SVG

POCKET PRIMER

CD-ROM
Included!

OSWALD CAMPESATO

SVG

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and disc (the "Work"), you agree that this license grants permission to use the contents contained herein, including the disc, but does not give you the right of ownership to any of the textual content in the book / disc or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

Companion files that appear on the disc for this book can be downloaded from an FTP site by writing to the publisher at info@merclearning.com.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or disc, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

SVG

Pocket Primer

Oswald Campesato

MERCURY LEARNING AND INFORMATION
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *SVG Pocket Primer.*
ISBN: 978-1944534-59-2

Library of Congress Control Number: 2016951108

161718321 Printed in the United States of America

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

PREFACE

WHAT IS THE PRIMARY VALUE PROPOSITION FOR THIS BOOK?

This book endeavors to provide you with as much up-to-date information as possible that can be reasonably included in a book of this size. There are many unique features of this book, including one chapter that is dedicated to CSS3 (with 2D/3D graphics and animation and how to leverage SVG in CSS selectors), one chapter for charts and graphs in SVG, and SVG (with custom code samples).

Other significant features of this book include code samples that show you how to render 3D shapes (including 3D bar charts with animation), and how to create SVG filters for your HTML Web pages. In addition, the author has written multiple open source projects (links are provided toward the end of this Preface) containing literally thousands of code samples so that you can explore the capabilities of CSS3, SVG, HTML5 Canvas, and jQuery combined with CSS3.

THE TARGET AUDIENCE

This book is intended to reach an international audience of readers with highly diverse backgrounds in various age groups. While many readers know how to read English, their native spoken language is not English (which could be their second, third, or even fourth language). Consequently, this book uses standard English rather than colloquial expressions that might be confusing to those readers. As you know, many people learn by different types of imitation, which includes reading, writing, or hearing new material (yes, some basic videos are also available). This book takes these points into consideration in order to provide a comfortable and meaningful learning experience for the intended readers.

GETTING THE MOST FROM THIS BOOK

Some programmers learn well from prose, others learn well from sample code (and lots of it), which means that there's no single style that can be used for everyone.

Moreover, some programmers want to run the code first, see what it does, and then return to the code to delve into the details (and others use the opposite approach).

Consequently, there are various types of code samples in this book: some are short, some are long, and other code samples "build" from earlier code samples.

The goal is to show (and not just tell) you a variety of visual effects that are possible, some of which you might not find anywhere else. You benefit from this approach because you can pick and choose the visual effects and the code that creates those visual effects.

HOW WAS THE CODE FOR THIS BOOK TESTED?

The code samples in this book have been tested in a Google Chrome browser (version 52) on a Macbook Pro with OS X 10.11.4. Unless otherwise noted, no special browser-specific features were used, which means that the code samples ought to work in Chrome on other platforms, and also in other modern browsers. Exceptions are due to limitations in the cross-platform availability of specific features of SVG itself. Although the code also works in several earlier versions of Chrome on a Macbook Pro, you need to test the code on your platform and browser (especially if you are using Internet Explorer).

Another point to keep in mind is that all references to "Web Inspector" refer to the Web Inspector in Chrome, which differs from the Web Inspector in Safari. If you are using a different (but still modern) browser or an early version of Chrome, you might need to check online for the sequence of keystrokes that you need to follow to launch and view the Web Inspector. Navigate to this link for additional useful information: *http://benalman.com/projects/javascript-debug-console-log/*

WHY ARE THE SCREENSHOTS IN BLACK AND WHITE?

The black and white images are less costly than the original color images, and therefore their inclusion means that this book is available at a lower cost. However, the color images are available on the companion disc, along with supplemental code samples that render in color when you launch them in a browser.

WHAT DO I NEED TO KNOW FOR THIS BOOK?

The most important prerequisite is familiarity with HTML Web pages and JavaScript. If you want to be sure that you can grasp the material in this book,

glance through some of the code samples to get an idea of how much is familiar to you and how much is new for you.

WHAT DO I NEED TO UNDERSTAND THIS BOOK?

No prior knowledge of SVG is required in order to read this book. You do need an understanding of HTML (how to use basic elements) and some knowledge of JavaScript. If you encounter concepts that are unfamiliar, you can quickly learn them by reading one of the many online tutorials that explain those concepts. In other words, the gaps in our knowledge can easily be filled.

WHY DOES THIS BOOK HAVE 250 PAGES INSTEAD OF 500 PAGES?

This book is part of a Pocket Primer series whose books are between 200 and 250 pages. Second, the target audience consists of readers ranging from beginners to intermediate in terms of their knowledge of HTML and JavaScript. During the preparation of this book, every effort has been made to accommodate those readers so that they will be adequately prepared to explore more advanced features of SVG during their self study.

WHY SO MANY CODE SAMPLES IN THE CHAPTERS?

One of the primary rules of exposition of virtually any kind is "show, don't tell". While this rule is not taken literally in this book, it's the motivation for showing first and telling second. You can decide for yourself if show-first-then-tell is valid in this book by performing a simple experiment: when you see the code samples and the accompanying graphics effects in this book, determine if it's more effective to explain ("tell") the visual effects or to show them. If the adage "a picture is worth a thousand words" is true, then this book endeavors to provide both the pictures and the words.

DOESN'T THE COMPANION DISC OBVIATE THE NEED FOR THIS BOOK?

The companion disc contains all the code samples to save you time and effort from the error-prone process of manually typing code into an HTML Web page. In addition, there are situations in which you might not have easy access to companion disc. Furthermore, the code samples in the book provide explanations that are not available on the companion disc.

Finally, as mentioned earlier in this Preface, there are some introductory videos available that cover HTML5, CSS3, HTML5 Canvas, and SVG. Navigate to the publisher's website to obtain more information regarding their availability.

NOTE *All of the companion files for this title are available for downloading from FTP by writing to the publisher at info@merclearning.com.*

DOES THIS BOOK CONTAIN PRODUCTION-LEVEL CODE SAMPLES?

The primary purpose of the code samples in this book is to illustrate various features of SVG. Clarity has higher priority than writing more compact code that is more difficult to understand (and possibly more prone to bugs). If you decide to use any of the code in this book in a production website, you ought to subject that code to the same rigorous analysis as the other parts of your HTML Web pages.

OTHER RELATED BOOKS BY THE AUTHOR

1) HTML5 Canvas and CSS3:

http://www.amazon.com/HTML5-Canvas-CSS3-Graphics-Primer/dp/1936420341

2) jQuery, HTML5, and CSS3:

http://www.amazon.com/jQuery-HTML5-Mobile-Desktop-Devices/dp/1938549031

3) HTML5 Pocket Primer:

http://www.amazon.com/HTML5-Pocket-Primer-Oswald-Campesato/dp/1938549104

4) jQuery Pocket Primer:

http://www.amazon.com/dp/1938549147

CHAPTER 1

INTRODUCTION TO SVG

This chapter introduces you to SVG and presents SVG features in code samples that are defined in SVG documents. If you are comfortable with XML, then you will be at ease with SVG, because it's an XML "vocabulary" (i.e., SVG uses XML-based syntax). If you are new to XML, you can find many free online tutorials to learn about XML elements, attributes of XML elements, the meaning of a well-formed XML document, and a basic understanding of namespaces.

The first part of this chapter contains a short overview of SVG, along with a description of some of its features, followed by an example of the SVG coordinate system. The second part of this chapter shows you how to render line segments with the SVG `<line>` element and how to render rectangles with the SVG `<rect>` element.

The third part introduces you to the SVG `<path>`, which is probably the most powerful element in SVG. This section also discusses the SVG `<polygon>` element and the SVG `<polyline>` element. The fourth section contains simple code samples for creating shadow effects, rendering an SVG cube, rendering text with fonts, how to use superscripts and subscripts, WOFF (Web Open Font Format), and responsive SVG.

WHAT IS SVG?

SVG is an XML-based technology, with built-in support for 2D shapes. SVG also enables you to define linear gradients, radial gradients, filter effects, transforms (`translate`, `scale`, `skew`, and `rotate`), and animation effects using an XML-based syntax. Although SVG does not support 3D effects (which *are* supported in CSS3), SVG provides support for arbitrary polygons, elliptic arcs, and quadratic and cubic Bezier curves, none of which is available

in CSS3. One other interesting point: SVG is also an image format. For instance, there are advantages (discussed in Chapter 9) in using SVG instead of PNG files.

In Chapter 9, you will learn about the use of SVG in hybrid mobile applications, including Google AMP (Accelerated Mobile Pages) for rendering Web pages very quickly on mobile devices. Interestingly, AMP-compliant Web pages contain no custom JavaScript, but they do allow for inline CSS and support for many SVG elements.

Advantages of SVG

There are various benefits in using SVG for graphically rich and interactive websites:

- No degradation of shapes with any zoom factor
- No loss of accuracy
- Good compression ratio
- No loss of quality in text rendering
- Highly scalable
- Renders well on higher resolutions
- XML-based format for import/export of data

Keep in mind that multiple assets are required for non-vector images (e.g., for mobile applications), whereas the scalability of SVG means that you can use one SVG document for different resolutions. Some key SVG scenarios and use-cases include: engineering documents, graphs/charts/maps, and media and graphics. You will also find SVG in CAD software, Google maps, and scientific applications.

Tools with Support for SVG

Some tools that provide support for SVG:

- Adobe Illustrator
- Inkscape
- Sketch
- Sketska
- Ikivo (commercial)
- Beatware MobileDesigner (commercial)
- Amaya
- SVGEdit (editor written in SVG and HTML)

The tools in the preceding list vary in terms of price, feature set, and complexity. For example, Inkscape is a powerful tool, and you can find free online tutorials that explain how to use the features of Inkscape. Evaluate each of these tools to determine which tool is the best fit for your needs.

Testing Browser Support for SVG

SVG is supported in modern browsers, and also in IE8+ and in Android 2.4 and higher.

You can test for SVG support via Modernizr, as shown here:

```
if (!Modernizr.svg) {
  $(".logo img").attr("src", "images/logo.png");
}
```

Notice that the SVG code is replaced with a PNG file if the browser does not support SVG.

Another alternative involves a JavaScript snippet, as shown here:

```
<img src="image.svg" onerror="this.onerror=null; this.src='image.
png'">
```

The following website provides plenty of information regarding the support for SVG features in various browsers: *http://caniuse.com/svg-html5*.

SVG and HTML

There are several ways that you can reference SVG documents in an HTML Web page. For example, you can embed an `<svg>` element directly in an HTML Web page. Alternatively, you can embed an SVG document in an HTML `<embed>` element, an HTML `<img>` element, or an HTML `<object>` element. You can even convert an SVG document to base64 data and then reference that data in an `<object>` element. Later in this chapter you will see examples that use the preceding techniques in order to render SVG documents in HTML Web pages.

Limitations of SVG

SVG does not support the `<video>` element or the `<audio>` element, both of which are available in HTML5, or support for 3D graphics and animation effects.

Although SVG does not provide an explicit z-index, the z-index in SVG is determined by the order the element appears in the document: elements that are later in a document have a "higher" z-index. You can change the order in which elements appear in an HTML page by changing their order in the SVG document. Later in this chapter you will see some other techniques for changing the z-index of SVG elements.

SVG ELEMENTS FOR 2D SHAPES

SVG supports many 2D shapes, including:

- Line segments (SVG `<line>` element)
- Rectangles (SVG `<rect>` element)

- Circles (SVG <circle> element)
- Ellipses (SVG <ellipse> element)
- Elliptic arcs (SVG <path> element)
- Polygons (SVG <polygon> element)
- Polylines (SVG <polyline> element)
- Quadratic Bezier curves (SVG <path> element)
- Cubic Bezier curves (SVG <path> element).

You can also render "composite" 2D shapes that you can create using the SVG <path> element.

In addition to 2D shapes, SVG supports linear gradients and radial gradients that you can apply to 2D shapes in SVG documents.

The SVG <defs> element (which appears in an SVG document before rendering any 2D shapes) is an optional SVG element that enables you to define gradients. The shapes that you define later in your SVG documents can reference those gradients.

In addition, SVG provides the <pattern> element that can be included in the <defs> element. The <pattern> element is extremely powerful because it enables you to define an arbitrary set of shapes and gradients as a "fill template" that can be referenced in the fill attribute of other SVG elements in an SVG document. In Chapter 2, you will see examples of how to use the SVG <pattern> element.

SVG supports transform effects, such as translate, rotate, scale, and skew. You can perform animation effects for some attributes (width, height, and color), and also create animation effects. Note that animation-related effects tend to be simpler to create (and easier to maintain) when you use JavaScript, and there are libraries that simplify the APIs. Finally, you can dynamically create SVG elements with JavaScript, and you can define mouse-related event handlers that invoke a JavaScript function to handle those events with custom code.

A "Generic" SVG Document

Listing 1.1 displays the contents of Generic1.svg, which illustrates the placement of several circles in the SVG coordinate system.

LISTING 1.1: Generic1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <!-- optional section -->
 <defs>
    <!--
     <linearGradient> elements
     <radialGradient> elements
```

```
    <pattern> elements
    <clipPath> elements
    <maskPath> elements
    other definitions
   -->
</defs>

<g id="g1">
  <!
    sample elements:
    <rect id="rect1" ...>
    <ellipse id="ellipse1" ...>
    <path id="path1" ...>
    <use xlink:href="#rect1" ...>
  -->
</g>

<!-- optional section -->
<g id="g2">
  <!-- more elements -->
</g>
```

Listing 1.1 starts with an `<xml>` element that declares the document as an XML document. The `DOCTYPE` declaration and the `<svg>` element (and the matching `</svg>` tag at the end of Listing 1.1) are essentially "static content" in the sense that you always need to include them in every SVG document. All the other sections in Listing 1.1 are optional, but obviously you need to include some of those sections in order to render some sort of graphics image.

NOTE *Nothing is displayed when you launch Listing 1.1 in a browser.*

There are various components that you can include in the SVG `<defs>` element. Chapter 2 discusses linear and radial gradients, and Chapter 3 discusses how to define patterns, clip paths, and masks.

Immediately following the `<defs>` element is a `<g>` ("group") element, and in this chapter you will learn how to create some of the 2D shapes that are listed in this `<g>` element. In addition to specifying consecutive `<g>` elements, you can also specify nested `<g>` elements. This functionality is very useful when you need to create component-like "containers" that can be referenced in other sections of SVG code. As you will see in Chapter 4, you can also apply SVG transforms to `<g>` elements.

Listing 1.1 is intended to give you a very basic overview of the structure of SVG documents, and through exposure to many types of SVG documents you will become better acquainted with the functionality that is available in SVG.

What about D3.js?

As you will see in Chapter 6, the D3.js open source toolkit is an excellent alternative to "pure" SVG. D3.js is a JavaScript-based toolkit that provides a layer of abstraction over SVG. Hence, you can leverage your knowledge of SVG when you use D3.js instead of "pure" SVG. In addition, you can create

HTML Web pages that combine D3.js, SVG, HTML5 Canvas, and CSS3. The D3.js toolkit enables you to make Ajax requests for SVG-based documents, and also access the data in CSV files.

THE SVG COORDINATE SYSTEM

The origin for the SVG coordinate system is the upper-left corner of the screen. The horizontal axis is positive in a left-to-right fashion (just like the horizontal number system), which is just as you would expect, whereas the vertical axis is positive in the *downward* direction.

Listing 1.2 displays the contents of `Coordinates1.svg` that illustrates the placement of four circles that can help you visualize the SVG coordinate system.

LISTING 1.2: Coordinates1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <circle cx="50"  cy="50"  r="5" fill="red" />
   <circle cx="250" cy="50"  r="5" fill="green" />
   <circle cx="50"  cy="150" r="5" fill="yellow" />
   <circle cx="250" cy="150" r="5" fill="blue" />
 </g>
</svg>
```

Listing 1.2 contains boilerplate code followed by a `<g>` element that contains four `<circle>` elements, each of which has a different color. Chapter 4 contains more information about circles (and ellipses and elliptic arcs), but as you can see, the definition of a `<circle>` element is straightforward and convenient for the purpose of illustrating the SVG coordinate system.

When you launch Listing 1.2 in a browser, the different colors will help you match each circle in the browser with its corresponding code in Listing 1.2.

Figure 1.1 displays a screenshot of `Coordinates1.svg` in a Chrome browser on a MacBook Pro.

THE SVG `<LINE>` ELEMENT

The SVG `<line>` element renders a line segment whose endpoints are `(x1,y1)` and `(x2,y2)`. A simple example is here:

```
<line x1="20" y1="20" x2="100" y2="150".../>
```

Listing 1.3 displays the contents of `LineSegments1.svg`, which illustrates how to render line segments in SVG.

FIGURE 1.1: The SVG coordinate system.

LISTING 1.3: LineSegments1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <!-- left-side figures -->
   <line x1="20" y1="20" x2="220" y2="20"
         stroke="blue" stroke-width="4"/>

   <line x1="20" y1="40" x2="220" y2="40"
         stroke="red" stroke-width="10"/>

   <line x1="20" y1="60" x2="220" y2="60"
         stroke-dasharray="8 2 4 2"
         stroke="blue" stroke-width="4"/>

   <line x1="20" y1="80" x2="220" y2="80"
         stroke-dasharray="8 2 4 2"
         stroke-dashoffset="6"
         stroke="blue" stroke-width="4"/>
 </g>
</svg>
```

Listing 1.3 contains several SVG `<line>` elements, each of which specifies two endpoints as described earlier, as well as a value for the `stroke` attribute and a value for the `stroke-width` attribute.

The third SVG `<line>` element shows an example of the optional `stroke-dasharray` attribute, whose value is a set of integers that represent an on-off pattern of short line segments that created a "dashed" effect. The fourth SVG `<line>` element shows an example of the optional `stroke-dasharray` attribute whose value specifies the distance into the dash pattern to start the dash.

Another optional attribute is the `transform` attribute, which is discussed in Chapter 3.

Figure 1.2 displays the result of rendering `LineSegments1.svg` in a browser.

FIGURE 1.2: SVG line segments.

THE SVG <RECT> ELEMENT

This section shows you how to render rectangles using the SVG `<rect>` element whose syntax looks like this:

```
<rect width="200" height="50" x="20" y="50".../>
```

The SVG `<rect>` element renders a rectangle whose width and height are specified in the `width` and `height` attributes. The upper-left vertex of the rectangle is specified by the point with coordinates `(x,y)`.

Listing 1.4 displays the contents of `Rectangles1.svg`, which illustrates how to render line segments and rectangles.

LISTING 1.4: Rectangles1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <!-- left-side figures -->
   <line x1="20" y1="20" x2="220" y2="20"
         stroke="blue" stroke-width="4"/>

   <line x1="20" y1="40" x2="220" y2="40"
         stroke="red" stroke-width="10"/>

   <rect width="200" height="50" x="20" y="70"
         fill="red" stroke="black" stroke-width="4"/>
 </g>
</svg>
```

The first SVG `<line>` element in Listing 1.4 specifies the color `blue` and a `stroke-width` (i.e., line width) of `4`, whereas the second SVG `<line>` element specifies the color `red` and a `stroke-width` of `10`.

Notice that the first SVG `<rect>` element renders a rectangle that looks the same (except for the color) as the second SVG `<line>` element, which shows that it's possible to use different SVG elements to render a rectangle (or a line segment).

Two optional attributes for the SVG `<rect>` element are `rx` and `ry` for rendering rectangles with rounded corners.

THE SVG <PATH> ELEMENT

The SVG <path> element is the most flexible and powerful element that can render arbitrarily complex shapes based on a "concatenation" of other SVG elements.

An SVG <path> element contains a d attribute that specifies the points in the desired path, as shown here:

```
<path d="M20,150 l200,0 l0,50 l-200,0 z"
      fill="blue" stroke="red" stroke-width="4"/>
```

Listing 1.5 displays the contents of Path1.svg, which illustrates how to render line segments, rectangles, and paths.

LISTING 1.5: Path1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <!-- left-side figures -->
   <line x1="20" y1="20" x2="220" y2="20"
         stroke="blue" stroke-width="4"/>

   <line x1="20" y1="40" x2="220" y2="40"
         stroke="red" stroke-width="10"/>

   <rect width="200" height="50" x="20" y="70"
         fill="red" stroke="black" stroke-width="4"/>

   <path d="M20,150 l200,0 l0,50 l-200,0 z"
         fill="blue" stroke="red" stroke-width="4"/>

   <!-- right-side figures -->
   <path d="M250,20 l200,0 l-100,50 z"
         fill="blue" stroke="red" stroke-width="4"/>

   <path d="M300,100 l100,0 l50,50 l-50,50 l-100,0 l-50,-50 z"
         fill="yellow" stroke="red" stroke-width="4"/>
 </g>
</svg>
```

Listing 1.5 contains SVG <line> elements and <rect> elements from the previous two sections, along with several SVG <path> elements. The first SVG <path> element in Listing 1.5 contains the following d attribute:

```
d="M20,150 l200,0 l0,50 l-200,0 z"
```

This is how to interpret the contents of the d attribute:

- Move to the absolute location point (20,150)
- Draw a horizontal line segment 200 pixels to the right

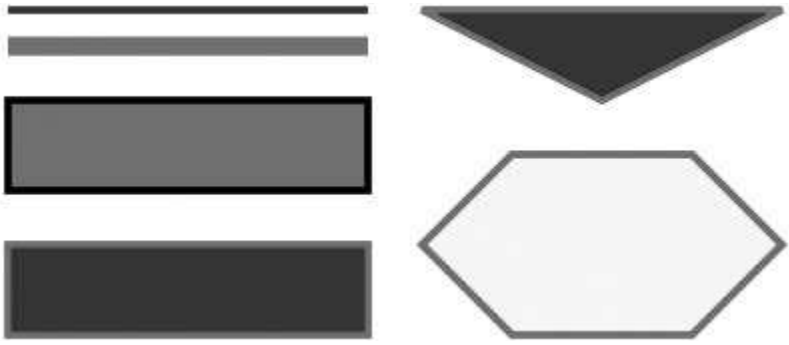

FIGURE 1.3: SVG lines, rectangles, and paths.

- Draw a line segment by moving `10` pixels to the right and `50` pixels down
- Draw a horizontal line segment by moving `200` pixels toward the left
- Draw a line segment to the initial point (specified by `z`)

Similar comments apply to the other two SVG `<path>` elements in Listing 1.5. One thing to keep in mind is that uppercase letters (`C`, `L`, `M`, and `Q`) refer to absolute positions whereas lowercase letters (`c`, `l`, `m`, and `q`) refer to relative positions with respect to the element that is to its immediate left. Experiment with the code in Listing 1.5 by using combinations of lowercase and uppercase letters to gain a better understanding of how to create different visual effects.

Figure 1.3 displays the result of rendering the SVG document `BasicShapes1.svg`.

CREATING A PYRAMID IN SVG

Listing 1.6 displays the contents of `Pyramid1.svg`, which illustrates how to render a pyramid using two SVG `<path>` elements.

LISTING 1.6: Pyramid1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <!-- front face -->
   <path d="M50,200 L150,200 L100,20 z"
         fill="blue" stroke="red" stroke-width="2"/>

   <!-- right face -->
   <path d="M100,20 L150,200 L175,175 z"
         fill="yellow" stroke="red" stroke-width="2"/>
 </g>
</svg>
```

FIGURE 1.4: A pyramid in SVG.

Listing 1.6 is straightforward: the first `<path>` element renders the front face of the pyramid, and the second `<path>` element renders the right face of the pyramid.

Figure 1.4 displays a screenshot of the SVG document `Pyramid1.svg` in a Chrome browser on a MacBook Pro.

THE SVG <POLYGON> ELEMENT

The SVG `<polygon>` element enables you to render arbitrary polygonal shapes that can be convex or concave.

Listing 1.7 displays the contents of `Polygons1.svg`, which illustrates how to render polygonal shapes in SVG.

LISTING 1.7: Polygons1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <g>
   <polygon points="250,20 200,80 300,350 z"
            fill="blue" stroke="red" stroke-width="4"/>

   <polygon points="300,100 100,80 50,50 100,250 z"
            fill="yellow" stroke="red" stroke-width="4"/>
 </g>
</svg>
```

Listing 1.7 contains boilerplate code with an SVG `<svg>` root element that also contains an SVG `<g>` element. The SVG `<g>` element contains an SVG `<polygon>` element whose points attribute specifies three points, so you might think this renders a triangle. However, the `points` attribute contains the `z` attribute, which "instructs" SVG to connect the third point with the first

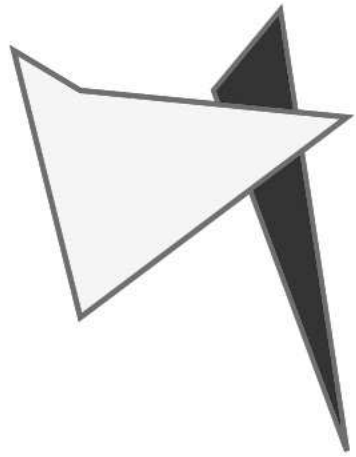

FIGURE 1.5: SVG polygons.

point; since these two points are different, the result is a four-sided figure (also known as a quadrilateral). The second `<polygon>` element inside the `<g>` element also renders a yellow polygon that ought to have four sides. However, the polygon is concave, which obscures the four sides and renders a triangular shape. Keep this point in mind when you are defining concave polygons that have a fill attribute.

Figure 1.5 displays the result of rendering the SVG document `Polygons1.svg`.

THE SVG `<POLYLINE>` ELEMENT

The SVG `<polyline>` element enables you to render arbitrary sets of line segments.

Listing 1.8 displays the contents of `Polyline1.svg`, which illustrates how to render a set of contiguous line segments.

LISTING 1.8: Polyline1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 <!-- original polygons -->
 <g>
   <polygon points="250,20 200,80 300,350 z"
            fill="none" stroke="red" stroke-width="4"/>

   <polygon points="300,100 100,80 50,50 100,250 z"
            fill="none" stroke="red" stroke-width="4"/>
 </g>
```

```
<!-- shift polygons as polylines -->
<g transform="translate(50,50)">
  <polyline points="250,20 200,80 300,350 z"
            opacity="0.8"
             fill="blue" stroke="red" stroke-width="4"/>

  <polyline points="300,100 100,80 50,50 100,250 z"
             opacity="0.8"
            fill="red" stroke="green" stroke-width="4"/>
</g>
</svg>
```

Listing 1.8 contains boilerplate code with an SVG `<svg>` root element that also contains two SVG `<g>` elements. The contents of the first SVG `<g>` element are the same as the contents of the `<g>` element in Listing 1.7. The second `<g>` element in Listing 1.8 contains two `<polyline>` elements whose `points` attributes are identical to the points attributes of the `<polygon>` elements in the first `<g>` element. In addition, the `<polyline>` elements contain an opacity attribute that "dims" the rendered shape. Notice also that the second `<g>` element contains a `transform` attribute that specifies the `translate()` transform, as shown here:

```
<g transform="translate(50,50)">
```

Although we have not discussed SVG transforms, the preceding code snippet is straightforward: the snippet `transform(50,50)` "translates" (or shifts) the contents of the `<g>` element 50 units to the right and 50 units downward from the origin.

Figure 1.6 displays the result of rendering the SVG document `Polyline1.svg` in a browser.

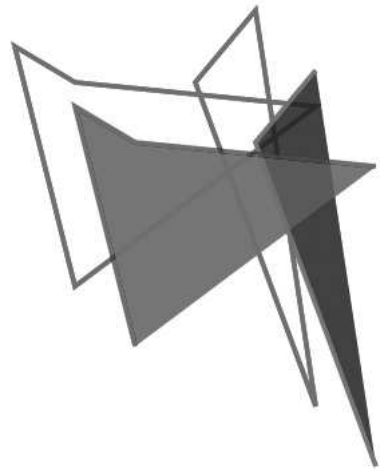

FIGURE 1.6: A set of contiguous line segments in SVG.

SIMPLE SHADOW EFFECTS IN SVG

Listing 1.9 displays the contents of `ShadowRect1.svg`, which illustrates how to create a shadow effect with two rectangles.

LISTING 1.9: ShadowRect1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

   <!-- draw shadow rectangle -->
   <g transform="translate(60,60)">
   <rect
      x="0" y="0" width="150" height="100"
      style="fill:black;"/>
   </g>

   <!-- draw colored rectangle -->
   <g transform="translate(50,50)">
   <rect
      x="0" y="0" width="150" height="100"
      style="fill:red;"/>
   </g>
</svg>
```

Listing 1.9 contains boilerplate code with an SVG `<svg>` root element that also contains two SVG `<g>` elements. The first SVG `<g>` element is shifted `60` units to the right and `60` units downward because of the `transform` attribute in the `<g>` element. As you can see, the `<g>` element contains a black rectangle. The second SVG `<g>` element is shifted `50` units to the right and `50` units downward because of the `transform` attribute in the `<g>` element. However, the second `<g>` element contains a red rectangle. The result is a red rectangle that has a black rectangle as a "shadow" effect.

SVG elements that support the `stroke` attribute also support some optional attributes, including `stroke-linecap`, `stroke-linejoin`, and `stroke-dasharray`. The `stroke-linecap` attribute can be `butt`, `square`, or `round`. The `stroke-linejoin` attribute can be `bevel`, `miter`, or `round`. You can find online examples that discuss the differences among these optional attributes and how to use them in SVG.

Figure 1.7 displays a screenshot of rendering `ShadowRect1.svg` in a browser.

RENDERING A CUBE IN SVG

The SVG `<polygon>` element contains a polygon attribute in which you can specify points that represent the vertices of a polygon. The SVG

FIGURE 1.7: SVG rectangle and a shadow effect.

<polygon> element is most useful when you want to create polygons with an arbitrary number of sides, but you can also use this element to render line segments and rectangles.

The syntax of the SVG <polygon> element looks like this:

```
<polygon path="specify a list of points" fill="…" />
```

Listing 1.10 displays the contents of SvgCube1.svg, which illustrates how to render a cube in SVG.

LISTING 1.10: SvgCube1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

   <!-- top face (counter clockwise) -->
   <polygon fill="red"
            points="50,50 200,50 240,30 90,30"/>

   <!-- front face -->
   <rect width="150" height="150" x="50" y="50"
         fill="blue"/>

   <!-- right face (counter clockwise) -->
   <polygon fill="yellow"
            points="200,50 200,200 240,180 240,30"/>
</svg>
```

Listing 1.10 contains an SVG <g> element contains the three faces of a cube; an SVG <polygon> element renders the top face (which is a parallelogram), an SVG <rect> element renders the front face, and another SVG <polygon> element renders the right face (which is also a parallelogram). The three faces of the cube are rendered with the linear gradient and the two radial gradients defined in the SVG <defs> element (not shown in Listing 1.10).

Figure 1.8 displays the result of rendering the SVG document SvgCube1.svg.

FIGURE 1.8: An SVG gradient cube.

WORKING WITH TEXT IN SVG

This section contains code samples that illustrate various effects that you can apply to text strings.

Listing 1.11 displays the contents of a portion of `Text1.svg` that illustrates how to render text strings in SVG.

LISTING 1.11: Text1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <g>
    <text id="text1" x="20" y="50"
          fill="red" font-size="48">
        Normal Text
    </text>

    <text id="text2" x="20" y="100" font-size="48"
          fill="red" stroke="black" stroke-width="3">
        Shadow Text
    </text>

    <text id="text3" x="20" y="150"
          font-size="48" font-family="serif"
          style="font-family: impact, georgia, times, serif;
          font-weight: normal; font-style: normal"
          fill="red" stroke="black" stroke-width="3">
        Font Family Shadow Text
    </text>
  </g>
</svg>
```

Listing 1.11 contains boilerplate code with an SVG `<svg>` root element with one SVG `<g>` element that in turn contains three `<text>` elements.

FIGURE 1.9: Rendering text in SVG.

All three SVG `<text>` elements contain `x` and `y` attributes that specify the absolute start position (relative to the origin) of each text string. The first two `<text>` elements specify red text with a font size of `48`. In addition, the second `<text>` element specifies a stroke color of black and a stroke width of 3. The third `<text>` element contains a `font-family` attribute and a `style` attribute that contains multiple attributes.

Figure 1.9 displays the result of rendering `Text1.svg` in a browser.

TEXT AND FONTS IN SVG

This section contains code samples that illustrate various effects that you can apply to text strings.

Listing 1.12 displays the contents of a portion of `Font2.svg` that illustrates how to render text strings in SVG.

LISTING 1.12: Font2.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%">
 <g fill="blue">
    <text x = "10" y = "25" font-size = "12">
        <tspan x = "10" dy = "20" font-family = "serif">
           The first sentence with a font
        </tspan>
        <tspan x = "10" dy = "20" font-family = "cursive">
           The second sentence with a font
        </tspan>
        <tspan x = "10" dy = "20" font-family = "fantasy">
           The third sentence with a font
        </tspan>
        <tspan x = "10" dy = "20" font-family = "monospace">
           The fourth sentence with a font
        </tspan>
    </text>
 </g>
</svg>
```

Listing 1.12 contains one SVG `<g>` element that contains a single `<text>` element that also specifies four `<tspan>` elements. Each `<tspan>` element

contains values for the x and y attributes that specify the start position of its associated text string that is rendered according to the value of its font-family attribute.

ITALICIZED TEXT IN SVG

This section contains code samples that illustrate various effects that you can apply to text strings.

Listing 1.13 displays the contents of a portion of Italicized1.svg that illustrates how to render italicized text strings in SVG.

LISTING 1.13: Italicized1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <g fill = "blue">
    <text x = "10" y = "25" font-size = "20">
        <tspan x = "10" y = "20" font-style = "italic">
            Italic font-style
        </tspan>
        <tspan x = "10" y = "80" font-style = "normal">
            Normal font-style
        </tspan>
        <tspan x = "10" y = "140"
               font-style = "italic" font-weight="bold">
            Italic font-style (bold)
        </tspan>
    </text>
  </g>
</svg>
```

Listing 1.13 contains one SVG <g> element that contains a single <text> element that also specifies three <tspan> elements. Each <tspan> element contains values for the x and y attributes that specify the start position of its associated text string that is rendered according to the value of its font-style attribute.

SUPERSCRIPTS AND SUBSCRIPTS IN TEXT

This section contains code samples that illustrate various effects that you can apply to text strings.

Listing 1.14 displays the contents of a portion of SuperSubscripts1.svg that illustrates how to render superscripts and subscripts in text strings in SVG.

LISTING 1.14: SuperSubscripts1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <g fill = "navy">
    <text x = "10" y = "25" font-size = "20">
        <tspan>
            e = mc
            <tspan baseline-shift = "super">
                2
            </tspan>
        </tspan>
        <tspan x = "10" y = "60">
            X
            <tspan baseline-shift = "sub">
                i+2
            </tspan>
            =X
            <tspan baseline-shift = "sub">
                i
            </tspan>
            + X
            <tspan baseline-shift = "sub">
                i+1
            </tspan>
        </tspan>
    </text>
  </g>
</svg>
```

Listing 1.14 contains one SVG <g> element that contains a single <text> element that also specifies two <tspan> elements. Both <tspan> elements contain additional nested <tspan> elements with a baseline-shift attribute: the value super specifies a superscript and the value sub specifies a subscript.

THE SVG VIEWBOX ATTRIBUTE

The viewBox attribute enables you to "stretch" the rendered graphic so that it fits a container element. The viewBox attribute consists of a list of four numbers: min-x, min-y, width, and height, separated by whitespace and/or a comma. These four numbers specify a rectangle in "user space," taking into account the preserveAspectRatio attribute.

The following is some terminology to clarify the preceding sentence. First, the viewport coordinate system is also called "viewport space" and the "user coordinate system" is also called user space.

Second, for elements that support that attribute (except for the <image> element), preserveAspectRatio applies when a value has been provided for viewBox on the same element. For these elements, if attribute viewBox is not provided, then preserveAspectRatio is ignored. For <image> elements, preserveAspectRatio indicates how referenced images should be fitted with respect to the referenced rectangle and whether the aspect ratio

of the referenced image should be preserved with respect to the current user coordinate system.

Keep in mind the (possibly counterintuitive) effect of a `viewbox` attribute: a smaller rectangle "magnifies" the graphics effect, whereas a larger rectangle "shrinks" the graphics effect.

Listing 1.15 displays the contents of `Viewbox1.svg`, which illustrates how to use the `viewbox` attribute in an SVG document.

LISTING 1.15: Viewbox1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     viewBox="0 0 100 100"
     width="100%" height="100%">
 <g>
   <line x1="20" y1="20" x2="220" y2="20"
         stroke="blue" stroke-width="4"/>

   <line x1="20" y1="40" x2="220" y2="40"
         stroke="red" stroke-width="10"/>

   <line x1="20" y1="60" x2="220" y2="60"
         stroke-dasharray="6 2 4 2"
         stroke="blue" stroke-width="4"/>
 </g>
</svg>
```

Listing 1.15 shows you the `viewbox` attribute in bold, and its value is `0 0 100 100`. Replace the current value of the `viewbox` attribute with `0 0 200 200` (or another set of four values of your choice) and you will see the difference in the rendered graphics.

SVG IN AN <EMBED> ELEMENT

In the previous section, you learned how to render a cube using "pure" SVG. You can also use the `<embed>` element in order to reference an SVG document. Listing 1.15 displays the contents of `SvgEmbed1.html`, which references `SvgCube1.svg` in an `<embed>` element.

LISTING 1.15: SvgEmbed1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>An SVG Embed Example</title>
</head>
```

```
<body>
  <div id="chart">
    <embed id="graphics" src="SvgCube1.svg"
           width="800" height="500" type="image/svg+xml">
    </embed>
  </div>
</body>
</html>
```

Listing 1.15 contains boilerplate code and an `<embed>` element that specifies the SVG document `SvgCube1.svg` as the value of its `src` attribute, along with values for the width, height, and type attributes.

SVG IN AN <OBJECT> ELEMENT

Listing 1.16 displays the contents of `SvgObject1.html`, which references the SVG document `SvgCube1.svg` in an HTML `<object>` element.

LISTING 1.16: SvgObject1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>SVG in an object Element</title>
</head>

<body>
  <object type="image/svg+xml" data="SvgCube1.svg" class="logo">
  SVG in an object Element
</object>
</body>
</html>
```

Listing 1.16 contains boilerplate code and an `<object>` element that specifies the SVG document `SvgCube1.svg` as the value of its `data` attribute, along with values for the class and type attributes.

SVG AS BASE64 DATA

Listing 1.17 displays the contents of `SvgAsBase64.html`, which references the SVG document `SvgCube1.svg` as base64-encoded data in an HTML `<object>` element.

LISTING 1.17: SvgAsBase64.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>SVG as base64 Data</title>
</head>
```

```
<body>
 <object type="image/svg+xml" data="[data] ">
    SVG as base64
 </object>
</body>
</html>
```

Listing 1.17 contains boilerplate code and an `<object>` element that specifies the SVG document `SvgCube1.svg` as the value of its `data` attribute, along with a value for the type attribute. The previous two code samples reference an SVG document, whereas Listing 1.17 specifies the encoded data in the `data` attribute.

If you have a MacBook, you can `base64` encode an SVG document with the following command that saves the encoded data in a file called `thedata`:

```
cat BasicShapes1.svg | base64  >thedata
```

Now replace the term `[data]` in Listing 1.17 with the contents of the file `thedata`, and you will see the same graphics rendered as the SVG code in `BasicShapes1.svg`.

The next section briefly discusses some available tools for converting SVG documents to base64 format.

Tools for Converting SVG to base64

Mobilefish is an online tool for converting files (not exceeding 100KB): *http://www.mobilefish.com/services/base64/base64.php*.

The `grunticon` tool will process a directory containing SVG and PNG files and generate CSS-based files in the form of SVG data urls and PNG data urls. A third option involves a CSS file with references to PNG files that are also automatically generated and placed in a directory.

The `iconizr` tool is a PHP command line tool for converting SVG images to a set of CSS icons, along with support for image optimization and Sass output.

A drag-and-drop online tool is here: *http://jpillora.com/base64-encoder*.

Two videos that discuss SVG optimization are here: *https://www.youtube.com/watch?v=iVzW3XuOm7E&feature=youtu.be* and *https://www.youtube.com/watch?v=1AdX8odLC8M&feature=youtu.be*.

Perform an Internet search for other conversion tools.

SVG IN AN <IMG> ELEMENT

The previous section shows you how to use the `<img>` element in order to reference an SVG document. Listing 1.18 displays the contents of `SvgAsImg1.html`, which references the SVG document `SvgCube1.svg` in an `<img>` element.

LISTING 1.17: SvgAsImg1.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8" />
  <title>SVG as img Elements</title>
</head>

<body>
  <img src="SvgCube1.svg" alt="SVG as an img Element" />
</body>
</html>
```

Listing 1.17 contains boilerplate code and an `<img>` element that specifies the SVG document `SvgCube1.svg` as the value of its `src` attribute. Chapter 2 contains the source code for `SvgCube1.svg`.

More information regarding SVG in an HTML `<img>` element is here: *http://caniuse.com/#feat=svg-img*.

THE "PAINTERS MODEL" AND THE Z-INDEX IN SVG

This section briefly mentions the painters model and also some techniques for working with the z-index in SVG.

The W3C Specification for SVG describes the painters model for SVG:

> SVG uses a "painters model" of rendering. Paint is applied in successive operations to the output device such that each operation paints over some area of the output device. When the area overlaps a previously painted area the new paint partially or completely obscures the old. When the paint is not completely opaque the result on the output device is defined by the (mathematical) rules for compositing described under Alpha Blending.

As mentioned earlier in this chapter, SVG does not support an explicit z-index, but there are several JavaScript-related techniques (either directly or with a JavaScript library) for changing the rendering of different SVG elements. These techniques have trade-offs in terms of the additional code that is required.

One technique is to use JavaScript in order to move an SVG element to a different location in the DOM, which can also be done with non-SVG elements in an HTML Web page.

Another approach is to use the open source `svg.js` toolkit that provides the `.front()` function for moving any element to the top, which is downloadable here: *https://github.com/wout/svg.js*.

A third technique is to place a set of SVG elements in a `<div>` element and then change the z-index of the `<div>` element.

The fourth technique (and somewhat advanced for this chapter) involves the SVG `<use>` element that references a previously defined SVG element, as shown here:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     style="width:100%; height: 100%">
    <circle id="c1" cx="50" cy="50" r="40" fill="green" />
    <rect id="r1" x="4" y="20" width="200" height="50" fill="blue" />
```

```
    <circle id="c2" cx="70" cy="70" r="50" fill="red" />
    <use id="use" xlink:href="#c1" />
</svg>
```

As you can see, the `<use>` element references the `<circle>` element whose `id` attribute is c1. Moreover, `<use>` element is the last element, which makes it the top-most element. You can select other SVG elements by changing the value of the `xlink:href` attribute.

USEFUL LINKS

An article with information about exporting SVG from some tools: *http://www.smashingmagazine.com/2014/11/styling-and-animating-svgs-with-css/*.

Some useful SVG-related Links for SVG software (free and commercial): *http://www.dmoz.org/Computers/Data_Formats/Graphics/Vector/SVG/Software/*.

An SVG plugin for Eclipse: *http://sourceforge.net/projects/svgplugin/*.

A nice historical summary of SVG is here: *http://blog.siliconpublishing.com/2015/12/the-fall-and-rise-of-svg/*.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The supplemental code samples provide you with tips and techniques for creating various visual effects in SVG, and you can select the code samples with effects that are useful for your custom code. The SVG document `Images1.svg` illustrates how to display PNG files in SVG and how to use opacity (discussed in a later chapter).

The following open source projects contain many examples of rendering 2D shapes in SVG: *https://github.com/ocampesato/svg-graphics* and *https://github.com/ocampesato/svg-filters-graphics*.

SUMMARY

This chapter started with examples of the SVG coordinate system, followed by samples of creating basic 2D shapes. You learned how to render line segments and rectangles in SVG. You also saw how to create simple shadow effects by rendering two shapes with different colors and rendering them in a slightly offset position to create the appearance of shading. Next, you learned how to render a 3D cube in SVG using the `<rect>` element and the `<polygon>` element. You also learned how to render a set of contiguous line segments with the `<polyline>` element.

CHAPTER 2

SVG GRADIENTS AND FILTERS

This chapter contains code samples that illustrate how to render 2D shapes using basic colors (the simplest technique), SVG gradients (moderate complexity), and SVG filters (the most complex). Basic colors consist of common names (such as red, green, and blue) as well as colors that are defined via a hexadecimal string. SVG supports both linear gradients and radial gradients that have counterparts in CSS3. However, keep in mind an important syntactic difference: SVG uses an XML-based syntax for defining gradients, whereas CSS3 uses a `property:value` syntax.

SVG also supports at least eighteen filter types, some of which are discussed in this chapter, with varying degrees of complexity. For example, the "output" of one SVG filter can act as the "input" of another filter; in other cases, the output from two SVG filters is required as the input for another SVG filter.

The first part of this chapter shows you how to work with basic colors (for example, `Red`, `(255,0,0)`, `(100%,0,0)`, `#FF0000`, `#F00`, and `#f00` all represent the color red). The second part of this chapter discusses SVG linear gradients and radial gradients, along with code samples that illustrate how to use these gradients in SVG. The third part of this chapter discusses SVG filters, such as Gaussian blur, emboss filters, and so forth. The filter-related code samples will show you some of the visual effects that are possible with SVG filters. However, the filter-related code samples do not provide a detailed explanation of the purpose of all the attributes for SVG `<filter>` elements (the latter is beyond the scope of this book).

WORKING WITH COLORS IN SVG

The SVG code samples in this book use `(R,G,B)` for representing colors. An `(R,G,B)` "triple" represents the `Red`, `Green`, and `Blue` components of a color. For instance, the triples `(255,0,0)`, `(255,255,0)`, and `(0,0,255)`

represent the colors Red, Yellow, and Blue. Other ways of specifying the color include: the hexadecimal pair of triples (FF,0,0) and (F,0,0); the decimal triple (100%,0,0); or the hexadecimal string #F00. You can also use (R,G,B,A), where the fourth component specifies the opacity, which is a decimal number between 0 (invisible) to 1 (opaque) inclusive.

However, there is also the HSL (Hue, Saturation, and Luminosity) color model for representing colors, where the first component is an angle between 0 and 360 (0 degrees is north), and the other two components are percentages between 0 and 100. For instance, (0,100%,50%), (120,100%,50%), and (240,100%,50%) represent the colors Red, Green, and Blue, respectively.

SVG GRADIENTS

SVG supports two types of gradients: linear gradients and radial gradients. Linear gradients can be horizontal, vertical, and diagonal. Linear gradients resemble a set of line segments whose colors vary as they "sweep" from a start position to an end position.

On the other hand, radial gradients consist of a set of circles that "emanate" from a start circle. In the simplest case, radial gradients are rendered in a set of concentric circles. As the radius of the circles increases, the color changes in accordance with its definition. Thus, as the circles expand radially (i.e., based on a change in consecutive values for the radius of the circles), their color changes in a linear fashion. If you can visualize connecting a line segment from the circle of radius 0 to the circle with the maximum radius, the color of that line segment looks like a linear gradient. This chapter contains code samples that will give you a better understanding of different types of SVG gradients.

SVG LINEAR GRADIENTS

A linear gradient is defined as an SVG <linearGradient> element that contains one or more <stop> elements that define the colors in the gradient.

Listing 2.1 displays the contents of LGradientRect1.svg, which illustrates how to define a linear gradient and also specify a linear gradient definition as the style attribute for an SVG <rect> element.

LISTING 2.1: LGradientRect1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
<defs>
   <linearGradient gradientUnits="objectBoundingBox"
                   id="linearGradient1"
                   spreadMethod="reflect"
                   x1="0.45" y1="0.45" x2="0.55" y2="0.55">
```

```
      <stop offset="0"  style="stop-color:yellow"/>
      <stop offset=".5" style="stop-color:blue"/>
      <stop offset="1"  style="stop-color:red"/>
   </linearGradient>

   <linearGradient gradientUnits="objectBoundingBox"
                   id="linearGradient2"
                   spreadMethod="pad"
                   x1="0.45" y1="0.45" x2="0.55" y2="0.55">
      <stop offset="0"  style="stop-color:yellow"/>
      <stop offset=".5" style="stop-color:blue"/>
      <stop offset="1"  style="stop-color:red"/>
   </linearGradient>

   <linearGradient gradientUnits="objectBoundingBox"
                   id="linearGradient3"
                   spreadMethod="repeat"
                   x1="0.45" y1="0.45" x2="0.55" y2="0.55">
      <stop offset="0"  style="stop-color:yellow"/>
      <stop offset=".5" style="stop-color:blue"/>
      <stop offset="1"  style="stop-color:red"/>
   </linearGradient>
</defs>

<g transform="translate(0,0)">
<rect x="20" y="20" width="300" height="100"
      style="fill:url(#linearGradient1)"/>

<rect x="20" y="150" width="300" height="100"
      style="fill:url(#linearGradient2)"/>

<rect x="20" y="280" width="300" height="100"
      style="fill:url(#linearGradient3)"/>
</g>
</svg>
```

Listing 2.1 contains an SVG `<defs>` element that defines three `<linearGradient>` elements, all of which contain multiple attributes as well as child `<stop>` elements. The `gradientUnits` attribute specifies the type of coordinate system that is used in the calculation of the gradient. The default value is `objectBoundingBox`, which uses the bounding box of the gradient to calculate the linear gradient, which will be horizontal, vertical, or diagonal. The other value is `userSpaceOnUse`, which uses the user coordinate system for the element referencing the gradient element via a "fill" or "stroke" property).

The `spreadMethod` attribute can have the value `reflect`, `pad`, and `repeat`, which specify different ways of rendering a gradient effect.

The `spreadMethod` determines the behavior of the gradient effect when the latter reaches its end, prior to filling in the object. When the value is `pad`, the final offset color is used to fill the rest of the object. When the value is `reflect`, the gradient continues with an inverse reflection, starting with the color offset at `100%` and moving backward to the color offset at `0%`. When the value is `repeat`, the gradient reinitializes to the start value and repeats itself.

The <linearGradient> element consists of one or more <stop> elements, where each element contains the attributes offset and stop-color. The offset represents the percentage "offset position," and the stop-color attribute specifies the color at that position.

Note that you can also specify the stop-opacity attribute in a <stop> element in definitions for linear and radial gradients, as shown here:

```
<stop offset=".8" stop-color="black" stop-opacity="0.5"/>
```

The second part of Listing 2.1 consists of an SVG <g> element with three child <rect> elements, each of which references one of the three <linearGradient> elements defined in the <defs> element.

Figure 2.1 displays the result of rendering LinearGradient1.svg in a browser.

SVG RADIAL GRADIENTS

A radial gradient is defined as an SVG <radialGradient> element that contains one or more <stop> elements that define the colors in the gradient.

Listing 2.2 displays the contents of Ripple4RadialGradient6.svg, which illustrates how to define two radial gradients and also reference those

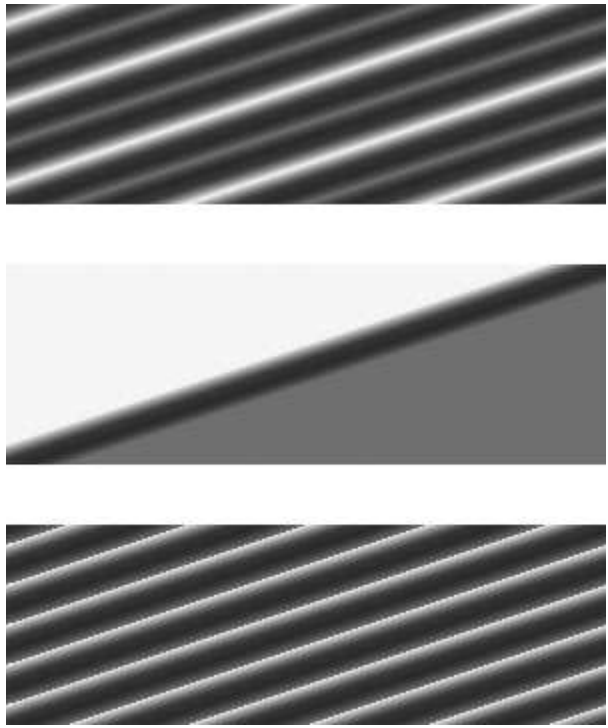

FIGURE 2.1: SVG linear gradient arcs.

radial gradients as the color for an SVG <ellipse> element and an SVG <rect> element.

LISTING 2.2: Ripple4RadialGradient6.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
    <defs>
      <radialGradient id="radialGradient1"
                      gradientUnits="userSpaceOnUse"
                      cx="400" cy="200" r="300"
                      fx="400" fy="200">
        <stop offset="0"    stop-color="white"/>
        <stop offset=".149" stop-color="red"/>
        <stop offset=".15"  stop-color="blue"/>
        <stop offset="1"    stop-color="black"/>
      </radialGradient>

      <radialGradient id="radialGradient2"
                      gradientUnits="userSpaceOnUse"
                      cx="400" cy="200" r="300"
                      fx="400" fy="200">
        <stop offset="0"    stop-color="yellow"/>
        <stop offset=".149" stop-color="red"/>
        <stop offset=".15"  stop-color="white"/>
        <stop offset="1"    stop-color="black"/>
      </radialGradient>
    </defs>

  <g transform="translate(50,20)">
    <ellipse fill="url(#radialGradient1)"
             stroke="black" stroke-width="8"
             cx="300" cy="100" rx="300" ry="100"/>

    <rect    fill="url(#radialGradient2)"
             stroke="black" stroke-width="8"
             x="0" y="200" width="600" height="300"/>
  </g>
</svg>
```

Listing 2.2 starts with an SVG <defs> element that defines two <radialGradient> elements, all of which contain multiple attributes as well as child <stop> elements.

As you already know, the cx, cy, and r attributes specify the center (cx,cy) and radius r of a circle that represents that "start circle" for the radial gradient. The cx, cy, and r attributes define the outermost circle and the fx and fy define the innermost circle for the radial gradient. An SVG <ellipse> element requires the mandatory attributes cx, cy, rx, and ry attributes that specify the center (cx,cy) , the major axis, and the minor axis,

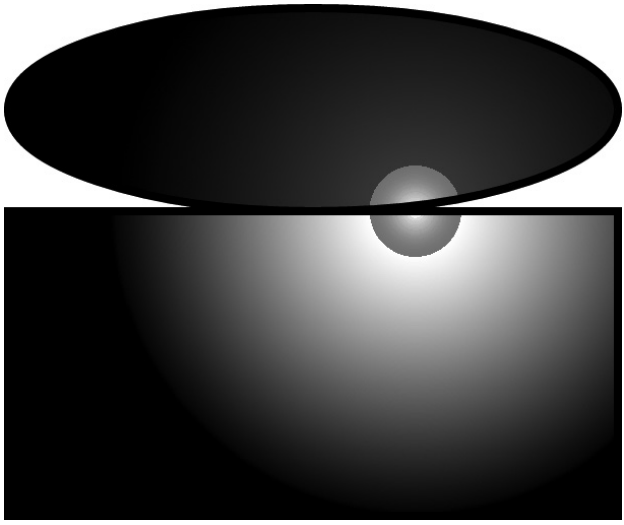

FIGURE 2.2: SVG radial gradient arcs.

respectively. Note that if you assign equal values to `rx` and `ry` you will render a circle.

The second part of Listing 2.2 consists of an SVG `<g>` element with an `<ellipse>` element and a `<rect>` element, both of which reference a `<radialGradient>` element defined in the `<defs>` element.

Figure 2.2 displays a screenshot of `Ripple4RadialGradient6.svg` in a browser.

RENDERING A CUBE WITH SVG RADIAL GRADIENTS

Listing 2.3 displays the contents of `SvgCube1Grad1.svg`, which illustrates how to render a cube in SVG with radial gradients.

LISTING 2.3: SvgCube1Grad1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
    <linearGradient id="pattern1"
                    x1="0%" y1="100%" x2="100%" y2="0%">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
    </linearGradient>
```

```
   <radialGradient id="pattern2">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
   </radialGradient>

   <radialGradient id="pattern3">
      <stop offset="0%"   stop-color="blue"/>
      <stop offset="40%"  stop-color="yellow"/>
      <stop offset="80%"  stop-color="red"/>
   </radialGradient>
  </defs>

   <!-- top face (counter clockwise) -->
   <polygon fill="url(#pattern1)"
            points="50,50 200,50 240,30 90,30"/>

   <!-- front face -->
   <rect width="150" height="150" x="50" y="50"
         fill="url(#pattern2)"/>

   <!-- right face (counter clockwise) -->
   <polygon fill="url(#pattern3)"
            points="200,50 200,200 240,180 240,30"/>
</svg>
```

Listing 2.3 starts with an SVG `<defs>` element that defines one `<linearGradient>` elements and two `<radialGradient>` elements.

The second part of Listing 2.3 consists of an SVG `<g>` element with three child elements: a `<polygon>` element, a `<rect>` element, and another `<polygon>` element that represent the three "faces" of a cube. The three faces also reference one of the gradient elements in the `<defs>` element.

Figure 2.3 displays the result of rendering `SvgCube1Grad1.svg` in a browser.

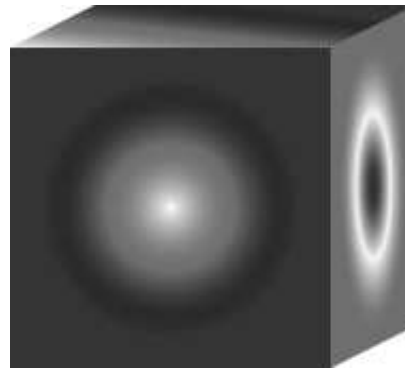

FIGURE 2.3: A Cube with radial gradients.

SVG GRADIENTS AND THE <PATH> ELEMENT

The SVG <path> element is a versatile element that supports many shapes. The SVG <path> element contains a d attribute for specifying the shapes in a path, as shown here:

```
<path d="specify a list of path elements" fill="…" />
```

Listing 2.4 displays the contents of BasicShapesLRG1.svg, which illustrates how to render 2D shapes with linear gradients and with radial gradients.

LISTING 2.4: BasicShapesLRG1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
    <linearGradient id="pattern1"
                    x1="0%" y1="100%" x2="100%" y2="0%">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
    </linearGradient>

   <radialGradient id="pattern2">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
   </radialGradient>
  </defs>

  <g>
   <ellipse cx="120" cy="80" rx="100" ry="50"
            fill="url(#pattern1)"/>

   <ellipse cx="120" cy="200" rx="100" ry="50"
            fill="url(#pattern2)"/>

   <ellipse cx="320" cy="80" rx="50" ry="50"
            fill="url(#pattern2)"/>

   <path d="M 505,145 v -100 a 250,100 0 0,1 -200,100"
         fill="black"/>

   <path d="M 500,140 v -100 a 250,100 0 0,1 -200,100"
         fill="url(#pattern1)"
         stroke="black" stroke-thickness="8"/>
```

```
   <path d="M 305,165 v  100 a 250,100 0 0,1  200,-100"
         fill="black"/>

   <path d="M 300,160 v  100 a 250,100 0 0,1  200,-100"
         fill="url(#pattern1)"
         stroke="black" stroke-thickness="8"/>

   <ellipse cx="450" cy="240" rx="50" ry="50"
            fill="url(#pattern1)"/>
  </g>
</svg>
```

Listing 2.4 contains an SVG `<defs>` element that specifies a `<linearGradient>` element (whose `id` attribute has value `pattern1`) with three stop values, followed by a `<radialGradient>` element with three `<stop>` elements and an `id` attribute whose value is `pattern2`.

The SVG `<g>` element consists of four `<ellipse>` elements, the first of which specifies the point `(120,80)` as its center `(cx,cy)`, with a major radius of `100`, a minor radius of `50`, filled with the linear gradient `pattern1`, as shown here:

```
<ellipse cx="120" cy="80" rx="100" ry="50"
         fill="url(#pattern1)"/>
```

Similar comments apply to the other three SVG `<ellipse>` elements.

The SVG `<g>` element also contains four `<path>` elements that render elliptic arcs. The first `<path>` element specifies a black background for the elliptic arc defined with the following `d` attribute:

```
d="M 505,145 v -100 a 250,100 0 0,1 -200,100"
```

Unfortunately, the SVG syntax for elliptic arcs is non-intuitive, and it's based on the notion of major arcs and minor arcs that connect two points on an ellipse. This example is only for illustrative purposes, so we won't delve into a detailed explanation of how elliptic arcs are defined in SVG.

The second SVG `<path>` element renders the same elliptic arc with a slight offset, using the linear gradient `pattern1`, which creates a shadow effect.

Similar comments apply to the other pair of SVG `<path>` elements, which render an elliptic arc with the radial gradient `pattern2` (also with a shadow effect).

Figure 2.4 displays the result of rendering `BasicShapesLRG1.svg` in a browser.

THE SVG `<PATTERN>` ELEMENT

You can create very sophisticated custom patterns using the SVG `<pattern>` element. Listing 2.5 displays the contents of the SVG document `PatternElement.svg`.

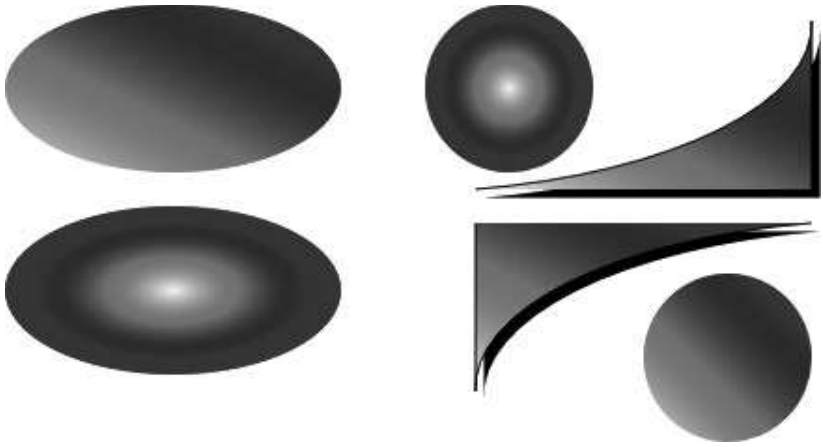

FIGURE 2.4: SVG radial gradient arcs.

LISTING 2.5: PatternElement.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
    <pattern id="dotPattern1" width="8" height="8"
             patternUnits="userSpaceOnUse">
       <circle id="circle1" cx="2" cy="2" r="3"
               style="fill:blue;"/>
    </pattern>

    <pattern id="dotPattern2" width="8" height="8"
             patternUnits="userSpaceOnUse">
     <rect x="0" y="0" width="4" height="4" fill="red" />
     <rect x="4" y="0" width="4" height="4" fill="black" />
     <rect x="0" y="4" width="4" height="4" fill="black" />
     <rect x="4" y="4" width="4" height="4" fill="red" />
    </pattern>
  </defs>

  <g id="largeCylinder" transform="translate(100,20)">
     <rect x="0" y="0" width="200" height="100"
           style="fill:url(#dotPattern1)"/>

     <ellipse cx="100" cy="150" rx="80"  ry="40"
              stroke="blue" stroke-width="4"
              style="fill:url(#dotPattern2)"/>
  </g>
</svg>
```

Listing 2.5 contains a `<defs>` element that defines two `<pattern>` elements. The first `<pattern>` element contains a `patternUnits` attribute

with the value `userSpaceOnUse`. This `<pattern>` element contains a small blue circle that creates a "dotted" pattern effect. The second `<pattern>` element defines four small rectangles that create a "checkerboard" pattern of red and black rectangles.

The second part of Listing 2.5 consists of a `<g>` element that contains a `<rect>` element and an `<ellipse>` element. The `<rect>` element specifies the first `<pattern>` element as its fill color, whereas the `<ellipse>` element specifies the second `<pattern>` element as its fill color.

Figure 2.5 displays a screenshot of `PatternElement.svg` in a browser.

SVG FILTERS

SVG supports a rich set of filters, and the SVG Filter specification has merged with the CSS Filter specification. The main SVG `<filter>` elements are listed here:

The SVG `<feBlend>` element
The SVG `<feColorMatrix>` element
The SVG `<feComponentTransfer>` element
The SVG `<feComposite>` element
The SVG `<feConvolveMatrix>` element
The SVG `<feDiffuseLighting>` element
The SVG `<feDisplacementMap>` element
The SVG `<feFlood>` element
The SVG `<feGaussianBlur>` element
The SVG `<feImage>` element
The SVG `<feMerge>` element
The SVG `<feMorphology>` element

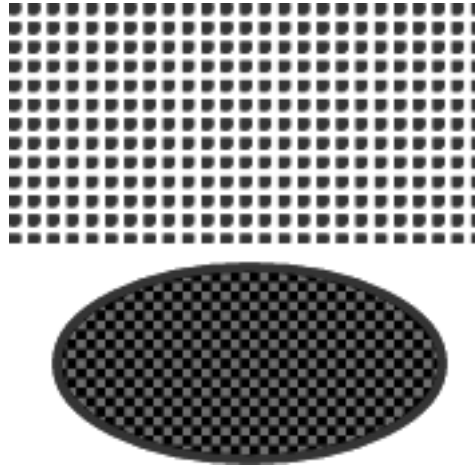

FIGURE 2.5: An SVG `<pattern>` element.

The SVG <feOffset> element
The SVG <feSpecularLighting> element
The SVG <feTile> element

Listing 2.6 displays the contents of SVGFilterTemplate.svg that you can use as a simple "template" for basic SVG filters.

LISTING 2.6: SVGFilterTemplate1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
 width="4in" height="3in">
  <defs>
    <filter id="AFilter">
      <!-- Definition of filter goes here -->
    </filter>
  </defs>
  <text style="filter:url(#AFilter)">a filter applied</text>
</svg>
```

Listing 2.6 contains a <defs> element that defines a <filter> element, followed by a <text> element that references the defined <filter> element using the url() syntax.

More complex SVG filters involve a "pipelined" relationship whereby the output of one <filter> element serves as the input for a second <filter> element. Keep in mind that filters tend to involve many computations, which can adversely affect performance.

Before you read the next several sections in this chapter, please keep in mind that they are intended to provide a conceptual overview of SVG filters. More detailed information about SVG filters is available in the W3C SVG Filters specification: *http://www.w3.org/TR/SVG/filters.html*.

SVG FILTERS AND SHADOW EFFECTS

You can create nice filter effects that you can apply to 2D shapes as well as text strings, and this section contains three SVG-based examples of creating such effects.

Listing 2.7 displays the contents of the SVG document BlurFilterText1.svg that creates a "turbulence" filter effect.

LISTING 2.7: BlurFilterText1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">
```

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
  <filter
     id="blurFilter1"
     filterUnits="objectBoundingBox"
     x="0" y="0"
     width="100%" height="100%">
     <feGaussianBlur stdDeviation="4"/>
  </filter>
  </defs>

<g transform="translate(50,100)">
  <text id="normalText" x="0" y="0"
        fill="red" stroke="black" stroke-width="4"
        font-size="72">
      Normal Text
  </text>

  <text id="horizontalText" x="0" y="100"
        filter="url(#blurFilter1)"
        fill="red" stroke="black" stroke-width="4"
        font-size="72">
      Blurred Text
  </text>
</g>
</svg>
```

The SVG `<defs>` element in Listing 2.7 contains an SVG `<filter>` element that specifies a Gaussian blur with the following line:

```
<feGaussianBlur stdDeviation="4"/>
```

You can specify larger values for the `stdDeviation` attribute if you want to create more "diffuse" filter effects.

The first SVG `<text>` element that is contained in the SVG `<g>` element renders a normal text string, whereas the second SVG `<text>` element contains a `filter` attribute that references the filter (defined in the SVG `<defs>` element) in order to render the same text string, as shown here:

```
filter="url(#blurFilter1)"
```

Figure 2.6 displays the result of rendering `BlurFilterText1.svg` that creates a filter effect in a browser.

SVG TURBULENCE FILTER EFFECTS WITH 2D SHAPES

The code sample in this section contains the SVG `<feTurbulence>` filter in conjunction with simple 2D shapes in SVG. Listing 2.8 displays the contents of the SVG documents `LGRefCBDispFilter5.svg`.

FIGURE 2.6: An SVG Filter Effect.

LISTING 2.8: LGRefCBDispFilter5.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
<defs>
  <filter id="turbFilter4" in="SourceImage"
          filterUnits="objectBoundingBox">
      <feTurbulence baseFrequency="0.05" numOctaves="0"
                    result="turbulenceOut1"/>
        <feDisplacementMap in="SourceGraphic" in2="turbulenceOut1"
                           xChannelSelector="B" yChannelSelector="B"
                           scale="100"/>
   </filter>

   <filter id="displaceFilter1">
    <feImage xlink:href="#3DMixedCircle"
             x="0" y="0" width="800" height="800"
             result="image1"/>
    <feImage xlink:href="#3DMixedCircle"
             x="0" y="0" width="800" height="800"
             result="image2"/>
    <feDisplacementMap scale="200"
                       in="image1"
                       in2="image2"
                       xChannelSelector="B"
                       yChannelSelector="B"
                       result="displacedResult">
    </feDisplacementMap>
   </filter>

   <filter id="displaceFilter2">
    <feImage xlink:href="#3DCBCircle"
             x="0" y="0" width="800" height="800"
             result="image1"/>
    <feImage xlink:href="#3DCBCircle"
             x="0" y="0" width="800" height="800"
             result="image2"/>
    <feDisplacementMap scale="200"
                       in="image1"
                       in2="image2"
```

```
                    xChannelSelector="B"
                    yChannelSelector="B"
                    result="displacedResult">
    </feDisplacementMap>
   </filter>

   <radialGradient id="mixedCircle"
                   gradientUnits="objectBoundingBox"
                   fx="20%" fy="20%">
     <stop offset="0%"   style="stop-color:#FFFFFF"/>
     <stop offset="40%"  style="stop-color:#AAAA00"/>
     <stop offset="41%"  style="stop-color:#AAAA00"/>
     <stop offset="42%"  style="stop-color:#000000"/>
     <stop offset="60%"  style="stop-color:#AAAA00"/>
     <stop offset="62%"  style="stop-color:#AAAA00"/>
     <stop offset="64%"  style="stop-color:#000000"/>
     <stop offset="100%" style="stop-color:#666600"/>
   </radialGradient>

   <circle id="3DMixedCircle" cx="150" cy="150" r="150"
            opacity=".8"
            style="fill:url(#mixedCircle)"/>

   <pattern id="checkerPattern"
            width="10" height="10"
            patternUnits="userSpaceOnUse">
      <rect fill="red"
            x="0" y="0" width="5" height="5"/>
      <rect fill="blue"
            x="5" y="0" width="5" height="5"/>
      <rect fill="blue"
            x="0" y="5" width="5" height="5"/>
      <rect fill="red"
            x="5" y="5" width="5" height="5"/>
      <circle cx="5" cy="5" r="4" style="fill:white"/>
   </pattern>

   <circle id="3DCBCircle" cx="150" cy="150" r="150"
            opacity=".8"
            style="fill:url(#checkerPattern)"/>
</defs>

<g id="gc" transform="translate(0,0)">
   <path d="M50,50 L450,50 L250,200 z"
      filter="url(#displaceFilter1)">
   </path>

   <path d="M50,200 L450,200 L250,50 z"
      filter="url(#displaceFilter2)">
   </path>
</g>
</svg>
```

Listing 2.8 is quite lengthy, with many element definitions in the `<defs>` element. However, the code consists primarily of SVG elements that you have encountered in previous code samples. After you finish this section and you

view the output in a browser, experiment with different values for the attributes in the filters in order to create your own variations that you can use in your Web pages.

The first part of Listing 2.8 contains a <defs> element that defines a "turbulence" filter, three "displacement" filters, a radial gradient (and a circle that references the gradient), and finally a <pattern> element (and a circle that references the pattern).

The first <filter> element is the simplest of the three filters, consisting of an SVG <feTurbulence> element filter whose output becomes the input for an <feDisplacementMap> filter, as shown here:

```
<filter id="turbFilter4" in="SourceImage"
        filterUnits="objectBoundingBox">
   <feTurbulence baseFrequency="0.05" numOctaves="0"
                 result="turbulenceOut1"/>
   <feDisplacementMap in="SourceGraphic" in2="turbulenceOut1"
                      xChannelSelector="B" yChannelSelector="B"
                      scale="100"/>
   </filter>
```

The second <filter> element contains two <feImage> filters that are the input for an <feImageDisplacementMap> filter.

The third <filter> element also contains two <feImage> filters that are the input for an <feImageDisplacementMap> filter, which is similar to the second <filter> element.

The next portion of the <defs> element defines a <radialGradient> element with eight <stop> elements, which is referenced as the fill value of a <circle> element whose definition follows this radial gradient.

The next portion of the <defs> element contains another "checkerboard" <pattern> element that also contains a circle, which is used to fill a <circle> element whose id attribute equals 3DCBCircle.

The next part of Listing 2.8 consists of a <g> element that contains two <path> elements, each of which references a <filter> element that is defined in the <defs> element.

Figure 2.7 displays a screenshot of LGRefCBDispFilter5.svg that creates a filter effect in Chrome on a MacBook.

ADDITIONAL FILTER EFFECTS

SVG provides a rich set of filters to create visually appealing effects in Web pages. There are two points to keep in mind. First, there can be a performance impact (especially those that create richer visual effects), so use them judiciously. You quickly can test your websites by noting the difference in speed when you use various filters.

Second, the following website provides a summary of the support (and the level of support) for SVG filters in desktop browsers and mobile browsers: *http://caniuse.com/svg-filters*.

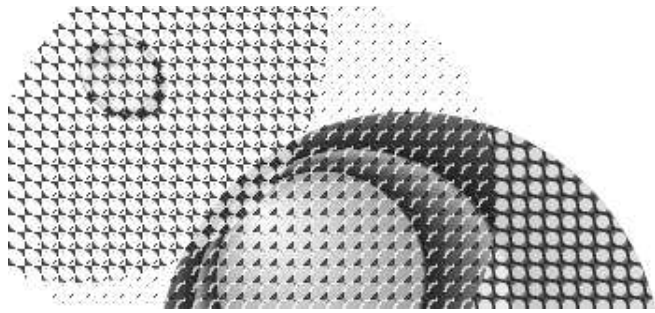

FIGURE 2.7: An SVG filter effect.

With the preceding points in mind, the companion disc contains an assortment of code samples with SVG filters.

You can create lighting effects with two light filter primitives (`feDiffuseLighting` and `feSpecularLighting`) and three light sources (`feDistantLight`, `fePointLight`, and `feSpotlight`), an example of which is shown here:

```
<filter id="LightingFilter1">
  <feDiffuseLighting>
    <feSpotLight x="0" y="0" z="50"
                 pointsAtX="300" pointsAtY="300" pointsAtZ="0"
                 limitingConeAngle="20" specularExponent="5"/>
  </feDiffuseLighting>
</filter>
```

Another example of a lighting filter is shown here:

```
<filter id="LightingFilter2">
  <feDiffuseLighting result="result1">
    <feSpotLight x="0" y="0" z="50"
                 pointsAtX="300" pointsAtY="300" pointsAtZ="0"
                 limitingConeAngle="20" specularExponent="5"/>
  </feDiffuseLighting>
  <feComposite operator="arithmetic" k1="1" k2="0" k3="0" k4="0"
               in="SourceGraphic" in2="result1"/>
</filter>
```

An example of a shadow filter is shown here:

```
<filter id="ShadowFilter">
  <feOffset dx="5" dy="5"/>
  <feGaussianBlur stdDeviation="3"/>
  <feColorMatrix type="matrix"
                 values="0 0 0 0 .2, 0 0 0 0 1, 0 0 0 0 .75, 0 0 0 1 0"
                 result="shadow"/>
  <feMerge>
    <feMergeNode in="shadow"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

Two examples of morphology filters are shown here:

```
<filter id="MorphologyeFilter1" >
  <feMorphology operator="erode" radius="7" />
<filter />

<filter id="MorphologyFilter2" >
  <feMorphology operator="erode"  radius="7" />
  <feMorphology operator="dilate" radius="6" />
<filter />
```

Two examples of color matrix filters are shown here:

```
<filter id="ColorMatrixFilter1" >
  <feColorMatrix type="saturate" values="2" />
<filter />

<filter id="ColorMatrixFilter2" >
  <feColorMatrix type="hueRotate" values="180" />
<filter />
```

An example of an `feFlood` filter:

```
<filter id="FeFloodFilter" filterUnits="userSpaceOnUse"
        x="0" y="0"  width="400" height="400">
  <feFlood x="100" y="100" width="100" height="100"
           flood-color="green" flood-opacity="0.5"/>
</filter>
```

An example of combining an `feImage` filter and an `feTile` filter:

```
<filter id="ImageTileFilter" filterUnits="userSpaceOnUse"
        x="0" y="0" width="400" height="400">
  <feImage x="0" y="0" width="100" height="100"
           xlink:href="sample.svg" result="picture"/>
  <feTile x="0" y="0" width="400" height="400" in="picture"/>
</filter>
```

The following example illustrates how to create sophisticated filter effects using a combination of SVG filters:

```
<filter id="MyFilter" filterUnits="userSpaceOnUse">
  <feGaussianBlur in="SourceAlpha" stdDeviation="4" result="blur"/>
  <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
  <feSpecularLighting in="blur" surfaceScale="5"
                      specularConstant=".75"
                      specularExponent="20"
                      lighting-color="#bbbbbb"
                      result="specOut">
    <fePointLight x="-5000" y="-10000" z="20000"/>
  </feSpecularLighting>

  <feComposite in="specOut" in2="SourceAlpha"
               operator="in" result="specOut"/>
  <feComposite in="SourceGraphic" in2="specOut"
               operator="arithmetic"
               k1="0" k2="1" k3="1" k4="0" result="litPaint"/>
```

```
  <feMerge>
    <feMergeNode in="offsetBlur"/>
    <feMergeNode in="litPaint"/>
  </feMerge>
</filter>
```

If you plan to develop hybrid mobile applications, keep in mind that support for SVG filters depends on the mobile device (and iOS generally has better support for filters than Android, but it's always a good idea to test the effects on different devices).

An extensive collection of code samples with SVG-based filter effects is here: *https://github.com/ocampesato/svg-filters-graphics*.

THE SVG <FECOLORMATRIX> FILTER

Listing 2.9 displays the contents of `FeColorMatrixBlue3DCircle1.svg`, which illustrates how to define a linear gradient and also reference that definition as the desired color for an SVG `<rect>` element.

LISTING 2.9: FeColorMatrixBlue3DCircle1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">

<defs>
  <filter id="blendFilter1">
    <feImage xlink:href="blue3DCircle1.svg" result="myOutput1"/>
    <feBlend in="myOutput1" in2="SourceGraphic" mode="lighten"/>
  </filter>

  <filter id="matrixFilter1">
     <feColorMatrix
          in="SourceGraphic" type="matrix"
          values="3 0 0 0 0 2 0 0 0 0 4 0 0 0 0 5 0 0 0 0"
          result="matrixOutput1"/>
  </filter>

  <filter id="matrixFilter2">
     <feColorMatrix
          in="SourceGraphic" type="matrix"
          values="3 3 3 3 3 2 0 0 0 0 4 -1 -1 -1 -1 5 1 1 1 1"
          result="matrixOutput2"/>
  </filter>

  <filter id="matrixFilter3">
     <feColorMatrix
          in="SourceGraphic" type="matrix"
          values="1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 1"
          result="matrixOutput3"/>
  </filter>
</defs>
```

```
<g transform="translate(50,0) scale(.8,.8)">
  <image x="0" y="0" width="200" height="200"
         xlink:href="blue3DCircle1.svg"/>

  <image x="250" y="0" width="200" height="200"
         xlink:href="blue3DCircle1.svg"
         filter="url(#matrixFilter1)"/>

  <image x="0" y="220" width="200" height="200"
         xlink:href="blue3DCircle1.svg"
         filter="url(#matrixFilter2)"/>

  <image x="250" y="220" width="200" height="200"
         xlink:href="blue3DCircle1.svg"
         filter="url(#matrixFilter3)"/>
 </g>
</svg>
```

The first part of Listing 2.9 contains a `<defs>` element that defines one "blend" filter and three "matrix" filters. The first `<filter>` element contains two filters: an `<feImage>` filter whose output is the input for the `<feBlend>` filter, as shown here:

```
<filter id="blendFilter1">
  <feImage xlink:href="blue3DCircle1.svg" result="myOutput1"/>
  <feBlend in="myOutput1" in2="SourceGraphic" mode="lighten"/>
</filter>
```

The next three `<filter>` elements specify an `feColorMatrix`, each of which specifies a different sequence of numbers for the `values` attribute. Since the details for filter-related attributes are omitted (as explained earlier in this chapter), the different sequences will give you a general idea of the different visual effects that can be created.

The next part of Listing 2.9 consists of a `<g>` element that contains four `<image>` elements, the first of which references an SVG document, and the latter three specify one of the `<filter>` elements defined in the `<defs>` element.

Figure 2.8 displays a screenshot of `FeColorMatrixBlue3DCircle1.svg` that creates an SVG `<feColorMatrix>` filter effect in a browser on a MacBook.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The companion disc contains `CompositeFilterBlue3DCircle2.svg`, which illustrates how to create a composite SVG filter effect, as shown in Figure 2.9.

The companion disc contains `GlossyFilterCBRect3.svg`, which illustrates how to create a composite SVG filter effect, as shown in Figure 2.10.

The companion disc contains `TurbFilterCBDRect2.svg`, which illustrates how to create a turbulence filter effect on checkerboard patterns, as shown in Figure 2.11.

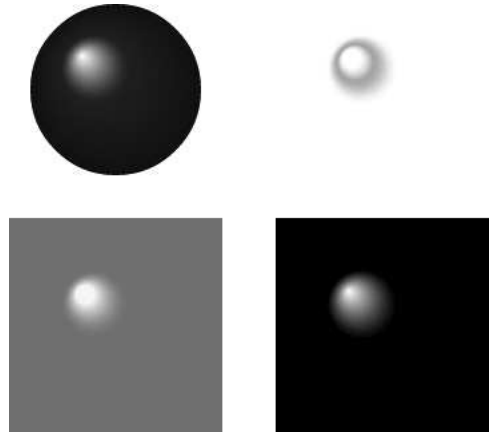

FIGURE 2.8: An SVG `<feColorMatrix>` filter effect.

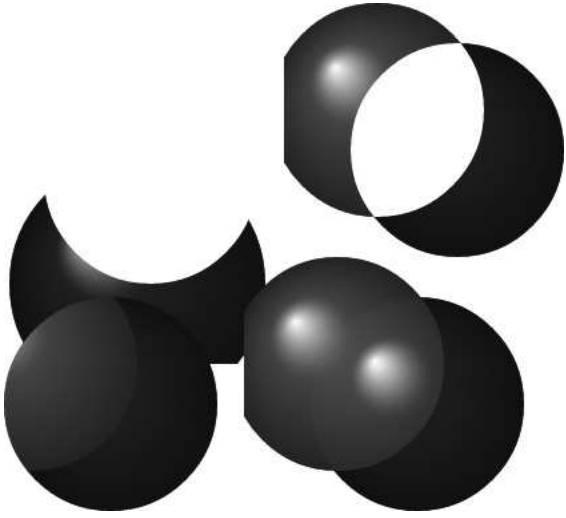

FIGURE 2.9: A composite SVG filter effect.

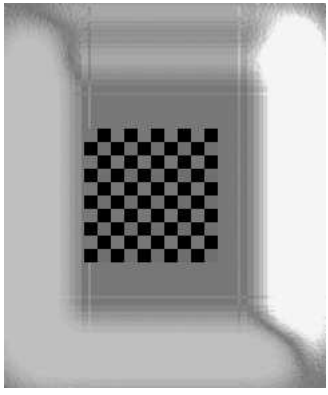

FIGURE 2.10: A glossy SVG filter effect.

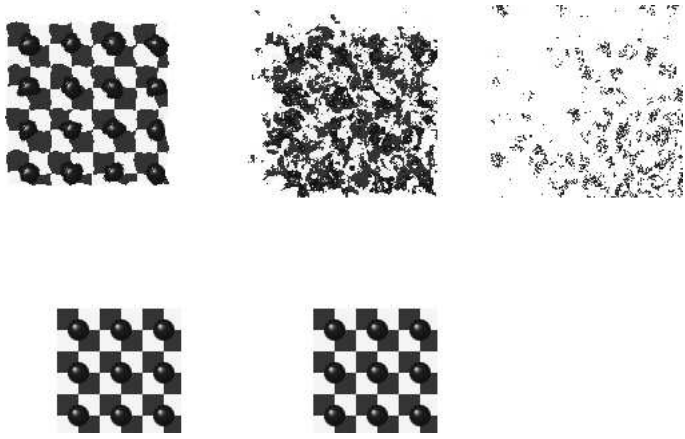

FIGURE 2.11: A turbulence filter effect on checkerboard patterns.

SUMMARY

This chapter showed you how to define linear gradients and radial gradients in SVG, and also how to use them to render shapes such as cubes. Next you learned about SVG filters, along with examples that showed you how to create "turbulence" filters as well as "displacement" filters. Finally, you saw examples of defining other types of SVG filters. Keep in mind that SVG filter effects can be computationally intensive, which in turn can affect performance and also affect the battery life of mobile devices on which you render Web pages that contain SVG filters.

The next chapter introduces you to SVG transforms and how to use them with 2D shapes.

CHAPTER 3

SVG TRANSFORMS

This chapter discusses SVG transforms, filters, and custom patterns that you can apply to SVG elements. Many of the code samples in this chapter involve SVG features that are discussed in earlier chapters, supplemented with SVG transforms to create new and interesting visual effects. If you encounter SVG functionality that is unfamiliar, you can review the appropriate section in an earlier chapter that discusses that functionality.

The first part of this chapter shows you how to render basic 2D shapes using SVG transforms, such as translate, rotate, scale, and skew. The second part of this chapter contains code samples that illustrate how to apply SVG filters to 2D shapes and text strings, and also how to create simple shadow effects. The third part of this chapter shows you how to perform clipping and apply masks to 2D shapes in SVG.

OVERVIEW OF SVG TRANSFORMS

SVG supports the transform functions `scale()`, `translate()`, and `rotate()`, and also the `skew()` function to create skew effects. There are also the transform functions `skewX()` and `skewY()` that perform a skew operation on the horizontal axis and the vertical axis, respectively. The SVG `matrix()` function is another transform that enables you to specify a combination of the preceding transforms.

In case you don't already know, CSS3 supports the preceding transforms, and also provides 3D-based counterparts called `scale3D()`, `translate3D()`, and `rotate3D()`. There is no 3D counterpart to the `skew()` transform. CSS3 also supports a `matrix()` transform and a `perspective()` transform. One point to keep in mind is that the CSS3 `skew()` transform has been deprecated.

Although SVG does not support 3D transforms, in Chapter 5 you will see an example of defining an SVG document that contains a CSS3-based `<style>` element that creates 3D animation effects on a set of SVG-based cubes.

SVG TRANSFORMS AND BASIC 2D SHAPES

The example in this section shows you how to apply the SVG transforms `scale()`, `translate()`, `rotate()`, and `skew()` to a 2D shape.

The code sample in this section involves XLink (a W3C Recommendation) for creating hyperlinks within XML documents. XLink has its own namespace, as shown here:

```
xmlns:xlink="http://www.w3.org/1999/xlink"
```

The purpose of the `<use>` element in this code sample is to reference another DOM element in order to "map" the graphical contents of the referenced element onto a rectangular region within the current coordinate system. In fact, you can create SVG documents with <use> elements with a "building block" approach. For example, if you want to render a house, one way to do so involves creating separate SVG elements that represent a front wall, a front door, a roof, and a window (and other elements as well). Then you create a house consisting of those elements. The beauty of this approach is that you can create multiple houses, each of which can be scaled to different sizes. Moreover, it's easy to apply other transformations to that house as well, such as rotate, translate, and skew transformations.

An example of the `<use>` element with the `xlink` attribute is here:

```
<use xlink:href="#largeCylinder" x="0" y="0"/>
```

The preceding snippet references the SVG element (defined in the `<use>` element) whose `id` attribute has the value `largeCylinder`.

Listing 3.1 displays the contents of `TransformEffects.svg`, which applies these transforms to a cylindrical shape.

LISTING 3.1: TransformEffects1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
<defs>
  <linearGradient id="gradientDefinition1"
     x1="0" y1="0" x2="200" y2="0"
     gradientUnits="userSpaceOnUse">
     <stop offset="0%"   style="stop-color:#FF0000"/>
     <stop offset="100%" style="stop-color:#440000"/>
  </linearGradient>
```

```
  <pattern id="dotPattern" width="8" height="8"
           patternUnits="userSpaceOnUse">

     <circle id="circle1" cx="2" cy="2" r="2"
        style="fill:red;"/>
  </pattern>
</defs>

<!-- full cylinder -->
<g id="largeCylinder" transform="translate(100,20)">
   <ellipse cx="0"  cy="50" rx="20" ry="50"
            stroke="blue" stroke-width="4"
            style="fill:url(#gradientDefinition1)"/>

   <rect x="0" y="0" width="300" height="100"
         style="fill:url(#gradientDefinition1)"/>

   <rect x="0" y="0" width="300" height="100"
         style="fill:url(#dotPattern)"/>

   <ellipse cx="300" cy="50" rx="20"  ry="50"
            stroke="blue" stroke-width="4"
            style="fill:yellow;"/>
</g>

<!-- half-sized cylinder -->
<g transform="translate(100,100) scale(.5)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>

<!-- skewed cylinder -->
<g transform="translate(100,100) skewX(40) skewY(20)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>

<!-- rotated cylinder -->
<g transform="translate(100,100) rotate(40)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>
</svg>
```

Listing 3.1 starts with an SVG `<defs>` element that defines one `<linearGradient>` element and one `<pattern>` element.

The second part of Listing 3.1 consists of an SVG `<g>` element with four child elements: two `<ellipse>` elements and two `<rect>` elements in order to render a "full" cylinder.

The next three `<g>` elements reference the first `<g>` element via its `id` attribute (whose value is `#largeCylinder`) in order to apply the transforms `scale()`, `skew()`, and `rotate()` to the initial cylinder.

Figure 3.1 displays a screenshot of `TransformEffects.svg` in a Chrome browser.

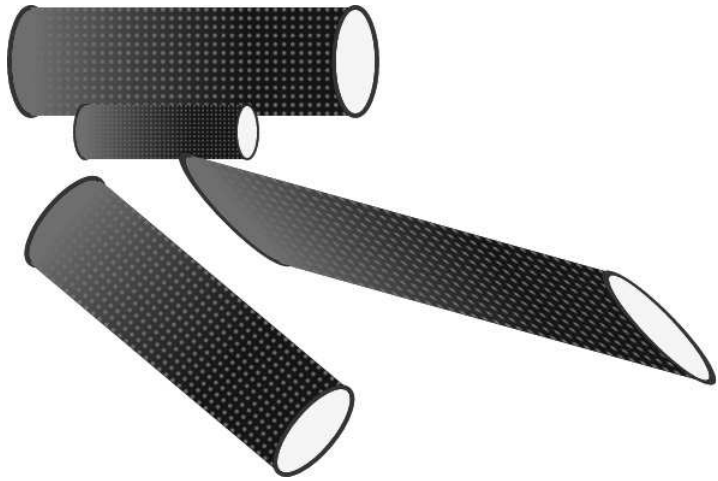

FIGURE 3.1: SVG transform effects on a cylindrical shape.

USING SVG FILTERS WITH SVG RECTANGLES

In earlier chapters you saw how to render rectangles with simple colors and with linear or radial gradients, whereas this section shows you how to render rectangles with an SVG `<filter>` element that you learned in the previous chapter.

Listing 3.2 displays the contents of `RectanglesTurbFilter1.svg`, which applies an SVG transform and an SVG filter to a rectangle.

LISTING 3.2: RectanglesTurbFilter1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
 <defs>
  <filter id="turbFilter4" in="SourceImage"
          filterUnits="objectBoundingBox">
      <feTurbulence baseFrequency="0.05" numOctaves="2"
                    result="turbulenceOut1"/>
        <feDisplacementMap in="SourceGraphic" in2="turbulenceOut1"
                           xChannelSelector="B" yChannelSelector="B"
                           scale="100"/>
   </filter>
 </defs>

  <g id="gc" transform="translate(10,10)">
    <g x="0" y="0" width="600" height="250">
      <rect x="0"   y="0"
            width="300" height="250" fill="red" />
```

```
      <rect x="300" y="0"
            width="300" height="250" fill="blue" />
    </g>

    <g transform="translate(0,250)">
      <rect x="0"   y="0"
            width="300" height="250"
            fill="red" filter="url(#turbFilter4)">
      </rect>

      <rect x="300" y="0"
            width="300" height="250"
            fill="blue" filter="url(#turbFilter4)">
      </rect>
    </g>
  </g>
</svg>
```

Listing 3.2 starts with an SVG `<defs>` element that contains an SVG `<filter>` element that specifies a turbulence filter. Notice that there is an outer `<g>` element with an SVG transform that contains two child `<g>` elements, each of which specifies yet another SVG transform. Although these transforms are not strictly necessary, they do illustrate the fact that you can easily include these and other transforms (such as scale, skew, and rotate) to SVG code.

The next portion consists of a `<g>` element that contains two child `<g>` elements. Both of these `<g>` elements contain a pair of `<rect>` elements, along with a different transform attribute in order to shift the location at which the rectangles are rendered.

Figure 3.2 displays the result of launching `RectanglesTurbFilter1.svg` in a Chrome browser on a MacBook.

FIGURE 3.2: SVG transform effects on text strings.

USING SVG FILTERS AND TRANSFORMS WITH TEXT

This section contains an example of applying SVG transforms to text strings. Listing 3.3 displays the contents of `TransformBlurFilterText1.svg`, which illustrates how to render blurred text strings and how to apply various transforms to the text strings.

LISTING 3.3: TransformBlurFilterText1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
   <filter id="blurFilter1"
           x="0" y="0"
           filterUnits="objectBoundingBox"
           width="100%" height="100%">
     <feGaussianBlur stdDeviation="4"/>
   </filter>
  </defs>

  <g transform="translate(50,100)">
    <text id="normalText" x="0" y="0" fill="red"
          stroke="black" stroke-width="4"
          font-size="72">
        Normal Text
    </text>

    <text id="horizontalText" x="0" y="100"
          filter="url(#blurFilter1)"
          fill="red" stroke="black" stroke-width="4"
          font-size="72">
        Blurred Text
    </text>

   <!-- half-sized text -->
   <g transform="translate(100,100) scale(.5)">
      <use xlink:href="#horizontalText" x="10" y="10"/>
   </g>

   <g transform="translate(100,100) scale(.5)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>

   <!-- skewed text -->
   <g transform="translate(100,100) skewX(40) skewY(20)">
      <use xlink:href="#horizontalText" x="10" y="10"/>
   </g>

   <g transform="translate(100,100) skewX(40) skewY(20)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>
```

```
   <!-- rotated text -->
   <g transform="translate(100,100) rotate(40)">
      <use xlink:href="#horizontalTextFilter" x="10" y="10"/>
   </g>

   <!-- rotated text -->
   <g transform="translate(100,100) rotate(40)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>
  </g>
</svg>
```

Listing 3.3 starts with an SVG `<defs>` element that contains an SVG `<filter>` element that specifies a Gaussian blur filter. The next portion consists of an "outer" `<g>` element that contains two child `<text>` elements, followed by six nested `<g>` elements. The `<text>` elements display "normal" text along with text that is rendered via the `<filter>` element that is defined in the `<defs>` element.

Notice that each of the six nested `<g>` elements render text with different transforms, such as `rotate()`, `skewX()`, and `skewY()`. You can also specify multiple consecutive transforms, as shown in Listing 3.3. This flexibility enables you to create complex visual effects using simple sequences of transforms. You can see how they affect the rendered text when you launch the code in a browser.

Figure 3.3 displays the result of launching `TransformBlurFilterText1.svg` in a Chrome browser on a MacBook.

TEXT WITH SVG TRANSFORMS AND SHADOW EFFECTS

In Chapter 2, you learned how to render text strings in SVG using SVG filters and shadow backgrounds. This section contains an example of applying SVG transforms and shadow effects to text strings.

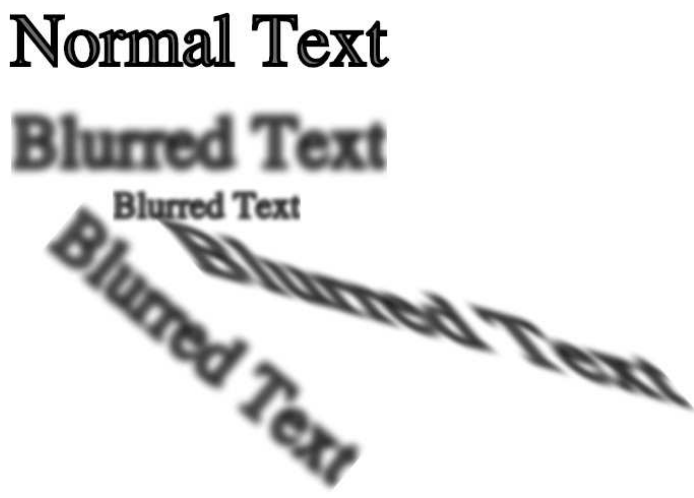

FIGURE 3.3: SVG transform effects on text strings.

Listing 3.4 displays the contents of `TransformShadowFilterText1.svg`, which illustrates how to render text strings with shadows and apply various transforms to the text.

LISTING 3.4: TransformShadowFilterText1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
   <filter
     id="blurFilter1"
     filterUnits="objectBoundingBox"
     x="0" y="0"
     width="100%" height="100%">
     <feGaussianBlur stdDeviation="4"/>
   </filter>
  </defs>

  <g transform="translate(50,150)">
    <text id="horizontalTextFilter" x="15" y="15"
          filter="url(#blurFilter1)"
          fill="red" stroke="black" stroke-width="2"
          font-size="72">
        Shadow Text
    </text>

    <text id="horizontalText" x="0" y="0"
          fill="red" stroke="black" stroke-width="4"
          font-size="72">
        Shadow Text
    </text>

   <!-- half-sized text -->
   <g transform="translate(100,100) scale(.5)">
      <use xlink:href="#horizontalTextFilter" x="10" y="10"/>
   </g>

   <g transform="translate(100,100) scale(.5)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>

   <!-- skewed text -->
   <g transform="translate(100,100) skewX(40) skewY(20)">
      <use xlink:href="#horizontalTextFilter" x="10" y="10"/>
   </g>

   <g transform="translate(100,100) skewX(40) skewY(20)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>
```

```
   <!-- rotated text -->
   <g transform="translate(100,100) rotate(40)">
      <use xlink:href="#horizontalTextFilter" x="10" y="10"/>
   </g>

   <!-- rotated text -->
   <g transform="translate(100,100) rotate(40)">
      <use xlink:href="#horizontalText" x="0" y="0"/>
   </g>
</g>
</svg>
```

Listing 3.4 starts with an SVG `<defs>` element that contains an SVG `<filter>` element that specifies a Gaussian blur filter. The next portion consists of an "outer" `<g>` element that contains two child `<text>` elements, followed by six nested `<g>` elements. The `<text>` elements display "normal" text along with text that is rendered via the `<filter>` element that is defined in the `<defs>` element.

The six nested `<g>` elements render text with different transforms, such as `rotate()` and `skew()`, and you can see how they affect the rendered text when you launch the code in a browser.

Figure 3.4 displays a screenshot of `TransformShadowFilterText1.svg` in a Chrome browser.

APPLYING SVG TRANSFORMS TO A CUBE

Listing 3.5 displays a portion of `ScaleSvgCube1Grad1.svg`, which illustrates how to apply SVG transforms to a cube that is rendered with SVG gradients. The omitted code is identical to the code in `ScaleSvgCube1Grad1.svg`

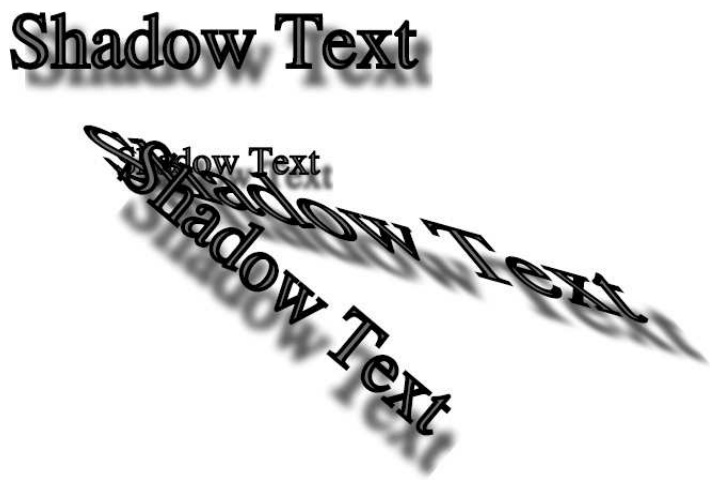

FIGURE 3.4: SVG transform effects on text strings.

that you saw in Chapter 2, and the complete code for Listing 3.5 is available on the companion disc.

LISTING 3.5: ScaleSvgCube1Grad1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

  <-- code omitted for brevity -->
  <g id="cube1">
   <!-- top face (counter clockwise) -->
   <polygon fill="url(#pattern1)"
            points="50,50 200,50 240,30 90,30"/>

   <!-- front face -->
   <rect width="150" height="150" x="50" y="50"
         fill="url(#pattern2)"/>

   <!-- right face (counter clockwise) -->
   <polygon fill="url(#pattern3)"
            points="200,50 200,200 240,180 240,30"/>
  </g>

  <g id="cube2" transform="scale(0.5, 0.5)">>
    <use xlink:href="#cube1"   x="100" y="100"/>
  </g>

  <g id="cube3" transform="rotate(20)">>
    <use xlink:href="#cube1"   x="300" y="100"/>
  </g>

  <g id="cube4" transform="skewX(50)">>
    <use xlink:href="#cube1"   x="0" y="100"/>
  </g>
```

Listing 3.5 defines four "cube" elements, each of which is defined in an SVG `<g>` element with consecutive `id` attributes whose values are `cube1`, `cube2`, `cube3`, and `cube4`, respectively. The code for `cube1` is identical to the code in Listing 2.3. The next three cubes are rendered by applying the transforms `scale()`, `rotate()`, and `skew()`, respectively, to the first cube.

Incidentally, you can also place the CSS3-based selectors in a separate stylesheet and then reference the stylesheet in the SVG document. For example, if you create a CSS stylesheet called `cubestyles.css`, you can modify the first two lines in Listing 3.5 with the following code:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet href="cubestyles.css" type="text/css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">
```

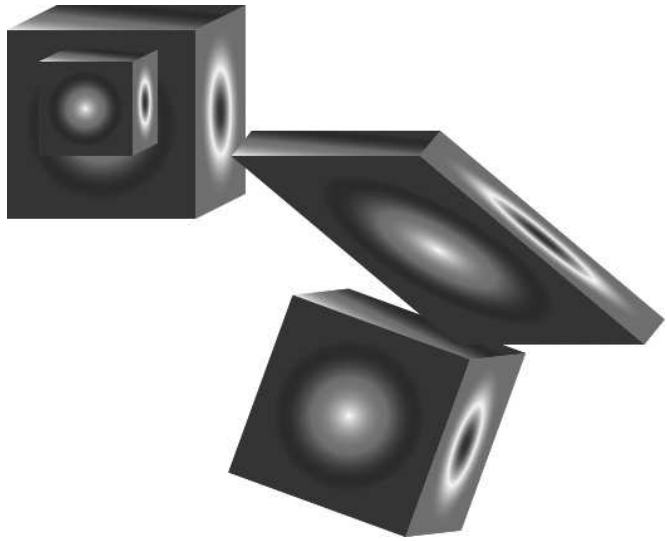

FIGURE 3.5: SVG transforms and a cube with SVG gradients.

Figure 3.5 displays a screenshot of `ScaleSvgCube1Grad1.svg` in a browser.

APPLYING SVG TRANSFORMS AND FILTERS TO A CUBE

Listing 3.6 displays a portion of `SvgCube1Grad1TurbFilter1.svg` that illustrates how to apply SVG transforms to a cube that is rendered with SVG gradients. The omitted filter definition is the same code in `SvgCube1Grad1TurbFilter1.svg` that you saw in Chapter 2, and the complete code for Listing 3.6 is available on the companion disc.

LISTING 3.6: SvgCube1Grad1TurbFilter1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg>
 <g id="cube1">
   <!-- top face (counter clockwise) -->
   <polygon fill="url(#pattern1)"
            points="50,50 200,50 240,30 90,30"/>

   <!-- front face -->
   <rect width="150" height="150" x="50" y="50"
         fill="url(#pattern2)"/>

   <!-- right face (counter clockwise) -->
   <polygon fill="url(#pattern3)"
            points="200,50 200,200 240,180 240,30"/>
  </g>
```

```
  <g id="cube2" x="100" y="100"
     transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="100" y="100"
         filter="url(#turbFilter4)" />
 </g>

  <g id="cube3"
     transform="rotate(-20)">
    <use xlink:href="#cube1" x="150" y="200"
         filter="url(#turbFilter4)" />
  </g>

  <g id="cube4" x="500" y="100"
     transform="skewX(-20)">
    <use xlink:href="#cube1" x="100" y="200"
         filter="url(#turbFilter4)" />
  </g>
</svg>
```

Listing 3.6 defines the same four "cube" elements that you saw in Listing 3.5, except that the cube elements are rendered with `<pattern>` elements and `<filter>` elements (not shown in Listing 3.6). The SVG code that renders the initial 3D cube with gradient shading effects is enclosed in an SVG `<g>` element that is shown in bold in Listing 3.7, which provides a convenient way to reference this code in the other code blocks.

Figure 3.6 displays a screenshot of `SvgCube1Grad1TurbFilter1.svg` in a browser.

REPLICATING <G> ELEMENTS AND CSS3 ANIMATION IN SVG

Recall that in Chapter 2 you learned how to render an SVG-based cube, and in this section you will learn two more techniques: how to replicate an SVG element and how to apply CSS3-based animation to SVG elements.

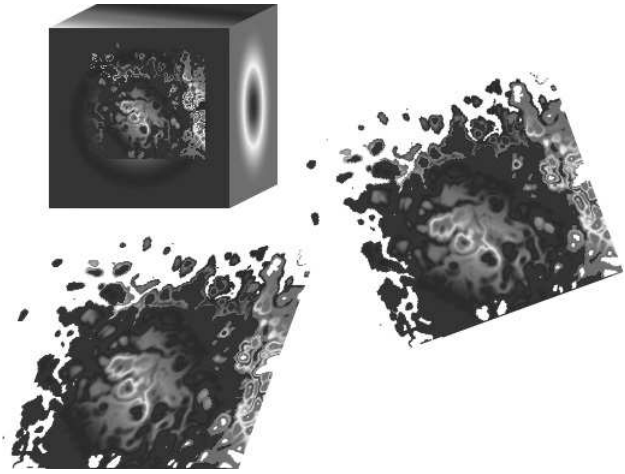

FIGURE 3.6: SVG transforms and a filter applied to a cube.

NOTE *The example in this section requires an understanding of CSS3 keyframes, which are discussed in Chapter 6.*

Listing 3.7 displays `MultiSVGCube1Grad1CSS3.svg`, which demonstrates how to use the SVG `<use>` element to replicate existing SVG elements, and also how to define CSS3 selectors to create animation effects on SVG elements.

LISTING 3.7: MultiSVGCube1Grad1CSS3.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
    <style>
     @keyframes cubeAnim {
       0% {
         transform: translate3d(50px,50px,50px)
                    rotate3d(30,40,50,-90deg)
                    skew(-15deg,0) scale3d(1.25, 1.25, 1.25);
       }
       33% {
         transform: matrix(1.0, 0.5,  1.0, 1.5, 0, 0);
       }
       66% {
         transform: translate3d(-50px,-50px,-50px)
                    rotate3d(-5,-5,5, 120deg)
                    skew(135deg,0) scale3d(0.3, 0.4, 0.5);
       }
       100% {
         transform: translate3d(50px,50px,50px)
                    rotate3d(30,40,50,-90deg)
                    skew(-15deg,0) scale3d(1.25, 1.25, 1.25);
       }
     }

     g.cube, g.cube:hover {
        animation-name: cubeAnim;
        animation-duration: 10s;
     }
    </style>

    <linearGradient id="pattern1"
                    x1="0%" y1="100%" x2="100%" y2="0%">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
    </linearGradient>

   <radialGradient id="pattern2">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
```

```
      <stop offset="80%"  stop-color="blue"/>
    </radialGradient>

    <radialGradient id="pattern3">
       <stop offset="0%"   stop-color="blue"/>
       <stop offset="40%"  stop-color="yellow"/>
       <stop offset="80%"  stop-color="red"/>
    </radialGradient>
   </defs>

   <g id="cube1">
    <!-- top face (counter clockwise) -->
    <polygon fill="url(#pattern1)"
             points="50,50 200,50 240,30 90,30"/>

    <!-- front face -->
    <rect width="150" height="150" x="50" y="50"
          fill="url(#pattern2)"/>

    <!-- right face (counter clockwise) -->
    <polygon fill="url(#pattern3)"
             points="200,50 200,200 240,180 240,30"/>
  </g>

 <g class="cube" transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="100" y="100"/>
  </g>

  <g class="cube" transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="0" y="0"/>
  </g>

  <g class="cube" transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="200" y="0"/>
  </g>

  <g class="cube" transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="0" y="300"/>
  </g>

  <g class="cube" transform="scale(0.5,0.5)">
    <use xlink:href="#cube1" x="200" y="300"/>
  </g>
</svg>
```

Listing 3.7 contains a `<defs>` element that starts with a `<style>` element that defines a CSS3-based animation effect using a `keyframes` rule discussed in Chapter 6. The `<style>` element also contains the definition of several SVG-based gradients that are similar to the gradients that are discussed in Chapter 2.

The next portion of Listing 3.7 contains the definition of an SVG-based cube that you saw in Chapter 1. The last section of Listing 3.7 (which consists of new material) contains five SVG `<g>` elements that define `<use>` elements that replicate the contents of the `<g>` element that defines the cube. Notice

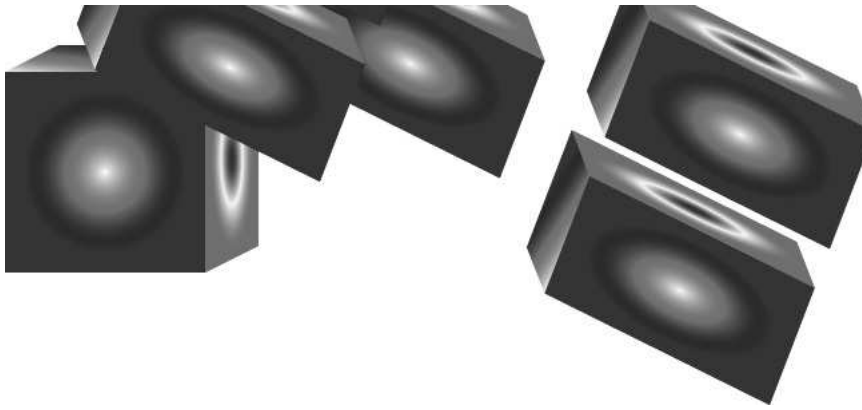

FIGURE 3.7: SVG transform effects.

that these five SVG <g> elements specify a `scale()` transform in order to resize their respective child elements. In addition, these <g> elements contain a `class` attribute whose value is `cube`, which is also the name of the class-based selector in the <style> element, as shown here:

```
g.cube, g.cube:hover {
  animation-name: cubeAnim;
  animation-duration: 10s;
}
```

The preceding code block references the `keyframes` rule `cubeAnim` that is defined at the beginning of the <style> element. The use of a `class` attribute enables you to specify an arbitrary number of <g> elements with the same animation effect.

Figure 3.7 displays a screenshot of `MultiSVGCube1Grad1CSS3.svg` in a Chrome browser.

SVG TRANSFORMS WITH THE <PATTERN> ELEMENT

In Chapter 2, you learned how to use the SVG <pattern> element in a code sample that renders a cube. Listing 3.8 displays the contents of `TransformEffects1.svg`, which illustrates how to combine the <pattern> element in conjunction with applying transforms to rectangles and circles in SVG.

LISTING 3.8: TransformEffects1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
```

```
<defs>
  <linearGradient id="gradientDefinition1"
     x1="0" y1="0" x2="200" y2="0"
     gradientUnits="userSpaceOnUse">
     <stop offset="0%"   style="stop-color:#FF0000"/>
     <stop offset="100%" style="stop-color:#440000"/>
  </linearGradient>

  <pattern id="dotPattern" width="8" height="8"
           patternUnits="userSpaceOnUse">

     <circle id="circle1" cx="2" cy="2" r="2"
        style="fill:red;"/>
  </pattern>
</defs>

<!-- full cylinder -->
<g id="largeCylinder" transform="translate(100,20)">
   <ellipse cx="0"  cy="50" rx="20" ry="50"
            stroke="blue" stroke-width="4"
            style="fill:url(#gradientDefinition1)"/>

   <rect x="0" y="0" width="300" height="100"
         style="fill:url(#gradientDefinition1)"/>

   <rect x="0" y="0" width="300" height="100"
         style="fill:url(#dotPattern)"/>

   <ellipse cx="300" cy="50" rx="20"  ry="50"
            stroke="blue" stroke-width="4"
            style="fill:yellow;"/>
</g>

<!-- half-sized cylinder -->
<g transform="translate(100,100) scale(.5)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>

<!-- skewed cylinder -->
<g transform="translate(100,100) skewX(40) skewY(20)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>

<!-- rotated cylinder -->
<g transform="translate(100,100) rotate(40)">
   <use xlink:href="#largeCylinder" x="0" y="0"/>
</g>
</svg>
```

The SVG `<defs>` element in Listing 3.8 contains a `<linearGradient>` element that defines a linear gradient, followed by an SVG `<pattern>` element that defines a custom pattern, which is shown here:

```
<pattern id="dotPattern" width="8" height="8"
          patternUnits="userSpaceOnUse">
```

```
     <circle id="circle1" cx="2" cy="2" r="2"
        style="fill:red;"/>
</pattern>
```

As you can see, the SVG `<pattern>` element contains an SVG `<circle>` element that is repeated in a grid-like fashion inside an 8x8 rectangle (note the values of the `width` attribute and the `height` attribute). As an exercise, experiment with different values for the `width` and `height` attributes to create interesting effects. The SVG `<pattern>` element has an `id` attribute whose value is `dotPattern` because (as you will see) this element creates a "dotted" effect.

Listing 3.8 contains four SVG `<g>` elements, each of which renders a cylinder that references the SVG `<pattern>` element that is defined in the SVG `<defs>` element.

The first SVG `<g>` element in Listing 3.8 contains two SVG `<ellipse>` elements and two SVG `<rect>` elements. The first `<ellipse>` element renders the left-side "cover" of the cylinder with the linear gradient that is defined in the SVG `<defs>` element. The first `<rect>` element renders the "body" of the cylinder with a linear gradient, and the second `<rect>` element renders the "dot pattern" on the body of the cylinder. Finally, the second `<ellipse>` element renders the right-side "cover" of the ellipse.

The other three cylinders are easy to create: they simply reference the first cylinder and apply a transformation to change the size, shape, and orientation. Specifically, these three cylinders reference the first cylinder with the following code:

```
<use xlink:href="#largeCylinder" x="0" y="0"/>
```

and then they apply scale, skew, and rotate functions in order to render scaled, skewed, and rotated cylinders.

Figure 3.8 displays the result of rendering `TransformEffects1.svg` in a browser.

THE <CLIPPATH> AND <MASK> ELEMENTS

The SVG `<clipPath>` element enables you to "clip" a rendered image. You can use various SVG elements as a `<clipPath>` element. In fact, you can use the `<path>` element to create complex clipped patterns. On the other hand, the SVG `<mask>` element enables you to specify a graphics object as an alpha mask for compositing the current object into the background. As you will see later in this section, the SVG `<clipPath>` and `<mask>` element act as "containers" in the sense that they contain other SVG elements (such as rectangles, ellipses, and so forth) that define the specific mask or clip path. The W3C SVG Masking specification contains many additional details regarding clipping, masking, and compositing: *http://www.w3.org/TR/SVG/masking.html#ClippingPaths* and *http://www.smashingmagazine.com/2015/12/animating-clipped-elements-svg/*.

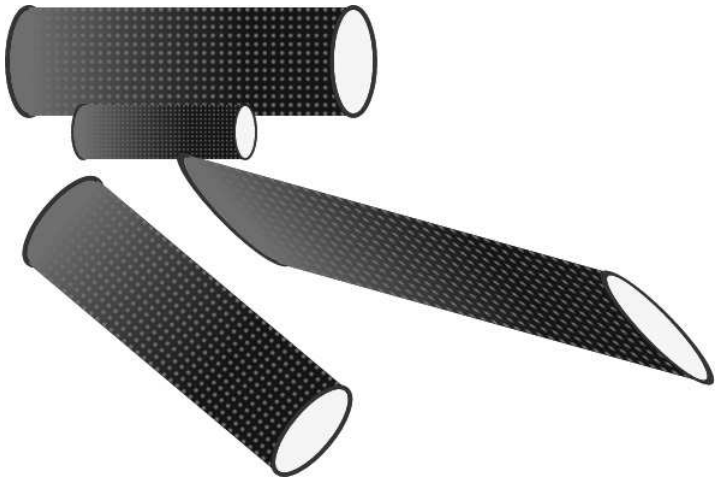

FIGURE 3.8: SVG transform effects.

Listing 3.9 displays the contents of `ClipAndMask1.svg`, which illustrates how to render both of these SVG elements.

LISTING 3.9: ClipAndMask1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

 <defs>
    <radialGradient id="rgrad1" cx="50%" cy="50%"
                    r="50%" fx="25%" fy="25%">
      <stop stop-color="blue"  offset="0%"/>
      <stop stop-color="red"   offset="50%"/>
      <stop stop-color="white" offset="100%"/>
    </radialGradient>

    <rect id="rect1" x="20" y="20" width="500" height="300"
          fill="url(#rgrad1)" stroke="blue" stroke-width="3"/>

    <clipPath id="clip1">
      <rect x="20" y="20" width="100" height="100"/>
      <circle cx="150" cy="150" r="40"/>
      <polygon points="200 50 350 100 250 150" />
      <ellipse cx="300" cy="100" rx="50" ry="20"/>
      <rect x="400" y="50" width="100" height="100"/>
    </clipPath>

    <mask id="mask1">
```

```
      <rect x="10" y="80" width="100" height="100" fill="red" />
      <circle cx="150" cy="100" r="50" fill="yellow" />
      <polygon points="300 30 350 130 250 130" fill="white" />
      <ellipse cx="350" cy="100" rx="80" ry="40" fill="#f0f" />
      <rect x="400" y="80" width="100" height="100" fill="orange" />
    </mask>
 </defs>

 <g>
   <use xlink:href="#rect1" x="20" y="300" mask="url(#mask1)"/>
   <use xlink:href="#rect1" x="20" y="20" clip-path="url(#clip1)"/>
 </g>
</svg>
```

Listing 3.9 contains a `<defs>` element that starts with a simple radial gradient that is referenced in the `<rect>` element that follows the radial gradient definition. The `<defs>` element also contains a `<clipPath>` element consisting of five SVG elements. The final portion of the `<defs>` element contains a `<mask>` element that contains the same five elements as the `<clipPath>` element: this enables you to see the differences in the effects between these two SVG elements.

The next portion of Listing 3.9 defines a `<g>` element that contains two `<use>` elements, both of which reference the `<rect>` element in the `<defs>` element. The first `<use>` element references the previously defined `<mask>` element, whereas the second `<use>` element specifies the `<clipPath>` element.

Figure 3.9 displays the result of rendering `ClipAndMask1.svg` in a Chrome browser on a MacBook Pro.

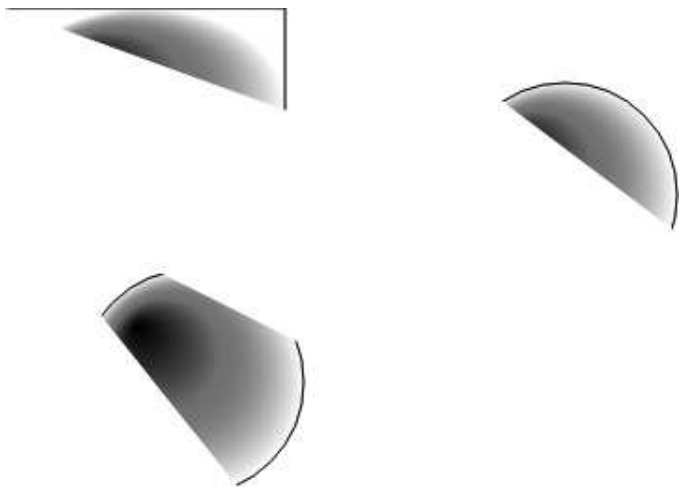

FIGURE 3.9: SVG `<clipPath>` and `<mask>` in a Chrome browser.

SUMMARY

This chapter showed you how to create SVG transforms, including translate, scale, rotate, and skew. You learned how to apply these transforms to an SVG cube that is rendered with a radial gradient (which you learned in Chapter 2). Next you learned how to use the SVG `<pattern>` element to define custom patterns.

The next chapter delves into ellipses, elliptic arcs, and Bezier curves and also applies some of the SVG techniques that you learned in this chapter.

CHAPTER 4

ELLIPSES, ARCS, AND BEZIER CURVES

This chapter contains code samples for rendering ellipses, elliptic arcs, and Bezier curves (quadratic and cubic). The code samples also involve gradient shading that you learned in Chapter 2, and you can enhance the code samples with SVG `<filter>` elements (also discussed in Chapter 2).

The first part of this chapter shows you how to use the SVG `<ellipse>` element in order to render ellipses. SVG also provides a `<circle>` element for rendering circles, which are a special case of ellipses (i.e., the major axis equals the minor axis). The second part of this chapter shows you how use the SVG `<path>` element in order to render elliptic arcs to create common objects. One detail to remember is that other programming languages typically use a "start angle" and "end angle" as their representation for elliptic arcs, whereas SVG uses concepts called "minor arc" and "major arc."

The third part of this chapter shows you how to use the SVG `<path>` element in order to render quadratic Bezier curves and cubic Bezier curves. You will also learn how to "concatenate" multiple Bezier curves using the SVG `<path>` element. This section also contains an example of rendering a text string along the path of a Bezier curve. In addition to pleasing visual effects, Bezier curves are useful for calculating the shape of letters in different fonts and for defining easing functions for animation effects.

THE SVG <ELLIPSE> ELEMENT

The code sample in this section shows you how to create simple 3D-like effects using the SVG `<ellipse>` element (described in Chapter 2) and other SVG elements from previous chapters.

Listing 4.1 displays the contents of `BlueSphereEllipse2.svg`, which illustrates how to use radial gradients, ellipses, and filters to create a spherical shape in SVG.

LISTING 4.1: BlueSphereEllipse2.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

  <defs>
     <filter
        id="blurFilter1"
        filterUnits="objectBoundingBox"
        x="0" y="0"
        width="100%" height="100%">
        <feGaussianBlur stdDeviation="4"/>
     </filter>

     <radialGradient id="blueEllipse"
                     gradientUnits="objectBoundingBox"
                     fx="30%" fy="30%">
       <stop offset="0%"   style="stop-color:#FFFFFF"/>
       <stop offset="60%"  style="stop-color:#0000AA"/>
       <stop offset="100%" style="stop-color:#000066"/>
     </radialGradient>

     <radialGradient id="shadeEllipse"
                     gradientUnits="objectBoundingBox"
                     fx="30%" fy="30%">
       <stop offset="0%"   style="stop-color:#BBBBBB"/>
       <stop offset="40%"  style="stop-color:#888888"/>
       <stop offset="100%" style="stop-color:#333333"/>
     </radialGradient>

     <ellipse id="3DBlueEllipse"
              cx="0" cy="0" rx="140" ry="160"
              style="fill:url(#blueEllipse)"/>

     <ellipse id="3DShadeEllipse"
              cx="0" cy="0" rx="140" ry="160"
              filter="url(#blurFilter1)"
              style="fill:url(#shadeEllipse)"/>
  </defs>

  <g width="100%" height="100%">
     <use xlink:href="#3DShadeEllipse"  x="300" y="200"/>
     <use xlink:href="#3DBlueEllipse"   x="200" y="220"/>
  </g>
</svg>
```

Listing 4.1 contains an SVG `<defs>` element that first defines an SVG `<filter>` element based on a Gaussian blur. Next, two radial gradients are defined that are referenced by two SVG `<ellipse>` elements (also in the `<defs>` element) to create a 3D effect and a shadow effect, respectively.

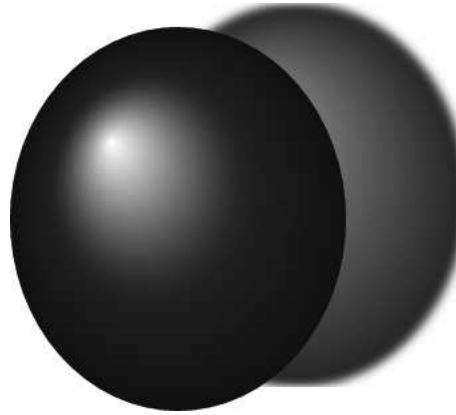

FIGURE 4.1: A 3D-like effect using two SVG <ellipse> elements.

The final portion of Listing 4.1 contains an SVG <g> element that performs the actual rendering. The SVG <g> element contains two SVG <use> elements that reference the pair of SVG <ellipse> elements (both of which are defined in the <defs> element), as shown here:

```
<g width="100%" height="100%">
     <use xlink:href="#3DShadeEllipse"  x="300" y="200"/>
     <use xlink:href="#3DBlueEllipse"   x="200" y="220"/>
</g>
```

The preceding code snippet is intended to draw your attention to the fact that the actual rendering is only a few lines of code, whereas the <defs> element is much more extensive in terms of the definitions of various elements.

Figure 4.1 displays the 3D-like effect involving two SVG <ellipse> elements based on the code in Listing 4.1.

SVG ELLIPSES AND 3D EFFECTS

The code sample in this section shows you how to create simple 3D-like effects using the SVG <ellipse> element.

Listing 4.2 displays the contents of BlueSphereEllipse2Ripple6RG6.svg, which illustrates how to use radial gradients, ellipses, and filters to create a spherical shape in SVG.

LISTING 4.2: BlueSphereEllipse2Ripple6RG6.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
```

```
 <defs>
    <radialGradient id="radialGradient1"
                    gradientUnits="userSpaceOnUse"
                    cx="0" cy="0" r="200"
                    fx="50" fy="50">
      <stop offset="0"    stop-color="white"/>
      <stop offset=".149" stop-color="red"/>
      <stop offset=".15"  stop-color="blue"/>
      <stop offset=".849" stop-color="red"/>
      <stop offset=".85"  stop-color="blue"/>
      <stop offset="1"    stop-color="black"/>
    </radialGradient>

    <radialGradient id="radialGradient2"
                    gradientUnits="userSpaceOnUse"
                    cx="0" cy="0" r="180"
                    fx="100" fy="0">
      <stop offset="0"    stop-color="yellow"/>
      <stop offset=".149" stop-color="red"/>
      <stop offset=".15"  stop-color="white"/>
      <stop offset=".549" stop-color="red"/>
      <stop offset=".55"  stop-color="white"/>
      <stop offset="1"    stop-color="black"/>
    </radialGradient>

    <radialGradient id="blueCircle"
                    gradientUnits="objectBoundingBox"
                    fx="30%" fy="30%">
      <stop offset="0%"   style="stop-color:#FFFFFF"/>
      <stop offset="40%"  style="stop-color:#0000AA"/>
      <stop offset="100%" style="stop-color:#000066"/>
    </radialGradient>

    <ellipse id="3DBlueCircle" cx="0" cy="0" rx="80" ry="60"
             opacity=".8"
             style="fill:url(#blueCircle)"/>

    <ellipse id="3DBlueCircleRG6" cx="0" cy="0" rx="180" ry="160"
             opacity=".8"
             style="fill:url(#radialGradient1)"/>

    <ellipse id="3DBlueCircle2RG6" cx="0" cy="0" rx="180" ry="160"
            style="fill:url(#radialGradient2)"/>
 </defs>

<g width="100%" height="100%">
  <use xlink:href="#3DBlueCircle2RG6" x="400" y="220"/>
  <use xlink:href="#3DBlueCircleRG6"  x="200" y="220"/>
  <use xlink:href="#3DBlueCircle"     x="200" y="120"/>
</g>
</svg>
```

Listing 4.2 contains code that superficially resembles the code in Listing 4.1, yet the graphics effect is substantively different. Specifically, Listing 4.2 contains an SVG `<defs>` element that defines three SVG `<radialGradient>` elements and then three SVG `<ellipse>` elements to create a 3D effect.

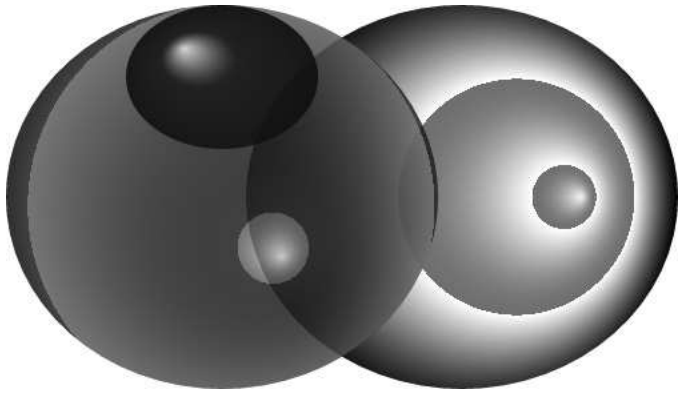

FIGURE 4.2: Multiple gradient effects with SVG <ellipse> elements.

The next portion of Listing 4.2 contains an SVG <g> element that contains three SVG <use> elements, each of which references one of the SVG <ellipse> elements in the <defs> element, as shown here:

```
<g width="100%" height="100%">
   <use xlink:href="#3DBlueCircle2RG6" x="400" y="220"/>
   <use xlink:href="#3DBlueCircleRG6"  x="200" y="220"/>
   <use xlink:href="#3DBlueCircle"     x="200" y="120"/>
</g>
```

Figure 4.2 displays a screenshot of multiple gradient effects involving SVG <ellipse> elements based on the code in Listing 4.2.

ELLIPTIC ARCS IN SVG

As you learned in the introduction, the SVG <path> element enables you to render elliptic arcs. Listing 4.3 displays the contents of EllipticArcs1.svg, which illustrates the various types of elliptic arcs that you can create in SVG.

LISTING 4.3: EllipticArcs1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <g stroke="black" stroke-width="3">
    <path d="M 20  200 a 100 50 0   1 1 250 50"
          fill="red"/>
    <path d="M 300 100 a 100 50 30  1 1 250 50"
          fill="green"/>
    <path d="M 300 300 a 100 50 45  1 1 250 50"
          fill="yellow"/>
```

```
    <path d="M 500 200 a 100 50 135 1 1 250 50"
          fill="blue"/>
  </g>
</svg>
```

Listing 4.3 contains four SVG `<path>` elements, each of which specifies a `d` attribute for an elliptic arc and a `fill` attribute for the color. An elliptic arc has the following format:

```
a rx ry x-axis-rotation large-arc-flag sweep-flag dx dy
```

The `rx` and `ry` values specify the radii along the x-axis and y-axis, followed by the rotation of the arc. The next two values can be either 0 or 1, which results in four combinations: `0 0` or `0 1` or `1 0` or `1 1`. Note that Listing 4.3 only uses the `1 1` combination for the flag-related values, but you can modify the code with any of the other three combinations to see the result. Finally, the `dx` and `dy` values specify the terminating point for the elliptic arc.

By way of comparison, some people feel that the SVG syntax for elliptic arcs is non-intuitive when compared to the syntax for elliptic arcs in HTML `Canvas`, but with practice you can familiarize yourself with SVG elliptic arcs.

Figure 4.3 displays a screenshot of four elliptic arcs based on the code in Listing 4.3.

RENDERING A TRAFFIC SIGN IN SVG

The previous section showed you how to render elliptic arcs using the SVG `<path>` element. The code sample in this section is mainly for fun: rendering "real life" objects is a good way to practice with elliptic arcs.

Listing 4.4 displays the contents of `NoUTurn1.svg`, which illustrates how to render a basic "No U-Turn" sign in SVG.

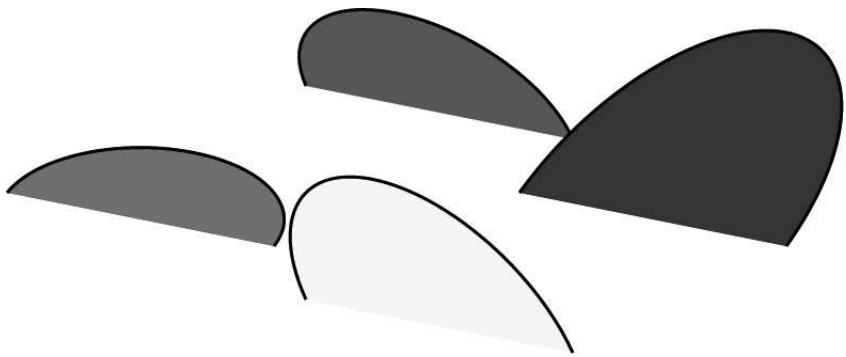

FIGURE 4.3: SVG elliptic arcs.

LISTING 4.4: NoUTurn1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">

<defs>
  <rect id="border1"
        x="0" y="0" width="100" height="100"
        fill="white" stroke="black" stroke-width="2"/>

  <circle id="thickCircle1"
        x="0" y="0" r="42"
        fill="white" stroke="red" stroke-width="8"/>

  <path id="longArc1"
        d="m0,0 l0,-40 a10,10 0 0,0 -28,0 l0,20"
        fill="none" stroke="black" stroke-width="10"/>

  <polygon id="triangle"
        points="0,0 10,0 0,10 -10,0"
        fill="black" stroke="black" stroke-width="1"/>

  <line id="slashLine1"
        x1="-28" y1="-28" x2="28"  y2="28"
        fill="none" stroke="red" stroke-width="8"/>
</defs>

<g transform="translate(50,50)">
  <use xlink:href="#border1"      x="0"   y="0"/>
  <use xlink:href="#thickCircle1" x="50"  y="50"/>
  <use xlink:href="#longArc1"     x="65"  y="80"/>
  <use xlink:href="#triangle"     x="37"  y="60"/>
  <use xlink:href="#slashLine1"   x="50"  y="50"/>
</g>
</svg>
```

Listing 4.4 uses a "building block" technique: multiple shapes are defined in an SVG `<defs>` element, and then a `<g>` element references those shapes via the SVG `<use>` element in order to display the graphics image.

Listing 4.4 starts with a `<defs>` element that contains several SVG elements. The first SVG element is an SVG `<rect>` element to display the outer rectangle of the traffic sign, followed by an SVG `<circle>` element that displays a circle (with a thick stroke) inside that rectangle. The SVG `<path>` element renders a shape that resembles an asymmetric and upside-down letter "U," followed by an SVG `<polygon>` element that renders a triangle that looks like a downward-pointing arrow. The final portion of the `<defs>` element is an SVG `<line>` element that is rendered diagonally in order to prohibit U-Turns.

FIGURE 4.4: A traffic sign with various SVG elements.

Figure 4.4 displays a screenshot of a traffic sign using the SVG elements in Listing 4.4.

RENDERING CONCAVE CUBES IN SVG

This section shows you how to create a richer visual effect using more complex SVG code. You will see that there aren't any new SVG elements in the code sample: the novelty factor is based on the manner in which various SVG elements are combined.

Listing 4.5 displays the contents of `DeformedBoxCB2.svg`, which illustrates how to render an SVG cube with a carved out portion of one of its vertices.

LISTING 4.5: DeformedBoxCB2.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

<defs>
  <radialGradient id="blueCircle"
                  gradientUnits="objectBoundingBox"
                  fx="70%" fy="70%">
    <stop offset="0%"   style="stop-color:#FFFFFF"/>
    <stop offset="40%"  style="stop-color:#0000AA"/>
    <stop offset="100%" style="stop-color:#000066"/>
  </radialGradient>

  <pattern id="checkerPattern1"
           width="8" height="8"
           patternUnits="userSpaceOnUse">
     <rect fill="blue"
           x="0"  y="0" width="4" height="4"/>
     <rect fill="red"
           x="4" y="0"  width="4" height="4"/>
```

```
      <rect fill="red"
            x="0"  y="4" width="4" height="4"/>
      <rect fill="blue"
            x="4" y="4"  width="4" height="4"/>
  </pattern>

  <pattern id="checkerPattern2"
           width="8" height="8"
           patternUnits="userSpaceOnUse">
      <rect fill="red"
            x="0"  y="0" width="4" height="4"/>
      <rect fill="yellow"
            x="4" y="0"  width="4" height="4"/>
      <rect fill="yellow"
            x="0"  y="4" width="4" height="4"/>
      <rect fill="red"
            x="4" y="4"  width="4" height="4"/>
  </pattern>

  <g id="boxShape">
    <path id="frontFace"
          d="m0,0 100,0 0,80 -100,0"
          fill="url(#checkerPattern1)"/>

    <path id="topFace"
          d="m0,0 l100,0 60,-30 -100,0"
          fill="url(#blueCircle)"/>

    <path id="rightFace"
          d="m100,0 l60,-30 0,80 -60,30"
          fill="url(#blueCircle)"/>
  </g>

  <path id="ellipticArc1"
        d="m50,0 a150,150 0 0,1 70,-10
                 a150,150 0 0,1 -20,80
                 a150,150 0 0,1 -50,-70"
        fill="url(#blueCircle)"/>

  <path id="ellipticArcCB1"
        d="m50,0 a150,150 0 0,1 70,-10
                 a150,150 0 0,1 -20,80
                 a150,150 0 0,1 -50,-70"
        opacity=".5"
        fill="url(#checkerPattern1)"/>
</defs>

<g transform="translate(20,100) scale(2,2)">
  <use xlink:href="#boxShape"       x="0" y="0"/>
  <use xlink:href="#ellipticArc1"   x="0" y="0"/>
  <use xlink:href="#ellipticArcCB1" x="0" y="0"/>
</g>
</svg>
```

Listing 4.5 uses the same "building block" technique that is described in Listing 4.4, except with a different set of SVG elements.

Listing 4.5 starts with a <defs> element that contains a <radialGradient> element and two <pattern> elements. The next portion of Listing 4.5 contains a <g> element that represents a cube: this element is inside the <defs> element, which is the first time you've seen this particular construction. The final part of the <defs> element consists of two <path> elements that reference the initial <radialGradient> and <pattern> element.

The next portion of Listing 4.5 consists of a <g> element that contains three SVG <use> elements that reference each of the elements in the <defs> element.

Figure 4.5 displays a screenshot of a "deformed cube" shape that involves various SVG elements, including elliptic arcs.

SALT AND PEPPER "SHAKERS" WITH ELLIPTIC ARCS

The code sample in this section is another example of using SVG for fun: the code involves elliptic arcs, radial gradient shading, transforms, and the SVG <use> element in order to render a pair of salt and pepper "shakers." Keep in mind that the various SVG elements were determined using an iterative process involving lots of experimentation. While you probably won't need to create a similar effect, the value of this code sample is the practice that you will gain, which in turn can significantly increase your familiarity (and also your confidence) regarding SVG graphics. By contrast, software tools that support SVG-based graphics typically generate "machine friendly" SVG code rather than "user friendly" code.

Listing 4.6 displays the contents of SaltAndPepper1.svg, which illustrates how to create this effect.

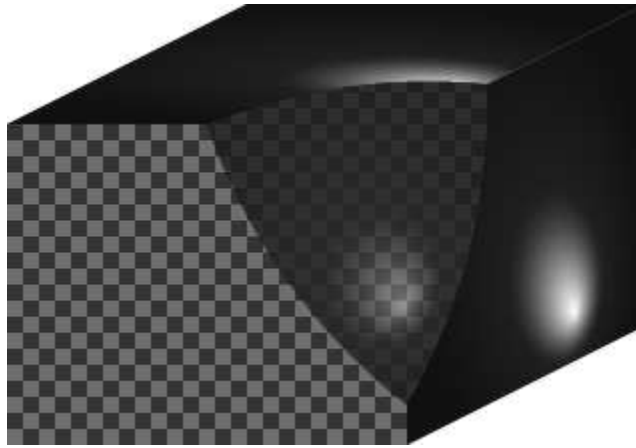

FIGURE 4.5: Rendering a deformed cube in SVG with elliptic arcs.

LISTING 4.6: SaltAndPepper1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

<defs>
 <radialGradient id="redCircle"
                 gradientUnits="objectBoundingBox"
                 fx="70%" fy="70%">
   <stop offset="0%"   style="stop-color:#FFFFFF"/>
   <stop offset="40%"  style="stop-color:#AA0000"/>
   <stop offset="100%" style="stop-color:#660000"/>
 </radialGradient>

 <radialGradient id="blueCircle"
                 gradientUnits="objectBoundingBox"
                 fx="70%" fy="70%">
   <stop offset="0%"   style="stop-color:#FFFFFF"/>
   <stop offset="40%"  style="stop-color:#0000AA"/>
   <stop offset="100%" style="stop-color:#000066"/>
 </radialGradient>

 <path id="jarTop1"
   d="m0,0 a30,30 0 0,1 60,0"
   style="fill:url(#blueCircle)"/>

 <ellipse id="jarTop2"
   cx="0" cy="0" rx="30" ry="10"
   style="fill:url(#blueCircle)"/>

 <path id="upperFace"
   d="m0,0 a80,80 0 0,1 80,0
           a80,80 0 0,0 80,0
           a260,260 0 0,0 -50,-120
           a60,60 0 0,0 -60,0
           a260,260 0 0,0 -50,120"
   style="fill:url(#redCircle)"/>

 <path id="frontFace"
   d="m0,0 a80,80 0 0,1    80,0
           a80,80 0 0,0    80,0
           a160,160 0 0,1 -160,0"
   stroke="black" stroke-width="2"
   style="fill:url(#redCircle)"/>

 <path id="stripedArc1"
   d="m40,0 a160,160 0 0,1 10,-120"
   stroke="white" stroke-width="1"
   style="fill:none"/>

 <path id="stripedArc2"
   d="m75,0 a160,160 0 0,1 0,-125"
```

```
    stroke="white" stroke-width="1"
    style="fill:none"/>

  <path id="stripedArc3"
    d="m85,0 a160,160 0 0,0 0,-125"
    stroke="white" stroke-width="1"
    style="fill:none"/>

  <path id="stripedArc4"
    d="m120,20 a160,160 0 0,0 -10,-140"
    stroke="white" stroke-width="1"
    style="fill:none"/>

  <text id="saltLabel" x="80" y="-60"
     font-size="36">SALT
  </text>

  <text id="pepperLabel" x="55" y="-60"
     font-size="36">PEPPER
  </text>

  <path id="frontFace2"
    d="m0,0 a80,80 0 0,1    80,0
            a80,80 0 0,0    80,0
            a160,160 0 0,1 -160,0"
    stroke="green" stroke-width="4"
    stroke-dasharray="4 4 4 4"
    style="fill:white"/>
</defs>

<g transform="translate(50,260) scale(1.5,1.5)"
   width="100%" height="100%">
     <use xlink:href="#jarTop1"     x="50" y="-130"/>
     <use xlink:href="#jarTop2"     x="80" y="-130"/>
     <use xlink:href="#upperFace"   x="0"  y="0"/>
     <use xlink:href="#stripedArc1" x="0"  y="0"/>
     <use xlink:href="#stripedArc2" x="0"  y="0"/>
     <use xlink:href="#stripedArc3" x="0"  y="0"/>
     <use xlink:href="#stripedArc4" x="0"  y="0"/>
     <use xlink:href="#frontFace"   x="0"  y="0"/>
</g>

<g transform="translate(90,270) scale(.5,.5)"
   width="100%" height="100%">
     <use xlink:href="#frontFace2" x="0"  y="0"/>
</g>

<g transform="translate(50,240) scale(1,1)"
   width="100%" height="100%">
     <use xlink:href="#saltLabel"  x="-4" y="-4"
                                   fill="black"/>
     <use xlink:href="#saltLabel"  x="0" y="0"
                                   fill="white"/>
</g>

<g transform="translate(200,260) scale(1.5,1.5)"
```

```
   width="100%" height="100%">
     <use xlink:href="#jarTop1"     x="50" y="-130"/>
     <use xlink:href="#jarTop2"     x="80" y="-130"/>
     <use xlink:href="#upperFace"   x="0"  y="0"/>
     <use xlink:href="#stripedArc1" x="0"  y="0"/>
     <use xlink:href="#stripedArc2" x="0"  y="0"/>
     <use xlink:href="#stripedArc3" x="0"  y="0"/>
     <use xlink:href="#stripedArc4" x="0"  y="0"/>
     <use xlink:href="#frontFace"   x="0"  y="0"/>
</g>

<g transform="translate(240,270) scale(.5,.5)"
   width="100%" height="100%">
     <use xlink:href="#frontFace2" x="0"  y="0"/>
</g>

<g transform="translate(200,240) scale(1,1)"
   width="100%" height="100%">
     <use xlink:href="#pepperLabel" x="-4" y="-4"
                                    fill="black"/>
     <use xlink:href="#pepperLabel" x="0" y="0"
                                    fill="white"/>
</g>
</svg>
```

Although Listing 4.6 consists of SVG elements that you have seen in previous code samples, the complexity arises from the number of SVG elements whose purpose is not immediately obvious. However, the value arises from the experience that you will gain from creating this type of code sample. As an analogy, recall the movie scene with the mind-numbing "wax on, wax off" routine that vastly increased the student's agility to ward off the blows from his opponents: this code sample can help you achieve greater agility in SVG.

Listing 4.6 consists of a `<defs>` element that contains two radial gradients and multiple `<path>` elements, sometimes with `id` attributes whose values have intuitive names. Notice how the `<path>` elements render elliptic arcs, an example of which is here:

```
<path id="upperFace"
   d="m0,0 a80,80 0 0,1 80,0
           a80,80 0 0,0 80,0
           a260,260 0 0,0 -50,-120
           a60,60 0 0,0 -60,0
           a260,260 0 0,0 -50,120"
   style="fill:url(#redCircle)"/>
```

The preceding code block (which is the most complex `<path>` element of the code sample) specifies five contiguous elliptic arcs that represent the "upper face" of the shakers.

The next portion of Listing 4.6 consists of six `<g>` elements containing one or more `<use>` elements that reference SVG elements that are defined in the `<defs>` element.

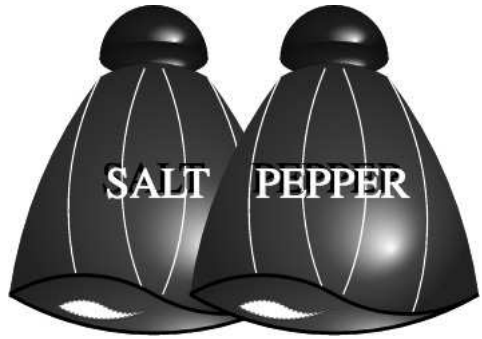

FIGURE 4.6: SVG salt and pepper shakers.

Figure 4.6 displays the graphics image that is rendered by the code in the HTML Web page in Listing 4.6.

BASIC BEZIER CURVES IN SVG

The code sample in this section shows how to render quadratic Bezier curves and cubic Bezier curves.

Listing 4.7 displays the contents of `BasicBezierCurves1.svg`, which illustrates how to render Bezier curves.

LISTING 4.7: BasicBezierCurves1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <defs>
    <linearGradient id="pattern1"
                    x1="0%" y1="100%" x2="100%" y2="0%">
      <stop offset="0%"   stop-color="yellow"/>
      <stop offset="40%"  stop-color="red"/>
      <stop offset="80%"  stop-color="blue"/>
    </linearGradient>

   <linearGradient id="pattern2"
                   gradientTransform="rotate(90)">
      <stop offset="0%"   stop-color="#C0C040"/>
      <stop offset="30%"  stop-color="#303000"/>
      <stop offset="60%"  stop-color="#FF0F0F"/>
      <stop offset="90%"  stop-color="#101000"/>
   </linearGradient>
  </defs>

  <g transform="scale(1.5,0.5)">
   <path d="m 0,50 C 400,200 200,-150 100,350"
```

```
       stroke="black" stroke-width="4"
       fill="url(#pattern1)"/>
</g>

<g transform="translate(50,50)">
  <g transform="scale(0.5,1)">
   <path d="m 50,50 C 400,100 200,200 100,20"
         fill="red" stroke="black" stroke-width="4"/>
  </g>

  <g transform="scale(1,1)">
   <path d="m 50,50 C 400,100 200,200 100,20"
         fill="yellow" stroke="black" stroke-width="4"/>
  </g>
</g>

<g transform="translate(-50,50)">
  <g transform="scale(1,2)">
   <path d="M 50,50 C 400,100 200,200 100,20"
         fill="blue" stroke="black" stroke-width="4"/>
  </g>
</g>

<g transform="translate(-50,50)">
 <g transform="scale(0.5, 0.5) translate(195,345)">
  <path
    d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
    fill="blue"
    style="stroke:#880088;stroke-width:4;"/>
 </g>

 <g transform="scale(0.5, 0.5) translate(185,335)">
  <path
    d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
    fill="url(#pattern2)"
    style="stroke:#880088;stroke-width:4;"/>
 </g>

 <g transform="scale(0.5, 0.5) translate(180,330)">
  <path
    d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
    fill="blue"
    style="stroke:#880088;stroke-width:4;"/>
 </g>

 <g transform="scale(0.5, 0.5) translate(170,320)">
  <path
    d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
    fill="url(#pattern2)"
    style="stroke:black;stroke-width:4;"/>
 </g>
</g>

<g transform="scale(0.8,1) translate(380,120)">
 <path
   d="M0,0 C200,150 400,300 20,250"
```

```
      fill="url(#pattern2)"
      style="stroke:blue;stroke-width:4;"/>
  </g>

  <g transform="translate(350,-80)">
   <path
     d="M200,150 C0,0 400,300 20,250"
     fill="url(#pattern2)"
     style="stroke:blue;stroke-width:4;"/>
  </g>
</svg>
```

Listing 4.7 contains a `<defs>` element with two linear gradients, followed by six `<g>` elements that contain a `<path>` element that defines a cubic Bezier curve. Each `<g>` element also specifies a `transform` attribute with a value of `scale` or `translate`, and you can easily modify this code sample to specify a value of `rotate` or `skew` for the `transform` attribute.

In addition, you can also render semicircles using Bezier curves, as shown in this code snippet:

```
<path d="M100,200 C100,100 250,100 250,200 S400,300 400,200"
      fill="red" />
```

Figure 4.7 displays a screenshot of multiple scaled cubic Bezier curves based on the code in Listing 4.7.

RENDERING TEXT ALONG AN SVG <PATH> ELEMENT

The SVG `<path>` element supports both quadratic Bezier curves and cubic Bezier curves. In addition, the SVG `<path>` element enables you to render a text string that follows any path that is defined via the SVG `<path>` element.

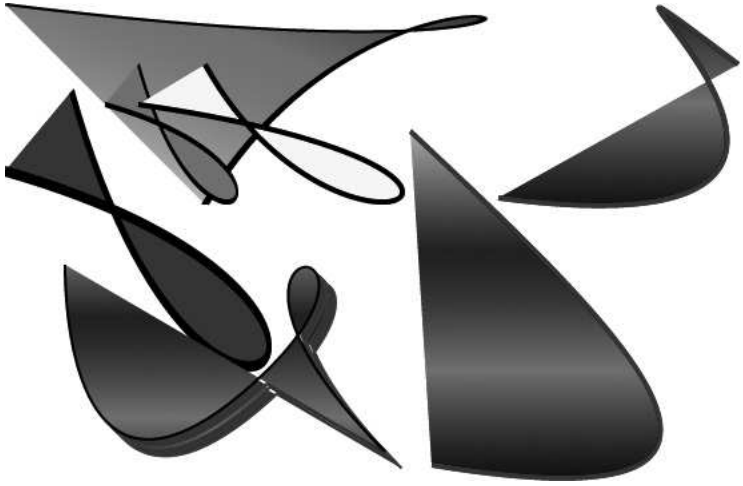

FIGURE 4.7: Multiple scaled cubic Bezier curves.

Listing 4.8 displays the contents of the document TextOnQBezierPath1.svg, which illustrates how to render a text string along the path of a quadratic Bezier curve.

LISTING 4.8: TextOnQBezierPath1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
<defs>
  <path id="pathDefinition"
        d="m0,0 Q100,0 200,200 T300,200 z"/>
</defs>

<g transform="translate(30,50)">
   <use xlink:href="#pathDefinition"
        x="0" y="0" fill="#c44"/>

   <text id="textStyle" fill="red"
         stroke="blue" stroke-width="1" font-size="32">
     <textPath xlink:href="#pathDefinition">
       Sample Text that follows a path specified by a Quadratic
       Bezier curve
     </textPath>
   </text>
 </g>
</svg>
```

The SVG <defs> element in Listing 4.8 contains an SVG <path> element that defines a quadratic Bezier curve (note the Q in the d attribute). This SVG <path> element has an id attribute whose value is pathDefinition, which is referenced later in this code sample.

The SVG <g> element starts with an SVG <use> element that references the quadratic Bezier curve that is defined in the SVG <defs> element. The SVG <g> element also contains an SVG <text> element that specifies a text string to render, as well as an SVG <textPath> child element that specifies the path along which the text is rendered, as shown here:

```
<textPath xlink:href="#pathDefinition">
  Sample Text that follows a path specified by a Quadratic Bezier
  curve
</textPath>
```

Notice that the SVG <textPath> element contains the attribute xlink:href whose value is pathDefinition, which is also the id of the SVG <path> element that is defined in the SVG <defs> element. As a result, the text string is rendered along the path of a quadratic Bezier curve instead of rendering the text string horizontally (which is the default behavior).

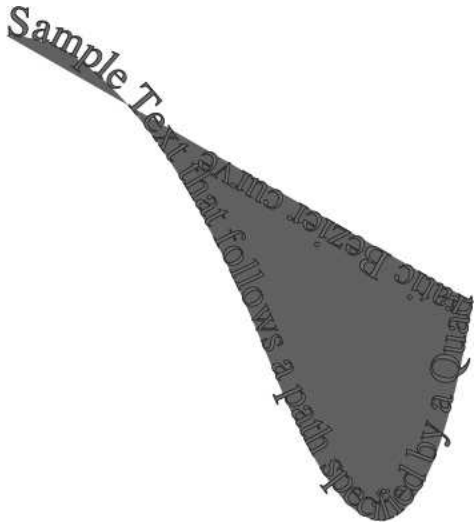

FIGURE 4.8: SVG text on a quadratic Bezier.

Figure 4.8 displays the result of rendering `TextOnQBezierPath1.svg`, which renders a text string along the path of a quadratic Bezier curve in a browser.

BEZIER CURVES AND TRANSFORMS

SVG supports quadratic and cubic Bezier curves that you can render with linear gradient or radial gradients. You can also concatenate multiple Bezier curves using an SVG `<path>` element. Listing 4.9 displays the contents of `TransBezierCurves1.svg`, which illustrates how to render various Bezier curves. Note that the transform-related effects are discussed later in this chapter.

LISTING 4.9: TransBezierCurves1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

 <svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
  <!-- <defs> element omitted for brevity -->

  <g transform="scale(1.5,0.5)">
   <path d="m 0,50 C 400,200 200,-150 100,350"
         stroke="black" stroke-width="4"
         fill="url(#pattern1)"/>
  </g>

  <g transform="translate(50,50)">
```

```
    <g transform="scale(0.5,1)">
     <path d="m 50,50 C 400,100 200,200 100,20"
           fill="red" stroke="black" stroke-width="4"/>
    </g>

    <g transform="scale(1,1)">
     <path d="m 50,50 C 400,100 200,200 100,20"
           fill="yellow" stroke="black" stroke-width="4"/>
    </g>
  </g>

  <g transform="translate(-50,50)">
    <g transform="scale(1,2)">
     <path d="M 50,50 C 400,100 200,200 100,20"
           fill="blue" stroke="black" stroke-width="4"/>
    </g>
  </g>

  <g transform="translate(-50,50)">
   <g transform="scale(0.5, 0.5) translate(195,345)">
    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="blue" style="stroke:#880088;stroke-width:4;"/>
   </g>

   <g transform="scale(0.5, 0.5) translate(185,335)">
    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="url(#pattern2)"
          style="stroke:#880088;stroke-width:4;"/>
   </g>

   <g transform="scale(0.5, 0.5) translate(180,330)">
    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
      fill="blue" style="stroke:#880088;stroke-width:4;"/>
   </g>

   <g transform="scale(0.5, 0.5) translate(170,320)">
    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="url(#pattern2)" style="stroke:black;stroke-width:4;"/>
   </g>
  </g>

  <g transform="scale(0.8,1) translate(380,120)">
   <path d="M0,0 C200,150 400,300 20,250"
         fill="url(#pattern2)" style="stroke:blue;stroke-width:4;"/>
  </g>

  <g transform="scale(2.0,2.5) translate(150,-80)">
   <path d="M200,150 C0,0 400,300 20,250"
         fill="url(#pattern2)" style="stroke:blue;stroke-width:4;"/>
  </g>
</svg>
```

Listing 4.9 contains an SVG <defs> element that defines two linear gradients, followed by ten SVG <path> elements, each of which renders a cubic Bezier curve. The SVG <path> elements are enclosed in SVG <g> elements

whose transform attribute contains the SVG scale() function or the SVG translate() function (or both).

The first SVG <g> element invokes the SVG scale() function to scale the cubic Bezier curve that is specified in an SVG <path> element, as shown here:

```
<g transform="scale(1.5,0.5)">
   <path d="m 0,50 C 400,200 200,-150 100,350"
         stroke="black" stroke-width="4"
         fill="url(#pattern1)"/>
</g>
```

The preceding cubic Bezier curve has an initial point (0,50), with control points (400,200) and (200,−150), followed by the second control point (100,350). The Bezier curve is black, with a width of 4, and its fill attribute is the color defined in the <linearGradient> element (whose id attribute has value pattern1) that is defined in the SVG <defs> element.

Since the remaining SVG <path> elements are similar to the first SVG <path> element, we can skip their detailed description.

Figure 4.9 displays the result of rendering the elliptic arcs that are defined in the SVG document TransBezierCurves1.svg in a browser.

THE SVG <CLIPPATH> ELEMENT WITH BEZIER CURVES

Listing 4.10 contains a lengthy portion of code from DrcBezier3D CircleCB1.svg that illustrates how to combine the SVG <clipPath> element with Bezier curves and other SVG elements.

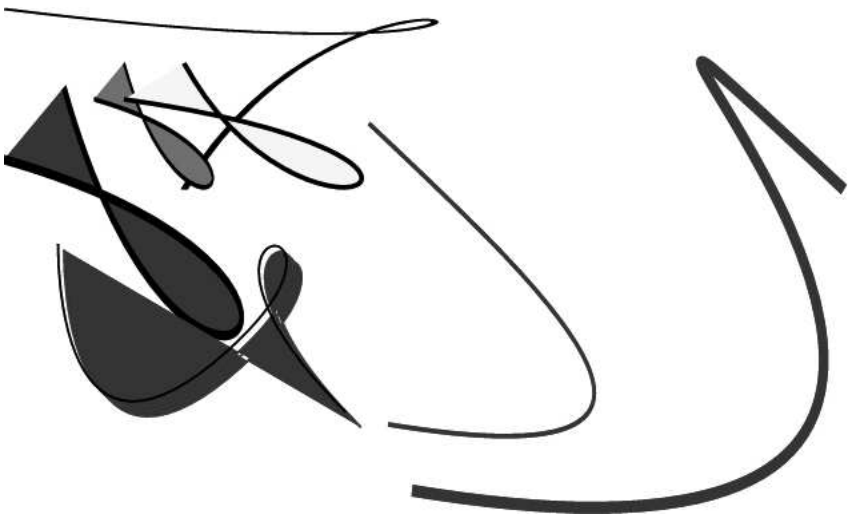

FIGURE 4.9: SVG Bezier curves.

LISTING 4.10: DrcBezier3DCircleCB1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">

<defs>
  <pattern id="multiPattern"
           width="50" height="50"
           patternUnits="userSpaceOnUse">

   <linearGradient id="repeatedGradientDefinition"
      x1="0" y1="0" x2="50" y2="0"
      gradientUnits="userSpaceOnUse" spreadMethod="repeat">
      <stop offset="0%"   style="stop-color:#FF0000"/>
      <stop offset="100%" style="stop-color:#000000"/>
   </linearGradient>

   <rect
      style="fill:url(#repeatedGradientDefinition)"
      x="0" y="0" width="50" height="50"/>
   </pattern>

  <clipPath id="clipPathDefinition"
            clipPathUnits="userSpaceOnUse">
    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="blue"
          stroke-dasharray="2 2 2 2"
          style="stroke:#880088;stroke-width:4;"/>

    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="white"
          stroke-dasharray="8 4 8 4"
          style="stroke:#880088;stroke-width:4;"/>

    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="blue"
          stroke-dasharray="2 2 2 2"
          style="stroke:#880088;stroke-width:4;"/>

    <path d="m20,20 C20,50 20,450 300,200 s-150,-250 200,100"
          fill="red"
          stroke-dasharray="8 4 8 4"
          style="stroke:black;stroke-width:4;"/>

    <path d="m0,0 l50,50 l50,-50"/>

    <rect   x="30"     y="50"
            width="40" height="80"/>

    <circle cx="100" cy="50"  r="10"/>
    <circle cx="150" cy="90"  r="40"/>
    <circle cx="200" cy="125" r="10"/>
```

```
    <ellipse cx="250" cy="150" rx="40" ry="20"/>
  </clipPath>

  <filter
     id="blurFilter1"
     filterUnits="objectBoundingBox"
     x="0" y="0"
     width="100%" height="100%">
     <feGaussianBlur stdDeviation="2"/>
  </filter>

  <radialGradient id="redCircle"
                  gradientUnits="objectBoundingBox"
                  fx="30%" fy="30%">
    <stop offset="0%"   style="stop-color:#FFFFFF"/>
    <stop offset="40%"  style="stop-color:#AA0000"/>
    <stop offset="100%" style="stop-color:#660000"/>
  </radialGradient>

  <pattern id="checkerPattern"
           width="20" height="20"
           patternUnits="userSpaceOnUse">
     <rect fill="black"
           x="0"  y="0"  width="10" height="10"/>
     <rect fill="yellow"
           x="10" y="0"  width="10" height="10"/>
     <rect fill="yellow"
           x="0"  y="10" width="10" height="10"/>
     <rect fill="black"
           x="10" y="10" width="10" height="10"/>
  </pattern>

  <rect   id="3DRedCircle" x="0" y="0" width="40" height="40"
          style="fill:url(#checkerCircle)"/>

  <pattern id="checkerPatternNew"
           width="40" height="40"
           patternUnits="userSpaceOnUse">

  <circle id="3DRedCircleOld" cx="20" cy="20" r="20"
          style="fill:url(#redCircle)"/>

  <circle id="3DRedCircleOld2" cx="20" cy="20" r="20"
          style="fill:url(#checkerPattern);opacity:.5"/>
  </pattern>
</defs>

<g transform="translate(50,50)"
   clip-path="url(#clipPathDefinition)"
   stroke="black" fill="none">

  <rect
     style="fill:url(#checkerPatternNew)"
     x="0" y="0" width="400" height="400"/>
```

```
   <use xlink:href="#3DRedCircle"   x="0" y="0"/>
</g>
</svg>
```

Listing 4.10 is a very lengthy code sample that contains most of the SVG elements that you have seen in previous code samples as well as previous chapters. Specifically, Listing 4.10 contains SVG <pattern>, <linearGradient>, <clipPath>, <filter>, and <radialGradient> elements. The novelty in this example is the overall visual effect, and it's probably quicker for you to launch the code sample to view the graphics effect, and then look at the corresponding code for each "component" in Listing 4.10.

Figure 4.10 displays the result of rendering DrcBezier3DCircleCB1.svg in a Chrome browser on a MacBook Pro.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The companion disc contains MultiNoUTurn1.svg, which illustrates how to render a traffic sign using SVG transforms (such as scale, rotate, and skew). It also includes AbstractFace4.svg, which illustrates how to render abstract graphics effects using elliptic arcs. The companion disc contains the SVG document TransformedPeelingTetrahedron3.svg, which illustrates how to combine SVG transforms, elliptic arcs, and gradient effects. It also contains PartialBlueSphereCB5.svg, which illustrates how to render a cutaway view of a sphere using SVG <pattern> elements, elliptic arcs, and gradient shading. The companion disc also includes TurbFilterShCBQRect4.svg, which illustrates how to apply SVG filters, SVG <pattern> elements, and SVG transforms to checkerboard shapes that are superimposed with quadratic Bezier curves.

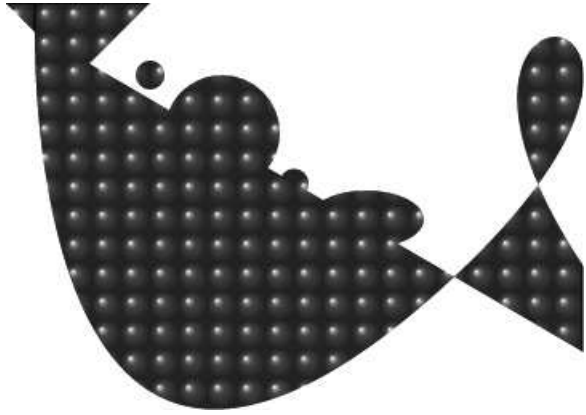

FIGURE 4.10: SVG <clipPath> in a Chrome browser on a MacBook Pro.

SUMMARY

This chapter showed you how to render ellipses using gradient shading. You then saw how to render elliptic arcs to create some common objects. Next you learned how to render quadratic Bezier curves and cubic Bezier curves, and how to "concatenate" multiple Bezier curves using the SVG `<path>` element. Finally, you saw how to render a text string along the path of a Bezier curve.

The next chapter delves into how to combine CSS3 with SVG documents to create 2D and 3D animation effects.

CHAPTER 5

INTRODUCTION TO CSS3 GRAPHICS AND ANIMATION

This chapter introduces various aspects of CSS3, such as 2D/3D graphics and 2D/3D animation. In some cases, CSS3 concepts are presented without code samples due to space limitations; however, those concepts are included because it's important for you to be aware of their existence. By necessity, this chapter assumes that you have a moderate understanding of CSS, which means that you know how to set properties in CSS selectors. If you are unfamiliar with CSS selectors, there are many introductory articles available through an Internet search. If you are convinced that CSS operates under confusing and seemingly arcane rules, then it's probably worth your while to read an online article about CSS box rules, after which you will have a better understanding of the underlying logic of CSS.

The first part of this chapter contains code samples that illustrate how to create shadow effects, how to render rectangles with rounded corners, and also how to use linear and radial gradients. The second part of this chapter covers CSS3 transforms (scale, rotate, skew, and translate), along with code samples that illustrate how to apply transforms to HTML elements and to binary files.

The third part of this chapter covers CSS3 3D graphics and animation effects, and the fourth part of this chapter briefly discusses `CSS3 Media Queries`, which enable you to detect some characteristics of a device, and therefore render an HTML5 Web page based on those properties.

You can launch the code samples in this chapter in a modern browser on a desktop or a laptop; you can also view them on mobile devices, provided that you launch them in a browser that supports the CSS3 features that are used in the code samples. For your convenience, many of the code samples in this chapter are accompanied by screenshots of the code samples on an iPad3, which enables you to compare those screenshots with the corresponding images that are rendered on modern browsers on desktops and laptops.

An open source project with a plethora of code samples (think "swatches") for graphics and animation effects using "pure" CSS3 (no JavaScript, Canvas, or SVG) is here: *https://github.com/ocampesato/css3-graphics*.

CSS3 SUPPORT AND BROWSER-SPECIFIC PREFIXES FOR CSS3

Before we delve into the details of CSS3, there are two important details that you need to know about defining CSS3-based selectors for HTML pages. First, you need to know the CSS3 features that are available in different browsers. One of the best websites for determining browser support for CSS3 features is here: *http://caniuse.com/*.

The preceding link contains tabular information regarding CSS3 support in IE, Firefox, Safari, Chrome, and Opera.

A useful tool that checks the HTML, CSS, and JavaScript features that are supported in a browser is `Modernizr`, and its home page is here: *https://modernizr.com/*.

Another highly useful tool that checks for CSS3 feature support is `Enhance.js`, which tests browsers to determine whether or not they can support a set of essential CSS and JavaScript properties, and then delivers features to those browsers that satisfies the test. You can download `Enhance.js` here: *http://filamentgroup.com/lab/introducing_enhancejs_smarter_safer_apply_progressive_enhancement/*.

The second detail that you need to know is that many CSS3 properties no longer require browser-specific prefixes in order for them to work correctly. However, if you do need such prefixes, the prefixes `-ie-`, `-moz-`, and `-o-` are for Internet Explorer, Firefox, and Opera, respectively. As an illustration, the following code block shows examples of these prefixes:

```
-ie-webkit-border-radius: 8px;
-moz-webkit-border-radius: 8px;
-o-webkit-border-radius: 8px;
border-radius: 8px;
```

In your CSS selectors, specify the attributes with browser-specific prefixes before the "generic" attribute, which serves as a default choice in the event that the browser-specific attributes are not selected. The CSS3 code samples in this book contain prefix-free code samples that reduce the size of the CSS stylesheets. If you need CSS stylesheets that work on multiple browsers (for current versions as well as older versions), there are essentially two options available. One option involves manually adding the CSS3 code with all the required browser-specific prefixes, which can be tedious to maintain and also error-prone. Another option is to use CSS toolkits or frameworks (discussed in the next chapter) that can programmatically generate the CSS3 code that contains all browser-specific prefixes.

Finally, an extensive list of browser-prefixed CSS properties is here: *http://peter.sh/experiments/vendor-prefixed-css-property-overview/*.

QUICK OVERVIEW OF CSS3 FEATURES

CSS3 adopts a modularized approach for extending existing CSS2 functionality as well as supporting new functionality. As such, CSS3 can be logically divided into the following categories:

- Backgrounds/Borders
- Color
- Media Queries
- Multicolumn layout
- Selectors

With CSS3 you can create boxes with rounded corners and shadow effects; create rich graphics effects using linear and radial gradients; detect portrait and landscape mode; detect the type of mobile device using media query selectors; and produce multicolumn text rendering and formatting.

In addition, CSS3 enables you to define sophisticated node selection rules in selectors using pseudo-classes, first or last child (`first-child`, `last-child`, `first-of-type`, and `last-of-type`), and also pattern-matching tests for attributes of elements. Several sections in this chapter contain examples of how to create such selection rules.

CSS3 PSEUDO-CLASSES, ATTRIBUTE SELECTION, AND RELATIONAL SYMBOLS

This brief section contains examples of some pseudo-classes, followed by snippets that show you how to select elements based on the relative position of text strings in various attributes of those elements. Although this section focuses on the `nth-child()` pseudo-class, you will become familiar with various other CSS3 pseudo-classes, and in the event that you need to use those pseudo-classes, a link is provided at the end of this section which contains more information and examples that illustrate how to use them.

CSS3 supports an extensive and rich set of pseudo-classes, including `nth-child()`, along with some of its semantically related "variants," such as `nth-of-type()`, `nth-first-of-type()`, `nth-last-of-type()`, and `nth-last-child()`.

CSS3 also supports Boolean selectors (which are also pseudo-classes) such as `empty`, `enabled`, `disabled`, and `checked`, which are very useful for `Form`-related HTML elements. One other pseudo-class is `not()`, which returns a set of elements that do not match the selection criteria.

CSS3 uses the meta-characters `^`, `$`, and `*` (followed by the = symbol) in order to match an initial, terminal, or arbitrary position for a text string. If you are familiar with the Unix utilities `grep` and `sed`, as well as the `vi` text editor, then these meta-characters are very familiar to you.

CSS3 Pseudo-Classes

The CSS3 `nth-child()` is a very powerful and useful pseudo-class, and it has the following form:

```
nth-child (insert-a-keyword-or-linear-expression-here)
```

The following list provides various examples of using the `nth-child()` pseudo-class in order to match various subsets of child elements of an HTML `<div>` element (which can be substituted by other HTML elements as well):

`div:nth-child(1)`: matches the first child element
`div:nth-child(2)`: matches the second child element
`div:nth-child(even)`: matches the even child elements
`div:nth-child(odd)`: matches the odd child elements

The interesting and powerful aspect of the `nth-child()` pseudo-class is its support for linear expressions of the form `an+b`, where `a` is a positive integer and `b` is a non-negative integer, as shown here (using an HTML5 `<div>` element):

`div:nth-child(3n)`: matches every third child, starting from position 0
`div:nth-child(3n+1)`: matches every third child, starting from position 1
`div:nth-child(3n+2)`: matches every third child, starting from position 2

CSS3 Attribute Selection

You can specify CSS3 selectors that select HTML elements as well as HTML elements based on the value of an attribute of an HTML element using various regular expressions. For example, the following selector selects `img` elements whose `src` attribute starts with the text string `Laurie`, and then sets the `width` attribute and the `height` attribute of the selected `img` elements to `100px`:

```
img[src^="Laurie"] {
  width: 100px; height: 100px;
}
```

The preceding CSS3 selector is useful when you want to set different dimensions to images based on the name of the images (`Laurie`, `Shelly`, `Steve`, and so forth).

The following HTML `<img>` elements match the preceding selector:

```
<img src="Laurie" width="200" height="200" />
<img src="Laurie123" width="200" height="200" />
```

On the other hand, the following HTML `<img>` elements do not match the preceding selector:

```
<img src="3Laurie" width="200" height="200" />
<img src="3Laurrie" width="200" height="200" />
```

The following selector selects HTML `img` elements whose `src` attribute ends with the text string `jpeg`, and then sets the `width` attribute and the `height` attribute of the selected `img` elements to `150px`:

```
img[src$="jpeg"] {
  width: 150px; height: 150px;
}
```

The preceding CSS3 selector is useful when you want to set different dimensions to images based on the type of the binary images (`jpg`, `png`, `jpeg`, and so forth).

The following selector selects HTML `img` elements whose `src` attribute contains any occurrence of the text string `baby`, and then sets the `width` attribute and the `height` attribute of the selected HTML `img` elements to `200px`:

```
img[src*="baby"] {
  width: 200px; height: 200px;
}
```

The preceding CSS3 selector is useful when you want to set different dimensions to images based on the "classification" of the images (`mybaby`, `yourbaby`, `babygirl`, `babyboy`, and so forth).

If you want to learn more about patterns (and their descriptions) that you can use in CSS3 selectors, an extensive list is available here: *http://www.w3.org/TR/css3-selectors*.

This concludes part one of this chapter, and the next section delves into CSS3 graphics-oriented effects, such as rounded corners and shadow effects.

CSS3 SHADOW EFFECTS AND ROUNDED CORNERS

CSS3 shadow effects are useful for creating vivid visual effects with simple selectors. You can use shadow effects for text as well as rectangular regions. CSS3 also enables you to easily render rectangles with rounded corners, so you do not need binary files in order to create this effect.

Specifying Colors with RGB and HSL

Before we delve into the interesting features of CSS3, you need to know how to represent colors. One method is to use `(R,G,B)` triples, which represent the `Red`, `Green`, and `Blue` components of a color. For instance, the triples `(255,0,0)`, `(255,255,0)`, and `(0,0,255)` represent the colors `Red`, `Yellow`, and `Blue`. Other ways of specifying the color include: the hexadecimal triples `(FF,0,0)` and `(F,0,0)`; the decimal triple `(100%,0,0)`; or the string `#F00`. You can also use `(R,G,B,A)`, where the fourth component specifies the opacity, which is a decimal number between `0` (invisible) to `1` (opaque) inclusive.

However, there is also the `HSL` (`Hue`, `Saturation`, and `Luminosity`) representation of colors, where the first component is an angle between

0 and 360 (0 degrees is north), and the other two components are percentages between 0 and 100. For instance, the three triples (0,100%,50%), (120,100%,50%), and (240, 100%, 50%) represent the colors Red, Green, and Blue, respectively.

The code samples in this book use (R,G,B) and (R,G,B,A) for representing colors, but you can perform an Internet search to obtain more information regarding HSL.

CSS3 and Text Shadow Effects

A shadow effect for text can make a Web page look more vivid and appealing, and many websites look better with shadow effects that are not overpowering for users (unless you specifically need to do so).

Listing 5.1 displays the contents of the HTML5 page TextShadow1.html that illustrate how to render text with a shadow effect, and Listing 5.2 displays the contents of the CSS stylesheet TextShadow1.css that is referenced in Listing 5.1.

LISTING 5.1: TextShadow1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>CSS Text Shadow Example</title>
  <link href="TextShadow1.css" rel="stylesheet" type="text/css">
</head>

<body>
  <div id="text1">Line One Shadow Effect</div>
  <div id="text2">Line Two Shadow Effect</div>
  <div id="text3">Line Three Vivid Effect</div>
  <div id="text4">
    <span id="dd">13</span>
    <span id="mm">August</span>
    <span id="yy">2016</span>
  </div>
  <div id="text5">
    <span id="dd">13</span>
    <span id="mm">August</span>
    <span id="yy">2016</span>
  </div>
  <div id="text6">
    <span id="dd">13</span>
    <span id="mm">August</span>
    <span id="yy">2016</span>
  </div>
</body>
</html>
```

The code in Listing 5.1 is straightforward: there is a reference to the CSS stylesheet TextShadow1.css that contains two CSS selectors. One selector specifies how to render the HTML element whose id attribute

has value `text1`, and the other selector matches the HTML `<div>` element whose `id` attribute is `text2`. Although the CSS3 `rotate()` function is included in this example, we'll defer a more detailed discussion of this function until later in this chapter.

LISTING 5.2: TextShadow1.css

```
#text1 {
  font-size: 24pt;
  text-shadow: 2px 4px 5px #00f;
}

#text2 {
  font-size: 32pt;
  text-shadow: 0px 1px 6px #000,
               4px 5px 6px #f00;
}

#text3 {
  font-size: 40pt;
  text-shadow: 0px 1px 6px  #fff,
               2px 4px 4px  #0ff,
               4px 5px 6px  #00f,
               0px 0px 10px #444,
               0px 0px 20px #844,
               0px 0px 30px #a44,
               0px 0px 40px #f44;
}

#text4 {
  position: absolute;
  top: 200px;
  right: 200px;
  font-size: 48pt;
  text-shadow: 0px 1px 6px  #fff,
               2px 4px 4px  #0ff,
               4px 5px 6px  #00f,
               0px 0px 10px #000,
               0px 0px 20px #448,
               0px 0px 30px #a4a,
               0px 0px 40px #fff;
  transform: rotate(-90deg);
}

#text5 {
  position: absolute;
  left: 0px;
  font-size: 48pt;
  text-shadow: 2px 4px 5px #00f;
  transform: rotate(-10deg);
}

#text6 {
  float: left;
  font-size: 48pt;
```

```
  text-shadow: 2px 4px 5px #f00;
  transform: rotate(-170deg);
}

/* 'transform' is explained later */
#text1:hover, #text2:hover, #text3:hover,
#text4:hover, #text5:hover, #text6:hover {
 transform : scale(2) rotate(-45deg);
}
```

The first selector in Listing 5.2 specifies a `font-size` of `24` and a `text-shadow` that renders text with a blue background (represented by the hexadecimal value `#00f`). The attribute `text-shadow` specifies (from left to right) the horizontal offset, the vertical offset, the blur radius, and the color of the shadow. The second selector specifies a font-size of `32` and a red shadow background (`#f00`). The third selector creates a richer visual effect by specifying multiple components in the `text-shadow` property, which were chosen by experimenting with effects that are possible with different values in the various components.

The final CSS3 selector creates an animation effect whenever users hover over any of the six text strings, and the details of the animation will be deferred until later in this chapter.

Figure 5.1 displays the result of matching the selectors in the CSS stylesheet `TextShadow1.css` with the HTML `<div>` elements in the HTML page `TextShadow1.html`.

CSS3 and Box Shadow Effects

You can also apply a shadow effect to a box that encloses a text string, which can be effective in terms of drawing attention to specific parts of a Web page.

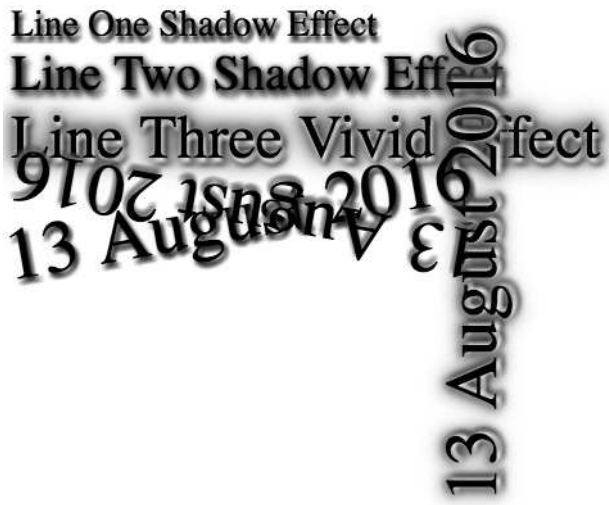

FIGURE 5.1: CSS3 text shadow effects.

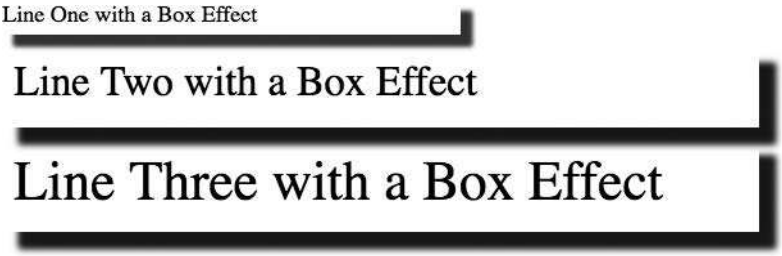

FIGURE 5.2: CSS3 box shadow effect.

However, the same caveat regarding overuse applies to box shadows (as well as performance-related issues).

The HTML page `BoxShadow1.html` and `BoxShadow1.css` are not shown here, but they are available on the companion disc, and together they render a box shadow effect.

The key property is the `box-shadow` property, as shown here:

```
#box1 {
  position:relative;top:10px;
  width: 50%;
  height: 30px;
  font-size: 20px;
  box-shadow: 10px 10px 5px #800;
```

Figure 5.2 displays a box shadow effect.

CSS3 and Rounded Corners

Web developers have waited a long time for rounded corners in CSS, and CSS3 makes it very easy to render boxes with rounded corners. Listing 5.3 displays the contents of the HTML Web page `RoundedCorners1.html`, which renders text strings in boxes with rounded corners, and Listing 5.4 displays the CSS file `RoundedCorners1.css`.

LISTING 5.3: RoundedCorners1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>CSS Text Shadow Example</title>
  <link href="RoundedCorners1.css" rel="stylesheet" type="text/css">
</head>

<body>
  <div id="outer">
    <a href="#" class="anchor">Text Inside a Rounded Rectangle</a>
  </div>
  <div id="text1">Line One of Text with a Shadow Effect</div>
  <div id="text2">Line Two of Text with a Shadow Effect</div>
</body>
</html>
```

Listing 5.3 contains a reference to the CSS stylesheet `RoundedCorners1.css` that contains three CSS selectors that match the elements whose `id` attribute has value `anchor`, `text1`, and `text2`, respectively. The CSS selectors defined in `RoundedCorners1.css` create visual effects, and as you will see, the `hover` pseudo-selector enables you to create animation effects.

LISTING 5.4: RoundedCorners1.css

```
a.anchor:hover {
background: #00F;
}

a.anchor {
background: #FF0;
font-size: 24px;
font-weight: bold;
padding: 4px 4px;
color: rgba(255,0,0,0.8);
text-shadow: 0 1px 1px rgba(0,0,0,0.4);
transition: all 2.0s ease;
border-radius: 8px;
}
```

Listing 5.4 contains the selector `a.anchor:hover` that changes the text color from yellow (`#FF0`) to blue (`#00F`) during a two-second interval whenever users hover over any anchor element with their mouse.

The selector `a.anchor` contains various attributes that specify the dimensions of the box that encloses the text in the `<a>` element, along with two new pairs of attributes. The first pair specifies the `transition` attribute, which we will discuss later in this chapter. The second pair specifies the `border-radius` attribute whose value is `8px`, which determines the radius (in pixels) of the rounded corners of the box that encloses the text in the `<a>` element. The last two selectors are identical to the selectors in Listing 5.1.

Figure 5.3 displays the result of matching the selectors that are defined in the CSS stylesheet `RoundedCorners1.css` with elements in the HTML page `RoundedCorners1.html` in a landscape-mode screenshot taken from an iPad3.

CSS3 GRADIENTS

CSS3 supports linear gradients and radial gradients, which enable you to create gradient effects that are as visually rich as gradients in other technologies such as SVG. The code samples in this section illustrate how to define

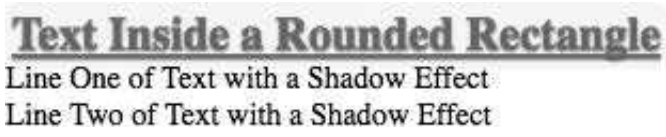

FIGURE 5.3: CSS3 rounded corners effect on an iPad3.

linear gradients and radial gradients in CSS3 and then match them to HTML elements.

Linear Gradients

CSS3 linear gradients require one or more "color stops," each of which specifies a start color, an end color, and a rendering pattern. Browsers support the following syntax to define a linear gradient:

- A direction (or an angle)
- One or more color stops

If the *direction* is specified as `to bottom`, the gradient will be rendered horizontally from top to bottom. If the *direction* property is specified as `to right`, the gradient will be rendered vertically from left to right. If the *direction* property is specified as `to bottom right`, the gradient will be rendered diagonally from upper left to lower right.

You can specify an angle instead of the predefined directions: the angle value is the angle between a horizontal line and the gradient line. For example, the following code snippet renders a diagonal linear gradient that forms a thirty-degree angle with the horizontal axis:

```
background: linear-gradient(30deg, red, yellow);
```

The color varies from red to yellow, and is displayed from upper left to lower right.

Listing 5.5 displays the contents of `LinearGradient1.html` and Listing 5.6 displays the contents of `LinearGradient1.css`, which illustrate how to use linear gradients with text strings that are enclosed in `<p>` elements and an `<h3>` element.

LISTING 5.5: LinearGradient1.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>CSS Linear Gradient Example</title>
  <link href="LinearGradient1.css" rel="stylesheet" type="text/css">
</head>

<body>
  <div id="outer">
    <p id="line1">line 1 with a linear gradient</p>
    <p id="line2">line 2 with a linear gradient</p>
    <p id="line3">line 3 with a linear gradient</p>
    <p id="line4">line 4 with a linear gradient</p>
    <p id="outline">line 5 with Shadow Outline</p>
    <h3><a href="#">A Line of Gradient Text</a></h3>
  </div>
</body>
</html>
```

Listing 5.5 is a simple Web page containing four <p> elements and one <h3> element. Listing 5.5 also references the CSS stylesheet LinearGradient1.css that contains CSS selectors that match the four <p> elements and the <h3> element in Listing 5.5.

LISTING 5.6: LinearGradient1.css

```
#line1 {
width: 50%;
font-size: 32px;
background-image: linear-gradient(to bottom, #fff, #f00);
border-radius: 4px;
}

#line2 {
width: 50%;
font-size: 32px;
background-image: linear-gradient(to bottom left, #fff, #f00);
border-radius: 4px;
}

#line3 {
width: 50%;
font-size: 32px;
background-image: linear-gradient(to bottom, #f00, #00f);
border-radius: 4px;
}

#line4 {
width: 50%;
font-size: 32px;
background-image: linear-gradient(to bottom left, #f00, #00f);
border-radius: 4px;
}

#outline {
font-size: 2.0em;
font-weight: bold;
color: #fff;
text-shadow: 1px 1px 1px rgba(0,0,0,0.5);
}

h3 {
width: 50%;
position: relative;
margin-top: 0;
font-size: 32px;
font-family: helvetica, ariel;
}

h3 a {
position: relative;
color: red;
text-decoration: none;
background-image: linear-gradient(to bottom, #fff, #f00);
}
```

```
h3:after {
content:"This is a Line of Text";
color: blue;
}
```

The first selector in Listing 5.6 specifies a `font-size` of `32` for text, a `border-radius` of `4` (which renders rounded corners), and a linear gradient that varies from `white` to `red`, as shown here:

```
#line1 {
width: 50%;
font-size: 32px;
background-image: linear-gradient(to bottom, #fff, #f00);
border-radius: 4px;
}
```

As you can see, the first selector contains two standard attributes without this prefix. Since the next three selectors in Listing 5.6 are similar to the first selector, their contents will not be discussed.

The next CSS selector creates a text outline with a nice shadow effect by rendering the text in white with a thin black shadow, as shown here:

```
color: #fff;
text-shadow: 1px 1px 1px rgba(0,0,0,0.5);
```

The final portion of Listing 5.6 contains three selectors that affect the rendering of the `<h3>` element and its embedded `<a>` element: the `h3` selector specifies the width and font size; the `h3` selector specifies a linear gradient; and the `h3:after` selector specifies the text string to display. Other attributes are specified, but these are the main attributes for these selectors.

Figure 5.4 displays the result of matching the selectors in the CSS stylesheet `LinearGradient1.css` to the HTML page `LinearGradient1.html` in a browser.

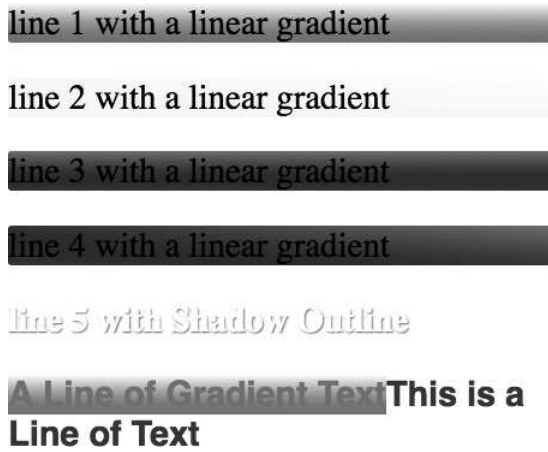

FIGURE 5.4: CSS3 linear gradient effect.

Radial Gradients

CSS3 radial gradients are more complex than CSS3 linear gradients, and you can use them to create more complex gradient effects.

A radial gradient requires at least two color stops, and its syntax is here:

```
background: radial-gradient(shape size at position, color1, ...,
last-color);
```

The default value for `shape` is `ellipse`; the default value for `size` is `farthest-corner`; and the default value for `position` is `center`.

A simple example of a radial gradient is here:

```
background: radial-gradient(red, yellow, green);
```

The preceding code snippet renders "evenly spaced" radial gradients, which is the default behavior. An example that specifies offset values for the color stops is here:

```
background: radial-gradient(red 25%, yellow 50%, green 60%);
```

The preceding code snippet renders the color `red` at the position that is `25%` from the beginning of the radial gradient, `yellow` at the halfway point, and `green` at the `60%` position.

The HTML5 Web page `RadialGradient1.html` and the CSS stylesheet `RadialGradient1.css` are not shown here, but the full listing is available on the companion disc. The essence of the code in the HTML5 code involves this code block:

```
<div id="outer">
  <div id="radial3">Text3</div>
  <div id="radial2">Text2</div>
  <div id="radial4">Text4</div>
  <div id="radial1">Text1</div>
 </div>
```

The CSS stylesheet `RadialGradient1.css` contains five CSS selectors that match the five HTML `<div>` elements, and one of the selectors is shown here:

```
#radial1 {
font-size: 24px;
width:  100px;
height: 100px;
position: absolute; top: 300px; left: 300px;

background: radial-gradient(farthest-side at 60% 55%, red, yellow,
#400);
}
```

The `#radial1` selector contains a `background` attribute that defines a radial gradient that specifies the farthest-side property, followed by the colors `red`, `yellow`, and `#400`.

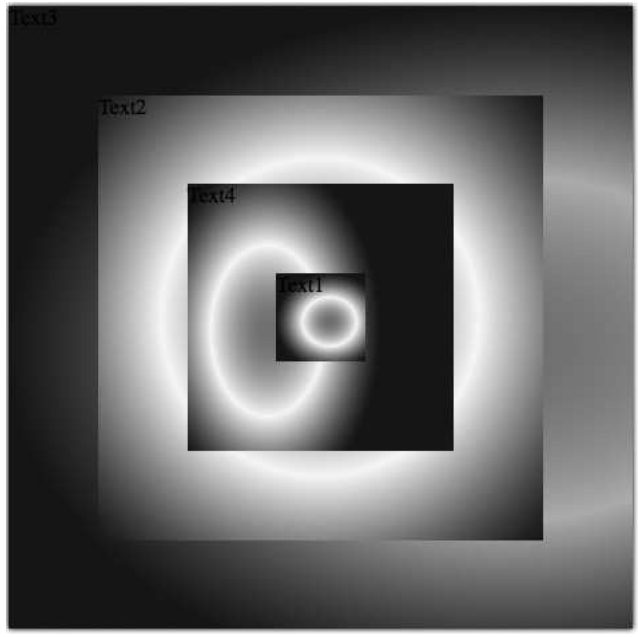

FIGURE 5.5: CSS3 radial gradient effect on an iPad3.

The other selectors are simple variations of the first selector that create a slightly contrasting collection of radial gradients. You can create these (and other) effects by specifying different start points and endpoints, and by specifying a start radius that is larger than the end radius.

Figure 5.5 displays the result of matching the selectors in the CSS stylesheet `RadialGradient1.css` to the HTML page `RadialGradient1.html`.

CSS3 2D TRANSFORMS

In addition to transitions, CSS3 supports four transforms that you can apply to HTML elements. The four CSS3 transforms are `scale`, `rotate`, `skew`, and `translate`. The following sections contain code samples that illustrate how to apply each of these CSS3 transforms to a set of images. The animation effects occur when users hover over any of the images; moreover, you can create "partial" animation effects by moving your mouse quickly between adjacent images.

Listing 5.7 displays the contents of `Scale1.html` and Listing 5.8 displays the contents of `Scale1.css`, which illustrate how to scale PNG files to create a "hover box" image gallery.

LISTING 5.7: Scale1.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8" />
  <title>CSS Scale Transform Example</title>
  <link href="Scale1.css" rel="stylesheet" type="text/css">
</head>

<body>
  <header>
   <h1>Hover Over any of the Images:</h1>
  </header>

  <div id="outer">
    <img src="sample2.png" class="scaled" width="150" height="150"/>
    <img src="sample3.png" class="scaled" width="150" height="150"/>
    <img src="sample2.png" class="scaled" width="150" height="150"/>
    <img src="sample3.png" class="scaled" width="150" height="150"/>
  </div>
</body>
</html>
```

Listing 5.7 references the CSS stylesheet `Scale1.css` (which contains selectors for creating scaled effects) and four HTML `<img>` elements that reference the PNG files `sample1.png` and `sample2.png`. The remainder of Listing 5.7 is straightforward, with simple boilerplate text and HTML elements.

LISTING 5.8: Scale1.css

```
#outer {
float: left;
position: relative; top: 50px; left: 50px;
}

img {
  transition: transform 1.0s ease;
}

img.scaled {
  box-shadow: 10px 10px 5px #800;
}

img.scaled:hover {
  transform : scale(2);
}
```

The `img` selector in Listing 5.8 contains specifies a `transition` property that applies a `transform` effect that occurs during a one-second interval using the `ease` function, as shown here:

```
transition: transform 1.0s ease;
```

Next, the selector `img.scaled` specifies a `box-shadow` property that creates a reddish shadow effect (which you saw earlier in this chapter), as shown here:

```
img.scaled {
  box-shadow: 10px 10px 5px #800;
}
```

Finally, the selector `img.scaled:hover` specifies a `transform` attribute that uses the `scale()` function in order to double the size of the associated PNG file whenever users hover over any of the `<img>` elements with their mouse, as shown here:

```
transform : scale(2);
```

Since the `img` selector specifies a one-second interval using an `ease` function, the scaling effect will last for one second. Experiment with different values for the CSS3 `scale()` function and also different values for the time interval to create the animation effects that suit your needs.

Another point to remember is that you can scale both horizontally and vertically:

```
img {
  transition: transform 1.0s ease;
}

img.mystyle:hover {
  transform : scaleX(1.5) scaleY(0.5);
}
```

Figure 5.6 displays the result of matching the selectors in the CSS stylesheet `Scale1.css` to the HTML page `Scale1.html`.

Rotate Transforms

The CSS3 `transform` attribute allows you to specify the `rotate()` function in order to create scaling effects, and its syntax looks like this:

```
rotate(someValue);
```

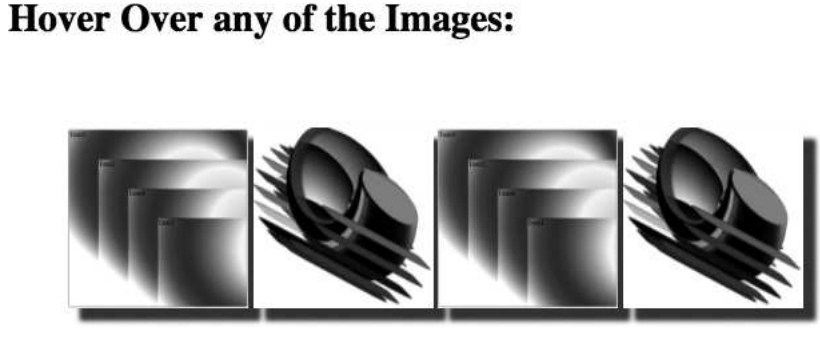

FIGURE 5.6: CSS3-based scaling effect on PNG files.

You can replace someValue with any number and the string deg (to specify "degrees"). When someValue is positive, the rotation is clockwise; when someValue is negative, the rotation is counterclockwise; and when someValue is zero, there is no rotation effect. In all cases the initial position for the rotation effect is the positive horizontal axis.

The HTML5 Web page Rotate1.html and the CSS stylesheet Rotate1.css on the companion disc illustrate how to create rotation effects, a sample of which is shown here:

```
img.imageL:hover {
  transform : scale(2) rotate(-45deg);
}
```

The img selector that specifies a transition attribute that creates an animation effect during a one-second interval using the ease timing function is shown here:

```
transition: transform 1.0s ease;
```

The CSS3 transform attribute allows you to specify the skew() function in order to create skewing effects, and its syntax looks like this:

```
skew(xAngle, yAngle);
```

You can replace xAngle and yAngle with any number. When xAngle and yAngle are positive, the skew effect is clockwise; when xAngle and yAngle are negative, the skew effect is counterclockwise; and when xAngle and yAngle are zero, there is no skew effect. In all cases the initial position for the skew effect is the positive horizontal axis.

The HTML5 Web page Skew1.html and the CSS stylesheet Skew1.css are on the companion disc, and they illustrate how to create skew effects. The CSS stylesheet contains the img selector that specifies a transition attribute that creates an animation effect during a one-second interval using the ease timing function, as shown here:

```
transition: transform 1.0s ease;
```

There are also the four selectors img.skewed1, img.skewed2, img.skewed3, and img.skewed4 to create background shadow effects with darker shades of red, yellow, green, and blue, respectively (all of which you have seen in earlier code samples).

The selector img.skewed1:hover specifies a transform attribute that performs a skew effect whenever users hover over the first <img> element with their mouse, as shown here:

```
transform : scale(2) skew(-10deg, -30deg);
```

The other three CSS3 selectors also use a combination of the CSS functions skew() and scale() to create distinct visual effects. Notice that the fourth

hover selector also sets the `opacity` property to `0.5`, which takes place in parallel with the other effects in this selector.

Figure 5.7 displays the result of matching the selectors in the CSS stylesheet `Skew1.css` to the elements in the HTML page `Skew1.html`. The landscape-mode screenshot is taken from an iPad3.

The CSS3 transform attribute allows you to specify the `translate()` function in order to create an effect that involves a horizontal and/or vertical "shift" of an element, and its syntax looks like this:

```
translate(xDirection, yDirection);
```

The translation is in relation to the origin, which is the upper-left corner of the screen. Thus, positive values for `xDirection` and `yDirection` produce a shift toward the right and a shift downward, respectively, whereas negative values for `xDirection` and `yDirection` produce a shift toward the left and a shift upward; zero values for `xDirection` and `yDirection` do not cause any translation effect.

The Web page `Translate1.html` and the CSS stylesheet `Translate1.css` on the companion disc illustrate how to apply a translation effect, as well as a scale effect, to a PNG file.

```
img.trans2:hover {
  transform : scale(0.5) translate(-50px, -50px);
}
```

The CSS stylesheet contains the `img` selector that specifies a transform effect during a one-second interval using the `ease` timing function, as shown here:

```
transition: transform 1.0s ease;
```

The four selectors `img.trans1`, `img.trans2`, `img.trans3`, and `img.trans4` create background shadow effects with darker shades of red, yellow, green, and blue, respectively, just as you saw in the previous section.

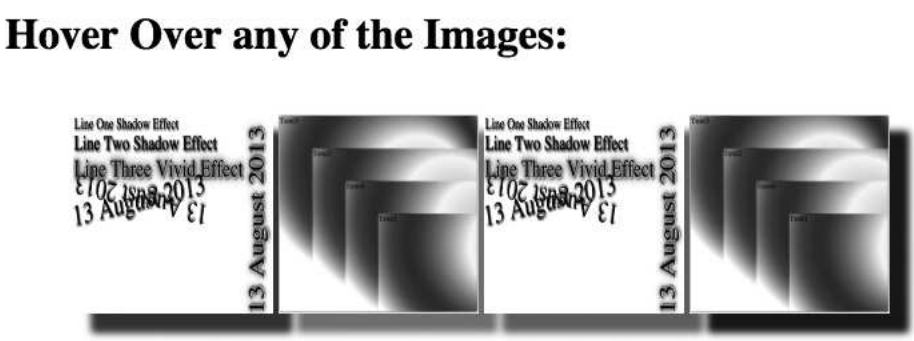

FIGURE 5.7: CSS3-based skew effects on PNG files.

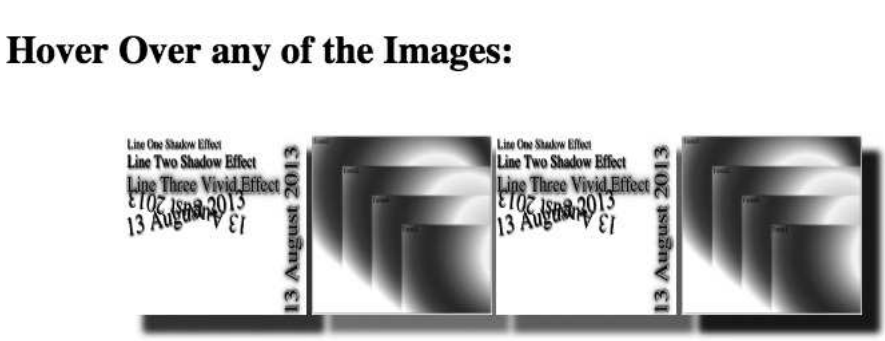

FIGURE 5.8: PNG files with CSS3 scale and translate effects.

The selector `img.trans1:hover` specifies a `transform` attribute that performs a scale effect and a translation effect whenever users hover over the first `<img>` element with their mouse, as shown here:

```
transform : scale(2) translate(100px, 50px);
```

Figure 5.8 displays the result of matching the selectors defined in the CSS3 stylesheet `Translate1.css` to the elements in the HTML page `Translate1.html`.

CSS3 3D ANIMATION EFFECTS

As you know by now, CSS3 provides `keyframes` that enable you to create different animation effects at various points during an animation sequence. The example in this section uses CSS3 `keyframes` and various combinations of the CSS3 functions `scale3d()`, `rotate3d()`, and `translate3d()` in order to create an animation effect that lasts for four minutes.

Listing 5.9 displays the contents of `Anim240Flicker3DLGrad4.html`, which is a very simple HTML page that contains four HTML `<div>` elements.

LISTING 5.9: Anim240Flicker3DLGrad4.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>CSS3 Animation Example</title>

  <link href="Anim240Flicker3DLGrad4.css"
        rel="stylesheet" type="text/css">
</head>

<body>
 <div id="outer">
  <div id="linear1">Text1</div>
  <div id="linear2">Text2</div>
```

```
  <div id="linear3">Text3</div>
  <div id="linear4">Text4</div>
 </div>
</body>
</html>
```

Listing 5.9 is a very simple HTML5 page with CSS selectors in the corresponding CSS stylesheet that match some of the elements in Listing 5.9. As usual, the real complexity occurs in the CSS selectors that contain the code for creating the animation effects.

The CSS selectors in `Anim240Flicker3DLGrad4.css` make extensive use of the CSS3 `matrix()` function. This function requires a knowledge of matrices (which is beyond the scope of this book) to fully understand the reasons for the effects that you can create with the CSS3 `matrix()` function. If you are interested in learning about matrices, you can read this introduction to matrices (in the context of CSS3): *http://www.eleqtriq.com/2010/05/css-3d-matrix-transformations/*.

Since `Anim240Flicker3DLGrad4.css` is such a lengthy code sample (over six pages of selectors), only a very limited portion of the code is displayed in Listing 5.10. However, the complete code is available on the companion disc for this book.

LISTING 5.10: A Small Section in Anim240Flicker3DLGrad4.css

```
@keyframes lowerLeft {
   0% {
      transform: matrix(1.5, 0.5,  0.0, 1.5, 0, 0)
                 matrix(1.0, 0.0,  1.0, 1.0, 0, 0);
   }
   10% {
      transform: translate3d(50px,50px,50px)
                 rotate3d(50,50,50,-90deg)
                 skew(-15deg,0) scale3d(1.25, 1.25, 1.25);
   }
   // lots of similar code omitted
   100% {
      transform: matrix(1.0, 0.0,  0.0, 1.0, 0, 0)
                 matrix(1.0, 0.5,  1.0, 1.5, 0, 0);
   }
}
// even more code omitted for brevity
#linear1 {
// minor details omitted
animation-name: lowerLeft;
animation-duration: 240s;
}
```

In case you're wondering, the selectors in Listing 5.10 contain two `matrix()` transforms instead of one transform, just to show you that it's possible to do so (and you can add even more `matrix()` transforms if you wish to do so).

Listing 5.10 contains a `keyframes` definition called `upperLeft` that starts with the following line:

```
@keyframes lowerLeft {
  // percentage-based definitions go here
}
```

The `#linear` selector contains properties that you have seen already, along with a property that references the `keyframes` identified by `lowerLeft`, and a property that specifies a duration of `240` seconds, as shown here:

```
#linear1 {
  // code omitted for brevity
  animation-name: lowerLeft;
  animation-duration: 240s;
}
```

Now that you know how to reference a `keyframes` definition in a CSS3 selector, let's look at the details of the definition of `lowerLeft`, which contains nineteen elements that specify various animation effects; only three elements are shown in Listing 5.10.

Each element of `lowerLeft` occurs during a specific stage during the animation. For example, the eighth element in `lowerLeft` specifies the value `50%`, which means that it will occur at the halfway point of the animation effect. Since the `#linear` selector contains an `animation-duration` property whose value is `240s` (shown in bold in Listing 5.10), this means that the animation will last for four minutes, starting from the point in time when the HTML5 page is launched.

The eighth element of `lowerLeft` specifies a translation, rotation, skew, and scale effect (all of which are in three dimensions), an example of which is shown here:

```
50% {
   transform: translate3d(250px,250px,250px)
              rotate3d(250px,250px,250px,-120deg)
              skew(-65deg,0) scale3d(0.5, 0.5, 0.5);
}
```

The animation effect occurs in a sequential fashion, starting with the translation and finishing with the scale effect, which is also the case for the other elements in `lowerLeft`.

Figure 5.9 displays the initial view of matching the CSS3 selectors defined in the CSS3 stylesheet `Anim240Flicker3DLGrad4.css` with the HTML elements in the HTML page `Anim240Flicker3DLGrad4.html` in a landscape-mode screenshot taken from an iOS application running on an iPad3.

FIGURE 5.9: CSS3 3D animation effects on an iPad3.

CSS3 AND SVG

You can reference entire SVG documents using the CSS3 `url()` function. On the other hand, you can style SVG elements with CSS in several ways:

- Use inline CSS property declarations
- Use internal CSS stylesheets
- Use external CSS stylesheets

The following sections contain short examples of the preceding functionality.

The CSS3 `url()` Function

CSS3 selectors can reference SVG documents using the CSS3 `url()` function, which means that you can incorporate SVG-based graphics effects (including animation) in your HTML pages. As a simple example, Listing 5.11 shows you how to reference the SVG document `Blue3DCircle1.svg` in a CSS3 selector, and Listing 5.12 displays the contents of `Blue3DCircle1.svg`.

LISTING 5.11: Blue3DCircle1.css

```
#circle1 {
opacity: 0.5; color: red;
width: 250px; height: 250px;
position: absolute; top: 0px; left: 0px;
font-size: 24px;
border-radius: 4px;
background: url(Blue3DCircle1.svg) top right;
}
```

Listing 5.11 contains various property/value pairs, and the portion containing the CSS `url()` function is shown here:

```
background: url(Blue3DCircle1.svg) top right;
```

This name/value pair specifies the SVG document `Blue3DCircle1.svg` as the background for an HTML `<div>` element in an HTML5 page whose `id` attribute is `circle1`.

LISTING 5.12: Blue3DCircle1.svg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">

  <defs>
     <pattern id="checkerPattern1"
              width="10" height="10"
              patternUnits="userSpaceOnUse">
        <rect fill="blue"
              x="0"  y="0"  width="5" height="5"/>
        <rect fill="yellow"
              x="5" y="0"   width="5" height="5"/>
        <rect fill="red"
              x="0"  y="5"  width="5" height="5"/>
        <rect fill="blue"
              x="5" y="5"   width="5" height="5"/>
     </pattern>

     <circle id="3DBlueCircle1" cx="0" cy="0" r="80"
             opacity=".5"
             style="fill:url(#checkerPattern1)"/>
  </defs>

  <g transform="translate(20,20)">
     <use xlink:href="#3DBlueCircle1" x="100" y="100"
               opacity=".8"/>
  </g>
</svg>
```

Listing 5.12 contains SVG code that you have already seen in earlier chapters, so the details are omitted here.

Inline Property Declarations and SVG

The following code snippet contains an SVG `<rect>` element with a `style` attribute that contains the attributes fill, stroke, and stroke-width (as well as their values):

```
<svg width="500" height="300">
  <rect id="rect1" x="200" y="100" width="100" height="80"
```

```
        style="fill:red; stroke:blue; stroke-width:3"/>
</svg>
```

If you prefer not to embed styling-related attributes in an SVG element, you can move the contents of the preceding `style` attribute to a CSS selector (defined inside a `<style>` element) that matches the `<rect>` element whose `id` attribute is `rect1`, as shown in the next section.

Inline CSS Selectors and SVG

You might be surprised to discover that you can specify CSS-based styling for SVG elements. Listing 5.13 displays the contents of `SVGAndInlineCSS.svg`, which shows you how to define an inline CSS selector in an SVG document.

LISTING 5.13: SVGAndInlineCSS.svg

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
 <defs>
    <style type="text/css"><![CDATA[
       circle {
          stroke: #f00;
          stroke-width: 4;
          fill: #00f;
       }
       circle:hover {
          stroke: #00f;
          stroke-width: 8;
          fill: #f00;
       }
    ]]></style>
 </defs>

 <g>
   <circle cx="100" cy="100" r="80" />
 </g>
</svg>
```

Listing 5.13 contains a `<defs>` element that contains a CSS-based `<style>` element, which in turn contains two CSS selectors. The first selector (shown in bold) matches the `<circle>` element (also shown in bold) that is defined at the bottom of Listing 5.13. The second selector specifies some CSS property/value pairs that are used for styling the `<circle>` element whenever users hover over this element with their mouse.

Keep in mind that `CDATA` sections are useful because they ensure that the enclosed characters (including meta-characters) are not interpreted by XML parsers. Although this section is not required in Listing 5.13, in general it's a good idea to always include the `CDATA` section.

Incidentally, the CSS selector that corresponds to the style attribute in the <rect> element in the previous section looks like this:

```
#rect1 {
  fill: red;
  stroke: #00f;
  stroke-width: 3;
}
```

The next section shows you yet another option: how to work with external CSS stylesheets with SVG.

SVG and External CSS Stylesheets

There are two parts to using external CSS stylesheets in SVG: first create a CSS stylesheet and then reference the CSS stylesheet in the SVG document. For example, create the CSS stylesheet mystyle.css with the following contents:

```
#rect1 {
  fill: red;
  stroke: #00f;
  stroke-width: 3;
}
```

Next, create the SVG document SVGAndExternalCSS.svg, whose contents are displayed in Listing 5.14.

LISTING 5.14: SVGAndExternalCSS.svg

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet href="mystyle.css" type="text/css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
  <rect id="#rect1" x="200" y="100" width="100" height="80"
</svg>
```

When you launch the code in Listing 5.13 in a modern browser, you will see the effect of matching the CSS selector in mystyle.css with the SVG <rect> element in Listing 5.14.

SIMILARITIES AND DIFFERENCES BETWEEN SVG AND CSS3

This section briefly summarizes the features that are common to SVG and CSS3 and also the features that are unique to each technology. Chapter 6 contains code samples that illustrate many of the features that are summarized in this section.

SVG and CSS3 both provide support for the following:

- Linear and radial gradients
- 2D graphics and animation effects

- The ability to create shapes such as rectangles, circles, and ellipses
- WAI-ARIA (Web Accessibility Initiative – Accessible Rich Internet Applications)

SVG supports the following features that are not available in CSS3:

- Bezier curves
- Hierarchical object definitions
- Custom glyphs
- Rendering text along an arbitrary path
- Defining event listeners on SVG objects
- Programmatic creation of 2D shapes using JavaScript
- "Accessibility" to XML-based technologies and tools

CSS3 supports the following features that are not available in SVG:

- 3D graphics and animation effects
- Multicolumn rendering of text

Note that SVG Filters and CSS Filters will become one and the same at some point in the not-too-distant future.

In general, SVG is better suited than CSS3 for large data sets that will be used for data visualization, and you can reference the SVG document (which might render some type of chart) in a CSS3 selector using the CSS3 `url()` function. You have already seen such an example in Chapter 3, where the SVG document contains the layout for a bar chart. In general, there might be additional processing involved where data is retrieved or aggregated from one or more sources (such as databases and web services), and then manipulated using some programming language (such as XSLT, Java, or JavaScript) in order to programmatically create an SVG document or perhaps create SVG elements programmatically in a browser session.

SUMMARY

This chapter showed you how to create graphics effects, shadow effects, and how to use CSS3 transforms in CSS3. You learned how to create animation effects that you can apply to HTML elements, and you saw how to define CSS3 selectors to do the following:

- Render rounded rectangles
- Create shadow effects for text and 2D shapes
- Create linear and radial gradients
- Use the methods `translate()`, `rotate()`, `skew()`, and `scale()`
- Create CSS3-based animation effects

The next chapter shows you how to combine D3 with CSS3, SVG, and HTML5 `Canvas` in HTML Web pages.

CHAPTER 6

INTRODUCTION TO D3

This chapter introduces you to D3 and provides a collection of short code samples that illustrate how to use some useful D3 APIs. This chapter moves quickly, so even if you are already familiar with D3 it's worth your while to read the material in this chapter.

The first part of this chapter provides a brief description of the D3 toolkit. The second part of this chapter shows you how to use some basic D3 methods by rendering simple 2D shapes. In addition, you will learn how to create linear gradients and radial gradients in this section.

NOTE *Launch the HTML Web pages in a browser whenever you read the code samples in this book, because this can help you understand what the code actually does instead of struggling with the purpose of code blocks in the samples.*

WHAT IS D3?

Mike Bostock created the open source toolkit `Protovis` (sort of a "predecessor" of D3), and then he created the `D3` toolkit, which is a JavaScript-based open source project for creating very appealing data visualization. D3 is an acronym for "Data Driven Documents," and its home page is here: *http://mbostock.github.com/d3/*.

Although D3 can be used for practically any type of data visualization, common use cases include: rendering maps, geographic-related data, economic data (such as employment figures) in conjunction with various locales, and medical data (diabetes seems to be very popular).

In December of 2011, D3 was named the data visualization project of the year (by `Flowing Data!`), which is not surprising when you see the functionality that is available in D3.

D3 provides a layer of abstraction that generates underlying SVG code. D3 enables you to create a surprisingly rich variety of data visualizations. If you need to generate graphics-oriented Web pages, and you prefer to work with JavaScript instead of working with "raw" SVG, then you definitely ought to consider using D3.

Two key aspects of D3 involve tools for reading data in multiple formats and also the ability to transform the data and render the data in many forms.

D3 supports the following features:

- Creation of SVG-based 2D shapes
- 2D graphics and animation effects
- Method chaining

D3 has an extensive collection of "helper methods," such as `select()`, `append()`, `data()`, and `attr()`, along with many other helper methods. Read the online documentation about these and other D3 methods.

D3 ON MOBILE DEVICES

D3 works on any device that supports JavaScript and SVG, including mobile devices such as smart phones and tablets. These devices do vary in terms of their support for SVG features. For instance, Android 3.x has some support for SVG, and currently no version of Android supports SVG filters. In general, iOS devices support more SVG features than Android-based mobile devices, but functionality does change so it's worth testing feature support on different devices.

If you are writing HTML Web pages for desktops as well as mobile devices, you probably need to take into account issues such as handling mouse-related events versus touch-related events. In particular, you ought to test multi-touch support on multiple mobile devices and OSes to ensure that your Web pages exhibit the expected behavior.

In addition, you might encounter D3 bugs on mobile devices that are not readily apparent in HTML Web pages with D3 on laptops or desktops. If you do encounter inconsistent behavior, check the issues-related link in the previous section to see if it's a known issue. If you do not find entries, it's possible that you have discovered an unreported bug in D3, in which case you can file a new issue.

D3 BOILERPLATE

If you have worked with HTML5, you are probably familiar with various "Boilerplate" toolkits that are available. In a similar spirit, there is a D3 Boilerplate toolkit (d3.js-boilerplate) here: *https://github.com/zmaril/d3.js-boilerplate*.

According to the D3 Boilerplate website:

> d3.js boilerplate is an opinionated template system designed to help you build a sophisticated data driven document as fast as possible. By providing a full featured template and encouraging the use of useful tools, this project aims to help developers passively and actively cut down on development time.

If you are a D3 novice, you might not be ready to use this toolkit, but it's a good idea to be aware of its functionality.

This concludes the brief introduction to D3. The next section of this chapter introduces you to the concept of "method chaining" in order to facilitate the discussion of subsequent code samples in this chapter (and the rest of this book).

METHOD CHAINING IN D3

Practically every code sample in this book (and almost all the online code samples in various forums) use method chaining, so it's worth your time to understand method chaining before delving into the code samples. If you have used jQuery extensively, you are probably familiar with method chaining.

The key idea to remember is that a block of D3 code with a select method performs a search in the DOM of the current HTML Web page. The result of that search is actually a "result set," which is the set of elements that match the selection criteria (again, similar to the way that jQuery works). You can then apply an "action" to that set of elements. For example, you can find all the paragraphs in an HTML Web page and then set their text to red. Here is an example that uses the `d3.selectAll()` method to select all the HTML `<p>` elements in an HTML Web page and then invokes the `style()` method to set the color of the text in those paragraphs to red:

```
d3.selectAll("p").style("color", "red");
```

Returning to our previous discussion: after applying an action to a set of elements, a new set of elements is returned. In fact, you can apply a second action to that modified set, which returns yet another set. This process of applying multiple methods to a set is called method chaining, and the good news is that you "chain" together as many function invocations as you wish. Method chaining enables you to write very compact yet powerful code, as you will see in the code examples in this chapter.

THE D3 METHODS `SELECT()` AND `SELECTALL()`

D3 supports various selection-based methods that return arrays of elements in order to maintain the hierarchical structure of sub-selections. D3 also binds additional methods to the array of selected elements, thereby enabling you to perform operations on those elements.

D3 provides the method `selectAll()` that you saw in the previous section, and also the method `select()`. Both methods accept selector strings, and both are used for selecting elements.

The select() method selects only the first matching element, whereas the selectAll() method selects all matching elements (in document traversal order).

Due to space constraints this chapter covers a subset of the D3 selection-based methods, but you can find a complete list of those methods here: *https://github.com/mbostock/d3/wiki/Selections*.

CREATING NEW HTML ELEMENTS

The code sample in this section shows you how to add new HTML elements to an existing HTML Web page using the `d3.append()` method.

Listing 6.1 displays the contents of `AppendElement1.html`, which illustrates how to add an HTML <p> element to an HTML Web page.

LISTING 6.1: AppendElement1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Append Elements</title>
   <script src="d3.js"></script>
 </head>

<body>
   <script>
     d3.select("body").append("p").text("Hello1 D3");
     d3.select("body").append("p").text("Hello2 D3");
     d3.select("body").append("p").text("Hello3 D3");
     d3.select("body").append("p").text("Hello4 D3");

     d3.selectAll("p").style("color", "red");
   </script>
 </body>
</html>
```

Listing 6.1 starts by referencing the D3 JavaScript file `d3.js`. Next, the <script> element appends four HTML <p> elements to the <body> element using the `d3.select()` method.

Finally, Listing 6.1 changes the color of all four HTML <p> elements to red with this code snippet:

```
d3.selectAll("p").style("color", "red");
```

Incidentally, if you want to alternate the colors in the four HTML <p> elements, insert the following code in Listing 6.1:

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#f00" : "#eee";
});
```

FIGURE 6.1: Appending <p> Elements with D3.

The preceding code snippet uses a ternary operator to return the color `#f00` for even-numbered HTML <p> elements and `#eee` for odd-numbered HTML <p> elements.

In case you're wondering, you can cache the results of lookup variables (as you can in jQuery), which results in better performance than the redundant lookups that are performed in Listing 6.1.

Figure 6.1 displays the result of launching the HTML Web page in Listing 6.1.

THE MOST COMMON IDIOM IN D3

The most common idiom in D3 uses the following type of construct (which involves method chaining, of course):

```
var theData = [1,2,3,4,5];

var paras = d3.select("body")
              .selectAll("p")
              .data(theData)
              .enter()
              .append("p")
              .text("D3 ");
```

Here is how to read the code in the preceding code block, starting from the definition of the `paras` variable:

Step 1: start by selecting the <body> element of the current HTML Web page (using the `select()` method)
Step 2: return the result set of all the child <p> elements by means of the `selectAll()` method (and if there are no child <p> elements, the returned set is a set of length zero)
Step 3: iterate or "loop" through the numbers in the `theData` JavaScript array to create a new HTML <p> element whose text value is the string `D3`
Step 4: after each iteration in Step 3, append the newly created <p> element to the result set in Step 2.

NOTE *The `d3.selectAll()` method always returns a result set, which can be an empty set (and therefore this method never returns a null or undefined value).*

The `enter()` method is subtle (non-intuitive?) but important because it determines how many new elements to create and append to the DOM. After the `selectAll("p")` method is invoked, a binding is created with any existing <p> elements. If there are no such elements, then five new <p> elements are created because there are five elements in the `theData` array. On the other hand, if there are (for example) two <p> elements in the DOM, then only three new <p> elements are created. Thus, the number of existing elements (if any) affects the number of new elements that are created.

When the preceding code snippet has completed, the JavaScript variable `paras` will consist of five new HTML <p> elements. If you understand this sequence of events, you are ready for the code sample in the next section. If you do not understand, then continue to the next section and launch the HTML Web page in a browser to convince yourself that the preceding explanation is correct.

BINDING DATA TO DOM ELEMENTS

Now that you understand method chaining in D3 and you understand how to use the most common idiom in D3, you are ready to see how to perform both in an HTML Web page.

Listing 6.2 displays the contents of `Binding1.html`, which illustrates how to combine JavaScript variables with the D3 methods `.data()` and `.text()` in order to append a set of HTML <p> elements to an HTML Web page.

LISTING 6.2: Binding1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Appending Sets of Elements</title>
   <script src="d3.js"></script>
 </head>

 <body>
   <script>
     var theData = [1,2,3,4,5];

     var paras = d3.select("body")
                   .selectAll("p")
                   .data(theData)
                   .enter()
                   .append("p")
                   .text("D3 ");

     d3.select("body").append(paras);
   </script>
 </body>
</html>
```

Listing 6.2 starts by referencing the D3 JavaScript file, and the code in the <script> element has already been explained to you in the preceding section (how convenient!).

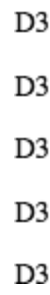

FIGURE 6.2: Iterating through an array to append <p> elements.

The only new code in Listing 6.2 is the following code snippet:

```
d3.select("body").append(paras);
```

The preceding code snippet appends the contents of the `paras` variable (which consists of five new HTML <p> elements) to the existing HTML <p> elements (if there are any) that are child elements of the <body> element.

NOTE *Since every HTML Web page in this book starts with the same "boilerplate" code in the HTML <head> element, we will omit this duplicated description for the rest of the book.*

In addition, the code that appears in the variable `paras` is a "standard" element creation code block, and you will see this type of code used elsewhere in this chapter.

Figure 6.2 displays the graphics image that is rendered by the code in the HTML Web page in Listing 6.2.

GENERATING TEXT STRINGS

Listing 6.3 displays the contents of `GenerateText1.html`, which illustrates how to iterate through an array and render text strings.

LISTING 6.3: GenerateText1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title>Iterating Through Arrays</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
  <script>
    var dataValues1 = [50, 100, 250, 150, 300];

    d3.select("body")
        .selectAll("p")
```

```
        .data(dataValues1)
        .enter()
        .append("p")
        .text(function(d) { return "Paragraph Number: "+d; })
        .style("font-size", "16px")
        .style("color", "blue");
   </script>
 </body>
</html>
```

Listing 6.3 contains a `<script>` element that initializes a JavaScript variable `dataValues1`, followed by a standard element creation code block to create and append a set of new HTML `<p>` elements to the existing HTML Web page. The only new construct is the use of a function, as shown in the following code snippet:

```
.text(function(d) { return "Paragraph Number: "+d; })
```

When you define a function in a standard code block, D3 "understands" that it must populate the variable `d` with the value of the current iteration through the numbers in the JavaScript array `dataValues1`.

By the way, you can use any legitimate name that you want in the preceding function, but perhaps it helps to think of the variable `d` as "datum," or a single piece of data (such as a number in an array).

The next section shows you how to leverage the information that you have learned so far in order to create gradient effects in an HTML Web page.

Figure 6.3 displays the graphics image that is rendered by the code in the HTML Web page in Listing 6.3.

ADDING HTML <DIV> ELEMENTS WITH GRADIENT EFFECTS

If you have worked with CSS3 graphics, the CSS3 selectors in Listing 6.4 will be familiar. If you are new to CSS3, this code sample gives you a preview of how easily you can combine CSS3 with D3 in an HTML Web page. CSS3 graphics and animation is the subject of Chapter 7 so you can revisit this code sample after you have finished reading Chapter 7.

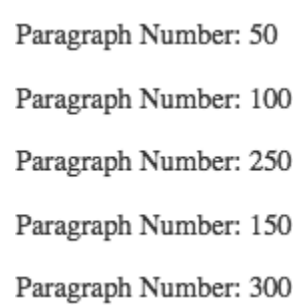

FIGURE 6.3: Generating text with D3.

Listing 6.4 displays the contents of `GenerateGradientDivs1.html`, which illustrates how to iterate through an array and render text strings that are displayed with CSS3 gradients.

LISTING 6.4: GenerateGradientDivs1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title>Creating Gradients </title>
   <script src="d3.min.js"></script>

   <style>
     .gradient {
       display: inline-block;
       top:    20px;
       width:  100px;
       height: 100px;

       background-image: linear-gradient(linear,
                         to bottom left,
                         #f00, #00f);

       border-radius: 8px;
     }
   </style>
 </head>

<body>
  <script>
    var dataValues1 = [0, 150, 300, 450, 600];

    d3.select("body")
      .selectAll("p")
      .data(dataValues1)
        .enter()
        .append("div")
        .attr("left", function(d) {
            return d+"px";
        })
        .attr("class", "gradient");
   </script>
 </body>
</html>
```

Listing 6.4 contains the usual boilerplate code and the `<script>` element uses a standard D3 idiom to create and append a set of HTML `<div>` elements to the existing HTML Web page. There are two new things to notice about this code. First, there is a function definition that uses the D3 `.attr()` method, as shown here:

```
.attr("left", function(d) {
   return d+"px";
})
```

FIGURE 6.4: Generating a row of gradient-colored elements.

The preceding code block uses each and every value in the JavaScript array `dataValues1` (remember that D3 is iterating through an array "behind the scenes") to set the pixel value of the CSS `left` property of each new HTML `<div>` element.

Second, the following code snippet sets the CSS class property to `.gradient` (which is defined as a selector in the `<style>` element near the top of Listing 6.4):

```
.attr("class", "gradient");
```

Launch the code in Listing 6.4 and view the source code to verify the previous statements and to familiarize yourself with this technique for using D3 to create gradient effects in HTML Web pages.

Figure 6.4 displays the graphics image that is rendered by the code in the HTML Web page in Listing 6.4.

CREATING SIMPLE 2D SHAPES

This section contains a code sample that shows you how to create simple 2D shapes in D3. The D3 code specifies attributes that are the same as the SVG-based attributes for each 2D shape. For example, an ellipse is defined in terms of its center point `(cx, cy)`, its major axis `rx`, and its minor axis `ry`. Similar comments apply for creating a rectangle (`x, y, width`, and `height` attributes) and for creating a line segment (`(x1,y1)` and `(x2,y2)` as the coordinates of the two endpoints of the line segment). In fact, Listing 6.5 is nothing more than using a standard D3 idiom and the D3 `.attr()` method to set the attributes of various 2D shapes. Note that in Chapter 7 you will learn about the SVG attributes for numerous 2D shapes.

Listing 6.5 displays the contents of `SimpleShapes1.html`, which illustrates how to create a circle, ellipse, rectangle, and a line segment in D3.

LISTING 6.5: SimpleShapes1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Create Simple 2D Shapes</title>
   <script src="d3.min.js"></script>
  </head>
```

```
  <body>
   <script>
     var width = 600, height = 400;

     // circle and ellipse attributes
     var cx = 50,   cy = 80,  radius1 = 40,
         ex = 250,  ey = 80,  radius2 = 80;

     // rectangle attributes
     var rectX = 20, rectY = 200;
     var rWidth = 100, rHeight = 50;

     // line segment attributes
     var x1=150,y1=150,x2=300,y2=250,lineWidth=4;

     var colors = ["red", "blue", "green"];

     // create an SVG element
     var svg = d3.select("body")
                 .append("svg")
                 .attr("width",  width)
                 .attr("height", height);

     // append a circle
     svg.append("circle")
        .attr("cx", cx)
        .attr("cy", cy)
        .attr("r",  radius1)
        .attr("fill", colors[0]);

     // append an ellipse
     svg.append("ellipse")
        .attr("cx", ex)
        .attr("cy", ey)
        .attr("rx", radius2)
        .attr("ry", radius1)
        .attr("fill", colors[1]);

     // append a rectangle
     svg.append("rect")
        .attr("x", rectX)
        .attr("y", rectY)
        .attr("width",  rWidth)
        .attr("height", rHeight)
        .attr("fill", colors[2]);

     // append a line segment
     svg.append("line")
        .attr("x1", x1)
        .attr("y1", y1)
        .attr("x2", x2)
        .attr("y2", y2)
        .attr("stroke-width", lineWidth)
        .attr("stroke", colors[0]);
   </script>
  </body>
 </html>
```

Listing 6.5 contains a `<script>` element that creates multiple SVG elements and uses the D3 `.attr()` method to set the value of the attributes of each SVG element. When you launch the HTML Web page in Listing 6.5, D3 appends the following code block to the existing HTML Web page:

```
<svg width="600" height="400">
  <circle cx="50" cy="80" r="40" fill="red"></circle>
  <ellipse cx="250" cy="80" rx="80" ry="40" fill="blue"></ellipse>
  <rect x="20" y="200" width="100" height="50" fill="green"></rect>
  <line x1="150" y1="150" x2="300" y2="250"
        stroke-width="4" stroke="red"></line>
</svg>
```

Compare the code in Listing 6.5 with the preceding code block to verify that the preceding SVG elements correspond to the code in Listing 6.5.

You can use an alternate coding style that defines multiple JavaScript variables, as shown in the following code block:

```
var theBody = d3.select("body");

var theSVG = theBody.append("svg")
      .attr("width",  100)
      .attr("height", 100);

var circleSelection = theSVG.append("circle")
      .attr("cx", 50)
      .attr("cy", 50)
      .attr("r", 30)
      .style("fill", "red");
```

Figure 6.5 displays the graphics image that is rendered by the code in the HTML Web page in Listing 6.5.

BEZIER CURVES AND TEXT

Listing 6.6 displays the contents of `BezierCurvesAndText1.html`, which illustrates how to use D3 in order to render a quadratic Bezier curve and a cubic Bezier curve and text strings that follow the path of the Bezier curves.

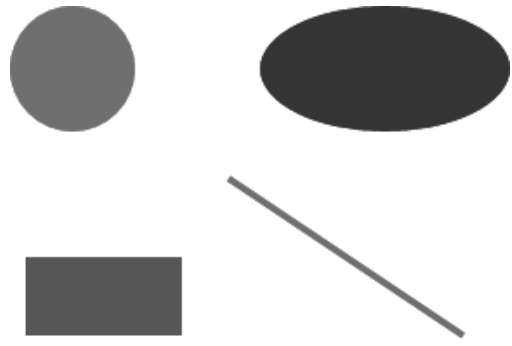

FIGURE 6.5: A line segment, circle, ellipse, and rectangle in D3.

LISTING 6.6: BezierCurvesAndText1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Bezier Curves and Text</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
     var width = 600, height = 400, opacity=0.5;
     var cubicPath  = "M20,20 C300,200 100,500 400,100";
     var quadPath   = "M200,20 Q100,300 500,100";

     var cValues    = "M20,20 C300,200 100,500 400,100";
     var qValues    = "M200,20 Q100,300 500,100";

     var fontSizeC  = "24";
     var fontSizeQ  = "18";

     var textC =
      "Sample Text that follows a path of a cubic Bezier curve";

     var textQ =
      "Sample Text that follows a path of a quadratic Bezier curve";

     var fillColors = ["red", "blue", "green", "yellow"];

     // create an SVG container...
     var svgContainer = d3.select("body").append("svg")
                          .attr("width",  width)
                          .attr("height", height);

     var defs     = svgContainer
                        .append("svg:defs");

     var patternC = defs.append("svg:path")
                        .attr("id", "pathInfoC")
                        .attr("d",  cValues);

     var patternQ = defs.append("svg:path")
                        .attr("id", "pathInfoQ")
                        .attr("d",  qValues);

     // now add the 'g' element...
     var g1 = svgContainer.append("svg:g");

     // create a cubic Bezier curve...
     var bezierC = g1.append("path")
                     .attr("d", cubicPath)
                     .attr("fill",   fillColors[0])
                     .attr("stroke", "blue")
                     .attr("stroke-width", 2);

     // text following a cubic Bezier curve...
```

```
    var textC   = g1.append("text")
                    .attr("id", "textStyleC")
                    .attr("stroke", "blue")
                    .attr("fill",   fillColors[1])
                    .attr("stroke-width", 2)
                      .append("textPath")
                      .attr("font-size", fontSizeC)
                      .attr("xlink:href", "#pathInfoC")
                      .text(textC);

    // create a quadratic Bezier curve...
    var bezierQ = g1.append("path")
                    .attr("d", quadPath)
                    .attr("fill",   fillColors[1])
                    .attr("opacity", opacity)
                    .attr("stroke", "blue")
                    .attr("stroke-width", 2);

    // text following a cubic Bezier curve...
    var textQ   = g1.append("text")
                    .attr("id", "textStyleQ")
                    .attr("stroke", fillColors[3])
                    .attr("stroke-width", 2)
                      .append("textPath")
                      .attr("font-size", fontSizeQ)
                      .attr("xlink:href", "#pathInfoQ")
                      .text(textQ);
  </script>
 </body>
</html>
```

Listing 6.6 contains the usual boilerplate code and a `<script>` element that defines a quadratic Bezier curve and a cubic Bezier curve. The JavaScript variables `bezierC`, `bezierQ`, `textC`, and `textQ` are set to the values for a cubic Bezier curve, a quadratic Bezier curve, the text for the cubic Bezier curve, and the text for the quadratic Bezier curve, respectively. These variables are used in the D3 code that creates a `<path>` element, which is how you specify quadratic and cubic Bezier curves in SVG.

As you can see, most of the code in Listing 6.6 does two things: create SVG elements with the D3 `.append()` method, and then set the required attributes with the D3 `.attr()` method.

Moreover, the definitions for the cubic Bezier curve and the quadratic Bezier curve consist of a string of values, as shown here:

```
var cubicPath  = "M20,20 C300,200 100,500 400,100";
var quadPath   = "M200,20 Q100,300 500,100";
```

In fact, if you want to use standard SVG code instead of D3, you could literally copy and paste the preceding strings as the value for the `d` attribute in the SVG `<path>` element.

Figure 6.6 displays the graphics image that is rendered by the code in the HTML Web page in Listing 6.6.

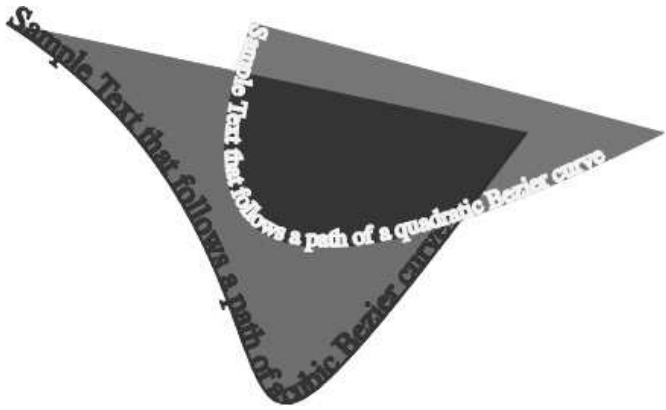

FIGURE 6.6: Text along cubic and quadratic Bezier curves.

2D TRANSFORMS

D3 provides support for the following 2D transforms: rotate, scale, skew, and translate. You can apply transforms to 2D shapes using the D3 `.attr()` method, as shown in the following examples:

```
.attr("transform", "translate("+transX+","+transY+")");
.attr("transform", "rotate("+rotateX+")");
.attr("transform", "scale("+scaleX+","+scaleY+")");
.attr("transform", "skewX("+skewX+")");
```

The companion disc contains the HTML Web page `Transforms1.html` that fully illustrates how to create four 2D transform effects.

A DIGRESSION: SCALING ARRAYS OF NUMBERS TO DIFFERENT RANGES

This section covers the `d3.range()` method for determining the range of a set of numbers, followed by the `d3.scale()` function for scaling a set of numbers. The rationale for including this section here is that the `d3.range()` method is used in the next two gradient-related code samples. This method is both easy to use and straightforward to understand, and you will see this method in many code samples in this book.

The simplest use of the `d3.range()` method is to generate a list of integers. For example, `d3.range(15)` generates the integers between `0` and `14`. You can verify this fact by including the following code snippet in a D3-based HTML Web page:

```
console.log("range from 0 to 14: "+d3.range(15))
```

When you open the Web Inspector, you will see the following:

```
Numbers from 0 to 14: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14
```

Another use for the `d3.range()` method enables you to scale the values in a JavaScript array by "mapping" them to a different range of values. This functionality is very useful whenever you need to scale the elements in a bar chart or graph so that the graphics output fits the dimensions of the screen where you are rendering the chart or graph.

Suppose that you have the following set of numbers in a JavaScript array:

```
var dataValues = [10, 20, 30, 40, 50];
```

You can scale them to the range `[0,10]` with the following code snippet:

```
x = d3.scale.linear().domain([10,50]).range([0,10]);
```

Although the preceding code snippet is correct, there are two limitations. First, the minimum and maximum values of the JavaScript array `dataValues` are hard-coded with the values `10` and `50`. Second, the range of values is also hard-coded in the code.

The following code snippet is a better way to scale a set of numbers:

```
var dataValues = [10, 20, 30, 40, 50], left=0, right=10;
var xScale = d3.scale.linear()
               .domain([d3.min(dataValues), d3.max(dataValues)])
               .range([left, right]);
```

Notice that all the hard-coded values have been replaced by JavaScript variables. The preceding code block uses a linear scale (with `d3.scale.linear()`) to "scale down" the set of numbers in the `dataValues` array so that they fit into the interval `[0,10]`. This is very convenient because this scaling effect can be achieved with arrays that contain very small numbers and very large numbers (such as 1,000,000) in order to ensure that the entire rendered bar chart or graph will be visible.

Thus, the advantage of the preceding code block is that it works correctly for *any* JavaScript array of numbers. However, you do need to make a manual change to the range values if you want to use a different range. Although we have not discussed the `d3.min()` and `d3.max()` methods, they return the minimum and maximum values, respectively, in a JavaScript array of numbers.

NOTE *You will probably indent the range of values so that there is "padding" on the left and on the right of your charts and graphs in order to avoid inadvertently clipping portions of data from the visual display.*

For example, if you want to render a scatter plot in the horizontal range of `[0,600]` and you also want to indent by `20` on the left and on the right of the chart, you can use something like the following code snippet:

```
var pad=20;
xScale = d3.scale.linear()
           .domain([d3.min(dataValues), d3.max(dataValues)])
           .range([left+pad, right-pad]);
```

The preceding code block is for scaling numbers along the horizontal axis, and the same type of code works for scaling numbers along the vertical axis, as shown here:

```
yScale = d3.scale.linear()
           .domain([d3.min(dataValues), d3.max(dataValues)])
           .range([pad, height-pad]);
```

NOTE *The vertical axis is positive in the top-to-bottom direction, and if you want to reverse the "polarity" of the vertical axis, simply reverse the two numbers in the* `d3.range()` *method in the preceding definition for* `yScale`.

TWEENING IN D3

D3 provides support for three tweening methods: `styleTween()`, `attrTween()`, and `tween()`. An example of using `styleTween()` is here:

```
d3.select("body").transition()
  .styleTween("color", function() {
       return d3.interpolate("green", "red");
  });
```

A fun tip: you can use colors in addition to numbers in the `d3.range()` function. For example, the following code snippet is valid in D3:

```
var colorScale = d3.scale.linear()
                   .domain([0,100])
                   .range(['red', 'blue']);
...
.attr('fill', function(d) {
    return colorScale[d];
})
...
```

The preceding code block sets the color of a shape (not shown here) to a value that is a linear interpolation between `red` and `blue`. Moreover, you can specify hexadecimal values, `(R,G,B)` values, and `HSL` values in addition to common color names, which is a very nice feature of D3.

You will see many code samples in this book that use this type of code, so you will have plenty of opportunity to become more comfortable with this coding technique in D3.

FORMATTING NUMBERS

D3 supports various formats for numbers. As a simple example, you can insert a comma (",") in a number. For example, if you want to render `1234000` as `1,234,000` you can use the following code snippet:

```
var x1 = 1234000;
var x2 = d3.format(",")(x1)
d3.format(".3s");
```

If you want to display `1234000` as `1.234M`, use the following code snippet:

```
var x1 = 1234000;
var x3 = d3.format(".4s")(x1);
```

More information about D3 formatting is here: *https://github.com/mbostock/d3/wiki/Formatting#wiki-d3_format*.

LINEAR GRADIENTS IN D3

D3 enables you to define linear gradients in a straightforward manner, which you will see in this section (the next section discusses radial gradients in D3).

The HTML Web page `LinearGradient1.html` on the companion disc illustrates how to create linear gradients in D3. The definition of a linear gradient is shown in the following code block:

```
var gradient = svg.append("svg:defs")
      .append("svg:linearGradient")
        .attr("id", "gradient")
        .attr("x1", "0%")
        .attr("y1", "0%")
        .attr("x2", "100%")
        .attr("y2", "100%")
        .attr("spreadMethod", "pad");

gradient.append("svg:stop")
        .attr("offset", "0%")
        .attr("stop-color", "#0c0")
        .attr("stop-opacity", 1);

gradient.append("svg:stop")
        .attr("offset", "100%")
        .attr("stop-color", "#c00")
        .attr("stop-opacity", 1);
```

The preceding code block starts with a code snippet by creating an SVG `<defs>` element that is appended to an SVG `<svg>` element (not shown here). The next two code snippets define a so-called "stop color" that specifies the attributes of the color to render in the linear gradient.

Figure 6.7 displays the graphics image that is rendered by the code in the HTML Web page `LinearGradient1.html`.

FIGURE 6.7: A linear gradient in D3.

RADIAL GRADIENTS IN D3

In addition to linear gradients, D3 enables you to define radial gradients. The HTML Web page `RadialGradient1.html` illustrates how to create radial gradients in D3. The definition of a radial gradient is very similar to a linear gradient, and the difference is shown in the following code block:

```
var gradient = svg.append("svg:defs")
  .append("svg:radialGradient")
    .attr("id", "gradient")
    .attr("x1", "0%")
    .attr("y1", "0%")
    .attr("x2", "100%")
    .attr("y2", "100%");

gradient.append("svg:stop")
    .attr("offset", "0%")
    .attr("stop-color", "#f00")
    .attr("stop-opacity", 1.0);
// add other stop colors as needed
```

Read the entire source code that is available on the companion disc to see the omitted details.

WORKING WITH IMAGE FILES

You can use D3 in order to specify binary images in an HTML Web page. For example, the HTML Web page `Images1.html` on the companion disc illustrates how to render three PNG files. You need to specify values for the attributes `x`, `y`, `width`, and `height` and also the name of the PNG file. The key idea is shown in the following code block:

```
var imageWidth=100, imageHeight=100;
var dataValues = [[0,0], [100, 100], [200, 200]];
var images = ["sample1.png", "sample2.png", "sample3.png"];

// code omitted for brevity
.append("svg:image")
.attr("xlink:href", function(d,i) {
    return images[i];
})
.attr("width",  imageWidth)
.attr("height", imageHeight)
.attr("x",  function(d) { return d[0]; })
.attr("y",  function(d) { return d[1]; })
```

Read the entire source code that is available on the companion disc to see the omitted details.

D3 AND FILTERS

You can use SVG filters in D3 code by converting the SVG code into its corresponding D3 code. For example, the following code block is the D3 code that

corresponds to the SVG filters that are defined in `TurbFilterCBDRect2.svg` in Chapter 2:

```
var filter = svg.append("svg:defs")
                .append("svg:filter")
                  .attr("id", "turbFilter1")
                  .attr("in", "SourceImage")
                  .attr("filterUnits", "objectBoundingBox");

var turb1 = filter.append("svg:feTurbulence")
                    .attr("baseFrequency", 0.05)
                    .attr("numOctaves", 1)
                    .attr("result", "turbulenceOut1");

var disp1 = filter.append("svg:feDisplacementMap")
                  .attr("in", "SourceGraphic")
                  .attr("in2", "turbulenceOut1")
                  .attr("xChannelSelector", "B")
                  .attr("yChannelSelector", "R")
                  .attr("scale", 2);

var diff1 = filter.append("svg:feDiffuseLighting")
                  .attr("in", "SourceGraphic")
                  .attr("diffuseConstant", 2)
                  .attr("surfaceScale", 20)
                  .attr("style", "lighting-color:#FFFFCC")
                .append("svg:feSpotLight")
                  .attr("x", 30)
                  .attr("y", 10)
                  .attr("z", 80)
                  .attr("pointsAtX", 30)
                  .attr("pointsAtY", 30)
                  .attr("pointsAtZ", 0)
                  .attr("specularExponent", 4);
```

Use the preceding sample as a guideline for converting other SVG filters into the corresponding D3 code.

OTHER D3 FEATURES AND D3 APIs

D3 is a very powerful toolkit with an extensive set of features that are not covered in this chapter. However, if you perform an online search regarding D3, here are some features that might be useful to you:

- Voronoi diagrams
- Force diagrams
- Choropleths (maps)
- Statistical functions
- Many chart types

Peruse the D3 gallery that provides a great assortment of D3-based data visualization: *https://github.com/mbostock/d3/wiki/Gallery*.

D3 API Reference

If you are impatient and you want to dive into the D3 APIs, they are listed here: *https://github.com/mbostock/d3/wiki/API-Reference*.

The D3 APIs are listed in various categories, which include: Ajax, arrays, axes, chord, cluster, colors, force, geography, hierarchy, histogram, ordinals, pie, projections, scales, shapes (SVG), selections, stack, string formatting, time, transitions, tree, and treemap.

In addition, known issues for D3 are listed here: *https://github.com/mbostock/d3/issues?page=1&state=open*.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The companion disc contains additional HTML Web pages that use D3. The HTML Web page `Polygon1.html` illustrates how to use D3 to render a polygon. The HTML Web page `EllipticArcs1.html` renders an elliptic arc, and the HTML Web page `BezierCurves1.html` renders a quadratic Bezier curve and a cubic Bezier curve. The HTML Web page `NestedRectangles1.html` renders a set of nested rectangles with D3.

The HTML Web page `MultiQBezierScaledGlasses1.html` shows you how to render a set of "eyeglasses" with D3. The HTML Web page `NoULeftTurn1.html` shows you how to render a "traffic sign" with D3. The HTML Web page `PartialBlueSphereCB5.html` shows you how to render a modified sphere with D3. Note that the preceding three code samples use an SVG `<pattern>` element and also an SVG `<use>` element, neither of which are discussed in this chapter.

The HTML Web page `CardioidEllipses1Grad2.html` renders a set of gradient ellipses, each of which follow the path of a Cardioid polar equation, and the HTML Web page `Transforms1.html` illustrates how to create four 2D transform effects. The HTML Web pages `Cube1.html` and `Pyramid1.html` demonstrate how to render a cube and a pyramid, respectively, in D3.

The HTML Web page `Cone1.html` shows you how to render a cone in D3. The HTML Web pages `GridCubes1.html` shows you how to render a grid-like set of cubes. The HTML Web pages `Cone2.html`, `Pyramid2.html`, `Pyramid2.html`, and `CardioidEllipses1Grad2Use1.html` have one thing in common: they all define an SVG `svg:use` element in order to create multiple scaled copies of a "base" graphics image that is defined in a lower-numbered code sample (e.g., `Cone2.html` contains multiple copies of a "base" cone that is rendered in `Cone1.html`, and so forth).

The following article shows you how to use the Web Audio API with D3: *http://blog.scottlogic.com/2016/01/06/audio-api-with-d3.html*.

Finally, an open source project with more than 1,000 D3-based code samples (many of which are similar variants) using various polar equations is here: *https://github.com/ocampesato/d3-graphics*.

SUMMARY

This chapter started with an introduction to D3, followed by examples of using D3 methods to render simple 2D shapes, such as circles, line segments, and rectangles. The companion disc also contains code samples for rendering polygons, ellipses, elliptic arcs, and Bezier curves. In addition, you learned about a standard D3 idiom, which is an extremely useful code construct that is ubiquitous in HTML Web pages that use D3. You also learned about the following D3 methods: `select()`, `selectAll()`, `append()`, `enter()`, `domain()`, and `range()`.

Finally, you learned how to render linear gradients and radial gradients, along with examples of rendering abstract graphics on the companion disc.

The next chapter explores arrays, mouse events, and animation effects with D3.

CHAPTER 7

MOUSE EVENTS AND ANIMATION EFFECTS

This chapter covers a diverse set of topics, including JavaScript arrays, mouse events, and simple animation effects. For instance, D3 uses JavaScript arrays to represent data, so it's obviously important to know how to deal with arrays. In addition, you can use mouse events to create an interactive user experience, and also to display more detailed information whenever users hover over different parts of an HTML Web page. Finally, animation effects can create even more vivid array effects in your data visualization, and also create a time-based effect for datasets. If you are a D3 novice, it's probably better for you to read (or at least skim through) the previous chapter before reading this chapter.

The first part of this chapter shows you how to work with multidimensional arrays in D3, which is useful for representing complex data sets. The second part of this chapter shows you how to use D3 with mouse-related events, which are handy for creating interactive data visualization websites.

The final portion of this chapter contains code samples for creating animation effects. D3 enables you to create fine-grained animation effects more easily than "pure SVG," thereby enabling you to enhance your Web pages with subtle and pleasing visual effects.

One more point: the only way to capture the full visual effect of the Web pages that create animation effects or mouse-related resizing effects is by launching those Web pages in a browser. A series of screenshots that approximate the visual effects cannot provide a comparable substitute, so screenshots for those Web pages are not included in this chapter.

FINDING THE MAXIMUM AND MINIMUM VALUES IN AN ARRAY

D3 provides the `d3.domain()` and the `d3.range()` functions in order to "map" a set of source values to another set of values. D3 also provides the

functions `d3.min()` and `d3.max()`, which find the minimum and the maximum value of a set of numbers. Although you can programmatically determine the maximum and minimum numbers in a JavaScript array, the D3 methods conveniently perform these calculations for you.

For example, suppose that you have a set of data points whose minimum and maximum values can change dynamically. In addition, suppose that you also want the ability to "map" the data points to different value ranges. The following code block provides the required type of flexibility and functionality:

```
var dataValues = [10, 20, 30, 40, 50], lower=100, upper=400;

var xScale = d3.scale.ordinal()
               .domain([d3.min(dataValues), d3.max(dataValues)])
               .range([lower, upper]);
```

As you can see, the use of the `d3.min()` and `d3.max()` functions in the preceding code block ensures that you can replace the `dataValues` array with any realistic set of data values. In addition, you can change the values of `lower` and `upper` independently of the data values, which means that you can scale the vertical axis to accommodate your needs (and device types).

WORKING WITH MULTIDIMENSIONAL ARRAYS

D3 supports several data formats (including CSV files that contain data in the form of comma-separated values), and you can also write D3 code that references data that is defined in multidimensional JavaScript arrays.

Listing 7.1 displays the contents of `IterateArrays1.html`, which illustrates how to iterate through some of the values of three arrays.

LISTING 7.1: IterateArrays1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title>Iterating Through Arrays</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
  <script>
    var width    = 600, height  = 400,
        sWidth   = 20,  sHeight = 20,
        offsetX  = 0,   offsetY = 0;

    var svg = d3.select("body")
                .append("svg:svg")
                  .attr("width",  width)
                  .attr("height", height);

    var dataValues1 = [50, 100, 250, 150, 300];
```

```
    var dataValues2 = [
      [50, 20], [100, 90], [250, 50], [150, 33], [300, 95]
    ];

    var dataValues3 = [
      [50, 20, 90], [100, 90, 50], [250, 50, 20]
    ];

    d3.max(dataValues1, function(d) {
console.log("current dataValues1 d: "+d);

      var mySquare = svg.append("rect")
        .attr("x", d)
        .attr("y", offsetY)
        .attr("width", sWidth)
        .attr("height",sHeight)
        .style("fill","red");

      return d;
    });

    d3.max(dataValues2, function(d) {
console.log("current dataValues2 d: "+d[0]);
      return d[0];
    });

    d3.max(dataValues3, function(d) {
console.log("current dataValues3 d: "+d[1]);
      return d[1];
    });
 </script>
</body>
</html>
```

Listing 7.1 starts with the usual boilerplate code, followed by a `<script>` element that defines three JavaScript arrays `dataValues1`, `dataValues2`, and `dataValues3` containing one-dimensional, two-dimensional, and three-dimensional data values, respectively. Listing 7.1 uses the values in `dataValues1` to generate a set of rectangles, which you have seen in earlier code samples. The next code snippet displays the maximum value of the first component in the `dataValues2` array, as shown here:

```
d3.max(dataValues2, function(d) {
   console.log("current dataValues2 d: "+d[0]);
   return d[0];
});
```

Similarly, the final code snippet displays the maximum value of the second component in the `dataValues3` array. While there is nothing exceptional about what this does, it illustrates how to access different elements of multidimensional JavaScript arrays and how to apply D3 methods to those arrays.

In addition, Listing 7.1 shows you how to define a JavaScript function that uses D3 to generate a set of random numbers, as shown here:

```
function randomData(rangeCount, maxSize) {
    return d3.range(rangeCount).map(function(i) {
        // return a 2-dimensional array...
        return {x: i, y: maxSize*Math.random()};
    });
}

// 20 random numbers (maximum value of each is 100)
var data = randomData(20, 100);

// find the maximum value for 2d array
var maxValue2 = d3.max(data, function(d) {return d.y;});
```

Figure 7.1 is a screenshot of a Chrome browser session that displays the generated output from Listing 7.1.

2D ARRAYS AND SCATTER CHARTS

Now that you understand how to iterate through the values of 2D arrays, it's very easy to generate scatter charts. The idea involves rendering circles whose center point is based on the pairs of values that belong to a 2D array. Although this code sample could easily be part of Chapter 2, it's a good opportunity to "leverage" the concepts that you learned in the previous section.

Listing 7.2 displays the contents of `IterateArrays2.html`, which illustrates how to render a scatter chart.

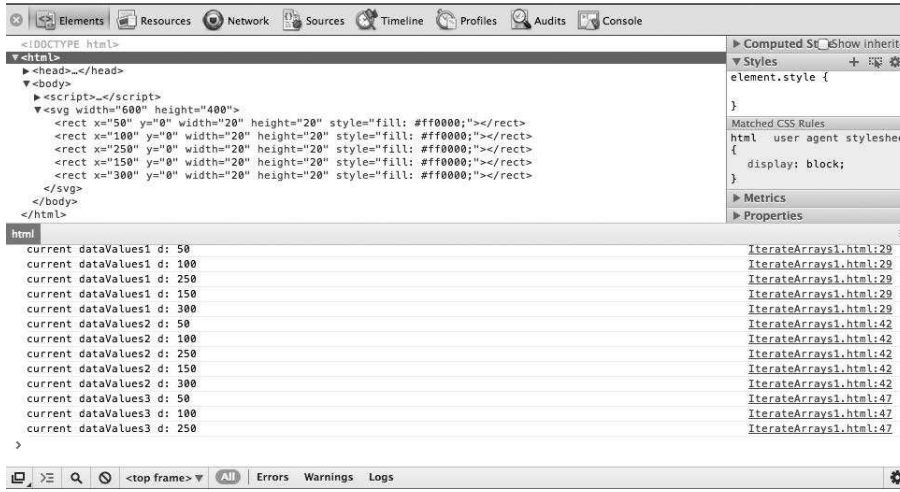

FIGURE 7.1: Output values from the loops in Listing 7.1.

LISTING 7.2: IterateArrays2.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title>Iterating Through Arrays</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
  <script>
    var width = 600, height = 400, radius = 8;

    var svg = d3.select("body")
                .append("svg:svg")
                  .attr("width",  width)
                  .attr("height", height);

    var dataValues2 = [
      [50, 20], [100, 200], [250, 150], [150, 30],
      [30, 80], [200, 140], [150, 190], [350, 50]
    ];

    // render the circles...
    d3.max(dataValues2, function(d) {
      svg.append("circle")
        .attr("cx", d[0])
        .attr("cy", d[1])
        .attr("r",  radius)
        .style("fill","red");
    });

    // display the coordinates...
    svg.selectAll("text")
       .data(dataValues2)
       .enter()
       .append("text")
       .text(function(d) {
         return d[0] + "," + d[1];
       })
       .attr("x", function(d) {
            return d[0];
       })
       .attr("y", function(d) {
            return d[1];
       })
       .attr("font-family", "sans-serif")
       .attr("font-size", "11px")
       .attr("fill", "blue");
  </script>
 </body>
</html>
```

Listing 7.2 defines the JavaScript array `dataValues2` that contains two-dimensional elements. Next, Listing 7.2 uses the values in `dataValues2` to

generate a set of circles (which you have seen in a previous code sample), and sets values for the attributes `cx` and `cy` by specifying `d[0]` and `d[1]` (which you learned in the previous section).

The next portion of code uses a standard D3 idiom to render the coordinates of each point in the scatter chart. The key point to observe is the following code snippet:

```
.text(function(d) {
    return d[0] + "," + d[1];
})
```

The preceding code snippet returns a pair of values that represent the coordinates of each point in the scatter chart. For example, the first pair is `(50,100)`. The location of the text is the same location as the location of its corresponding circle, which is set with the following code block:

```
.attr("x", function(d) {
   return d[0];
})
.attr("y", function(d) {
   return d[1];
})
```

Figure 7.2 is a screenshot of a Chrome browser session that displays the generated output from Listing 7.2.

D3 DATA SCALING FUNCTIONS

As you saw earlier in this chapter, D3 provides the methods `d3.domain()` and `d3.range()` to scale data values with a linear function (shown below), along with the methods `d3.min()` and `d3.max()` that calculate the minimum value and the maximum value in an array.

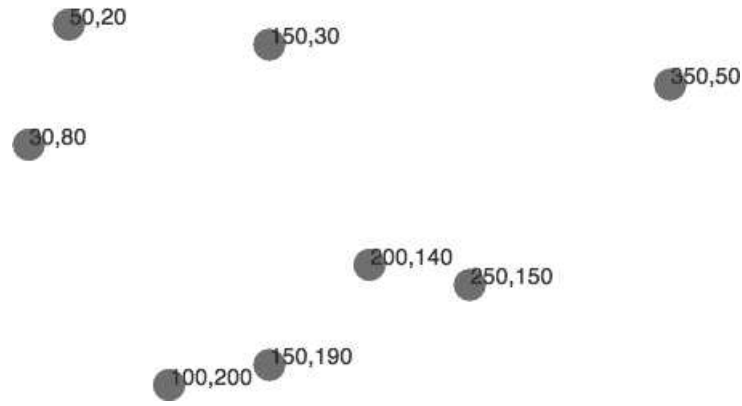

FIGURE 7.2: Rendering a scatter plot of circles and their coordinates.

Listing 7.3 displays the contents of `Scaling1.html`, which illustrates how to scale or "map" a set of domain and range values.

LISTING 7.3: Scaling1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Scaling Effects with D3</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
     var w = 640, h = 400,
     deltaU = 0.2, deltaV = 0.2,
     x = d3.scale.linear().domain([-1,1]).range([0,w]),
     y = d3.scale.linear().domain([0, 1]).range([0,h]);

     for(var u=-1; u<2; u += deltaU) {
console.log("u: "+u+" x(u): "+x(u));
     }

     for(var v=0; v<2; v += deltaV) {
console.log("v: "+v+" y(v): "+y(v));
     }
   </script>
 </body>
<html>
```

Listing 7.3 starts with the usual boilerplate code, followed by a `<script>` element that maps the points in the interval `[-1,1]` to the points in the interval `[0,w]`. This means that the endpoints of `[-1,1]` are mapped to the endpoints of `[0,w]`, which is to say that the left endpoint `-1` of `[-1,1]` is mapped to the left endpoint `0` and the right endpoint `1` of `[-1,1]` is mapped to `w`. Since the mapping is linear, the midpoint of `[-1,1]` is mapped to the midpoint of `[0,w]`: the former midpoint is `0` and the latter midpoint is `w/2`.

The next code snippet maps the points in the interval `[0,1]` to the points in the interval `[0,h]`. Consequently, the endpoints of `[0,1]` are mapped to the endpoints of `[0,h]`. In other words, the left endpoint `0` of `[0,1]` is mapped to the left endpoint `0` of the interval `[0,h]` and the right endpoint 1 of `[0,1]` is mapped to the right endpoint h of `[0,h]`. Since this mapping is also linear, the midpoint of `[0,1]` is mapped to the midpoint of `[0,h]`: the former midpoint is `0.5` and the latter midpoint is `h/2`.

Next there are two simple loops that iterate through both newly mapped sets of points.

Figure 7.3 is a screenshot of a Chrome browser session that displays the generated output from Listing 7.3.

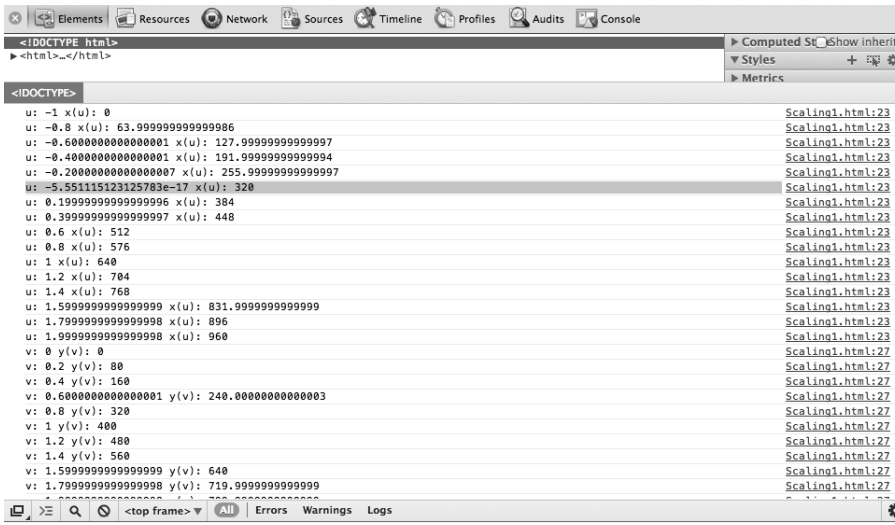

FIGURE 7.3: Output values from the loops in Listing 7.3.

If you want to specify horizontal and vertical indentation, you can use four JavaScript variables, or you can use the following type of code block, where the JavaScript variable `m` maintains the margins in one convenient location:

```
var m = {top: 20, right: 20, bottom: 20, left: 60};

var xRange = d3.scale.linear().range([m.left, width - m.right]),
    yRange = d3.scale.linear().range([height - m.top, m.bottom]);
```

As a supplemental set of related code exercises, see if you can predict the behavior of the following code block:

```
var x1 = d3.scale.linear();
var x2 = d3.scale.linear().domain([0,10000]);
var x3 = d3.scale.linear().domain([0,10000]).range([0,100]);

var myNumbers = [100, 500, 300, 900, 2000];

// map [0,2000] to [1,100]
var x4 = d3.scale.linear()
           .domain([0,d3.max(myNumbers)])
           .range([0,100]);

// map [0,100] to [1,100]
var x5 = d3.scale.linear()
           .domain([0,d3.min(myNumbers)])
           .range([0,100]);
```

Just to be sure that you are correct, copy the preceding code block into an HTML Web page and add some `console.log()` statements. Launch your Web page and examine the Web Inspector to view the displayed information.

OTHER D3 SCALING FUNCTIONS

In addition to the `Linear Scale` that we have already discussed, D3.js also supports `Quantitative Scales` and `Ordinal Scales`. In brief, the `Quantitative Scales` have a continuous domain such as dates, times, real numbers, and so forth. On the other hand, the `Ordinal Scales` are for discrete domains such as names, categories, colors, and so forth. Although this book does not use these scales in this book, you can perform an Internet search to find examples of other D3 scales if you need to incorporate them into your HTML5 Web pages.

One other point to keep in mind: D3.js `Linear Scales` enable us to resize our data to fit into our predefined SVG coordinate space instead of resizing our SVG coordinate space to fit our data.

D3 PATH DATA GENERATOR

This section shows you how to use the D3 `Path Data Generator` to create visual effects that are more complex than the code samples that you saw earlier in this chapter.

Listing 7.4 displays the contents of `PathLineSegments1.html`, which illustrates how to render a polyline that consists of a set of line segments.

LISTING 7.4: PathLineSegments1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Path-based Line Segments </title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
     var width = 600, height = 400, radius = 5, factor=3;

     var lineData = [{"x": 20,   "y": 50},   {"x": 200, "y": 200},
                     {"x": 240,  "y": 10},   {"x": 400, "y": 300},
                     {"x": 380,  "y": 150},  {"x": 500, "y": 250}];

     // define a Path Data Generator for lines...
     var lineFunction  = d3.svg.line()
                           .x(function(d) { return d.x; })
                           .y(function(d) { return d.y; })
                           .interpolate("linear");

     // create an SVG container...
     var svgContainer = d3.select("body").append("svg")
                          .attr("width",  width)
                          .attr("height", height);

     // add the path element (with line segments)...
     var lineGraph     = svgContainer.append("path")
```

```
                          .attr("d", lineFunction(lineData))
                          .attr("stroke", "blue")
                          .attr("stroke-width", 2)
                          .attr("fill", "none");

     // add circles at each vertex...
     d3.max(lineData, function(d) {
        var circles = svgContainer
                          .append("circle")
                          .data(lineData)
                          .attr("cx", d.x)
                          .attr("cy", d.y)
                          .attr("r",  radius)
                          .attr("fill", "red")
                          .on("mouseover", function() {
                            var r = d3.select(this).attr("r");
                            d3.select(this).attr("r", factor*r);
                          })
                          .on("mouseout", function() {
                             var r = d3.select(this).attr("r");
                             d3.select(this).attr("r", r/factor);
                          })
     });
   </script>
 </body>
</html>
```

Listing 7.4 starts with the usual boilerplate code, followed by a `<script>` element that defines a JavaScript array `lineData` with JSON-based (JavaScript Object Notation) data that specifies the x-coordinate and the y-coordinate for a set of points in a so-called scatter chart.

The key idea is to define a "line function" that uses linear interpolation to display the points that are defined in the array `lineData`, as shown here:

```
var lineFunction  = d3.svg.line()
                      .x(function(d) { return d.x; })
                      .y(function(d) { return d.y; })
                      .interpolate("linear");
```

Notice that previous code samples in the chapter used `d[0]` and `d[1]` to reference the first and second coordinates of "unnamed" data values in two-dimensional JavaScript arrays. On the other hand, the preceding code snippet uses "keyed" data values `d.x` and `d.y` because that is how the first and second coordinates are identified in the array `lineData`.

Next, Listing 7.4 appends an SVG `<svg>` element to the HTML Web page, and then renders a set of line segments based on the values in the variable `lineData`. The key point to observe is the portion in bold that is shown in the following code snippet:

```
var lineGraph = svgContainer.append("path")
                          .attr("d", lineFunction(lineData))
                          .attr("stroke", "blue")
                          .attr("stroke-width", 2)
                          .attr("fill", "none");
```

The preceding code snippet does a lot of work behind the scenes: it uses the definitions in the `lineFunction` variable to create an SVG `<path>` element that consists of a set of line segments whose vertices are from the data points in the variable `lineData` (re-read this sentence to absorb all of the information).

The final portion of code in Listing 7.4 creates a set of SVG `<circle>` elements based on the value in the array `lineData` using code that you have seen in previous code samples. In addition, each circle has a `mouseover` event handler that *multiplies* the radius of the currently "hovered" circle by the value of `factor`. Since `factor` is `3`, a `mouseover` event triples the size of the current circle. Each circle also has a `mouseout` event handler that *divides* the radius of the currently "hovered" circle by the value of `factor`, which sets the radius of the current circle to its original value. One other point: there is nothing "special" about the value of `factor`, nor is it necessary to add a `mouseover` or a `mouseout` event handler: they are included in this code sample to illustrate this effect, which you can create in your own code if you believe that doing so will enhance the user experience.

Incidentally, D3.js provides eleven different types of line interpolations for the `d3.svg.line()` function, and you can perform an Internet search for code samples that use those interpolations if you are interested in using them in your HTML Web pages.

WHAT ABOUT `THIS`, `$THIS`, AND `$(THIS)`?

The `this` keyword is quite common in JavaScript and HTML Web pages, and unfortunately, it can be a source of confusion. The `this` keyword in JavaScript is determined as follows:

- If a method of an object `callingObject` caused the current block of code to be executed, then `this` is the object `callingObject`, unless
- The function reference is passed to an event handler, in which case `this` is the DOM element that declares the event handler
- In an anonymous function, `this` is the window object

One of the really nice things about jQuery is that it simplifies the rather tricky rules regarding the `this` keyword.

NOTE *If you work with jQuery, you must use `$(this)` in order to invoke jQuery functions, and jQuery adds jQuery-specific functions to the `$(this)` object.*

The next part of this chapter shows you how to add mouse-related event handlers to create visual effects that respond to user gestures.

D3 AND MOUSE EVENTS

This section shows you how to handle several mouse-related events in an HTML5 Web page with D3 code.

Listing 7.5 displays the contents of `ResizeCircle1.html`, which illustrates how to resize an SVG `<circle>` element during mouse events.

LISTING 7.5: ResizeCircle1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Handling Mouse Events with D3</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
     d3.select("body")
       .append("svg")
       .attr("width",  300)
       .attr("height", 300)
       .append("circle")
       .attr("cx", 150)
       .attr("cy", 150)
       .attr("r",  30)
       .attr("fill", "blue")
       .on("mouseover", function() {
          return d3.select(this)
                   .attr("r", 80)
                   .attr("fill", "red");
       })
       .on("mouseout", function() {
         return d3.select(this)
                  .attr("r", 50)
                  .attr("fill", "green");
       }).on("click", function(d, i) {
         return console.log(d, i);
       });
   </script>
 </body>
</html>
```

Listing 7.5 starts with the usual boilerplate code, followed by a `<script>` element that appends an SVG `<svg>` element and an SVG `<circle>` element to the `<body>` element of the current HTML Web page. Let's look at the D3 syntax to handle a "mouseover" event, as shown in the following code snippet:

```
.on("mouseover", function() {
       return d3.select(this)
                .attr("r", 80)
                .attr("fill", "red");
})
```

The key point to observe is that the `return` statement references the current element (which is an SVG `<circle>` element) with the `this` keyword, and then changes the `r` attribute to `80` and the color to `red`. In a similar fashion

during the `mouseout` event, the value of `r` is set to `50` and the color changes to green. Finally, when users click on the circle, a message is displayed.

MOUSE EVENTS AND RANDOMLY LOCATED 2D SHAPES

This section shows you how to create and render circles at random locations on the screen whenever users move their mouse on the screen.

Listing 7.6 displays the contents of `RandomCircles1.html`, which illustrates how to create and render SVG `<circle>` elements at random locations during mouse events.

LISTING 7.6: RandomCircles1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Rendering Random Circles with D3</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
     var width = 800, height = 500;
     var radius = 30, moveCount = 0, index = 0;
     var circleColors = ["red", "yellow", "green", "blue"];

     var svg = d3.select("body")
       .append("svg")
       .attr("width",  width)
       .attr("height", height);

     svg.on("mousemove", function() {
       index = (++moveCount) % circleColors.length;

       var circle = svg.append("circle")
          .attr("cx", (width-100)*Math.random())
          .attr("cy", (height-100)*Math.random())
          .attr("r",  radius);

       circle.attr("fill", circleColors[index]);
     });
   </script>
 </body>
</html>
```

Listing 7.6 bears some resemblance to Listing 7.5, with code to handle a `mousemove` event. The key idea is illustrated in the following code snippet:

```
var circle = svg.append("circle")
                .attr("cx", (width-100)*Math.random())
                .attr("cy", (height-100)*Math.random())
                .attr("r",  radius);
```

The preceding code snippet uses the `Math.random()` function to generate random values that are assigned to the attributes `cx` and `cy` (the coordinates of the center point of a circle), as well as assigning the value of the variable `radius` to the attribute `r` (which is the radius of the circle). This code snippet is executed whenever users move their mouse (which can render many circles on the screen).

You can use slightly different syntax to reference a JavaScript function for handling an event. For example, compare the following code snippet with the `mousemove` code in Listing 7.6:

```
svg.on("mousemove", addCircle);

function addCircle() {
  // insert the circle-related code here
}
```

A "FOLLOW THE MOUSE" EXAMPLE

This section shows you how to create and render circles at the current location on the screen whenever users move their mouse.

Listing 7.7 displays the contents of `FollowTheMouse2.html`, which illustrates how to create and render SVG `<circle>` elements at the current mouse location.

LISTING 7.7: FollowTheMouse2.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Follow the Mouse with D3</title>
   <script src="d3.min.js"></script>
 </head>

<body>
   <script>
     var width = 800, height = 500;
     var stripWidth = 20, stripCount = 0;
     var radius = 30, moveCount = 0, index = 0;
     var factor = 0, factors = [1.0, 0.5];

     var circleColors = ["red", "yellow", "green", "blue"];

     var svg = d3.select("body")
       .append("svg")
       .attr("width",  width)
       .attr("height", height);

     svg.on("mousemove", function() {
       index = (++moveCount) % circleColors.length;

       stripCount = Math.floor(moveCount/stripWidth);
       factor = factors[stripCount%2];
```

```
      var circle = svg.append("circle")
         .attr("cx", d3.event.pageX)
         .attr("cy", d3.event.pageY)
         .attr("r",  radius*factor);

      circle.attr("fill", circleColors[index]);
     });
   </script>
 </body>

</html>
```

Listing 7.7 contains a `<script>` element with code that is familiar to you from previous code samples in this chapter. The key idea is to determine the current location of the user's mouse during a `mousemove` event and then append a new SVG `<circle>` (at the location of the mouse) to the `<body>` element of the current HTML Web page. This is accomplished by the following code snippet:

```
var circle = svg.append("circle")
              .attr("cx", d3.event.pageX)
              .attr("cy", d3.event.pageY)
              .attr("r",  radius*factor);
```

As you can see in the previous code snippet, the D3 `.attr()` method uses the values `d3.event.pageX` and `d3.event.pageY` to set the values of the attributes `cx` and `cy` of a newly created SVG `<circle>` element, thereby creating a "follow the circle" effect.

While you can use `pageX` and `pageY` that are available in the native event, transforming the event position to the local coordinate system of the container that received the event gives you some benefits. For instance, if you embed an SVG in the normal flow of an HTML Web page, you may want the event position relative to the top-left corner of the SVG image. In addition, if the SVG code contains a `transform` effect, you might want the position of the event relative to those transforms. Use the `d3.mouse` operator for the standard mouse pointer, and use `d3.touches` for multi-touch events on iOS.

The preceding paragraph was taken (and slightly modified) from the following website: *https://github.com/mbostock/d3/wiki/Selections#wiki-on*.

One other detail: the radius `r` of each circle is based on the value of `radius*factor`, where factor can be either `0.5` or `1`. The value of `r` changes in such a way as to create a "tubular" effect. If you want all the circles to have the same radius, then simply replace the term `radius*factor` with the variable `radius`.

A DRAG-AND-DROP EXAMPLE (DND)

If you have worked with DnD (Drag-and-Drop) in JavaScript, you will be happy to discover that D3 supports DnD functionality that is both simpler and more robust than the HTML5 version. Moreover, D3 DnD functionality uses a syntax that is similar to other mouse-related events.

Before delving into a DnD code sample, it's worth noting that you can prevent the default DnD behavior (if you need to do so) with this code snippet:

```
function preventBehavior(e) {
   e.preventDefault();
}
document.addEventListener("touchmove", preventBehavior, false);
```

Listing 7.8 displays the contents of `MouseDragAndDrop1.html`, which illustrates how to "drag and drop" a circle in an HTML Web page using D3.

LISTING 7.8: MouseDragAndDrop1.html

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Drag-and-Drop with D3</title>
   <script src="d3.min.js"></script>
 </head>

 <body>
   <script>
    var width=600, height=400, circle1;

    var drag = d3.behavior.drag()
                 .on('dragstart', dragStart)
                 .on('drag',      dragElement)
                 .on('dragend',   dragEnd);

    d3.select("#circle1").call(drag);

    function dragStart(d, i) { }

    function dragElement(d, i) {
       circle1.attr("cx", d3.event.sourceEvent.pageX)
              .attr("cy", d3.event.sourceEvent.pageY);
    }

    function dragEnd(d, i) { }

    var svg = d3.select("body")
                .append("svg")
                .attr("width",  width)
                .attr("height", height);

        circle1 = svg.append("circle")
                     .attr("id", "circle1")
                     .attr("cx", 50)
                     .attr("cy", 50)
                     .attr("r",  30)
                     .attr("fill", "blue")
                     .call(drag);
   </script>
  </body>
 </html>
```

Listing 7.8 contains a `<script>` element that uses the `d3.behavior.drag()` method to define drag-related behavior that is "attached" to an SVG `<circle>` element, as shown here:

```
var drag = d3.behavior.drag()
             .on('dragstart', dragStart)
             .on('drag',      dragElement)
             .on('dragend',   dragEnd);

d3.select("#circle1").call(drag);
```

The preceding code snippet "binds" the JavaScript functions `dragStart()`, `dragElement()`, and `dragEnd()` to the mouse events `dragstart`, `drag`, and `dragend`, respectively. The JavaScript `dragElement()` function is the only one that does something. This function updates the values of the attributes `cx` and `cy` with the current location of a user's mouse using the identical code that you saw in the previous section:

```
circle1.attr("cx", d3.event.sourceEvent.pageX)
       .attr("cy", d3.event.sourceEvent.pageY);
```

Another example of handling drag events in D3 is here: *http://bl.ocks.org/mbostock/3680958*.

The D3 drag-handling code provides other functionality, and the source code is here: *https://github.com/mbostock/d3/blob/master/src/behavior/drag.js*.

The companion disc contains additional DnD code samples, such as the HTML Web page `DragAndDropTheDots1.html`, which combines mouse events, delayed animation, and scaling effects and applies them to all the circles in the code sample. The simple yet clever way to render the initial set of circles in a grid-like layout (using a combination of the `d3.range()` method and the `d3.map()` method) is from a code sample by Mike Bostock: *http://bl.ocks.org/mbostock/1557377*.

Since these preceding code samples use D3 animation effects, you can read their contents after reading the next section, which introduces you to animation effects in D3.

ANIMATION EFFECTS WITH D3

The code sample in this section shows you how to create simple animation effects with D3. Listing 7.9 displays the contents of `Transition1.html`, which illustrates how to render a rectangle and programmatically change its position and dimensions to create an animation effect.

LISTING 7.9: Transition1.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Animation Effects with D3</title>
```

```
  <script src="d3.min.js"></script>
 </head>

 <body>
  <script>
    var width    = 600, height  = 400;
    var sWidth   = 100, sHeight = 100;
    var offsetX  = 100, offsetY = 100;
    var newWidth = 200, opacity = 0.5;
    var duration = 2000, delay   = 400; // milliseconds

    var svg = d3.select("body")
                .append("svg:svg")
                  .attr("width",  width)
                  .attr("height", height);

    var mySquare = svg.append("rect")
      .attr("x", offsetX)
      .attr("y", offsetY)
      .attr("width", sWidth)
      .attr("height",sHeight)
      .style("fill","red");

    mySquare
      .transition()
      .duration(duration)
      .style("opacity",opacity);

    svg.on("click", function() {
      mySquare
        .transition()
        .duration(duration)
        .delay(delay)
        .attr("x", 20)
        .attr("y", 20)
        .style("fill","blue")
        .style("opacity", 1.0)
        .attr("width", newWidth);
    })
 </script>
</body>
</html>
```

Listing 7.9 contains a `<script>` element that starts by defining an SVG `<rect>` element. Next, Listing 7.9 changes the value of the `opacity` attribute from `1` (which is the default value) to `0.5`, as shown here:

```
mySquare
    .transition()
    .duration(duration)
    .style("opacity",opacity);
```

As you can see, the preceding code snippet uses the D3 `.transition()` function to change the `opacity` attribute because it is specified in the `.style()` method that immediately follows the `.transition()` method in the code snippet.

Finally, Listing 7.9 specifies a click event handler for the SVG `<rect>` element that changes the values of the attributes `x`, `y`, `fill`, `opacity`, and `width` to the values that are specified in the final code block.

Incidentally, a very good animation sequence is here, and after you have read the book chapters covering bar charts and graphs, it's worth your time to read the code that creates the multiple animation effects: *http://bl.ocks.org/mbostock/1256572*.

EASING FUNCTIONS IN D3

In non-technical terms, an easing function defines the manner in which an animation effect is created. In real life, the perception of motion depends on the vantage point of the viewer. For example, if you are in an airplane that is one thousand meters in the air, the cars below will appear to move at a constant rate. In D3 you can simulate this type of motion with the linear easing function because the transition from the start state to the end state is performed in constant time.

As a second (and more complex) example of perceived motion, imagine that you are directly facing a baseball player (from a safe distance) and you watch the player hit a baseball into the air. The ball initially seems to move very quickly, then slow down and almost "hang" in the air for a short period of time, and then quickly drop to the ground.

There are several other concepts that you will encounter when you work with easing functions. The first is "ease-in," which means "start slow and speed up." The second is "ease-out," which means "start fast and slow to a stop." The third is "ease-in-out" which is a combination of the previous two: start slow, speed up, and then slow down again.

Now that you have a basic understanding of easing functions and how they create different motion effects, the good news is that D3 supports an extensive set of easing functions, including:

```
Linear: the identity function t
poly(k): raises t to the specified power k (such as
   3, 4, etc.)
quad: equivalent to poly(2)
cubic: equivalent to poly(3)
sin: applies the trigonometric function sin
exp: raises 2 to a power based on t
```

You can specify an easing function in D3 as follows:

```
.ease("cubic"); // or any other easing function
```

Navigate to the following link to see a complete list of D3 easing functions and their description: *https://github.com/mbostock/d3/wiki/Transitions#wiki-d3_ease*.

The companion disc also contains a set of code samples that illustrate how to use the D3 easing functions with animation effects. The easing function is embedded in the filename. For example, the HTML Web page `MouseMoveFadeAnim1Bounce1.html` shows you how to use the "bounce" easing function, which simulates the motion of a bouncing object.

ZOOM, PAN, AND RESCALE EFFECTS WITH D3

This section does not delve into the details of how to create zoom, pan, and rescale effects, but since this type of functionality might be useful for some people, some links are included so that you can read the D3 code.

Although the following code sample for creating a zoom effect is more sophisticated than the other examples in this chapter, you have seen almost every D3 method in the code (except for working with CSV files, which is covered in Chapter 4): *http://mbostock.github.com/d3/talk/20111116/pack-hierarchy.html*.

The following HTML Web page illustrates how to create zoom, pan, and rescale effects: *https://gist.github.com/stepheneb/1182434*.

Note that the preceding Web page contains a line graph, so the code will make more sense to you after reading about line graphs in Chapter 4.

HANDLING KEYBOARD EVENTS WITH D3

In addition to mouse events and touch events, D3 supports keyboard events, which involves examining the value of `d3.event.keycode`. Listing 7.10 displays the contents of `Keyboard1.html`, which illustrates how to detect keyboard events and examples of determining specific keys.

LISTING 7.10: Keyboard1.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>D3 and Keyboard Events</title>
    <script src="d3.min.js"></script>
  </head>

  <body>
    <script>
     d3.select("body").on("keydown", function() {
       switch (d3.event.keyCode) {
         case 8:  console.log("delete");      break;
         case 32: console.log("space");       break;
         case 33: console.log("page up");     break;
         case 34: console.log("page down");   break;
         case 37: console.log("left arrow");  break;
         case 39: console.log("right arrow"); break;
         case 65: console.log("lower a");     break;
         case 97: console.log("upper A");     break;
```

```
            default: return;
          }

          d3.event.preventDefault();
      });
    </script>
  </body>
</html>
```

Listing 7.10 is straightforward: a `switch` statement checks several well-known values that represent various keys on a keyboard and then prints a message that is visible in the Web Inspector.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The three HTML Web pages `Cone2Anim1.html`, `Pyramid2Anim1.html`, and `BezierCurves2Anim1.html` add animation effects to the HTML Web pages

`Cone2.html`, `Pyramid2.html`, and `BezierCurves2.html` that you saw in Chapter 6. The HTML Web page `SimpleSketch1.html` is a variation of the code in `FollowTheMouse1.html` that shows you how you can render freestyle sketches.

The HTML Web page `IterateArrays3.html` illustrates how to handle multiple mouse events and how to render a set of line segments in order to create a so-called Moiret effect.

The HTML Web page `D3Timer1.html` uses the `d3.timer()` method to create an animation effect that changes the opacity value for a set of randomly generated circles. You can easily modify this code sample to create "fade in" and "fade out" effects.

The HTML Web page `MouseMoveFadeAnimation1.html` detects a `mouseover` event, an easing function, and a judicious choice of opacity values to create a very pleasing animation effect.

The HTML Web page `MultiPartialBlueSphereCB5Anim.html` creates scaling-based animation effects, and also scales the set of partial spheres during `mouseover` events.

SUMMARY

This chapter showed you how to work with multidimensional JavaScript arrays, followed by an example of using such arrays in a scatter plot with D3. Next you learned about scaling functions in D3 and the D3 `Path Data Generator`. You also learned how to handle mouse events, followed by some mouse-related code samples (such as "follow the mouse") with D3. Finally, you learned how to create animation effects in HTML Web pages using D3. The next chapter is devoted to rendering bar charts with D3.

CHAPTER 8

DATA VISUALIZATION

Data visualization is a feature of various desktop applications as well as mobile applications, and there are dozens of software packages (open source and commercial) that provide data visualization. This chapter discusses data visualization and presents various code samples for 2D visualization using SVG and D3.

As you learned in a previous chapter, D3 enables you to write JavaScript code that acts as a layer of abstraction over SVG. The initial examples are presented in pure SVG, followed by examples of creating D3-based data visualization. After completing this chapter, you will be in a better position to decide when to use pure SVG versus D3 for data visualization (hint: D3 is probably a superior choice for any non-trivial scenario).

The first part of this chapter shows you how to create a simple 2D bar chart and a multi-line graph in SVG. You will also learn how to handle mouse-related events and how to create animation effects in SVG. The second part of this chapter uses D3 to produce similar functionality. As you will see, D3 makes it very easy to handle mouse events and create animation. The final part of this chapter provides information about integrating SVG, CSS, and jQuery in the same Web page, which is important because real-world Web pages often involve a combination of technologies for data visualization. This section contains a code sample that illustrates when it's necessary to specify the SVG namespace during the creation of new SVG elements.

WHAT IS DATA VISUALIZATION?

One of the goals of data visualization is to present data in ways that enable users to make inferences about relationships that exist in data sets. Data visualization has many facets, and a more detailed explanation about data visualization is here: *https://en.wikipedia.org/wiki/Data_visualization*.

D3 is arguably one of the most powerful open source toolkits for data visualization because of its tremendous flexibility and facility for rendering data in a myriad of ways.

Although this chapter is primarily about D3, there are other data visualization toolkits available. Explore the features of the toolkits that are available to determine which ones are best suited for your specific needs, such as the ones described here: *https://codegeekz.com/30-best-tools-for-data-visualization/*.

Please keep in mind that the SVG and D3 examples in this chapter provide a rudimentary introduction to data visualization. The code samples are no more than a starting point to provide you with a foundation regarding data visualization. In addition, the D3-based code samples reference the D3 JavaScript file that is co-located with the HTML Web pages, which you can replace by referencing the D3 JavaScript file from a CDN (Content Delivery Network).

With the preceding points in mind, the next section shows you how to create a very simple SVG-based bar chart.

A SIMPLE 2D BAR CHART WITH PURE SVG

Listing 8.1 displays the contents of `BarChart2D1.svg`, which illustrates how to create a 2D bar chart in SVG.

LISTING 8.1: BarChart2D1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg id="bargraph" width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="lgradient1"
                    x1="0%" x2="0%" y1="10%" y2="100%">
        <stop style="stop-color:#888888" offset="0"></stop>
        <stop style="stop-color:#FFF000" offset="0.2"></stop>
        <stop style="stop-color:#0000FF" offset="1"></stop>
    </linearGradient>

    <linearGradient id="lgradient2"
                    x1="0%" x2="0%" y1="10%" y2="100%">
        <stop style="stop-color:#888888" offset="0"></stop>
        <stop style="stop-color:#FFF000" offset="0.2"></stop>
        <stop style="stop-color:#FF0000" offset="1"></stop>
    </linearGradient>
  </defs>

  <g transform="translate(50,50)">
    <rect width="50" height="30"  x="0"   y="170"
          fill="url(#lgradient1)"></rect>
    <rect width="50" height="80"  x="50"  y="120"
          fill="url(#lgradient2)"></rect>
    <rect width="50" height="50"  x="100" y="150"
          fill="url(#lgradient1)"></rect>
```

```
    <rect width="50" height="100" x="150" y="100"
          fill="url(#lgradient2)"></rect>
    <rect width="50" height="120" x="200" y="80"
          fill="url(#lgradient1)"></rect>
    <rect width="50" height="180" x="250" y="20"
          fill="url(#lgradient2)"></rect>
    <rect width="50" height="150" x="300" y="50"
          fill="url(#lgradient1)"></rect>
    <rect width="50" height="130" x="350" y="70"
          fill="url(#lgradient2)"></rect>
  </g>
</svg>
```

Listing 8.1 starts with an SVG `<defs>` element that defines two linear gradients whose `id` attributes have the values `lgradient1` and `lgradient2`, respectively. The next portion of Listing 8.1 contains a `<g>` element with a transform attribute that shifts the chart to the right and downward by 50 units. The `<g>` element also contains eight SVG `<rect>` elements whose `fill` attribute alternates between the two linear gradients. Each SVG `<rect>` element is a "bar" element in the 2D bar chart.

Figure 8.1 displays the graphics image that is rendered by the code in the HTML Web page in Listing 8.1.

A 2D BAR CHART WITH PURE SVG ANIMATION

Listing 8.2 displays the contents of `AnimBarChart2D1.svg`, which illustrates how to add animation effects to a 2D bar chart in SVG.

LISTING 8.2: AnimBarChart2D1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg id="bargraph" width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
```

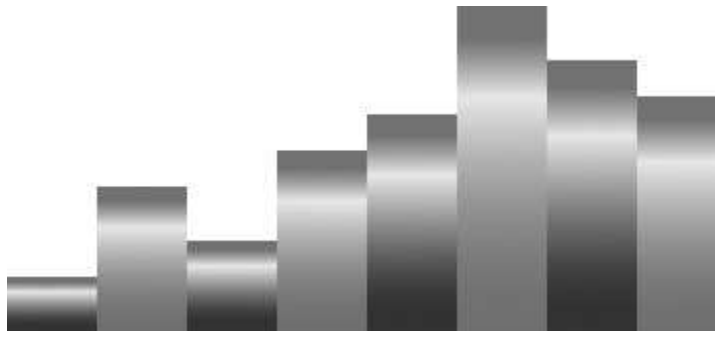

FIGURE 8.1: An SVG 2D bar chart.

```
<defs>
  <linearGradient id="lgradient1"
                  x1="0%" x2="0%" y1="10%" y2="100%">
      <stop style="stop-color:#888888" offset="0"></stop>
      <stop style="stop-color:#FFF000" offset="0.2"></stop>
      <stop style="stop-color:#0000FF" offset="1"></stop>
  </linearGradient>

  <linearGradient id="lgradient2"
                  x1="0%" x2="0%" y1="10%" y2="100%">
      <stop style="stop-color:#888888" offset="0"></stop>
      <stop style="stop-color:#FFF000" offset="0.2"></stop>
      <stop style="stop-color:#FF0000" offset="1"></stop>
  </linearGradient>
</defs>

<g transform="translate(50,400) scale(1,-1)">
  <rect width="50" height="30"  x="0"   y="170"
        fill="url(#lgradient1)">
     <animate attributeName="height" from="0" to="30"
              dur="2s" fill="freeze">
     </animate>
  </rect>

  <rect width="50" height="80"  x="50"  y="170"
        fill="url(#lgradient2)">
     <animate attributeName="height" from="0" to="80"
              dur="2s" fill="freeze">
     </animate>
  </rect>

  <rect width="50" height="50"  x="100" y="170"
        fill="url(#lgradient1)">
     <animate attributeName="height" from="0" to="50"
              dur="2s" fill="freeze">
     </animate>
  </rect>

  <rect width="50" height="100" x="150" y="170"
        fill="url(#lgradient2)">
     <animate attributeName="height" from="0" to="100"
              dur="2s" fill="freeze">
     </animate>
  </rect>

  <rect width="50" height="120" x="200" y="170"
        fill="url(#lgradient1)">
     <animate attributeName="height" from="0" to="120"
              dur="2s" fill="freeze">
     </animate>
  </rect>

  <rect width="50" height="180" x="250" y="170"
        fill="url(#lgradient2)">
     <animate attributeName="height" from="0" to="180"
              dur="2s" fill="freeze">
```

```
      </animate>
    </rect>

    <rect width="50" height="150" x="300" y="170"
          fill="url(#lgradient1)">
       <animate attributeName="height" from="0" to="150"
                dur="2s" fill="freeze">
       </animate>
    </rect>

    <rect width="50" height="130" x="350" y="170"
          fill="url(#lgradient2)">
       <animate attributeName="height" from="0" to="130"
                dur="2s" fill="freeze">
       </animate>
    </rect>
  </g>
</svg>
```

Listing 8.2 starts with an SVG `<defs>` element that defines two linear gradients whose `id` attributes have the values `lgradient1` and `lgradient2`, respectively. The next portion of Listing 8.2 contains a `<g>` element with a transform attribute that shifts the chart to the right and downward by `50` units. The `<g>` element also contains eight SVG `<rect>` elements whose `fill` attribute alternates between the two linear gradients. Each SVG `<rect>` element is a "bar" element in the 2D bar chart.

In addition, each `<rect>` element contains an SVG `<animate>` element that specifies the animation effect that is produced whenever the code is launched in a browser. For example, the first `<rect>` element contains an `<animate>` element that changes the height of the rectangle from `0` to `30` in `2` seconds, as shown here:

```
<animate attributeName="height" from="0" to="30"
         dur="2s" fill="freeze">
</animate>
```

The other SVG `<rect>` elements contain similar `<animate>` elements.

Figure 8.2 displays the graphics image that is rendered by the code in the HTML Web page in Listing 8.2.

A 2D BAR CHART WITH MOUSE EVENTS AND PURE SVG ANIMATION

The code sample in this section shows you how to initiate animation effects when users hover over the bar elements in the 2D bar chart. The animation effects involve changing the color of the bar elements when during `mousemove` and `mouseout` events involving the bar elements.

Listing 8.3 displays a portion of the contents of `MouseAnimBarChart2D1.svg` that illustrates how to combine mouse-related events with animation effects in a 2D bar chart in SVG.

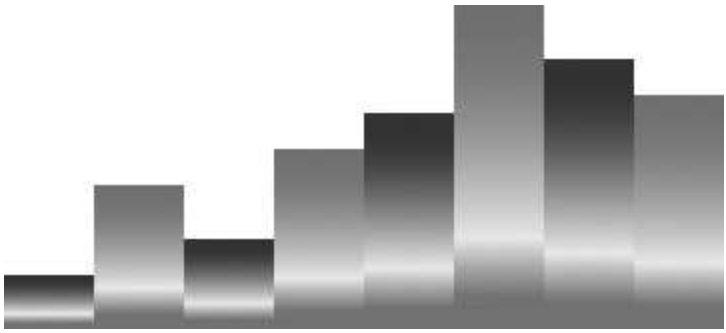

FIGURE 8.2: A 2D bar chart with SVG animation.

LISTING 8.3: MouseAnimBarChart2D1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg id="bargraph" width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="lgrad1"
                    x1="0%" x2="0%" y1="10%" y2="100%">
        <stop style="stop-color:#888888" offset="0"></stop>
        <stop style="stop-color:#FFF000" offset="0.2"></stop>
        <stop style="stop-color:#0000FF" offset="1"></stop>
    </linearGradient>

    <linearGradient id="lgrad2"
                    x1="0%" x2="0%" y1="10%" y2="100%">
        <stop style="stop-color:#888888" offset="0"></stop>
        <stop style="stop-color:#FFF000" offset="0.2"></stop>
        <stop style="stop-color:#FF0000" offset="1"></stop>
    </linearGradient>
  </defs>

  <g transform="translate(50,50)">
    <rect width="50" height="30"  x="0"   y="170" fill="url(#lgrad1)">
      <set attributeName="fill" to="blue" begin="mousemove"/>
      <set attributeName="fill" to="red"  begin="mouseout"/>
    </rect>

    <rect width="50" height="80"  x="50"  y="120" fill="url(#lgrad2)">
      <set attributeName="fill" to="red"  begin="mousemove"/>
      <set attributeName="fill" to="blue" begin="mouseout"/>
    </rect>

    <!-- <rect> elements omitted for brevity -->
  </g>
</svg>
```

FIGURE 8.3: A 2D bar chart with mouse events and animation.

Listing 8.3 starts with an SVG `<defs>` element that defines two linear gradients whose `id` attributes have the values `lgradient1` and `lgradient2`, respectively. The next portion of Listing 8.3 contains a `<g>` element with a transform attribute that shifts the chart to the right and downward by 50 units. The `<g>` element also contains eight SVG `<rect>` elements whose `fill` attribute alternates between the two linear gradients. Each SVG `<rect>` element is a "bar" element in the 2D bar chart.

In addition, each `<rect>` element contains two SVG `<set>` elements that specify the animation effect that is produced whenever users move their mouse inside and outside the SVG `<rect>` element. The following code block displays the contents of the first SVG `<rect>` element in Listing 8.3:

```
<rect width="50" height="30"  x="0"   y="170" fill="url(#lgrad1)">
  <set attributeName="fill" to="blue" begin="mousemove"/>
  <set attributeName="fill" to="red"  begin="mouseout"/>
</rect>
```

Figure 8.3 displays the graphics image that is rendered by the code in the HTML Web page in Listing 8.3.

A LINE GRAPH IN SVG

Listing 8.4 displays the contents of `LineGraph2D1.svg`, which illustrates how to create a line graph in SVG.

LISTING 8.4: LineGraph2D1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg id="linegraph" width="100%" height="100%"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">

  <g transform="translate(0,0)">
    <!-- render the line graph -->
```

```
    <line x1="50"  y1="30"  x2="100" y2="150"
          stroke-width="3" stroke="blue"/>
    <line x1="100" y1="150" x2="150" y2="200"
          stroke-width="3" stroke="red"/>
    <line x1="150" y1="200" x2="200" y2="80"
          stroke-width="3" stroke="blue"/>
    <line x1="200" y1="80"  x2="250" y2="300"
          stroke-width="3" stroke="red"/>
    <line x1="250" y1="300" x2="300" y2="150"
          stroke-width="3" stroke="blue"/>
    <line x1="300" y1="150" x2="350" y2="120"
          stroke-width="3" stroke="red"/>
    <line x1="350" y1="120" x2="400" y2="220"
          stroke-width="3" stroke="blue"/>
    <line x1="400" y1="220" x2="450" y2="150"
          stroke-width="3" stroke="red"/>

    <!-- render the circles on the vertices -->
    <circle cx="50"  cy="30"  r="5" fill="black"/>
    <circle cx="100" cy="150" r="5" fill="black"/>
    <circle cx="150" cy="200" r="5" fill="black"/>
    <circle cx="200" cy="80"  r="5" fill="black"/>
    <circle cx="250" cy="300" r="5" fill="black"/>
    <circle cx="300" cy="150" r="5" fill="black"/>
    <circle cx="350" cy="120" r="5" fill="black"/>
    <circle cx="400" cy="220" r="5" fill="black"/>
    <circle cx="450" cy="150" r="5" fill="black"/>

    <!-- render the horizontal axis -->
    <line x1="20" y1="320" x2="500" y2="320"
          stroke-width="3" stroke="black"/>

    <!-- render the horizontal arrow -->
    <path d="M500,320 v0,10 l20,-10 l-20,-10 z"
          stroke-width="3" fill="black"/>

    <!-- render the vertical axis -->
    <line x1="20" y1="320" x2="20" y2="20"
          stroke-width="3" stroke="black"/>

    <!-- render the vertical arrow -->
    <path d="M20,20 h10,0 l-10,-20 l-10,20 z"
          stroke-width="3" fill="black"/>
  </g>
</svg>
```

Listing 8.4 starts with a set of SVG `<line>` elements for rendering the line graph, followed by a set of SVG `<circle>` elements that are rendered on the vertices of the line segments. The final portion of Listing 8.4 renders a horizontal axis with an arrow tip and a vertical axis with an arrow tip.

The code in Listing 8.4 is simple and can be created manually, but since the values are hard-coded, it's not possible to generate a line graph with dynamic data values. In the latter case, you can do so by creating an SVG document with JavaScript embedded in a `CDATA` section, as shown in the next section.

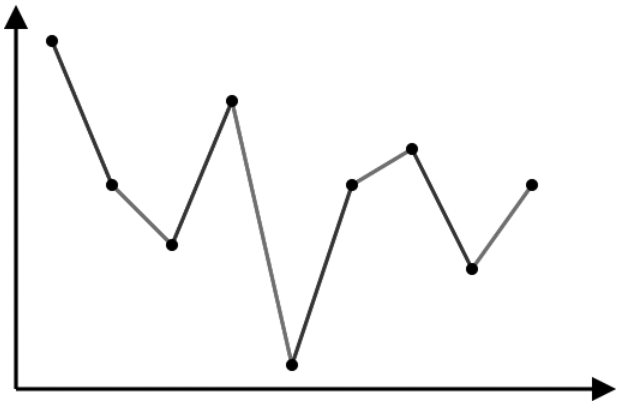

FIGURE 8.4: An SVG line graph.

Figure 8.4 displays the graphics image that is rendered by the code in the HTML Web page in Listing 8.4.

A 2D LINE GRAPH WITH JAVASCRIPT IN SVG

Listing 8.5 displays the contents of `LineGraphGrid2D1.svg`, which illustrates how to use JavaScript in order to create a line graph with a grid background in SVG.

LISTING 8.5: LineGraphGrid2D1.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg width="100%" height="100%" onload="init(evt)"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg">

  <script type="text/ecmascript">
    <![CDATA[
     var basePointX  = 150, basePointY  = 0;
     var indentX     = 50,  indentY     = 50;
     var offsetX     = 0,   offsetY     = 0;
     var strokeWidth = 3,   strokeColor = "black";
     var radius      = 5,   lineWidth   = 3;
     var leftX       = 20,  topX        = 20;
     var cellWidth   = 50,  cellHeight  = 50;
     var rowCount    = 6,   colCount    = 10;
     var hTriangle   = "M500,320 v0,10 l20,-10 l-20,-10 z";
     var vTriangle   = "M20,20 h10,0 l-10,-20 l-10,20 z";
     var dataXValues = [50, 100, 150, 200, 250, 300, 350, 400, 450];
     var dataYValues = [30, 150, 200, 80, 300, 150, 120, 220, 150];

     var fillColors  = ["#f00", "#00f"], style = "";
     var lineNode    = null, pathNode = null;
```

```
var circleNode  = null, gcNode   = null;
var svgNS       = "http://www.w3.org/2000/svg";

function init(evt) {
   gcNode = document.getElementById("gc");
   drawGrid();
   drawLineGraph();
}

function drawLineGraph() {
   // render the line graph
   for(var i=0; i<dataXValues.length-1; i++) {
      lineNode = document.createElementNS(svgNS, "line");
      lineNode.setAttribute("x1", dataXValues[i]);
      lineNode.setAttribute("y1", dataYValues[i]);
      lineNode.setAttribute("x2", dataXValues[i+1]);
      lineNode.setAttribute("y2", dataYValues[i+1]);
      lineNode.setAttribute("stroke-width", strokeWidth);
      lineNode.setAttribute("stroke",
          fillColors[i%fillColors.length]);
      gcNode.appendChild(lineNode);
   }

   // render the circles on the vertices
   for(var i=0; i<dataXValues.length; i++) {
      circleNode = document.createElementNS(svgNS, "circle");
      circleNode.setAttribute("cx", dataXValues[i]);
      circleNode.setAttribute("cy", dataYValues[i]);
      circleNode.setAttribute("r",  radius);
      circleNode.setAttribute("fill",
          fillColors[i%fillColors.length]);
      gcNode.appendChild(circleNode);
   }

   // render the horizontal axis -->
   lineNode = document.createElementNS(svgNS, "line");
   lineNode.setAttribute("x1", 20);
   lineNode.setAttribute("y1", 320);
   lineNode.setAttribute("x2", 500);
   lineNode.setAttribute("y2", 320);
   lineNode.setAttribute("stroke-width",strokeWidth);
   lineNode.setAttribute("stroke",strokeColor);
   gcNode.appendChild(lineNode);

   // render the horizontal arrow -->
   pathNode = document.createElementNS(svgNS, "path");
   pathNode.setAttribute("d", hTriangle);
   pathNode.setAttribute("stroke",strokeColor);
   gcNode.appendChild(pathNode);

   // render the vertical axis -->
   lineNode = document.createElementNS(svgNS, "line");
   lineNode.setAttribute("x1", 20);
   lineNode.setAttribute("y1", 320);
   lineNode.setAttribute("x2", 20);
   lineNode.setAttribute("y2", 20);
```

```
         lineNode.setAttribute("stroke-width",strokeWidth);
         lineNode.setAttribute("stroke",strokeColor);
         gcNode.appendChild(lineNode);

         // render the vertical arrow -->
         pathNode = document.createElementNS(svgNS, "path");
         pathNode.setAttribute("d", vTriangle);
         pathNode.setAttribute("stroke",strokeColor);
         gcNode.appendChild(pathNode);
      }

      function drawGrid() {
         style =  "fill:none;stroke:red;";
         style += "stroke-dasharray:2 2 2 2;line-width:1";

         for(var row=0; row<rowCount; row++) {
            for(var col=0; col<colCount; col++) {
               offsetX = leftX+(col)*cellWidth;
               offsetY = topX+(row)*cellHeight;

               cellNode = document.createElementNS(svgNS, "rect");
               cellNode.setAttribute("x", offsetX);
               cellNode.setAttribute("y", offsetY);
               cellNode.setAttribute("width",  cellWidth);
               cellNode.setAttribute("height", cellHeight);
               cellNode.setAttribute("style", style);
               gcNode.appendChild(cellNode);
            }
         }
      }
   ]]>
</script>

<!-- ============================ -->
<g id="gc" transform="translate(0,20)">
   <rect x="0" y="0"
         width="800" height="500"
         fill="none" stroke="none"/>
</g>
</svg>
```

Listing 8.5 renders a line graph with the same data values as the line graph in Listing 8.4. In addition, Listing 8.5 renders a background grid with a dashed pattern.

The first part of Listing 8.5 initializes a set of JavaScript variables for the line graph. The next portion of Listing 8.5 contains the JavaScript function `init()` that is invoked when the SVG document is launched in a browser. This function initializes the `gcNode` JavaScript variable, which is a reference to the SVG `<g>` element that is displayed at the bottom of Listing 8.5.

The `init()` function then invokes the function `drawGrid()` that contains a nested loop for rendering a rectangular grid of dashed rectangles. The final portion of the `init()` function invokes the JavaScript function `drawLineGraph()` that renders the line segments.

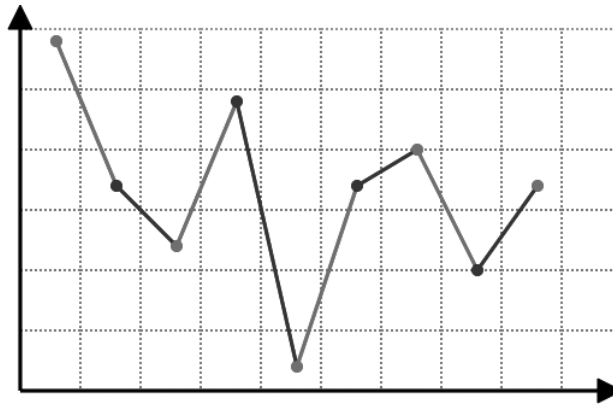

FIGURE 8.5: An SVG line graph with JavaScript.

The function `drawLineGraph()` contains four sections of code. The first section is a loop that renders the line segments, where the data values are stored in two JavaScript arrays, `dataXValues` and `dataYValues`. As you can see, these arrays contain the same data values as Listing 8.5. The second section is a loop that renders the circles on the vertices of the line segments. The third section renders a horizontal axis and arrow, and the fourth section renders a vertical axis and arrow.

Figure 8.5 displays the graphics image that is rendered by the code in the HTML Web page in Listing 8.5.

Now that you have learned how to render a line graph in SVG using JavaScript, modify the code in Listing 8.5 in order to render a set of line graphs. You can create a multidimensional JavaScript array to store the data values for each line graph.

A LINE GRAPH IN D3

Listing 8.6 displays the contents of `D3LineGraphDateAxis1.html`, which illustrates how to use D3 in order to create a line graph with a date-based horizontal axis in D3.

LISTING 8.6: D3LineGraphDateAxis1.html

```
<html>
<head>
   <meta charset="utf-8" />
   <title></title>
   <script src="d3.min.js"></script>

   <style>
     .yaxis path, .yaxis line {
        fill: none;
        stroke: red;
```

```
        shape-rendering: crispEdges;
     }

     .xaxis path, .xaxis line {
        fill: none;
        stroke: red;
        shape-rendering: crispEdges;
     }
   </style>
</head>

<body>
  <script>
    var margin = { top: 20, right: 20, bottom: 30, left: 50 },
        width = 600 - margin.left - margin.right,
        height = 300 - margin.top - margin.bottom;

    var xScale = d3.time.scale()
                   .range([0, width]);

    var yScale = d3.scale.linear()
                   .range([height,0]);

    var xAxis = d3.svg.axis()
                  .scale(xScale)
                  .orient("bottom");

    var yAxis = d3.svg.axis()
                  .scale(yScale)
                  .orient("left");

    // create <svg> element
    var svg = d3.select("body")
                .append("svg")
                .attr("width", width + margin.left + margin.right)
                .attr("height", height + margin.top + margin.bottom)
                .append("g")
                .attr("transform",
                      "translate("+margin.left+","+margin.top+")");

    // get the data values
    var dataValues = []; // [{year:"", revenue:""}]
    var startYear=1990, yearCount = 20;

    var parseDate = d3.time.format("%Y").parse;

    for(var year=startYear; year<startYear+yearCount; year++) {
      var level = Math.floor(height*Math.random());
      dataValues.push({year : parseDate(String(year)), revenue: level})
    }

    var line = d3.svg.line()
                 .x(function (d) { return xScale(d.year); })
                 .y(function (d) { return yScale(d.revenue); });

    xScale.domain(d3.extent(dataValues, function (d) {
```

```
        return d.year; }));

    yScale.domain(d3.extent(dataValues, function (d) {
        return d.revenue; }));

    // display the data
    svg.append("path")
        .datum(dataValues)
        .attr("class", "line")
        .attr("fill", "none")
        .attr("stroke", "red")
        .attr("d", line(dataValues));

    // render the x axis
    svg.select("g")
         svg.append("g")
             .attr("class", "xaxis")
             .attr("transform", "translate(0," + height + ")")
             .call(xAxis);

    // render the y axis
    svg.append("g")
        .attr("class", "yaxis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Millions (USD)");
  </script>
</body>
</html>
```

Listing 8.6 contains boilerplate code and a `<script>` element that defines various JavaScript variables. The JavaScript array `dataValue` is populated with JSON-based objects containing `year` and `revenue` properties that represent an endpoint of a line segment. The next portion of Listing 8.6 initializes variables that scale the x-values and the y-values of the endpoints of the line segments, followed by the definition of a variable that acts as a "line function," as shown here:

```
var line = d3.svg.line()
             .x(function(d){return xScale(d[0]);})
             .y(function(d){return yScale(d[1]);})
```

The next portion of Listing 8.6 creates and then appends an SVG `<svg>` element to the `<body>` element, followed by a standard D3 idiom that displays a circle over each vertex in the line graph. In addition, users can increase and decrease the radius of the circles during a `mouseover` and `mouseout` event, respectively, as shown in bold in the following code block:

```
// add circles at the vertices...
svg.selectAll("circle")
         .data(dataValues)
         .enter()
         .append("circle")
         .attr("cx", function(d){return xScale(d[0])})
         .attr("cy", function(d){return yScale(d[1])})
         .attr("r", cRadius)
         .style("fill", cColor)
         .style("fill-opacity", cOp)
         .style("stroke", cStroke)
         .on("mouseover", function() {
            var r = d3.select(this).attr("r");
            d3.select(this).attr("r", factor*r);
         })
         .on("mouseout", function() {
            var r = d3.select(this).attr("r");
            d3.select(this).attr("r", r/factor);
         })
```

Notice that all the line segments in Listing 8.6 are rendered with the same color.

A 2D BAR CHART WITH ANIMATION EFFECTS USING D3

Listing 8.7 displays the contents of `D3BarChart2AutoAnimation1.html`, which illustrates how to render a histogram with automatic animation effects in D3.

LISTING 8.7: D3BarChart2AutoAnimation1.html

```
<!DOCTYPE html>
<html lang="en">
 <head>
    <meta charset="utf-8">
    <title>Rendering a Bar Chart with Animation</title>
    <script src="d3.min.js"></script>
 </head>

 <body>
   <p>Click to update the bar chart:</p>

    <script>
      var width=600, height=300, currWidth=0;
      var offsetY=15, roundBand=0.05, theColor="";
      var clickCount=0, weight=0, multiplier=100;
      var duration=1500, delay=1000, textColor="white";
      var easingType = ["bounce", "circle",
                        "elastic", "linear"];

      // initial values for the bar chart...
      var dataValues = [20, 15, 30, 25, 12, 18, 28, 38,
                        21, 32, 26, 20, 18,  8, 16, 18];
```

```
var dataCount = dataValues.length;

var xScale = d3.scale.ordinal()
               .domain(d3.range(dataValues.length))
               .rangeRoundBands([0, width], roundBand);

var yScale = d3.scale.linear()
               .domain([0, d3.max(dataValues)])
               .range([0, height]);

//Create SVG element
var svg = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// Create bar elements...
svg.selectAll("rect")
   .data(dataValues)
   .enter()
   .append("rect")
   .attr("x", function(d, i) {
       return xScale(i);
   })
   .attr("y", function(d) {
       return height - yScale(d);
   })
   .attr("width", xScale.rangeBand())
   .attr("height", function(d) {
       return yScale(d);
   })
   .attr("fill", function(d) {
    return "rgb(0, 0, " + (d * 10) + ")";
   })
   .on("mouseover", function() {
      currWidth = d3.select(this).attr("width");
      d3.select(this).attr("width", currWidth/2)
   })
   .on("mouseout", function() {
      d3.select(this).attr("width", currWidth);
   })

// Create bar labels...
svg.selectAll("text")
   .data(dataValues)
   .enter()
   .append("text")
   .text(function(d) {
       return d;
   })
   .attr("text-anchor", "middle")
   .attr("x", function(d, i) {
       return xScale(i) + xScale.rangeBand() / 2;
   })
   .attr("y", function(d) {
       return height - yScale(d) + offsetY;
   })
```

```
            .attr("font-family", "sans-serif")
            .attr("font-size", "12px")
            .attr("fill", textColor);

         svg.selectAll("circle")
            // generate a set of random values...
            dataValues = [];
            for(var x=0; x<dataCount; x++) {
               randValue = multiplier*Math.random();
               dataValues.push(randValue);
            }

            xScale = d3.scale.ordinal()
                    .domain(d3.range(dataValues.length))
                    .rangeRoundBands([0, width], roundBand);

            yScale = d3.scale.linear()
                    .domain([0, d3.max(dataValues)])
                    .range([0, height]);

            svg.selectAll("rect")
               .data(dataValues)
               .transition()
             //.delay(delay)
               .duration(duration)
               .ease(easingType[0])
               .attr("x", function(d, i) {
                   return xScale(i);
               })
               .attr("y", function(d) {
                   return height - yScale(d);
               })
               .attr("width", xScale.rangeBand())
               .attr("height", function(d) {
                   return yScale(d);
               })
               .attr("fill", function(d) {
                  if(++clickCount % 3 == 0) {
                    theColor = "red";
                  } else if(clickCount % 3 == 1) {
                    theColor = "green";
                  } else if(clickCount % 3 == 2) {
                    theColor = "yellow";
                  }

                  return theColor;
               })
      </script>
   </body>
</html>
```

Listing 8.7 renders an initial vertical bar chart using the same code as Listing 8.6, so a discussion of that code will not be repeated here. The new code in Listing 8.7 involves replacing the current vertical bar chart with a new vertical bar chart whenever users click on the bar chart with their mouse.

The first portion of the D3 code that handles a mouse click event is shown here:

```
svg.on("click", function() {
     ++clickCount;

     // generate a set of random values...
     dataValues = [];
     for(var x=0; x<dataCount; x++) {
        randValue = multiplier*Math.random();
        dataValues.push(randValue);
     }
```

The preceding code block sets the JavaScript array `dataValues` to an empty array, followed by a simple loop that uses random values in order to populate the array with a new set of numbers.

The next portion of D3 code (which is also in the same click handler function) resets the scale-related variables for the horizontal and vertical values of the bar elements, as shown here:

```
// reset the scaled x and y values...
xScale = d3.scale.ordinal()
           .domain(d3.range(dataValues.length))
           .rangeRoundBands([0, width], roundBand);

yScale = d3.scale.linear()
           .domain([0, d3.max(dataValues)])
           .range([0, height]);
```

The preceding code is the same as the code that you have seen in previous code samples, and it is required every time a new vertical bar chart is rendered.

The next portion of code in the click handler uses a standard D3 idiom to render a set of bar elements based on the newly generated set of numbers in the `dataValues` array. The color for each bar element is determined with the following code block:

```
.attr("fill", function(d) {
    weight = Math.floor((d*10) % 255);
    colorR = "rgb("+weight+", 0, 0)";
    colorG = "rgb(0,"+weight+",0)";
    colorB = "rgb(0, 0,"+weight+")";

    if(clickCount % 3 == 0) {
      theColor = colorR;
    } else if(clickCount % 3 == 1) {
      theColor = colorG;
    } else if(clickCount % 3 == 2) {
      theColor = colorB;
    }

      return theColor;
});
```

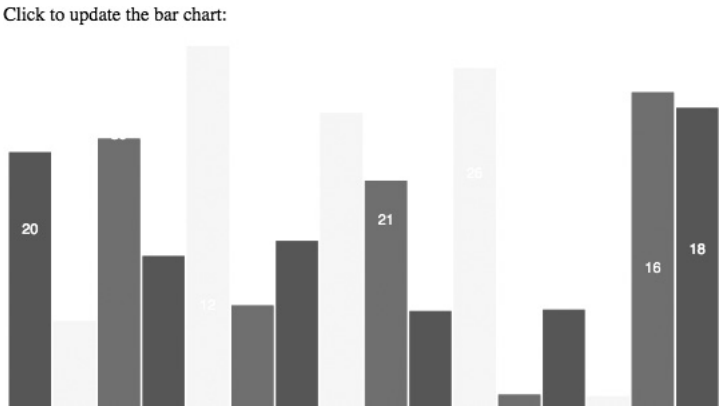

FIGURE 8.6: An updated bar chart in a Chrome browser on a MacBook Pro.

The preceding code block uses the quantity `d*10 % 255` (which is an integer between `0` and `255`) to compute an `(R,G,B)` color value. Notice that the variables `colorR`, `colorG`, and `colorB` contain a single non-zero component, which will render a variation of red, green, and blue, respectively.

The second portion of the preceding code block uses conditional logic to determine which color to assign to the variable `theColor` that is used to render the color of the current bar element.

Keep in mind that although the preceding block of code for setting the color is probably more complex than your needs, it's included here so that you will learn how to incorporate this type of functionality in HTML Web pages if you need to do so.

The final block of code in the click event handler sets the text labels to display the height of each bar element in the vertical bar chart.

Figure 8.6 displays an updated bar chart in a Chrome browser on a MacBook Pro.

SVG, CSS, AND JQUERY

You can combine these technologies and D3 in the same Web page. However, you need to add some extra code (shown later) when you dynamically create SVG elements in a Web page with jQuery. If you do not plan to use jQuery, feel free to skip this section. If you are interested in jQuery and have not worked with jQuery, read the relevant Appendix on the companion disc.

Listing 8.8 displays the contents of `JQuerySVGCSSCube1.html`, which illustrates how to update the fill color of an SVG-based cube whenever users click on any of its faces.

LISTING 8.8: JQuerySVGCSSCube1.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
 <meta charset="utf-8" />
 <title>SVG, CSS, and jQuery Example</title>

 <script src="https://cdnjs.cloudflare.com/ajax/libs/
              jquery/3.0.0-alpha1/jquery.js">
 </script>

 <style>
  .newcolor {
    fill: #c8f;
    stroke: white;
    stroke-width: 2;
  }
 </style>
</head>

<body>
 <svg class="none" width="600" height="300">
  <!-- top/front/right faces -->
  <polygon fill="yellow"
           points="50,50 200,50 240,30 90,30"/>
  <rect width="150" height="150" x="50" y="50"
        fill="red"/>
  <polygon fill="blue"
           points="200,50 200,200 240,180 240,30"/>
 </svg>

 <script>
   var mysvg = $("svg > *");

   $(document).ready(function() {
      mysvg.on("click", function() {
        mysvg.removeClass("none");
        $(this).toggleClass("newcolor");
      //$(this).newClass("newcolor");
      });
   });
 </script>
</body>
</html>
```

Listing 8.8 contains boilerplate code and familiar SVG code for rendering a cube. The new code is the jQuery code block in the `<script>` element, which shows you how to define an event handler that responds to click events on any of the faces of the cube.

HOW TO CREATE NEW SVG ELEMENTS WITH JQUERY

The `document.createElement()` to add SVG DOM Nodes does not work because a namespace is required. Use the `document.createElementNS()` method, which jQuery does not handle for you.

Listing 8.9 displays the contents of `JQuerySVG.html`, which illustrates how to create an SVG ellipse and then append that ellipse to the DOM via jQuery.

LISTING 8.9: JQuerySVG.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>An SVG Ellipse and jQuery</title>

    <script src="http://code.jquery.com/jquery-1.7.1.min.js">
    </script>
  </head>

  <body>
    <svg id="outersvg" width="800" height="500">
    </svg>

    <script>
      function createSVGElemWithNS(elem)
      {
        return document.createElementNS('http://www.w3.org/2000/svg',
                                         elem);
      }

      $(document).ready(function() {
         var $mySVG = $('#outersvg');

         $(createSVGElemWithNS('circle'))
                          .attr('cx', 150)
                          .attr('cy', 100)
                          .attr('r', 50)
                          .attr('fill', 'none')
                          .attr('stroke', 'red')
                          .attr('stroke-width', 3)
                          .appendTo($mySVG);

      })
    </script>
  </body>
</html>
```

Listing 8.9 starts with boilerplate code and a `<body>` element that contains an `<svg>` element that is the container for SVG elements. The next part of the `<body>` element is a `<script>` element that starts with the JavaScript function `createSVGElemWithNS` whose sole purpose is to create elements with the SVG namespace. The second part of the `<script>` element contains the standard jQuery "ready" code block that is executed after the Web page is loaded into memory. As you can see, this code block first obtains a reference to the `<svg>` element in the Web page and then invokes the JavaScript function `createSVGElemWithNS` to create an SVG `<circle>` element.

Method chaining is used to assign values to the SVG <circle> element before appending the element to the DOM.

HOW TO MODIFY EXISTING SVG ELEMENTS WITH JQUERY

The previous section showed you how to dynamically create new SVG elements and append them to the DOM with jQuery. You can also manipulate existing SVG elements with jQuery, and you can do so without having to deal with namespaces.

Listing 8.10 displays the contents of UpdateSVGViaJQuery.html, which illustrates how to use jQuery in order to update the width attribute of an SVG <rect> element.

LISTING 8.10: UpdateSVGViaJQuery.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
   <meta charset="utf-8" />
   <title>Updating an SVG Rectangle with jQuery</title>

   <script src="http://code.jquery.com/jquery-2.0.0b1.js">
   </script>
  </head>

  <body>
   <svg width="600" height="400">
      <rect id="rect1" x="50" y="50" width="200"
            height="50" fill="red"/>

      <rect id="rect2" x="50" y="150" width="200"
            height="50" fill="red"/>
   </svg>

   <script>
     $(document).ready(function() {
       $('#rect1').attr('width', 400);
     });
   </script>
  </body>
</html>
```

Listing 8.10 contains boilerplate code with two <script> elements that reference jQuery-related scripts. The <body> element in Listing 8.10 contains an SVG <svg> element that in turn contains two <rect> elements, followed by a <script> element that contains a jQuery code block. After the Web page is loaded into memory, this code block executes and sets the width attribute of the first SVG <rect> element to 400. This simple code sample shows you how easy it is to update attributes of SVG elements via jQuery.

jQuery Plugins for SVG

If you prefer to use jQuery instead of D3 in your HTML5 Web pages, you can add SVG support by means of jQuery plugins. For example, the following jQuery plugin provides some SVG support: *http://keith-wood.name/svg.html*.

The following code block illustrates how to render several 2D shapes using this jQuery plugin:

```
svg.rect(50, 50, 100, 100,
         {fill: 'blue', stroke: 'red', 'stroke-width': 3});
svg.line(50, 50, 100, 100,
         {'stroke': 3, 'stroke-width': 3});
svg.circle(50, 50, 100, 100,
           {'fill': 'red', 'stroke': 3, 'stroke-width': 3});
svg.ellipse(250, 250, 100, 50,
            {'fill': 'red', 'stroke': 3, 'stroke-width': 3});
```

In the preceding code block, the JavaScript variable `svg` is a reference to an HTML `<div>` element that is defined elsewhere in an HTML5 Web page.

A jQuery plugin for animating SVG `<path>` elements is here: *http://www.jqueryrain.com/2015/10/jquery-drawsvg-animate-svg-paths/* and *http://www.justinmccandless.com/blog/Patching+jQuery's+Lack+of+SVG+Support*.

SVG, CSS, AND THE `classlist` METHOD

Listing 8.10 uses jQuery to update the fill color of an SVG-based cube whenever users click on any of its faces. However, you can achieve the same functionality with the `classlist` method instead of jQuery.

Listing 8.11 displays the contents of `ClassListSVGCube1.html`, which illustrates how to use the `classlist` method in order to update the fill color of an SVG-based cube whenever users click on any of its faces.

LISTING 8.11: ClassListSVGCube1.html

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <title>SVG, CSS, and jQuery Example</title>

  <style>
   .newcolor {
     fill: #484;
     stroke: white;
     stroke-width: 2;
   }
  </style>
 </head>

 <body>
  <svg id="mysvg" class="none" width="600" height="300">
   <!-- top/front/right faces -->
```

```
    <polygon id="topface" fill="yellow"
             points="50,50 200,50 240,30 90,30"/>
    <rect id="frontface" width="150" height="150" x="50" y="50"
          fill="red"/>
    <polygon id="rightface" fill="blue"
             points="200,50 200,200 240,180 240,30"/>
  </svg>

  <script>
    var topface = document.getElementById("topface");
    topface.addEventListener('click', function() {
console.log("class list = "+topface.classList.toString());
      topface.classList.toggle('newcolor'); }, false);

    var frontface = document.getElementById("frontface");
    frontface.addEventListener('click', function() {
      frontface.classList.toggle('newcolor'); }, false);

    var rightface = document.getElementById("rightface");
    rightface.addEventListener('click', function() {
      rightface.classList.toggle('newcolor'); }, false);
  </script>
 </body>
</html>
```

Notice that the SVG elements in Listing 8.11 now contain an `id` attribute, and the `<script>` element uses the value of the `id` attribute in order find those SVG elements and then add a click handler. Notice that a click handler is added to each face of the cube, whereas there is only one click handler for all three faces in Listing 8.10. The addition of an individual click handler can become impractical if you have many elements in a Web page, and the use of jQuery (or some other toolkit) is probably a better alternative.

Incidentally, the `DOMTokenList` specification provides the following methods that can be used on the `classList`:

- Add() for adding a class to the list
- Remove() for removing a class from the list
- Contains() to check if a class is in the list
- Toggle() for toggling a class in and out of the list (with a twist)
- Item() to return the class at a specified position in the list
- ToString() for turning the list into a string
- Length to return the number of classes in the list
- Value to add custom properties and methods to the classList object

HOW TO OBTAIN FOCUS IN SVG ELEMENTS WITH CSS

Listing 8.12 displays the contents of `SVGFocus.html`, which illustrates how to use CSS in order to change the highlighted SVG element based on user click events.

LISTING 8.12: SVGFocus.html

```
<!doctype html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <title>Example of SVG Focus</title>

  <style>
    a:focus {
      fill: red;
      fill-opacity: 0.4;
    }
  </style>
 </head>

 <body>
  <svg width="600px" height="300px">
    <g>
      <a xlink:href="#">
        <circle cx="100" cy="100" r="30" />
      </a>
      <a xlink:href="#">
        <circle cx="180" cy="100" r="30" />
      </a>
      <a xlink:href="#">
        <circle cx="260" cy="100" r="30" />
      </a>
      <a xlink:href="#">
        <circle cx="340" cy="100" r="30" />
      </a>
    </g>
  </svg>
 </body>
</html>
```

Listing 8.12 starts with a CSS selector that changes the opacity of an `<a>` element whenever users click on such a link. The next part of Listing 8.12 is the `<body>` element that contains an SVG `<svg>` element, which in turn contains a `<g>` element. As you can see, the `<g>` element contains four `<a>` elements, each of which contains an SVG `<circle>` element. Each time users click on one of the circles, its color changes to a pale red color.

BOOTSTRAP 4 AND SVG

Listing 8.13 displays the contents of `BS4SVG.html`, which illustrates how to combine SVG and Bootstrap 4 in an HTML Web page.

LISTING 8.13: BS4SVG.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
   <meta charset="utf-8" />
   <title>SVG and Bootstrap 4</title>
```

```
<!-- Bootstrap 4 requirements:
   1) jQuery:  1.9 <= version < 3.0
   2) BS 4 tooltips require tether
-->

<link rel="stylesheet" type="text/css"
      href="https://cdnjs.cloudflare.com/ajax/libs/
            tether/1.1.1/css/tether.css">

<link rel="stylesheet" type="text/css"
  href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
        alpha.2/css/bootstrap.min.css">

<link rel="import"
      href="https://cdnjs.cloudflare.com/ajax/libs/
            webicons/2.0.0/webicons.css">

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.1.1/js/
       tether.js">
</script>

<script src="http://cdnjs.cloudflare.com/ajax/libs/
             jquery/2.1.3/jquery.min.js">
</script>

<script src="http://cdn.rawgit.com/icons8/bower-webicon/
             v0.10.7/jquery-webicon.min.js">
</script>

<script
  src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.2/
       js/bootstrap.min.js">
</script>
</head>
<style>
  webicon {
     position: relative;
     display: inline-block;
     height: 1rem;
     width: 1rem;
  }
</style>

<body>
  <div class='container'>
    <div class='row'>
      <div class='col-sm-6 col-sm-offset-3 col-lg-4 col-lg-
                  offset-4'>
        <div class='list-group'>
          <a class='list-group-item' href='#'>
            <webicon icon='color-icons:about'></webicon>
            About
          </a>
          <a class='list-group-item active' href='#'>
            <webicon icon='color-icons:address-book'></webicon>
```

```
            Address Book
          </a>
          <a class='list-group-item' href='#'>
            <webicon icon='color-icons:sports-mode'></webicon>
            Sports Mode
          </a>
        </div>
        <br>
      </div>
    </div>
  </div>
</body>
</html>
```

Listing 8.13 contains a number of CSS and JavaScript files that are referenced from a CDN. Bootstrap 4 (currently in alpha as this book goes to print) has restrictions on the versions of jQuery that are permitted and also a dependency on the Tether open source toolkit.

The `<body>` element in Listing 8.13 contains nested `<div>` elements, each of which references Bootstrap-specific classes. The innermost `<div>` element contains three `<a>` elements, each of which contains a `<webicon>` element that renders the icons that are highlighted in bold.

One other point: the `<style>` element contains a selector that matches the `<webicon>` methods in order to render very small icons: remove the width and height properties to render the icons with their default sizes.

Open Chrome Inspector and expand the elements until you see any of the `<webicon>` elements; expand those elements and you will see the embedded SVG code. For example, the `<webicon>` element for the "About" icon contains the following SVG code block (which consists of SVG elements that are familiar to you):

```
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 48 48" version="1.0"
     fit="" height="100%" width="100%"
     preserveAspectRatio="xMidYMid meet"
     style="pointer-events: none; display: inline-block;">
  <path fill="#2196F3"
        d="M37 40H11l-6 6V12c0-3.3 2.7-6 6-6h26c3.3 0 6 2.7 6
        6v22c0 3.3-2.7 6-6 6z">
  </path>

  <g fill="#fff">
    <path d="M22 20h4v11h-4z"></path>
    <circle cx="24" cy="15" r="2"></circle>
  </g>
</svg>
```

OTHER THIRD PARTY CHART LIBRARIES

If you prefer not to create custom charts and graphs, you can use existing libraries for rendering charts and graphs as an alternative to creating your own

charts and graphs. One advantage of using existing libraries is the fact that you do not need to be responsible for bug fixes, and you can avail yourself of upgrades containing new features or enhancements to those libraries.

You can also find third party chart libraries that use jQuery, and several are discussed here: *http://www.1stwebdesigner.com/css/top-jquery-chart-libraries-interactive-charts/*.

You might also be interested in using the `jqPlot` chart library, which is a jQuery plugin whose home page is here: *http://www.jqplot.com/*.

`jqPlot` provides "hooks" for adding your own custom event handlers and for creating new plot types. Moreover, `jqPlot` has been tested on Safari, Firefox, IE, and Opera, and you can even use `jqPlot` on tablets. Some live `jqPlot` samples are available online here: *http://www.jqplot.com/tests/*.

If you prefer to use SVG instead of HTML5 `Canvas`, another toolkit is the SVG Google Charts API. You can see some code samples that use this toolkit here: *http://www.netmagazine.com/tutorials/create-beautiful-data-visualisations-svg-google-charts-api*.

The jQuery Visualize plugin uses Progressive Enhancement (and also provides support for `ARIA`, which is a specification defining support for accessibility of Web applications) to render charts and graphs. A set of detailed code samples and download information are available here: *http://www.filamentgroup.com/lab/update_to_jquery_visualize_accessible_charts_with_html5_from_designing_with/*.

In addition, various other open source packages as well as commercial products are available that provide charts and graphs, and the decision will depend on your budget and the functionality that you need for your Web pages.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

The companion disc contains `AreaGraph2D1.svg` that uses gradient shading in order to create an area chart. Another example is `SineBarChartGraphGrid1.svg` that uses a sine wave to render a 2D bar chart. The `ThreeDGridMultiCBShadowRipple4RG6.svg` code sample uses gradient shading in order to create a more subdued visual effect.

SUMMARY

This chapter showed you how to create 2D bar charts and multi-line graphs using pure SVG. Next you learned how to render 2D bar charts and multi-line graphs using JavaScript and SVG. Then you saw how to create the same charts and graphs using D3 instead of pure SVG. Along the way you learned how to handle mouse events and also create animation effects. The final portion of this chapter showed you how to combine SVG, CSS, and jQuery in the same Web page, and the circumstances in which you need to create SVG elements with the SVG namespace.

CHAPTER 9

DESIGNING MOBILE APPS

The goal of this chapter is to provide you with some guidelines for designing good HTML5 mobile applications. Even if you currently do not work with mobile apps, it's worth skimming through the material because you might find material that will be useful to you in the future. One topic of particular relevance is Google AMP (Accelerated Mobile Pages) because of its support for many SVG elements.

This chapter discusses aspects of the UI (User Interface) experience and how to create aesthetically appealing mobile Web applications. You will also learn about some of the toolkits that can help you support the desired functionality without sacrificing important design considerations. Some of the sections in this chapter involve CSS3 Media Queries, so you do need a basic understanding of this topic.

The first part of this chapter discusses some of the major aspects of the mobile metaphor, which is touch-oriented instead of mouse-oriented. The second part of this chapter delves into design-related aspects of mobile forms. The final portion of this chapter discusses Google AMP (Accelerated Mobile Pages) and Progressive Mobile Apps (also developed by Google). Although Google AMP is still a new technology, it's relevant because Google AMP supports some CSS as well as most SVG elements.

Keep in mind the following points when you read this chapter. First, this chapter contains significantly less code than earlier chapters because its focus involves more stylistic considerations rather than code for creating specific visual effects. Second, there are various toolkits for creating hybrid mobile applications (a detailed discussion is beyond the scope of this book). In particular, PhoneGap (Cordova) is a very popular toolkit, and it has influenced other mobile toolkits. Another toolkit is Ionic (acquired by IBM), which uses Angular 2 for creating hybrid mobile applications and also uses PhoneGap internally. AppGyver is yet another toolkit that extends the functionality in

PhoneGap. In the enterprise space, Oracle and IBM provide frameworks for hybrid mobile development that rely on PhoneGap as one of the components of their frameworks.

Alternatively, there are toolkits and products for developing mobile applications that do not use a DOM structure. React Native from Facebook and Xamarin (acquired by Microsoft) are two such products. Visit the respective Web sites of these toolkits for code samples and tutorials.

WHAT IS GOOD MOBILE DESIGN?

The answer to this simple question is more complex than you might think, but there are some guidelines to help you make design decisions for hybrid HTML5 mobile applications (and also native applications, such as iOS and Android, which are outside the scope of this book). As you will see, some of these guidelines include touch-orientation, performance and responsiveness, detecting screen sizes, resizing assets, and determining the actual content.

The next several sections provide more information about these and other important factors, and then discuss toolkits or provide code snippets that can help you implement the desired functionality.

IMPORTANT FACETS OF MOBILE WEB DESIGN

Some of the important considerations for designing hybrid HTML5 mobile applications are here:

- A touch-oriented design (not mouse-oriented)
- Improving response times of user gestures
- Detecting different screen sizes (especially for Android devices)
- Resizing assets (such as images)
- Determining the content of a Web page

Instead of handling mouse clicks, mobile web applications handle touch-related events (single tap and multiple taps) and user gestures (swipe, flick, pan, and so forth).

Fortunately, jQuery Mobile provides a virtualization of events that takes care of mouse events and touch events through a single set of APIs. This virtualization simplifies your code so you only need to handle one set of events.

A TOUCH-ORIENTED DESIGN

You might think that you can accomplish this goal simply by replacing all mouse-related events with their touch-related counterparts. Although this is a good starting point, there are several caveats to this approach. First, there is no touch event that is the counterpart for a mouse-based hover event. Consequently, CSS stylesheets with selectors of the type `#myDiv:hover` will work

correctly on laptops and desktops, but they do not work correctly on mobile devices.

Second, when users touch a mobile screen with one of their fingers, the second gesture could be one of the following:

1) Touch up
2) Touch hold
3) Touch move
4) Swipe gesture

For this reason, there is a delay (roughly 300 milliseconds) to allow for a second event. The first event (touch down) can be combined with the second event to determine the correct user gesture. However, this delay can affect the perceived responsiveness of a mobile web application. Fortunately, there are JavaScript toolkits that improve touch-related responsiveness of mobile web applications, which is the topic of the next section.

IMPROVING RESPONSE TIMES OF USER GESTURES

There are several toolkits for eliminating the 300 millisecond delay between the first touch event and the second touch event. The first JavaScript toolkit with nice touch-related functionality is `fastclick.js`, and its home page is here: *https://github.com/ftlabs/fastclick*.

Mobile browsers wait about 300 milliseconds after an initial tap event in order to determine if users will perform a double tap. Since FastClick eliminates the 300-millisecond delay between a physical tap and the firing of a click event on mobile browsers, mobile applications feel more responsive when the delay is eliminated.

Listing 9.1 displays the contents of the HTML5 Web page `FastClick1.html`, which illustrates how to use `fastclick.js` in a Web page.

LISTING 9.1: FastClick1.html

```
<html>
<head>
  <meta charset="utf-8">
  <script src='/path/to/fastclick.js'></script>
</head>

<body>
  <script>
    window.addEventListener('load', function() {
        FastClick.attach(document.body);
    }, false);
  </script>
</body>
</html>
```

Listing 9.1 is straightforward: include a `<script>` element that references fastclick.js and then add an event listener that invokes the `attach()` method of `fastclick.js` during a browser load event.

You can also use `fastclick.js` with mobile web applications that use jQuery Mobile, and a detailed description is provided here: *http://forum.jquery.com/topic/how-to-remove-the-300ms-delay-when-clicking-on-a-link-in-jquery-mobile*.

RESIZING ASSETS IN MOBILE WEB APPLICATIONS

There are at least two common techniques for detecting screen sizes, which in turn is related to resizing assets. The first technique involves CSS3 Media Queries to load different CSS stylesheets for different screen widths. The second technique is to use JavaScript to set different CSS properties for different screen sizes.

When you place assets (such as PNG files) on an HTML5 Web page in a mobile Web application, you need to decide how to handle the following situations:

1) Switching from portrait to landscape (or vice versa)
2) Deploying to mobile devices with different screen sizes
3) Resizing text strings and text areas
4) Scaling non-binary images (such as SVG)

You need to address the preceding situations whenever you create an HTML5 Web page that includes not only PNG files, but also other types of assets, such as embedded SVG documents or HTML5 `<canvas>` elements.

In the case of rendering text strings, how do you determine the correct font size? Even if you use `em` for the unit of measure, you don't know the DPI resolution of a particular device. Of course, you could make an educated guess and decide that text will be rendered with `14em` for a mobile phone and `16em` for a tablet (or phablet), but the increasing variety of screens with different DPIs means that you can't guarantee that your choice will be correct in all cases.

In the case of SVG documents that are embedded in an HTML5 Web page, you can use a CSS-based technique for automatically resizing the contents of SVG documents in an HTML5 Web page with this code snippet:

```
<div style="background-size: contain">
```

A detailed description of this technique is here: *https://developer.mozilla.org/en-US/docs/Web/CSS/background-size*.

DETERMINING THE CONTENT LAYOUT FOR MOBILE WEB PAGES

Some design considerations for determining the content of a Web page are here:

- Provide a single column of text for smart phones
- Remove extra links and content

- Remove items from sidebars/footers
- Avoid absolute sizes
- Set wrapper widths to percentages
- Set paragraphs to display block

Keep in mind that the preceding points are guidelines, and not absolute rules. For example, some smart phones in landscape mode can accommodate two columns, depending on the width of the columns, and many tablets can display two columns in portrait mode. Some people recommend that you reduce the amount of scrolling that is required in a screen, but many users are accustomed to frequent scrolling on a mobile device (especially on smart phones). When in doubt, create two (or possible more) layouts and get feedback from users to determine which layout might have greater appeal to a broad audience. If you still cannot decide which layout design is best for your purposes, perhaps you can benefit from "A/B testing," which is described here: *http://en.wikipedia.org/wiki/A/B_testing*.

MOBILE DESIGN FOR HTML WEB PAGES

There are several approaches to designing HTML Web pages for mobile devices, including "mobile first," "mobile only," and using a separate domain for mobile applications.

A "mobile first" approach ensures that an HTML Web page will render correctly on a mobile device. However, you need to take into account different screen sizes among mobile devices, such as smart phones, "phablets," and tablets. This detail is particularly evident with Android mobile devices (and to a lesser extent it's true for iOS devices).

A "mobile only" approach still requires you to take into account different screen sizes on mobile devices, but in this scenario you do not need to contend with HTML Web pages for laptops or desktops. Keep in mind that this approach is far from trivial: it can involve a combination of client (or server) templates, local data stores, and efficient view-model binding that are provided by toolkits such as `BackboneJS`. Another point to consider is whether or not you can leverage NoSQL datastores, such as MongoDB.

A third approach is to use separate domains for mobile versus desktop Web pages. If you are developing thin mobile web applications (i.e., Web pages that are hosted on a server and accessed by various device types), this approach provides several advantages. First, this makes your mobile site easier to find. Second, you can advertise the mobile url separately from the url for desktop devices. Third, users can switch between the "regular" website and the mobile website simply by changing the domain. Fourth, the code logic for detecting mobile users (and then sending them to a separate domain) is simpler than making modifications to CSS stylesheets.

Now that you have an overview of design considerations for mobile Web pages, let's take a look at an example of styling mobile forms, which is the topic of the next section.

HIGH-LEVEL VIEW OF STYLING MOBILE FORMS

When you render a mobile form, there are high-level techniques (such as how to display input fields) as well as techniques that are specific in nature (such as indicating required fields).

The following list contains some high-level techniques for rendering mobile forms:

- Using semantic markup
- Using a single column to display input fields (for smart phones)
- Deciding on label alignment (left or top)
- Grouping/chunking input fields
- Tooltips or information bubbles
- Displaying error messages
- Appropriate field width
- Suitable color scheme
- Tabs instead of radio buttons
- Handling long drop-down lists (use predictive search or links)
- Primary and secondary buttons
- Spacing between input elements
- Pop-up menu controls
- Voice input

Notice that there is no "reduce scrolling" in the previous list: users are accustomed to scrolling on smart phones and tablets. In addition, a single-column display was more important for smaller smart phones, but some smart phones can display two columns in landscape mode, and tablets are even more accommodating in terms of multiple columns of text.

If you render text in a left-to-right direction, then left-aligned labels enable you to place more input fields in a screen, and they are also easier to scan during field input, but they don't work well with long input fields. Top-aligned labels have the opposite advantages and disadvantages of left-aligned labels. Yet another technique is use of inline labeling, which essentially means that the "text prompt" for the input field provides a suggestion for the type of data that is expected.

SPECIFIC TECHNIQUES FOR STYLING MOBILE FORMS

In addition to the design techniques that are listed in the previous section, you can use some or all of these specific techniques:

- Use CSS to style input fields differently
- Provide labels for input fields
- Specify keyboard types for input fields
- Provide default or suggested values for input fields

- Use constraints for numeric fields
- Indicate required fields
- Validate each field after user input
- Use password masking
- Use field zoom

The following subsections provide additional details for some of the bullet items in the preceding list.

Use CSS to Style Input Fields Differently

There are two steps involved. The first step is to apply the default styling and then apply your own styling rules. The second step is to use the `[type=]` attribute selector to apply different styles to different input fields.

As a specific example you can use the `appearance` property to remove the default styling for all input elements, as shown here:

```
input, textarea, select {
   appearance: none;
}
```

Next, specify your own CSS styles that depend on the type of input field, as outlined here:

```
input[type=checkbox]
   /* specify styles for checkbox */
}

input[type=radio] {
   /* specify styles for radio */
}

input[type=textarea] {
   /* specify styles for textarea */
}
```

You can use the preceding example as a guideline for defining CSS selectors to change the appearance of other HTML elements in your Web pages.

Specify Keyboard Types for Input Fields

HTML5 introduced the following new properties for the type attribute for input elements:

```
an email address: type='email'
a website address: type='url'
telephone number: type='tel'
```

The preceding types only work on mobile browsers (nothing will happen in a laptop or desktop browser).

The following example shows you how to specify different keyboards for input fields:

```
<form id="form1" name="form1" method="post" action="">
<label for="name">Name:</label><input name="name" id="name"
type="text" /><br />
<label for="email">Email:</label><input name="email" id="email"
type="email" /><br />
<label for="phone">Phone:</label><input name="phone" id="phone"
type="tel" /><br />
<input name="submit" type="button" value="Submit" />
</form>
```

Different Countries and Languages

There are several points to keep in mind when you create applications for different countries and languages:

- Different phone formats in different countries
- Left-to-right versus right-to-left text input
- Culturally appropriate content

If you intend to publish Android applications that support multiple languages, the following link provides useful information: *http://developer.vodafone.com/how-support-multiple-languages-android/*.

If you intend to publish iOS applications that support multiple countries and languages, Apple provides extensive information here: *https://developer.apple.com/internationalization/*.

The details of creating internationalized iOS mobile applications (such as how to create additional string resources) are here: *http://www.slideshare.net/cxpartners/web-and-mobile-forms-design-userfriendly-2010-workshop*.

DESIGN-RELATED TOOLS AND ONLINE PATTERNS

`Balsamiq` is an online design tool, and its home page is here: *http://www.balsamiq.com/*.

`Balsamic Mockups` is a tool for rapidly creating wireframes, designed to reproduce the experience of sketching interfaces on a whiteboard. Since these mockups are on your computer, you can share them quickly and easily with other people.

Another toolkit for creating wireframes is `justinmind`, and its home page is here: *http://justinmind.com*.

Both of the preceding tools are free to use, and they have videos that illustrate how to use them.

Two websites that provide a collection of design patterns (the second is purely for mobile) are here: *http://pttrns.com* and *http://mobile-patterns.com*.

WORKING WITH FONT SIZES AND UNITS OF MEASURE

If you have already worked with CSS stylesheets, then you are already familiar with different units that are available for expressing font sizes for text in HTML5 Web pages. Several units are available, including `em`, `rem`, `px`, `pt`, and `%`.

The relationship among these units is as follows: `1em = 12pt = 16px = 100%`

Pixel units (`px`) are very common in Web pages (especially for desktops and laptops), and they can vary in terms of their resolution and their DPI (dots per inch). Pixel units are virtual screen pixels rather than physical pixels. One point to keep in mind is that the pixel unit does not scale correctly for visually-impaired readers.

You can use the `px` unit for text, images, borders, rounded corners, and drop shadows. However, if you are building a fluid layout that uses relative sizes, it's probably better to use the `em` unit or the `%` unit for text.

The `em` unit is becoming more popular, especially on mobile devices, because this unit is well suited for scaling a Web page. Unlike the `px` unit, the `em` unit is relative to its parent element in a Web page. It might be helpful to think of the `px` unit as "global" whereas the `em` unit is "local."

One useful technique is to specify the value `62.5%` for the `<body>` element. Since 62.5% equals the fraction `5/8`, the product of `16px` and `5/8` is `10px`. The value `10px` is a convenient "base point" for performing conversions quickly and easily between `px` units and `em` units in CSS selectors. For example, `1.5em` is `15px`, `2em` is `20px`, `2.5em` is `25px`, and so forth. In case you need help in performing conversions, an online `em` calculator is here: *http://riddle.pl/emcalc/* and *http://www.ready.mobi/launch.jsp?locale=en_EN*.

Additional useful information is here: *http://www.w3.org/QA/Tips/font-size*.

WHAT IS GOOGLE AMP?

Google AMP (Accelerated Mobile Pages) is an open source initiative for rendering Web pages more quickly on mobile devices, and its home page is here: *https://www.ampproject.org/*.

The initial AMP specification was announced in October 2015 and code samples are available here: *https://github.com/ampproject/amphtml*.

Accelerated Mobile Pages can be loaded in any modern browser or Web view. However, AMPs impose a restricted subset of functionality compared to "regular" HTML Web pages in order to reduce the time to load AMPS in a browser. Moreover, AMPs can be cached in the cloud, which can yield additional speed improvements.

Some of the AMP restrictions for AMP-compliant Web pages are here:

No developer-written or third-party JavaScript
No input elements of any kind, including standard input and textarea

No external style sheets and only one style tag in the document head
No inline styles
Style rules must be at or below 50kb

Consumers of the AMP specification include content producers who can create AMP Web pages that can be crawled, indexed, and displayed. The good news is that Google AMP supports most SVG elements, and perhaps additional support will be available in the future.

NOTE *Google AMP supports most SVG elements.*

You can think of AMP as a framework for creating mobile web pages that consists of AMP HTML (a subset of HTML with AMP-specific elements), AMP JS (a JavaScript framework for AMP that disallows third-party JavaScript), and an optional AMP CDN.

One of the goals of AMP is to support all published content in Accelerated Mobile Pages.

More information about AMP is available in the following FAQ: *https://www.ampproject.org/docs/support/faqs.html*.

The RAIL Framework

RAIL is an acronym for the following time-based measurements: Reaction time (100ms), Animation time (60FPS), Idle time (<50ms), and Load time (< one second). The parenthesized numbers in the preceding sentence are the goals of the respective time measurements. The availability of functionality such as Service Workers (albeit an experimental technology) in modern browsers will assist in reaching the RAIL goal of adequate performance of Web pages.

Currently there is partial support for Service Workers in Chrome, Opera, and Firefox, and more information about Service Workers is here: *https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API*.

Styles in AMP

All styles must appear in a `style` tag in the head of the document the style tag must contain the `amp-custom` attribute.

```
<amp-lightbox id="lightbox" class="lightbox" layout="nodisplay">
    <div class="lightbox-content">
      <amp-img
        src="https://cdn.auth0.com/website/jobs/myimage.jpg"
        width="2000"
        height="1035"
        layout="responsive">
      </amp-img>
      <p> some text in a paragraph </p>
    </div>
</amp-lightbox>
```

AMP components support for ads includes: A9, AdReactor, AdSense, AdTech, and Doubleclick. Companies involved in Google AMP include Twitter, Pinterest, WordPress.com, Chartbeat, Parse.ly, Adobe Analytics, and LinkedIn.

The Status of HTML Elements and Core AMP Components

Elements prohibited by AMP include:

base, frame, frameset
object, param, applet
embed, form, script (*)
input, textarea, select, option

(*) unless it's of type "application/ld+json"

Allowed HTML elements (with caveats in some cases) include: button, link, style, most SVG elements, and some support for stylesheets.

Built-in "core" AMP components are:

amp-img, amp-audio, amp-video
amp-pixel, amp-anim, amp-iframe
amp-carousel, amp-lightbox, amp-ad
amp-instagram, amp-twitter, amp-youtube

NOTE *Built-in components are always available in an AMP document and they have an* `<amp-*>` *element.*

The Core/Extended AMP components include the following:

amp-img amp-audio amp-video
amp-pixel amp-anim amp-iframe
amp-carousel amp-lightbox amp-ad
amp-instagram amp-twitter amp-youtube

Elements replaced by AMP include `img (amp-img)`, `video (amp-video)`, `audio (amp-audio)`, and `iframe (amp-iframe)`.

An example of the `<amp-img>` element is here:

```
<amp-img
  src="https://mysite/myimage.jpg"
  width="2000"
  height="1035"
  layout="responsive">
</amp-img>
```

An AMP extended component involves loading the JavaScript in the `<head>` element. Navigate to the following link to see an example of a simple AMP Web page: *https://www.ampproject.org/docs/get_started/create/basic_markup.html*.

WHAT ARE PROGRESSIVE WEB APPS?

Progressive Web Apps are based on pages in browser tabs, and they use the app shell model to give them a "look and feel" that is similar to native mobile applications. Progressive Web Apps are responsive (they work with any form factor), safe (served via HTTPS), and always up-to-date because of their use of Service Workers.

A progressive web app performs the following steps:

1) Registers a service worker
2) Runs on HTTPS (required in order to prevent man-in-the-middle attacks)
3) Creates a JSON-based app manifest file with information about the app

Service Workers consist of a collection of APIs (by Google) for offline access and Web push notifications. The manifest file contains metadata, such as icons, orientation, and so forth. Note that the Polymer startup kit contains a valid manifest file. Chrome for Android and Opera for Android allow you to save the app on your homescreen, which means it looks just like a native application on your homescreen. You can open the app with a default orientation. A key point is that the app works like a "normal" website in browsers that do not support progressive apps. An example of a progressive Web app manifest is here: *https://mobiforge.com/design-development/web-app-manifests-usher-new-wave-progressive-apps-to-your-homescreen*.

IMPROVING HTML5 WEB PAGE PERFORMANCE

Performance is always a very important consideration, and although a detailed and lengthy discussion is beyond the constraints of this book, it's important to be aware of some of the techniques that are available:

- Concatenate JavaScript/CSS files into a single file
- Use file compression (minification)
- Use spritesheets
- Use base64 encoding of images
- Defer the loading of JavaScript files

A good blog post regarding the use of base64 for encoding images is here: *http://davidbcalhoun.com/2011/when-to-base64-encode-images-and-when-not-to?goback=%2Egde_2071438_member_234328886*.

You can defer the loading of JavaScript files by placing `<script>` tags at the end of the `<body>` element in a Web page.

You can also explore `RequireJS`, which is a JavaScript toolkit for asynchronously downloading JavaScript files (as well as managing JavaScript dependencies), and its home page is here: *http://requirejs.org/*.

Steve Souders has written at least two performance-related books and has written many blog posts regarding performance, and it's definitely worth reading his work.

USEFUL LINKS

The following links are helpful for testing Mobile applications:

- *http://css3test.com/*
- *http://www.css3.info/selectors-test/*
- *http://quirksmode.org/html5/tests/video.html*
- *http://double.co.nz/video_test/*
- *http://www.terrillthompson.com/tests/html5-audio.html*

Other resources include www.stackoverflow.com and videos by Paul Irish.

A more recent website is `HTML5 Please`, which consists of contributions from well-known industry people: *http://html5please.com*.

The preceding website provides an input field where you can specify HTML5 and CSS3 features to determine if they are ready for use, and also see how to use them.

The W3C provides a free online validation service for HTML Web pages, including HTML5 Web pages: *http://validator.w3.org/#validate_by_uri+with_options*.

The preceding website enables you to validate a URL, a file, or direct input of code.

The following website provides "boilerplate" templates for HTML5 email input fields (and also templates for CSS and jQuery): *http://favbulous.com/post/848/6-useful-web-development-boilerplates*.

The following two links are for the HTML5 draft specification and the HTML5 Draft Recommendation (May 2009): *http://www.w3.org/TR/html5/* and *http://www.whatwg.org/specs/web-apps/current-work/*.

There is also a "web developer edition" of the HTML5 specification that is streamlined for readability (without vendor-oriented details) that you can read here: *http://developers.whatwg.org*.

SUMMARY

In this chapter, you learned about the touch-oriented metaphor of the mobile environment, along with design-related aspects of mobile forms. In addition, you saw how to improve the response time of user gestures via fastclick.js, and discovered that you can use CSS3 Media Queries and JavaScript to detect the screen sizes of mobile devices. You also learned about Google AMP and its support for many SVG elements. Next you learned about Progressive Mobile Apps, and a high-level view of performance-related details.

CHAPTER 10

MISCELLANEOUS TOPICS

This chapter discusses various other SVG features that are not covered in previous chapters. The intent of this chapter is to expose you to ways to combine SVG with other technologies, including Angular 2 (with TypeScript), ReactJS, and GSAP, which is impressive when you consider the fact that SVG is almost two decades old. In addition to the rather eclectic nature of this chapter, sections vary considerably in terms of their length.

Given the breadth of topics in this chapter, the ones that will appeal to you will probably be different for another person, and therefore no sections are marked "optional." Although some topics in this chapter probably won't be relevant to you right now, it's possible that they will be relevant in the future.

The first section briefly covers some utilities for SVG, followed by CSS3 Media Queries in SVG. The second section discusses SVG and Sprites, SVG Icons, and JavaScript toolkits for SVG. The third section covers SVG animation effects, including GreenSock (GSAP). You will also see a condensed comparison of CSS, SVG, and HTML5 `Canvas`, along with a performance comparison of these technologies. The fourth section contains examples of combining SVG with Angular 2 and ReactJS, and also an example of ReactJS and local CSS. The final section discusses responsive SVG and also SVG and ARIA.

SVG UTILITIES AND IDEs

There are various toolkits and IDEs available that support SVG, some of which are discussed in the following subsections. The descriptions are brief and highlight some of their features, and obviously you will find much more information by visiting their respective websites.

Adobe Illustrator

Adobe Illustrator (AI) is commercial product from Adobe that can handle very complex artwork, and its home page is here: *http://www.adobe.com/products/illustrator.html*.

AI supports a wide variety of file formats, including PDF, EPS, FXG, Photoshop (PSD), TIFF, GIF, JPEG, SWF, SVG, DWG, and DXF. AI is designed with integration with other tools, such as Adobe Photoshop®, InDesign®, After Effects®, and Acrobat®. Filter effects are notoriously slow, whereas AI can handle them with more than adequate speed.

The SVG Export workflow (File > Export > SVG) exports web-optimized SVG files for Web and screen design workflows. An export option allows you to export individual objects instead of an entire art board. AI Draw also provides support for Android and iOS.

Scour

Vector editors such as Inkscape and Adobe Illustrator generate extra code in SVG documents, and Scour reduces the size of those SVG documents by removing redundant code. Keep in mind that Scour does not always preserve the original rendering of SVG documents (so keep track of your origin files).

Scour removes various types of elements, such as empty elements, metadata elements, and unused id attribute values. Scour also removes unrenderable elements and vector editor metadata. Scour is included as an Inkscape extension (described in the next section), and as a Python package on some Linux distributions: *https://github.com/codedread/scour*.

Inkscape

The Inskcape IDE is an open source tool that is very good for creating SVG code, and you can download a copy here: *https://inkscape.org/en/download*.

Inkscape supports object creation, drawing tools (pen, pencil, calligraphy, shape tools (like SVG), and text tools (including multi-line text). Inkscape also supports Grouping objects, layers, alignment, fill and stroke, color selector (RGB, HSL, CMYK, color wheel, CMS), and a gradient editor with support for multi-stop gradients.

Inkscape provides compliant SVG format file generation, and support for other export formats, including PNG, OpenDocument Drawing, DXF, sk1, PDF, EPS, and PostScript.

Inkscape SVG Filter tutorial: *http://tavmjong.free.fr/blog/?m=201509*.

CSS3 MEDIA QUERIES AND SVG

CSS3 media queries are very useful logical expressions that enable you to detect mobile applications on devices with differing physical attributes and orientation. For example, with CSS3 media queries you can change the dimensions and layout of your applications so that they render appropriately on smart phones as well as tablets.

Specifically, you can use CSS3 media queries in order to determine the following characteristics of a device:

- Browser window width and height
- Device width and height
- Orientation (landscape or portrait)
- Aspect ratio
- Device aspect ratio
- Resolution

CSS3 media queries are Boolean expressions that contain one or more "simple terms" (connected with `and` or `or`) that evaluate to `true` or `false`. Thus, CSS3 media queries represent conditional logic that evaluates to either `true` or `false`.

As an example, the following link element loads the CSS stylesheet `mystuff.css` only if the device is a screen and the maximum width of the device is `480px`:

```
<link rel="stylesheet" type="text/css"
      media="screen and (max-device-width: 480px)" href="mystuff.css"/>
```

The preceding link contains a media attribute that specifies two components: a `media` type of `screen` and a query that specifies a `max-device-width` whose value is `480px`. The supported values for media in CSS3 media queries are `braille`, `embossed`, `handheld`, `print`, `projection`, `screen`, `speech`, `tty`, and `tv`.

The next CSS3 media query checks the `media` type, the `maximum-device-width`, and the `resolution` of a device:

```
@media screen and (max-device-width: 480px) and (resolution: 160dpi) {
  #innerDiv {
    float: none;
  }
}
```

If the CSS3 media query in the preceding code snippet evaluates to `true`, then the nested CSS selector will match the HTML element whose `id` attribute has the value `innerDiv`, and its `float` property will be set to `none` on any device whose maximum screen width is `480px`. As you can see, it's possible to create compact CSS3 media queries that contain non-trivial logic, which is obviously very useful because CSS3 does not have any `if/then/else` construct that is available in other programming languages.

JAVASCRIPT TOOLKITS FOR SVG

There are various other JavaScript toolkits available for rendering SVG, some of which are described in this section.

Snap.Svg

The `snap.svg` toolkit is an open source toolkit for rendering SVG, and its home page is here: *www.snapsvg.io*.

In addition to generating SVG, Snap can also work with SVG generated from tools such Adobe Illustrator, Inkscape, or Sketch. Moreover, Snap enables you to load SVG text strings asynchronously and extract the desired portions in order to convert SVG files into sprite sheets.

Snap.svg is available under an Apache 2 license, and you can download samples from the following Github repository: *https://github.com/adobe-webplatform/Snap.svg*.

RuneJS

Rune.js is a library that uses SVG in order to create graphic design systems, in a browser or with NodeJS, and its home page is here: *http://runemadsen.github.io/rune.js/*.

Rune.js supports a chainable drawing API, a scene graph, and various features for graphic designers: native support for color conversion, grid systems, typography, pixel iteration, as well as an expanding set of computational geometry helpers. One other interesting aspect of Rune.js is its use of a virtual-dom.

FabricJS

Fabric.js is a powerful and simple JavaScript canvas library, and its home page is here: *www.fabricjs.com*.

Fabric provides an interactive object model on top of a canvas element. Fabric also has SVG-to-canvas (and canvas-to-SVG) parser.

PAPERJS FOR SVG

Paper.js is an open source vector graphics toolkit that runs on top of the HTML5 `Canvas`. Paper.js provides a Scene Graph / Document Object Model and a lot of powerful functionality to create and work with vector graphics and Bezier curves.

Listing 10.1 displays the contents of `ConchoidTubeModEllipses1.html`, which illustrates how to render SVG graphics with Paper.js.

LISTING 10.1: ConchoidTubeModEllipses1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Conchoid Ovals</title>
  <script type="text/javascript" src="paper.js"></script>

  <script type="text/paperscript" canvas="canvas">
      var fillColor  = "rgb(255, 0, 0)";
      var basePointX = 200, basePointY = 200;
      var rectWidth = 200, rectHeight = 120;
```

```
      var currentX  = 0, currentY = 0;
      var deltaAngle = 3, maxAngle = 720;
      var Constant1 = 120, Constant2 = 80, newNode;
      var rVal = 0, gVal = 0, bVal = 0;
      var stripCount = 10, currStrip = 0;
      var stripWidth = Math.floor(maxAngle/stripCount);
      var factor, path;

      for(var angle=0; angle<=maxAngle; angle+=deltaAngle) {
         radius   = Constant1+Constant2/
                      Math.cos(angle*Math.PI/180);

         offsetX  = radius*Math.cos(angle*Math.PI/180);
         offsetY  = radius*Math.sin(angle*Math.PI/180);
         currentX = basePointX+offsetX;
         currentY = basePointY-offsetY;

         rVal = 0; gVal = 0; bVal = 0;
         currStrip = Math.floor(angle/stripWidth);

         if(currStrip % 2 == 0) {
           factor = 1.0;
         } else {
           factor = 0.5;
         }

         if(currStrip % 3 == 0) {
           rVal = Math.floor(255*(angle%stripWidth)/stripWidth);
         }
         else if(currStrip % 3 == 1) {
           rVal = Math.floor(255*(angle%stripWidth)/stripWidth);
           gVal = Math.floor(255*(angle%stripWidth)/stripWidth);
         }
         else {
           bVal = Math.floor(255*(angle%stripWidth)/stripWidth);
         }

         fillColor  = "rgb("+rVal+","+gVal+","+bVal+")";

         var topLeft = new Point(currentX, currentY);
         var size = new Size(angle%(factor*rectWidth),
                             angle%(factor*rectHeight));

         var rectangle = new Rectangle(topLeft, size);
         var path = new Path.Oval(rectangle);
         path.strokeColor = fillColor;
         path.fillColor = fillColor;
      }
  </script>
</head>

<body>
  <canvas id="canvas" resize></canvas>
</body>
</html>
```

Listing 10.1 contains some JavaScript variables and a loop that computes the values on a `Conchoid`. Each value is used for specifying the location of an ellipse via the `Path.Oval()` method from Paper.js. Conditional logic determines the color of each ellipse, along with the other required attributes for rendering an ellipse.

SVG ANIMATION EFFECTS

In Chapter 7, you learned how to create animation effects in D3. In many cases, animation effects in D3 are simpler to create and update than the corresponding effects in SVG.

However, if you decide not to use the D3 toolkit, there are several other options for creating animation effects in SVG:

- SMIL (deprecated in Chrome)
- CSS3
- JavaScript+CSS
- GSAP MorphSVG
- Snap.svg or Morpheus

If you decide to use CSS3-based animation effects, you can find samples in an upcoming *CSS3 Pocket Primer* from Mercury Learning and Information. An example of GSAP-based animation in an Angular 2 application is provided later in this chapter, and code samples that combine ReactJS with SVG and GSAP-based animation are here: *https://github.com/ocampesato/react-svg-gsapi*.

The GSAP home page provides code samples and documentation for creating animation effects.

Several other good online resources with examples of creating animation effects with SVG are here:

https://davidwalsh.name/svg-animations-snap
https://jakearchibald.com/2013/animated-line-drawing-svg/
http://www.smashingmagazine.com/2014/11/styling-and-animating-svgs-with-css/

WHAT IS GREENSOCK?

GreenSock (also referred to as GSAP) animates anything that is accessible via JavaScript: CSS properties, canvas library objects, SVG, and generic objects. The GSAP home page is here: *http://greensock.com*.

GSAP handles many browser inconsistencies, with performance that is up to 20x faster than jQuery. GSAP consists of core tools, various plugins, eases, and special utilities such as Draggable and SplitText.

Later in the chapter you will see an example of combining GSAP with SVG in an Angular 2 application, so this section provides some relevant links.

The following "Why GSAP?" article contains additional details regarding GSAP: *https://greensock.com/why-gsap/*.

An article discussing SVG animation with GreenSock is here: *http://greensock.com/svg-tips*.

Information about the MorphSVGPlugin for creating animation effects is here: *http://greensock.com/morphsvg-update*.

A list of GSAP plugins is here: *http://greensock.com/plugins/?product_id=4921*.

An article from Google that mentions GSAP for JavaScript-based animations: *https://developers.google.com/web/fundamentals/design-and-ui/animations/css-vs-javascript*.

If you are new to CSS Properties (formally called CSS Custom Properties), the following links contain useful information:

https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_variables
http://philipwalton.com/articles/why-im-excited-about-native-css-variables/
https://drafts.csswg.org/css-variables/

A CONDENSED COMPARISON OF CSS, SVG, AND HTML5 CANVAS

CSS3 provides 2D/3D graphics/animation, GPU support, embeddable in SVG `<defs>` element, "easing functions" for animation, and animates HTML elements.

SVG provides 2D graphics/animation, some GPU support, support for arbitrary 2D shapes, custom `<pattern>` elements, grouping via the `<g>` element, and "easing functions" in D3 (but SVG cannot animate HTML).

Canvas provides 2D graphics/animation, GPU support, good for updating many small objects (games), works with video (use ThreeJS/WebGL for 3D animation). `Canvas` is often faster than SVG for showing polygons. Learn about tracing `Canvas` calls here: *http://www.html5rocks.com/en/tutorials/canvas/inspection/*.

CSS3+SVG is useful when a) you already have SVG-based data and b) you must support IE6 (can do with Raphael toolkit but not D3).

CSS3+D3 is good for modern browsers, and also easier for defining event handlers and animation (probably also easier to maintain/enhance).

In addition, keep in mind that CSS3+SVG might not have GPU support (and perhaps consider D3 with BackboneJS, Angular 2, React, and so forth).

An example of using CSS3 to create 3D animation effects with SVG is shown in the SVG document `3DSineWave4RG2TurbFilterAnim1.svg`.

NOTE *You can also reference an external CSS stylesheet in SVG documents.*

WHICH IS FASTER: CSS3/CANVAS/SVG?

This brief section contains some general rules regarding the relative performance of these technologies in HTML Web pages.

First, CSS3 is faster than `Canvas` for simple animation: *http://phrogz.net/tmp/image_move_speed.html*.

In addition, keep in mind the following general rules:

- CSS3 is faster than SVG for gradients
- D3 is better/faster than SVG for "follow the mouse"
- `Canvas` is better than SVG for many small objects

The preceding rules are general guidelines, so make sure that you test and compare the performance characteristics of a Web page on various mobile devices.

The next part in this chapter contains code samples that illustrate how to create Web pages that combine SVG with Angular 2 and ReactJS. Due to space constraints, there is limited coverage of the code samples.

SVG AND ANGULAR 2

The companion disc contains an Appendix with an introduction to Angular 2 (beta) and code samples for rendering SVG in Angular 2 Web applications. If you are unfamiliar with Angular 2, please read the relevant sections of that Appendix before you delve into the code sample in this section.

Listing 10.2 displays the contents of `main.ts`, which illustrates how to define a top-level component (using TypeScript) that "bootstraps" an Angular 2 application, and the SVG functionality is defined in the TypeScript file `mousemove.ts` (displayed in Listing 10.3).

LISTING 10.2: main.ts

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';
import {MouseMove} from './mousemove';

@Component({
   selector: 'my-app',
   directives: [MouseMove],
   template: '<div><mouse-move></mouse-move></div>'
})
class MyApp {}

bootstrap(MyApp);
```

Listing 10.2 contains standard `import` statements, along with an `import` statement for the `MouseMove` custom component that is defined in `mouse move.ts`. The `@Component` decorator contains the `selector` property

that specifies the standard custom element, followed by the `directives` property that specifies the `MouseMove` child component. The `template` property acts as a placeholder for the `<mouse-move>` element, which is where the SVG elements will be displayed.

Listing 10.3 displays the contents of `mousemove.ts`, which illustrates how to define an Angular 2 custom component to render SVG `<ellipse>` elements in order to create "follow the mouse" functionality.

LISTING 10.3: mousemove.ts

```
import {Component} from 'angular2/core';

@Component({
 selector: 'mouse-move',
 template: '<svg id="svg" width="600" height="400"
              (mousemove)="mouseMove($event)">
            </svg>
           '
})
export class MouseMove{
   radiusX = 25;
   radiusY = 50;

   mouseMove(event) {
     var svgns = "http://www.w3.org/2000/svg";
     var svg   = document.getElementById("svg");
     var colors = ["#ff0000", "#88ff00", "#3333ff"];

     var sum = Math.floor(event.clientX+event.clientY);

     var ellipse = document.createElementNS(svgns, "ellipse");
     ellipse.setAttribute("cx", event.clientX);
     ellipse.setAttribute("cy", event.clientY);
     ellipse.setAttribute("rx", this.radiusX);
     ellipse.setAttribute("ry", this.radiusY);
     ellipse.setAttribute("fill", colors[sum % colors.length]);
     svg.appendChild(ellipse);
   }
}
```

Listing 10.3 contains a standard `import` statement, followed by a `selector` property that specifies a `<mouse-move>` custom element as the parent element that will contain the SVG-based graphics. The template property specifies a top-level `<svg>` element and the `(mousemove)` event, which triggers the `mouseMove()` method whenever users move their mouse.

As you can see, the `mouseMove()` method initializes the `svgns` with the value of the SVG namespace, and the variable `svg` is initialized as a reference to the `<svg>` element that is specified in the template property. The sum of the x-coordinate and the y-coordinate of the current mouse position is used as an index into the `colors` array (which contains three colors) in order to set the color of the current SVG `<ellipse>` element. Each time that a new SVG `<ellipse>` element is instantiated, its mandatory attributes are set, and then the element is appended to the DOM.

D3 AND ANGULAR 2

Listing 10.4 displays the contents of `d3-angular2.ts`, which illustrates how to define a custom component in Angular 2 that contains D3 code for rendering SVG elements.

LISTING 10.4: d3-angular2.ts

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';

@Component({
   selector: 'my-app',
   template: `<mysvg></mysvg>`
})
class MyApp {
  constructor() {
     var width = 600, height = 400;

     // circle and ellipse attributes
     var cx = 50,   cy = 80,  radius1 = 40,
         ex = 250,  ey = 80,  radius2 = 80;

     // rectangle attributes
     var rectX = 20, rectY = 200;
     var rWidth = 100, rHeight = 50;

     // line segment attributes
     var x1=150,y1=150,x2=300,y2=250,lineWidth=4;

     var colors = ["red", "blue", "green"];

     // create an SVG element
     var svg = d3.select("mysvg")
                 .append("svg")
                 .attr("width",  width)
                 .attr("height", height);

     // append a circle
     svg.append("circle")
        .attr("cx", cx)
        .attr("cy", cy)
        .attr("r",  radius1)
        .attr("fill", colors[0]);

     // append an ellipse
     svg.append("ellipse")
        .attr("cx", ex)
        .attr("cy", ey)
        .attr("rx", radius2)
        .attr("ry", radius1)
        .attr("fill", colors[1]);

     // append a rectangle
     svg.append("rect")
        .attr("x", rectX)
```

```
            .attr("y", rectY)
            .attr("width",  rWidth)
            .attr("height", rHeight)
            .attr("fill", colors[2]);

         // append a line segment
         svg.append("line")
            .attr("x1", x1)
            .attr("y1", y1)
            .attr("x2", x2)
            .attr("y2", y2)
            .attr("stroke-width", lineWidth)
            .attr("stroke", colors[0]);
      }
   }

   bootstrap(MyApp);
```

Listing 10.4 contains standard `import` statements, followed by the `@Component` decorator with the usual `selector` property and a `template` property that specifies the custom `<mysvg>` element, which acts as the parent element for the SVG elements that are rendered.

The next portion of Listing 10.4 is the `MyApp` custom component whose constructor contains all the D3-related code for rendering the SVG elements. Refer to Chapter 6 for an explanation of the D3-related code.

GSAP AND ANGULAR 2

Listing 10.5 displays the contents of `main-gsap.ts`, which illustrates how to define a top-level custom directive in Angular 2 that contains GSAP code for adding animation to SVG elements that are programmatically rendered in `ArchTubeOvals.ts` (displayed in Listing 10.6).

LISTING 10.5: main-gsap.ts

```
import {bootstrap}  from 'angular2/platform/browser';
import {Component}  from 'angular2/core';
import {ArchOvals1} from './ArchTubeOvals1';

@Component({
   selector: 'my-app',
   directives: [ArchOvals1],
   template: '<graphics></graphics>'
})
class MyApp {
   ngAfterContentInit() {
       var deltaAngle = 1, maxAngle = 721;

       for(var angle=0; angle<maxAngle; angle+=deltaAngle) {
          var index = Math.floor(angle/deltaAngle);

          if(index % 3 == 0) {
            TweenLite.to("#elem"+angle, 4,
```

```
                        {scale:1.5, rotationX:45, rotationY:225,
                         x:10, y:0, z:-200});
            TweenLite.fromTo("#elem"+angle, 2, {x: '+=400px'},
                             {x: -150, y:-50, ease:Power1.easeInOut,
                              scaleX:0.5, scaleY:0.5});
          } else if(index % 3 == 1) {
            TweenLite.fromTo("#elem"+angle, 2, {x: '+=400px'},
                             {x: 100, y:50, ease:Power1.easeInOut,
                                      scaleX:0.4, scaleY:0.4});
          } else {
            TweenLite.fromTo("#elem"+angle, 4, {x: '+=400px'},
                             {x: 50, y:50, ease:Power1.easeInOut,
                              scaleX:0.3, scaleY:0.3});
          }
       }
   }
}
bootstrap(MyApp);
```

Listing 10.5 contains three `import` statements, one of which references the TypeScript file `ArchTubeOvals1.ts` that defines the `ArchOvals` custom component.

The next portion of Listing 10.5 is the `MyApp` custom component with the lifecycle method `ngAfterContentInit`, which contains all the GSAP-related code.

LISTING 10.6: ArchTubeOvals1.ts

```
import {Component} from 'angular2/core';

@Component({
 selector: 'graphics',
 template: '<svg id="svg" width="800" height="500">
            </svg>
           '
})
export class ArchOvals1 {
   constructor() {
      this.graphics();
   }

   graphics() {
     var svgns = "http://www.w3.org/2000/svg";
     var svg   = document.getElementById("svg");
     var colors = ["#ff0000", "#0000ff"];

     var basePointX  = 240,  basePointY  = 200;
     var currentX    = 0,    currentY    = 0;
     var offsetX     = 0,    offsetY     = 0;
     var majorX      = 30,   majorY      = 50;
     var Constant    = 0.25, angle       = 0;
     var deltaAngle  = 1,    maxAngle    = 721;
     var radius      = 1;
```

```
      for(angle=0; angle<maxAngle; angle+=deltaAngle) {
         radius   = Constant*angle;
         offsetX  = radius*Math.cos(angle*Math.PI/180);
         offsetY  = radius*Math.sin(angle*Math.PI/180);
         currentX = basePointX+offsetX;
         currentY = basePointY-offsetY;

         var ellipse = document.createElementNS(svgns, "ellipse");
         ellipse.setAttribute("id", "elem"+angle);
         ellipse.setAttribute("cx", currentX);
         ellipse.setAttribute("cy", currentY);
         ellipse.setAttribute("rx", majorX);
         ellipse.setAttribute("ry", majorY);
         ellipse.setAttribute("fill", colors[angle % colors.length]);
         svg.appendChild(ellipse);
      }
   }
}
```

Listing 10.6 contains an `import` statement, followed by the `@Component` decorator. This decorator contains a `selector` property whose value is the custom `<graphics>` element, followed by a `template` property that specifies the `<svg>` element, which is the root element for the SVG elements that are rendered.

The next portion of Listing 10.6 is the `ArchOvals` custom component whose constructor invokes the `graphics()` method that generates the SVG graphics. The `graphics()` element contains a loop in which SVG `<ellipse>` elements are constructed, and their position is based on different locations of an Archimedean spiral.

Notice how the SVG namespace is specified during the creation of each SVG `<ellipse>` element, as shown here:

```
var ellipse = document.createElementNS(svgns, "ellipse");
```

Listing 10.7 displays the contents of index.html, which contains the three GSAP-related `<script>` elements that are required in order to create the animation effects.

LISTING 10.7: gsap-index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Angular 2 SVG Ellipse</title>

    <script src="https://code.angularjs.org/2.0.0-beta.0/angular2-
                polyfills.js">
    </script>
    <script src="https://code.angularjs.org/tools/system.js">
    </script>
    <script src="https://code.angularjs.org/tools/typescript.js">
    </script>
```

```
    <script src="https://code.angularjs.org/2.0.0-beta.0/Rx.js">
    </script>
    <script src="https://code.angularjs.org/2.0.0-beta.0/angular2.
                dev.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/
                1.18.2/TweenMax.min.js">
    </script>

    <script src="http://cdnjs.cloudflare.com/ajax/libs/gsap/
                latest/TweenLite.min.js">
    </script>

    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/gsap/1.18.2/
           TimelineLite.min.js">
    </script>

    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.0.0-
           beta1/jquery.min.js">
    </script>

    <script>
      System.config({
        transpiler: 'typescript',
        typescriptOptions: { emitDecoratorMetadata: true },
        packages: {'app': {defaultExtension: 'ts'}}
      });
      System.import('app/main')
            .then(null, console.error.bind(console));
    </script>
  </head>
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

Listing 10.7 contains the standard contents of `index.html` for Angular 2 applications, along with three GSAP-specific `<script>` elements that are placed in the `<head>` element after the standard `<script>` elements.

REACTJS AND LOCAL CSS

The companion disc includes an Appendix that contains information about local CSS in ReactJS, and it's worth glancing through that Appendix before reading this section.

This section contains a code sample that illustrates how to display a "Hello World' message and also how to define CSS selectors that are local to a custom component.

Listing 10.8 displays the contents of `ReactLocalCSS.html`, which illustrates how to render a text string and also define local CSS in a ReactJS custom component.

LISTING 10.8: ReactLocalCSS.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Local CSS in ReactJS</title>

  <script src="https://fb.me/react-15.0.0-rc.2.js"> </script>
  <script src="https://fb.me/react-dom-15.0.0-rc.2.js"> </script>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/
               5.8.23/browser.min.js">
  </script>

  <style>
    #mydiv2 {
      background-color: #88f;
      width: 50%; height: 20px;
      margin: 10px; padding: 20px
    }
  </style>
</head>

<body>
  <div id="mydiv1"></div>
  <div id="mydiv2">Hello Dave2</div>

  <script type="text/babel">
    var divStyle = {
       background: "#fcc",
       width: "50%",
       margin: "10px",
       padding: "20px"
    };

    class Hello extends React.Component {
      render() {
        return <div style={divStyle}>Hello {this.props.name}</div>;
      }
    };

    ReactDOM.render(<Hello name="Dave1" />,
                    document.getElementById("mydiv1"));
  </script>
 </body>
</html>
```

Listing 10.8 contains `<script>` elements for ReactJS, followed by a `<style>` element with a selector that matches the `<div>` element whose `id` attribute has the value `mydiv2`. The `<body>` element contains two `<div>` elements, where the first `<div>` element is a placeholder for a ReactJS custom component.

The next portion of Listing 10.8 contains a `<script>` element that defines the variable `divStyle` that specifies various CSS-related properties.

The `divStyle` variable is specified in the code block that defines a ReactJS custom component called `Hello`. The final code snippet in Listing 10.8 renders the `Hello` component.

Launch the code in Listing 10.8 and you will see that the first `<div>` element has the style properties that are specified by the `divStyle` variable.

SVG AND REACTJS

ReactJS supports many SVG elements, including the standard SVG elements for rendering 2D shapes. As a simple example, Listing 10.9 displays the contents of `MySVGReact1.html`, which illustrates how to render SVG elements in an HTML Web page via a React custom component.

LISTING 10.9: MySVGReact1.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>A Simple SVG ReactJS Example</title>

  <script src="https://fb.me/react-15.0.0-rc.2.js"> </script>
  <script src="https://fb.me/react-dom-15.0.0-rc.2.js"> </script>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/
              5.8.23/browser.min.js">
  </script>
 </head>

<body>
  <svg id="svg"></svg>

  <script type="text/babel">
    class MySVG extends React.Component {
      render() {
          return (
            <g transform="translate(50, 20)">
               <rect width="200" height="100" fill="red"/>
            </g>
          );
      }
    };

    ReactDOM.render(<MySVG />,
                    document.getElementById('svg'));
  </script>
 </body>
</html>
```

Listing 10.9 contains two `<script>` elements for ReactJS, followed by a `<script>` element for BabelJS that transpiles (converts) the ReactJS code into ECMA5 that can be rendered in a modern browser.

The next portion of Listing 10.9 contains a `<body>` element that contains a `<div>` elements that acts as a placeholder for a ReactJS custom component. The third portion of Listing 10.9 contains a ReactJS custom component that renders a red rectangle inside an SVG `<svg>` element.

Additional code samples containing React and SVG are here: *https://github.com/ocampesato/reactjs-graphics*.

SVG, SPRITES, AND ICONS

The `svg-sprite` open source toolkit allows you to specify SVG in sprites, and its home page is here: *https://github.com/jkphl/svg-sprite*.

Embedding SVG: *http://www.schepers.cc/svg/blendups/embedding.html*.

Constraint-based layouts in the browser (good for SVG and D3): *http://marvl.infotech.monash.edu/webcola/*.

As you know by now, icons based on SVG can be scaled and will always look sharp, whereas bitmaps become pixelated when they are scaled up and they lose quality (pixels) when scaled down. Fortunately, you have a number of choices available, some of which are free while others are commercial products.

A good collection of free SVG icons (around 3,000) is here: *http://iconmonstr.com*.

A good article that describes how to combine the SVG `<defs>` element and `<symbol>` element to create SVG icons is here: *https://css-tricks.com/svg-sprites-use-better-icon-fonts*.

The following link describes how to use a Gulp-based system with SVG icons: una.im/svg-icons.

This article describes how to work with SVG icons and also provide fallback: *http://maketea.co.uk/2015/12/14/svg-icons-are-easy-but-the-fallbacks-arent.html*.

SVG Morpheus is a library for morphing between SVG icons, and its home page is here: *https://github.com/alexk111/SVG-Morpheus* and *http://dailyjs.com/2014/11/27/svg-morph/*.

Nucleo is a commercial SVG icon library whose home page is here: *https://nucleoapp.com/*.

OTHER SVG TOPICS

Responsive Web Design for Web pages has been popular for several years, and the SVG counterpart is Responsive SVG. Despite the flexibility of SVG, making SVG responsive in a Web page requires some additional markup. For example, sometimes you need to embed SVG code inside a `<div>` element and also add a `preserveAspectRatio` attribute and class to the `<svg>` root element (which is neither intuitive nor obvious). A good article that contains detailed examples for Responsive SVG is here: *http://demosthenes.info/blog/744/Make-SVG-Responsive*.

SVG and ARIA is another topic that attracts attention as SVG gains popularity in HTML Web pages.

SVG 1.1 accessibility support is limited in browsers and screen readers, and SVG2 will provide improvements. Currently there are two ARIA attributes that can be used to improve the accessibility of basic SVG 1.1 code.

The ARIA attribute `role="img"` ensures that the element is identified as a graphic. The ARIA attribute `aria-labelledby="title desc"` references the `id` values of the `title` and `desc` elements, thereby providing the accessible name and accessible description.

A sample SVG code block with the two preceding ARIA attributes is here:

```
<svg xmlns=http://www.w3.org/2000/svg role="img"
     aria-labelledby="title  desc">
   <title id="title">Circle</title>
   <desc id="desc">Large red circle with a black border</desc>

   <circle role="presentation" cy="60" r="55"
           stroke="black" stroke-width="2" fill="red" />
</svg>
```

More detailed information regarding SVG and ARIA is here: *https://specs.webplatform.org/SVG1.1-ARIA/webspecs/master/*.

USEFUL LINKS

Using inline SVG versus embed: *http://edutechwiki.unige.ch/en/Using_SVG_with_HTML5_tutorial*.

An online tool for converting SVG to various bitmap formats, including BMP, GIF, JPG, PDF, PNG, and TIFF file formats: *http://www.zamzar.com/convert/svg-to-png/*.

A toolkit that provides SVG support for older browsers: *https://code.google.com/p/svgweb/*.

`DrawSVGPlugin` enables you to progressively reveal (or hide) the stroke of an SVG `<path>`, `<line>`, `<polyline>`, `<polygon>`, `<rect>`, or `<ellipse>`, and you can even animate outward from the center of the stroke (or any position/segment). The `DrawSVGPlugin` is downloadable here: *http://greensock.com/drawSVG*.

ADDITIONAL CODE SAMPLES ON THE COMPANION DISC

Code samples that illustrate how to use Google Go to generate SVG are here: *https://github.com/ocampesato/google-go-graphics*.

The companion disc contains the HTML Web page `TroughPattern3S2.html` that is the "counterpart" of `TroughPattern3S2.svg` (which is part of the preceding open source project).

Figure 10.1 displays the result of rendering `TroughPattern3S2.svg` in a Chrome browser on a MacBook Pro.

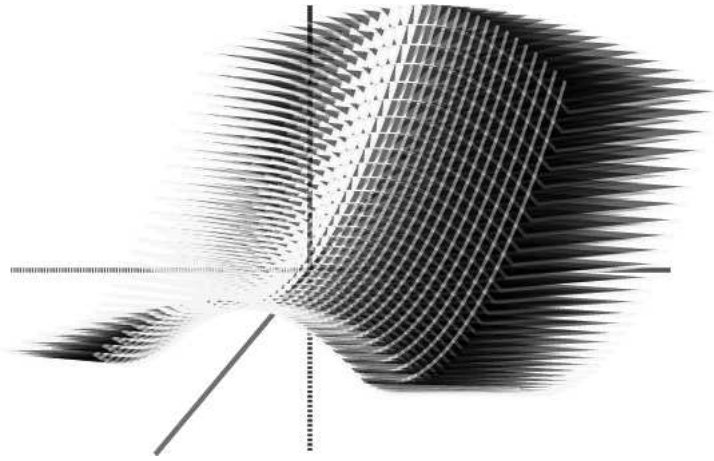

FIGURE 10.1: SVG 3D "trough" shape in a Chrome browser on a MacBook.

An open source project with numerous SVG code samples that illustrate how to create 3D effects is here: *https://github.com/ocampesato/svg-filters-graphics*.

Additional information (including details about matrix manipulation) and code samples about 3D effects in SVG is here: *http://msdn.microsoft.com/en-us/library/hh535759(v=vs.85).aspx*.

The Raphael toolkit for SVG supports IE6 through IE8, and some code samples, are here: *https://github.com/ocampesato/raphael-graphics*.

SUMMARY

This chapter started with some utilities for SVG, followed by CSS3 Media Queries in SVG. Then you learned how to work with SVG and Sprites, SVG Icons, as well as some JavaScript toolkits for SVG. In addition, you learned about GreenSock (GSAP) for animation effects, followed by a comparison of CSS, SVG, and HTML5 Canvas.

Then you saw how to combine SVG with Angular 2 and ReactJS. Finally, you learned about responsive SVG and also SVG and ARIA.

APPENDIX: GRAPHICS WITH REACTJS AND ANGULAR 2

This Appendix provides a short introduction to ReactJS and Angular 2, along with various code samples that illustrate some of the features of these two technologies.

Since the theme of this Pocket Primer is SVG, this Appendix is rudimentary in scope, but there are many resources available, including articles and Github repositories.

WHAT IS REACTJS?

ReactJS is an open source JavaScript-based toolkit from Facebook and its home page is here: *https://github.com/facebook/react*.

ReactJS has a component-based architecture and provides the view layer in Web applications. Hence, you need to complement ReactJS with other libraries in order to create complete Web applications.

One reason for the good performance of ReactJS is due to its virtual DOM that decouples ReactJS from the DOM. Whenever there are changes in a Web page, the virtual DOM is updated and only the latest changes are "pushed" to the DOM of the current Web page. This approach makes Web pages containing ReactJS highly performant.

ReactJS provides a one-way data binding model, which provides a simpler model and arguably makes it easier to debug ReactJS applications. As you will see later, Angular 2 also supports one-way data binding (two-way data binding is possible), whereas Angular 1.x provides two-way data binding.

The View Layer

React is the "View" layer of a single page application. There are no "controllers," no "models," and no utilities. The one-way data flow means that data flows into an element but never toward its parent.

As of version 0.14 of ReactJS, use the following syntax to create ReactJS custom components:

```
<script type="text/babel">
 // other code here
 class HelloWorld extends React.Component {
   // your custom code here
 }
</script>
```

Render the contents of a component via the `render()` method. For example, this code snippet displays the text string `Hello` in the element whose `id` attribute has the value `hello`:

```
<body>
  <div id="hello"></div>

  <script type="text/babel">
    // other code here
    ReactDOM.render(<div>Hello</div>,
                    document.getElementById("hello")
  </script>
</body>
```

The following code block (with some details omitted) displays three text strings:

```
<body>
  <div id="friends"></div>

  <script type="text/babel">
    var people = ["Jane","Joe","Jack","John"];

    ReactDOM.render(
      <div>{people.map(function(person) {
         return<div>My name is {person}</div>})
      }
    </div>,
    document.getElementById("friends"));
  </script>
</body>
```

NOTE *ReactJS displays a warning message whenever you append content directly to the <body> element.*

In case you're interested, a comparison of rendering a list in AngularJS and ReactJS is here: *http://antjanus.com/blog/web-development-tutorials/front-end-development/comprehensive-beginners-guide-to-reactjs/*.

React Boilerplate Toolkits

Although you have not seen any complete React code samples yet, the following link provides a self-contained startup kit with boilerplate for React: *https://github.com/gaearon/react-hot-boilerplate*.

An extensive set of links for other React boilerplate toolkits is here: *https://github.com/gaearon/react-hot-loader/blob/master/docs/README.md#starter-kits*.

Keep in mind that these toolkits have various dependencies, such as node, babel, and Webpack (which is the case for the first link), and sometimes Gulp and Browserify. The dependencies for these toolkits do change because of the preferences of the authors, and also because tools are superseded (sometimes quickly) by newer tools. For example, Webpack is a powerful and popular toolkit that appears to have attracted developers away from other toolkits such as Gulp (which is also very powerful). However, it's worth acquiring some knowledge of multiple toolkits in case you need to use them to set up projects that you download from resources such as Github.

You can avail yourselves of these toolkits at any time, or you can wait until you feel more comfortable with React-based Web applications.

WHAT IS JSX?

ReactJS supports JSX, which uses an XML-based syntax for defining components, as an alternative to using "pure" JavaScript. As you will see, JSX is useful when you need to create Web pages containing multiple components, especially when data needs to flow to components. However, keep in mind that JSX is not XML; for instance, JSX does not support namespaces.

The ReactJS code samples rely on the Babel transpiler in order to transpile ES6 code as well as JSX code to JavaScript. Moreover, Babel supports some ES7 features that you can use in ReactJS code. For production environments, compile JSX-based code into JavaScript from the command line using the babel utility. The necessary setup steps and code samples (and handling the combination of ES2015 and React) are described here: *http://facebook.github.io/react/docs/tooling-integration.html#jsx*.

A list of editors and tools that support JSX integration is here: *https://github.com/facebook/react/wiki/Complementary-Tools#jsx-integrations*.

One important point: the code in the JavaScript file JSXTransformer.js has been deprecated and replaced with other JavaScript files, as discussed in the next section.

JSXTransformer: Deprecated in Version 0.13

Prior to React 0.14, JSX code in a Web page requires the following JavaScript files:

```
<script src="https://fb.me/react-0.13.3.js"></script>
<script src="https://fb.me/JSXTransformer-0.13.3.js">
```

The new JavaScript files are shown below:

```
<!-- The core React library -->
<script src="https://fb.me/react-15.0.0-rc.2.js"></script>
```

```
<!-- The ReactDOM Library -->
<script src="https://fb.me/react-dom-15.0.0-rc.2.js"></script>

<!-- The Babel Library -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/
             5.8.23/browser.min.js">
</script>
```

For versions of JSX prior to 0.14, JSX requires this code snippet:

```
<script type="text/jsx">
  /** @jsx React.DOM */
```

The preceding code snippet must be replaced with following `<script>` element:

```
<script type="text/babel">
```

Instead, use the babel transpiler that is available online here: *https://babeljs.io/repl/*.

REACTJS AND "HELLO WORLD" CODE SAMPLES

This section shows you several Web pages with React in order to display a "Hello World" text string in a browser. The first Web page contains the simplest use of React, and the other two code samples use JSX, including one that involves deprecated code (so you know how to avoid that type of code).

Listing App.1 displays the contents of `HelloWorld1.html` using the new `<script>` elements and the modified syntax.

LISTING App.1: HelloWorld1.html

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Hello World in ReactJS</title>

  <script src="https://fb.me/react-15.0.0-rc.2.js"> </script>
  <script src="https://fb.me/react-dom-15.0.0-rc.2.js"> </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/
               5.8.23/browser.min.js">
  </script>
 </head>

 <body>
  <div id="hello"></div>

  <script type="text/babel">
    class Hello extends React.Component {
      constructor () {
         super();
      }

      render() {
        return <div>Hello World</div>
```

```
    }
  }

  // ES6 class with a click handler
  class Hello2 extends React.Component {
    constructor () {
       super();
    }

    render() {
      return <div onClick={this.handleClick}>Click Me</div>
    }

    handleClick() {
    //console.log(this);
    }
  }

  ReactDOM.render(<Hello/>,  document.getElementById('hello'));
 </script>
</body>
</html>
```

Notice the `<script>` elements in Listing App.1 as well as the portions shown in bold that differ from the older versions of React.

REACTJS AND LOCAL CSS

There are some well-known issues surrounding CSS, such as the global nature of CSS selectors. If you need a refresher about the pros and cons of CSS, including a discussion about inline styles, the following post provides some useful information (along with some interesting reader comments): *https://css-tricks.com/the-debate-around-do-we-even-need-css-anymore/*.

ReactJS supports inline styles ("the React way"), sometimes called "CSS in your JS."

The following presentation presents the problems of CSS and how Facebook has addressed them in ReactJS: *https://speakerdeck.com/vjeux/react-css-in-js*.

The following link provides a comparison of CSS in JS libraries for React: *https://github.com/FormidableLabs/radium/tree/master/docs/comparison*.

A second link with a comparison of CSS in JS libraries for React is here: *https://github.com/MicheleBertoli/css-in-js*.

CSS Modules provides another approach (unrelated to ReactJS) for local CSS via the local-style property: *https://github.com/css-modules*.

An article describing CSS Modules in greater depth is here: *http://glenmaddern.com/articles/css-modules* and *http://www.sitepoint.com/understanding-css-modules-methodology*.

SVG ELEMENTS AND REACTJS

As you've seen in previous code samples, a ReactJS-based Web application involves the creation of a class via `React.createClass()` and then defining the

mandatory function `render()`. According to the React documentation, React supports the following SVG elements: `circle`, `clipPath`, `defs`, `ellipse`, `g`, `line`, `linearGradient`, `mask`, `path`, `pattern`, `polygon`, `polyline`, `radialGradient`, `rect`, `stop`, `svg`, `text`, `tspan`, and `xlink`.

Listing App.2 displays the contents of `MySVG2.html`, which illustrates a variation of `MySVG1.html` and also how to render other SVG shapes, such as an SVG ellipse and an SVG line segment, in ReactJS.

LISTING App.2: MySVG2.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>A Simple SVG ReactJS Example</title>

  <script src="https://fb.me/react-15.0.0-rc.2.js"> </script>
  <script src="https://fb.me/react-dom-15.0.0-rc.2.js"> </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/
               5.8.23/browser.min.js">
  </script>
 </head>

 <body>
  <div id="graphics"></div>
  <script type="text/babel">
    class MySVG extends React.Component {
      render() {
          return (
            <svg width="100%" height="100%">
              <g transform="translate(50, 20)">
                 <rect width="160" height="80" fill="red"/>
                 <ellipse cx="80" cy="150"
                          rx="80" ry="40" fill="blue"/>
                 <line x1="0" y1="200" x2="150" y2="200"
                       strokeWidth="4" stroke="green"/>
              </g>
            </svg>
          );
      }
    };

    ReactDOM.render(<MySVG />, document.getElementById("graphics"));
  </script>
 </body>
</html
```

Listing App.2 differs from the code in Listing App.1 in several ways. First, the `<svg>` element has been moved from the `<div>` element into the `render()` method. Second, the SVG `<line>` element contains the strokewidth attribute that must be specified in "camel case" because of the hyphen. Third, the `document.getElementById()` method has been replaced with just `document.body`.

This concludes the portion of this Appendix that discusses ReactJS. The next portion of this Appendix provides a brief overview of Angular 2, followed by examples of rendering SVG in Angular 2 applications.

A HIGH-LEVEL VIEW OF ANGULAR 2

Angular 2 provides powerful new technologies that you need to learn in order to become proficient in writing Angular 2 applications. Interestingly, these new technologies lead to the following questions regarding current Angular 1.x applications:

- +when is it time to migrate Angular 1.x projects to ES6?
- +is it better to use ES6 or TypeScript?
- +how can companies perform a cost/benefit analysis?

The goal of this chapter (and the next chapter) is to provide insight into Angular 2 to help you make a more informed decision or recommendation.

Another point to keep in mind is that Angular 2 is in beta, and although future changes will probably be less frequent and (hopefully) less significant, you might need to change API calls in your code and/or change the import statements. You can always consult the official online website that contains the list of APIs for Angular 2: *https://angular.io/docs/ts/latest/api/*.

COMPONENTS IN ANGULAR 2

Angular 2 applications comprise a set of components for the UI elements, screens, and routes. An application always has a root component that contains all other components. Hence, every Angular 2 application also has a component tree.

Angular 2 applications contain an HTML5 Web page index.html (which you can rename) that references various JavaScript files and then invokes the 'System' module loader. The standard JavaScript files (whose version numbers will be different after this book goes to print) are shown here:

```
<script
 src="https://code.angularjs.org/2.0.0-beta.9/angular2-polyfills.js">
</script>
<script src="https://code.angularjs.org/tools/system.js"></script>
<script src="https://code.angularjs.org/tools/typescript.js">
</script>
<script src="https://code.angularjs.org/2.0.0-beta.9/Rx.js">
</script>
<script src="https://code.angularjs.org/2.0.0-beta.9/angular2.js">
</script>
<script src="https://code.angularjs.org/2.0.0-beta.9/http.min.js">
</script>
```

In addition, the module loader contains configuration-related information that specifies the name of a directory name and a TypeScript file in that directory that serves as the "entry point" for the application. An example is here:

```
<script>
  System.config({
    transpiler: 'typescript',
    typescriptOptions: { emitDecoratorMetadata: true },
    packages: {'app': {defaultExtension: 'ts'}}
  });
  System.import('app/main')
        .then(null, console.error.bind(console));
</script>
```

The preceding code block specifies the `app` subdirectory and the file `main.ts`, where the suffix `ts` is omitted because the `defaultExtension` property specifies the value `ts`.

An Angular 2 application contains one or more custom Components that are defined by you, typically in separate files with a "ts" extension.

Angular 2 components are often a combination of @Component decorator and a class that can optionally contain a constructor. A @Component decorator contains several properties. First you specify a "selector" property that indicates the HTML element (whether it's an existing element or a custom element) that serves as the "root" of your Angular 2 application. Next you specify a "template" property or a templateUrl property that works in essentially the same way as you saw in Angular 1.x. One difference: the template property requires "backticks" when its value spans multiple lines.

However, modern browsers only support ECMA5, which means that a transpilation step is required to convert TypeScript code and ECMA6 code in an AngularJS application into ECMA5.

Setting Up the Environment and Project Structure

Be prepared for a high-level yet lengthy section that describes the files in a generic Angular 2 application. The information is enough to give you a basic understanding of the purpose of the contents of the various files (such as the <script> elements in index.html). Consult online documentation for more detailed information.

As a starting point, you must install two command line binary executable files on your system: `npm` (version 3.3.12 or higher) to install the modules that are listed in `package.json` and `tsc` (version 1.8.5 or higher) to transpile ES6 code and TypeScript code into ECMA5. The version numbers in parentheses are the current versions for these binary files, which will probably be different by the time this book goes to print.

The tsc and typings Commands for Transpiling Files

The tsc command transpiles the contents of TypeScript files, which relies on TypeScript definition files that are installed after you invoke npm install

from the command line. The complete set of TypeScript definition files is here: *https://github.com/typings/typings*.

Install the typings executable by launching the following command:

```
sudo npm install typings -g
```

NOTE *The command TypeScript executable tsd is deprecated.*

The typings command installs TypeScript definitions. This command uses typings.json, which can resolve to GitHub, NPM, Bower, HTTP, and local files. Packages can use type definitions from various sources and different versions (and they never cause a conflict for users).

A sample invocation of the typings command is here:

```
typings install debug --save
```

Later in this chapter you will learn some basic concepts of TypeScript.

FOUR STANDARD FILES IN ANGULAR 2 APPLICATIONS

Basic Angular 2 applications involve four files: `index.html`, `tsconfig.json`, `package.json`, and `app/main.ts`. This section briefly describes these files, and you will see their contents after you have performed the required setup steps.

First of all, the HTML Web page index.html contains the JavaScript files that are necessary to transpile an Angular 2 application into ECMA5 and also to "bootstrap" the application. Next, the JSON file tsconfig.json contains property values for the TypeScript compiler tsc. Third, the JSON file package.json contains the list of modules that will be installed when you invoke npm from the command line.

Finally, the TypeScript file app/main.ts contains the code for the top-level component of an Angular 2 application.

NOTE *The scaffolding for Angular 2 application is described in the next section.*

In brief, AngularJS 2 Web applications involve `import` statements, Decorators (which are specified with an "@" symbol followed by a keyword) and a JavaScript class that is defined via the `class` keyword. In Angular 2 a decorator refers to metadata, which is a mechanism for providing more information about an ECMA6 class.

NOTE *A component with UI in Angular 2 is a class with a `@Component` decorator.*

Listing App.3 displays the content of `main.ts` that defines a simple Angular 2 component.

LISTING App.3: main.ts

```
import {Component} from 'angular2/core';
import {bootstrap} from 'angular2/platform/browser';
```

```
@Component({
   selector: 'my-app',
   template: '<h1>Hello World in Angular 2</h1>'
})
class MyApp {}

bootstrap(MyApp);
```

Listing App.3 starts with two common `import` statements that you will see in many Angular 2 code samples. The first `import` statement references `Component` from the core Angular2 module so that you can "decorate" the class `MyApp` in Listing App.3. The second `import` statement references bootstrap from another submodule of Angular 2, which enables you to "bootstrap" the Angular class `MyApp`.

The code block for the `@Component` annotation comprises a `selector` property and a `template` property. The value of the `selector` property is either a standard HTML element (such as `<div>`) or a custom element. In Listing App.3 the value of the selector property is `my-app`, which is a custom element that appears in the main HTML page.

NOTE *The value of the `selector` property corresponds to the name of a custom directive in AngularJS 1.x.*

The generated output will consist of the contents of the `template` property inside the element specified in the selector property, as shown here:

```
<my-app>
  <h1>Hello World in Angular 2<h1>
</my-app>
```

The next portion of Listing App.3 is the `class` keyword that defines the ECMA6 class called `MyApp` (which does not contain any code) and then the `bootstrap` keyword that "bootstraps" this Angular 2 application.

Now let's look at the three configuration files for this Angular2 project, as discussed in the next section.

THE `PACKAGE.JSON` CONFIGURATION FILE

This section shows you the contents of the `package.json` configuration file for application dependencies.

Listing App.4 displays the contents of `package.json`, which contains the required modules for this project.

LISTING App.4: package.json

```
{
  "name": "sample1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "angular2": "^2.0.0-beta.1",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.13",
    "reflect-metadata": "^0.1.2",
    "rxjs": "^5.0.0-beta.0",
    "zone.js": "^0.5.10"
  }
}
```

Listing App.4 contains a dependencies section that specifies six modules, along with their current version numbers. Place a copy of `package.json` in a convenient directory and then invoke the following command from the command line:

```
npm install
```

THE TSCONFIG.JSON CONFIGURATION FILE

This section shows you the contents of the `tsconfig.json` configuration file for the TypeScript compiler `tsc`.

You might be wondering how this text string was rendered without first transpiling the code in `main.ts` into ECMA5. The nice thing about this configuration is that the transpilation is performed automatically for you.

However, keep in mind that sometimes there are compilation errors that are not displayed in your browser's Inspector, and the only indication that something is amiss is the fact that the output does not appear (or is incorrect). In this case, manually transpile the TypeScript code by navigating to the root directory of this Angular 2 application and invoking the following command:

```
tsc
```

The preceding command will display any syntax errors in the TypeScript code in your project.

NOTE *Launch `tsc` from the command line in case there are syntax errors in your code.*

Listing App.5 displays the contents of `tsconfig.json`, which contains configuration-related properties that will enable you to invoke `tsc` from the command line without specifying any arguments.

LISTING App.5: tsconfig.json

```
{
  "version": "1.6.2",
  "compilerOptions": {
```

```
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "removeComments": true,
    "sourceMap": true,
    "outDir": "dist",
    "outFile":"bundle.js"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Listing App.5 contains various compiler-related properties and their values for the TypeScript compiler. Notice that the `outDir` property specifies the `dist` subdirectory as the location of generated files. The `sourceMap` is true, which means that a source map is generated during the transpilation process. The `outFile` property (introduced in TypeScript 1.8) configures the compiler to generate a single output file (which in this case is `bundle.js`).

Let's review where we are now: we started with `main.ts` in Listing App.3, followed by code listings for `package.json`, `index.html`, and `tsconfig.json`, and now we can launch this Angular application. That's a *lot* of setup and configuration just to display a "Hello World" message! Fortunately, you can use the code in this project as a sort of "template" for other Angular 2 applications, some of which are shown later in this chapter.

THE `INDEX.HTML` WEB PAGE

Listing App.6 displays the contents of `index.html`, which is the initial Web page for our Angular 2 application.

LISTING App.6: index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello World in Angular 2</title>

    <script
    src="https://code.angularjs.org/2.0.0-beta.0/angular2-polyfills.js">
    </script>
    <script src="https://code.angularjs.org/tools/system.js">
    </script>
    <script src="https://code.angularjs.org/tools/typescript.js">
    </script>
    <script src="https://code.angularjs.org/2.0.0-beta.0/Rx.js">
    </script>
    <script
      src="https://code.angularjs.org/2.0.0-beta.0/angular2.dev.js">
    </script>
```

```
    <script>
      System.config({
        transpiler: 'typescript',
        typescriptOptions: { emitDecoratorMetadata: true },
        packages: {'app': {defaultExtension: 'ts'}}
      });
      System.import('app/main')
            .then(null, console.error.bind(console));
    </script>
  </head>

  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

Listing App.6 contains five `<script>` elements that reference a CDN in order to access JavaScript code for this application. The five JavaScript files are briefly described as follows:

1) SystemJS

   Angular 1.x has a module system, whereas Angular 2 uses SystemJS that loads various types of modules, including ES6, CommonJS, and AMD. SystemJS uses typescript.js in order to transpile TypeScript into JavaScript, and necessary setup is declared in System.config, as shown here:

```
System.config({
  transpiler: 'typescript',   // here
  typescriptOptions: { emitDecoratorMetadata: true },
  packages: {'src': {defaultExtension: 'ts'}}
});
```

2) The file angular2-polyfills is sort of a "union" of zone.js and reflect-metadata. Angular 2 uses Zones to determine when to update the view.
3) The file reflect-metadata enables dependency injection through decorators.
4) RxJS (Reactive Extensions for JavaScript) is a library for Observables, which resemble the Promises, except they can be called multiple times. Moreover, Observables can be cancelled, whereas Promises cannot be cancelled.

The next portion of Listing App.6 contains a `<script>` element that specifies configuration-related information in order to transpile TypeScript code into ECMA5. The code snippet that references `main.ts` in the app subdirectory is shown here:

```
System.import('app/main')
      .then(null, console.error.bind(console));
```

The preceding code snippet returns a `Promise`, whose return is handled by the "then" statement. Sometimes you will see code samples that omit the "then" portion (which will not cause any errors).

The next portion of Listing App.6 contains a <body> element that in turn contains the custom <my-app> element: this is the same element that is specified in the selector property in Listing App.6.

Now perform the following steps: start a Web server (such as Python), open a browser tab, and navigate to the url localhost:8000. If you use another Web server (such as mongoose) the default port number might be different.

If everything was set up correctly, you will see the following output:

HELLO WORLD IN ANGULAR 2

The next section discusses Angular 2 template syntax, which you will use in your custom code in the template property.

CSS3 ANIMATION EFFECTS IN ANGULAR 2

The code sample in this section displays some data values and adds a CSS3 animation effect. This code sample uses the contents of employee.json in Listing App.7, which consists of JSON-based data.

LISTING App.7: main.ts

```
[
{"fname":"Jane","lname":"Jones","city":"San Francisco"},
{"fname":"John","lname":"Smith","city":"New York"},
{"fname":"Dave","lname":"Stone","city":"Seattle"},
{"fname":"Sara","lname":"Edson","city":"Chicago"}
]
```

Listing App.8 displays the contents of main.ts that illustrates how to change the color of list items whenever users hover over them with their mouse.

LISTING App.8: main.ts

```
import {bootstrap}      from 'angular2/platform/browser';
import {Component}      from 'angular2/core';
import {Inject}         from 'angular2/core';
import {Http}           from 'angular2/http';
import {HTTP_BINDINGS}  from 'angular2/http';
import 'rxjs/add/operator/map';

@Component({
   selector: 'my-app',
   template: '
     <button (click)="httpRequest()">Employee Info</button>
       <ul>
         <li *ngFor="var emp of employees">
          {{emp.fname}} {{emp.lname}} lives in {{emp.city}}
         </li>
       </ul>
```

```
    ',
    styles:  ['
      @keyframes hoveritem {
          0%   {background-color: red;}
          25%  {background-color: #880;}
          50%  {background-color: #ccf;}
          100% {background-color: #f0f;}
      }

      li:hover {
          width: 50%;
          animation-name: hoveritem;
          animation-duration: 4s;
      }
    ']
})
class MyApp {
  employees = [];

  constructor(@Inject(Http) public http:Http) {
  }

  httpRequest() {
    this.http.get('app/employees.json')
      .map(res => res.json())
      .subscribe(
        data => this.employees = data,
        err => console.log('error reading data: '+err),
        () => this.userInfo()
      );
  }

  userInfo() {
 //console.log("employees = "+JSON.stringify(this.employees));
  }
}

bootstrap(MyApp, [HTTP_BINDINGS]);
```

Listing App.8 contains the `styles` property, which contains a `@keyframes` definition for creating an animation effect involving color changes. This code block also contains an `li:hover` selector that references the previous `@keyframes` definition and also specifies a time duration of 4 seconds for the animation effect.

A BASIC SVG EXAMPLE IN ANGULAR 2

The code sample in this section shows you how to use the directives property in order to specify a custom component that contains SVG code for displaying an SVG element. This example serves as the foundation for the code in the next section, which involves dynamically creating and appending an SVG element to the DOM.

Listing App.9 displays the contents of `main.ts` that references an Angular 2 custom component in order to render an ellipse.

LISTING App.9: main.ts

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';
import {MyEllipse} from './my-ellipse';

@Component({
   selector: 'my-app',
   directives: [MyEllipse],
   template: '<div><my-svg></my-svg></div>'
})
class MyApp {}

bootstrap(MyApp);
```

Listing App.9 contains an `import` statement that references the `MyEllipse` custom component that is defined in my-ellipse.ts. Listing App.1 also contains the directives property that consists of an array of comma-separated components. In this example, `MyEllipse` is the lone custom component in the directives property.

Listing App.10 displays the contents of `my-ellipse.ts`, which contains the SVG code for an ellipse.

LISTING App.10: my-ellipse.ts

```
import {Component} from 'angular2/core';

@Component({
   selector: 'my-svg',
   template: '
     <svg width="500" height="300">
       <ellipse cx="100" cy="100"
                rx="50" ry="30"
                fill="red"/>
     </svg>
     '
})
export class MyEllipse{}
```

Listing App.10 is straightforward: the template property contains an SVG `<svg>` element with `width` and `height` attributes, and a nested SVG `<ellipse>` element with hard-coded values for the required attributes `cx`, `cy`, `rx`, `ry`, and `fill`.

D3 AND ANGULAR 2

The previous two sections showed you examples of Angular 2 applications with SVG. This section shows you how to combine D3 with Angular 2. In case you don't already know, D3 is an open source toolkit that provides a JavaScript-based layer of abstraction over SVG. Fortunately, the attributes

of every SVG element are the same attributes that you specify in D3 (so your work is cut in half).

Listing App.11 displays the contents of `main.ts`, which illustrates how to use D3 to render basic SVG graphics in an Angular 2 application.

LISTING App.11: main.ts

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';

@Component({
   selector: 'my-app',
   template: `<mysvg></mysvg>`
})
class MyApp {
  constructor() {
     var width = 600, height = 400;

     // circle and ellipse attributes
     var cx = 50,   cy = 80,  radius1 = 40,
         ex = 250,  ey = 80,  radius2 = 80;

     // rectangle attributes
     var rectX = 20, rectY = 200;
     var rWidth = 100, rHeight = 50;

     // line segment attributes
     var x1=150,y1=150,x2=300,y2=250,lineWidth=4;

     var colors = ["red", "blue", "green"];

     // create an SVG element
     var svg = d3.select("mysvg")
                 .append("svg")
                 .attr("width",  width)
                 .attr("height", height);

     // append a circle
     svg.append("circle")
        .attr("cx", cx)
        .attr("cy", cy)
        .attr("r",  radius1)
        .attr("fill", colors[0]);

     // append an ellipse
     svg.append("ellipse")
        .attr("cx", ex)
        .attr("cy", ey)
        .attr("rx", radius2)
        .attr("ry", radius1)
        .attr("fill", colors[1]);

     // append a rectangle
     svg.append("rect")
        .attr("x", rectX)
        .attr("y", rectY)
```

```
            .attr("width",  rWidth)
            .attr("height", rHeight)
            .attr("fill", colors[2]);

         // append a line segment
         svg.append("line")
            .attr("x1", x1)
            .attr("y1", y1)
            .attr("x2", x2)
            .attr("y2", y2)
            .attr("stroke-width", lineWidth)
            .attr("stroke", colors[0]);
   }
}

bootstrap(MyApp);
```

Listing App.11 contains a constructor that contains the D3-based code (but you are free to place this code in another method). The key point involves the `svg` variable that is essentially a reference to the `<mysvg>` element in the DOM, as shown here:

```
// create an SVG element
var svg = d3.select("mysvg")
            .append("svg")
            .attr("width",  width)
            .attr("height", height);
```

Notice how the various SVG elements are dynamically created and how their mandatory attributes (which depend on the SVG element in question) are assigned values via the `attr()` method, as shown here (and in the preceding code block as well):

```
// append a circle
svg.append("circle")
   .attr("cx", cx)
   .attr("cy", cy)
   .attr("r",  radius1)
   .attr("fill", colors[0]);
```

After you learn the mandatory attribute names for SVG elements, you can use the preceding syntax to create and append such elements to the DOM.

Many similar code samples involving SVG and Angular 2 are here: *https://github.com/ocampesato/angular2-svg-graphics*.

D3 ANIMATION AND ANGULAR 2

The previous two sections showed you examples of Angular 2 applications with SVG. This section shows you how to combine D3 with Angular 2.

The first change involves modifying the <my-app> element in index.html as follows:

```
<my-app><div id="mysvg">Loading...</div></my-app>
```

Now you're ready for the D3 code that is defined in `app/main.ts`. Listing App.12 displays the contents of `main.ts`, which illustrates how to create D3 animation effects in an Angular 2 application.

LISTING App.12: main.ts

```
import {bootstrap} from 'angular2/platform/browser';
import {Component} from 'angular2/core';

@Component({
   selector: 'my-app',
   template: '<mysvg></mysvg>'
})
class MyApp {
  constructor() {
     var width = 800, height = 500, duration=2000;
     var radius = 30, moveCount = 0, index = 0;
     var circleColors = ["red", "yellow", "green", "blue"];

     var svg = d3.select("body")
       .append("svg")
       .attr("width",  width)
       .attr("height", height);

     svg.on("mousemove", function() {
       index = (++moveCount) % circleColors.length;

       var circle = svg.append("circle")
          .attr("cx", (width-100)*Math.random())
          .attr("cy", (height-100)*Math.random())
          .attr("r",  radius)
          .attr("fill", circleColors[index])
          .transition()
          .duration(duration)
          .attr("transform", function() {
             return "scale(0.5, 0.5)";
           //return "rotate(-20)";
          })
     });
  }
}

bootstrap(MyApp);
```

Listing App.12 contains the usual boilerplate code, followed by a code block that performs animation effects, as shown here:

```
svg.on("mousemove", function() {
. . .
})
```

The code inside the preceding event handler is executed during each `mousemove` event, and an SVG `<ellipse>` element is dynamically generated. The new functionality involves the `transition()` method, the `duration()` method, and also setting the `transform` attribute. As you can

see, the `transform` attribute is set to a `scale()` value, which sets the width and height to `50%` of their initial value during an interval of `2` seconds (which equals `2000` milliseconds), thereby creating an animation effect.

GSAP GRAPHICS AND ANIMATION AND ANGULAR 2

This section shows you how to add SVG graphics and then add GSAP animation effects in an Angular 2 application.

Although you might think that this is "just another graphics sample," there is a key difference: this code sample relies on the Angular 2 lifecycle method (other lifecycle methods are discussed in Chapter 4) called `ngAfterContentInit()`, otherwise you will not see any animation effects. The reason is simple: the GSAP animation effects must be applied *after* the SVG elements have been dynamically generated, and the `ngAfterContentInit()` method solves this task.

Many similar code samples involving SVG, GSAP, and ReactJS are here: *https://github.com/ocampesato/react-svg-gsapi*.

SUMMARY

This Appendix introduced you to ReactJS, where you learned how to create Web pages containing JSX-based code. You saw how to render a "Hello World" message and also how to use ReactJS in order to render SVG in a Web page.

In the second part of this Appendix, you learned about the files that are part of Angular 2 Web applications. You also learned how to render graphics using SVG and D3, and saw a description of how to create animation effects with GSAP in Angular 2 applications.

INDEX