWorldImage = new Image();
         helloWorldImage.onload = function () {
            context.drawImage(helloWorldImage, 155, 110);
         }
         helloWorldImage.src = "helloworld.gif";

         //box
         context.strokeStyle = "#000000";
         context.strokeRect(5,  5, 490, 290);

      }

      drawScreen();

}

</script>

</head>
<body>
<div style="position: absolute; top: 50px; left: 50px;">
<canvas id="canvasOne" width="500" height="300">
 Your browser does not support HTML5 Canvas.
</canvas>
</div>
</body>
</html>
```

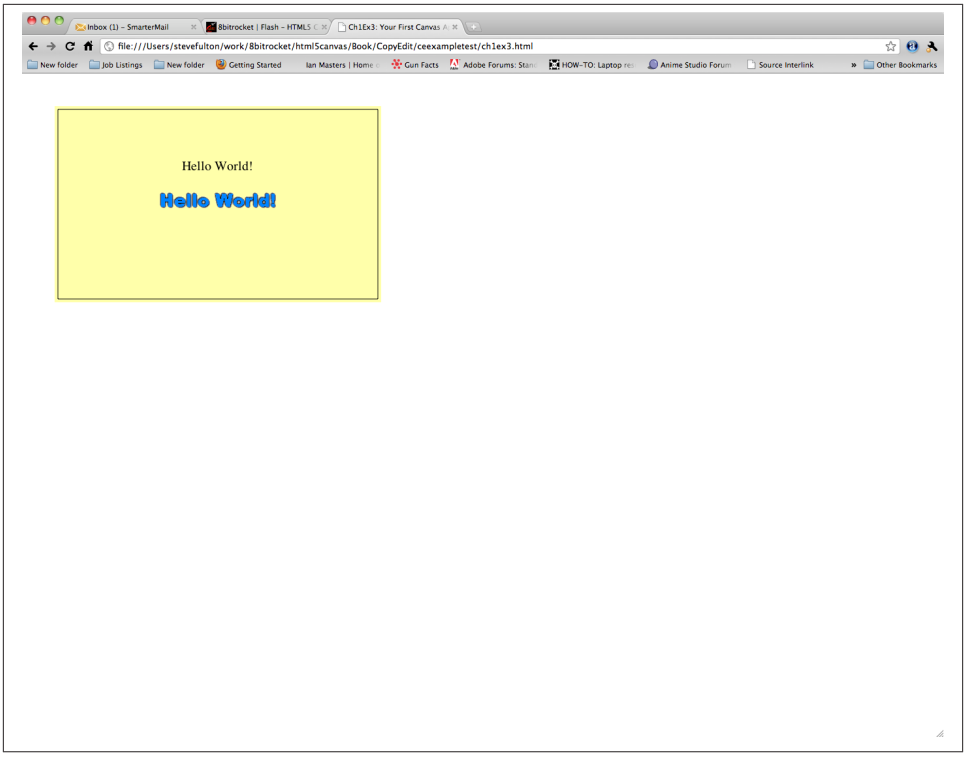


Figure 1-3. HTML5 Canvas Hello World!

Debugging with console.log

There is one more thing to discuss before we explore bigger and better things beyond "Hello World!" In this book, we have implemented a very simple debugging methodology using the `console.log` functionality of modern web browsers. This function lets you log text messages to the JavaScript console to help find problems (or opportunities!) with your code. Any browser that has a JavaScript console (Chrome, Opera, Safari, Firefox with Firebug installed) can make use of `console.log`. However, browsers without `console.log` support throw a nasty error.

To handle this error, we use a wrapper around `console.log` that makes the call only if the function is supported. The wrapper creates a class named `Debugger` and then creates a static function named `Debugger.log` that can be called from anywhere in your code, like this:

```
Debugger.log("Drawing Canvas");
```

Here is the code for the `console.log()` functionality:

```
var Debugger = function () { };
Debugger.log = function (message) {
   try {
      console.log(message);
   } catch (exception) {
      return;
   }
}
```

The 2D Context and the Current State

The HTML5 2D context (the `CanvasRenderingContext2D` object), retrieved by a call to the `getContext()` method of the `Canvas` object, is where all the action takes place. The `CanvasRenderingContext2D` contains all the methods and properties we need to draw onto the canvas. The `CanvasRenderingContext2D` (or context, as we will call it hereafter) uses a Cartesian coordinate system with 0,0 at the upper-left corner of the canvas, with coordinates increasing in value to the right and down.

However, all of these properties and methods are used in conjunction with *current state*, a concept that must be grasped before you can really understand how to work with HTML5 Canvas. The current state is actually a stack of drawing states that apply globally to the entire canvas. You will manipulate these states when drawing on the canvas. These states include:

Transformation matrix
: Methods for scale, rotate, transform, and translate.

Clipping region
: Created with the `clip()` method.

Properties of the context
: Properties include `strokeStyle`, `fillStyle`, `globalAlpha`, `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, `shadowColor`, `globalCompositeOperation`, `font`, `textAlign`, and `textBaseline`.

Don't worry; these should not look familiar to you just yet. We will discuss these properties in depth in the next three chapters.

Remember earlier in this chapter when we discussed immediate mode versus retained mode? The canvas is an immediate mode drawing surface, which means everything needs to be redrawn every time something changes. There are some advantages to this; for example, global properties make it very easy to apply effects to the entire screen. Once you get your head around it, the act of redrawing the screen every time there is an update makes the process of drawing to the canvas straightforward and simple.

On the other hand, retained mode is when a set of objects is stored by a drawing surface and manipulated with a display list. Flash and Silverlight work in this mode. Retained mode can be very useful for creating applications that rely on multiple objects with their

own independent states. Many of the same applications that could make full use of the canvas (games, activities, animations) are often easier to code with a retained mode drawing surface, especially for beginners.

Our challenge is to take advantage of the immediate mode drawing surface, while adding functionality to our code to help it act more like it works in retained mode. Throughout this book, we will discuss strategies that will help take this immediate mode operation and make it easier to manipulate through code.

The HTML5 Canvas Object

Recall that the `Canvas` object is created by placing the `<canvas>` tag in the `<body>` portion of an HTML page. You can also create an instance of a canvas in code like this:

```
var theCanvas = document.createElement("canvas");
```

The `Canvas` object has two associated properties and methods that can be accessed through JavaScript: `width` and `height`. These tell you the current width and height of the canvas rendered on the HTML page. It is important to note that they are *not* read-only; that is, they can be updated in code and changed on an HTML page. What does this mean? It means that you can dynamically resize the canvas on the HTML page without reloading.

You can also use CSS styles to change the scale of the canvas. Unlike resizing, scaling takes the current canvas bitmapped area and resamples it to fit into the size specified by the `width` and `height` attributes of the CSS style. For example, to scale the canvas to a 400×400 area, you might use this CSS style:

```
style="width: 400px; height:400px"
```

We include an example of scaling the Canvas with a transformation matrix in Chapter 3.

There are currently two public methods for the `Canvas` object. The first is `getContext()`, which we used earlier in this chapter. We will continue to use it throughout this book to retrieve a reference to the Canvas 2D context so we can draw onto the canvas.

The second method is `toDataURL()`. This method will return a string of data that represents the bitmapped image of the `Canvas` object as it is currently rendered. It's like a snapshot of the screen. By supplying different MIME types as a parameter, you can retrieve the data in different formats. The basic format is an *image/png*, but *image/jpeg* and other formats can be retrieved. We will use the `toDataURL()` method in the next application to export an image of the canvas into another browser window.

A third public method, `toBlob()`, has been defined and is being implemented across browsers. `toBlob([callback])` will return a file reference to an image instead of a base64 encoded string. It is currently not implemented in any browsers.

Another Example: Guess The Letter

Now we will take a quick look at a more involved example of a "Hello World!"–type application, the game "Guess The Letter." We've included this example to illustrate how much more Canvas programming is done in JavaScript than in the Canvas API.

In this game, shown in Figure 1-4, the player's job is to guess the letter of the alphabet that the computer has chosen randomly. The game keeps track of how many guesses the player has made, lists the letters he has already guessed, and tells the player whether he needs to guess higher (toward Z) or lower (toward A).

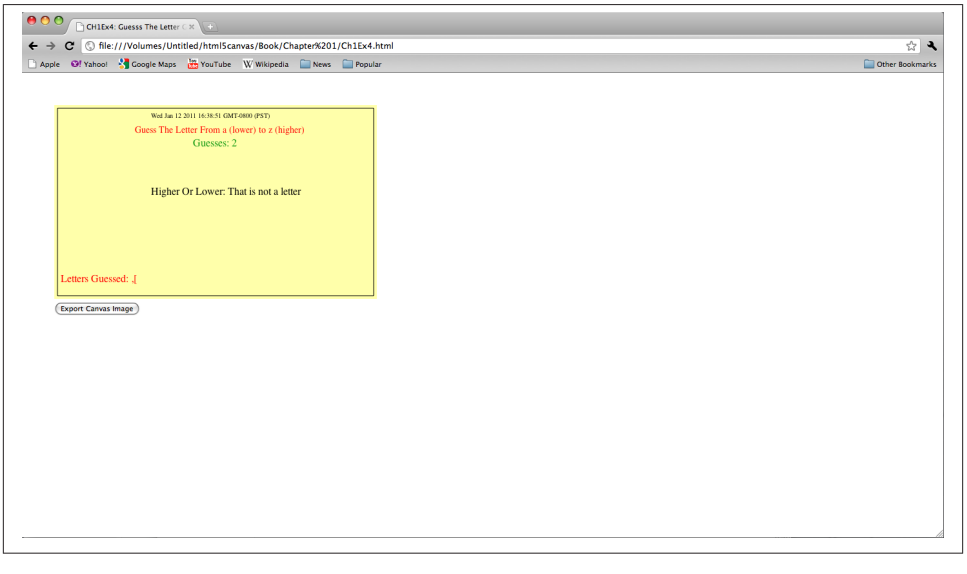


Figure 1-4. HTML5 Canvas "Guess The Letter" game

How the Game Works

This game is set up with the same basic structure as "Hello World!" `canvasApp()` is the main function, and all other functions are defined as local to `canvasApp()`. We use a `drawScreen()` function to render text on the canvas. However, there are some other functions included as well, which are described next.

The "Guess The Letter" Game Variables

Here is a rundown of the variables we will use in the game. They are all defined and initialized in `canvasApp()`, so they have scope to the encapsulated functions that we define locally:

`guesses`
: This variable holds the number of times the player has pressed a letter. The lower the number, the better he has done in the game.

`message`
: The content of this variable is displayed to give the user instructions on how to play.

`letters`
: This array holds one of each letter of the alphabet. We will use this array to both randomly choose a secret letter for the game and to figure out the relative position of the letter in the alphabet.

`today`
: This variable holds the current date. It is displayed on the screen but has no other purpose.

`letterToGuess`
: This variable holds the current game's secret letter that needs to be guessed.

`higherOrLower`
: This variable holds the text "Higher" or "Lower," depending on where the last guessed letter is in relation to the secret letter. If the secret letter is closer to "a," we give the "Lower" instruction. If the letter is closer to "z," we give the "Higher" instruction.

`lettersGuessed`
: This array holds the current set of letters that the player has guessed already. We will print this list on the screen to help the player remember what letters he has already chosen.

`gameOver`
: This variable is set to `false` until the player wins. We will use this to know when to put the "You Win" message on the screen and to keep the player from guessing after he has won.

Here is the code:

```
var guesses = 0;
var message = "Guess The Letter From a (lower) to z (higher)";
var letters = [
              "a","b","c","d","e","f","g","h","i","j","k","l","m","n","o",
              "p","q","r","s","t","u","v","w","x","y","z"
              ];
```

```
var today = new Date();
var letterToGuess = "";
var higherOrLower = "";
var lettersGuessed;
var gameOver = false;
```

The initGame() Function

The `initGame()` function sets up the game for the player. The two most important blocks of code are as follows. This code finds a random letter from the letters array and stores it in the `letterToGuess` variable:

```
var letterIndex = Math.floor(Math.random() * letters.length);
letterToGuess = letters[letterIndex];
```

This code adds an event listener to the `window` object of the DOM to listen for the keyboard `keydown` event. When a key is pressed, the `eventKeyPressed` event handler is called to test the letter pressed:

```
window.addEventListener("keydown",eventKeyPressed,true);
```

Here is the full code for the function:

```
function initGame() {
   var letterIndex = Math.floor(Math.random() * letters.length);
   letterToGuess = letters[letterIndex];
   guesses = 0;
   lettersGuessed = [];
   gameOver = false;
   window.addEventListener("keydown",eventKeyPressed,true);
   drawScreen();
}
```

The eventKeyPressed() Function

This function, called when the player presses a key, contains most of the action in this game. Every event handler function in JavaScript is passed an `event` object that has information about the event that has taken place. We use the `e` argument to hold that object.

The first test we make is to see whether the `gameOver` variable is `false`. If so, we continue to test the key that was pressed by the player; the next two lines of code are used for that purpose. The first line of code gets the key-press value from the event and converts it to an alphabetic letter that we can test with the letter stored in `letterToGuess`:

```
var letterPressed = String.fromCharCode(e.keyCode);
```

The next line of code converts the letter to lowercase so that we can test uppercase letters if the player unintentionally has Caps Lock on:

```
letterPressed = letterPressed.toLowerCase();
```

Next, we increase the `guesses` count to display and use the `Array.push()` method to add the letter to the `lettersGuessed` array:

```
guesses++;
lettersGuessed.push(letterPressed);
```

Now it is time to test the current game state to give feedback to the player. First, we test to see whether `letterPressed` is equal to `letterToGuess`. If so, the player has won the game:

```
if (letterPressed == letterToGuess) {
   gameOver = true;
```

If the player has not won, we need to get the index of `letterToGuess` and the index of `letterPressed` in the `letters` array. We are going to use these values to figure out whether we should display "Higher," "Lower," or "That is not a letter." To do this, we use the `indexOf()` array method to get the relative index of each letter. Because we alphabetized the letters in the array, it is very easy to test which message to display:

```
} else {
   letterIndex = letters.indexOf(letterToGuess);
   guessIndex = letters.indexOf(letterPressed);
```

Now we make the test. First, if `guessIndex` is less than zero, it means that the call to `indexOf()` returned `-1`, and the pressed key was not a letter. We then display an error message:

```
if (guessIndex < 0) {
   higherOrLower = "That is not a letter";
```

The rest of the tests are simple. If `guessIndex` is greater than `letterIndex`, we set the `higherOrLower` text to "Lower." Conversely, if `guessIndex` is less than `letterIndex`, we set the `higherOrLower` test to "Higher":

```
   } else if (guessIndex > letterIndex) {
      higherOrLower = "Lower";
   } else {
      higherOrLower = "Higher";
   }

}
```

Finally, we call `drawScreen()` to paint the screen:

```
drawScreen();
```

Here is the full code for the function:

```
function eventKeyPressed(e) {
      if (!gameOver) {
         var letterPressed = String.fromCharCode(e.keyCode);
         letterPressed = letterPressed.toLowerCase();
         guesses++;
```

```
            lettersGuessed.push(letterPressed);

            if (letterPressed == letterToGuess) {
               gameOver = true;
            } else {

               letterIndex = letters.indexOf(letterToGuess);
               guessIndex = letters.indexOf(letterPressed);
               Debugger.log(guessIndex);
               if (guessIndex < 0) {
                  higherOrLower = "That is not a letter";
               } else if (guessIndex > letterIndex) {
                  higherOrLower = "Lower";
               } else {
                  higherOrLower = "Higher";
               }

            }
            drawScreen();
           }
    }
```

The drawScreen() Function

Now we get to `drawScreen()`. The good news is that we have seen almost all of this before—there are only a few differences from "Hello World!" For example, we paint multiple variables on the screen using the Canvas Text API. We set `context.textBaseline = 'top';` only once for all the text we are going to display. Also, we change the color using `context.fillStyle`, and we change the font with `context.font`.

The most interesting thing we display here is the content of the `lettersGuessed` array. On the canvas, the array is printed as a set of comma-separated values, like this:

```
Letters Guessed: p,h,a,d
```

To print this value, all we do is use the `toString()` method of the `lettersGuessed` array, which prints out the values of an array as—you guessed it—comma-separated values:

```
context.fillText  ("Letters Guessed: " + lettersGuessed.toString(), 10, 260);
```

We also test the `gameOver` variable. If it is `true`, we put "You Got It!" on the screen in giant `40px` text so that the user knows he has won.

Here is the full code for the function:

```
function drawScreen() {
      //Background
      context.fillStyle = "#ffffaa";
      context.fillRect(0, 0, 500, 300);
      //Box
      context.strokeStyle = "#000000";
      context.strokeRect(5,  5, 490, 290);
```

```
        context.textBaseline = "top";
        //Date
        context.fillStyle = "#000000";
        context.font = "10px Sans-Serif";
        context.fillText  (today, 150 ,10);
        //Message
        context.fillStyle = "#FF0000";
        context.font = "14px Sans-Serif";
        context.fillText  (message, 125, 30);     //Guesses
        context.fillStyle = "#109910";
        context.font = "16px Sans-Serif";
        context.fillText  ('Guesses: ' + guesses, 215, 50);
        //Higher Or Lower
        context.fillStyle = "#000000";
        context.font = "16px Sans-Serif";
        context.fillText  ("Higher Or Lower: " + higherOrLower, 150,125);
        //Letters Guessed
        context.fillStyle = "#FF0000";
        context.font = "16px Sans-Serif";
        context.fillText  ("Letters Guessed: " + lettersGuessed.toString(),
                           10, 260);
        if (gameOver) {
           context.fillStyle = "#FF0000";
           context.font = "40px Sans-Serif";
           context.fillText  ("You Got It!", 150, 180);
        }
    }
```

Exporting Canvas to an Image

Earlier, we briefly discussed the `toDataUrL()` property of the `Canvas` object. We are going to use that property to let the user create an image of the game screen at any time. This acts almost like a screen-capture utility for games made on Canvas.

We need to create a button in the HTML page that the user can press to get the screen capture. We will add this button to `<form>` and give it the `id createImageData`:

```
<form>
<input type="button" id="createImageData" value="Export Canvas Image">
</form>
```

In the `init()` function, we retrieve a reference to that form element by using the `getElementById()` method of the `document` object. We then set an event handler for the button "click" event as the function `createImageDataPressed()`:

```
var formElement = document.getElementById("createImageData");
formElement.addEventListener('click', createImageDataPressed, false);
```

In `canvasApp()`, we define the `createImageDataPressed()` function as an event handler. This function calls `window.open()`, passing the return value of the `Canvas.toDa`

taURl() method as the source for the window. Since this data forms a valid *.png*, the image is displayed in the new window:

```
function createImageDataPressed(e) {

   window.open(theCanvas.toDataURL(),"canvasImage","left=0,top=0,width=" +
   theCanvas.width + ",height=" + theCanvas.height +",toolbar=0,resizable=0");
   }
```

We will discuss this process in depth in Chapter 3.

The Final Game Code

Check out the final game code for "Guess The Letter" in *CH1EX4.html* in the code distribution.

Hello World Animated Edition

The "Hello World" and "Guess The Letter" examples were fine, but they lacked an answer to the question "why?"—as in the question, "Why use the HTML5 Canvas at all?" Static images and text have been the realm of HTML since its inception, so why is the Canvas so different? To answer that question, we are going to create a second "Hello World" example that introduces the main feature that sets the Canvas from other methods of display in HTML: animation. In this example, we will simply fade the words "Hello World" in and out in the screen. While very simple, this is our first small step into the bigger world of the HTML5 Canvas. You can see an example of the final application in Figure 1-5.

Figure 1-5. HTML5 Canvas Animated Hello World

Some Necessary Properties

For this application we need a few properties to set everything up.

The `alpha` property is the value that we will apply to `context.globalAlpha` to set the transparency value for text that we will fade in and out. It is set to `0` to start, which means the text will start completely invisible. We will explain more about this in the next section.

The `fadeIn` property will tell our application if the text is currently fading in or fading out.

The `text` property holds the string we will display.

The `helloWorldImage` property will hold the background image we will display behind the fading text:

```
var alpha = 0;
var fadeIn = true;
var text = "Hello World";
var helloWorldImage = new Image();
helloWorldImage.src = "html5bg.jpg";
```

Animation Loop

To make anything move on the Canvas, you need an *animation loop*. An animation loop is a function called over and over on an interval. The function is used to clear the Canvas and redraw it with updated images, text, video, and drawing objects.

The easiest way to create an interval for animation is to use a simple `setTimeout()` loop. To do this, we create a function named `gameLoop()` (it can be called anything you like) that uses `window.setTimeout()` to call itself after a specified time period. For our application, that time period will be 20 milliseconds. The function then resets itself to call again in 20 milliseconds and then calls `drawScreen()`.

Using this method, `drawScreen()` is called every 20 milliseconds. We will place all of our drawing code in `drawScreen()`. This method does the same thing as using `setInterval()` but, because it clears itself and does not run forever, is much better for performance:

```
function gameLoop() {
    window.setTimeout(gameLoop, 20);
    drawScreen()
}

gameLoop();
```

requestAnimationFrame()

The *best* way to create an animation loop is by using the brand-new `window.requestAnimationFrame()` method. This new method uses a delta timer to tell your JavaScript program exactly when the browser is ready to render a new frame of animation. The code looks like this:

```
window.requestAnimFrame = (function(){
      return  window.requestAnimationFrame       ||
              window.webkitRequestAnimationFrame ||
              window.mozRequestAnimationFrame    ||
              window.oRequestAnimationFrame      ||
              window.msRequestAnimationFrame     ||
              function( callback ){
                window.setTimeout(callback, 1000 / 60);
              };
    })();

(function animloop(){
      requestAnimFrame(animloop);
      render();
    })();
(code originally developed by Paul Irish)
```

However, because this method is changing and has not been implemented across all browsers, we are going to use `window.setTimeout()` for applications in this book.

Alpha Transparency with the globalAlpha Property

We have chosen `context.globalAlpha` for this animation because it is very easy to explain and makes for an effective demonstration of animating on the Canvas. The `globalAlpha` property is the setting for transparency on the Canvas. The property accepts numbers from `0` through `1`, representing a percentage of opaqueness for what will be drawn *after* the property is set. For example:

```
context.globalAlpha = 0;
```

The preceding code would set everything drawn afterward to be rendered 0% opaque, or completely transparent.

```
context.globalAlpha = 1;
```

The preceding code would set everything drawn afterwards to be rendered 100% opaque, or 0% transparent.

```
context.globalAlpha = .5;
```

The preceding code would set everything drawn afterwards to be rendered 50% opaque, or 50% transparent.

By manipulating these values over time, we can make things drawn onto the Canvas appear to fade in or out.

`context.globalAlpha` affects *everything* drawn afterward, so if you don't want something drawn with the `globalAlpha` property of the last thing drawn, you need to reset the value before drawing onto the Canvas.

Clearing and Displaying the Background

In the `drawScreen()` function that is called every 20 milliseconds, we need to redraw the Canvas to update the animation.

Because our little application uses `globalAlpha` to change the transparency of things we are drawing, we first need to make sure to reset the property before we start our drawing operation. We do this by setting `context.globalAlpha` to `1` and then drawing the background (a black box). Next we set the `globalAlpha` property to `.25` and draw the `helloWorldImage` that we loaded. This will display the image at 25% opacity, with the black background showing through:

```
function drawScreen() {
    //background
    context.globalAlpha = 1;
    context.fillStyle = "#000000";
    context.fillRect(0, 0, 640, 480);
    //image
    context.globalAlpha = .25;
    context.drawImage(helloWorldImage, 0, 0);
```

Updating the globalAlpha Property for Text Display

Because the animation in this example is composed of fading text in and out on the Canvas, the main operation of the `drawScreen()` function is to update the `alpha` and `fadeIn` properties accordingly. If the text is fading in (`fadeIn` is `true`) we increase the alpha property by `.01`. If `alpha` is increased above `1` (the maximum it can be), we reset it back to `1` and then set `fadeIn` to `false`. This means that we will start fading out. We do the opposite if `fadeIn` is `false`, setting it back to `true` when the value of `alpha` hits `0`. After we set the `alpha` value, we apply it to the Canvas by setting `context.globalAlpha` to the value of the alpha property:

```
        if (fadeIn) {
            alpha += .01;
            if (alpha >= 1)  {
                alpha = 1;
                fadeIn = false;
            }
        } else {
            alpha -= .01;
            if (alpha < 0)  {
                alpha = 0;
                fadeIn = true;
            }
        }

        context.globalAlpha = alpha;
```

Drawing the Text

Finally, we draw the text to the Canvas, and the `drawScreen()` function is complete. In 20 milliseconds, `drawScreen()` will be called again, the `alpha` value will be updated, and the text will be redrawn:

```
        context.font         = "72px Sans-Serif";
        context.textBaseline = "top";
        context.fillStyle    = "#FFFFFF";
        context.fillText  (text, 150,200);
    }
```

The full code for this example is as follows:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>CH1EX5 : Hello World Animated </title>

<script src="modernizr.js"></script>
<script type="text/javascript">
window.addEventListener("load", eventWindowLoaded, false);

function eventWindowLoaded () {
    canvasApp();
}

function canvasSupport () {
      return Modernizr.canvas;
}

function canvasApp () {

          if (!canvasSupport()) {
             return;
          }

        var theCanvas = document.getElementById("canvasOne");
        var context = theCanvas.getContext("2d");

          function drawScreen() {
            //background
            context.globalAlpha = 1;
            context.fillStyle = "#000000";
              context.fillRect(0, 0, 640, 480);
            //image
            context.globalAlpha = .25;
            context.drawImage(helloWorldImage, 0, 0);


            if (fadeIn) {
                alpha += .01;
                if (alpha >= 1)  {
                    alpha = 1;
                    fadeIn = false;
                }
            } else {
                alpha -= .01;
                if (alpha < 0)  {
                    alpha = 0;
                    fadeIn = true;
                }
            }

            //text
            context.font         = "72px Sans-Serif";
```

```
            context.textBaseline = "top";


            context.globalAlpha = alpha;
            context.fillStyle    = "#FFFFFF";
            context.fillText  (text, 150,200);

        }

        var text = "Hello World";
        var alpha = 0;
        var fadeIn = true;
        //image
        var helloWorldImage = new Image();
        helloWorldImage.src = "html5bg.jpg";

        function gameLoop() {
            window.setTimeout(gameLoop, 20);
            drawScreen()
        }

        gameLoop();

}


</script>

</head>
<body>
<div style="position: absolute; top: 50px; left: 50px;">
<canvas id="canvasOne" width="640" height="480">
 Your browser does not support HTML 5 Canvas.
</canvas>
</div>
</body>
</html>
```

HTML5 Canvas and Accessibility: Sub Dom

The current method for implementing accessibility for the Canvas is referred to as the "Fallback DOM Concept," or "sub dom" (which involves adding text directly into the `<canvas></canvas>`).

It has been known for quite some time that the HTML5 Canvas, because it is an immediate mode bit-mapped area of the screen, does not lend itself to accessibility. There is no DOM or display list inherent in the Canvas to make it easy for accessibility devices (such as screen readers) to search for text and images and their properties drawn onto the Canvas. To make the Canvas accessible, a method known as "Fallback DOM

Concept," or *sub dom*, was devised. Using this method, developers create a DOM element to match each element on the Canvas and put it in the sub dom.

In the first Canvas "Hello World!" example we created (*CH1EX3.html*), the text "Hello World!" appeared above an image of the earth (see Figure 1-3). To create a sub dom for that example, we might do something like this:

```
<canvas id="canvasOne" width="500" height="300">
 <div>A yellow background with an image and text on top:
     <ol>
        <li>The text says "Hello World"</li>
        <li>The image is of the planet earth.</li>
        </ol>
   </div>
</canvas>
```

We should also make an accessible title for the page. Instead of:

```
<title>Ch1Ex6: Canvas Sub Dom Example </title>
```

Let's change it to:

```
<title>Chapter 1 Example 6 Canvas Sub Dom Example </title>
```

To test this, you need to get a screen reader (or a screen reader emulator). Fangs is a screen reader emulator add-on for Firefox that can help you debug the accessibility of your web pages by listing out in text what a screen reader might say when your page is read. After you install the add-on, you can right-click on the web page and choose the "View Fangs" option to see what a screen reader would see on your page.

For the Canvas page we just created, Fangs tells us that the page would read as follows:

"Chapter one Example six Canvas Sub Dom Example dash Internet Explorer A yellow background with an image and text on top *List of two items one* The text says quote Hello World quote *two* The image is of the planet earth.*List end*"

For Google Chrome, you can get the Google Chrome extension Chrome Vox, which will attempt to verbally read all the content on your pages.

(For the full example, see *CH1EX6.html* in the code distribution.)

Hit Testing Proposal

The "sub dom" concept can quickly become unwieldy for anyone who has tried to do anything more intricate than a simple Canvas animation. Why? Because associating fallback elements with Canvas interaction is not always an easy task, and it is complicated by screen readers that need to know the exact position of an element on the Canvas so that they can interpret it.

To help solve this issue, the Canvas needs some way to associate sub dom elements with an area on the bitmapped Canvas. The new W3C Canvas Hit Testing proposal outlines why this type of functionality should be added to the Canvas specification:

> In the current HTML5 specification, authors are advised to create a fallback DOM under the canvas element to enable screen readers to interact with canvas user interfaces. The size and position of those elements are not defined, which causes problems for accessibility tools—for example, what size/position should they report for these elements?
>
> Because canvas elements usually respond to user input, it seems prudent to solve the hit testing and accessibility issues with the same mechanism.

So what kind of mechanism are they suggesting?

The idea appears to be to create two new methods, `setElementPath(element)` and `clearElementPath(element)`, that will allow programmers to define (and delete) an area of the Canvas to use as a hit area, provided that it is associated with the fallback DOM element of the Canvas. It appears that you must have an accessible fallback DOM element to provide to `setElementPath()` in order to associate it for the hit detection. When a hit is detected, an event is fired, and all is right in the world.

So what does this mean for developers?

For user interfaces with stationary interfaces, it will make things a lot easier. How many times have you wanted to create a simple way to click buttons on a game interface but had to use the same hit detection routines you wrote for your in-game sprite interactions? (For us? Every time.) However, for moving sprites in your game, it might be less useful. You will have to update the `setElementPath()` method and the fallback DOM element with new coordinate data every time something moves, which means triple overhead for a game that is probably not accessible in the first place.

Still, this is a good move by the W3C, because making the Canvas accessible for user interfaces is another huge step in making it more widely accepted for web applications. We hope these two new methods are added to the specification as soon as possible. The good news is, as of December 2012, the "Hit Testing Proposal" has been incorporated into the specification for the next version of Canvas, dubbed Canvas Level-2.

What's Next?

So now you should have a basic understanding of the HTML and JavaScript that we will use to render and control HTML5 Canvas on an HTML page. In the next chapter, we will take this information and expand on it to create an interactive application that uses the canvas to render information on the screen.