PRENTICE HALL

CORE

HTML5 CANVAS

Graphics, Animation, and Game Development

DAVID GEARY

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Core HTML5 Canvas

This page intentionally left blank

Core HTML5 Canvas

Graphics, Animation, and Game Development

David Geary

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382–3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Geary, David M.
Core HTML5 canvas : graphics, animation, and game development / David Geary.
p. cm.
Includes index.
ISBN 978-0-13-276161-1 (pbk. : alk. paper)
1. HTML (Document markup language) 2. Computer games—Programming. 3. Computer animation. I. Title.
QA76.76.H94C66 2012
006.6'6—dc23

2012006871

Copyright © 2012 David Geary

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-276161-1
ISBN-10: 0-13-276161-0
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Third printing, June 2014

Contents

This page intentionally left blank

Preface

In the summer of 2001, after 15 years of developing graphical user interfaces and graphics-intensive applications, I read a best-selling book about implementing web applications by someone I did not know—Jason Hunter—but whom, unbeknownst to me, would soon become a good friend on the No Fluff Just Stuff (NFJS) tour.

When I finished Jason's Servlets book,[1] I put it in my lap and stared out the window. After years of Smalltalk, C++, and Java, and after writing a passionate 1622 pages for *Graphic Java 2: Swing*,[2] I thought to myself, *am I really going to implement user interfaces with print statements that generate HTML?* Unfortunately, I was.

From then on, I soldiered on through what I consider the Dark Ages of software development. I was the second Apache Struts committer and I invented the Struts Template Library, which ultimately became the popular Tiles project. I spent more than six years on the JavaServer Faces (JSF) Expert Group, spoke about server-side Java at more than 120 NFJS symposiums and many other conferences, and coauthored a book on JSF.[3] I got excited about Google Web Toolkit and Ruby on Rails for a while, but in the end the Dark Ages was mostly concerned with the dull business of presenting forms to users on the client and processing them on the server, and I was never again able to capture that passion that I had for graphics and graphical user interfaces.

In the summer of 2010, with HTML5 beginning its inexorable rise in popularity, I came across an article about Canvas, and I knew salvation was nigh. I immediately dropped everything in my professional life and devoted myself fulltime to write the best Canvas book that I could. From then on, until the book was finalized in March 2012, I was entirely immersed in Canvas and in this book. It's by far the most fun I've ever had writing a book.

Canvas gives you all the graphics horsepower you need to implement everything from word processors to video games. And, although performance varies on specific platforms, in general, Canvas is fast, most notably on iOS5, which

1. *Java Servlet Programming*, 2001, by Jason Hunter with William Crawford, published by O'Reilly.
2. *Graphic Java 2, Volume 2, Swing*, 1999, by David Geary, published by Prentice Hall.
3. *Core JavaServer™ Faces, Third Edition*, 2010, by David Geary and Cay Horstmann, published by Prentice Hall.

hardware accelerates Canvas in Mobile Safari. Browser vendors have also done a great job adhering to the specification so that well-written Canvas applications run unmodified in any HTML5-compliant browser with only minor incompatibilities.

HTML5 is the Renaissance that comes after the Dark Ages of software development, and Canvas is arguably the most exciting aspect of HTML5. In this book I dive deeply into Canvas and related aspects of HTML5, such as the Animation Timing specification, to implement real-world applications that run across desktop browsers and mobile devices.

Reading This Book

I wrote this book so that in the Zen tradition you can read it without reading.

I write each chapter over the course of months, constantly iterating over material without ever writing a word. During that time I work on outlines, code listings, screenshots, tables, diagrams, itemized lists, notes, tips, and cautions. Those things, which I refer to as scaffolding, are the most important aspects of this book. The words, which I write only at the last possible moment after the scaffolding is complete, are meant to provide context and illustrate highlights of the surrounding scaffolding. Then I iterate over the words, eliminating as many of them as I can.

By focusing on scaffolding and being frugal with words, this book is easy to read without reading. You can skim the material, concentrating on the screenshots, code listings, diagrams, tables, and other scaffolding to learn a great deal of what you need to know on any given topic. Feel free to consider the words as second-class citizens, and, if you wish, consult them only as necessary.

An Overview of This Book

This book has two parts. The first part, which spans the first four chapters of the book and is nearly one half of the book, covers the Canvas API, showing you how to draw shapes and text into a canvas, and draw and manipulate images. The last seven chapters of the book show you how to use that API to implement animations and animated sprites, create physics simulations, detect collisions, and develop video games. The book ends with a chapter on implementing custom controls, such as progress bars, sliders, and image panners, and a chapter that shows you how to create Canvas-based mobile applications.

The first chapter—*Essentials*—introduces the `canvas` element and shows you how to use it in web applications. The chapter contains a short section on getting started with HTML5 development in general, briefly covering browsers, consoles, debuggers, profilers, and timelines. The chapter then shows you how to implement Canvas essentials: drawing into a canvas, saving and restoring Canvas parameters and the drawing surface itself, printing a canvas, and an introduction to offscreen canvases. The chapter concludes with a brief math primer covering basic algebra, trigonometry, vector mathematics, and deriving equations from units of measure.

The second chapter—*Drawing*—which is the longest chapter in the book, provides an in-depth examination of drawing with the Canvas API, showing you how to draw lines, arcs, curves, circles, rectangles, and arbitrary polygons in a canvas, and how to fill them with solid colors, gradients, and patterns. The chapter goes beyond the mere mechanics of drawing, however, by showing you how to implement useful, real-world examples of drawing with the Canvas API, such as drawing temporary rubber bands to dynamically create shapes, dragging shapes within a canvas, implementing a simple retained-mode graphics subsystem that keeps track of polygons in a canvas so users users can edit them, and using the clipping region to erase shapes without disturbing the Canvas background underneath.

The third chapter—*Text*—shows you how to draw and manipulate text in a canvas. You will see how to stroke and fill text, set font properties, and position text within a canvas. The chapter also shows you how to implement your own text controls in a canvas, complete with blinking text cursors and editable paragraphs.

The fourth chapter—*Images and Video*—focuses on images, image manipulation, and video processing. You'll see how to draw and scale images in a canvas, and you'll learn how to manipulate images by accessing the color components of each pixel. You will also see more uses for the clipping region and how to animate images. The chapter then addresses security and performance considerations, before ending with a section on video processing.

The fifth chapter—*Animation*—shows you how to implement smooth animations with a method named `requestAnimationFrame()` that's defined in a W3C specification titled *Timing control for script-based animations*. You will see how to calculate an animation's frame rate and how to schedule other activities, such as updating an animation's user interface at alternate frame rates. The chapter shows you how to restore the background during an animation with three different strategies and discusses the performance implications of each. The chapter also illustrates how to implement time-based motion, scroll an animation's background, use parallax to create the illusion of 3D, and detect and react to user gestures during an animation. The chapter concludes with a look at timed animations and the implementation of a simple animation timer, followed by a discussion of animation best practices.

The sixth chapter—*Sprites*—shows you how to implement sprites (animated objects) in JavaScript. Sprites have a visual representation, often an image, and you can move them around in a canvas and cycle through a set of images to animate them. Sprites are the fundamental building block upon which games are built.

The seventh chapter—*Physics*—shows you how to simulate physics in your animations, from modeling falling objects and projectile trajectories to swinging pendulums. The chapter also shows you how to warp both time and motion in your animations to simulate real-world movement, such as the acceleration experienced by a sprinter out of the blocks (ease-in effect) or the deceleration of a braking automobile (ease-out).

Another essential aspect of most games is collision detection, so the eighth chapter in the book—*Collision Detection*—is devoted to the science of detecting collisions between sprites. The chapter begins with simple collision detection using bounding boxes and circles, which is easy to implement but not very reliable. Because simple collision detection is not reliable under many circumstances, much of this chapter is devoted to the Separating Axis Theorem, which is one of the best ways to detect collisions between arbitrary polygons in both 2D and 3D; however, the theorem is not for the mathematically faint of heart, so this chapter goes to great lengths to present the theorem in layman terms.

The ninth chapter—*Game Development*—begins with the implementation of a simple but effective game engine that provides support for everything from drawing sprites and maintaining high scores to time-based motion and multitrack sound. The chapter then discusses two games. The first game is a simple Hello World type of game that illustrates how to use the game engine and provides a convenient starting point for a game. It also shows you how to implement common aspects of most games such as asset management, heads-up displays, and a user interface for high scores. The second game is an industrial-strength pinball game that draws on much of the previous material in the book and illustrates complex collision detection in a real-world game.

Many Canvas-based applications require custom controls, so the tenth chapter—*Custom Controls*—teaches you how to implement them. The chapter discusses implementing custom controls in general and then illustrates those techniques with four custom controls: a rounded rectangle, a progress bar, a slider, and an image panner.

The final chapter of this book—*Mobile*—focuses on implementing Canvas-based mobile applications. You'll see how to control the size of your application's viewport so that your application displays properly on mobile devices, and how to account for different screen sizes and orientations with CSS3 media queries.

You'll also see how to make your Canvas-based applications indistinguishable from native applications on iOS5 by making them run fullscreen and fitting them with desktop icons and startup screens. The chapter concludes with the implementation of a keyboard for iOS5 applications that do not receive text through a text field.

Prerequisites

To make effective use of this book you must have more than a passing familiarity with JavaScript, HTML, and CSS. I assume, for example, that you already know how to implement objects with JavaScript's prototypal inheritance, and that you are well versed in web application development in general.

This book also utilizes some mathematics that you may have learned a long time ago and forgotten, such as basic algebra and trigonometry, vector math, and deriving equations from units of measure. At the end of the first chapter you will find a short primer that covers all those topics.

The Book's Code

All the code in this book is copyrighted by the author and is available for use under the license distributed with the code. That license is a modified MIT license that lets you do anything you want with the code, including using it in software that you sell; however, you may not use the code to create educational material, such as books, instructional videos, or presentations. See the license that comes with the code for more details.

When implementing the examples, I made a conscious decision to keep comments in code listings to a bare minimum. Instead, I made the code itself as readable as possible; methods average about five lines of code so they are easy to understand.

I also adhered closely to Douglas Crockford's recommendations in his excellent book *JavaScript, The Good Parts*.[4] For example, all function-scoped variables are always declared at the top of the function, variables are declared on a line of their own, and I always use === and its ilk for equality testing.

Finally, all the code listings in this book are color coded. Function calls are displayed in `blue`, so they stand out from the rest of the listing. As you scan listings, pay particular attention to the `blue` function calls; after all, function calls are the verbs of JavaScript, and those verbs alone reveal most of what you need to know about the inner workings of any particular example.

4. *JavaScript*, *The Good Parts*, 2008, by Douglas Crockford, published by O'Reilly.

The Future of Canvas and This Book

The HTML5 APIs are constantly evolving, and much of that evolution consists of new features. The Canvas specification is no exception; in fact, this book was just days from going to the printer when the WHATWG Canvas specification was updated to include several new features:

- An `ellipse()` method that creates elliptical paths
- Two methods, `getLineDash()` and `setLineDash()`, and an attribute `lineDashOffset` used for drawing dashed lines
- An expanded `TextMetrics` object that lets you determine the exact bounding box for text
- A `Path` object
- A `CanvasDrawingStyles` object
- Extensive support for hit regions

At that time, no browsers supported the new features, so it was not yet possible to write code to test them.

Prior to the March 26, 2012 update to the specification, you could draw arcs and circles with Canvas, but there was no explicit provision for drawing ellipses. Now, in addition to arcs and circles, you can draw ellipses with the new `ellipse()` method of the Canvas 2d context. Likewise, the context now explicitly supports drawing dashed lines.

The `TextMetrics` object initially only reported one metric: the width of a string. However, with the March 26, 2012 update to the specification, you can now determine both the width and height of the rectangle taken up by a string in a canvas. That augmentation of the `TextMetrics` object will make it much easier, and more efficient, to implement Canvas-based text controls.

In addition to ellipses and an improved `TextMetrics` object, the updated specification has also added `Path` and `CanvasDrawingStyles` methods. Prior to the updated specification, there was no explicit mechanism for storing paths or drawing styles. Now, not only are there objects that represent those abstractions, but many of the Canvas 2d context methods have been duplicated to also take a `Path` object. For example, you stroke a context's path by invoking `context.stroke()`, which strokes the *current* path; however, the context now has a method `stroke(Path)` and that method strokes the path you send to the method instead of the context's current path. When you modify a path with `Path` methods such as `addText()`, you can specify a `CanvasDrawingStyle` object, which is used by the path, in this case to add text to the path.

The updated specification contains extensive support for hit regions. A hit region is defined by a path, and you can associate an optional mouse cursor and accessibility parameters, such as an Accessible Rich Internet Application (ARIA) role and a label, with a hit region. A single canvas can have multiple hit regions. Among other things, hit regions will make it easier and more efficient to implement collision detection and improve accessiblity.

Finally, both the WHATWG and W3C specifications have included two Canvas context methods for accessibility, so that applications can draw focus rings around the current path, letting users navigate with the keyboard in a Canvas. That functionality was not part of the March 26, 2012 update to the specification, and in fact, has been in the specification for some time; however, while the book was being written, no browser vendors supported the feature, so it is not covered in this book.

As the Canvas specification evolves and browser vendors implement new features, this book will be updated on a regular basis. In the meantime, you can read about new Canvas features and preview the coverage of those features in the next edition of this book, at corehtml5canvas.com.

The Companion Website

This book's companion website is http://corehtml5canvas.com, where you can download the book's code, run featured examples from the book, and find other HTML5 and Canvas resources.

This page intentionally left blank

Acknowledgments

Writing books is a team sport, and I was lucky to have great teammates for this book.

I'd like to start by thanking my longtime editor and good friend Greg Doench, who believed wholeheartedly in this book from the moment I proposed it and who gave me the latitude to write the book exactly as I wanted. Greg also oversaw the book from the moment of conception until, and after, it went to print. I couldn't ask for more.

I'm also fortunate that Greg comes with a great team of his own. Julie Nahil did a wonderful job of managing production and keeping everything on track, and Alina Kirsanova took my raw docbook XML and turned it into the beautiful color book you hold in your hands. Alina also did a superb job proofreading, weeding out small errors and inconsistencies.

Once again I was thrilled to have Mary Lou Nohr copy edit this book. Mary Lou is the only copy editor I've had in 15 years of writing books, and she not only makes each book better than I possibly could, but she continues to teach me the craft of writing.

Technical reviewers are vital to the success of any technical book, so I actively recruit reviewers who I think have an appropriate skill set to make significant contributions. For this book I was fortunate to land an excellent group of reviewers who helped me mold, shape, and polish the book's material. First, I'd like to thank Philip Taylor for being one of the most knowledgeable and thorough reviewers that I've ever had. Philip, who has implemented nearly 800 Canvas test cases—see http://philip.html5.org/tests/canvas/suite/tests—sent me pages of insightful comments for each chapter that only someone who knows the most intimate Canvas nuances could provide. Philip went way beyond the call of duty and single-handedly made this a much better book.

Next, I'd like to thank Scott Davis at thirstyhead.com, one of the foremost experts in HTML5 and mobile web application development. Scott has spoken at many conferences on HTML5 and mobile development, cofounded the HTML5 Denver Users Group, and taught mobile development to Yahoo! developers. Like Philip, Scott went way beyond the call of duty by offering excellent suggestions in many different areas of the book. I'm deeply indebted to Scott for delaying the publishing of this book for a full three months, while I entirely rewrote nearly a quarter of the book as the result of his scathing review. That rewrite took this book to the next level.

Ilmari Heikkinen, of Runfield fame (http://fhtr.org/runfield/runfield), provided some great insights for the Animation, Sprites, Physics, and Collision Detection chapters. Ted Neward, Dion Almaer, Ben Galbraith, Pratik Pratel, Doris Chen, Nate Schutta, and Brian Sam-Bodden also provided great review comments.

I'd also like to thank Mathias Bynens, the creator of jsperf.com, for giving me permission to use screenshots from that website.

I would like to acknowledge MJKRZAK for the sprite sheet used in the Physics chapter. That sprite sheet was downloaded from the public domain section of the People's Sprites website. I would also like to thank Ilmari Heikkinen for giving me permission to use his sky image for the parallax example in the Animation chapter. Some images in Sprites chapter are from the popular open source Replica Island game.

Finally, I'd like to thank Hiroko, Gaspé, and Tonka for enduring over the past year and a half while this book utterly consumed my life.

About the Author

David Geary is a prominent author, speaker, and consultant, who began implementing graphics-based applications and interfaces with C and Smalltalk in the 1980s. David taught C++ and Object-Oriented Software Development for eight years at Boeing, and was a software engineer at Sun Microsystems from 1994–1997. He is the author of eight Java books, including two best-selling books on the Java component frameworks, Swing and JavaServer Faces (JSF). David's *Graphic Java 2: Swing* is the all-time best-selling Swing book, and *Core JavaServer*™ *Faces*, which David wrote with Cay Horstmann, is the best-selling book on JSF.

David is a passionate and prolific public speaker who has spoken at hundreds of conferences world-wide. He spoke on the No Fluff Just Stuff tour for six years, speaking at over 120 symposiums, and he is a three-time JavaOne Rock Star.

In 2011, David and Scott Davis co-founded the HTML5 Denver Meetup group—www.meetup.com/HTML5-Denver-Users-Group—which had grown to over 500 members when this book was published in 2012.

David can be found on Twitter (@davidgeary) and at the companion website for this book, http://corehtml5canvas.com.

This page intentionally left blank

CHAPTER 1

Essentials

In 1939, Metro-Goldwyn-Mayer Studios released a film that, according to the American Library of Congress, was destined to become the most watched film in history. *The Wizard of Oz* is the story of a young girl named Dorothy and her dog Toto, who are transported by a violent tornado from Kansas in the central United States to the magical land of Oz.

The film begins in Kansas and is shot in a bland and dreary black-and-white. When Dorothy and Toto arrive in the land of Oz however, the film bursts into vibrant color, and the adventure begins.

For more than a decade, software developers have been implementing bland and dreary web applications that do little more than present bored-to-death users with a seemingly unending sequence of banal forms. Finally, HTML5 lets developers implement exciting desktop-like applications that run in the browser.

In this HTML5 land of Oz, we will use the magical canvas element to do amazing things in a browser. We will implement image panning, as shown in Figure 1.1; an interactive magnifying glass; a paint application that runs in any self-respecting browser and that also runs on an iPad; several animations and games, including an industrial-strength pinball game; image filters; and many other web applications that in another era were almost entirely the realm of Flash.

Let's get started.

1.1 The canvas Element

The canvas element is arguably the single most powerful HTML5 element, although, as you'll see shortly, its real power lies in the Canvas context, which

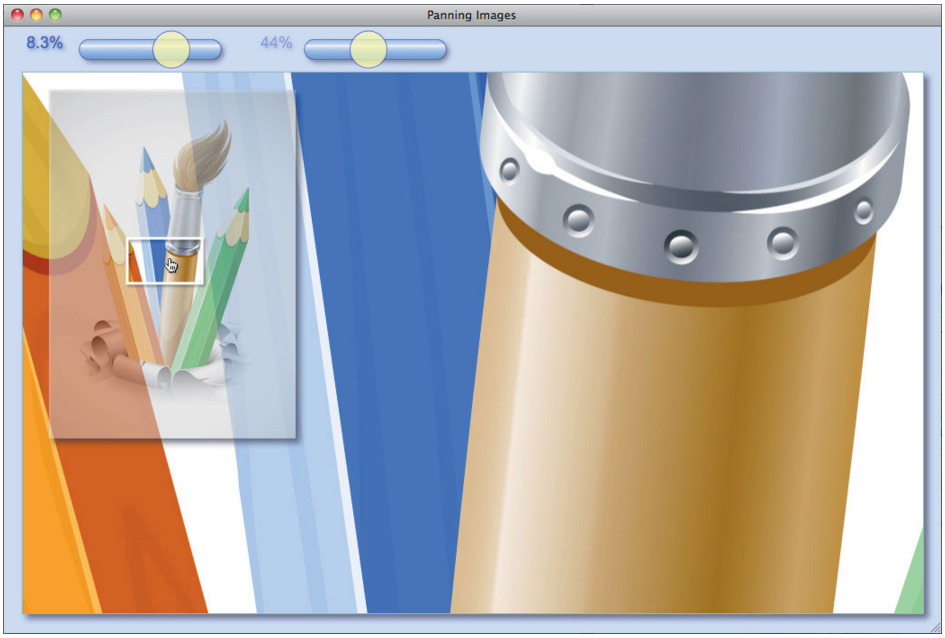


Figure 1.1 Canvas offers a powerful graphics API

you obtain from the canvas element itself. Figure 1.2 shows a simple use of the canvas element and its associated context.

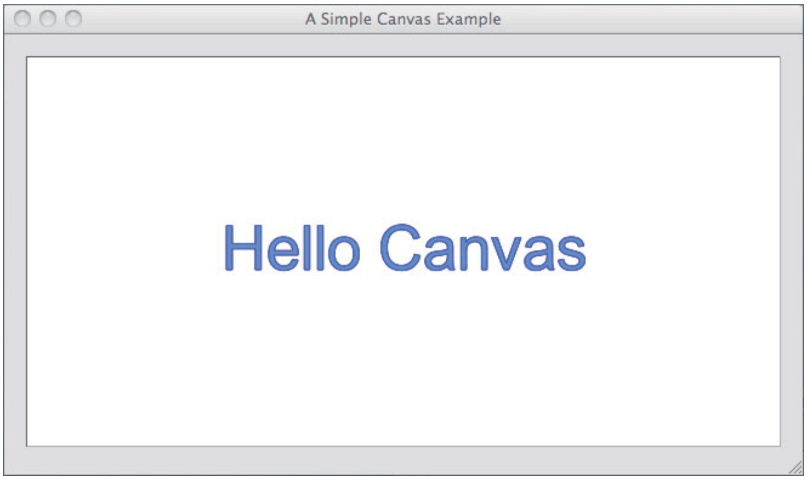


Figure 1.2 Hello canvas

The application shown in Figure 1.2 simply displays a string, approximately centered in the canvas itself. The HTML for that application is shown in Example 1.1.

The HTML in Example 1.1 uses a `canvas` element and specifies an identifier for the element and the element's width and height. Notice the text in the body of the `canvas` element. That text is known as the *fallback content*, which the browser displays only if it does not support the `canvas` element.

Besides those two elements, the HTML in Example 1.1 uses CSS to set the application's background color and some attributes for the `canvas` element itself. By default, a `canvas` element's background color matches the background color of its parent element, so the CSS sets the `canvas` element's background color to opaque white to set it apart from the application's light gray background.

The HTML is straightforward and not very interesting. As is typically the case for Canvas-based applications, the interesting part of the application is its JavaScript. The JavaScript code for the application shown in Figure 1.2 is listed in Example 1.2.

Example 1.1 `example.html`

```
<!DOCTYPE html>
<html>
   <head>
     <title>A Simple Canvas Example</title>

      <style>
         body {
            background: #dddddd;
         }
         #canvas {
            margin: 10px;
            padding: 10px;
            background: #ffffff;
            border: thin inset #aaaaaa;
         }
      </style>
   </head>

  <body>
    <canvas id='canvas' width='600' height='300'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

Example 1.2 example.js

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

context.font = '38pt Arial';
context.fillStyle = 'cornflowerblue';
context.strokeStyle = 'blue';

context.fillText('Hello Canvas', canvas.width/2 - 150,
                                 canvas.height/2 + 15);

context.strokeText('Hello Canvas', canvas.width/2 - 150,
                                   canvas.height/2 + 15 );
```

The JavaScript in **Example 1.2** employs a recipe that you will use in your Canvas-based applications:

1. Use `document.getElementById()` to get a reference to a canvas.
2. Call `getContext('2d')` on the canvas to get the graphics context (note: the 'd' in '2d' *must* be lowercase).
3. Use the context to draw in the canvas.

After obtaining a reference to the canvas's context, the JavaScript sets the context's `font`, `fillStyle`, and `strokeStyle` attributes and fills and strokes the text that you see in **Figure 1.2**. The `fillText()` method fills the characters of the text using `fillStyle`, and `strokeText()` strokes the outline of the characters with `strokeStyle`. The `fillStyle` and `strokeStyle` attributes can be a CSS color, a gradient, or a pattern. We briefly discuss those attributes in Section 1.2.1, "The 2d Context," on p. 9 and take a more in-depth look at both the attributes and methods in Chapter 2.

The `fillText()` and `strokeText()` methods both take three arguments: the text and an `(x, y)` location within the canvas to display the text. The JavaScript shown in **Example 1.2** approximately centers the text with constant values, which is not a good general solution for centering text in a canvas. In Chapter 3, we will look at a better way to center text.

CAUTION: The suffix px is not valid for canvas width and height

Although it's widely permitted by browsers that support Canvas, the px suffix for the canvas `width` and `height` attributes is not technically allowed by the Canvas specification. The values for those attributes, according to the specification, can only be non-negative integers.

NOTE: The default canvas size is 300 × 150 screen pixels

By default, the browser creates `canvas` elements with a width of 300 pixels and a height of 150 pixels. You can change the size of a `canvas` element by specifying the `width` and `height` attributes.

You can also change the size of a `canvas` element with CSS attributes; however, as you will see in the next section, changing the width and height of a `canvas` element may have unwanted consequences.

1.1.1 Canvas Element Size vs. Drawing Surface Size

The application in the preceding section sets the size of the `canvas` element by setting the element's `width` and `height` attributes. You can also use CSS to set the size of a `canvas` element, as shown in Example 1.3; however, using CSS to size a `canvas` element is not the same as setting the element's `width` and `height` attributes.

Example 1.3 Setting element size and drawing surface size to different values

```
<!DOCTYPE html>
   <head>
     <title>Canvas element size: 600 x 300,
            Canvas drawing surface size: 300 x 150</title>
      <style>
         body {
            background: #dddddd;
         }
         #canvas {
            margin: 20px;
            padding: 20px;
            background: #ffffff;
            border: thin inset #aaaaaa;
            width: 600px;
            height: 300px;
         }
      </style>
   </head>

  <body>
    <canvas id='canvas'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

The difference between using CSS and setting canvas element attributes lies in the fact that *a canvas actually has two sizes*: the size of the element itself and the size of the element's drawing surface.

When you set the element's width and height attributes, you set *both* the element's size and the size of the element's drawing surface; however, when you use CSS to size a canvas element, you set *only* the element's size and not the drawing surface.

By default, both the canvas element's size and the size of its drawing surface is 300 screen pixels wide and 150 screen pixels high. In the listing shown in Example 1.3, which uses CSS to set the canvas element's size, the size of the element is 600 pixels wide and 300 pixels high, but *the size of the drawing surface remains unchanged* at the default value of 300 pixels × 150 pixels.

And here is where things get interesting because when a canvas element's size does not match the size of its drawing surface, *the browser scales the drawing surface to fit the element*. That effect is illustrated in Figure 1.3.

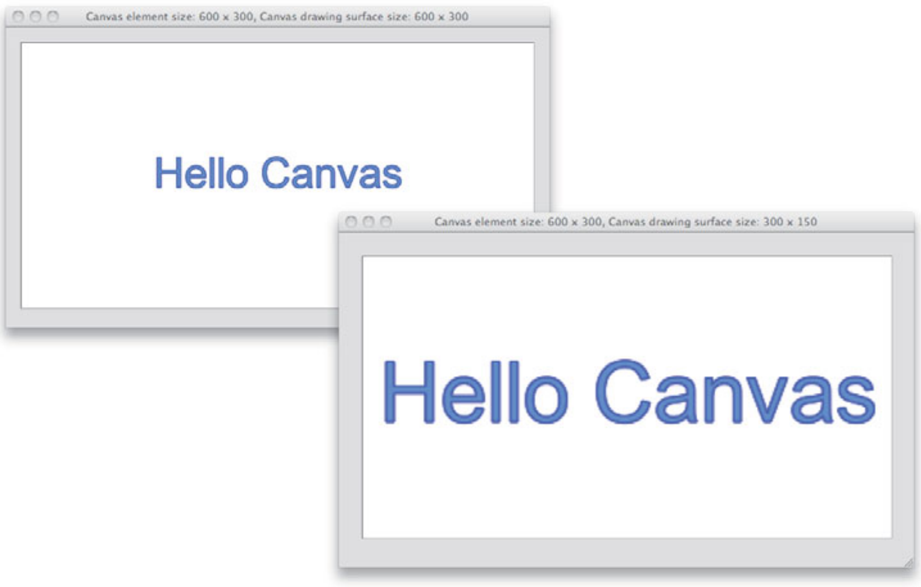

Canvas element size: 600 x 300, Canvas drawing surface size: 600 x 300
Hello Canvas
Canvas element size: 600 x 300, Canvas drawing surface size: 300 x 150
Hello Canvas

Figure 1.3 *Top*: element and coordinate system = 600 × 300; *bottom*: element = 600 × 300, coordinate system = 300 × 150

The application shown at the top of Figure 1.3 is the application that we discussed in the preceding section. It sets the canvas element's size with the element's width and `height` attributes, setting both the element's size and the size of the drawing surface to 600 pixels × 300 pixels.

The application shown at the bottom of Figure 1.3 is the application whose HTML is shown in Example 1.3. That application is identical to the application in the preceding section, except that it uses CSS to size the `canvas` element (and has a different title in the window's title bar).

Because the application shown in the bottom screenshot in Figure 1.3 uses CSS to size the `canvas` element and does not set the element's `width` or `height` attributes, the browser scales the drawing surface from 300 pixels × 150 pixels to 600 pixels × 300 pixels.

CAUTION: The browser may automatically scale your canvas

It's a good idea to use the `canvas` element's `width` and `height` attributes to size the element, instead of using CSS. If you use CSS to size the element without also specifying the `width` and `height` attributes of the `canvas` element, the element size will not match the canvas's drawing surface size, and the browser will scale the latter to fit the former, most likely resulting in surprising and unwanted effects.

1.1.2 The Canvas API

The `canvas` element does not provide much of an API; in fact, that API offers only two attributes and three methods that are summarized in Table 1.1 and Table 1.2.

Table 1.1 `canvas` attributes

Attribute	Description	Type	Allowed Values	Default
`width`	The width of the canvas's *drawing surface*. By default, the browser makes the `canvas` element the same size as its drawing surface; however, if you override the element size with CSS, then the browser will *scale* the drawing surface to fit the element.	non-negative integer	Any valid non-negative integer. You may add a plus sign or whitespace at the beginning, but technically, you cannot add a `px` suffix.	300

(Continues)

Table 1.1 *(Continued)*

Attribute	Description	Type	Allowed Values	Default
`height`	The height of the canvas's drawing surface. The browser may scale the drawing surface to fit the `canvas` element size. See the `width` attribute for more information.	non-negative integer	Any valid non-negative integer. You may add a plus sign or whitespace at the beginning, but technically, you cannot add a `px` suffix.	150

Table 1.2 `canvas` methods

Method	Description
`getContext()`	Returns the graphics context associated with the canvas. Each canvas has one context, and each context is associated with one canvas.
`toDataURL(type, quality)`	Returns a data URL that you can assign to the `src` property of an `img` element. The first argument specifies the type of image, such as `image/jpeg`, or `image/png`; the latter is the default if you don't specify the first argument. The second argument, which must be a `double` value from `0` to `1.0`, specifies a quality level for JPEG images.
`toBlob(callback, type, args...)`	Creates a `Blob` that represents a file containing the canvas's image. The first argument to the method is a function that the browser invokes with a reference to the blob. The second argument specifies the type of image, such as `image/png`, which is the default value. The final arguments represent a quality level from `0.0` to `1.0` inclusive, for JPEG images. Other arguments will most likely be added to this method in the future to more carefully control image characteristics.

1.2 Canvas Contexts

The `canvas` element merely serves as a container for a context. The context provides all the graphics horsepower. Although this book focuses exclusively on the 2d context, the Canvas specification embraces other types of contexts as well; for example, a 3d context specification is already well underway. This section looks at the attributes of the 2d context, with a brief nod to the 3d context.

1.2.1 The 2d Context

In your JavaScript code, you will find little use for the `canvas` element itself, other than occasionally using it to obtain the canvas width or height or a data URL, as discussed in the preceding section. Additionally, you will use the `canvas` element to obtain a reference to the canvas's context, which provides a capable API for drawing shapes and text, displaying and manipulating images, etc. Indeed, for the rest of this book our focus will mainly be on the 2d context.

Table 1.3 lists all of the 2d context attributes. Other than the `canvas` attribute, which gives you a reference to the canvas itself, all of the 2d context attributes pertain to drawing operations.

Table 1.3 `CanvasRenderingContext2D` attributes

Attribute	Brief Description
`canvas`	Refers to the context's canvas. The most common use of the `canvas` attribute is to access the width and height of the canvas: `context.canvas.width` and `context.canvas.height`, respectively.
`fillStyle`	Specifies a color, gradient, or pattern that the context subsequently uses to fill shapes.
`font`	Specifies the font that the context uses when you call `fillText()` or `strokeText()`.
`globalAlpha`	Is the global alpha setting, which must be a number between `0` (fully transparent), and `1.0` (fully opaque). The browser multiplies the alpha value of every pixel you draw by the `globalAlpha` property, including when you draw images.
`globalComposite-Operation`	Determines how the browser draws one thing over another. See Section 2.14 for valid values.
`lineCap`	Specifies how the browser draws the endpoints of a line. You can specify one of the following three values: `butt`, `round`, and `square`. The default value is `butt`.
`lineWidth`	Determines the width, in screen pixels, of lines that you draw in a canvas. The value must be a non-negative, non-infinite `double` value. The default is `1.0`.
`lineJoin`	Specifies how lines are joined when their endpoints meet. Valid values are: `bevel`, `round`, and `miter`. The default value is `miter`.

(Continues)

Table 1.3 *(Continued)*

Attribute	Brief Description
`miterLimit`	Specifies how to draw a miter line join. See Section 2.8.7 for details about this property.
`shadowBlur`	Determines how the browser spreads out shadow; the higher the number, the more spread out the shadows. The `shadowBlur` value is not a pixel value, but a value used in a Gaussian blur equation. The value must be a positive, non-infinite `double` value. The default value is `0`.
`shadowColor`	Specifies the color the browser uses to draw shadows. The value for this property is often specified as partially transparent to let the background show through.
`shadowOffsetX`	Specifies the horizontal offset, in screen pixels, for shadows.
`shadowOffsetY`	Specifies the vertical offset, in screen pixels, for shadows.
`strokeStyle`	Specifies the style used to stroke paths. This value can be a color, gradient, or pattern.
`textAlign`	Determines horizontal placement of text that you draw with `fillText()` or `strokeText()`.
`textBaseline`	Determines vertical placement of text that you draw with `fillText()` or `strokeText()`.

The table gives you an overview of all the 2d context attributes. In Chapter 2, we examine all those attributes on a case-by-case basis.

NOTE: You can extend the 2d context's capabilities

The context associated with each canvas is a powerful graphics engine that supports features such as gradients, image compositing, and animation, but it does have limitations; for example, the context does not provide a method for drawing dashed lines. Because JavaScript is a dynamic language, however, you can add new methods or augment existing methods of the context. See Section 2.8.6, "Drawing Dashed Lines by Extending `CanvasRenderingContext2D`," on p. 118 for more information.

1.2.1.1 The WebGL 3d Context

The Canvas 2d context has a 3d counterpart, known as WebGL, that closely conforms to the OpenGL ES 2.0 API. You can find the WebGL specification, which is maintained by the Khronos Group, at http://www.khronos.org/registry/webgl/specs/latest/.

At the time this book was written, browser vendors were just beginning to provide support for WebGL, and there are still some notable platforms, such as iOS4 and IE10, that do not provide support. Nonetheless, a 3d Canvas context is an exciting development that will open the door to all sorts of bleeding edge applications.

1.2.2 Saving and Restoring Canvas State

In Section 1.2.1, "The 2d Context," on p. 9 we discussed all of the attributes of the Canvas context. You will often set those attributes for drawing operations. Much of the time you will want to *temporarily* set those attributes; for example, you may draw a grid with thin lines in the background and subsequently draw on top of the grid with thicker lines. In that case you would temporarily set the `lineWidth` attribute while you draw the grid.

The Canvas API provides two methods, named `save()` and `restore()`, for saving and restoring all the canvas context's attributes. You use those methods like this:

```
function drawGrid(strokeStyle, fillStyle) {
   controlContext.save(); // Save the context on a stack

   controlContext.fillStyle = fillStyle;
   controlContext.strokeStyle = strokeStyle;

   // Draw the grid...

   controlContext.restore(); // Restore the context from the stack
}
```

The `save()` and `restore()` methods may not seem like a big deal, but after using Canvas for any length of time you will find them indispensable. Those two methods are summarized in Table 1.4.

NOTE: You can nest calls to save() and restore()

The context's `save()` method places the current state of the context onto a stack. The corresponding call to `restore()` pops the state from the stack and restores the context's state accordingly. That means you can nest calls to `save()/restore()`.

Table 1.4 `CanvasRenderingContext2D` state methods

Method	Description
`save()`	Pushes the current state of the canvas onto a stack of canvas states. Canvas state includes the current transformation and clipping region and all attributes of the canvas's context, including `strokeStyle`, `fillStyle`, `globalCompositeOperation`, etc. The canvas state does not include the current path or bitmap. You can only reset the path by calling `beginPath()`, and the bitmap is a property of the canvas, not the context. Note that although the bitmap is a property of the canvas, you access the bitmap through the context (via the context's `getImageData()` method).
`restore()`	Pops the top entry off the stack of canvas states. The state that resides at the top of the stack, after the pop occurs, becomes the current state, and the browser must set the canvas state accordingly. Therefore, any changes that you make to the canvas state between `save()` and `restore()` method calls persist only until you invoke the `restore()` method.

NOTE: Saving and restoring the drawing surface

This section shows you how to save and restore context state. It's also beneficial to be able to save and restore the drawing surface itself, which we discuss in Section 1.7, "Saving and Restoring the Drawing Surface," on p. 33.

1.3 Canonical Examples in This Book

Many of the examples in this book use the following canonical form:

```
<!-- example.html -->

<!DOCTYPE html>
<html>
   <head>
     <title>Canonical Canvas used in this book</title>

      <style>
         ...
         #canvas {
            ...
         }
      </style>
   </head>
```

```
      <body>
         <canvas id='canvas' width='600' height='300'>
            Canvas not supported
         </canvas>

         <script src='example.js'></script>
      </body>
   </html>

   // example.js

   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d');

   // Use the context...
```

The preceding example has one canvas whose ID is `canvas`, and it uses one JavaScript file named `example.js`. That JavaScript file has two variables, one for the canvas and another for the canvas's context. The preceding example uses `document.getElementById()` to obtain a reference to the canvas and gets a reference to the canvas's context.

Most applications in this book that adhere to the preceding canonical form omit the HTML listings in the interests of brevity. Likewise, for inline code listings, meaning listings like the preceding listing that do not have an Example heading, you will often see the variables `canvas` and `context` with no code showing their initialization.

Finally, again in the interests of brevity, not every example in the book is fully listed. Often examples in the book build upon one other, and when they do, you will often see the full listing for the last example and partial listings for the other related examples.

NOTE: A word about User Agents

The Canvas specification refers to the implementor of the `canvas` element as a *User Agent*, which is often abbreviated to UA. The specification uses that term instead of the word *browser* because `canvas` elements can be implemented by any piece of software, not just browsers.

This book refers to the implementor of the `canvas` element as a browser because the term User Agent, or worse, the abbreviation UA, can be foreign and confusing to readers.

NOTE: URLs referenced in this book

In this book you will occasionally find references to URLs for further reading. Sometimes, if they are readable and not too long, those URLs will be the actual URLs. For unwieldy URLs, this book refers to shortened URLs that may be difficult to remember but are easy to type.

1.4 Getting Started

This section gives you a brief overview of your development environment, from the browsers in which your application will run to the development tools, such as profilers and timelines, that you will use during development. Feel free to skim this section and use it as a reference as necessary.

1.4.1 Specifications

Three specifications are pertinent to this book:

- HTML5 Canvas
- Timing control for script-based animations
- HTML5 video and audio

For historical reasons, there are actually two Canvas specifications that are nearly identical. One of those specifications is maintained by the W3C and can be found at http://dev.w3.org/html5/spec; the other specification is maintained by the WHATWG and can be found at http://bit.ly/qXWjOl. Furthermore, whereas the Canvas context is included in the WHATWG's specification, the WC3 has a separate specification for the context, at http://dev.w3.org/html5/2dcontext.

For a long time, people used `window.setInterval()` or `window.setTimeout()` for web-based animations; however, as you will see in Chapter 5, those methods are not suitable for performance-critical animations. Instead, you should use `window.requestAnimationFrame()`, which is defined in a specification of its own named *Timing control for script-based animations*. You can find that specification at http://www.w3.org/TR/animation-timing.

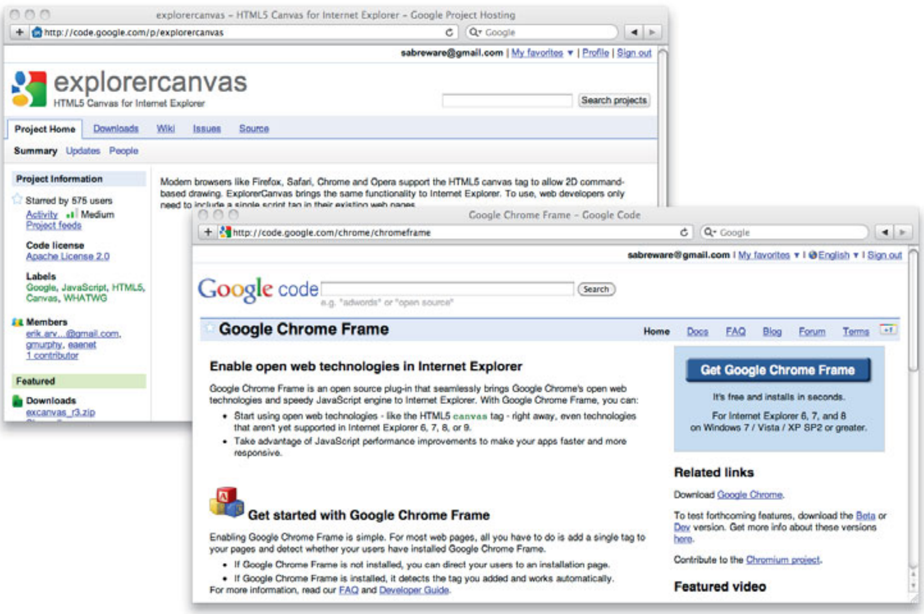
Finally, this book shows you how to incorporate HTML5 video and audio into your Canvas-based applications. HTML5 video and audio are covered in the same specification, which you can find at http://www.w3.org/TR/html5/video.html.

1.4.2 Browsers

At the time this book went to press in early 2012, all five major browsers—Chrome, Internet Explorer, Firefox, Opera, and Safari—provided extensive support for HTML5 Canvas. Although there are some minor incompatibilities that mostly stem from different interpretations of the Canvas specification—for example, see Section 2.14.1, "The Compositing Controversy," on p. 186, which explains incompatibilities for compositing—browser vendors have done an admirable job of both adhering to the specification and providing implementations that perform well.

Chrome, Firefox, Opera, and Safari have all had HTML5 support for some time. Microsoft's Internet Explorer was a bit late to the game and did not provide extensive support for HTML 5 until IE9. However, Microsoft has done a phenomenal job with Canvas in IE9 and IE10; in fact, as this book went to press, those two browsers had the fastest Canvas implementation from among the five major browsers.

If you are implementing a Canvas-based application and you must support IE6, IE7, or IE8, you have two choices, depicted in Figure 1.4: explorercanvas, which


explorercanvas – HTML5 Canvas for Internet Explorer – Google Project Hosting
http://code.google.com/p/explorercanvas
Google
sabreware@gmail.com | My favorites ▾ | Profile | Sign out
explorercanvas
HTML5 Canvas for Internet Explorer
Search projects
Project Home
Downloads
Wiki
Issues
Source
Summary
Updates
People

Project Information
Starred by 575 users
Activity Medium
Project feeds

Code license
Apache License 2.0

Labels
Google, JavaScript, HTML5, Canvas, WHATWG

Members
erik.arv...@gmail.com, gmurphy, eaenet
1 contributor

Featured
Downloads
excanvas_r3.zip

Modern browsers like Firefox, Safari, Chrome and Opera support the HTML5 canvas tag to allow 2D command-based drawing. ExplorerCanvas brings the same functionality to Internet Explorer. To use, web developers only need to include a single script tag in their existing web pages

Google Chrome Frame – Google Code
http://code.google.com/chrome/chromeframe
Google
sabreware@gmail.com | My favorites ▾ | English ▾ | Sign out
Google code
Search
e.g. "adwords" or "open source"
Google Chrome Frame
Home
Docs
FAQ
Blog
Forum
Terms
Get Google Chrome Frame

Enable open web technologies in Internet Explorer

Google Chrome Frame is an open source plug-in that seamlessly brings Google Chrome's open web technologies and speedy JavaScript engine to Internet Explorer. With Google Chrome Frame, you can:

- Start using open web technologies - like the HTML5 `canvas` tag - right away, even technologies that aren't yet supported in Internet Explorer 6, 7, 8, or 9.
- Take advantage of JavaScript performance improvements to make your apps faster and more responsive.

It's free and installs in seconds.

For Internet Explorer 6, 7, and 8 on Windows 7 / Vista / XP SP2 or greater.

Get started with Google Chrome Frame

Enabling Google Chrome Frame is simple. For most web pages, all you have to do is add a single tag to your pages and detect whether your users have installed Google Chrome Frame.

- If Google Chrome Frame is not installed, you can direct your users to an installation page.
- If Google Chrome Frame is installed, it detects the tag you added and works automatically.

For more information, read our FAQ and Developer Guide.

Related links

Download Google Chrome.

To test forthcoming features, download the Beta or Dev version. Get more info about these versions here.

Contribute to the Chromium project.

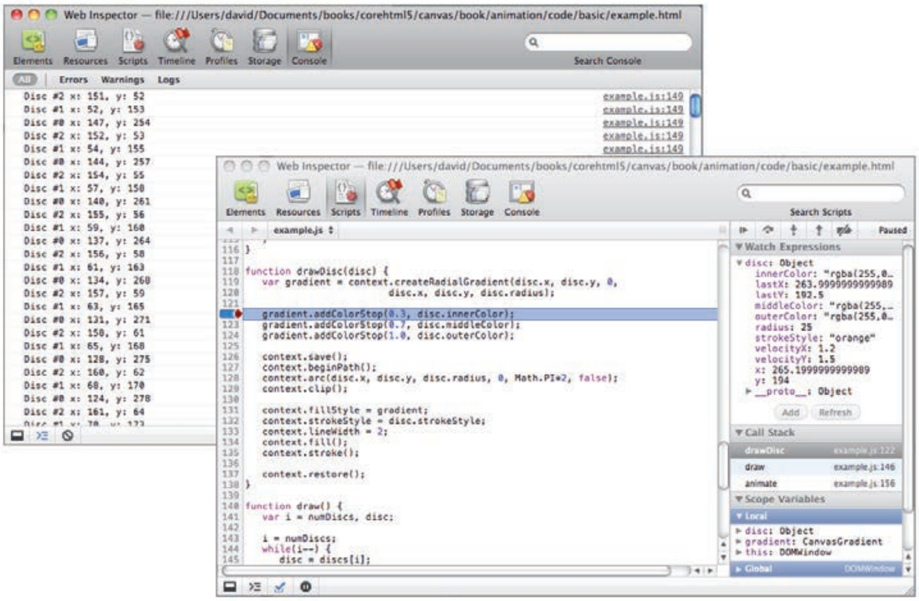
Featured video

Figure 1.4 explorercanvas and Google Chrome Frame for IE6/7/8, from Google adds Canvas support to those older versions of Internet Explorer, and Google Chrome Frame, which replaces the IE engine with the Google Chrome engine. Both explorercanvas and Google Chrome Frame are from Google.

1.4.3 Consoles and Debuggers

All the major browsers that support HTML5 give you access to a console and a debugger. In fact, because browser vendors often borrow ideas from each other, the consoles and debuggers provided by non-WebKit-based browsers—Firefox, Opera, and IE—are all pretty similar.

Figure 1.5 shows the console and debugger for Safari.



```
Disc #2 x: 151, y: 52                                  example.js:149
Disc #1 x: 52, y: 153                                  example.js:149
Disc #0 x: 147, y: 254                                 example.js:149
Disc #2 x: 152, y: 53                                  example.js:149
Disc #1 x: 54, y: 155                                  example.js:149
Disc #0 x: 144, y: 257
Disc #2 x: 154, y: 55
Disc #1 x: 57, y: 158
Disc #0 x: 140, y: 261
Disc #2 x: 155, y: 56
Disc #1 x: 59, y: 160
Disc #0 x: 137, y: 264
Disc #2 x: 156, y: 58
Disc #1 x: 61, y: 163
Disc #0 x: 134, y: 268
Disc #2 x: 157, y: 59
Disc #1 x: 63, y: 165
Disc #0 x: 131, y: 271
Disc #2 x: 158, y: 61
Disc #1 x: 65, y: 168
Disc #0 x: 128, y: 275
Disc #2 x: 160, y: 62
Disc #1 x: 68, y: 170
Disc #0 x: 124, y: 278
Disc #2 x: 161, y: 64
Disc #1 x: 70, y: 173
```

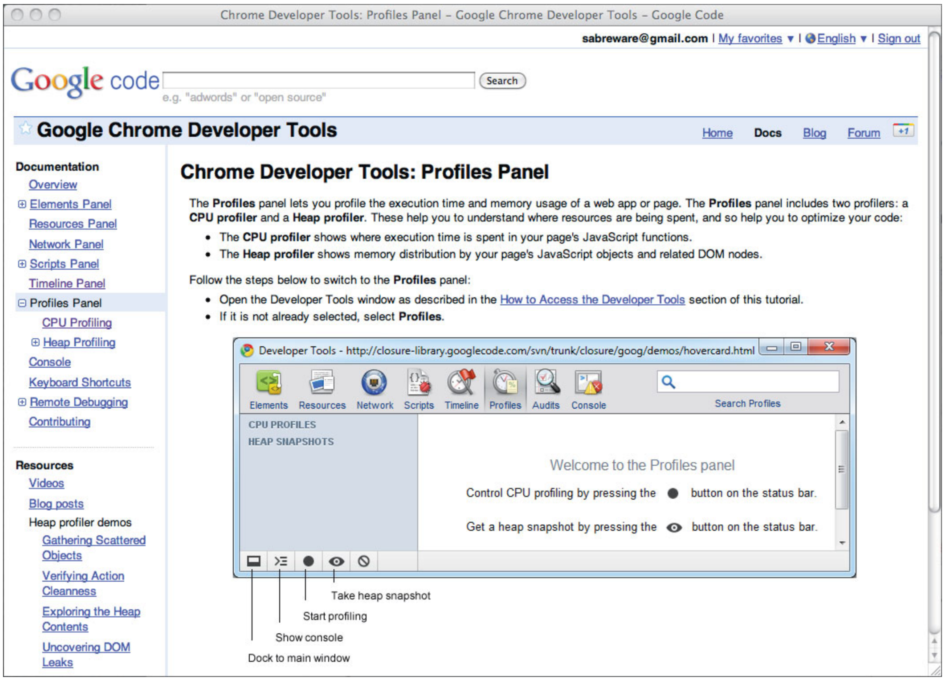
```
}

function drawDisc(disc) {
   var gradient = context.createRadialGradient(disc.x, disc.y, 0,
                        disc.x, disc.y, disc.radius);

   gradient.addColorStop(0.3, disc.innerColor);
   gradient.addColorStop(0.7, disc.middleColor);
   gradient.addColorStop(1.0, disc.outerColor);

   context.save();
   context.beginPath();
   context.arc(disc.x, disc.y, disc.radius, 0, Math.PI*2, false);
   context.clip();

   context.fillStyle = gradient;
   context.strokeStyle = disc.strokeStyle;
   context.lineWidth = 2;
   context.fill();
   context.stroke();

   context.restore();
}

function draw() {
   var i = numDiscs, disc;

   i = numDiscs;
   while(i--) {
      disc = discs[i];
```

```
▼ disc: Object
    innerColor: "rgba(255,0…
    lastX: 263.9999999999989
    lastY: 192.5
    middleColor: "rgba(255,…
    outerColor: "rgba(255,0…
    radius: 25
    strokeStyle: "orange"
    velocityX: 1.2
    velocityY: 1.5
    x: 265.1999999999989
    y: 194
  ▶ __proto__: Object

drawDisc        example.js:122
draw            example.js:146
animate         example.js:156

▶ disc: Object
▶ gradient: CanvasGradient
▶ this: DOMWindow
▶ Global        DOMWindow
```

Figure 1.5 The Safari console and debugger

You can write to the console with the `console.log()` method. Just pass that method a string, and it will appear in the console. The debugger is standard debugger fare; you can set breakpoints, watch expressions, examine variables and the call stack, and so on.

A full treatment of the developer tools for various browsers is beyond the scope of this book. For more information about developer tools for Chrome, take a look at the Chrome Developer Tools documentation, shown in **Figure 1.6**. Similar documentation is available for other browsers.



Chrome Developer Tools: Profiles Panel – Google Chrome Developer Tools – Google Code

sabreware@gmail.com | My favorites ▾ | English ▾ | Sign out

Google code [Search]
e.g. "adwords" or "open source"

Google Chrome Developer Tools Home **Docs** Blog Forum

Documentation
- Overview
- ⊕ Elements Panel
- Resources Panel
- Network Panel
- ⊕ Scripts Panel
- Timeline Panel
- ⊟ Profiles Panel
  - CPU Profiling
  - ⊕ Heap Profiling
- Console
- Keyboard Shortcuts
- ⊕ Remote Debugging
- Contributing

Resources
- Videos
- Blog posts
- Heap profiler demos
  - Gathering Scattered Objects
  - Verifying Action Cleanness
  - Exploring the Heap Contents
  - Uncovering DOM Leaks

Chrome Developer Tools: Profiles Panel

The **Profiles** panel lets you profile the execution time and memory usage of a web app or page. The **Profiles** panel includes two profilers: a **CPU profiler** and a **Heap profiler**. These help you to understand where resources are being spent, and so help you to optimize your code:

- The **CPU profiler** shows where execution time is spent in your page's JavaScript functions.
- The **Heap profiler** shows memory distribution by your page's JavaScript objects and related DOM nodes.

Follow the steps below to switch to the **Profiles** panel:

- Open the Developer Tools window as described in the How to Access the Developer Tools section of this tutorial.
- If it is not already selected, select **Profiles**.

Developer Tools - http://closure-library.googlecode.com/svn/trunk/closure/goog/demos/hovercard.html

Welcome to the Profiles panel

Control CPU profiling by pressing the ● button on the status bar.

Get a heap snapshot by pressing the 👁 button on the status bar.

Figure 1.6 The Chrome Developer Tools documentation

TIP: Start and stop the profiler programmatically

As you can see from **Figure 1.6**, you can start profiling in WebKit-based browsers by clicking the filled circle at the bottom of the profiler window.

Controlling the profiler by clicking buttons, however, is often insufficient; for example, you may want to start and stop profiling at specific lines of code. In WebKit-based browsers, you can do that with two methods: `console.profile()` and `console.profileEnd()`. You use them like this:

```
console.profile('Core HTML5 Animation,
                erasing the background');
//...

console.profileEnd();
```

1.4.4 Performance

Most of the time the applications that you implement with Canvas will perform admirably; however, if you are implementing animations or games or if you are implementing Canvas-based applications for mobile devices, you may need to make performance optimizations.

In this section we briefly look at the tools you have at your disposal for discovering performance bottlenecks in your code. To illustrate the use of those tools, we refer to the application shown in Figure 1.7. That animation, which is discussed in Chapter 5, simultaneously animates three filled circles.

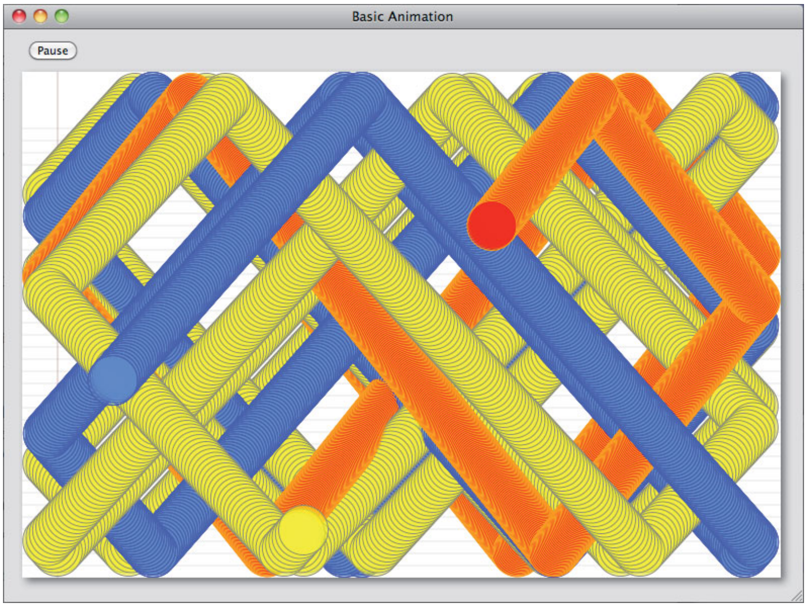


Figure 1.7 An animation from Chapter 5

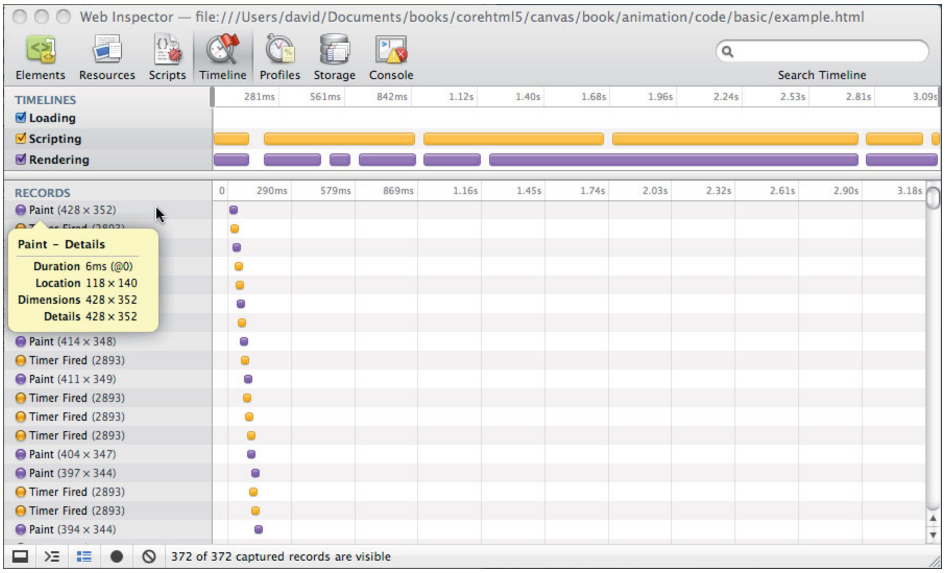
We discuss three tools:

- Profilers
- Timelines
- jsPerf

The first two tools in the preceding list are provided by browsers directly or are offered as add-ons. jsPerf, on the other hand, is a website that lets you create performance tests and make them public. In the sections that follow we will look at profiling and timeline tools available in Chrome and Safari, and then we will take a look at jsPerf.

1.4.4.1 Profiles and Timelines

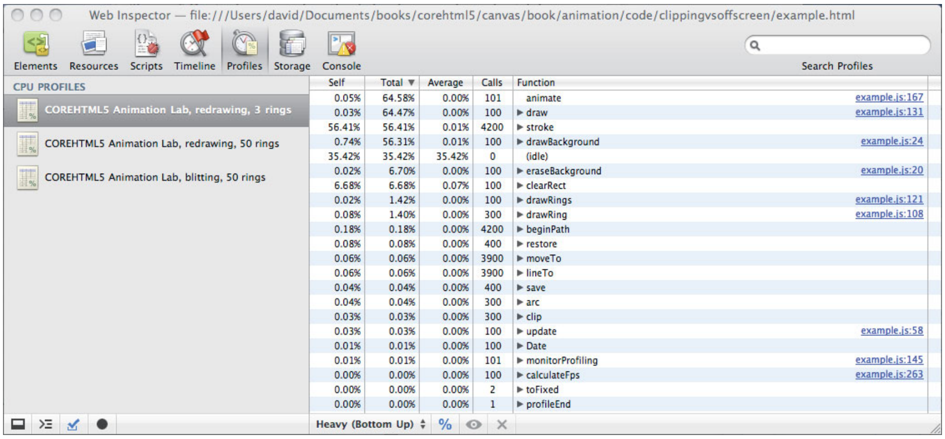
Profiles and timelines are indispensable for discovering performance bottlenecks in your code. Figures 1.8 and 1.9 show a timeline and a profile, respectively, for the animation shown in Figure 1.7.



```
Paint (428 × 352)
Timer Fired (2893)
Paint – Details
  Duration 6ms (@0)
  Location 118 × 140
  Dimensions 428 × 352
  Details 428 × 352
Paint (414 × 348)
Timer Fired (2893)
Paint (411 × 349)
Timer Fired (2893)
Timer Fired (2893)
Timer Fired (2893)
Paint (404 × 347)
Paint (397 × 344)
Timer Fired (2893)
Timer Fired (2893)
Paint (394 × 344)

372 of 372 captured records are visible
```

Figure 1.8 Timelines

Timelines give you a record of significant events that occur in your application, along with details of those events such as their duration and the area of the window they affect. In WebKit-based browsers, such as Chrome and Safari, you can hover the mouse over those events to obtain their associated details, as illustrated in Figure 1.8.



Web Inspector — file:///Users/david/Documents/books/corehtml5/canvas/book/animation/code/clippingvsoffscreen/example.html

CPU PROFILES

- COREHTML5 Animation Lab, redrawing, 3 rings
- COREHTML5 Animation Lab, redrawing, 50 rings
- COREHTML5 Animation Lab, blitting, 50 rings

Self	Total ▼	Average	Calls	Function	
0.05%	64.58%	0.00%	101	animate	example.js:167
0.03%	64.47%	0.00%	100	▶ draw	example.js:131
56.41%	56.41%	0.01%	4200	▶ stroke	
0.74%	56.31%	0.01%	100	▶ drawBackground	example.js:24
35.42%	35.42%	35.42%	0	(idle)	
0.02%	6.70%	0.00%	100	▶ eraseBackground	example.js:20
6.68%	6.68%	0.07%	100	▶ clearRect	
0.02%	1.42%	0.00%	100	▶ drawRings	example.js:121
0.08%	1.40%	0.00%	300	▶ drawRing	example.js:108
0.18%	0.18%	0.00%	4200	▶ beginPath	
0.08%	0.08%	0.00%	400	▶ restore	
0.06%	0.06%	0.00%	3900	▶ moveTo	
0.06%	0.06%	0.00%	3900	▶ lineTo	
0.04%	0.04%	0.00%	400	▶ save	
0.04%	0.04%	0.00%	300	▶ arc	
0.03%	0.03%	0.00%	300	▶ clip	
0.03%	0.03%	0.00%	100	▶ update	example.js:58
0.01%	0.01%	0.00%	100	▶ Date	
0.01%	0.01%	0.00%	101	▶ monitorProfiling	example.js:145
0.00%	0.00%	0.00%	100	▶ calculateFps	example.js:263
0.00%	0.00%	0.00%	2	▶ toFixed	
0.00%	0.00%	0.00%	1	▶ profileEnd	

Figure 1.9 Profiles

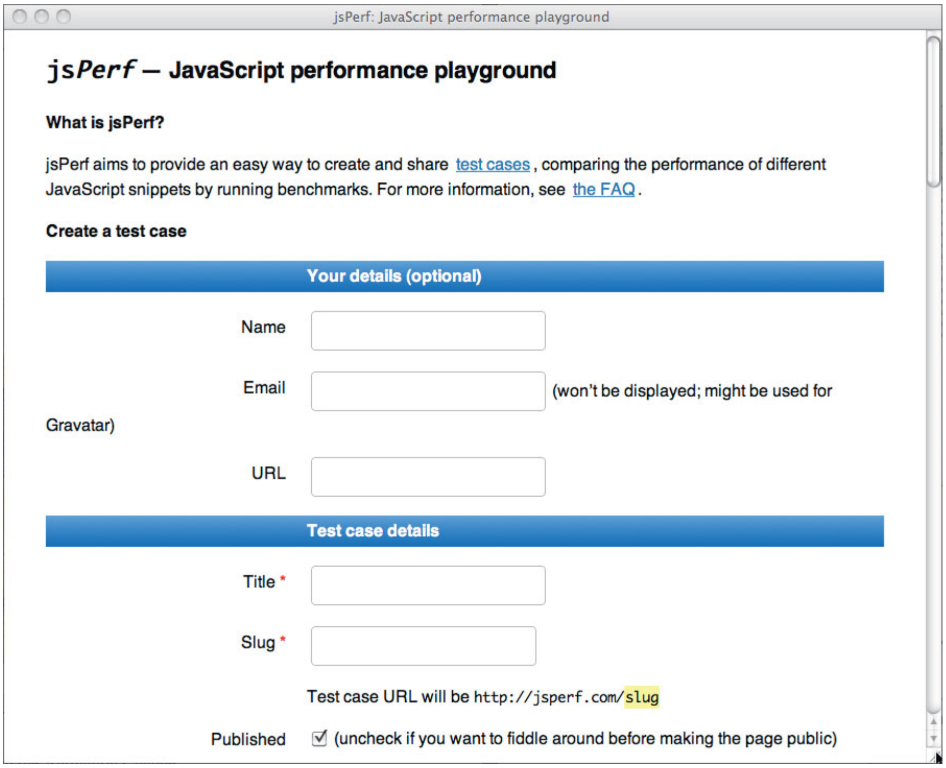
Profilers give you a much more detailed view of how your code performs at the function level. As you can see in Figure 1.9, profiles show you how many times each function in your application is called, and how long those functions take. You can see what percentage of the total execution time is taken up by each function, and you can also discover exactly how many milliseconds each function takes, on average, to execute.

1.4.4.2 jsPerf

jsPerf, shown in Figure 1.10, is a website that lets you create and share JavaScript benchmarks.

You may wonder, for example, what's the most efficient way to loop through pixels in an image that you are processing in a canvas. If you click the "test cases" link, shown at the top of the screenshot in Figure 1.10, jsPerf displays all of the publicly available test cases, as shown in Figure 1.11.

In fact, not only are there many Canvas-related tests at jsperf.com, there is a test case that matches the description in the preceding paragraph, which is highlighted in Figure 1.11. If you click the link for that test case, jsPerf shows you the code for the test case, as shown in Figure 1.12. You can run the test case yourself, and your results will be added to the test case. You can also look at the results for all the different browsers that users have used to run the test case (not shown in Figure 1.12).



jsPerf — JavaScript performance playground

What is jsPerf?

jsPerf aims to provide an easy way to create and share test cases, comparing the performance of different JavaScript snippets by running benchmarks. For more information, see the FAQ.

Create a test case

Your details (optional)

Name

Email (won't be displayed; might be used for Gravatar)

URL

Test case details

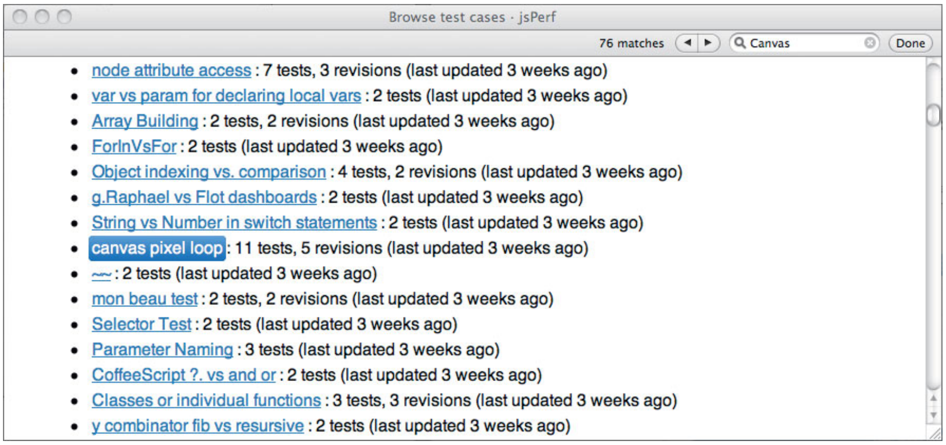
Title *

Slug *

Test case URL will be `http://jsperf.com/slug`

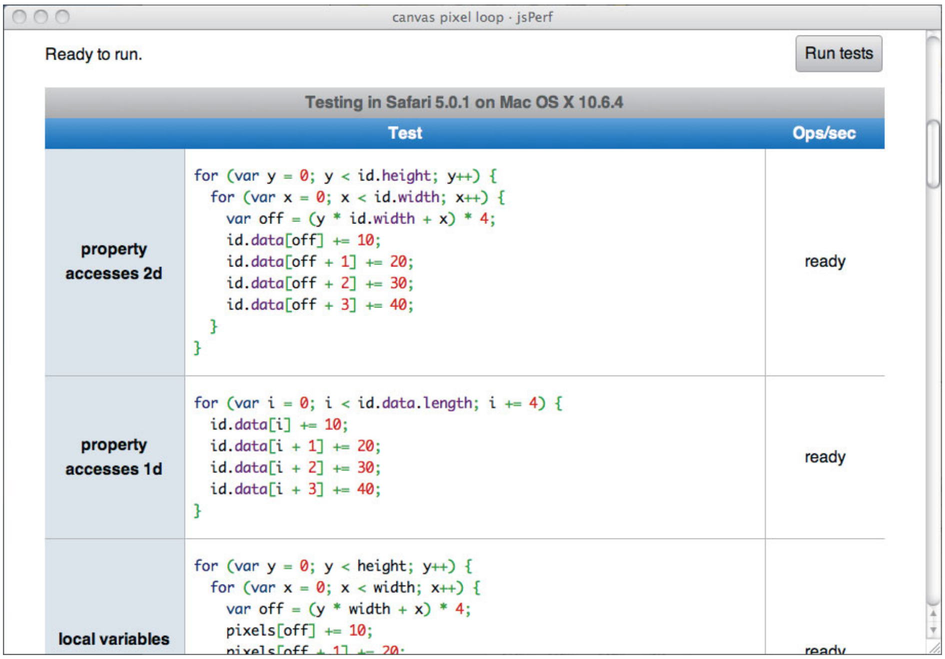
Published ☑ (uncheck if you want to fiddle around before making the page public)

Figure 1.10 jsperf.com homepage



- node attribute access : 7 tests, 3 revisions (last updated 3 weeks ago)
- var vs param for declaring local vars : 2 tests (last updated 3 weeks ago)
- Array Building : 2 tests, 2 revisions (last updated 3 weeks ago)
- ForInVsFor : 2 tests (last updated 3 weeks ago)
- Object indexing vs. comparison : 4 tests, 2 revisions (last updated 3 weeks ago)
- g.Raphael vs Flot dashboards : 2 tests (last updated 3 weeks ago)
- String vs Number in switch statements : 2 tests (last updated 3 weeks ago)
- canvas pixel loop : 11 tests, 5 revisions (last updated 3 weeks ago)
- ~~~ : 2 tests (last updated 3 weeks ago)
- mon beau test : 2 tests, 2 revisions (last updated 3 weeks ago)
- Selector Test : 2 tests (last updated 3 weeks ago)
- Parameter Naming : 3 tests (last updated 3 weeks ago)
- CoffeeScript ?. vs and or : 2 tests (last updated 3 weeks ago)
- Classes or individual functions : 3 tests, 3 revisions (last updated 3 weeks ago)
- y combinator fib vs resursive : 2 tests (last updated 3 weeks ago)

Figure 1.11 Code for a Canvas test case at jsfperf.com



Ready to run.

Testing in Safari 5.0.1 on Mac OS X 10.6.4		
	Test	Ops/sec
property accesses 2d	`for (var y = 0; y < id.height; y++) { for (var x = 0; x < id.width; x++) { var off = (y * id.width + x) * 4; id.data[off] += 10; id.data[off + 1] += 20; id.data[off + 2] += 30; id.data[off + 3] += 40; } }`	ready
property accesses 1d	`for (var i = 0; i < id.data.length; i += 4) { id.data[i] += 10; id.data[i + 1] += 20; id.data[i + 2] += 30; id.data[i + 3] += 40; }`	ready
local variables	`for (var y = 0; y < height; y++) { for (var x = 0; x < width; x++) { var off = (y * width + x) * 4; pixels[off] += 10; pixels[off + 1] += 20;`	ready

Figure 1.12 A test case for looping through image pixels

Now that we're done with the preliminaries, let's look at how to draw into a canvas.

1.5 Fundamental Drawing Operations

In the next chapter we will look closely at drawing in a canvas. For now, however, to familiarize you with the drawing methods that the Canvas API provides, let's begin with the application shown in Figure 1.13, which implements an analog clock.

The clock application, which is listed in Example 1.4, uses the following drawing methods from the Canvas API:

- `arc()`
- `beginPath()`
- `clearRect()`

- `fill()`
- `fillText()`
- `lineTo()`
- `moveTo()`
- `stroke()`

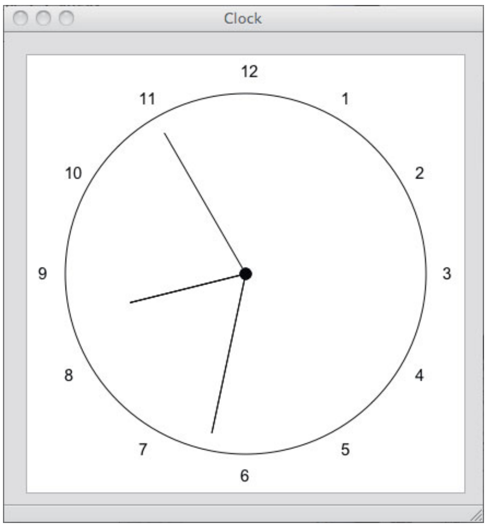


Figure 1.13 A clock

Like Adobe Illustrator and Apple's Cocoa, Canvas lets you draw shapes by creating invisible paths that you subsequently make visible with calls to `stroke()`, which strokes the outline of the path, or `fill()`, which fills the inside of the path. You begin a path with the `beginPath()` method.

The clock application's `drawCircle()` method draws the circle representing the clock face by invoking `beginPath()` to begin a path, and subsequently invokes `arc()` to create a circular path. That path is invisible until the application invokes `stroke()`. Likewise, the application's `drawCenter()` method draws the small filled circle at the center of the clock with a combination of `beginPath()`, `arc()`, and `fill()`.

The application's `drawNumerals()` method draws the numbers around the face of the clock with the `fillText()` method, which draws filled text in the canvas. Unlike the `arc()` method, `fillText()` does not create a path; instead, `fillText()` immediately renders text in the canvas.

The clock hands are drawn by the application's `drawHand()` method, which uses three methods to draw the lines that represent the clock hands: `moveTo()`, `lineTo()`, and `stroke()`. The `moveTo()` method moves the graphics pen to a specific location in the canvas, `lineTo()` draws an invisible path to the location that you specify, and `stroke()` makes the current path visible.

The application animates the clock with `setInterval()`, which invokes the application's `drawClock()` function once every second. The `drawClock()` function uses `clearRect()` to erase the canvas, and then it redraws the clock.

Example 1.4 A basic clock

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    FONT_HEIGHT = 15,
    MARGIN = 35,
    HAND_TRUNCATION = canvas.width/25,
    HOUR_HAND_TRUNCATION = canvas.width/10,
    NUMERAL_SPACING = 20,
    RADIUS = canvas.width/2 - MARGIN,
    HAND_RADIUS = RADIUS + NUMERAL_SPACING;

// Functions..........................................................

function drawCircle() {
   context.beginPath();
   context.arc(canvas.width/2, canvas.height/2,
               RADIUS, 0, Math.PI*2, true);
   context.stroke();
}

function drawNumerals() {
   var numerals = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
       angle = 0,
       numeralWidth = 0;

   numerals.forEach(function(numeral) {
      angle = Math.PI/6 * (numeral-3);
      numeralWidth = context.measureText(numeral).width;
      context.fillText(numeral,
         canvas.width/2  + Math.cos(angle)*(HAND_RADIUS) -
            numeralWidth/2,
         canvas.height/2 + Math.sin(angle)*(HAND_RADIUS) +
            FONT_HEIGHT/3);
   });
}
```

```
function drawCenter() {
   context.beginPath();
   context.arc(canvas.width/2, canvas.height/2, 5, 0, Math.PI*2, true);
   context.fill();
}

function drawHand(loc, isHour) {
   var angle = (Math.PI*2) * (loc/60) - Math.PI/2,
      handRadius = isHour ? RADIUS - HAND_TRUNCATION-HOUR_HAND_TRUNCATION
                          : RADIUS - HAND_TRUNCATION;

   context.moveTo(canvas.width/2, canvas.height/2);
   context.lineTo(canvas.width/2  + Math.cos(angle)*handRadius,
                  canvas.height/2 + Math.sin(angle)*handRadius);
   context.stroke();
}

function drawHands() {
   var date = new Date,
       hour = date.getHours();

   hour = hour > 12 ? hour - 12 : hour;

   drawHand(hour*5 + (date.getMinutes()/60)*5, true);
   drawHand(date.getMinutes(), false);
   drawHand(date.getSeconds(), false);
}

function drawClock() {
   context.clearRect(0,0,canvas.width,canvas.height);

   drawCircle();
   drawCenter();
   drawHands();
   drawNumerals();
}

// Initialization..............................................

context.font = FONT_HEIGHT + 'px Arial';
loop = setInterval(drawClock, 1000);
```

NOTE: A closer look at paths, stroking, and filling

The clock example in this section gives you an overview of what it's like to draw into a canvas. In Chapter 2, we will take a closer look at drawing and manipulating shapes in a canvas.

1.6 Event Handling

HTML5 applications are event driven. You register event listeners with HTML elements and implement code that responds to those events. Nearly all Canvas-based applications handle either mouse or touch events—or both—and many applications also handle various events such as keystrokes and drag and drop.

1.6.1 Mouse Events

Detecting mouse events in a canvas is simple enough: You add an event listener to the canvas, and the browser invokes that listener when the event occurs. For example, you can listen to mouse down events, like this:

```
canvas.onmousedown = function (e) {
   // React to the mouse down event
};
```

Alternatively, you can use the more generic `addEventListener()` method:

```
canvas.addEventListener('mousedown', function (e) {
   // React to the mouse down event
});
```

In addition to `onmousedown`, you can also assign functions to `onmousemove`, `onmouseup`, `onmouseover`, and `onmouseout`.

Assigning a function to `onmousedown`, `onmousemove`, etc., is a little simpler than using `addEventListener()`; however, `addEventListener()` is necessary when you need to attach multiple listeners to a single mouse event.

1.6.1.1 Translating Mouse Coordinates to Canvas Coordinates

The mouse coordinates in the event object that the browser passes to your event listener are *window* coordinates, instead of being relative to the canvas itself.

Most of the time you need to know where mouse events occur relative to the canvas, not the window, so you must convert the coordinates. For example, Figure 1.14 shows a canvas that displays an image known as a sprite sheet. Sprite sheets are a single image that contains several images for an animation. As an animation progresses, you display one image at a time from the sprite sheet, which means that you must know the exact coordinates of each image in the sprite sheet.

The application shown in Figure 1.14 lets you determine the location of each image in a sprite sheet by tracking and displaying mouse coordinates. As the user moves

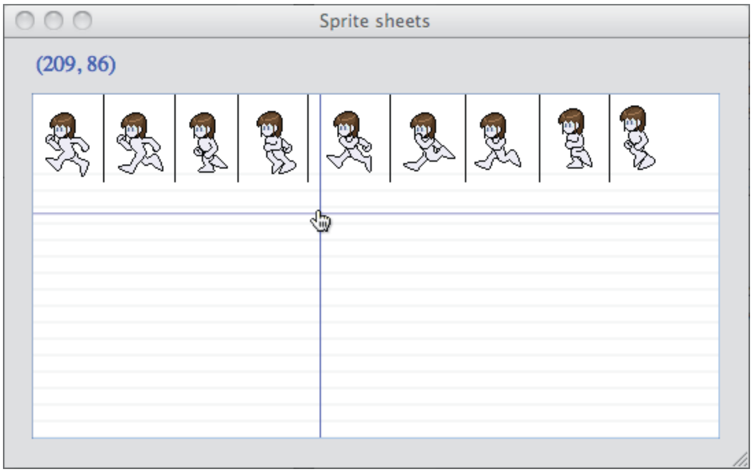


Figure 1.14 Sprite sheet inspector

the mouse, the application continuously updates the mouse coordinates above the sprite sheet and the guidelines.

The application adds a `mousemove` listener to the canvas, and subsequently, when the browser invokes that listener, the application converts the mouse coordinates from the window to the canvas, with a `windowToCanvas()` method, like this:

```
function windowToCanvas(canvas, x, y) {
   var bbox = canvas.getBoundingClientRect();

   return { x: (x - bbox.left) * (canvas.width  / bbox.width),
            y: (y - bbox.top)  * (canvas.height / bbox.height)
          };
}

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(canvas, e.clientX, e.clientY);

   drawBackground();
   drawSpritesheet();
   drawGuidelines(loc.x, loc.y);
   updateReadout(loc.x, loc.y);
};
...
```

The `windowToCanvas()` method shown above invokes the canvas's `getBoundingClientRect()` method to obtain the canvas's bounding box relative to the window. The `windowToCanvas()` method then returns an object with `x` and `y` properties that correspond to the mouse location in the canvas.

Notice that not only does `windowToCanvas()` subtract the left and top of the canvas's bounding box from the `x` and `y` window coordinates, it also scales those coordinates when the canvas element's size differs from the size of the drawing surface. See Section 1.1.1, "Canvas Element Size vs. Drawing Surface Size," on p. 5 for an explanation of canvas element size versus canvas drawing surface size.

The HTML for the application shown in **Figure 1.14** is listed in **Example 1.5**, and the JavaScript is listed in **Example 1.6**.

Example 1.5 A sprite sheet inspector: HTML

```
<!DOCTYPE html>
   <head>
     <title>Sprite sheets</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            position: absolute;
            left: 0px;
            top: 20px;
            margin: 20px;
            background: #ffffff;
            border: thin inset rgba(100,150,230,0.5);
            cursor: pointer;
         }

         #readout {
            margin-top: 10px;
            margin-left: 15px;
            color: blue;
         }
      </style>
   </head>

  <body>
    <div id='readout'></div>

    <canvas id='canvas' width='500' height='250'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

Example 1.6 A sprite sheet inspector: JavaScript

```
var canvas = document.getElementById('canvas'),
    readout = document.getElementById('readout'),
    context = canvas.getContext('2d'),
    spritesheet = new Image();

// Functions.........................................................

function windowToCanvas(canvas, x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: (x - bbox.left) * (canvas.width  / bbox.width),
            y: (y - bbox.top)  * (canvas.height / bbox.height)
          };
}

function drawBackground() {
   var VERTICAL_LINE_SPACING = 12,
       i = context.canvas.height;

   context.clearRect(0,0,canvas.width,canvas.height);
   context.strokeStyle = 'lightgray';
   context.lineWidth = 0.5;

   while(i > VERTICAL_LINE_SPACING*4) {
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
      i -= VERTICAL_LINE_SPACING;
   }
}

function drawSpritesheet() {
   context.drawImage(spritesheet, 0, 0);
}

function drawGuidelines(x, y) {
   context.strokeStyle = 'rgba(0,0,230,0.8)';
   context.lineWidth = 0.5;
   drawVerticalLine(x);
   drawHorizontalLine(y);
}

function updateReadout(x, y) {
   readout.innerHTML = '(' + x.toFixed(0) + ', ' + y.toFixed(0) + ')';
}
```

(Continues)

Example 1.6 *(Continued)*

```
function drawHorizontalLine (y) {
   context.beginPath();
   context.moveTo(0,y + 0.5);
   context.lineTo(context.canvas.width, y + 0.5);
   context.stroke();
}

function drawVerticalLine (x) {
   context.beginPath();
   context.moveTo(x + 0.5, 0);
   context.lineTo(x + 0.5, context.canvas.height);
   context.stroke();
}

// Event handlers.....................................................

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(canvas, e.clientX, e.clientY);

   drawBackground();
   drawSpritesheet();
   drawGuidelines(loc.x, loc.y);
   updateReadout(loc.x, loc.y);
};

// Initialization.....................................................

spritesheet.src = 'running-sprite-sheet.png';
spritesheet.onload = function(e) {
   drawSpritesheet();
};

drawBackground();
```

TIP: x and y vs. `clientX` and `clientY`

In pre-HTML5 days, obtaining window coordinates for mouse events from the event object that the browser passes to your event listeners was a mess. Some browsers stored those coordinates in `x` and `y`, and others stored them in `clientX` and `clientY`. Fortunately, modern browsers that support HTML5 have finally come to agreement, and they all support `clientX` and `clientY`. You can read more about those event properties at http://www.quirksmode.org/js/events_mouse.html.

TIP: Tell the browser to butt out . . .

When you listen to mouse events, the browser invokes your listener when the associated event occurs. After you handle the event, the browser also reacts to the event. Much of the time when you handle mouse events in a canvas, you don't want the browser to handle the event after you're done with it because you will end up with unwanted effects, such as the browser selecting other HTML elements or changing the cursor.

Fortunately, the event object comes with a `preventDefault()` method that, as its name suggests, prevents the browser from carrying out its default reaction to the event. Just invoke that method from your event handler, and the browser will no longer interfere with your event handling.

NOTE: The Canvas context's `drawImage()` method

The example shown in Figure 1.14 uses the 2d context's `drawImage()` method to draw the sprite sheet. That single method lets you copy all or part of an image stored in one place to another place, and if you wish, you can scale the image along the way.

The sprite sheet application uses `drawImage()` in the simplest possible way: The application draws all of an image, unscaled, that is stored in an `Image` object, into the application's canvas. In Chapter 4 and throughout the rest of this book, you will see more advanced uses for `drawImage()`.

1.6.2 Keyboard Events

When you press a key in a browser window, the browser generates key events. Those events are targeted at the HTML element that currently has focus. If no element has focus, key events bubble up to the `window` and `document` objects.

The canvas element is not a focusable element, and therefore in light of the preceding paragraph, adding key listeners to a canvas is an exercise in futility. Instead, you will add key listeners to either the `document` or `window` objects to detect key events.

There are three types of key events:

- `keydown`
- `keypress`
- `keyup`

The `keydown` and `keyup` events are low-level events that the browser fires for nearly every keystroke. Note that some keystrokes, such as command sequences, may be *swallowed* by the browser or the operating system; however, most keystrokes make it through to your `keydown` and `keyup` event handlers, including keys such as Alt, Esc, and so on.

When a `keydown` event generates a printable character, the browser fires a `keypress` event before the inevitable `keyup` event. If you hold a key that generates a printable character down for an extended period of time, the browser will fire a sequence of `keypress` events between the `keydown` and `keyup` events.

Implementing key listeners is similar to implementing mouse listeners. You can assign a function to the `document` or `window` object's `onkeydown`, `onkeyup`, or `onkeypress` variables, or you can call `addEventListener()`, with `keydown`, `keyup`, or `keypress` for the first argument, and a reference to a function for the second argument.

Determining which key was pressed can be complicated, for two reasons. First, there is a huge variety of characters among all the languages of the world. When you must take into consideration the Latin alphabet, Asian ideographic characters, and the many languages of India, just to mention a few, supporting them all is mind boggling.

Second, although browsers and keyboards have been around for a long time, key codes have never been standardized until DOM Level 3, which few browsers currently support. In a word, detecting exactly what key or combination of keys has been pressed is a mess.

However, under most circumstances you can get by with the following two simple strategies:

- For `keydown` and `keyup` events, look at the `keyCode` property of the event object that the browser passes to your event listener. In general, for printable characters, those values will be ASCII codes. Notice the *in general* caveat, however. Here is a good website that you can consult for interpreting key codes among different browsers: http://bit.ly/o3b1L2. Event objects for key events also contain the following boolean properties:
  - `altKey`
  - `ctrlKey`
  - `metaKey`
  - `shiftKey`

- For `keypress` events—which browsers fire only for printable characters—you can reliably get that character like this:

```
var key = String.fromCharCode(event.which);
```

In general, unless you are implementing a text control in a canvas, you will handle mouse events much more often than you handle key events. One other common use case for key events, however, is handling keystrokes in games. We discuss that topic in Chapter 9.

1.6.3 Touch Events

With the advent of smart phones and tablet computers, the HTML specification has added support for touch events. See Chapter 11 for more information about handling touch events.

1.7 Saving and Restoring the Drawing Surface

In Section 1.2.2, "Saving and Restoring Canvas State," on p. 11, you saw how to save and restore a context's state. Saving and restoring context state lets you make temporary state changes, which is something you will do frequently.

Another essential feature of the Canvas context is the ability to save and restore the drawing surface itself. Saving and restoring the drawing surface lets you draw on the drawing surface temporarily, which is useful for many things, such as rubber bands, guidewires, or annotations. For example, the application shown in Figure 1.15 and discussed in Section 2.13.1, "Translating, Scaling, and Rotating," on p. 171, lets users interactively create polygons by dragging the mouse.

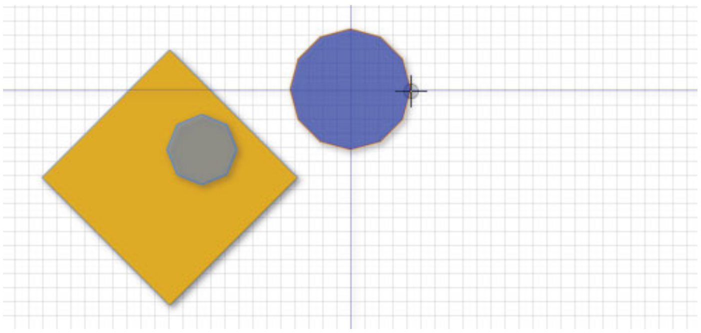


Figure 1.15 Drawing guidewires

On a mouse down event, the application saves the drawing surface. As the user subsequently drags the mouse, the application continuously restores the drawing surface to what it was when the mouse went down and then draws the polygon and the associated guidewires. When the user releases the mouse, the application restores the drawing surface one last time and draws a final representation of the polygon, without guidewires.

The JavaScript from the application shown in Figure 1.15 that pertains to drawing the guidewires is listed in Example 1.7. See Section 2.11.1, "Polygon Objects," on p. 147 for a more complete listing of the application.

NOTE: Image manipulation with `getImageData()` and `putImageData()`

The application shown in Figure 1.15 saves and restores the drawing surface with the context's `getImageData()` and `putImageData()` methods. Like `drawImage()`, `getImageData()` and `putImageData()` can be used in a number of different ways; one common use is implementing image filters that get an image's data, manipulate it, and put it back into a canvas. You will see how to implement image filters in Section 4.5.2.3, "Filtering Images," on p. 293, among other uses for `getImageData()` and `putImageData()`.

NOTE: Immediate-mode graphics

Canvas implements what's known as *immediate-mode graphics*, meaning that it immediately draws whatever you specify in the canvas. Then it immediately forgets what you have just drawn, meaning that canvases do not retain a list of objects to draw. Some graphics systems, such as SVG, do maintain a list of objects to draw. Those graphics systems are referred to as retained-mode graphics.

Immediate-mode graphics, because it does not maintain a list of objects to draw, is more low-level than retained-mode graphics. Immediate-mode graphics is also more flexible because you draw straight to the screen instead of adjusting objects that the graphics system draws for you.

Immediate-mode graphics is more suited to applications, such as paint applications, that do not keep track of what the user has drawn, whereas retained-mode graphics is more suited to applications, such as drawing applications, that let you manipulate graphical objects that you create.

In Section 2.11.1, "Polygon Objects," on p. 147 you will see how to implement a simple retained-mode graphics system that maintains an array of polygons in a drawing application, which lets users drag those polygons to reposition them.

Example 1.7 Drawing guidewires by saving and restoring the drawing surface

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

// Save and restore drawing surface...................................

function saveDrawingSurface() {
   drawingSurfaceImageData = context.getImageData(0, 0,
                             canvas.width,
                             canvas.height);
}

function restoreDrawingSurface() {
   context.putImageData(drawingSurfaceImageData, 0, 0);
}

// Event handlers.....................................................

canvas.onmousedown = function (e) {
   ...
   saveDrawingSurface();
   ...
};

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(e);

   if (dragging) {
      restoreDrawingSurface();
      ...

      if (guidewires) {
        drawGuidewires(mousedown.x, mousedown.y);
      }
   }
};

canvas.onmouseup = function (e) {
   ...
   restoreDrawingSurface();
};
```

1.8 Using HTML Elements in a Canvas

Canvas is arguably the coolest feature of HTML5, but when you use it to implement web applications, you will rarely use it alone. Most of the time you will combine one or more canvases with other HTML controls so that your users can provide input or otherwise control the application.

To combine other HTML controls with your canvases, you may first be inclined to embed those controls inside your canvas elements, but that won't work, because anything you put in the body of a canvas element is displayed by the browser only if the browser does not support the canvas element.

Because browsers will display either a canvas element or HTML controls that you put inside that element, but not both, you must place your controls outside of your canvas elements.

To make it appear as though HTML controls are inside a canvas, you can use CSS to place the controls above the canvas. The application shown in Figure 1.16 illustrates that effect.

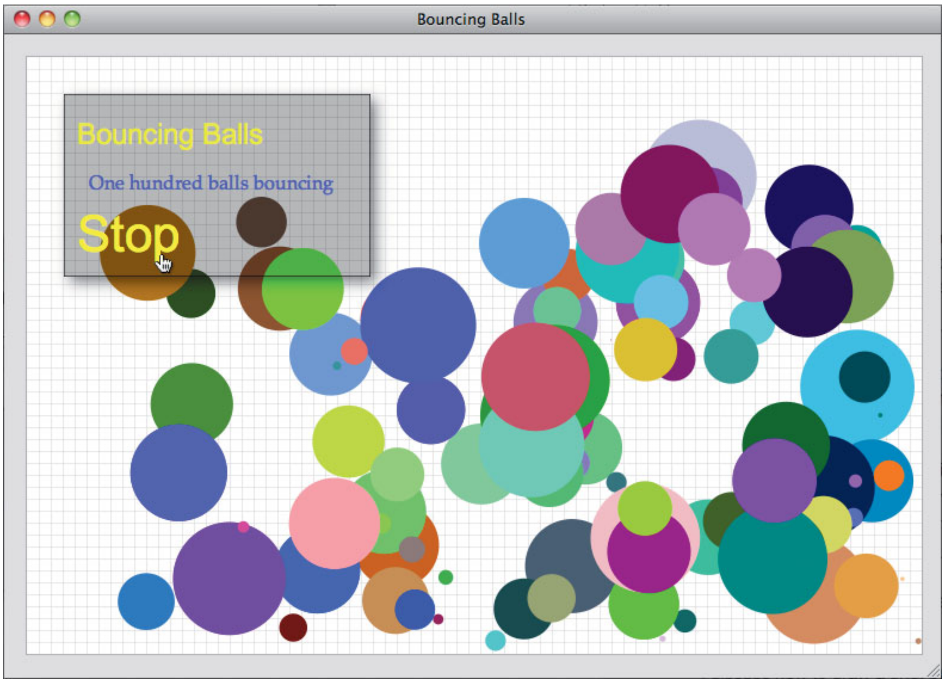


Figure 1.16 HTML elements above a canvas

The application shown in Figure 1.16 animates 100 balls and provides a link to start and stop the animation. That link resides in a DIV element that is partially transparent and floats above the canvas. We refer to that DIV as a *glass pane* because it appears to be a pane of glass floating above the canvas.

The HTML for the application shown in Figure 1.16 is listed in Example 1.8.

Example 1.8 HTML controls in a canvas: HTML

```
<!DOCTYPE html>
<html>
   <head>
      <title>Bouncing Balls</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            margin-left: 10px;
            margin-top: 10px;
            background: #ffffff;
            border: thin solid #aaaaaa;
         }

         #glasspane {
            position: absolute;
            left: 50px;
            top: 50px;
            padding: 0px 20px 10px 10px;
            background: rgba(0, 0, 0, 0.3);
            border: thin solid rgba(0, 0, 0, 0.6);
            color: #eeeeee;
            font-family: Droid Sans, Arial, Helvetica, sans-serif;
            font-size: 12px;
            cursor: pointer;
            -webkit-box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
            -moz-box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
            box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
         }

         #glasspane h2 {
            font-weight: normal;
         }
```

(Continues)

Example 1.8 *(Continued)*

```
         #glasspane .title {
            font-size: 2em;
            color: rgba(255, 255, 0, 0.8);
         }

         #glasspane a:hover {
            color: yellow;
         }

         #glasspane a {
            text-decoration: none;
            color: #cccccc;
            font-size: 3.5em;
         }

         #glasspane p {
            margin: 10px;
            color: rgba(65, 65, 220, 1.0);
            font-size: 12pt;
            font-family: Palatino, Arial, Helvetica, sans-serif;
         }
      </style>
   </head>

   <body>
      <div id='glasspane'>
         <h2 class='title'>Bouncing Balls</h2>

         <p>One hundred balls bouncing</p>

         <a id='startButton'>Start</a>
      </div>

      <canvas id='canvas' width='750' height='500'>
         Canvas not supported
      </canvas>

      <script src='example.js'></script>
  </body>
</html>
```

The HTML shown in Example 1.8 uses CSS absolute positioning to make the glass pane appear above the canvas, like this:

```
#canvas {
   margin-left: 10px;
   margin-top: 10px;
   background: #ffffff;
   border: thin solid #aaaaaa;
}

#glasspane {
   position: absolute;
   left: 50px;
   top: 50px;
   ...
}
```

The preceding CSS uses *static* positioning for the canvas, which is the default for the `position` CSS property, whereas it specifies *absolute* positioning for the glass pane. The CSS specification states that elements with absolute positioning are drawn on top of elements with static positioning, which is why the glass pane appears above the canvas in Figure 1.16.

If you also change the canvas's positioning to absolute, then the canvas will appear on top of the glass pane, and you won't see the glass pane because the canvas's background is not transparent. In that case, the glass pane is underneath the canvas because the `canvas` element comes after the glass pane's `DIV` element. If you switch the order of those elements, then the glass pane will once again appear above the canvas.

So, you have two options to position the glass pane above the canvas: Use relative positioning for the canvas and absolute positioning for the glass pane; or use either relative or absolute positioning for both elements and declare the glass pane's `DIV` after the `canvas` element.

A third option is to use either relative or absolute positioning for both elements and manipulate their `z-index` CSS property. The browser draws elements with a higher `z-index` above elements with a lower `z-index`.

In addition to placing HTML controls where you want them to appear, you also need to obtain references to those elements in your JavaScript so that you can access and manipulate their values.

The application shown in **Figure 1.16** obtains references to the glass pane and the button that controls the animation and adds event handlers to them, like this:

```
var context = document.getElementById('canvas').getContext('2d'),
    startButton = document.getElementById('startButton'),
    glasspane = document.getElementById('glasspane'),
    paused = false,
    ...

startButton.onclick = function(e) {
   e.preventDefault();
   paused = ! paused;
   startButton.innerHTML = paused ? 'Start' : 'Stop';
};
...

glasspane.onmousedown = function(e) {
   e.preventDefault();
};
```

The preceding JavaScript adds an `onclick` handler to the button that starts or pauses the animation based on the current state of the application, and adds an `onmousedown` event handler to the glass pane to prevent the browser from its default reaction to that mouse click. The `onmousedown` handler prevents the browser from reacting to the event to avoid inadvertent selections.

NOTE: You can implement your own Canvas-based controls

The Canvas specification states that you should prefer built-in HTML controls rather than implementing controls from scratch with the Canvas API, which in general is good advice. Implementing controls from scratch with the Canvas API generally involves a good deal of work, and most of the time it's wise to avoid a good deal of work when there's an easier alternative.

However, in some circumstances it makes sense to implement Canvas-based controls. In Chapter 10, will see both motivations for implementing your own Canvas-based controls and ways to do so.

NOTE: Drawing a grid

The application discussed in this section draws a grid underneath the bouncing balls to emphasize that the floating `DIV` is indeed floating above the canvas.

In Chapter 2, we discuss how to draw a grid, but for now you can safely forge ahead without knowing grid drawing details.

1.8.1 Invisible HTML Elements

In the preceding section you saw how to combine static HTML controls with a canvas. In this section we explore a more advanced use of HTML controls that involves dynamically modifying the size of a DIV as the user drags the mouse.

Figure 1.17 shows an application that uses a technique known as rubberbanding to select a region of a canvas. That canvas initially displays an image, and when you select a region of that image, the application reacts by zooming into the region that you selected.

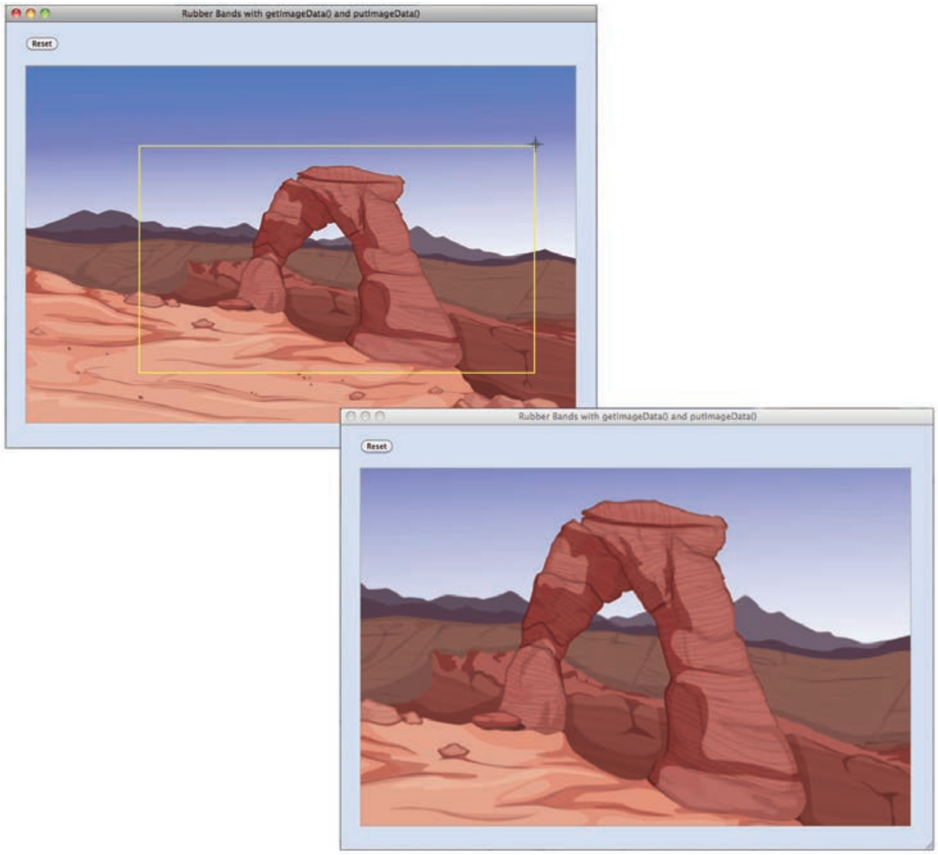

Rubber Bands with getImageData() and putImageData()
Reset
Rubber Bands with getImageData() and putImageData()
Reset

Figure 1.17 Implementing rubber bands with a DIV

First, let's take a look at the HTML for the application, which is listed in Example 1.9.

Example 1.9 Rubber band with a floating DIV

```
<!DOCTYPE html>
<html>
   <head>
     <title>Rubber bands with layered elements</title>

      <style>
         body {
            background: rgba(100, 145, 250, 0.3);
         }

         #canvas {
            margin-left: 20px;
            margin-right: 0;
            margin-bottom: 20px;
            border: thin solid #aaaaaa;
            cursor: crosshair;
            padding: 0;
         }

         #controls {
            margin: 20px 0px 20px 20px;
         }

         #rubberbandDiv {
            position: absolute;
            border: 3px solid blue;
            cursor: crosshair;
            display: none;
         }

      </style>
   </head>

  <body>
      <div id='controls'>
         <input type='button' id='resetButton' value='Reset'/>
      </div>

      <div id='rubberbandDiv'></div>

      <canvas id='canvas' width='800' height='520'>
         Canvas not supported
      </canvas>

     <script src='example.js'></script>
  </body>
</html>
```

The HTML uses a DIV that contains a button. If you click that button, the application draws the entire image as it is displayed when the application starts.

The application uses a second DIV for the rubber band. That DIV is empty, and its CSS display attribute is set to none, which makes it initially invisible. When you start dragging the mouse, the application makes that second DIV visible, which shows the DIV's border. As you continue dragging the mouse, the application continuously resizes the DIV to produce the illusion of a rubber band, as shown in Figure 1.17.

The JavaScript for the application shown in Figure 1.17 is listed in Example 1.10.

Example 1.10 Rubber bands with a DIV

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    rubberbandDiv = document.getElementById('rubberbandDiv'),
    resetButton = document.getElementById('resetButton'),
    image = new Image(),
    mousedown = {},
    rubberbandRectangle = {},
    dragging = false;

// Functions.......................................................

function rubberbandStart(x, y) {
   mousedown.x = x;
   mousedown.y = y;

   rubberbandRectangle.left = mousedown.x;
   rubberbandRectangle.top = mousedown.y;

   moveRubberbandDiv();
   showRubberbandDiv();

   dragging = true;
}

function rubberbandStretch(x, y) {
   rubberbandRectangle.left = x < mousedown.x ? x : mousedown.x;
   rubberbandRectangle.top  = y < mousedown.y ? y : mousedown.y;

   rubberbandRectangle.width  = Math.abs(x - mousedown.x),
   rubberbandRectangle.height = Math.abs(y - mousedown.y);

   moveRubberbandDiv();
   resizeRubberbandDiv();
}
```

(Continues)

Example 1.10 *(Continued)*

```
function rubberbandEnd() {
   var bbox = canvas.getBoundingClientRect();

   try {
      context.drawImage(canvas,
                        rubberbandRectangle.left - bbox.left,
                        rubberbandRectangle.top - bbox.top,
                        rubberbandRectangle.width,
                        rubberbandRectangle.height,
                        0, 0, canvas.width, canvas.height);
   }
   catch (e) {
      // Suppress error message when mouse is released
      // outside the canvas
   }

   resetRubberbandRectangle();

   rubberbandDiv.style.width = 0;
   rubberbandDiv.style.height = 0;

   hideRubberbandDiv();

   dragging = false;
}

function moveRubberbandDiv() {
   rubberbandDiv.style.top  = rubberbandRectangle.top  + 'px';
   rubberbandDiv.style.left = rubberbandRectangle.left + 'px';
}

function resizeRubberbandDiv() {
   rubberbandDiv.style.width  = rubberbandRectangle.width  + 'px';
   rubberbandDiv.style.height = rubberbandRectangle.height + 'px';
}

function showRubberbandDiv() {
   rubberbandDiv.style.display = 'inline';
}

function hideRubberbandDiv() {
   rubberbandDiv.style.display = 'none';
}

function resetRubberbandRectangle() {
   rubberbandRectangle = { top: 0, left: 0, width: 0, height: 0 };
}
```

```
// Event handlers.....................................................

canvas.onmousedown = function (e) {
   var x = e.clientX,
       y = e.clientY;

   e.preventDefault();
   rubberbandStart(x, y);
};

window.onmousemove = function (e) {
   var x = e.clientX,
       y = e.clientY;

   e.preventDefault();
   if (dragging) {
      rubberbandStretch(x, y);
    }
};

window.onmouseup = function (e) {
   e.preventDefault();
   rubberbandEnd();
};

image.onload = function () {
   context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

resetButton.onclick = function(e) {
   context.clearRect(0, 0, context.canvas.width,
                           context.canvas.height);
   context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

// Initialization.....................................................

image.src = 'curved-road.png';
```

Again, we're getting ahead of ourselves a little bit by using the `drawImage()` method to both draw and zoom in on the image. In Section 4.1, "Drawing Images," on p. 254, we will look closely at that method, and we will also see an alternative way to implement rubber bands that involves manipulating the image's pixels to draw the rubber band itself.

For now, however, our focus is on the rubberband `DIV` and how the code manipulates that `DIV` as the user drags the mouse.

The onmousedown event handler for the canvas invokes the rubberbandStart() method, which moves the DIV's upper left-hand corner to the mouse down location and makes the DIV visible. Because the rubberband DIV's CSS position attribute is absolute, the coordinates for the DIV's upper left-hand corner must be specified in window coordinates, and not as coordinates relative to the canvas.

If the user is dragging the mouse, the onmousemove event handler invokes rubberbandStretch(), which moves and resizes the rubberband DIV.

When the user releases the mouse, the onmouseup event handler invokes rubberbandEnd(), which draws the scaled image and shrinks and hides the rubberband DIV.

Finally, notice that all three mouse event handlers invoke preventDefault() on the event object they are passed. As discussed in Section 1.6.1.1, "Translating Mouse Coordinates to Canvas Coordinates," on p. 26, that call prevents the browser from reacting to the mouse events. If you remove those calls to preventDefault(), the browser will try to select elements on the page, which produces undesired effects if the user drags the mouse outside of the canvas.

1.9 Printing a Canvas

It's often convenient to let users of your application access a canvas as an image. For example, if you implement a paint application, such as the one discussed in Chapter 2, users will expect to be able to print their paintings.

By default, although every canvas is a bitmap, it is not an HTML img element, and therefore users cannot, for example, right-click a canvas and save it to disk, nor can they drag a canvas to their desktop to print later on. The fact that a canvas is not an image is illustrated by the popup menu shown in Figure 1.18.

Fortunately, the Canvas API provides a method—toDataURL()—that returns a reference to a data URL for a given canvas. You can subsequently set the src attribute of an img element equal to that data URL to create an image of your canvas.

In Section 1.5, "Fundamental Drawing Operations," on p. 22, you saw how to implement an analog clock with the Canvas API. Figure 1.19 shows a modified version of that application that lets you take a snapshot of the clock and display it as an image, as described above. As you can see from Figure 1.19, when you right-click on the ensuing image, you can save the image to disk, and because the clock image shown in the bottom screenshot is an img element, you can also drag the image to your desktop.

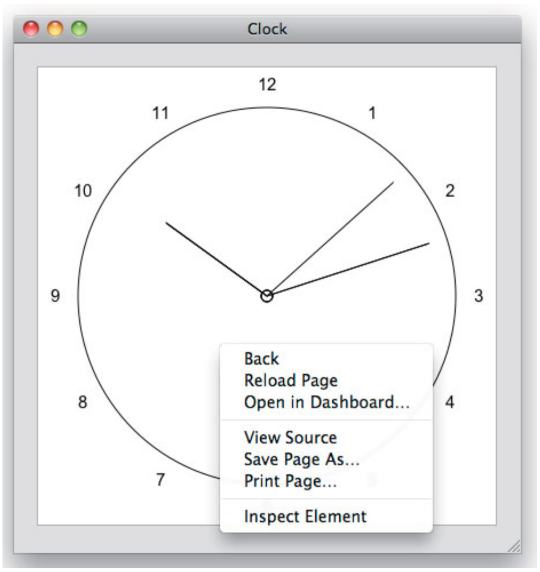


Figure 1.18 The right-click menu for a canvas

The application shown in Figure 1.19 implements a common use case for printing a canvas: It provides a control—in this case, the Take snapshot button—that lets users take a snapshot of the canvas. The application displays that snapshot as an image, so users can right-click the image and save it to disk. Subsequently, when the user clicks the Return to Canvas button, the application replaces the image with the original canvas. Here's a recipe for that use case:

In your HTML page:

- Add an invisible image to the page, and give the image an `id`, but no `src`.
- Use CSS to position and size the image to exactly overlap your canvas.
- Add a control to the page for taking a snapshot.

In your JavaScript:

- Get a reference to the invisible image.
- Get a reference to the snapshot control.
- When the user activates the control to take a snapshot:

  1. Invoke `toDataURL()` to get a data URL.
  2. Assign the data URL to the invisible image's `src` attribute.
  3. Make the image visible and the canvas invisible.

- When the user activates the control to return to the Canvas:
    1. Make the canvas visible and the image invisible.
    2. Redraw the canvas as needed.

Let's see how to translate that recipe to code. Example 1.11 lists the HTML for the application shown in Figure 1.19, and Example 1.12 lists the application's JavaScript.

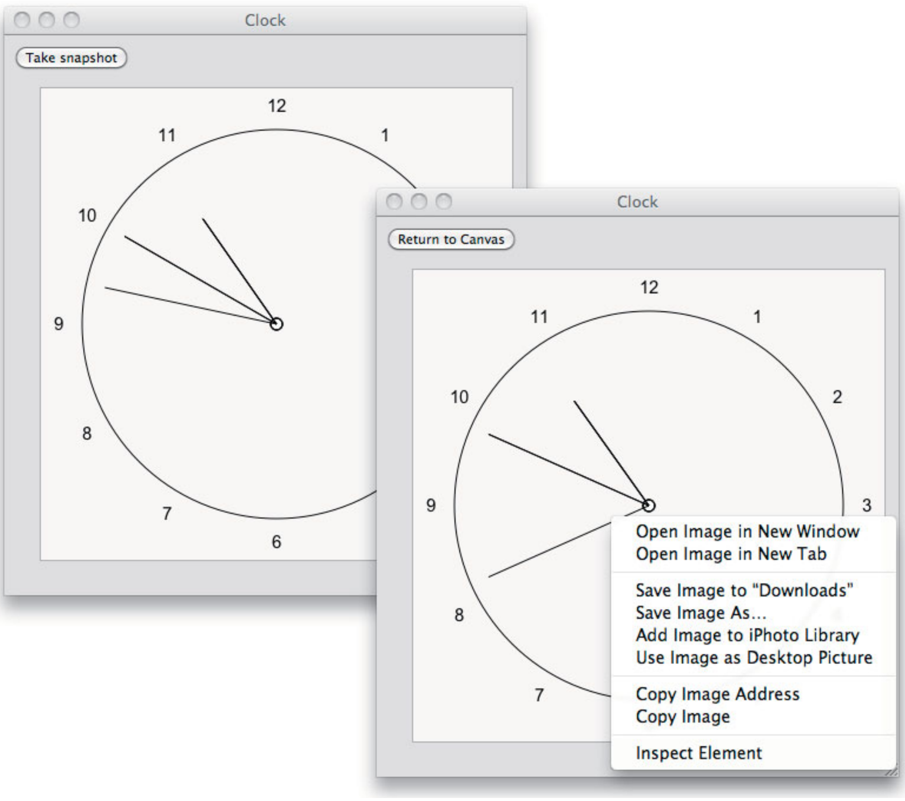


Figure 1.19 Using `toDataURL()`

Example 1.11 Using toDataURL() to print a canvas: HTML

```
<!DOCTYPE html>
   <head>
     <title>Clock</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            position: absolute;
            left: 10px;
            top: 1.5em;
            margin: 20px;
            border: thin solid #aaaaaa;
         }

         #snapshotImageElement {
            position: absolute;
            left: 10px;
            top: 1.5em;
            margin: 20px;
            border: thin solid #aaaaaa;
         }
      </style>
   </head>

  <body>
    <div id='controls'>
      <input id='snapshotButton' type='button' value='Take snapshot'/>
    </div>

    <img id='snapshotImageElement'/>

    <canvas id='canvas' width='400' height='400'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

Example 1.12 Using `toDataURL()` to print a canvas: JavaScript

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    snapshotButton = document.getElementById('snapshotButton'),
    snapshotImageElement =
       document.getElementById('snapshotImageElement'),
    loop;

// Clock drawing functions are omitted from this listing
// in the interests of brevity. See Example 1.4 on p. 24
// for a complete listing of those methods.

// Event handlers.....................................................

snapshotButton.onclick = function (e) {
   var dataUrl;

   if (snapshotButton.value === 'Take snapshot') {
      dataUrl = canvas.toDataURL();
      clearInterval(loop);
      snapshotImageElement.src = dataUrl;
      snapshotImageElement.style.display = 'inline';
      canvas.style.display = 'none';
      snapshotButton.value = 'Return to Canvas';
   }
   else {
      canvas.style.display = 'inline';
      snapshotImageElement.style.display = 'none';
      loop = setInterval(drawClock, 1000);
      snapshotButton.value = 'Take snapshot';
   }
};

// Initialization................................................

context.font = FONT_HEIGHT + 'px Arial';
loop = setInterval(drawClock, 1000);
```

The application accesses the `canvas` and `img` elements and uses CSS absolute positioning to overlap the two elements. When the user clicks the Take snapshot button, the application obtains a data URL from the canvas and assigns it to the `src` attribute of the image. Then it shows the image, hides the canvas, and sets the text of the button to Return to Canvas.

When the user clicks the Return to Canvas button, the application hides the image, displays the canvas, and returns the text of the button to Take snapshot.

NOTE: Canvas blobs

As this book was being written, the Canvas specification added a `toBlob()` method, so you can, among other things, save a canvas as a file. When the book went to press, no browsers supported that method.

1.10 Offscreen Canvases

Another essential Canvas feature is the ability to create and manipulate offscreen canvases. For example, you can, in most cases, considerably boost your performance by storing backgrounds in one or more offscreen canvases and copying parts of those offscreen canvases onscreen.

Another use case for offscreen canvases is the clock that we discussed in the preceding section. Although that application shows you how to implement a general solution that requires user interaction to switch from canvas to image, a clock is a better candidate for an application that does that switching behind the scenes without user intervention.

An updated version of the clock application is shown in Figure 1.20. Once a second, the application draws the clock into the offscreen canvas and assigns the

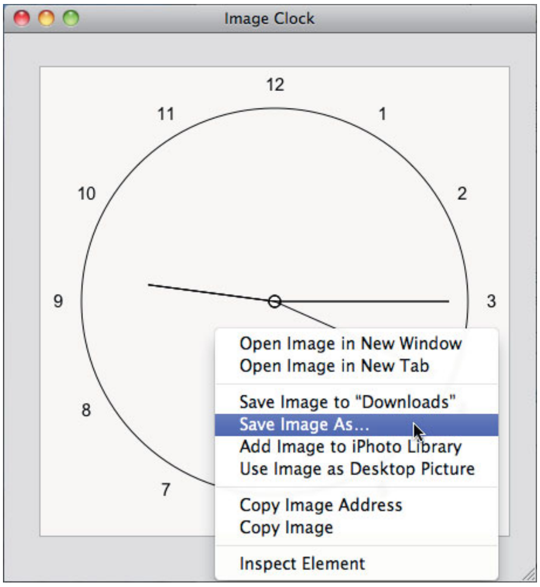


Figure 1.20 Using an offscreen canvas for an image clock

canvas's data URL to the `src` attribute of an image. The result is an animated image that reflects the offscreen canvas. See Section 1.9, "Printing a Canvas," on p. 46 for more information on canvas data URLs.

The HTML for the application shown in **Figure 1.20** is listed in **Example 1.13**.

Example 1.13 An image clock: HTML

```
<!DOCTYPE html>
   <head>
     <title>Image Clock</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            display: none;
         }

         #snapshotImageElement {
            position: absolute;
            left: 10px;
            margin: 20px;
            border: thin solid #aaaaaa;
         }
      </style>
   </head>

   <body>
      <img id='snapshotImageElement'/>

      <canvas id='canvas' width='400' height='400'>
         Canvas not supported
      </canvas>

      <script src='example.js'></script>
   </body>
</html>
```

Notice the CSS for the canvas in the HTML—the canvas is invisible because its `display` attribute is set to `none`. That invisibility makes it an *offscreen* canvas. You can also programmatically create an offscreen canvas, like this: `var offscreen = document.createElement('canvas');`.

The JavaScript pertinent to the offscreen canvas for the application shown in **Figure 1.20** is listed in **Example 1.14**.

Example 1.14 The image clock: JavaScript (excerpt)

```
// Some declarations and functions omitted for brevity.
// See Section 1.9 on p. 46 for a complete listing of
// the clock.

var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

// Functions..........................................................

function updateClockImage() {
   snapshotImageElement.src = canvas.toDataURL();
}

function drawClock() {
   context.clearRect(0, 0, canvas.width, canvas.height);

   context.save();

   context.fillStyle = 'rgba(255,255,255,0.8)';
   context.fillRect(0, 0, canvas.width, canvas.height);

   drawCircle();
   drawCenter();
   drawHands();

   context.restore();

   drawNumerals();

   updateClockImage();
}
...
```

1.11 A Brief Math Primer

To do anything interesting with Canvas, you need a good understanding of basic mathematics, especially working with algebraic equations, trigonometry, and vectors. It also helps, for more complex applications like video games, to be able to derive equations, given units of measure.

Feel free to skim this section if you're comfortable with basic algebra and trigonometry and you can make your way to pixels/frame given pixels/second and milliseconds/frame. Otherwise, spending time in this section will prove fruitful throughout the rest of this book.

Let's get started with solving algebraic equations and trigonometry, and then we'll look at vectors and deriving equations from units of measure.

1.11.1 Solving Algebraic Equations

For any algebraic equation, such as $(10x + 5) \times 2 = 110$, you can do the following, and the equation will still be true:

- Add any real number to both sides
- Subtract any real number from both sides
- Multiply any real number by both sides
- Divide both sides by any real number
- Multiply or divide one or both sides by 1

For example, for $(10x + 5) \times 2 = 110$, you can solve the equation by dividing both sides by 2, to get: $10x + 5 = 55$; then you can subtract 5 from both sides to get: $10x = 50$; and finally, you can solve for x by dividing both sides by 10: $x = 5$.

The last rule above may seem rather odd. Why would you want to multiply or divide one or both sides of an equation by 1? In Section 1.11.4, "Deriving Equations from Units of Measure," on p. 62, where we derive equations from units of measure, we will find a good use for that simple rule.

1.11.2 Trigonometry

Even the simplest uses of Canvas require a rudimentary understanding of trigonometry; for example, in the next chapter you will see how to draw polygons, which requires an understanding of sine and cosine. Let's begin with a short discussion of angles, followed by a look at right triangles.

1.11.2.1 Angles: Degrees and Radians

All the functions in the Canvas API that deal with angles require you to specify angles in radians. The same is true for the JavaScript functions `Math.sin()`, `Math.cos()`, and `Math.tan()`. Most people think of angles in terms of degrees, so you need to know how to convert from degrees to radians.

180 degrees is equal to π radians. To convert from degrees to radians, you can create an algebraic equation for that relationship, as shown in Equation 1.1.

$$180 \text{ degrees} = \pi \text{ radians}$$

Equation 1.1 Degrees and radians

Solving Equation 1.1 for radians, and then degrees, results in Equations 1.2 and 1.3.

$$\text{radians} = (\pi \;/\; 180) \times \text{degrees}$$

Equation 1.2 Degrees to radians

$$\text{degrees} = (180 \;/\; \pi) \times \text{radians}$$

Equation 1.3 Radians to degrees

π is roughly equal to 3.14, so, for example, 45 degrees is equal to (3.14 / 180) × 45 radians, which works out to 0.7853.

1.11.2.2 Sine, Cosine, and Tangent

To make effective use of Canvas, you must have a basic understanding of sin, cos, and tan, so if you're not already familiar with Figure 1.21, you should commit it to memory.

cos(θ) = adjacent / hypotenuse

sin(θ) = opposite / hypotenuse

tan(θ) = opposite / adjacent

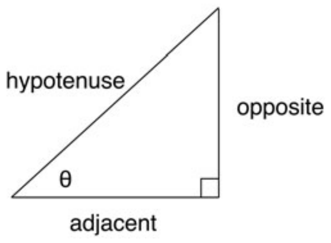


Figure 1.21 Sine, cosine, and tangent

You can also think of sine and cosine in terms of the X and Y coordinates of a circle, as illustrated in Figure 1.22.

Given the radius of a circle and a counterclockwise angle from 0 degrees, you can calculate the corresponding X and Y coordinates on the circumference of the circle by multiplying the radius times the cosine of the angle, and multiplying the radius by the sine of the angle, respectively.

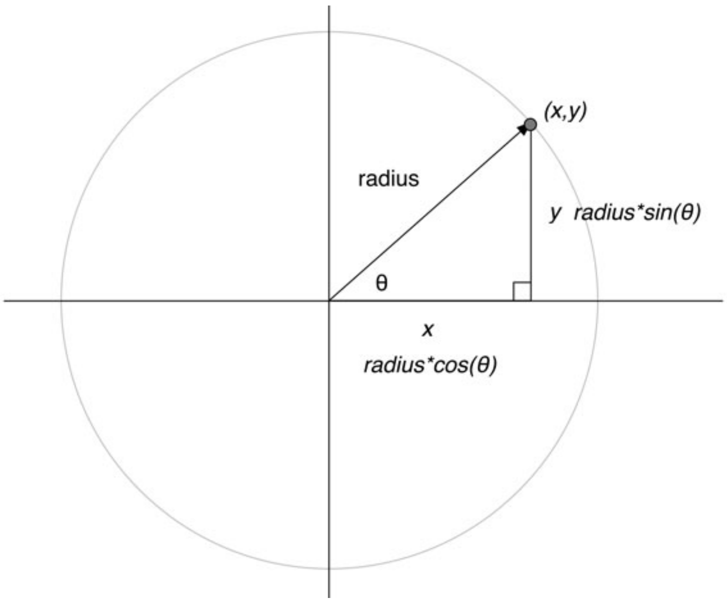


Figure 1.22 Radius, x, and y

NOTE: Soak a toe, ah!

One of many ways to remember how to derive sine, cosine, and tangent from a right triangle: SOHCAHTOA. SOH stands for sine, opposite, hypotenuse; CAH stands for cosine, adjacent, hypotenuse; and TOA is tangent, opposite, adjacent.

1.11.3 Vectors

The two-dimensional vectors that we use in this book encapsulate two values: direction and magnitude; they are used to express all sorts of physical characteristics, such as forces and motion.

In Chapter 8, "Collision Detection," we make extensive use of vectors, so in this section we discuss the fundamentals of vector mathematics. If you're not interested in implementing collision detection, you can safely skip this section.

Near the end of Chapter 8 we explore how to react to a collision between two polygons by bouncing one polygon off another, as illustrated in Figure 1.23.

In Figure 1.23, the top polygon is moving toward the bottom polygon, and the two polygons are about to collide. The top polygon's incoming velocity and outgoing velocity are both modeled with vectors. The edge of the bottom

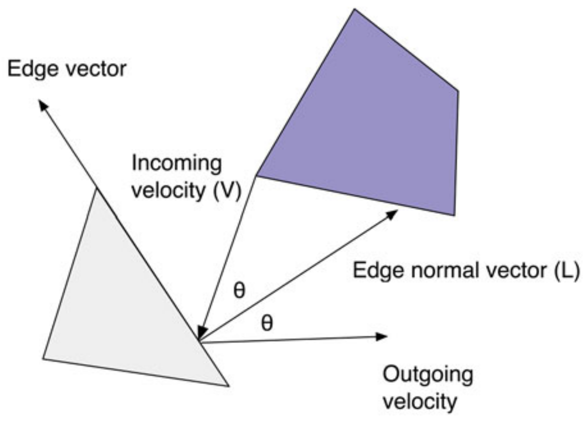


Figure 1.23 Using vectors to bounce one polygon off another

polygon with which the top polygon is about to collide is also modeled as a vector, known as a edge vector.

Feel free to skip ahead to Chapter 8 if you can't wait to find out how to calculate the outgoing velocity, given the incoming velocity and two points on the edge of the bottom polygon. If you're not familiar with basic vector math, however, you might want to read through this section before moving to Chapter 8.

1.11.3.1 Vector Magnitude

Although two-dimensional vectors model two quantities—magnitude and direction—it's often useful to calculate one or the other, given a vector. You can use the Pythagorean theorem, which you may recall from math class in school (or alternatively, from the movie the Wizard of Oz), to calculate a vector's magnitude, as illustrated in Figure 1.24.

The Pythagorean theorem states that the hypotenuse of any right triangle is equal to the square root of the squares of the other two sides, which is a lot easier to understand if you look at Figure 1.24. The corresponding JavaScript looks like this:

```
var vectorMagnitude = Math.sqrt(Math.pow(vector.x, 2) +
                                Math.pow(vector.y, 2));
```

The preceding snippet of JavaScript shows how to calculate the magnitude of a vector referenced by a variable named `vector`.

Now that you know how to calculate a vector's magnitude, let's look at how you can calculate a vector's other quantity, direction.

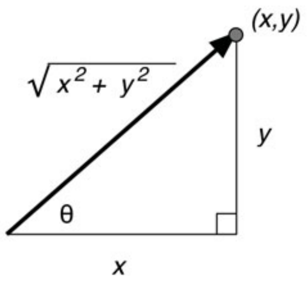


Figure 1.24 Calculating a vector's magnitude

1.11.3.2 Unit Vectors

Vector math often requires what's known as a unit vector. Unit vectors, which indicate direction only, are illustrated in Figure 1.25.

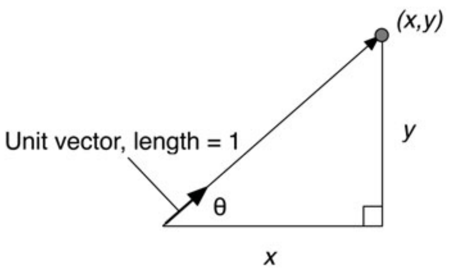


Figure 1.25 A unit vector

Unit vectors are so named because their magnitude is always 1 unit. To calculate a unit vector given a vector with an arbitrary magnitude, you need to strip away the magnitude, leaving behind only the direction. Here's how you do that in JavaScript:

```
var vectorMagnitude = Math.sqrt(Math.pow(vector.x, 2) +
                                Math.pow(vector.y, 2)),
    unitVector = new Vector();

    unitVector.x = vector.x / vectorMagnitude;
    unitVector.y = vector.y / vectorMagnitude;
```

The preceding code listing, given a vector named `vector`, first calculates the magnitude of the vector as you saw in the preceding section. The code then creates a new vector—see Chapter 8 for a listing of a `Vector` object—and sets that unit vector's X and Y values to the corresponding values of the original vector, divided by the vector's magnitude.

Now that you've seen how to calculate the two components of any two-dimensional vector, let's see how you combine vectors.

1.11.3.3 Adding and Subtracting Vectors

It's often useful to add or subtract vectors. For example, if you have two forces acting on a body, you can sum two vectors representing those forces together to calculate a single force. Likewise, subtracting one positional vector from another yields the edge between the two vectors.

Figure 1.26 shows how to add vectors, given two vectors named `A` and `B`.

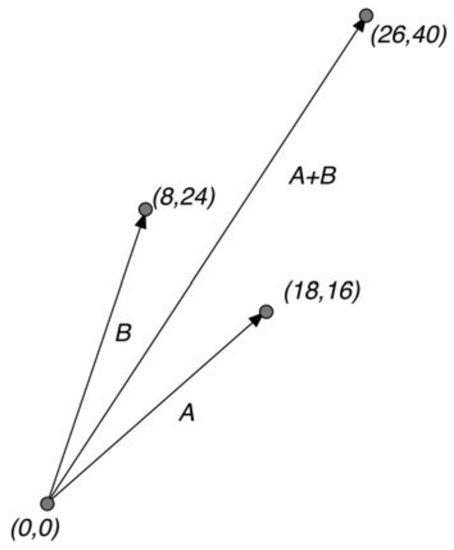


Figure 1.26 Adding vectors

Adding vectors is simple: You just add the components of the vector together, as shown in the following code listing:

```
var vectorSum = new Vector();

vectorSum.x = vectorOne.x + vectorTwo.x;
vectorSum.y = vectorOne.y + vectorTwo.y;
```

Subtracting vectors is also simple: you subtract the components of the vector, as shown in the following code listing:

```
var vectorSubtraction = new Vector();

vectorSubtraction.x = vectorOne.x - vectorTwo.x;
vectorSubtraction.y = vectorOne.y - vectorTwo.y;
```

Figure 1.27 shows how subtracting one vector from another yields a third vector whose direction is coincident with the edge between the two vectors. In Figure 1.27, the vectors `A-B` and `B-A` are parallel to each other and are also parallel to the edge vector between vectors `A` and `B`.

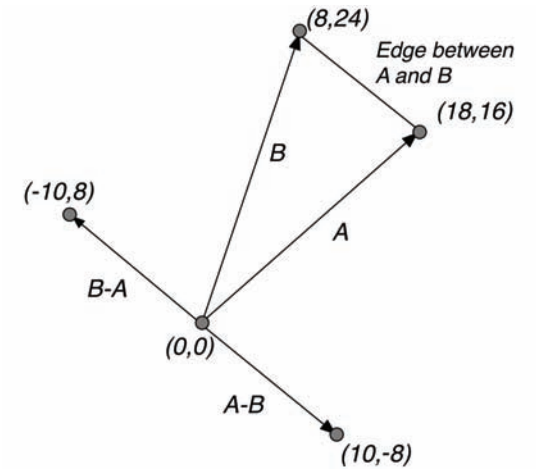


Figure 1.27 Subtracting vectors

Now that you know how to add and subtract vectors and, more importantly, what it means to do that, let's take a look at one more vector quantity: the dot product.

1.11.3.4 The Dot Product of Two Vectors

To calculate the dot product of two vectors you multiply the components of each vector by each other, and sum the values. Here is how you calculate the dot product for two two-dimensional vectors:

```
var dotProduct = vectorOne.x * vectorTwo.x + vectorOne.y * vectorTwo.y;
```

Calculating the dot product between two vectors is easy; however, understanding what a dot product means is not so intuitive. First, notice that unlike the result of adding or subtracting two vectors, the dot product is not a vector—it's what engineers refer to as a *scalar*, which means that it's simply a number. To understand what that number means, study Figure 1.28.

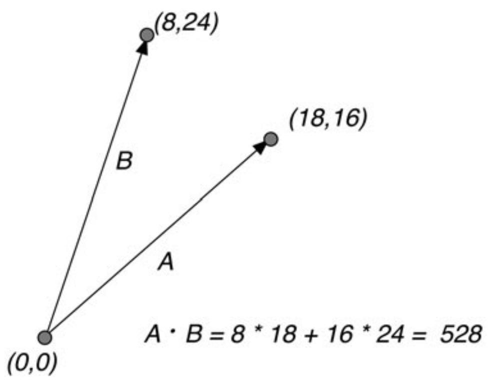


Figure 1.28 A positive dot product

The dot product of the two vectors in Figure 1.28 is 528. The significance of that number, however, is not so much its magnitude but the fact that it's greater than zero. That means that the two vectors point in roughly the same direction.

Now look at Figure 1.29, where the dot product of the two vectors is –528. Because that value is less than zero, we can surmise that the two vectors point in roughly different directions.

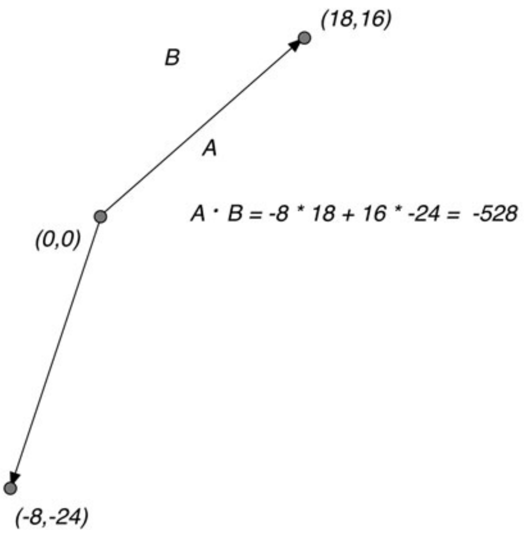


Figure 1.29 A negative dot product

The ability to determine whether or not two vectors point in roughly the same direction can be critical to how you react to collisions between objects. If a moving object collides with a stationary object and you want the moving object to bounce off the stationary object, you need to make sure that the moving object bounces *away* from the stationary object, and not toward the stationary object's center. Using the dot product of two vectors, you can do exactly that, as you'll see in Chapter 8.

That's pretty much all you need to know about vectors to implement collision detection, so let's move on to the last section in this brief math primer and see how to derive the equations from units of measure.

1.11.4 Deriving Equations from Units of Measure

As you will see in Chapter 5, motion in an animation should be time based, because the rate at which an object moves should not change with an animation's frame rate. Time-based motion is especially important for multiplayer games; after all, you don't want a game to progress more quickly for players with more powerful computers.

To implement time-based motion, we specify velocity in this book in terms of pixels per second. To calculate how many pixels to move an object for the current animation frame, therefore, we have two pieces of information: the object's velocity in *pixels per second*, and the current frame rate of the animation in *milliseconds per frame*. What we need to calculate is the number of *pixels per frame* to move any given object. To do that, we must derive an equation that has pixels per frame on the left side of the equation, and pixels per second (the object's velocity) and milliseconds per frame (the current frame rate) on the right of the equation, as shown in Equation 1.4.

$$\frac{\text{pixels}}{\text{frame}} \neq \frac{\text{X ms}}{\text{frame}} \times \frac{\text{Y pixels}}{\text{second}}$$

Equation 1.4 Deriving an equation for time-based motion, part I

In this *inequality*, X represents the animation's frame rate in milliseconds/frame, and Y is the object's velocity in pixels/second. As that inequality suggests, however, you cannot just multiply milliseconds/frame times pixels/second, because you end up with a nonsensical milliseconds-pixels/frame-seconds. So what do you do?

Recall the last rule we discussed in Section 1.11.1, "Solving Algebraic Equations," on p. 54 for solving algebraic equations: You can multiply or divide one or both sides of an equation by 1. Because of that rule, and because one second is equal to 1000 ms, and therefore 1 second / 1000 ms is equal to 1, we can multiply the right side of the equation by that fraction, as shown in **Equation 1.5**.

$$\frac{\text{pixels}}{\text{frame}} = \frac{\text{X ms}}{\text{frame}} \times \frac{\text{1 second}}{\text{1000 ms}} \times \frac{\text{Y pixels}}{\text{second}}$$

Equation 1.5 Deriving an equation for time-based motion, part 2

And now we are ready to move in for the kill because when you multiply two fractions together, *a unit of measure in the numerator of one fraction cancels out the same unit of measure in the denominator of the other fraction*. In our case, we cancel units of measure as shown in **Equation 1.6**.

$$\frac{\text{pixels}}{\text{frame}} = \frac{\text{X}\ \cancel{\text{ms}}}{\text{frame}} \times \frac{1\ \cancel{\text{second}}}{1000\ \cancel{\text{ms}}} \times \frac{\text{Y pixels}}{\cancel{\text{second}}}$$

Equation 1.6 Deriving an equation for time-based motion, part 3

Canceling those units of measure results in **Equation 1.7**.

$$\frac{\text{pixels}}{\text{frame}} = \frac{\text{X}}{\text{frame}} \times \frac{\text{Y pixels}}{1000}$$

Equation 1.7 Deriving an equation for time-based motion, part 4

Carrying out the multiplication results in the simplified equation, shown in **Equation 1.8**.

$$\frac{\text{pixels}}{\text{frame}} = \frac{\text{X} \times \text{Y}}{1000}$$

$$\text{X} = \text{frame rate in ms/frame}$$

$$\text{Y} = \text{velocity in pixels/second}$$

Equation 1.8 Deriving an equation for time-based motion, part 5

Whenever you derive an equation, you should plug some simple numbers into your equation to see if the equation makes sense. In this case, if an object is moving at 100 pixels per second, and the frame rate is 500 ms per frame, you can easily figure out, without any equations at all, that the object should move 50 pixels in that 1/2 second.

Plugging those numbers into **Equation 1.8** results in 500 × 100 / 1000, which equals 50, so it appears that we have a valid equation for any velocity and any frame rate.

In general, to derive an equation from variables with known units of measure, follow these steps:

1. Start with an inequality, where the result is on the left, and the other variables are on the right.
2. Given the units of measure on both sides of the equation, multiply the right side of the equation by one or more fractions, each equal to 1, whose units of measure cancel out the units of measure on the right side of the equation to yield the units of measure on the left side of the equation.
3. Cancel out the units of measure on the right side of the equation.
4. Multiply the fractions on the right side of the equation.
5. Plug simple values whose result you can easily verify into the equation to make sure the equation yields the expected value.

1.12 Conclusion

This chapter introduced you to the `canvas` element and its associated 2d context, and illustrated some essential features of that context, such as the difference between `canvas` element size and the size of the canvas's drawing surface.

From there we had a quick overview of your development environment, including browsers, consoles and debuggers, and performance tools.

Then we looked at the essentials of using a canvas, including fundamental drawing operations, event handling, saving and restoring the drawing surface, using HTML elements with a canvas, printing canvases, and using offscreen canvases. You will see the use of those essential features many times throughout this book, and you will use them yourself as you write Canvas-based applications.

Finally, we ended this chapter with a brief math primer, which you can consult as needed as you read the rest of the book.

In the next chapter we take a deep-dive into drawing in a canvas. In that chapter you will learn about the Canvas drawing API, and you'll see how to put that API to good use by implementing most of the features of a capable paint application.

This page intentionally left blank

Index