An Introduction

HTML5 for Publishers

O'REILLY®

Sanders Kleinfeld

HTML5 for Publishers

HTML5 for Publishers

Sanders Kleinfeld

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

HTML5 for Publishers
by Sanders Kleinfeld

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*.

Editor: Kathleen Meyer **Cover Designer:** Karen Montgomery

Revision History for the First Edition:

2011-10-06	First release
2011-10-18	Second release
2011-12-14	Third release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449314606* for release details.

ISBN: 978-1-449-31460-6
1323827047

Table of Contents

Introduction

HTML5 is revolutionizing the Web, and now it's coming to your ebook reader! In this book, I give an overview of three areas of HTML5 that offer great promise to ebook publishers looking to expand beyond traditional text-and-graphic narratives: Canvas, Geolocation, and Audio/Video. After a brief tutorial of the HTML markup and JavaScript code used to implement these features, I transition into some examples that put HTML5 in action:

- A graphing calculator to display algebraic equations on the Canvas
- A children's finger-painting application for drawing pictures on the page
- A geolocated work of fiction customized with details about the reader's current location
- An audio-enabled glossary that lets you click to hear the pronunciation of each term
- Embedded video content within instructional text to supplement a lesson

All code for the examples is available for download from GitHub. You can also demo the examples right in your browser by going to examples.oreilly.com.

For each topic area, I also discuss the current status of HTML5 compatibility with major EPUB reader platforms (for example, iBooks, Nook Color, Adobe Digital Editions). At the present time, support for HTML5/EPUB 3 is limited, and often quite experimental. But with the release of the EPUB 3 specification planned for this fall, HTML5 will officially be a part of the EPUB standard, and ereader support for HTML5's feature set should quickly follow suit. In the meantime, if you're reading the EPUB version of this book, the examples are embedded directly in the ebook, so you can experiment with them as your ereader compatibility permits.

Conventions Used in This Book

The following typographical conventions are used in this book:

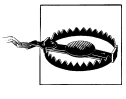
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*HTML5 for Publishers* by Sanders Kleinfeld (O'Reilly). Copyright 2011 O'Reilly Media, Inc, 978-1-4493-1460-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/0636920022473

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Thanks to Brian Sawyer, Kat Meyer, and Joe Wikert for giving me the opportunity to write this overview on HTML5 for publishers. In researching and compiling this piece, I relied heavily on a wealth of wonderful resources from O'Reilly Media, as well as some excellent web references and tutorials; please see the "Bibliography/Additional Resources" sections at the end of each chapter for details and links. In particular, I highly recommend *HTML5 Canvas* by Steve Fulton and Jeff Fulton for anyone who wants to

learn more about Canvas, and the many cited resources by Liza Daly for those looking to learn more about EPUB development.

And special thanks to Adam Witwer for being a great sounding board, and for generously helping me set aside time to work on this project.

CHAPTER 1

Canvas for Publishers

With the `<canvas>` element, publishers now have the opportunity to embed a dynamic sketchpad into HTML5 content. The HTML markup for doing so is quie simple:

```
<canvas id="my_first_canvas" width="200" height="225">
  The content you put here will show up if your rendering engine
  doesn't support the <canvas> element.
</canvas>
```

The `<canvas>` element accepts two attributes that specify the dimensions of your drawing area in pixels: `width` and `height`. Anything you place within the opening and closing tags of the element will only be displayed if the rendering engine does not support `<canvas>`; this gives you the option of providing fallback content for backward compatibility with non-HTML5 environments (see "HTML5 Canvas, EPUB, and Ereader compatibility" on page 17 for more on compatibility).

And that's where the HTML starts and ends; it merely sets aside the space within the HTML document in which to place your graphics. To actually draw on your `<canvas>`, you'll use JavaScript code to interact with the Canvas API, which provides you with an elegant set of functions for creating lines, arcs, shapes, and text. You also have access to more advanced graphic-manipulation calls to scale, rotate, or crop your images.

Drawing on your <canvas>

Let's draw a smiley face on the canvas we just created above. Here's a list of the Canvas API functions we'll use:

`strokeRect(`x_1`, `y_1`, `x_2`, `y_2`)`
: Draw a rectangular outline from the point (x_1, y_1) to (x_2, y_2). *Note*: by default, the "origin" of the Canvas (0,0) is its top-left corner, and x- and y-coordinates are measured to the right and down, respectively.

`beginPath()`
: Start a line drawing.

`closePath()`
: End a line drawing that was started with `beginPath()`.

`arc(x, y, arc_radius, angle_radians_beg, angle_radians_end)`
: Specify an arc, where (*x*, *y*) is the center of the circle encompassing the arc, *`arc_radius`* is the radius of this circle, and *`angle_radians_beg`* and *`angle_radians_end`* indicate the beginning and end of the arc angle in radians.

`stroke()`
: Draw the border of the path specified within `beginPath()`/`closePath()`. *Note*: If you don't include the `stroke()` call, your path will not appear on the canvas.

`fill()`
: Fill in the path specified within `beginPath()`/`closePath()`.

`fillText(your_text, x1, y1)`
: Add text to the canvas, starting at the point (x_1, y_1).

We'll also use the following attributes in conjunction with these properties to specify colors and styles:

`lineWidth`
: Width of the border of your bath

`strokeStyle`
: Color of the border of your path

`fillStyle`
: Color of the fill (interior) of your path

`font`
: Font and size of your text

And here's the code that puts it all together:

```
function drawPicture() {

    my_canvas.strokeRect(0,0,200,225) // to start, draw a border around the canvas

    //draw face
    my_canvas.beginPath();
    my_canvas.arc(100, 100, 75, (Math.PI/180)*0, (Math.PI/180)*360, false); // circle
dimensions
    my_canvas.strokeStyle = "black"; // circle outline is black
    my_canvas.lineWidth = 3; // outline is three pixels wide
    my_canvas.fillStyle = "yellow"; // fill circle with yellow
    my_canvas.stroke(); // draw circle
    my_canvas.fill(); // fill in circle
    my_canvas.closePath();

    // now, draw left eye
    my_canvas.fillStyle = "black"; // switch to black for the fill
    my_canvas.beginPath();
    my_canvas.arc(65, 70, 10, (Math.PI/180)*0, (Math.PI/180)*360, false); // circle
dimensions
```

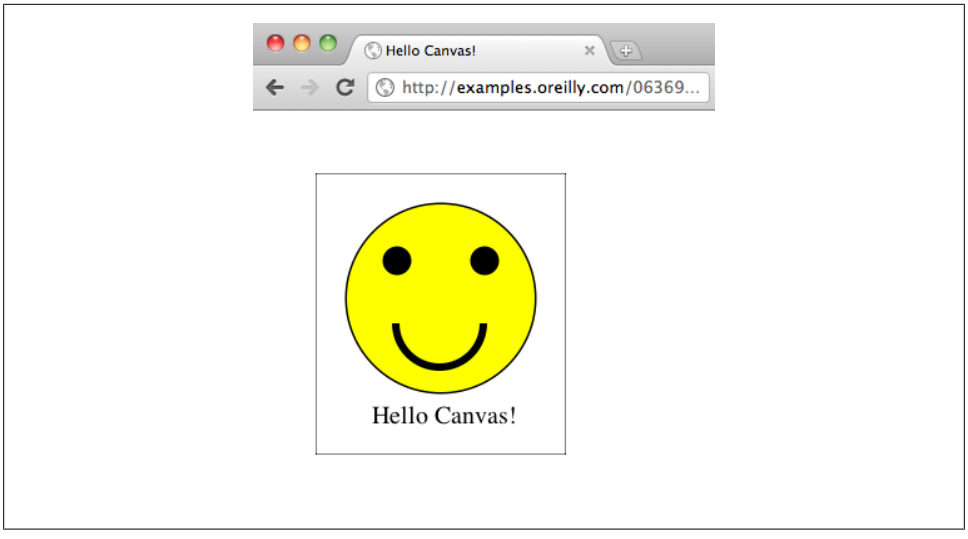


Figure 1-1. Hello Canvas!

```
        my_canvas.stroke(); // draw circle
        my_canvas.fill(); // fill in circle
        my_canvas.closePath();

        // now, draw right eye
        my_canvas.beginPath();
        my_canvas.arc(135, 70, 10, (Math.PI/180)*0, (Math.PI/180)*360, false); // circle
    dimensions
        my_canvas.stroke(); // draw circle
        my_canvas.fill(); // fill in circle
        my_canvas.closePath();

        // draw smile
        my_canvas.lineWidth = 6; // switch to six pixels wide for outline
        my_canvas.beginPath();
        my_canvas.arc(99, 120, 35, (Math.PI/180)*0, (Math.PI/180)*-180, false); // semicircle
    dimensions
        my_canvas.stroke();
        my_canvas.closePath();

        // Smiley Speaks!
        my_canvas.fillStyle = "black"; // switch to black for text fill
        my_canvas.font         = '20px _sans'; // use 20 pixel sans serif font
        my_canvas.fillText  ("Hello Canvas!", 45, 200); // write text
    }
```

Figure 1-1 shows the image displayed in the Safari Web browser. Click here to load this example in your browser, or take a look at the source code in GitHub.

If the functionality of the HTML5 Canvas were limited to the display of static images, however, its appeal would likely be quite limited. Who wants to write all that JavaScript

code, when you can easily to add images to an HTML document the old-school way—with an <img> tag!

But all that JavaScript is exactly what makes Canvas so powerful and feature-rich. Because you can directly manipulate the artwork with code, you can dynamically update what's displayed on the <canvas> in real time, and in response to user input. Instead of an inert smiley face, you can have a smiley face that winks every 18 seconds, or a smiley face that frowns when you click on it. The possibilities are endless: from games and jigsaw puzzles, to undulating photo galleries and molecular modeling.

Next, we'll look at a couple of HTML5 Canvas examples that can be used to enhance ebook content: a graphing calculator for linear algebraic equations, and a children's finger painting app.

Canvas Graphing Calculator

Most first-year algebra curricula contain a unit on graphing on the Cartesian coordinate plane. Many students intially have some difficulty grasping the concept of representing algebraic equations visually, as it's a real paradigm shift from traditional arithmetic. Graphing calculators, both hardware and software, are helpful tools in the teaching process, as they allow learners to quickly and efficiently experiment with plotting equations, so they can understand how changes made in an equation affect the shape of the graph.

In this section, we'll use HTML5 Canvas to implement a very basic graphing calculator for simple linear equations that can be embedded in algebra ebooks. Figure 1-2 displays the graphing calculator interface we'll create: a two-dimensional coordinate plane with *x*- and *y*-axes marked in red, and a set of buttons below for graphing linear equations on the grid.

Here's the HTML we'll use to construct the graphing calculator page. Our coordinate plane will be constructed in the <canvas> element, highlighted in **bold**:

```
<html lang="en">
<head>
<title>Graphing Calculator</title>
<script src="modernizr-1.6.min.js" type="text/javascript"></script>
<script src="graph_calc.js" type="text/javascript"/></script>
</head>
<body>
<div>
<h1>Graphing Calculator</h1>
<p style="color: red;"><span id="status_message">Click a button below the grid to
graph an equation</span></p>
<canvas id="canvas" width="400" height="400">
 Your browser does not support the HTML 5 Canvas.
</canvas>
<form>
<input type="button" id="y_equals_x" value="y = 1x" style="color: green;"/>
<input type="button" id="y_equals_negative_x" value="y = -1x" style="color: purple;"/>
```

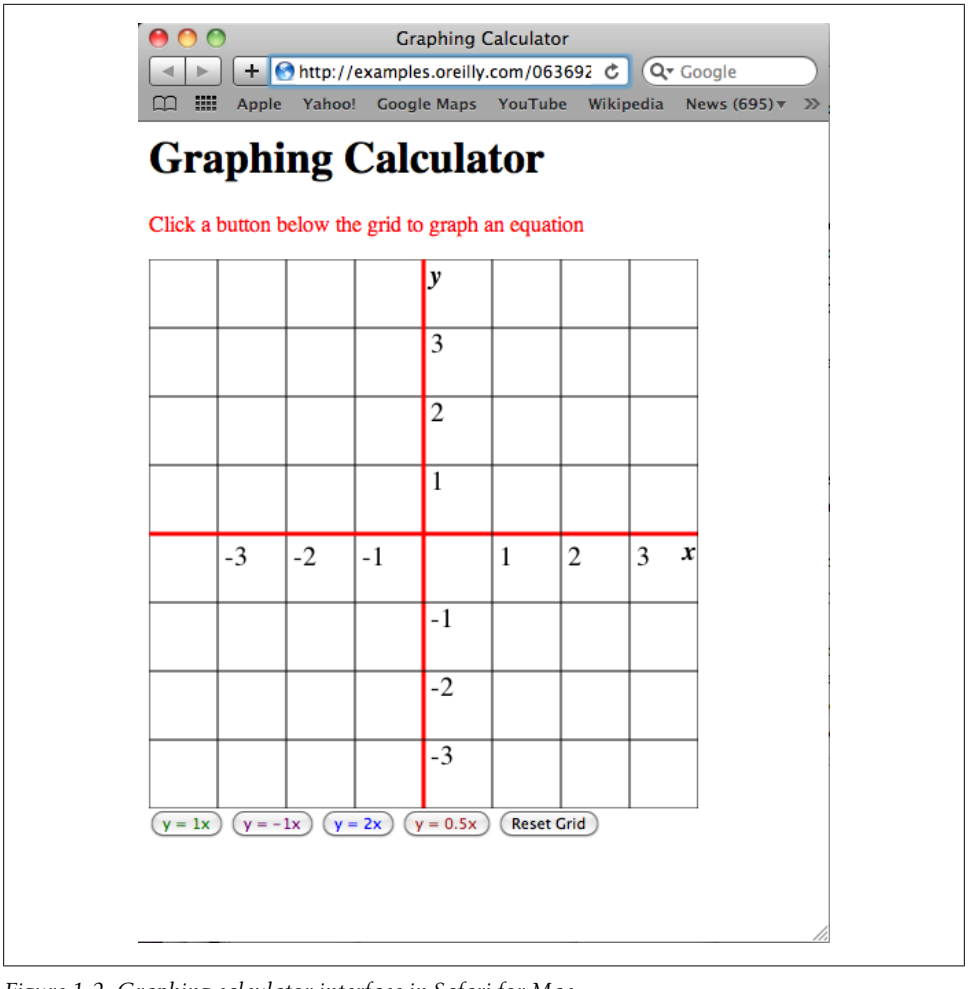


Figure 1-2. Graphing calculator interface in Safari for Mac

```
<input type="button" id="y_equals_two_x" value="y = 2x" style="color: blue;"/>
<input type="button" id="y_equals_one_half_x" value="y = 0.5x" style="color: brown"/>
<input type="button" id="reset_grid" value="Reset Grid"/>
</form>
</div>
</body>
</html>
```

To construct the grid on the `<canvas>` and graph lines, we'll make use of a few new Canvas API functions:

`moveTo(x, y)`
: Move the Canvas "cursor" to the (*x*, *y*) location specified. Subsequent drawing operations you perform will use this location as the starting point.

`lineTo(x, y)`
: Draw a line from the current Canvas "cursor" location to the (*x*, *y*) location specified.

`translate(x, y)`
: Allows you to set a new "origin" for the Canvas, from which *x*- and *y*-coordinates are measured. By default, the Canvas origin is its top-left corner, but to simplify the graphing calculator code, it will be helpful to relocate the Canvas origin to coincide with the coordinate-plane origin at the center of the grid.

Here's the `drawGrid()` function for creating the coordinate grid on the Canvas:

```
function drawGrid() {

    var i = 0;
    axis_pos = 1;
    can_width = theCanvas.width; // Get the width of the canvas

    // Loop through and draw horizontal/vertical lines at each eighth of the grid
    // All logic below presumes canvas has square dimensions
    for (i=0;i<=can_width;i+=(can_width)/8)
    {
        if (i == (can_width)/2) // Special handling for horiz/vert axes
        {
            context.lineWidth = 3; // Axes are thicker...
            context.strokeStyle = 'red'; //... and in red
        }
        else
        {
            context.lineWidth = 1;
            context.strokeStyle = 'black';
        }
        // First draw vertical line
        context.beginPath();
        context.moveTo(i, 0);
        context.lineTo(i, can_width);
        context.stroke();
        context.closePath();
        // Then draw horizontal line
        context.beginPath();
        context.moveTo(0, i);
        context.lineTo(can_width, i);
        context.stroke();
        context.closePath();
    }
    // Then add axis number labels
    context.font         = '20px _sans';
    context.textBaseline = 'top';
    // Move canvas origin to center of grid
    context.translate(can_width / 2, can_width / 2);
    for (i=-3;i<=3;i++) {
        if (i != 0) { // Skip labeling origin
            // horizontal label
            context.fillText  (i, i*(can_width/8) + 5, 5);
```

```
            // vertical label
            context.fillText  (i, 5, -i*(can_width/8));
        }
    }
    // Add bold-italic x- and y labels on the axes, too
    context.font = 'italic bold 20px _sans';
    context.fillText ("x", (can_width/2)-12, 1);
    context.fillText ("y", 4, -(can_width/2));
}
```

First, we grab the width of the `<canvas>` element (`theCanvas.width`), and then we run a `for` loop to draw eight evenly spaced vertical and horizontal lines across the grid; the *x*- and *y*-axes are handled specially, bolded and colored red. Then we run one more `for` loop to add number labels (from -3 to 3) on both axes. Finally, we add *x*- and *y*-labels to clearly identify the two axes.

Now that the grid is in place, we also need a function that will graph a specified linear equation on the plane. We'll create a function called `draw_grid_line()` that is capable of plotting any linear equation that can be expressed in the format $y = \mathbf{m}x$, where **m** is the slope of the equation. This function will take two parameters: ***`slope`*** and ***`color`***, which accepts a valid CSS color value. Here's the code:

```
function draw_grid_line (slope, color) {
    if (graph_in_progress == "yes") {
        // Only draw one line at a time
        alert("Another line is being drawn. Please wait until it's complete");
    } else {
        init_x = -(theCanvas.width)/2; // start with x = left edge of grid
        // Note: Must reverse sign y-coordinate, as negative y-coordinates are top half of grid by default,
not bottom
        init_y = -(init_x) * slope // y = mx
        new_x = init_x;
        new_y = init_y;
        var drawLineIntervalId = 0;
        status_message.innerHTML = "Drawing equation y = " + slope + "x";
        graph_in_progress = "yes" // line now being drawn
        drawLineIntervalId = setInterval(do_animation, 33);
    }

    function do_animation () {
        context.lineWidth = 6;
        context.strokeStyle = color;
        context.beginPath();
        context.moveTo(init_x, init_y);
        context.lineTo(new_x, new_y);
        context.stroke();
        context.closePath();
        new_x = new_x + 5
        new_y = -(new_x) * slope
        context.lineTo(new_x, new_y)
        if (new_x == theCanvas.width + 5) {
            clearInterval(drawLineIntervalId); // stop animation when line is complete
            graph_in_progress = "no" // line is now done
            status_message.innerHTML = "Click a button below the grid to graph an
```

```
equation"
        }
    }
}
```

First, we check to see if another line is currently being drawn, and only proceed if this is not the case; this ensures that the function is not called twice simultaneously, since it is designed to track the coordinates of one line at a time. Then we calculate the initial *x*- and *y*-coordinates for the line (`init_x` and `init_y`). For `init_x`, we start at the left edge of the grid; since we reset the origin of the Canvas to the center of the grid in the `drawGrid()` function, the leftmost *x*-coordinate is now equal to the negative of one-half of the canvas width (`-(theCanvas.width)/2`). Then, we calculate the corresponding `init_y` by taking the negative of `init_x` and multiplying by the slope.

It's necessary to reverse the sign when calculating the *y*-coordinate, because even though we reset the origin of the Canvas to the center of the grid, y-coordinates are still measured differently on the canvas than on the traditional Cartesian coordinate plane. On the Cartesian coordinate plane, y- values go from negative to positive as you *travel up the y-axis from bottom to top*, but on the Canvas, they go from negative to positive as you *travel down the y-axis from top to bottom*. Flipping the sign on the *y*-value resolves this discrepancy.

Once we have the starting point of the line, we can go ahead and trigger the animation that draws the line on the grid. We update the status message above the graphing calculator, and then set the `graph_in_progress` variable to `yes` to indicate that the line is now being drawn. Then we call the embedded function `do_animation()` using the JavaScript `setInterval()` method. `setInterval` allows us to repeatedly call a function at designated intervals of time, measured in milliseconds. Here, we call `do_animation()` every 33 milliseconds, which will draw the line at a nice speed.

Each time `do_animation()` is called, we calculate a new ending point for our line (`new_x` and `new_y`) by increasing the *x*-coordinate by 5 and calculating the corresponding *y*-coordinate by taking the negative of `new_x` and multiplying by the slope. Then we draw a line from (`init_x`, `init_y`) to (`new_x`, `new_y`). As `do_animation()` is called in succession, each new line drawn is a little bit longer than the last, which creates the visual impression that one continuous line is being drawn across the grid.

When the *x*-coordinate in `new_x` exceeds the right edge of the Canvas, we call `clearInterval()` to end the animation, and then set `graph_in_progress` to `no` and reset the status message above the calculator, so that `draw_grid_line()` is now ready to graph another linear equation when triggered.

All that's left to code is the initial setup upon page load, and the functionality for the graphing calculator buttons. Here's the code that initializes the graphing calculator:

```
window.addEventListener('load', eventWindowLoaded, false);
function eventWindowLoaded() {
```

```
    canvasApp();
}

function canvasSupport () {
  return Modernizr.canvas;
}

function canvasApp(){

    if (!canvasSupport()) {
       return;
    } else {
        var theCanvas = document.getElementById('canvas');
        var context = theCanvas.getContext('2d');
    }

    initGraphCalculator();
    var graph_in_progress = "no"

    function initGraphCalculator() {
        drawGrid();
        var y_equals_x_button = document.getElementById("y_equals_x");
        y_equals_x_button.addEventListener('click', y_equals_xPressed, false);
        var y_equals_negative_x_button =
document.getElementById("y_equals_negative_x");
        y_equals_negative_x_button.addEventListener('click',
y_equals_negative_xPressed, false);
        var y_equals_two_x_button = document.getElementById("y_equals_two_x");
        y_equals_two_x_button.addEventListener('click', y_equals_two_xPressed, false);
        var y_equals_one_half_x_button =
document.getElementById("y_equals_one_half_x");
        y_equals_one_half_x_button.addEventListener('click',
y_equals_one_half_xPressed, false);
        var reset_grid_button = document.getElementById("reset_grid");
        reset_grid_button.addEventListener('click', reset_grid_buttonPressed, false);
        status_message = document.getElementById("status_message");
    }
```

First, when the window finishes loading, we check and see if the user's environment supports the `<canvas>` tag (if not, the code stops executing). Then, `drawGrid()` is triggered, and event listeners are added to the buttons below the graphing calculator, so that when the user clicks them, the corresponding functions will be executed:

```
function y_equals_xPressed(e) {
    draw_grid_line(1, "green");
}

function y_equals_negative_xPressed(e) {
    draw_grid_line(-1, "purple");
}

function y_equals_two_xPressed(e) {
    draw_grid_line(2, "blue");
}
```

```
function y_equals_one_half_xPressed(e) {
    draw_grid_line(1/2, "brown");
}

function reset_grid_buttonPressed(e) {
     theCanvas.width = theCanvas.width; // Reset grid
     drawGrid();
}
```

Now, when any of the four equation buttons is clicked, the `draw_grid_line()` function is called with the appropriate slope and color values.

When the Reset Grid button is clicked, the `width` attribute is reset to its current value, which results in all contents of the `<canvas>` elements being deleted. Then, the `drawGrid()` function is called again to redraw the coordinate plane on the Canvas.

With our code complete, we're now ready to test out the graphing calculator. Go ahead and try it out on examples.oreilly.com. Figure 1-3 shows the graphing calculator in action in the iBooks reader for iPad.

You can also download the full graphing calculator code from GitHub and experiment with it locally in your Web Browser or ereader.

Canvas Finger Painting

Doing animations on the HTML5 Canvas is cool, but what's even cooler is letting the user draw on the Canvas herself. With the advent of touchscreen phones, tablets, and ereaders, this becomes even more compelling, as the user can draw directly on the screen with her finger, rather than using a mouse or trackpad. In this section, we'll look at how to implement a simple "finger painting" app in the Canvas, which would be a nice fit for a children's ebook—for example, a story that lets kids draw their own illustrations to accompany the text, or a preschool textbook that uses the finger painting to teach colors and shapes.

Here's the HTML we'll use to construct the Finger Painting page; the `<canvas>` tag which will hold the drawing area is higlighted in **bold**:

```
<!doctype html>
<html lang="en">
<head>
<title>Finger Painting</title>
<script src="modernizr-1.6.min.js"></script>
<script src="finger_painting.js"></script>
</head>
<body>
<div>
<canvas id="canvas" width="500" height="500">
 Your browser does not support the HTML 5 Canvas.
</canvas>
</div>
<div>
```

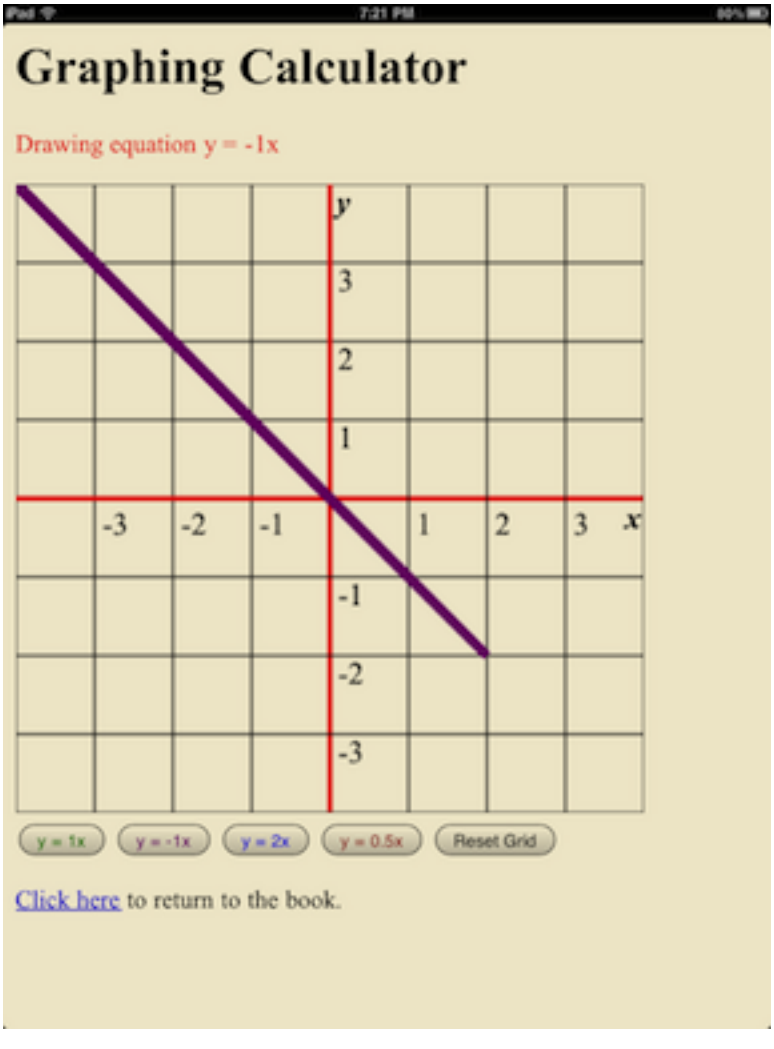


Figure 1-3. Graphing calculator in iBooks

```
<h1>Finger Painting</h1>
<p>Click/tap a color below to select a color, and then drag/swipe on the
  canvas above to draw a picture.</p>
<p>Color selected: <span id="color_chosen">Black</span></p>
<p>
<input type="button" id="Red" style="background-color: red; width: 25px;
height: 25px;"/>
<input type="button" id="Orange" style="background-color: orange; width: 25px;
height: 25px;"/>
<input type="button" id="Yellow" style="background-color: yellow; width: 25px;
height: 25px;"/>
```

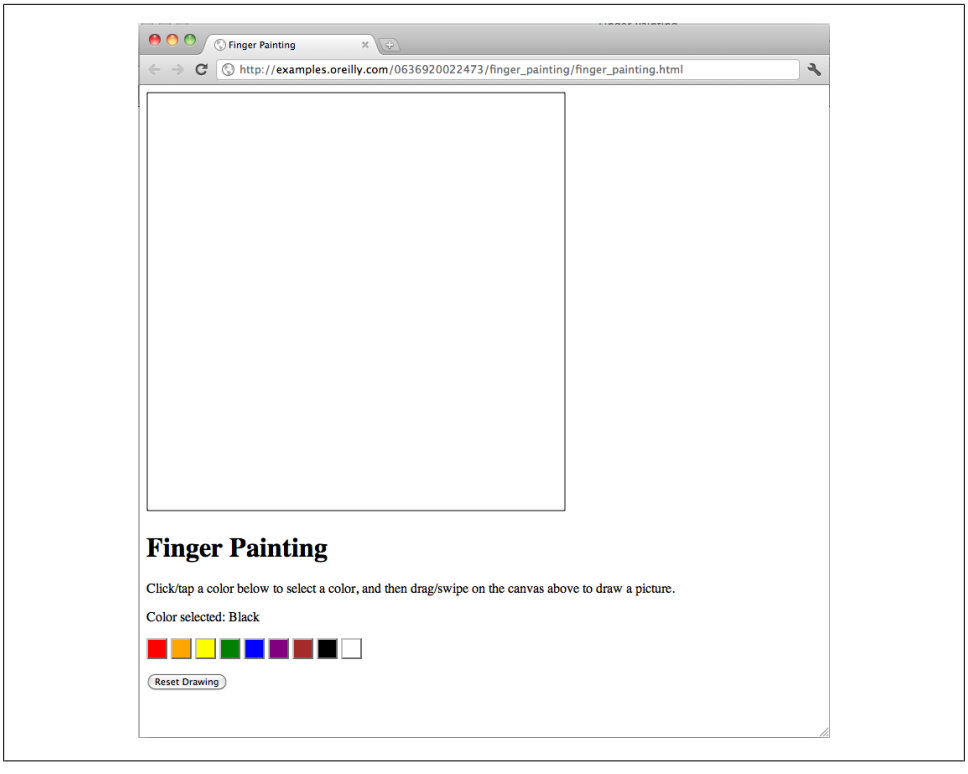


Figure 1-4. Finger painting interface in Google Chrome

```
<input type="button" id="Green" style="background-color: green; width: 25px;
height: 25px;"/>
<input type="button" id="Blue" style="background-color: blue; width: 25px;
height: 25px;"/>
<input type="button" id="Purple" style="background-color: purple; width: 25px;
height: 25px;"/>
<input type="button" id="Brown" style="background-color: brown; width: 25px;
height: 25px;"/>
<input type="button" id="Black" style="background-color: black; width: 25px;
height: 25px;"/>
<input type="button" id="White" style="background-color: white; width: 25px;
height: 25px;"/>
</p>
<p><input type="button" id="reset_image" value="Reset Drawing"/></p>
</div>
</body>
</html>
```

Note that the color palette below the Canvas has been implemented using `<input>` buttons, which are styled with CSS to be the appropriate color and size. Figure 1-4 displays the page in Chrome for Mac.

In order for the user to be able to draw on the screen, we'll need to be able to track his cursor motions and clicks within the Canvas. We can do so by adding event listeners to the `<canvas>` element as follows:

```
theCanvas.addEventListener('mousedown', mouse_pressed_down, false);
theCanvas.addEventListener('mousemove', mouse_moved, false);
theCanvas.addEventListener('mouseup', mouse_released, false);
```

Now when a user presses down on the mouse within the `<canvas>`, a `mousemove` event is triggered in the browser, and our event listener calls the `mouse_pressed_down` function. Similarly, when the mouse is moved within the dimensions of the Canvas, the `mouse_moved` function is called, and when the mouse button is released, the `mouse_released` function is called. Let's take a look at these three functions:

```
function mouse_pressed_down (ev) {
    begin_drawing = true;
    context.fillStyle = colorChosen.innerHTML;
}

function mouse_moved (ev) {
    var x, y;
    // Get the mouse position in the canvas
    x = ev.pageX;
    y = ev.pageY;

    if (begin_drawing) {
        context.beginPath();
        context.arc(x, y, 7, (Math.PI/180)*0, (Math.PI/180)*360, false);
        context.fill();
        context.closePath();
    }
}

function mouse_released (ev) {
   begin_drawing = false;
}
```

The `mouse_pressed_down` function serves to "turn on" a drawing event on the canvas. It sets the variable `begin_drawing` to `true`, and then sets the fill color to be used to the current color selected from the color palette.

Then when the `mouse_moved` function is called (which occurs any time the mouse is moved somewhere within the Canvas), we get the cursor's coordinates using the `pageX`/`pageY` properties. We check if the `begin_drawing` variable is set to `true`, which means that the user has the mouse button pressed down, and if so, we draw a circle of the designated color with a radius of 7 pixels at the cursor location.

As long as the mouse button is held down while the mouse is moved over the Canvas, the `mouse_moved` function will be called every single time the cursor location changes, which means that circles will continue to be drawn as the mouse moves, resulting in an effect quite similar to the Paintbrush tool in many image-editing applications.

When the mouse button is released, the `begin_drawing` variable is set back to `false`, which "turns off" the drawing event. This ensures that drawing occurs only when the mouse is held down, and not when the mouse is moved over the Canvas without the button being pressed.

The above code works great on desktop and laptop browsers, where a mouse is used to interface with screen elements, but what about touchscreen devices like the iPad? In general, touchscreen browsers do not support `mousedown`/`mousemove`/`mouseup` events, as there is no mouse button or mouse cursor that they can track; all those features are replaced with finger taps and swipes. However, WebKit-based browsers support a corresponding set of events for tracking finger motions in the browser: `touchstart`/`touchend`/`touchmove`. So we can implement the same drawing functionality as above using a `touchmove` event listener:

```
theCanvas.addEventListener('touchmove', touch_move_gesture, false);
```

And the following `touch_move_gesture` function:

```
function touch_move_gesture (ev) {
    // For touchscreen browsers/readers that support touchmove
    var x, y;
    ev.preventDefault(); //override default UI behavior for better results on touchscreen devices
    context.beginPath();
    context.fillStyle = colorChosen.innerHTML;
    if(ev.touches.length == 1){
        var touch = ev.touches[0];
        x = touch.pageX;
        y = touch.pageY;
        context.arc(x, y, 7, (Math.PI/180)*0, (Math.PI/180)*360, false);
        context.fill();
    }
}
```

The `touchmove` handling for touchscreen devices is actually much simpler than the mouse-based version, because we don't even need to track `touchstart` and `touchend` events. When dealing with a mouse, we need to keep track of whether the mouse button is pressed or not when it's being moved on the canvas. In the touch version, we know that if the `touchmove` event has been triggered, the user has his finger on the screen and is intending to draw.

And that's the meat of the finger painting code. All that's left is the code to initialize the event listeners, track color palette selections, and implement the Reset Drawing button functionality. Example 1-1 shows the full JavaScript code for our finger painting application.

Example 1-1. Finger painting JavaScript code (finger_painting.js)

```
window.addEventListener('load', eventWindowLoaded, false);
function eventWindowLoaded() {
```

```
    canvasApp();
}

function canvasSupport () {
    return Modernizr.canvas;
}


function canvasApp(){
    if (!canvasSupport()) {
        return;
    }else{
        var theCanvas = document.getElementById('canvas');
        var context = theCanvas.getContext('2d');
        var redButton = document.getElementById("Red");
        var orangeButton = document.getElementById("Orange");
        var yellowButton = document.getElementById("Yellow");
        var greenButton = document.getElementById("Green");
        var blueButton = document.getElementById("Blue");
        var purpleButton = document.getElementById("Purple");
        var brownButton = document.getElementById("Brown");
        var blackButton = document.getElementById("Black");
        var whiteButton = document.getElementById("White");
        var colorChosen = document.getElementById("color_chosen");
        var resetButton = document.getElementById("reset_image");
        redButton.addEventListener('click', colorPressed, false);
        orangeButton.addEventListener('click', colorPressed, false);
        yellowButton.addEventListener('click', colorPressed, false);
        greenButton.addEventListener('click', colorPressed, false);
        blueButton.addEventListener('click', colorPressed, false);
        purpleButton.addEventListener('click', colorPressed, false);
        brownButton.addEventListener('click', colorPressed, false);
        blackButton.addEventListener('click', colorPressed, false);
        whiteButton.addEventListener('click', colorPressed, false);
        resetButton.addEventListener('click', resetPressed, false);
        drawScreen();
    }

    function drawScreen() {
        theCanvas.addEventListener('mousedown', mouse_pressed_down, false);
        theCanvas.addEventListener('mousemove', mouse_moved, false);
        theCanvas.addEventListener('mouseup', mouse_released, false);
        theCanvas.addEventListener('touchmove', touch_move_gesture, false);
        context.fillStyle = 'white';
        context.fillRect(0, 0, theCanvas.width, theCanvas.height);
        context.strokeStyle = '#000000';
        context.strokeRect(1,  1, theCanvas.width-2, theCanvas.height-2);
    }

    // For the mouse_moved event handler.
    var begin_drawing = false;

    function mouse_pressed_down (ev) {
        begin_drawing = true;
        context.fillStyle = colorChosen.innerHTML;
```

```
    }

    function mouse_moved (ev) {
        var x, y;
        // Get the mouse position in the canvas
        x = ev.pageX;
        y = ev.pageY;

        if (begin_drawing) {
            context.beginPath();
            context.arc(x, y, 7, (Math.PI/180)*0, (Math.PI/180)*360, false);
            context.fill();
            context.closePath();
        }
    }

    function mouse_released (ev) {
        begin_drawing = false;
    }

    function touch_move_gesture (ev) {
        // For touchscreen browsers/readers that support touchmove
        var x, y;
        ev.preventDefault(); //override default UI behavior for better results on touchscreen devices
        context.beginPath();
        context.fillStyle = colorChosen.innerHTML;
        if(ev.touches.length == 1){
            var touch = ev.touches[0];
            x = touch.pageX;
            y = touch.pageY;
            context.arc(x, y, 7, (Math.PI/180)*0, (Math.PI/180)*360, false);
            context.fill();
        }
    }

    function colorPressed(e) {
        var color_button_selected = e.target;
        var color_id = color_button_selected.getAttribute('id');
        colorChosen.innerHTML = color_id;
    }

    function resetPressed(e) {
        theCanvas.width = theCanvas.width; // Reset grid
        drawScreen();
    }
}
```

You can experiment with the Finger Painting app on examples.oreilly.com. Figure 1-5 shows a completed drawing in the Finger Painting app in the iBooks reader for iPad.

Pretty cool, right? Although maybe not as impressive as what you can do in some other touchscreen finger painting apps.

Figure 1-5. Author self-portrait in Finger Painting app in iBooks

HTML5 Canvas, EPUB, and Ereader compatibility

So, as we've seen, HTML5 Canvas is incredibly powerful and versatile, but the $64K question that's probably in your head is, "Which major ereading devices are currently compatible with `<canvas>` content?" Unfortunately, the answer at the time of writing (September 2011) is "Only one." Currently, the iBooks reader for iPad/iPhone/iPod touch is the only major ereader that supports and can render `<canvas>` content, which means that if you want to embed `<canvas>` apps directly in your EPUBs, you're likely limiting the audience of your ebook quite significantly. That said, here are a couple

options you may want to consider if you'd like to include Canvas apps in your EPUB, but want to mitigate the incompatibility with other EPUB readers (e.g., Nook, Sony Reader, Adobe Digital Editions):

- Include *fallback content* within your `<canvas>` elements that will be displayed if the user's ereader doesn't have Canvas support (see the beginning of this chapter for more details). This way, while readers won't be able to use the app, you can display text, images, etc., that can potentially convey some of the same information that would have been displayed on the Canvas.
- Instead of (or in addition to) just embedding your Canvas apps directly in the EPUB, consider hosting them on the Web and linking to them from your EPUB, so that readers can access them from a traditional Desktop or mobile web browser. Many modern hardware ereaders (again, iBooks, but also the Nook Color) have built-in web browsers, so even if the ereader software itself doesn't support `canvas`, the web browser may. Additionally, readers viewing your EPUB on a desktop/laptop machine (say, in Adobe Digital Editions) can click on your link and run the Canvas app in Firefox or Chrome. The one downside of this approach is that readers will obviously still need Internet access in order to access the app.

Testing for HTML5 Compatibility with Modernizr

You may have noticed that the preceding <canvas> examples in this chapter included a script called *modernizr-1.6.min.js* in the HTML:

```
<script src="modernizr-1.6.min.js" type="text/javascript"></script>
```

Modernizr is a free JavaScript library that is widely used across the Web to test browser compatibility with HTML5 and CSS3 features. Specifically, it provides the following functions for testing the features covered in this book:

`Modernizr.canvas`
: Tests for HTML5 Canvas support

`Modernizr.geolocation`
: Tests for Geolocation API support

`Modernizr.audio`
: Tests for HTML5 audio support. It can also test specifically for support for *.m4a*, *.mp3*, *.ogg*, and *.wav* file formats.

`Modernizr.video`
: Tests for HTML5 video support. It can also test specifically for support for H.264 *.mp4*, *.ogg*, and *.webm* file formats.

It's good practice to use a library like Modernizr to do compatibility testing for HTML5 features, so that you can provide fallbacks in your JavaScript code in the event the user's browser does not have support for the requisite HTML5 elements. However, note that many ereaders do not supoort scripting via JavaScript, so when adding HTML5 to ebook content, don't rely solely on Modernizr to provide your fallbacks. Your best bet

is to use Modernizr in conjunction with fallback content included directly in your HTML5 elements.

Longer-term, it's likely we'll start seeing more widespread support of HTML5 Canvas in ereaders within the next 6–12 months. HTML5 support is an integral part of the EPUB 3 specification approved by the IDPF (International Digital Publishing Forum) on October 11, 2011. Under EPUB 3, content documents must use HTML5 syntax, which means EPUB 3–compliant reading systems must support the `<canvas>` tag. That said, the spec also currently says that "EPUB Reading System support for scripting is optional," which means that ereaders are not required to support the JavaScript code that drives your Canvas applications. That said, in order to offer the full benefits of HTML5 and EPUB 3, I think it's a safe bet that most touchscreen, non-eInk EPUB ereaders will be providing full Canvas support in the near future, if for no other reason than to stay competitive with iBooks in offering publishers a platform for delivering rich interactive ebook content.

Bibliography/Additional HTML5 Canvas Resources

Here are some additional resources I highly recommend for learning more about Canvas:

HTML5 Canvas by Steve Fulton and Jeff Fulton (O'Reilly Media)
: A great introduction to HTML5 Canvas for beginners, and an even better reference book for advanced JavaScript programmers. This book covers everything from simple Canvas animations to advanced physics-based movement, and shows you how to design simple drawing apps and advanced arcade games alike.

Canvas Pocket Reference by David Flanagan (O'Reilly Media)
: Excellent mini-reference guide to the complete Canvas API.

Client-side Graphics with HTML5 Canvases: An O'Reilly Breakdown by David Griffiths (O'Reilly Media)
: Video tutorial on HTML5 Canvas. Learn how to build a retro arcade game.

Creating an HTML5 canvas painting application by Mihai Sucan
: If you're interested in building an HTML5 canvas painting application of your own, you may want to check out this cool tutorial.

Touching and Gesturing on the iPhone by nroberts
: The best tutorial I found online on touch events for WebKit browsers. If you have a mobile touchscreen browser, definitely check out this demo.

CHAPTER 2
Geolocation for Publishers

Location-based web sites have become so commonplace that we frequently take their functionality for granted. Type *Starbucks* into your Google search bar, and you'll get a list of numerous store locations within your immediate vicinity, without you even needing to specify a town or city. Flickr's map page has a "Find my location" button that will show you pictures taken in areas near you. And if you want to geotag your blog entris, WordPress has a plugin for that.

With the advent of HTML5, building geolocation functionality into web content has become incredibly easy, thanks to the release of the Geolocation API, which provides a standardized mechanism across all major web browsers for querying and receiving user location data.

Obtaining location data via the web browser requires just one line of JavaScript code to your script:

```
navigator.geolocation.getCurrentPosition(callback_function);
```

Where *`callback_function`* is the function that will be called by the browser when it completes its attempt to retrieve location data. Not every browser supports the geolocation API, however, and geolocation services are not available at all times in all locations, so you'll probably want to build in a bit more error handling—for example:

```
if (Modernizr.geolocation) {
    navigator.geolocation.getCurrentPosition(callback_function, throw_error);
} else {
    alert('Your browser/ereader does not support geolocation. Sorry.');
}

function throw_error(position) {
    alert('Unable to geolocate you. Sorry.');
}
```

The Geolocation API returns to two properties that contain location data to your callback function: `position.coords.latitude`, which contains the user's latitude, and `position.coords.longitude`, which contains his longitude. That's neat, but unless your

users are geography savants, the values (35.046872, -90.024971) probably mean far less to you than 3734 Elvis Presley Blvd, Memphis, TN.

Luckily, there are many great web services out there that will translate latitude/longitude coordinates into information far more transparent and valuable to humans: addresses, street maps, weather data, and more. Google Maps has a set of APIs available for obtaining location data and embedding maps right in your HTML documents, and in the next section, we'll query the GeoNames database to add real-time geographical data to a work of fiction.

A Geolocated Tale

Wouldn't it be great if authors could tailor their short stories, novels, and poems to the hometown, state, and country of each and every one of their readers? Instead of *The Merchant of Venice*, you could have *The Merchant of Dallas*, or *The Merchant of Yonkers*. Whether you find the idea enthralling or a bit appalling, the Geolocation API makes it possible.

To illustrate what's feasible on a smaller scale, we'll take the introduction to a short story, and geolocate it with details about the reader's current location. We'll start with some skeleton paragraphs that include placeholders for street address, city name, and current temperature—styled in bold red for emphasis. Example 2-1 shows the HTML, and Figure 2-1 shows it displayed in Safari for Mac:

Example 2-1. HTML for our story skeleton

```
<!doctype html>
<html lang="en">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8"/>
<title>A Geolocated Tale</title>
<script src="modernizr-1.6.min.js"></script>
<script src="geolocation-story.js"></script>
<script src="jquery-1.6.2.min.js"></script>
<style media="screen" type="text/css">
body {
 margin: 10px 5px 10px 5px;
}

em {
 font-weight: bold;
 font-style: normal;
 color: red;
}
</style>
</head>
<body>
<h1>A Geolocated Tale</h1>
<p>It was your typical <em id="weather_temp">LOADING
 TEMPERATURE</em>&#xb0;F day in <em id="city">LOADING CITY NAME</em> when
```

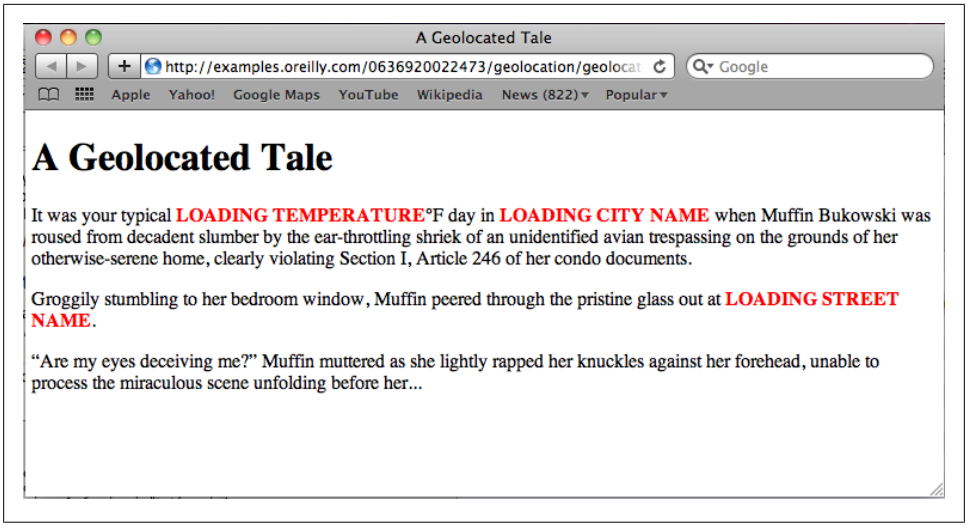


Figure 2-1. Our skeleton story in Safari

```
Muffin Bukowski was roused from decadent slumber by the
ear-throttling shriek of an unidentified avian trespassing on the
grounds of her otherwise-serene home, clearly violating Section I,
Article 246 of her condo documents.</p>

<p>Groggily stumbling to her bedroom window, Muffin peered
through the pristine glass out at <em id="street_address">LOADING STREET
NAME</em>.</p>

<p>&#8220;Are my eyes deceiving me?&#8221; Muffin muttered as she lightly
  rapped her knuckles against her forehead, unable to process the
  miraculous scene unfolding before her...</p>
</body>
</html>
```

Now, we'll need some JavaScript code to do the following:

1. Query the Geolocation API for the reader's latitude and longitude
2. Use the latitude and longitude values to then query the GeoNames database for the reader's current temperature, and fill in the corresponding placeholder in the story.
3. Use the latititude and longitude values to query GeoNames for the reader's street address and city, and again fill in the corresponding placeholders.

GeoNames has several dozen web services available for getting different types of geographical data. For our example, we can use their *extendedFindNearby* service to get street-address and city data, and their *findNearByWeather* service to get the temperate data. For most of their web services, GeoNames makes data available in both XML and JSON formats, but in the case of *extendedFindNearby*, only XML data is available. So, to make things simple, we'll query both services for XML. And to make things even

easier, we'll use the jQuery JavaScript library to help us interface with GeoNames and update our HTML placeholders (jQuery offers a set of convenience functions that greatly simplify both these tasks).

The code for Step 1 should look familiar:

```
window.addEventListener('load', eventWindowLoaded, false);

function eventWindowLoaded() {
    get_location();

    function get_location() {
        if (Modernizr.geolocation) {
            navigator.geolocation.getCurrentPosition(geolocate_story, throw_error);
        } else {
            alert('Your browser/ereader does not support geolocation. Sorry.');
        }

    function throw_error(position) {
        alert('Unable to geolocate you. Sorry.');
    }
}
```

As we saw in the beginning of the chapter, this code calls the `getCurrentPosition()` function to obtain latitude/longitude, with some error handling in place in case the user's environment doesn't support geolocation, or the geolocation attempt fails. This time, however, if geolocation succeeds, we'll call the `geolocate_story()` function to perform Steps 2 and 3.

In Step 2, we query GeoNames for temperature info:

```
function geolocate_story(position) {
    var geo_lat = position.coords.latitude;
    var geo_long = position.coords.longitude;
    // Get weather information
    $.ajax({
        type: 'GET',
        url: 'http://ws.geonames.org/findNearByWeatherXML?lat=' + geo_lat + '&lng=' +
geo_long,
        dataType: 'xml',
        success: function (weather_resp, xmlstatus) {
            var temperature_celsius = $(weather_resp).find("temperature").text();
            if (temperature_celsius != "") {
                // Weather temp data given in Celsius; convert to Fahrenheit, because I'm American
                var temperature_fahrenheit = 9/5*temperature_celsius + 32;
                $('#weather_temp').text(temperature_fahrenheit);
            } else {
                $('#weather_temp').text("TEMP NOT FOUND");
            }
        },
        error: function (xhr, status, error) {
            alert(error);
            $('#weather_temp').text("TEMP NOT FOUND");
        }
    })
```

The `geolocate_story()` function receives the latitude and longitude data in `position`, which is passed to it from the Geolocation API, and then we store that data in `geo_lat` and `geo_long`, respectively. To interface with GeoNames, we call jQuery's `$.ajax()` function, which lets us set up an XML query with the following parameters:

`type: 'GET'`
: Specifies that we'll make a HTTP GET request (as opposed to a POST request), which is compatible with the GeoNames API

`url: 'http://ws.geonames.org/findNearByWeatherXML?lat=' + geo_lat + '&lng=' + geo_long`
: Specifies the URL to be queried. For the *findNearByWeather* service, the URL is *http://ws.geonames.org/findNearByWeatherXML*, followed by the `lat` parameter for latitude and the `lng` parameter for longitude, where we supply the `geo_lat` and `geo_long` values we got from the Geolocation API.

`datatype: 'xml'`
: GeoNames is going to return XML to us, so this tells the `$.ajax()` function to parse the incoming data accordingly

`success: ...`
: Specifies what to do if our API call is successful; here, we'll call a function to process the weather data, which performs the following three steps

  1. Grabs the value of the `<temperature>` element in the XML returned from GeoNames (`$(weather_resp).find("temperature").text();`)
  2. If the temperature value is present, converts it from Celsius to Fahrenheit (`var temperature_fahrenheit = 9/5*temperature_celsius + 32;`)
  3. Updates the `weather_temp` `<span>` in the HTML with the Fahrenheit temperature (`$('#weather_temp').text(temperature_fahrenheit);`), or if no temperature value was returned, inserts the text "TEMP NOT FOUND"

`error: ...`
: Specifies what to do if our API call fails; here, we'll call a function that updates the `weather_temp` `<span>` in the HTML with the text "TEMP NOT FOUND" (`$('#weather_temp').text("TEMP NOT FOUND");`)

In Step 3 we again query the GeoNames API in similar fashion, but this time we get the user's location data (street address and city):

```
// Get full location information
$.ajax({
    type: 'GET',
    url: 'http://ws.geonames.org/extendedFindNearby?lat=' + geo_lat + '&lng=' +
geo_long,
    dataType: 'xml',
    success: function (loc_resp, xmlstatus) {
        var city_name = $(loc_resp).find("placename").text();
        if (city_name != "") {
            $('#city').text(city_name);
        } else {
```

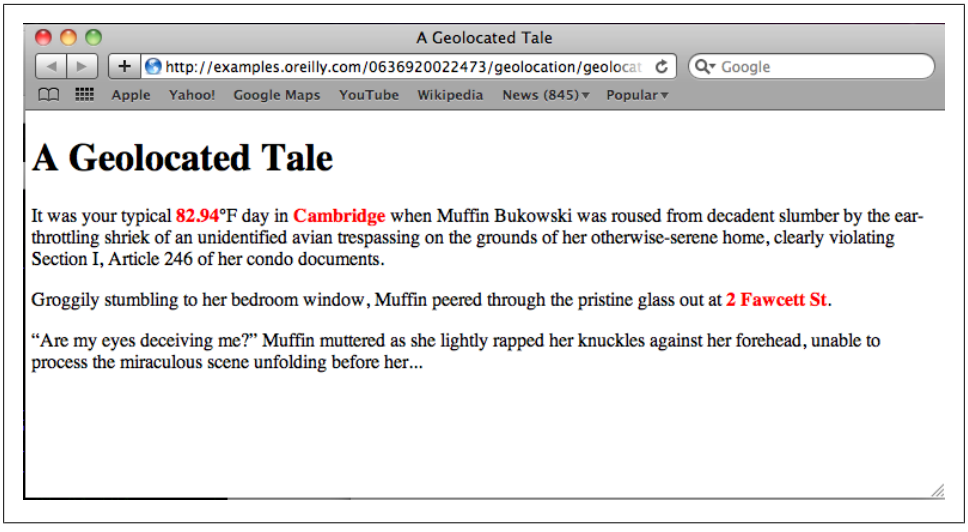


Figure 2-2. A Geolocated tale

```
            $('#city').text("CITY NOT FOUND");
        }
        var street_address = $(loc_resp).find("streetNumber").text() + " " + $
(loc_resp).find("street").text();
        if (street_address != "") {
            $('#street_address').text(street_address);
        } else {
            $('#street_address').text("ADDRESS NOT FOUND");
        }
    },
    error: function (xhr, status, error) {
        alert(error);
        $('#city').text("CITY NOT FOUND");
        $('#street_address').text("ADDRESS NOT FOUND");
    }
})
```

Most of the `$.ajax()` parameters are identical to those for the temperature query. For the `url` parameter, we substitute in the *extendedFindNearby* URL, which is *http://ws.geonames.org/extendedFindNearby*. For our `success` function, we get the values for the `placename` (typically corresponds to city), `streetNumber`, and `street` elements in the XML from GeoNames, and update the corresponding `<span>`s in the HTML. For our `error` function, we update the `<span>`s with boilerplate "NOT FOUND" text.

Figure 2-2 shows what the final story looks like, post-geolocation, if you happen to be visiting O'Reilly Media's Cambridge, Massachusetts, office on a warm, late-summer day.

Try loading the story in your own browser, and see what the text looks like. You can also download the full code from GitHub.

HTML5 Geolocation, EPUB, and Ereader Compatibility

As with HTML5 Canvas, Geolocation support is not yet widespread in EPUB readers. At the time of writing (September 2011) , among the major ereaders, iBooks is again the only one that supports the Geolocation API.

However, it's important here to draw a distinction between "supports the Geolocation API" and "supports querying geolocation web services (like GeoNames)." While iBooks can query the Geolocation API and will return the user's latitude/longitude coordinates, it *does not support* the necessary `XMLHttpRequest` functionality for querying Internet web services, throwing an `ABORT_ERR: XMLHttpRequest Exception 102` error (see Figure 2-3)

So at this time, it's not possible to embed our Geolocated Tale in an EPUB and have it successfully render in iBooks. However, you can instead post the story on the Web, and link to it within your EPUB (Mobile Safari on iPhone/iPod/iPad will indeed render the story successfully).

That said, it's still a bit disappointing that geolocation support really isn't available in ereaders at the present time. And what's even more unfortunate is that because the Geolocation API is not technically part of the HTML5 specification (it's its own separate W3C spec), it's also not technically a requirement of the EPUB 3 spec that ereaders support the Geolocation API. And of course, geolocation support is arguably much more controversial than support for Canvas, due to very legitimate concerns regarding security and privacy.

Also potentially a bit controversial is whether the use of geolocation services in EPUB runs counter to the specifications of the format, which maintain that all resources included directly in the book content need to be embedded directly in the EPUB file, and referenced in the EPUB's manifest. The philosophy here is that whether the user is online or offline, they should be able to access and view all the book content; a lack of Internet access should not cripple the reading experience. Does a geolocated work of fiction violate this precept? The answer to this question is a bit subjective, and likely depends on how integral a role geolocation plays in the book content, the type of fallbacks that are in place, etc.

Regardless, we've already reached a point on the Web where geolocation functionality is omnipresent and often taken for granted. So it seems likely that EPUB content creators and ereader developers alike will be strongly motivated to move toward a future that allows for geolocation-enhanced ebooks. The potential inherent in geolocated travel and restaurant guides alone seems huge, not to mention the opportunites for more avant-garde experimentation.

Bibliography/Additional Resources

Here's a list of additional Geolocation resources you may find useful:

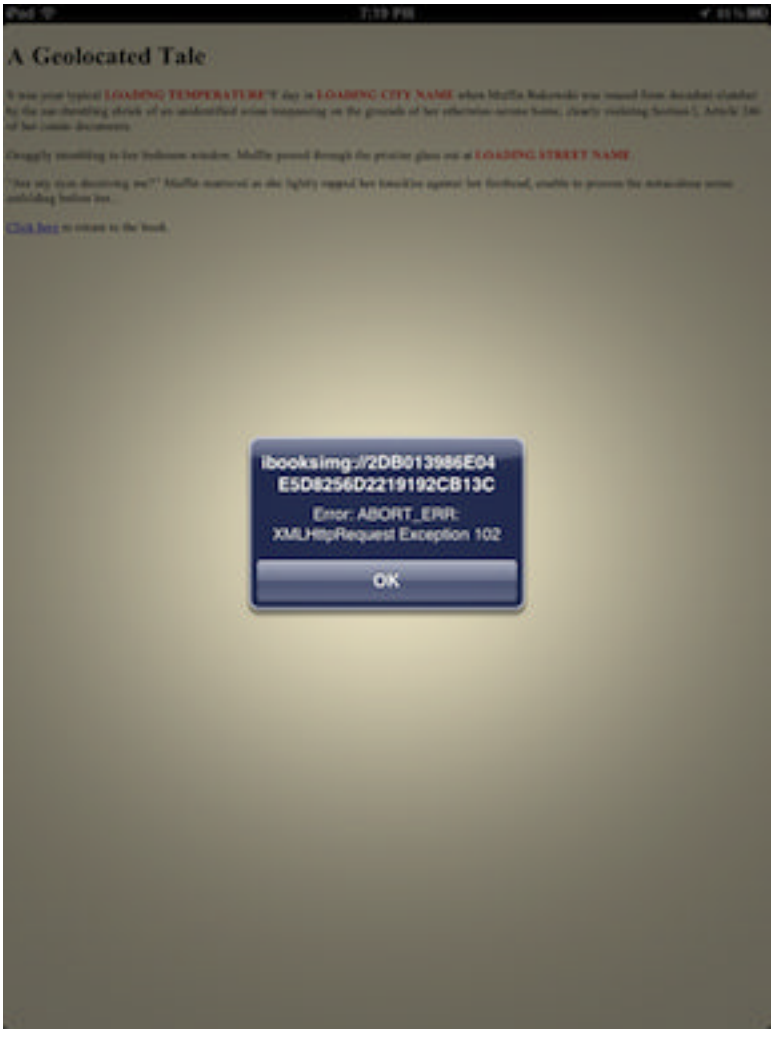


Figure 2-3. Geolocation XMLHttpRequest Exception 102 error in iBooks

HTML5 Geolocation by Anthony T. Holdener III
: Great primer that covers both how geolocation technology works, and provides many examples of how to harness it in your HTML5 applications

Who's using the W3C Geolocation API?
: Nice guide to which prominent websites are using the Geolocation API, their privacy policies, and whether users gets a heads-up that they are being geolocated

Geo-aware ebook demo *by Liza Daly*

Cutting-edge geolocation ebook demo in which the book's text adapts as the user's location changes.

CHAPTER 3

<audio>/<video> for Publishers

One of the most exciting features of HTML5 is that it offers native support for audio and video content. On the Web, this means that reliance on browser plugins in order faciliate display of multimedia content is becoming a thing of the past. On the ereader side, HTML5 and EPUB 3 open the door to embedding this same multimedia content directly within an ebook. Let's take a quick look at HTML5's new `<audio>` and `<video>` elements.

A Two-Minute Introduction to the <audio> and <video> Elements

The standard HTML5 `<audio>` element looks like this:

```
<audio id="new_slang">
<source src="new_slang.wav" type="audio/wav"/>
<source src="new_slang.mp3" type="audio/mp3"/>
<source src="new_slang.ogg" type="audio/ogg"/>
<em>(Sorry, &lt;audio&gt; element not supported in your
  browser/ereader, so you will not be able to listen to
  this song.)</em>
</audio>
```

The `<audio>` element serves as a container, which contains a series of `<source>` elements that reference your audio files (`src` attribute) in whichever formats you have available (`type` attribute). If you only have one format available, you can abbreviate the markup as follows:

```
<audio id="new_slang" src="new_slang.wav">No song for you!</audio>
```

However, current best practice is to provide audio in multiple audio formats—usually WAV, MP3, and Ogg—in order to ensure compatibility across the range of HTML5 audio–compliant browsers and ereaders (see "HTML5 Audio/Video Compatibility in the Browser and Ereaders" on page 38).

The <audio> element also accepts a handful of optional boolean attributes for customizing playback: controls, which displays a standard set of audio playback control buttons for the user; autoplay, which makes the audio play automatically, as soon as it's been loaded; and loop, which makes the audio repeat over and over and over...

```
<audio id="new_slang" src="new_slang.wav" controls autoplay loop>No song for you!</
audio>
```

Note that HTML5 permits boolean attributes to be supplied without a corresponding value (e.g., controls instead of controls="true"), but at the present time, for better compatibility in ereaders that are expecting XHTML content, I recommend including attribute values:

```
<audio id="new_slang" src="new_slang.wav" controls="true" autoplay="true"
loop="true">No song for you!</audio>
```

Note that the value of the attribute is immaterial: its mere presence is always equivalent to true and triggers its functionality. So, somewhat counterintuitively, both controls="true" and controls="false" (or controls="*whatever*") will all trigger the playback buttons to be displayed. If you don't want playback buttons, don't include the controls attribute.

The standard HTML5 <video> element is structured similarly to <audio>:

```
<video id="dancing_pony" width="300" height="300">
<source src="dancing_pony.mp4" type="video/mp4"/>
<source src="dancing_pony.ogg" type="video/ogg"/>
(Sorry, &lt;video&gt; element not supported in your
  browser/ereader, so you will not be able to watch the pony dance.)</video>
```

The width and height attributes on the <video> element specify the dimensions of the video. Additionally, <video> also supports the same boolean controls, autoplay, and loop attributes as <audio>, as well as the same shorthand markup if you only have one video format:

```
<video id="dancing_pony" width="300" height="300" src="dancing_pony.mp4"
controls="true" autoplay="true" loop="true">
No pony for you!
</video>
```

Also, as with <audio>, browser/ereader compatibility varies for different video formats. Encoding video in both MPEG-4 and Ogg formats is a safe bet (see "HTML5 Audio/Video Compatibility in the Browser and Ereaders" on page 38 for more details). In the following sections, we'll look at a couple of simple demos of audio and video in action.

An Audio-Enabled Glossary

One great use of HTML5 audio element is to add supplemental text-to-speech functionality to your book content. In this example, we'll add audio functionality to a glossay so that you can click/tap a button to hear the pronunciation of each term. We'll

use the <audio> element to embed the sound bytes, and JavaScript to control the audio playback. Example 3-1 shows the HTML for our glossary, which defines a few terms ebook publishers will likely be familiar with; <audio> elements are highlighted in **bold**.

Example 3-1. Audio-enabled glossary HTML

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Digital Publishing Mini-Glossary</title>
<script src="modernizr-1.6.min.js"></script>
<script src="glossary.js"></script>
<style media="screen" type="text/css">
dl {
  width: 400px;
}

dt {
  padding-top: 10px;
  padding-bottom: 5px;
  font-style: italic;
  color: red;
}

dd {
  margin-left: 1.5em;
}

.play-button {
  font-style: normal;
  color: blue;
  padding: 3px;
  border:2px solid;
  border-radius:6px;
  border-color: black;
  background-color: gray;
}

dt .play-button {
  margin-left: 6px;
}
</style>
</head>
<body>
<h1>Digital Publishing Mini-Glossary</h1>
<p>Click the <span class="play-button">&#x25b6;</span> button to hear the
  pronunciation of a term</p>
<!--Audio content -->
<audio id="epub">
<source src="audio/epub.wav" type="audio/wav"/>
<source src="audio/epub.mp3" type="audio/mp3"/>
<source src="audio/epub.ogg" type="audio/ogg"/>
<em>(Sorry, &lt;audio&gt; element not supported in your
  browser/ereader.)</em>
```

```
</audio>
<audio id="mobi">
<source src="audio/mobi.wav" type="audio/wav"/>
<source src="audio/mobi.mp3" type="audio/mp3"/>
<source src="audio/mobi.ogg" type="audio/ogg"/>
</audio>
<audio id="pdf">
<source src="audio/pdf.wav" type="audio/wav"/>
<source src="audio/pdf.mp3" type="audio/mp3"/>
<source src="audio/pdf.ogg" type="audio/ogg"/>
</audio>
<div class="glossary">
<dl>
<dt>EPUB <input type="submit" class="play-button" id="epub_button" value="&#x25b6;"/></dt>
<dd>An open standard for reflowable ebook content created and maintained by the <a
  href="http://idpf.org/">International Digital Publishing Forum
  (IDPF)</a> based on HTML, CSS, and XML technologies. Version 3.0 of
  EPUB will support HTML5.</dd>
<dt>Mobipocket <input type="submit" class="play-button" id="mobi_button" value="&#x25b6;"/
></dt>
<dd>A proprietary standard for reflowable ebook content developed by <a
  href="http://en.wikipedia.org/wiki/Mobipocket">Mobipocket SA</a>,
  and used by Amazon on its hardware and software Kindle
  platforms.</dd>
<dt>Portable Document Format (PDF) <input type="submit" class="play-button"
id="pdf_button" value="&#x25b6;"/></dt>
<dd>An open standard for page-based (non-reflowable) electronic documents created by Adobe
Systems
  that has been in use since the 1990s. Many ereader devices support
  PDF files, as well as EPUB or Mobi.</dd>
</dl>
</div>
</body>
</html>
```

Each glossary term is followed by an `<input>` button styled with CSS to resemble a play button. Figure 3-1 shows the glossary displayed in iBooks for iPad.

Next, we'll write some JavaScript that initiates the audio playback when one of the `<input>` buttons is clicked. Example 3-2 shows the code.

Example 3-2. Glossary JavaScript

```
window.addEventListener('load', eventWindowLoaded, false);

function eventWindowLoaded() {
    if (audio_support()) {
        set_up_audio();
    }
}

function audio_support () {
    return Modernizr.audio;
}
```

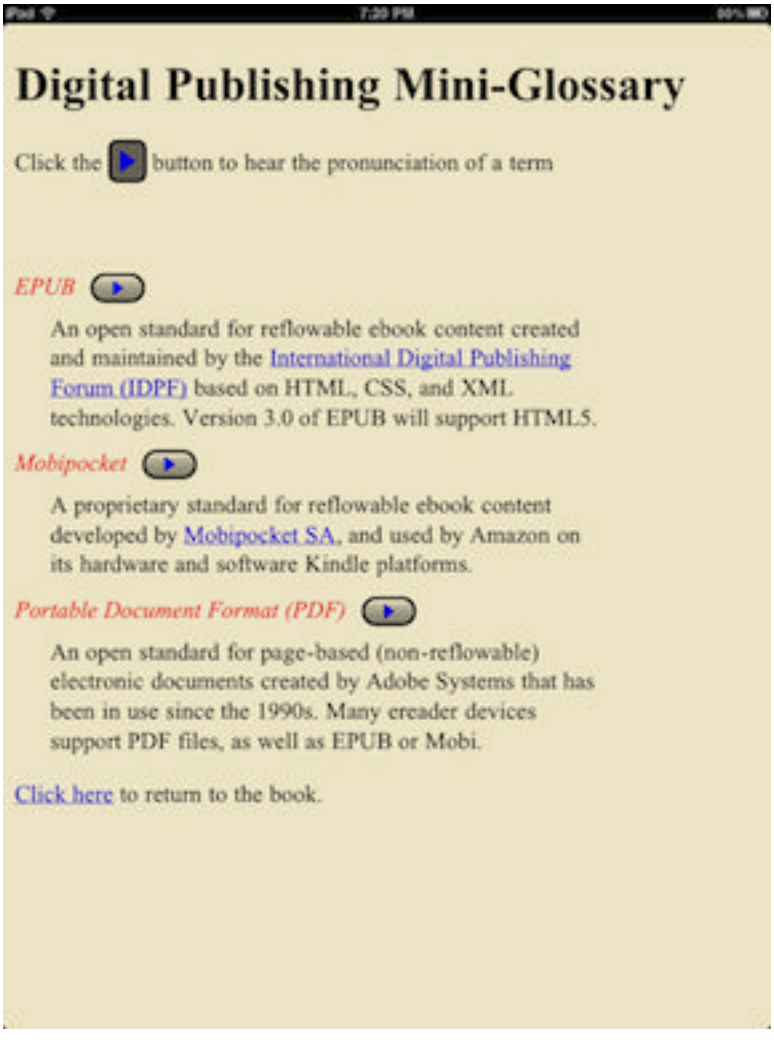


Figure 3-1. Audio-enabled glossary in iBooks

```
function set_up_audio() {
    var epub_audio = document.getElementById("epub");
    var mobi_audio = document.getElementById("mobi");
    var pdf_audio = document.getElementById("pdf");
    // Add play button functionality
    var epub_play_button = document.getElementById("epub_button");
    var mobi_play_button = document.getElementById("mobi_button");
    var pdf_play_button = document.getElementById("pdf_button");
    epub_play_button.addEventListener("click", play_epub, false);
    mobi_play_button.addEventListener("click", play_mobi, false);
    pdf_play_button.addEventListener("click", play_pdf, false);
```

```
    function play_epub() {
        epub_audio.play();
    }
    function play_mobi() {
        mobi_audio.play();
    }
    function play_pdf() {
        pdf_audio.play();
    }
}
```

As we've seen in previous examples, event listeners are used to track when each of the terms' play buttons is clicked, and call the corresponding `play_format` function. The one piece of audio-specific code is the `play()` method (highlighted in **bold** above) called on each of the `<audio>` elements. As you'd expect, this triggers the playback of the audio.

Try loading the glossary in your browser to hear the terms spoken aloud in all their glory. You can also download the code and audio media from GitHub.

An HTML5 Video About HTML5 Canvas

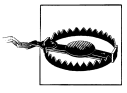
Chapter 1 gave an overview of the HTML Canvas and many of its applications, but wouldn't it have been cool if we had also included a video illustrating the Canvas in action? Well, now we know how to do that with the `<video>` element. Example 3-3 shows an HTML5 page that includes a clip from O'Reilly's Client-side Graphics with HTML5 Canvases demoing an Canvas adaptation of the arcade game Asteroids.

Example 3-3. Native HTML5 video content

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>HTML5 Video Illustrating HTML5 Canvas</title>
</head>
<body>
<h1>HTML5 Video Illustrating HTML5 Canvas</h1>
<p>Check out this excerpt from <a
href="http://bitly.com/html5canvasvideo"><em>Client-side Graphics
with HTML5 Canvases</em></a> showing the retro arcade game Asteroids implemented
using HTML5 Canvas.</p>
<video id="asteroids_video" width="480" height="270" controls="true">
<source src="video/html5_asteroids.mp4" type="video/mp4"/>
<source src="video/html5_asteroids.ogg" type="video/ogg"/>
<em>(Sorry, &lt;video&gt; element not supported in your
  browser/ereader, so you will not be able to watch this video.)</em>
</video>
</body>
</html>
```

Note the `width` and `height` values specified in order to set the dimensions of the video, and the addition of `controls` attribute to give the user access to the traditional video-

player buttons for controlling playback. For increased web browser compatibility, two video files are made available: one in MPEG-4 format and one in Ogg format.



If you're planning to embed `<video>` content in EPUB, however, at this time, I'd recommend limiting video files to MP4 format, which is currently supported by both iBooks and NOOK Color. Ogg files are not supported by either of these ereaders, and may interfere with video display.

Additionally, when embedding video in EPUB, you may want to optimize for file size, as large video files can quickly bloat your EPUB document—another good reason to stick with just one video format.

Take a look at the video clip in your browser. The code and video clips are available for download in GitHub.

EPUB 3 Media Overlays

The preceding examples are well suited to situations in which you want to intersperse audio and video throughout your content, but what if you want to incorporate more comprehensive functionality—say, provide an audio track for an entire book? For cases like these, EPUB 3 provides a specification for media overlay documents that allows you to sync audio with text:

> Books featuring synchronized audio narration are found in mainstream e-books, educational tools and e-books formatted for persons with print disabilities. In EPUB 3, these types of books are created by using Media Overlay Documents to describe the timing for the pre-recorded audio narration and how it relates to the EPUB Content Document markup. The file format for Media Overlays is defined as a subset of SMIL, a W3C recommendation for representing synchronized multimedia information in XML.
>
> The Media Overlays feature is designed to be transparent to EPUB Reading Systems that do snot support the feature. The inclusion of Media Overlays in an EPUB Publication has no impact on the ability of Media Overlay-unaware Reading Systems to render that Publication as a "regular" EPUB Publication.
>
> Although future versions of this specification may incorporate support for video media (e.g., synchronized text/sign-language books), this version supports only synchronizing audio media with the EPUB Content Document.*

As stated above media overlays are currently limited *only to audio* content (no support for syncing video to text at the present time), and furthermore, support for overlays is *optional*, so EPUB 3–compliant ereaders are allowed to ignore them.

To sync audio with text using media overlays, you make use of Media Overlay Documents, which are based on the Synchronized Multimedia Integration Language (SMIL)

* From 8 September 2011 draft of "EPUB Media Overlays 3.0" specification: *http://idpf.org/epub/30/spec/epub30-mediaoverlays.html#sec-overlays-introduction*

standard, an XML vocabulary for multimedia content. Media Overlay Documents are structured as a series of `<par>` elements that map text in the HTML content documents to the appropriate portion of corresponding audio files. For example:

```
<par id="hamlet_act_3_scene_1">
   <text src="act3_scene_1.xhtml#to_be_or_not_to_be"/>
   <audio src="to_be_or_not_to_be.mp3" clipBegin="0s clipEnd="45s"/>
</par>
```

Full details and sample Media Overlay Document structure can be found here in the spec. Details on how to incorporate Media Overlay documents into the EPUB 3 package document are also covered here.

HTML5 Audio/Video Compatibility in the Browser and Ereaders

HTML5 Audio/Video is currently supported across most major Web browsers (including Firefox, Safari, Google Chrome, and even [finally!] Internet Exporer), the specific audio/video formats supported vary from platform to platform, as the HTML5 spec itself is currently format-agnostic. Wikipedia has some nice tables tracking the current status of HTML5 audio and video support across the different browsers, but here's a quick summary of audio formats you should supply to ensure good compatibilty:

- **HTML5 Audio:** WAV, MP3, Ogg
- **HTML5 Video:** H.264 MPEG-4, Ogg

On EPUB ereaders, HTML5 audio/video support is more widespread than support for either Canvas or Geolocation, but is still limited to a few platforms. Here's a rundown of formats supported by HTML5 Audio/Video–compliant ereaders:

iBooks (v.1.1.1 and higher) for iPhone/iPod/iPad
: Video: MP4 (H.264)

  Audio: MP3, AAC, WAV

NOOK Color
: Video: "3gp, 3g2, mp4, m4v; MPEG-4 Simple Profile up to 854x480; H.263 up to 352x288; H.264 Baseline profile up to 854x480"†

  Audio: MP3, WAV, Ogg

Ibis Reader
: Video: MP4 (H.264)

† From the Nook Color FAQs: *http://www.barnesandnoble.com/u/nookcolor-support-beyond-ebooks/379002553/*

Adobe Digital Editions does not support HTML5 audio/video, but does support the embedding of Flash video in EPUB documents; see Liza Daly's tutorial, "Using Flash video in ePub," for details.

Bibliography/Additional Resources

If you're interested in learning more about HTML5 Audio and Video, you may be interested in some of these resources:

HTML5 Media by Shelley Powers (O'Reilly Media)
: A comprehensive look at incorporating audio/video content in HTML5 documents, converting media files to different formats, styling media with CSS, and advanced scripting with JavaScript,

Native Video in HTML5: An O'Reilly Breakdown by David Griffiths (O'Reilly Media)
: Nice series of video tutorials on HTML5 video

HTML5 Canvas by Steve Fulton and Jeff Fulton (O'Reilly Media)
: Chapter 6 of *HTML5 Canvas*, "Mixing HTML5 and Canvas," shows how to "draw" video content on the Canvas, and take advantage of the Canvas API to manipulate video in exciting ways.

"Jaraoke" by Randall A. Gordon
: A slick implementation of karaoke using HTML5 audio

jPlayer's "HTML5 <audio> and Audio() Support Tester"
: Test your Web browser's audio format support.

CHAPTER 4

Embedding HTML5 in EPUB

Thus far, we've built several HTML5 applications well suited to be embeded in ebooks. Now we'll take a look at how to structure and embed this HTML5 content in an EPUB.

An EPUB document (both EPUB 2.01 and EPUB 3.0) is a ZIP archive comprising five main components:

- A *mimetype* document containing the text `application/epub+zip`, which identifies the document as an EPUB
- A set of HTML content documents and referenced media files that contain all the book content
- A Package Document (often referred to as the OPF file), which contains a `<manifest>` that lists all the resources in the document and a `<spine>` that specifies the proper sequencing of the HTML content
- A *META-INF* directory containing a *container.xml* file that identifies the location of the Package Document and, optionally, an *encryption.xml* file that holds encryption info if your EPUB will contain DRM
- A Table of Contents document (in EPUB 2.01, a NCX file; in EPUB 3, a Navigation Document)

A detailed discussion/tutorial on constructing EPUB documents is beyond the scope of this book, but see "Additional EPUB Resources" on page 43 at the end of the chapter for some great articles that provide more guidance.

Embedding HTML5 content within an EPUB is done in the same fashion as any other HTML content; just add the file to your EPUB zip, and reference it in the OPF file. However, one important caveat is that many ereaders (most notably, iBooks) will not successfully parse HTML5 content unless the standard XHTML namespace is included on the `<html>` tag as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
Exciting HTML5 content goes here...
</html>
```

So be careful not to leave out the declaration highlighted in **bold** above. Also, any and all resources referenced in your HTML content need to be listed in the OPF `<manifest>`. Here's a sample manifest `<item>` for an external JavaScript file:

```
<item id="modernizr" href="modernizr-1.6.min.js" media-type="text/javascript"/>
```

Here's an `<item>` for an MP3 audio file:

```
<item id="rem_song" href="losing_my_religion.mp3" media-type="audio/mp3"/>
```

And here's an <item> for an MP4 video:

```
<item id="teen_vampires" href="new_moon.mp4" media-type="video/mp4"/>
```

The `media-type` attribute should contain the appropriate MIME MEDIA type for the file format; you can find a list of MIME types at *http://www.iana.org/assignments/media-types/index.html*

Alternatives to HTML5 and EPUB

As an open standard widely supported by nearly all major ereader devices (with one notable exception), EPUB is an excellent option for doing HTML5 ebook development, especially once the EPUB 3.0 specification is formally adopted. However, if you're interested in adding multimedia and interactivity to your ebook content, but don't want to go the HTML5/EPUB 3 route, here are some other options.

HTML5 and Mobi

If you're interested in making your ebook content available on Amazon's Kindle hardware and software platforms, EPUB is not an option. Kindle devices support only the proprietary Mobipocket (Mobi) format. Amazon provides a tool called Kindlegen for converting EPUB to Mobi, but Kindlegen and the Kindle's HTML and CSS support in Mobi is generally less robust than that in EPUB.

However, Kindle does now support embedded audio and video in Mobi content via the HTML5 `<audio>` and `<video>` tags. Per Version 1.8 of the Amazon Kindle Publishing Guidelines, videos in *.mp4*, *.mpg*, *.ps*, and *.ts* formats are accepted. Audio files must be in *.mp3* format. See pages 22–28 of the guidelines for more details on content and metadata requirements.

HTML5 and Ebook Apps

If instead of EPUB, you're interested in making ebook apps, you may want to look into PhoneGap. PhoneGap allows you to write your application using HTML5, CSS, and JavaScript, and then deploy as an app for multiple platforms, including Apple iOS, Android, BlackBerry, and WebOS. In addition to fully supporting HTML5, PhoneGap has APIs for accessing and controlling many common smartphone features, including the camera, accelerometer, and compass. PhoneGap makes use of Apple's Xcode infrastructure, and thus requires an Intel-based Mac. Check out their Getting Started guide for detailed information on how to get up and running.

You may also be interested in looking into the Baker ebook framework, a lighter-weight alternative to PhoneGap designed expressly for the release of interactive ebook content to Apple IOS devices. For more information, see Baker's tutorial on compiling an ebook app and releasing to Apple's App Store.

Additional EPUB Resources

If you're interested in learning more about EPUB, here are some additional resources to check out:

What is EPUB 3? by Matt Garrish
: A comprehensive overview of the new EPUB 3 specification and what it offers publishers

IDPF EPUB 3.0 Specification
: The official EPUB 3.0 specification. An absolute must-read if you're planning on creating EPUB 3.0 files

"Build a digital book with EPUB" by Liza Daly
: Comprehensive instructions on how to construct an EPUB file. This tutorial is EPUB 2–specific, but the majority of the content is still applicable under EPUB 3 (consult the 3.0 spec for more information on the Navigation Document, which replaces the NCX TOC)

"Creating epub files" by Bob Ducharme
: Additional discussion on some of the nuances of EPUB creation and validation. Pay extra attention to the discussion of how to properly zip up your EPUB archives, which is a bit less straightforward than you might expect.

EpubCheck
: The definitive tool for validating your EPUB files. A development build of epubcheck for validating EPUB 3.0 documents was recently released

About the Author

Sanders Kleinfeld has been employed at O'Reilly Media since 2004 and has held a variety of positions, including roles on O'Reilly's Production, Editorial, and Tools teams. Currently, he works as a Publishing Technologies Specialist, maintaining O'Reilly's XML-based toolchain for generating EPUB and Mobi formats of both frontlist and backlist titles. He also helps coordinate O'Reilly's digital distribution efforts to electronic sales channels, and is currently assisting in R&D efforts surrounding HTML5 and EPUB 3, helping to develop next-generation ebook content for O'Reilly and its publishing partners. In his spare time, Sanders loves to read, but primarily print books.